NEXTEST™ SYSTEMS CORPORATION

Magnum 1 Programmer's Manual

2/27/09

Nextest Systems Corporation
875 Embedded Way
San Jose, CA 95138
(408)-960-2400

Revision History

Date	Type	Details
8/19/01	Add	Initial Document
1/15/04	Add Update	Parallel Test Operation
Note: many incremental changes were made between the date above and below.		
4/4/26	Change	`set_results()`, `get_results()` were renamed to `results_set(), results_get()`.
	Add	`result_set(), result_get()`, `ecr_fast_image_write(), ecr_fast_image_read()`, `MULTI_DUT_CALL_BLOCK()`. Static Error Choice Functions, Branch-on-error, Test Patterns, Functional Tests, User Variables, Host / Site / Tool Communication, Resources, User Tools, User Dialogs, Excel Related Functions, Debug Hook and Pin Status Hook, MonitorApp, Environmental Variables, DUT Board TDR Functions, Miscellaneous.
6/7/04	Update	Data Buffer Memory (DBM).
	Add	New Magnum-only versions of `xtopo()` and `ytopo()` (2 ea). Data Buffer Memory Software (DBM) section.
6/24/04	Add	Controlling PE Levels from the Test Pattern.
6/30/04	Correct	Corrected chapter numbering problem.
7/23/04	Update Add	Added 4th PE driver mode (Dclk Mode) to Magnum PE Driver Modes, Double Clock Mode and `pe_driver_mode_set(), pe_driver_mode_get()`. Updated Double Data Rate (DDR) Mode section. Updated Pattern Attributes.
8/4/04	Add	Timing and Formatting Functions.
8/16/04	Add Update	Added APFP Tool-bar and LVM Branch/Label Limitations. Updated the PE Driver model in Pin Electronics (PE) Block Diagram and PE Driver Block Diagram.
8/17/04	Add	Added Magnum GUI images to the following sections: SummaryTool, TimingTool, FrontPanelTool, ECRTool, Voltage and Current Tool.

Date	Type	Details
8/23/04	Add	DUT Manager.
10/8/04	Add Update	Add DPS Operating Area and updated DPS Force Voltage Range. Added ±4A range to DPS Test/Measure Current Ranges for DPS in DPS Current Sharing mode.
10/19/04	Update	Updated Binning, Test Bin Functions, Test Bin Group Functions.
12/3/04	Update	First-pass effort in clarifying that Logic Vector Memory (LVM) and Scan Vector Memory (SVM) are the same physical memory. Initial draft version of Redundancy Analysis (RA) documentation.
12/8/04	Correct	Corrected operational descriptions and added boolean expressions to most of the MAR Multi-DUT Branch-condition Operands.
12/13/04	Add	Added `ecr_cache_enable()`.
1/17/05	Update Add	Redundancy Analysis (RA) documentation is complete. Added a separate Data Inversion Logic diagram, APG Data Inversion Enable Functions, APG Data Inversion Bank Select Functions, `bit2fen()` and APG Background Bank-A, Bank-B Inversion. Various changes to other data inversion topics to support the two-bank data inversion hardware and software. Corrected PMU Built-in Force Voltage Settling Time and DPS Built-in Settling Time (ranges vs. times were reversed). Substantial additions to WafermapTool, including WafermapTool Die-Bitmap Support and related subsections. Added `ui_BitmapFailColor, ui_BitmapPassColor`.
1/19/05	Deleted	Deleted `ecr_fail_signal_mux_set()` and `ecr_fail_signal_mux_get()`. The software will use the original `fail_signal_mux()` function.
2/3/05	Deleted	Deleted `cs_receive_only()`: it has never worked in Maverick-I/-II or Magnum and equivalent functionality is readily obtained from the test pattern.
2/7/05	Update Add	Updated Scan Test Patterns, added SCANDEF Compiler Directive and SVEC Pattern Instruction.
2/14/05	Update	Setting DUT Pin State (`setpin()`).

Date	Type	Details
3/4/05	Add	APG User RAM, APG User RAM Functions and USERRAM Instruction with USERRAM Operation Operands, USERRAM SourceA Operands, USERRAM SourceB Operands.
3/7/05	Add Update	Added Multi-DUT Shmoos. Updated `ui_ShmooOutputFile`, `ui_OutputFile` and `ui_OutputFormat`.
3/14/05	Add Update	Added `ecr_ddr_mode_set(), ecr_ddr_mode_get()`. Updated PTU Current Force, Test & Measure Ranges, PTU Voltage Clamp Range.
3/18/05	Add Update	Added `ra_spare_repaired_errors_get()`. Added the `INCR` operand to the list of logic pattern VCOUNT Function Operands. Rewrite Mixed Memory/Logic Patterns section, and describe the `mixedsync` Pattern Type Attributes.
3/24/05	Correct	Corrected argument types in several APG User RAM Functions and examples.
4/1/05	Add	Important Note: (and Note:) regarding a new scan test pattern vs. DUT pin usage rule. This affects future DUT board and test program compatibility.
4/18/05	Correct Change	Significant changes in v1.0.27: When Controlling Magnum 1 Levels from the Test Pattern, it is now legal to specify a pin scramble selection in a `LEVELSET` instruction. Previously this was not allowed. Magnum VAR counters now can be incremented (`INCR`) and decremented-by-2 (`DEC2`).

Date	Type	Details
4/21/05	Delete Change Add	These changes were required to eliminate out-of-memory situations when performing Redundancy Analysis (RA) on DUTs with large segments with many errors. The `ra_scan()` function is replaced by `ra_scan_area_callback()` which operates somewhat differently than `ra_scan()`. Similarly, the `RaScanAreaFunc` Call-back Function was replaced by RaScanRCFunc Call-back Function. Replaced `ra_scan_area_func_set()` with `ra_scan_area_callback_func_set()` and `ra_scan_area_func_get()` with `ra_scan_area_callback_func_get()`. One argument type changed for the `ra_config_set()` and `ra_config_get()` functions.
4/25/05	Update Add	Reorganized information relating to APG branch logic, to correctly reflect that some of the hardware is shared by the memory pattern controller (MAR Engine) and logic pattern controller (VAR Engine). Moved the block diagrams from the Test Pattern MAR instruction section to Branch-on-error Logic in the hardware section. Updated MAR Error-choice Operands and Static Error Choice Functions, Branch-on-error. Added VAR Multi-DUT Branch-condition Operands.
4/29/05	Add	Added four tables describing the operation of each APG branch operand vs. each combination of static error choice (see Static Error Choice Functions, Branch-on-error) and MAR Error-choice Operands. These tables are sequential, starting with Branch Operand Operation: t_errmode1. Simplified the descriptions in MAR Multi-DUT Branch-condition Operands and VAR Multi-DUT Branch-condition Operands and added references to MAR Error-choice Operands.
4/29/05	Add	Added Double Data Rate (1/29/08 replaced by MUX, Super-MUX and DDR).

Date	Type	Details
5/6/05	Add Update Correct	Add Shmoo Call-back Function, Shmoo Call-back GUI Controls, Shmoo Symbols Dialog to the Multi-DUT Shmoos section. Two new call-backs and call-back registration functions were also added: `shmoo_duts_int_callback`/ `shmoo_duts_int_callback_set()`, `shmoo_duts_string_callback`/ `shmoo_duts_string_callback_set()`. Updated all GUI/ dialog images in the ShmooTool / SearchTool section to use latest design. Corrected the dates of the previous 5 entries in this Revision History. All had 5/xx/05 dates and should have been 4/ xx/05.
5/20/05	Add	Added documentation for the RTC timing format.
5/30/05	Correct	Corrected range vs. value data in PMU Built-in Force Voltage Settling Time and DPS Built-in Settling Time. Order was reversed.
6/1/05	Add	Added Double Data Rate (DDR) section (1/29/08 replaced by MUX, Super-MUX and DDR). Added `active_dut_get()`, `all_hv()`, `no_hv()`.
6/30/05	Update	Clarified example in RA Pseudo-Code Example. Updated DPS Compensation Capacitors description. Updated legal boilerplate page and changed title/chapter page graphics.
7/8/05	Add	Added a note to `ecr_config_set()` and `ecr_all_clear()` that executing `ecr_config_set()` also executes `ecr_all_clear()`. This was not previously documented.
7/18/05	Add	Added Note: and Note: to PMU: Testing HV Pins: it is a fatal error to attempt a PMU-on-HV test with a negative force voltage (`vpar_force()`) or negative test limits (`vpar_high()`, `vpar_low()`).
8/3/05	Add	Shared Tester Pins and `ecr_dut_number_set()`, `ecr_dut_number_get()`. In Per Pin Error Status, added `DutPin` versions of `test_pin()` and `test_pin()`. These were not previously documented.
8/15/05	Correct	Corrected the Redundancy Analysis (RA) call-back example code in Example 4: and Example 5:.
8/23/05	Add	Magnum 1/2/2x Simulation Setup (replacing the Maverick instructions which don't work using Magnum).

Date	Type	Details
5/25/05	Correct	Corrections were made to details in: APG Interrupt Timer, MAR Interrupt Operands (for INTEN and INTENADR), and the operational description of the example in APG Instruction Execution. Additional details were added to the description of the RELOAD# operand in the COUNT Counter Operands section.
9/27/05	Add Update Fixed	Updated Environmental Variables by adding Nextest Environment Variables table. Updated various other topics which reference Environmental Variables. Fixed and updated Pattern Load PATH (the information has been corrupted for some time).
10/23/05	Correct	Corrected DBM size vs. cycle period information in DBM Memory Size Options, and related information in DBM Usage Rules.
1/11/06	Update Change	Changed the title of the section titled, “Registering User Variables with UI” to Intercepting User Variables. Substantial updates to Intercepting User Variables content.
1/20/06	Add	Added `testerpin_value()`, `testerpin_offset()` and Pin Iteration.
1/30/05	Change	Corrected the description of the test pattern MAR CLEARERR instruction/operand, to match the hardware implementation.
2/6/06	Add	Retrieving the Nextest Software Version (`current_release()`).

Date	Type	Details
2/8/06	Add Update Correct	Significant changes in software release h1.1.23: Completed Error Pipeline Requirements for Magnum. Added DUT Board TDR Functions for Magnum. Shmoo Functions: `shmoo_title_get()`, `shmoo_type_get()`, `shmoo_direction_get()`, `shmoo_axis_params_get()`, `shmoo_param_get()`, `shmoo_param_pointval_get()`, `shmoo_duts_subtitle_set()`, `shmoo_duts_subtitle_get()`, `search_results_get()`. Added information about *StartUI.exe* in Starting UI from Windows. Added enhancements to WafermapTool: Die Field Display and Subtitle (`wmap_set()` with `wmap_subtitle`). Added `ui_HideTool`, `ui_Show` and updated `ui_ShowTool`. Added a related comment to the ToolLauncher introduction. Added ui_ShowTool / ui_HideTool Support to the ToolLauncher section. Rewrite Logic Error Catch (LEC) for Magnum, including new APIs (Maverick users should review the entire section). Added LVMTool and LEC Tool documentation. Added RBoot Client File, used in multi-site situations to set environment variables and/or execute commands on the Site computers at boot-time. Update and additions to Magnum 1/2/2x Simulation Setup. Added new overload of `label_offset()` to support logic patterns. Updated Sequential Test Block, Conflict List and Conflict List Macros. The need for the `dbm_pattern_use()` function was eliminated, see Note:. Added the hardware JAM RAM option (see JAM Logic) and related functions (see APG JAM Logic Configuration Functions). Also added new JAM RAM-related operands the DATGEN Dataout Operands table. ...continued...

Date	Type	Details
2/8/06	Add Update Correct	... continues... Significant changes in software release h1.1.23: The documentation section previously titled *ActiveDutIterator* was renamed Active DUTs Set Iterators, added SoftwareOnlyActiveDutIterator, and documented changes and enhancements to `ActiveDutIterator`. Documented ECRTool, which includes enhancements noted in ECRTool Next Error Search and the tool dialog controls. Added DUT Board Status Check and `ui_DutBoardStatusCheckDisable`. New Measurement Average Count Functions to support static DC test measurement averaging. Includes new overloads of `test_supply()` and `ptu_partest()` and enhancements to `partest()` and `hv_test_supply()`. Added `ui_SiteDone` and updated related information in `ui_TestDone`. Corrected PTU table of RL Values to include the nominal 50Ω series FET switch. This includes the RL Values table, the Useful RL Combinations table and the RL Values table.
2/8/06	Change Add	Changed the order that UI tools are arranged in the document to alphabetical, by name. Added UI Tool Persistence.
2/10/06	Update	Updated Active DUTs Set (ADS) text to correctly reflect how Parking Blocks operate, including examples and minor related changes/clarifications to Parking Blocks and Overview. ~~Added Note:, Note:, Note:, Note: and Note: regarding test pattern RESET to STROBE rquirements.~~ Note that the previous restriction was subsequently removed in software release h2.2.7/h1.2.7.
2/23/06	Correct	Corrected the DPS Built-in Settling Time table and PMU Built-in Force Voltage Settling Time: the range values were reversed vs. the settling time values.
2/27/06	Add	Added DC Level Response Time information to Controlling Magnum 1 Levels from the Test Pattern.
3/10/06	Deleted	Deleted the text in Program Migration to MsDev 6.0. Added a reference to the appnote which retains this information,
3/21/06	Change	Changed all references to `t_na` to `a_na` or `b_na` to correct potential problems in future software releases. Magnum test programs should not use `t_na`.

Date	Type	Details
3/31/06	Add Correct	Added ui_ShmooDone and related information in search_results_get(). Added dmain(), dbase() functions. Added a Note: indicating that the drive_only() funcion prevents pins from reacting to the VIHH signal from the test pattern; i.e. drive-only pins cannot be set to the VIHH drive level. Corrected details regarding APG Data Register Fill-bit operation during SHLDR/SHRDR instructions (the fill-bit does not come directly from the UDATA bit-36, as previously documented, but rather from a separate register which can be loaded from UDATA as desired). Added documentation describing that the APG Data Generator DMAIN and DBASE registers have separate shift/ fill bits and how these bits are modified by various DATGEN instruction options. See Data Register Fill-bit.
4/10/06	Add	Changes in software release h2.0.xx. Note that these changes are all rather minor except for the increase in the maximum number of Sites-per-Controller from 4 to 10. The Release Notes for h2.0.xx discuss this in some detail but that information is mostly not included in this document.
4/17/06	Correct	Corrected numerous table formatting errors (effected display only)
4/24/06	Add	Added previously undocumented rules regarding memory pattern branch operation vs. MAR error choice selection. See Note: and Note:
5/8/06	Correct	Corrected DPS Current Measurement Ranges to remove range7 and correctly describe range6 operation.
5/9/06	Correct	Corrected PMU Measure Voltage Ranges (HW) and PMU Measure Voltage Ranges (SW) to include the 2nd voltage range usable on PE pins.
5/12/06	Correct	Corrected the variable type of the following UI User Variables: ui_TestStarted, ui_SiteLoaded, ui_SiteUnloaded, ui_ProgLoaded,
7/17/06	Correct	Fixed missing links in Test Pattern Programming section.
7/18/06	Add	Added ui_DbmDialogDecMode, ui_ECRDialogDecMode. Added new overloads of vecdata() which support DDR use.
7/19/06	Add	Significant changes in software release h2.0.yy. Review the Release Notes for details.

Date	Type	Details
7/24/06	Add	Added Note: (and several like it in other locations) to note that, "*In any Multi-DUT Test Program testing more than 2 DUTs, a VECDEF directive is required (not optional) in any test patterns containing logic instructions.*"
8/9/06	Add Move Rename	Moved the section previously titled, *PE Error Logic* to Branch-on-error Logic in the Algorithmic Pattern Generator (APG) hardware section. Added MAR BOE Type Operands and added related references several key places in the document (first available in software release h2.0.9).
8/8/06	Change	Changed the default value of ui_BitmapRowsChunk from 512 to 1024, to improve the display update performance of BitmapTool. This change is first available in software releases h1.0.25 and h2.0.10.
9/22/06	Update	Added information regarding how `ptu_partest()` limits `partime()` to 5mS maximum. See `Parametric Settling Time`. Clarified purpose of min/max limit arguments for `db_tdr()` and updated Example.
10/2/06	Update	Updated Over-programming Controls and Parallel Test and consolidated information on this topic. This affected Control of Branch on Error Flag, Over-programming Control Stimulus Selection and includes minor changes to DPS Output Mode, VPulse Function, VIHH Voltage.
10/10/06	Add	Added DUT-specific Pin Lists.
10/16/06	Correct	Corrected the requirements for updating the *Units Passed* counter in FrontPanelTool (page 2013).
10/20/06	Add	Added 2Gig ECR info to Magnum ECR Memory Size Options. The 2Gig ECR is first supported in software releases h2.0.12/ h1.0.27.
10/31/06	Change	Support for ECR absolute write mode (`t_abs_write`) was un-documented. This mode was never implemented due to hardware limitations.

Date	Type	Details
11/8/06	Correct	Corrected the PTU current range selected when the background voltage is used during PMU tests. See Background Voltage Functions. Added several notes indicating that proper operation of the test pattern branch-on-abort (or -no-abort) requires that all DUTs be in the Active DUTs Set (ADS). For example, see Note:, Note: and Note:.
11/17/06	Add Correct	Added Note:, Note:, Note:, Note: and Note: to warn users that test results will be invalid when executing pattern-triggered DC tests in which the test pattern fails to generate a `vcomp` trigger. This applies to `hv_ac_test_supply()`, `ac_partest()`, `ac_test_supply()` and `ptu_ac_partest()` (the latter when measurements are enabled). Corrected the description of how the Active DUTs Set (ADS) affects the operation of `test_pin()` and `test_pin_first_error()`.
11/17/06	Add	Added the following previously undocumented versions of DBM functions. These all have an `HSBBoard` argument, usable when Sites-per-Controller > 1: `dbm_config_set()`, `dbm_config_get()`, `dbm_segment_set()`, `dbm_segment_get()`, `dbm_fill()`, `dbm_write()`, `dbm_read()`, `dbm_file_image_write()`, `dbm_file_image_read()`, `dbm_masks_set()`, `dbm_masks_get()`, `dbm_num_segments_get()`.

Date	Type	Details
12/4/06	Correct Add	Corrected several errors related to RL Values which affects the following: - The 50-ohm FET switch resistance, previously described as being a per-RL-resistor adjustment, actually represents the resistance of the FET switch which connects a PTU to the DUT. The text was corrected and the various RL values were adjusted. - PE Driver: added columns and corrected errors in the RL Values table. - PE Load Reference Voltage: VZ: added 2 columns and corrected errors in the RL Values table. Moved the the adjoining graph to a separate location (below the table) and corrected the values. - Deleted the table titled *Useful RL Combinations*. - Corrected the operational description for `rl_get()`: it returns the last programmed bit-wise mask used to select RL values, not the resulting RL resistance.
12/29/06	Correct	Corrected the variable type of `ui_ProgLoaded` from `CSTRING_VARIABLE` to `VOID_VARIABLE`.
1/11/07	Add	Added Note: and references to it, to clarify which `TesterFunc` enumerated type scan-related values are valid for Maverick vs. Magnum.
1/18/07	Add	Added rule-9. to the Magnum Timing Rules, to correct and clarify the valid range of edge times.

Date	Type	Details
1/24/07	Add Update	Significant changes in software release h2.2.7/h1.2.7. Pin Frequency Measurement (PFM), including Pin Frequency Measurement Operation, Pin Frequency Measure Software (`pin_frequency_meas()`, `pin_frequency_meas_get()`). Added Save/Load Sequence/Binning Table Modifications. New icons and icon combinations were added to UI's UI Sequence and Binning sub-window and the STOP action was added to Modifying the Sequence and Binning Table options. Added information related to Address Cross-over Bits, including Address Cross-over Bit Functions and changes to APG Y Address Generator Block Diagram (and text near it) and Address TOPO RAM Block Diagram. Added Changing Dialog Button Text. Added `ui_BreakPointRemoveAll`.Added Waveform Functions and WaveformTool (MSWT). Added DUT-specific Pin Lists. Added Run Buttons in Breakpoint Monitor. Add support for pattern triggered DC measurements via DPS Dynamic Current Test Functions, PMU Dynamic Test Functions, HV Dynamic Test Functions. Complete re-write of Dynamic DC Tests. Added the `More(BOOL wrap)` member function to SoftwareOnlyActiveDutIterator, to improve iteration performance when SoftwareOnlyActiveDutIterator is used within an outer loop. A new DBM configuration option was added (DBM Sequential Mode) which required reorganizing information in Data Buffer Memory (DBM) and DBM Architecture hardware sections and additions to `dbm_config_set()`. In addition, the Data Buffer Memory Software (DBM) section was updated to add an Overview and new DBM Interleave Mode, DBM Sequential Mode and DBM Usage Rules sections. Complete re-write of DBMTool documentation, including new features, including DBM segment selection.
1/29/07	Correct	Corrected rule-2. in DBM Usage Rules to properly state the rule which applies when using the DBM in `t_dbm_slow_speed` mode.

Date	Type	Details
2/14/07	Change Update	Additional analysis and testing indicates that, in a test pattern, it is legal to strobe in a cycle after MAR/VAR RESET. The restriction which was added on 2/10/06 was removed from this manual (5 places). This restriction also appeared in the h2.0.11/h1.1.26 release notes. Notice of this change was included in the h2.2.7/h1.2.7 release notes. Revised some details in DBM & Multiple Sites-per-controller and `dbm_file_image_read()` and `dbm_file_image_write()`regarding the the use of the `HSBBoard` argument when Sites-per-Controller > 1.
2/27/07	Updated Correct	Updated the list of supported PatStopCond options and corrected the description of several in Pattern Execution Stop Condition Options, Pattern Execution Stop Condition Options, Pattern Execution Stop Condition Options, Pattern Execution Stop Condition Options, etc. Corrected Example 3: in SCRAMBLE_32DUT Work-around.
3/2/07	Change	Changes to DBMTool segment selection controls to delete segment tabs (which don't work well when there are many segments).
3/20/07	Update	Updated Pin Frequency Measurement (PFM) to add a 3rd range (adds 32 counts to the original 80/5000 count options) which lowers the minimum measurable frequency.
3/26/07	Correct	Corrected errors in example code for `gpio_direction_set()` and `spi_cmd()`.
3/28/07	Change	Changed the type of the `ui_TestStarted` (see UI User Variables) from `VOID_VARIABLE` to `UINT64_VARIABLE`. This is backwards compatible.
4/9/07	Change	Changed the minimum window strobe width from 6nS to 5nS, to match the Magnum product specification. See Timing Specifications.

Date	Type	Details
7/6/07	Delete	The use of "**Functions, MACROs & Keywords**" was removed throughout the document. These only ever existed due to a limitation in the documentation authoring program, to allow links to individual words, tokens, etc. An alternative method has been developed to allow these link targets to be inserted at the desired locations.
8/8/07	Delete	The documentation of the APG User RAM, the USERRAM Instruction and related sections were modified to remove all references to features which are not currently supported by the Magnum hardware and software.
9/14/07	Add	Added a new rule affecting Data Buffer Memory (DBM) use in test patterns which are to be paused (MAR PAUSE) and restarted, using `restart()` or `restart_and_wait()`. See page 1444.
9/20/07	Add Change Update Delete	Significant changes in software release h2.2.11/h1.2.11. Automated Pattern File Processing (APFP) was completely re-implemented. DO read about this. User Variables Tool was completely redesigned. Changed the value returned by `all_dps()`, `no_dps()`, `all_hv()` and `no_hv()` to FALSE (was TRUE) when the passed pin list is NULL or contains no pins. Added `all_pe()` and `no_pe()`. Added Test Flow Synchronization and related information. Added `rl_ohms_get()`. In MixedSync Pattern Rules, added a new rule (rule-8.) and updated rule-7. UseRel was modified to prompt the user whether to execute *StartServer*. The use of `PTU_IRANGE_2UA` through `PTU_IRANGE_32MA` is deprecated. These were defined as equivalents to the less self-documenting `enum Range` tokens (`range1`, `range2`, etc.). But, because future system types will have similar but different ranges, the use of these equivalents will create problems when, for example, `PTU_IRANGE_128UA` equates to `range4` on one system type and `range5` on another system type. This difference in range selection affects the actual range of the voltage/current parameter, with associated effects on resolution and accuracy. All references to these equivalent tokens were removed from this document.
11/5/07	Correct	Corrected information about which `UDATA` bits are used for X vs. Y addresses in the table titled UDATA Bit Applications.
3/5/08	Add	Added `SimulationMode()`.

Date	Type	Details
3/18/08	Add Delete	Added Pattern Sets documentation. Deleted references to the MAR READUDATAV and READUDATAZ operands; they were not implemented.
3/26/08	Add Updated	Complete re-write of MUX, Super-MUX and DDR documentation (Super-MUX applies to Magnum 2 only). This includes adding a separate timing diagram for Magnum 1/2 in DDR Strobe Timing & Formats section. Updated Magnum 1/2/2x Simulation Setup to support Magnum 2 (and 2x). This clarified some Magnum 1 details too.
4/8/08	Correct Updated Add	Corrected the description of how Pattern Sets operate when a non-loaded test pattern is executed. All functions which execute patterns and return PASS/FAIL return FAIL if a non-loaded test pattern is executed (a warning is also issued). Previous documentation incorrectly indicated that the test would return PASS. Corrected the maximum number of supported VIHH Maps from 128 to 64 (the Magnum hardware only ever supported 64 VIHH maps, the software allowed up to 128). Updated the description of `pmu_connect_at()` operation to include information about how PMU connections may be made in force-voltage mode even when subsequent operation uses force-current mode. Added information to Error Pipeline Requirements, starting with Note:. This applies to logic patterns in which multiple branch decisions are made based on errors which have already been pipelined to the branch logic. Updated DBM DRAM Interleaving, DBM Sequential Mode, DBM Usage Rules, `dbm_config_set()` and `dbm_config_get()`.
4/25/08	Add Updated	Significant changes in software release h2.3.xx/h1.3.xx. `ecr_rams_update()` operation was enhanced to stop tracking ECR modifications after 16K changes using `ecr_error_delete()`, `ecr_error_set()` and `ecr_area_clear()`. See `ecr_rams_update()`. The function `hold_pattern_state()` is now silently ignored. The related operation is now controlled by hardware, see Note:. Added WaveTool (this tool is still evolving, as noted in the documentation).

Date	Type	Details
4/30/08	Add Moved	Added Setting a Static Pin-state using Level Sets and related text in Pin DC Static State Functions and Magnum 1/2/2x Test Pattern Set/Tweak Parameter List. Moved Logical vs. Physical, vs. Electrical Addresses to the Manipulating Tester Hardware section from the Redundancy section.,
5/5/08	Add	Added a new rule to APG Timer Functions, page 1322, “If the `timer()` function is used in Pattern Initial Conditions it must be the last function specified.”.
6/2/08	Correct	Corrected the maximum negative DPS voltage to -15V, to match the product specification. See DPS Force Voltage Range. Corrected the voltage resolution of the Parametric Background Voltage, to match the product specification. See Background Voltage Range.
7/22/08	Update	Complete rewrite (updated) of the section [newly] titled DUT-pin to Tester-pin Connection Requirements. This section was previously titled, “*DUT-to-Pin Connection Requirements*“. The new version is much simpler and better explains what is involved.
7/29/08	Add	Added Note: which explains the following APFP rule: Note: APFP expects that the file name of test program executable file (i.e. .../Debug/myProg.exe) matches the file name of the Visual Studio project file (i.e. .../myProg.dsp). If these file names do not match, APFP execution will enter an end-less loop, continuously compiling test patterns in stub mode.

Date	Type	Details
8/4/08	Add Update Correct Delete	Added the following note to the documentation for `pe_driver_mode_set(), pe_driver_mode_get()`, "the pe_driver_mode_set() function will not affect pins which are not currently connected. This is an issue when pe_driver_mode_set() is used in the Site Begin Block. The system software automatically connects all PE pins in the Site Begin Block, but this does not occur until after all user-code has been executed. For this reason, it is recommended that pe_driver_mode_set() be executed in a Before-testing Block or Test Block.". The COUNT Counter Operands section has been rewritten to clarify and correct the operational description of related pattern operands, most significantly the RELOAD# operand. These changes also call attention to a difference in operation between Maverick-I/-II and Magnum 1 vs. Magnum 2/2x. Un-documented (deleted) the VAR Interrupt Operands for Magnum 1. The related functionality will not be implemented. All APG Interrupt Timer use requires the use of related MAR Instructions which requires either a Memory Test Pattern or Mixed Memory/Logic Pattern. Clarified logic pattern rules in Pattern Sets.
8/19/08	Add	Added `ecr_configured_get()`, `lec_configured_get()`. Added Note: to the Pattern Sets which states, "when a pattern set includes a logic test pattern from a given .pat file then all patterns in that .pat file must be included in the pattern set. This is required even when some patterns in the file are not subsequently used or executed. Proper pattern operation may not occur if this rule is violated.". Added Magnum System Type Get Function (`is_magnum()`).
8/27/08	Update	Updated the maximum frequency which can be measured using Pin Frequency Measurement (PFM) from 83.3MHz to 113Mhz.
9/8/08	Add	Added a new rule to Error Pipeline Requirements, page 1272, which applies in any test pattern which uses LVM (i.e. Logic Test Patterns and Mixed Memory/Logic Patterns). The rule states, "*When using Logic Vector Memory (LVM) (i.e. in Logic Test Patterns or Mixed Memory/Logic Patterns), all error pipeline cycles must be executed using a single instruction which executes immediately prior to the branch instruction. This rule was added 9/8/2008.*".

Date	Type	Details
10/14/08	Delete	The ecr_x_y_data_set() function was un-documented as a way of deprecating its use. Use ecr_config_set(), to ensure all dependent hardware and software is correctly configured.
10/22/08	Correct	Corrected the legal values for the address argument to apg_user_ram_address_set(). Previously, the first value = 1, now = 0. Similarly, the upper value was reduced by 1. Note that in the pattern language the URAM1 token is equivalent to the address value 0 when using apg_user_ram_address_set().
11/10/08	Add	Added ecr_interleave_get(), ECR Hardware Size Functions.
11/12/08	Change	"Magnum-III" was renamed to "Magnum 2x" "Magnum-II" was renamed to "Magnum 2" "Magnum-I" was renamed to "Magnum 1". The corresponding files on disk were also renamed. Consolidated two sections with the same title into one location: Mixed Memory/Logic Patterns.
12/1/08	Add	Added information regarding the drive capabilies of SPI Port & GPIO Port pins. See page 170.
1/4/08	Add	Significant changes in software release h2.3.19. Added STDF Software.
2/9/09	Correct	Corrected the resolution value for the 8mA range in PTU Current Force, Test & Measure Ranges (was 8uA, now 4uA).
2/12/09	Correct	Magnum 1's maximum DBM read access rate in t_dbm_sequential mode was incorrectly stated to be 20nS/50MHz. This was corrected to 30nS/33MHz in two places: DBM Usage Rules and dbm_config_set().
2/26/08	Delete Correct	Removed the "Note: *not usable on Magnum* 1" comments from the VAR OVER, VCOMP and VPULSE operands, VEC/RPT OVER, VCOMP, VPULSE and operands and VPINFUNC OVER, VCOMP, VPULSE and operands. In the VAR Instruction section, corrected how operands are grouped in VAR Error-control Operands and VAR Misc Operands (some operands moved from one set to the other). This has no effect on test program operation.

Table of Contents

List Of Illustrations

Chapter 1 Magnum System Overview

This section provides an overview of the Magnum 1 configurations, architecture and key hardware features.

- Magnum Configurations
- Multi-Site System Architecture
- Site Assembly Board
- PE Sub-site Architecture
- Pin Electronics (PE)
  - PE Driver
  - PE Comparators
  - Per-pin Parametric Test Unit (PTU)
  - Error Flag vs. Error Latch
  - DC-only Pins
- DC Sub-System
  - DUT Power Supply (DPS)
  - High Voltage Source/Measure Unit (HV)
  - Parametric Measurement Unit (PMU)
  - Parametric Background Voltage
  - DC Test and Measure System
  - DC Source Select MUX
    - DC Comparators and Error Logic
    - DC A/D Converter
  - DC Comparators and Error Logic
  - DC A/D Converter
- Pattern and Timing System
  - Overview
  - Pin Scramble MUX
  - Pin Scramble RAM
  - Timing & Formatting
  - System Clock

- Algorithmic Pattern Generator (APG)
  - APG Controller Engine
    - uRAM
    - vRAM
  - Branch-on-error Logic
  - APG Address Generator
    - Address TOPO RAM
  - APG Data Generator + Data Inversion Logic, JAM Logic
  - APG Chip Selects
  - APG Interrupt Timer
  - APG User RAM
  - Data Buffer Memory (DBM)
    - DBM Architecture
- Logic Vector Memory (LVM)
- Scan Vector Memory (SVM)
- Error Catch RAM (ECR)
- DUT Board I/O Ports
  - I2C Bus
  - SPI Port & GPIO Port

1.1 Magnum Configurations

See Magnum System Overview.

The following Magnum configurations are available:

Table 1.1.0.0-1 Magnum System Configurations

System	Max Boards	Total Channels	Sub-site A Channels	Sub-site B Channels	Site Controllers
Magnum PV	5	640	320	320	5
Magnum SV	10	1280	640	640	10
Magnum SSV	20	2560	1280	1280	20
Magnum GV	40	5120	2560	2560	40

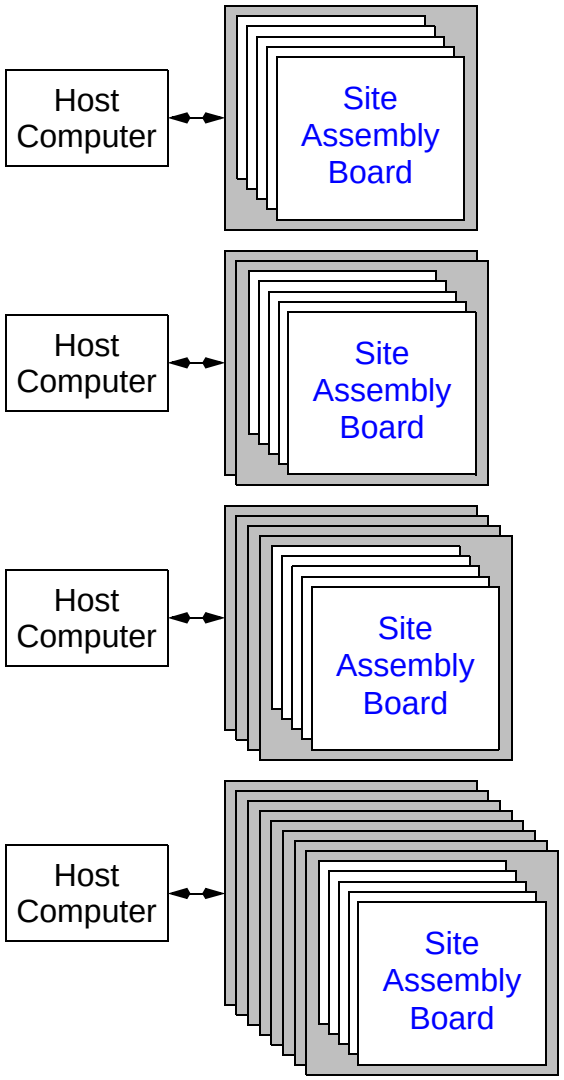
Also supported are:

- DUT Power & Measurement Unit (DPMU) via an Magnum options board

1.2 Multi-Site System Architecture

See Magnum System Overview.

The Magnum is available in several configurations:



Magnum PV
1 Chassis /w 1 to 5 Site Assembly Boards
128 to 640 Digital Pins
8 to 40 DPSs, 8 to 40 PMUs
16 to 80 High Voltage S/M

Magnum SV
2 Chassis /w up to 10 Site Assembly Boards
Up to 1280 Digital Pins, 80 DPSs, 80 PMUs,
160 High Voltage Source/Measure Unit (HV)

Magnum SSV
4 Chassis /w up to 20 Site Assembly Boards
Up to 2560 Pins, 160 DPSs, 160 PMUs,
320 High Voltage Source/Measure Unit (HV)

Magnum GV
8 Chassis /w 40 Site Assembly Boards
Up to 5120 Pins, 320 DPSs, 320 PMUs,
640 High Voltage Source/Measure Unit (HV)

Figure-1: Magnum System Configuration Options

1.3 Site Assembly Board

See Magnum System Overview, Pin Electronics (PE).

The diagram below shows the key components of the Magnum Site Assembly:

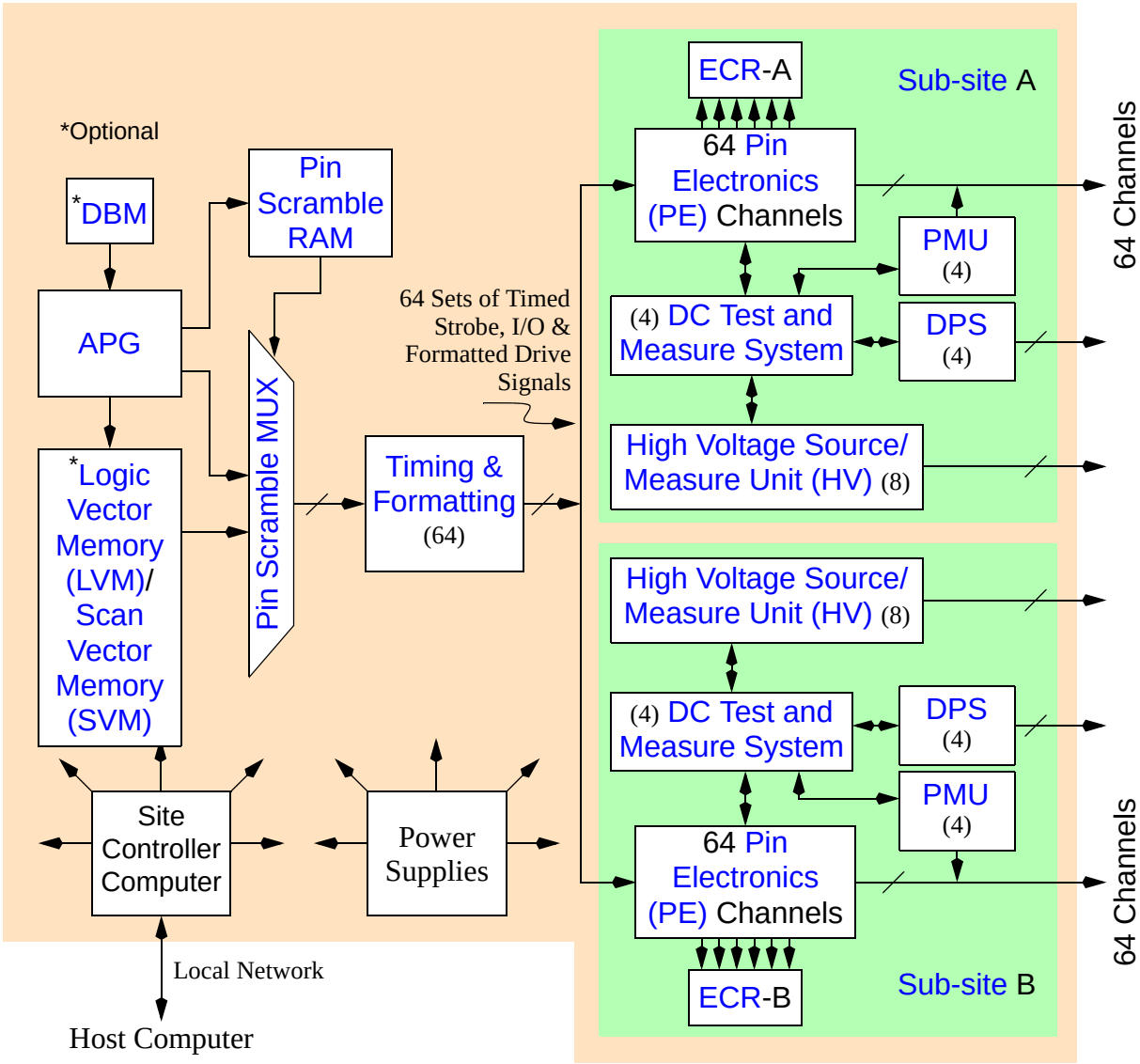


Figure-2: Site Assembly Board Block Diagram

Each Site Assembly Board contains the following:

- 128 test channels. Pairs of test channels share the same timing (strobe, I/O and drive signals) and test pattern data source. See PE Sub-site Architecture and Functional Pin-pairs.
- Each of the 128 test channels has an independent Per-pin Parametric Test Unit (PTU).
- Eight DUT Power Supply (DPS).
- Sixteen High Voltage Source/Measure Unit (HV).
- Eight Parametric Measurement Unit (PMU)s. Each PMU can connect to 16 test pins, 16 Per-pin Parametric Test Unit (PTU)s, 1 DUT Power Supply (DPS), and 2 High Voltage Source/Measure Unit (HV)s.
- Eight DC Test and Measure Systems, each with DC Comparators and Error Logic, used to perform DC Go/NoGo tests, and DC A/D Converter used to make DC measurements.
- Optionally, two Error Catch RAM (ECR).

1.4 PE Sub-site Architecture

See Magnum System Overview, Site Assembly Board.

The Magnum is targeted at parallel testing of multiple DUTs. The sub-site architecture outlined below provides the high pin-counts needed while reducing overall system costs by sharing timing, formatting, and test pattern hardware.

Each Site Assembly Board (each site) contains one set of 64 functional test *channels* which are used to control 128 functional test *pins*. Note the following:

- Half of the 128 pins (a_1, a_2, ... a_64) are collectively called sub-site A pins. Similarly, the other 64 pins (b_1, b_2, ... b_64) are collectively called sub-site B pins.
- Each pin of sub-site A is paired with one corresponding pin of sub-site B. For example, a_1 and b_1, a_13 and b_13, etc.
- During functional tests, each pin-pair receives identical drive, strobe and I/O signals. See Pattern and Timing System and Functional Pin-pairs.
- Each pin of a pin-pair has independent timing deskew circuitry, to correct for signal path differences between the pins and maintain timing accuracy.

- All 128 pins have independent drive levels (VIL/VIH), strobe levels (VOL/VOH), termination voltage (VTT), resistive load (RL) and termination voltage (VZ) and drive super voltage (VIHH). Note that there are three PE Driver modes which do constrain how a given pin-pair can utilize VTT/VZ/VIHH (see PE Driver and Magnum PE Driver Modes).
- All 128 pins have an independent Per-pin Parametric Test Unit (PTU), each with independent force voltage/current and PASS/FAIL test limits. It is the PTU which generates the VIHH and VZ voltages noted in the previous bullet. It also is the PTU which generates the Parametric Background Voltage.
- Each sub-site has an optional Error Catch RAM (ECR), which captures failures from the 64 pins of that sub-site.
- When only one pin of a given pin-pair is used for functional testing, the other pin can be used for DC-only purposes. See DC-only Pins.
- Each Site Assembly Board contains 8 DUT Power Supply (DPS), grouped into 2 sets: 4 A DPS and 4 B DPS. In software these are identified using A/B designations; i.e. `a_dps1a`, `b_dps1a`, etc.
- Each Site Assembly Board has 16 High Voltage Source/Measure Unit (HV)s, grouped into 8 A HV and 8 B HV. In software these are identified using A/B designations; i.e. `a_hv1`, `b_hv1`, etc.
- Each Site Assembly Board contains 8 Parametric Measurement Unit (PMU)s. Each PMU can connect to 16 test pins, 16 Per-pin Parametric Test Unit (PTU)s, 1 DUT Power Supply (DPS), and 2 High Voltage Source/Measure Unit (HV)s. In software, PMUs are identified implicitly via the members of a pin list.
- Each Site Assembly Board has 8 DC Test and Measure Systems, each with DC Comparators and Error Logic, used to perform DC Go/NoGo tests, and DC A/D Converter used to make DC measurements. Each of the DC Test and Measure Systems supports 16 pins, 1 DPS, 2 HV, and 1 PMU.

For a usage overview see Magnum 1, 2 & 2x Parallel Test.

1.5 Pin Electronics (PE)

See Magnum System Overview, Site Assembly Board.

This section includes the Pin Electronics (PE) Block Diagram, and a detailed description of each of the major features, including :

- PE Driver
- PE Comparators
- Per-pin Parametric Test Unit (PTU)
- DC-only Pins

The diagram below shows the main components of the Magnum Pin Electronics:

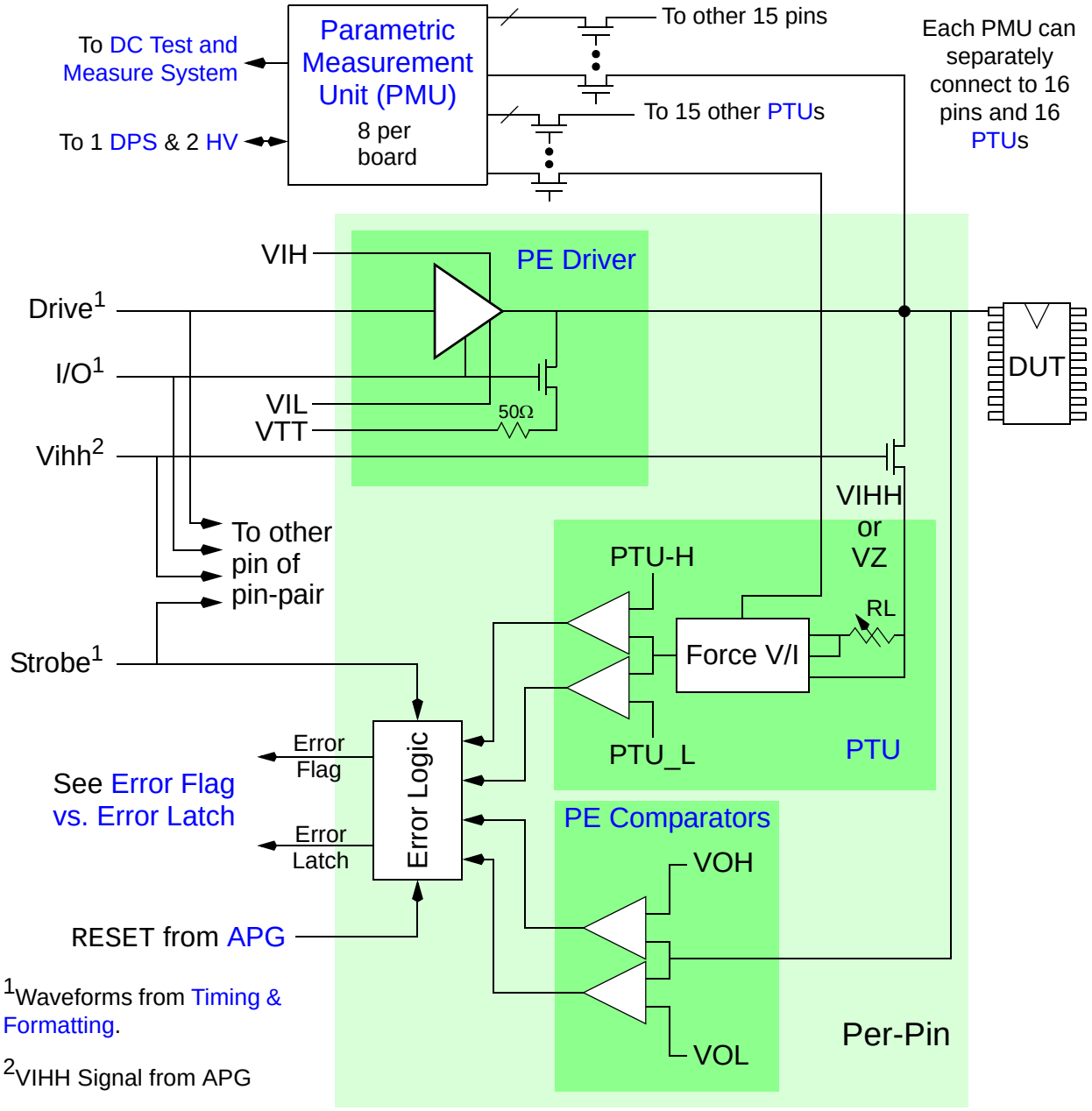


Figure-3: Pin Electronics (PE) Block Diagram

1.5.1 PE Driver

See Pin Electronics (PE), Pin Electronics (PE) Block Diagram

The PE driver generates the digital drive signals applied to the DUT during functional tests. In Magnum, the driver has six possible states:

Table 1.5.1.0-1 Pin Electronics Driver States

State	Set Funcion	Range	Resolution	Termination
Drive High = VIH	`vih()`	-1V to +7V	5mV	50Ω
Drive Low = VIL	`vil()`	-1V to +7V	5mV	50Ω
Terminate to VTT	`vtt()`	-1V to +7V	5mV	50Ω
Terminate to VZ/RL	`vz()`[1]	-1V to +7V	5mV	81Ω to 500K
Drive to 3rd level (VIHH)	`vihh()`[1]	0V to +12.5V	5mV	81Ω
Tri-state	n/a	None. Shown as *Hi-Z* in diagram below = high impedance.	n/a	n/a
Note-1: the actual voltage range of both VZ and VIHH is affected by the amount of current sourced. See PTU Operating Area, `vz()` and `vihh()`.				

Note that not all 6 driver states are usable at one time and that the usable states are affected when using Double Clock Mode. The table below describes 4 modes, indicating the legal combinations of states which can be obtained in each mode:

Table 1.5.1.0-2 Magnum PE Driver Modes

Mode	VIL	VIH	Tri-stat	VTT/50Ω	VZ/RL	VIHH	Comment
Vz Mode	Yes	Yes	Note-1	No	Yes	No	VIHH, VTT & Hi-Z disabled
Vihh Mode	Yes	Yes	Note-2	No	No	Yes	VZ and VTT disabled
Vtt Mode	Yes	Yes	Note-3	Yes	No	No	VIHH, VZ & Hi-Z disabled
Dclk Mode	Yes	Yes	Note-4	No	No	No	Double Clock Mode, a special drive-only mode.

Notes:
1) In Vz Mode, when the driver tri-states the Vz voltage from the PTU is enabled.
2) In Vihh Mode when the driver tri-states the PE channel is high impedance.
3) In Vtt Mode, when the driver tri-states the Vtt voltage is enabled.
4) In Dclk Mode the driver cannot be tri-stated.

Note the following:

- During the initial program load the PE driver mode is set to Vz Mode with RL = 500K. The system software does not otherwise change the driver mode.
- The `pe_driver_mode_set()` function is used to set the PE driver mode. The `pe_driver_mode_get()` function can be used to get the currently set PE driver mode for one pin.
- In Vz Mode, the Vz state is enabled in the test pattern using the `PINFUNC ADHIZ` instruction (Memory Test Patterns) or `X`, `H`, `L`, `V` and `Z` tokens (Logic Test Patterns & Scan Test Patterns). Pin(s) in Vz Mode ignore the test pattern VIHH Map selection. The Hi-Z and Vtt states are not available on pins in Vz Mode.
- In Vihh Mode, the tri-state (Hi-Z) state is enabled in the test pattern using the `PINFUNC ADHIZ` instruction (Memory Test Patterns) or `X`, `H`, `L`, `V` and `Z` tokens (Logic Test Patterns & Scan Test Patterns). Pin(s) in Vihh Mode do respond to the test pattern VIHH Map selection. To enable the VIHH voltage also requires defining one or more VIHH Maps, to identify combinations of pin(s) which will drive to the

Vihh state in a given pattern instruction. In the test pattern, to switch Vihh Mode pin(s) to the Vihh state, a non-default VIHH Map is selected, per cycle, using PINFUNC VIHH# instruction (Memory Test Patterns) or VPINFUNC VIHH# and VEC/RPT VIHH# instructions (Logic Test Patterns). Pin(s) in the Vihh state are not tri-stated by the PINFUNC ADHIZ instructions (Memory Test Patterns) and X, H, L, V and Z tokens (Logic Test Patterns & Scan Test Patterns) i.e. they continue to drive to the VIHH level. Vihh Mode pin(s) which are not switched to the Vihh state in a given cycle will tri-state (Hi-Z). The Vz and Vtt states are not available on pins in Vihh Mode.

- In Vtt Mode, the Vtt state is enabled in the test pattern using the PINFUNC ADHIZ instruction (Memory Test Patterns) or X, H, L, V and Z tokens (Logic Test Patterns & Scan Test Patterns). Pin(s) in Vtt Mode ignore the test pattern VIHH Map selection. The tri-state (Hi-Z) and Vz states are not available on pins in Vtt Mode.
- In Dclk Mode (see Double Clock Mode) the pin is set to drive-only and ignores the various test pattern tri-state signals and the Vihh signal. Values programmed for VZ, VTT and VIHH have no effect on pins in Dclk Mode.

The diagram below shows the components important to PE driver mode:

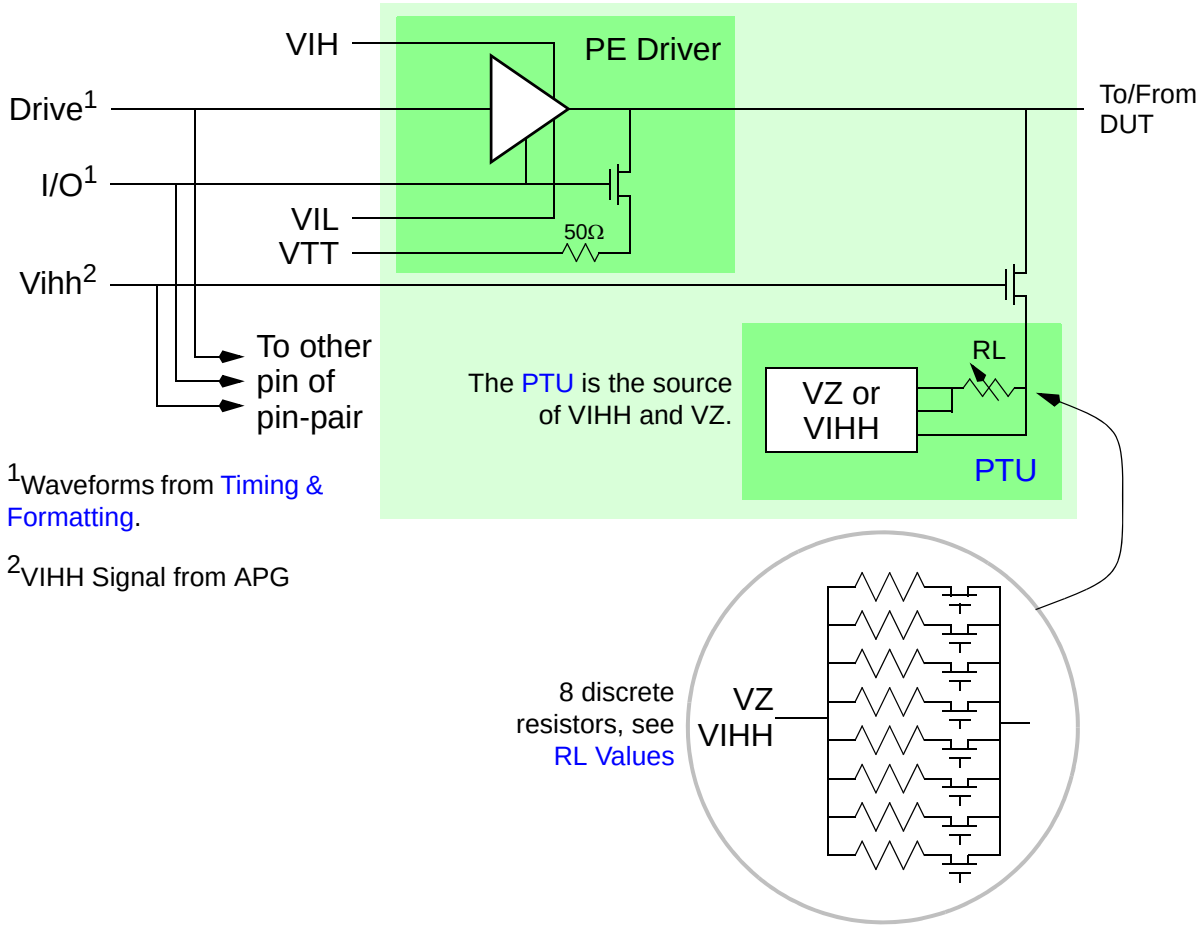


Figure-4: PE Driver Block Diagram

Note the following:

- Each pin has independent DC levels: VIL, VIH, VIHH, VTT, VZ, and independent RL. These voltages may be programmed from the test program and also controlled from an executing test pattern (see Controlling PE Levels from the Test Pattern).
- During pattern execution, the driver will switch states as controlled by the Drive, I/O and Vihh signals from the Timing & Formatting logic, which is itself controlled by signals from the Pin Scramble MUX which selects, cycle by cycle, from the various pattern data sources (APG, LVM/SVM, etc.).

- For any given pin, the other pin of the pin-pair (see Functional Pin-pairs) receives the same Drive, I/O and Vihh signals. However, the PE driver mode can be independently configured for each pin of the pin-pair (see Magnum PE Driver Modes).
- The `vz()` function sets both the VZ level and the value of the termination resistance, RL. There are 8 discrete RL resistor values, which can be used in combination to provide 256 values. Nominal resistor values are ±20% but the FET switch used to connect the PTU to the DUT adds an additional nominal resistance of 50Ω (30-85 ohms, not ±20%). In the table below, the left 3 columns represent just the RL resistors. The right 3 columns include the effect of the 50Ω FET switch resistance:

Table 1.5.1.0-3 RL Values

RL Nominal	Min (-20%)	Max (+20%)	RL Actual Incl 50Ω	Min (/w 30Ω)	Max (/w 85Ω)
500K	400K	600K	500K	400K	600K
125K	100K	150K	125K	100K	150K
31.25K	25K	35.5K	31.3K	25030	37585
7.81K	6248	9372	7.86K	6278	9457
1.95K	1560	2340	2.0K	1590	2425
500	400	600	550	430	685
125	100	150	175	130	235
31	25	37.5	81	55	122.5
The effect of the series 50Ω was not included in the 2 highest resistor values.					

Note that it is possible to combine multiple RL resistors (in parallel only) to obtain other effective values, but the 50Ω (30-85 ohms) must be added to the resulting parallel resistance calculation. Also, the PTU voltage limits and maximum current the PTU can supply may constrain the use of some parallel RL combinations.

1.5.2 PE Comparators

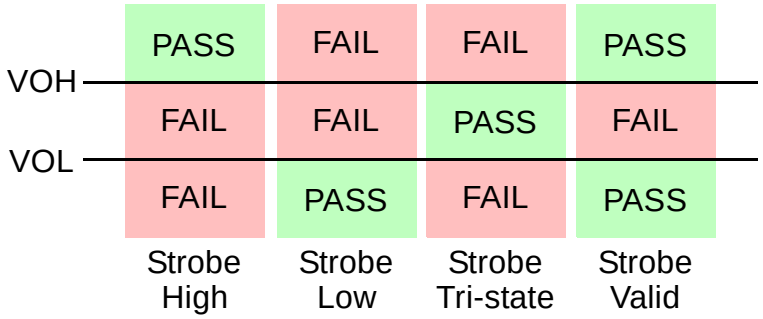
See Pin Electronics (PE), Pin Electronics (PE) Block Diagram

The Magnum PE comparator and error logic are used during functional tests, to test for four logic states:

Table 1.5.2.0-1 PE Comparator Levels

State & Reference Voltage	Set Func.	Range	Resolution
Strobe High = VOH	voh()	-1V to +7V	5mV
Strobe High = VOL	vol()	-1V to +7V	5mV
Strobe Tri-state	Both	Same	Same
Strobe Valid	Both	Same	Same

The diagram below shows each option and how the values of VOL/VOH apply:



The following diagram shows the DC comparators and error logic for one pin:

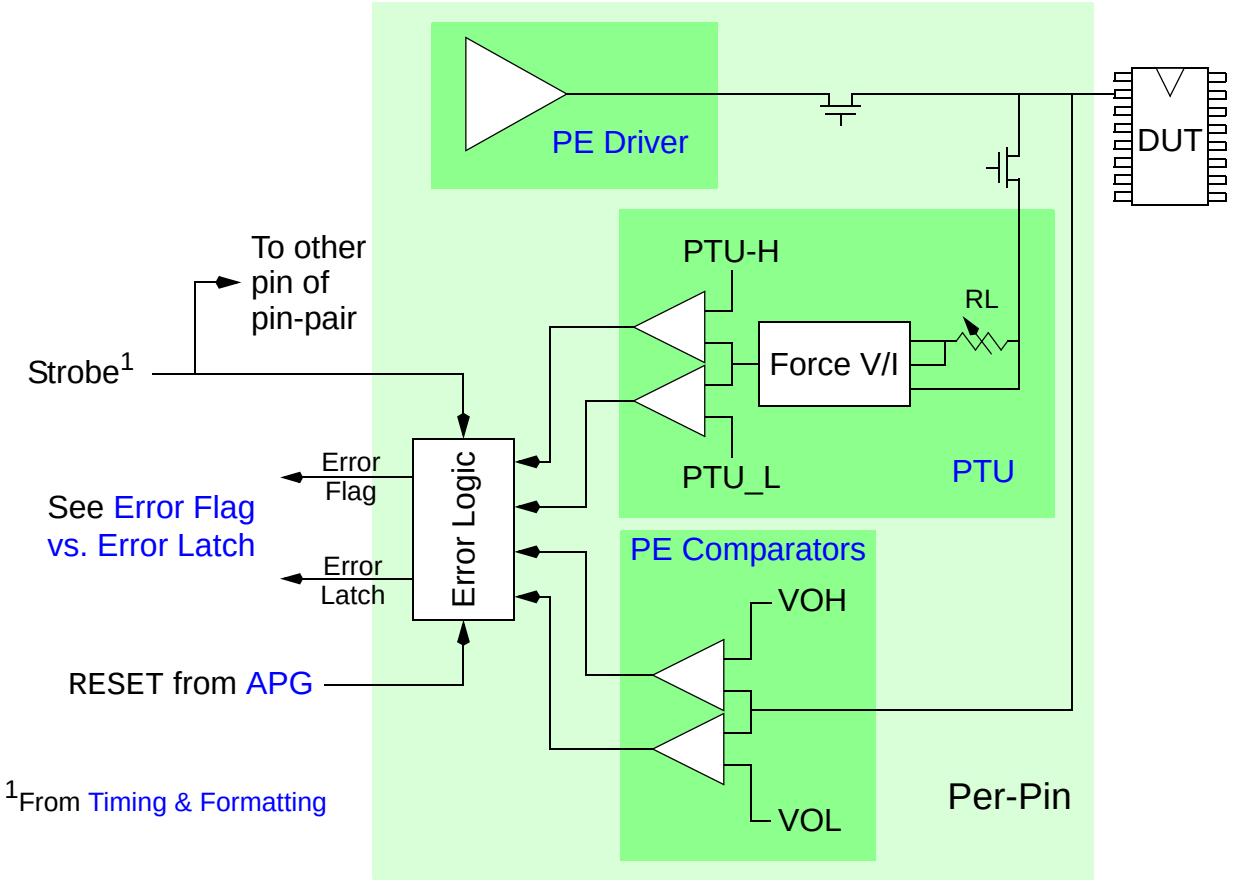


Figure-5: PE Comparators and Error Logic Block Diagram

Note the following:

- During pattern execution, the error logic at the output of the DC comparators will receive strobe signals from the Timing & Formatting logic, which is itself controlled by signals from the Pin Scramble MUX which selects, cycle by cycle, from the various pattern data sources (APG, LVM/SVM, etc.). Each pin-pair (see Functional Pin-pairs) shares the same set of drive, strobe and I/O signals. See Pin Electronics (PE) Block Diagram.
- Strobe signals can generate edge strobes or window strobes, per-pin.
- The comparator error logic detects a FAIL if, during the entire time of a window strobe, or at the time of the 1st edge of an edge strobe, the strobe fails. This will occur as indicated above.

- Strobe enable and polarity is controlled (per-pin, per-cycle) by the test pattern data source selected by the Pin Scramble MUX in each pattern cycle, which is controlled from the test pattern using the PINFUNC PS# instruction (Memory Test Patterns) and VEC/RPT PS# or VPINFUNC PS# instruction (Logic Test Patterns, Scan Test Patterns).
- The error latch output from the error logic determines the overall PASS/FAIL result for each pin. If, at the end of test pattern execution, a given pin's error latch is set that pin fails, as does the functional test.
- The error flag output from the error logic is used by the Algorithmic Pattern Generator (APG) to determine branch-on-error or stop-on-error operation. All per-pin error flags and the DC Error Flags from each DC Test and Measure System are logically OR'ed together, to provide one error signal to the APG. See Error Flag vs. Error Latch.
- It is possible to inhibit the error latch, per-cycle, using one of the NOLATCH test pattern options (MAR NOLATCH in Memory Test Patterns, and VEC/RPT NOLATCH, VAR NOLATCH, or VPINFUNC NOLATCH in Logic Test Patterns). In these instructions, a failing strobe will only set the error flag (not the error latch). This allows test pattern branch-on-error operations to use the error flag signal, reset the flag as needed, and PASS/FAIL in conditional pattern cycles without affecting the overall test pattern PASS/FAIL result. See Error Flag vs. Error Latch.
- The test pattern MAR RESET and CHIPS RESET instructions (Memory Test Patterns) and VEC/RPT RESET, VAR RESET, and VPINFUNC RESET instructions (Logic Test Patterns) will clear the error flag but not the error latch. See Error Flag vs. Error Latch.
- The error logic is also involved during per-pin PTU Go/NoGo tests. The site controller computer strobes the error logic during PTU Go/NoGo tests.

1.5.3 Per-pin Parametric Test Unit (PTU)

See Pin Electronics (PE), Pin Electronics (PE) Block Diagram

Each Magnum PE pin has an independent per-pin Parametric Test Unit (PTU), used for the following:

- A PTU can be used to perform static DC tests to force voltage and test/measure current or vice versa. See PTU Static Test Functions.
- A PTU can be used as statically connected voltage or current source. See PTU as Voltage/Current Source.

- The PE Driver VIHH voltage for a given pin is supplied by the pin's PTU. See PE Driver Block Diagram, VIHH Voltage, and VIHH Maps.
- The PE Driver VZ voltage for a given pin is supplied by the pin's PTU. See PE Driver Block Diagram and PE Load Reference Voltage: VZ.
- The Parametric Background Voltage for a given pin is supplied by the pin's PTU. See Background Voltage Functions.

The PTU has a single force voltage range, programmed using PTU Force-voltage Functions:

Table 1.5.3.0-1 PTU Force Voltage Range

Parameter	Range	Resolution	Comments
PTU Force Voltage	-2V to +12V	1mV	Also supplies VIHH voltage and VZ voltage (see PE Driver) and the Parametric Background Voltage.

The PTU may be used to test or measure voltage. Pass/Fail test limit values are programmed using PTU Voltage Test Limit Functions:

Table 1.5.3.0-2 PTU Voltage Test/Measure Range

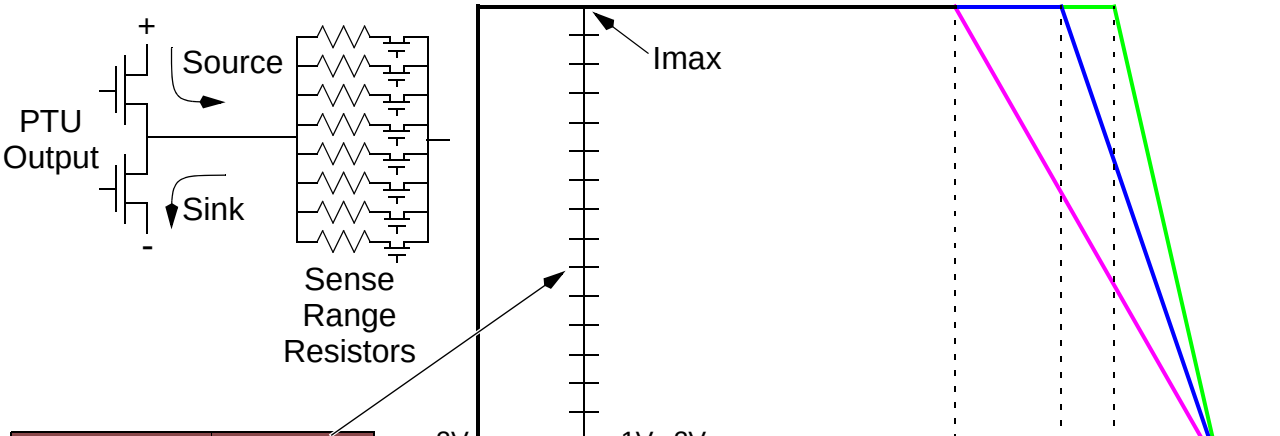
Parameter	Range	Resolution
PTU Measure Voltage	-2V to +12V	2mV

The PTU may be used to force current and test or measure current. The same ranges apply:

Table 1.5.3.0-3 PTU Current Force, Test & Measure Ranges

Parameter	Range	Resolution
PTU Force, Test & Measure Current	±2uA	1nA
	±8uA	4nA
	±32uA	16nA
	±128uA	64nA
	±512uA	256nA
	±2mA	1uA
	±8mA	4uA
	±32mA	16uA

The diagram below illustrates the V/I operating areas for the PTU based on which current range is enabled. This model also applies to VIHH and Parametric Background Voltage usage:



Range Imin/ IMax	Tick Value
±2uA	125nA
±8uA	500nA
±32uA	2uA
±128uA	8uA
±512uA	32uA
±2mA	125uA
±8mA	500uA
±32mA	2mA

Figure-6: PTU Operating Area

The diagram above should be consulted to determine the PTU operating range based on the selected current range. This applies when the PTU is forcing current or forcing voltage. A simplified model of the PTU output structure is shown, to help explain the diagram. For example, given a PTU force current = -28mA (i.e. the PTU is configured to sink a constant 28mA) the minimum PTU output voltage is +1.5V (which might seem strange). To obtain -28mA (2nd Y-axis tick mark from the bottom) requires the PTU be set to the ±32mA current range. In the diagram, this range is indicated by the magenta line, which intersects the -28mA value at point **A**. Following this up, the minimum voltage is interpolated as half-way between +1V and +2V; i.e. +1.5V. Conversely, if the PTU is configured to force +1.5V on the

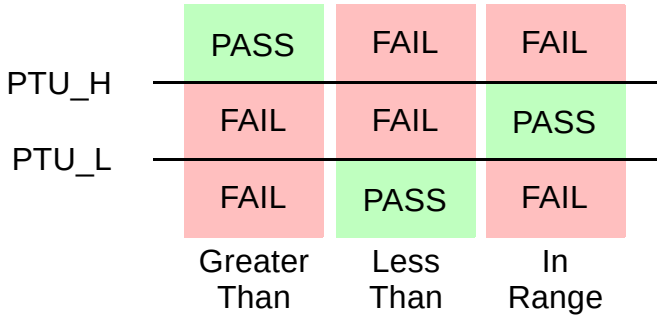
±32mA current range, the PTU will sink up to -28mA max, again indicated by point **A**. In both cases, it is the voltage drop across the PTU output structure and sense resistor which limits the output voltage.

Each PTU uses a set of local DC comparators to perform Go/NoGo tests. To make a measurement, the sense signal is routed, via the Parametric Measurement Unit (PMU), to the DC A/D Converter. See DC Sub-System.

The PTU Static Test Functions can test for three PASS/FAIL states:

- Greater-than PTU_H
- Less-than PTU_L
- In-range, between PTU_H and PTU_L

The diagram below shows each option and how the values of PTU_H and PTU_L apply:



Each PTU has two programmable voltage clamps, which are programmed using PTU Voltage Clamp Functions. PTU voltage clamps are only active when the PTU is in the force-current mode:

Table 1.5.3.0-4 PTU Voltage Clamp Range

Range	Resolution	
0.5V to +12V	4mV	Positive Clamp
-2V to +11V		Negative Clamp

PTU force and test limits may be programmed from the test program or from an executing test pattern (see Controlling PE Levels from the Test Pattern).

A PTU test on a given pin uses the same fail logic as the functional comparators on that pin.

1.5.4 Error Flag vs. Error Latch

See Pin Electronics (PE), Pin Electronics (PE) Block Diagram

With respect to failing strobes in functional tests, there are two important hardware error signal types:

- Error *Flag*
- Error *Latch*

Figure-7: and Figure-8: are are used to describe how these signals operate:

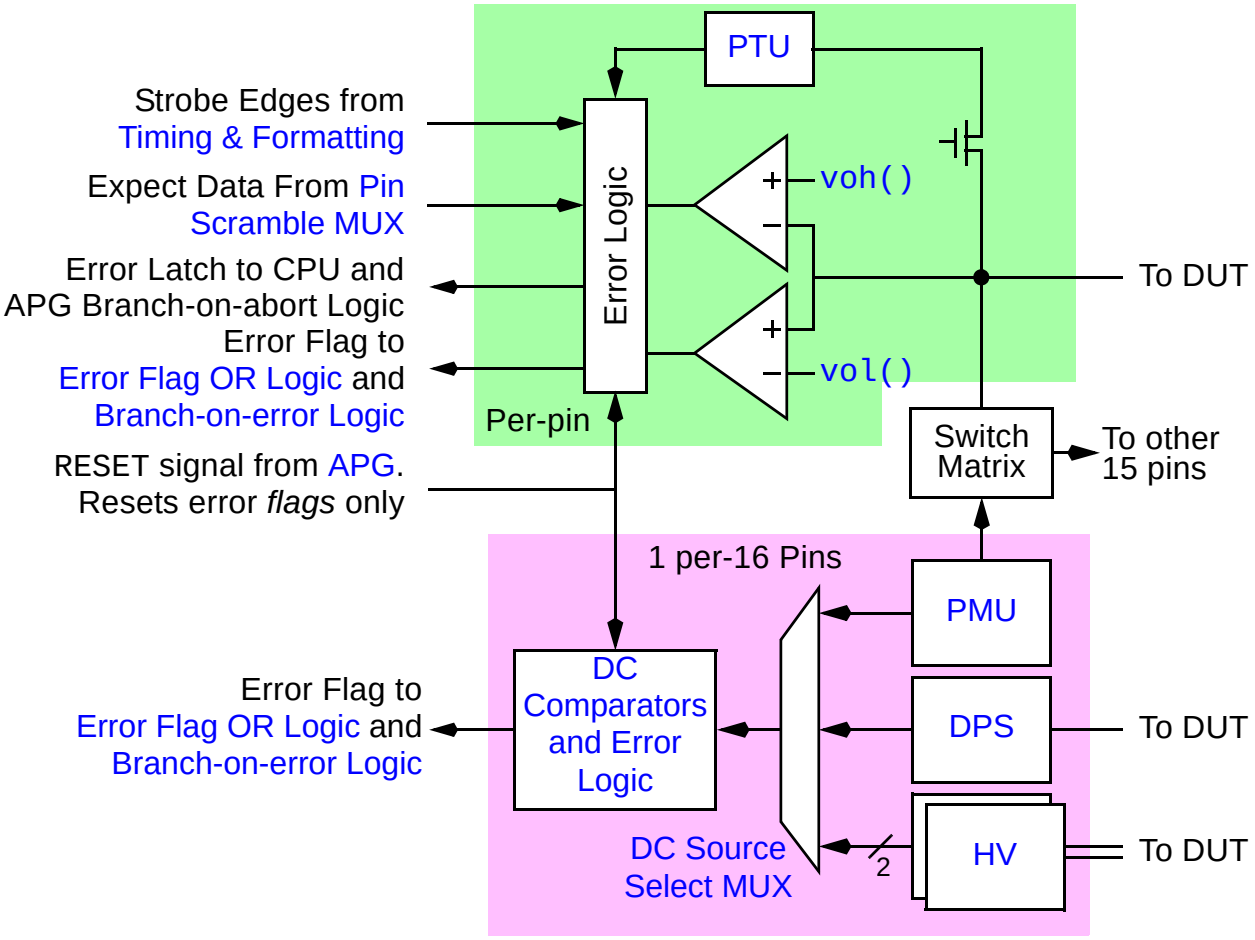


Figure-7: PE Error Flag vs. Error Latch Diagram

The error *latches* have the following attributes:

- Each tester channel has a separate error latch signal.

- The overall PASS/FAIL results of a functional test is solely determined by reading the error latches, after test pattern execution has ended. The error latches are also read by Dynamic DC Tests, to determine if the functional pattern passed or failed.
- Error latches are set in pattern instructions which generate a failing strobe. Strobes are generated in Memory Test Patterns using the various `MAR READ` operations (`READUDATA`, `READZ`, etc.) and in Logic Test Patterns and Scan Test Patterns using the H, L, V and Z Logic Vector Bit Codes. In Memory Test Patterns since `READ` is the default, any memory test pattern instruction performing a `READ` without an explicit `NOLATCH` will set an error latch if one or more strobes fail.
- In Logic Test Patterns error latches are set by any failing strobe(s) in instructions without an explicit `VEC`/`RPT NOLATCH`, `VAR NOLATCH`, or `VPINFUNC NOLATCH` instruction.
- It is the error latch signals which affect test pattern branch-on-abort operations.
- Error latches can not be dynamically cleared using test pattern instructions; i.e. once an error latch is set the test result will be FAIL. Error latches can be reset only from the test site controller, either by the system software in preparation for the next test, or by calling `reset_error()` from user C Code (rarely needed).
- Using `pin_info()`, individual error latch signals can be read to identify which tester pins have failed.
- It is the error latch signals (not error flags) which are logged to the ECR using Logic Error Catch (LEC) or executing `funtest()` using the `fullec` execution option.

The following diagram shows how 8 error flags for each set of 8 pins are logically OR'ed together to generate a single output signal:

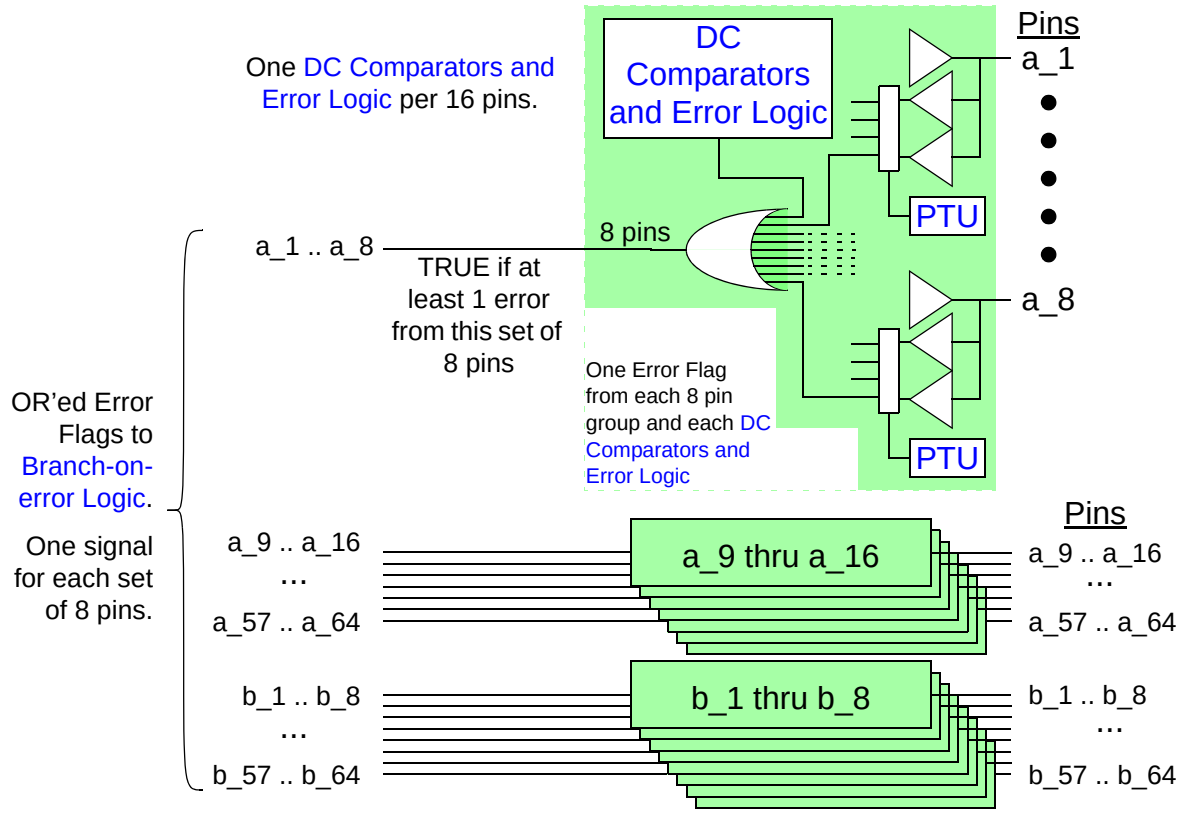


Figure-8: Error Flag OR Logic

The error *flags* have the following attributes:

- Each group of 8 pins has a single error *flag* output, representing the logical OR of the error flags of those 8 PE channels.
- Each group of 16 pins has one error flag output from the DC Comparators and Error Logic associated with those pins. The latter is used by both Static DC Tests and Dynamic DC Tests when measurements are disabled. Further, the error flags from all pins on a Site Assembly Board are logically OR'ed in to form a single error flag signal which is used for branch-on-error and stop-on-error operations in the test pattern. See Figure-8: . Note that the branch-on-abort operation tests the error latches not the error flags.
- When performing a functional test the per-pin error flags have no effect on overall PASS/FAIL test results; i.e. only the error latches are considered.

- When performing Dynamic DC Tests (all of which execute a test pattern) the DC Error Flag output from the DC Comparators and Error Logic determines whether the DC portion of the test passes or fails, and the PE error *latches* determine whether the functional pattern passed or failed.
- When executing Memory Test Patterns, error flags can be set without setting the error latch using `READ NOLATCH` in the pattern instruction. Executing Logic Test Patterns, error flags can be set without setting the error latch using using `VEC`/`RPT NOLATCH`, `VAR NOLATCH`, or `VPINFUNC NOLATCH` instruction.
- Traditionally, test pattern *branch-on-error* and *stop-on-error* operations are based solely on these error flags. However, using Magnum, the ECR Counter Comparators can also be treated as error sources.
- Some conditional operations evaluate the logical OR of all errors in a test site while others consider error signals organized by DUT. The user must understand some of the hardware architecture when selecting which signals are selected as the error source. Details are covered in Branch-on-error Logic.
- In Memory Test Patterns, error flags can be dynamically cleared (reset) on-the-fly from the pattern using the `MAR RESET` instruction. Combined with `READ NOLATCH` this allows the pattern instruction sequence to branch about, based on strobe PASS/FAIL results AND DC comparator PASS/FAIL test results, without causing the overall test to fail.
- In Logic Test Patterns, the `RESET` operation noted above is controlled using the `VEC`/`RPT RESET`, `VAR RESET`, or `VPINFUNC RESET` instructions.

1.5.5 DC-only Pins

See Pin Electronics (PE).

The Magnum 1, 2 & 2x Parallel Test software supports the concept of DC-only pins. Note the following:

- DC-only pins are not formally defined. Rather, any digital pin can be effectively made into a DC-only pin by executing `pin_dc_state_set()`.
- Using `pin_dc_state_set()`, the PE Driver of specified pin(s) may be statically set to drive to logic-1, logic-0, or tri-state. The Magnum PE Driver Modes does apply to DC-only pins.
- `pin_dc_state_set()` provides two options related to whether the specified pin(s) respond to test pattern stimuli:

  - If the `hold_state` argument is set TRUE the specified pin(s) will not receive test pattern signals and thus will remain the specified DC state until `pin_dc_state_set()` is executed again to change the pin mode. This is the basis for a DC-only pin.
  - If the `hold_state` argument is set FALSE the specified pin(s) will receive test pattern signals and change state accordingly. This takes a pin out of the DC-only state.
- See `pin_dc_state_set()` for additional details.

1.6 DC Sub-System

See Magnum System Overview, Site Assembly Board Block Diagram, Pin Electronics (PE) Block Diagram.

The DC sub-system is used to test or measure voltage or current from a DUT Power Supply (DPS), a High Voltage Source/Measure Unit (HV), or a Parametric Measurement Unit (PMU).

The block diagram below shows the main components of the DC sub-system:

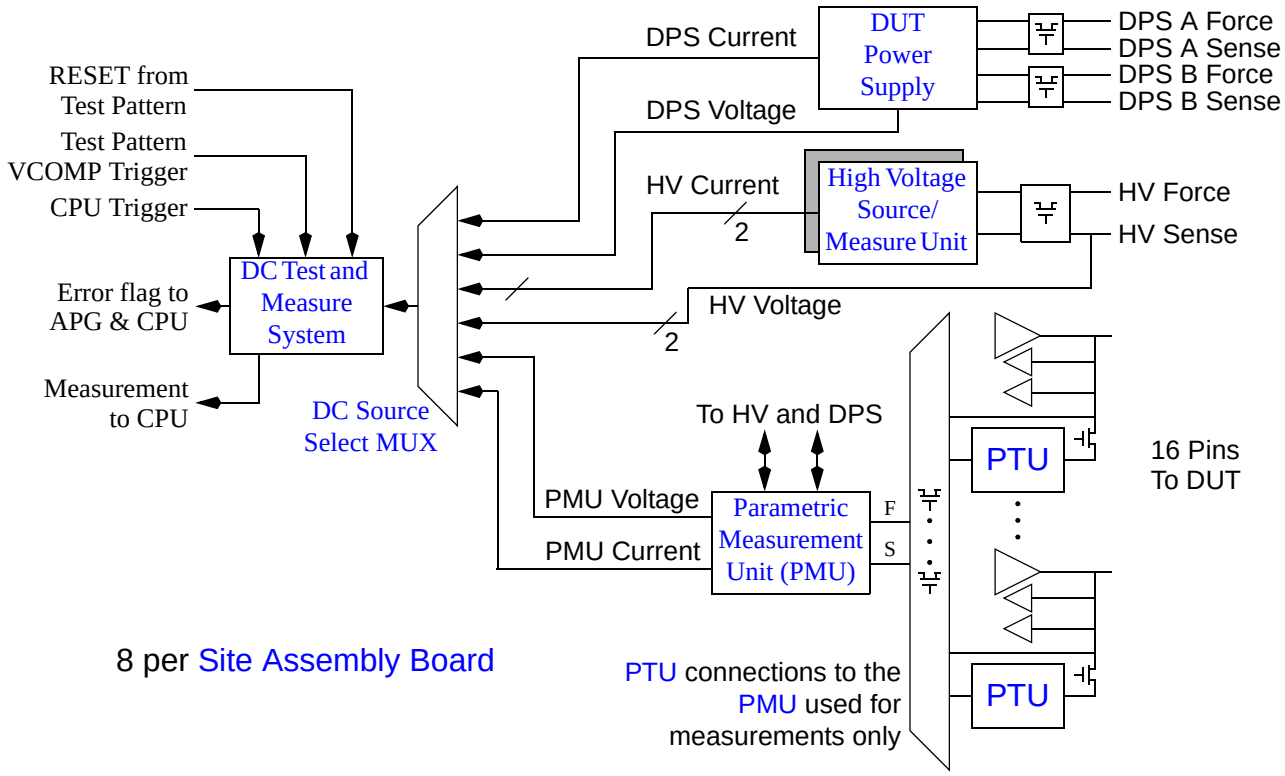


Figure-9: DC Sub-System Block Diagram

Note the following:

- This circuit is replicated 8 times on each Site Assembly Board. Each instance connects one DUT Power Supply (DPS), 2 High Voltage Source/Measure Unit (HV), and 1 Parametric Measurement Unit (PMU) to the DC Source Select MUX. The PMU can connect to 16 DUT pins, 16 PTU, 2 HV and 1 DPS.
- The DC Source Select MUX selects which DC instrument is routed to the DC Test and Measure System. The DC Source Select MUX is controlled by system software, based on the type of test being performed.
- It is the DC Test and Measure System which contains the DC comparators used to perform Go/NoGo DC parametric tests (except for the PTU, more below).
- It is the DC Test and Measure System which contains the A/D converter used to make DC parametric measurements, including PTU measurements.
- When using the PTU, the DC Test and Measure System is only used when making a measurement (PTU Go/NoGo tests do not use the DC Test and Measure System).

- When making a PTU measurement, only one pin at a time can be measured. The PMU selects and routes the voltage/current signal to the DC Source Select MUX when making PTU measurements.
- The DC Test and Measure System can only test or measure voltage. Thus, to test or measure a current value, the DUT Power Supply (DPS), High Voltage Source/ Measure Unit (HV), Per-pin Parametric Test Unit (PTU), and Parametric Measurement Unit (PMU) generate a voltage proportional to the current being supplied. It is this voltage that is tested or measured when performing a current test.

1.6.1 DUT Power Supply (DPS)

See DC Sub-System , Pin Electronics (PE), Pin Electronics (PE) Block Diagram.

Each Site Assembly Board contains eight DPS, 1 for each 16 pins.

Note the following:

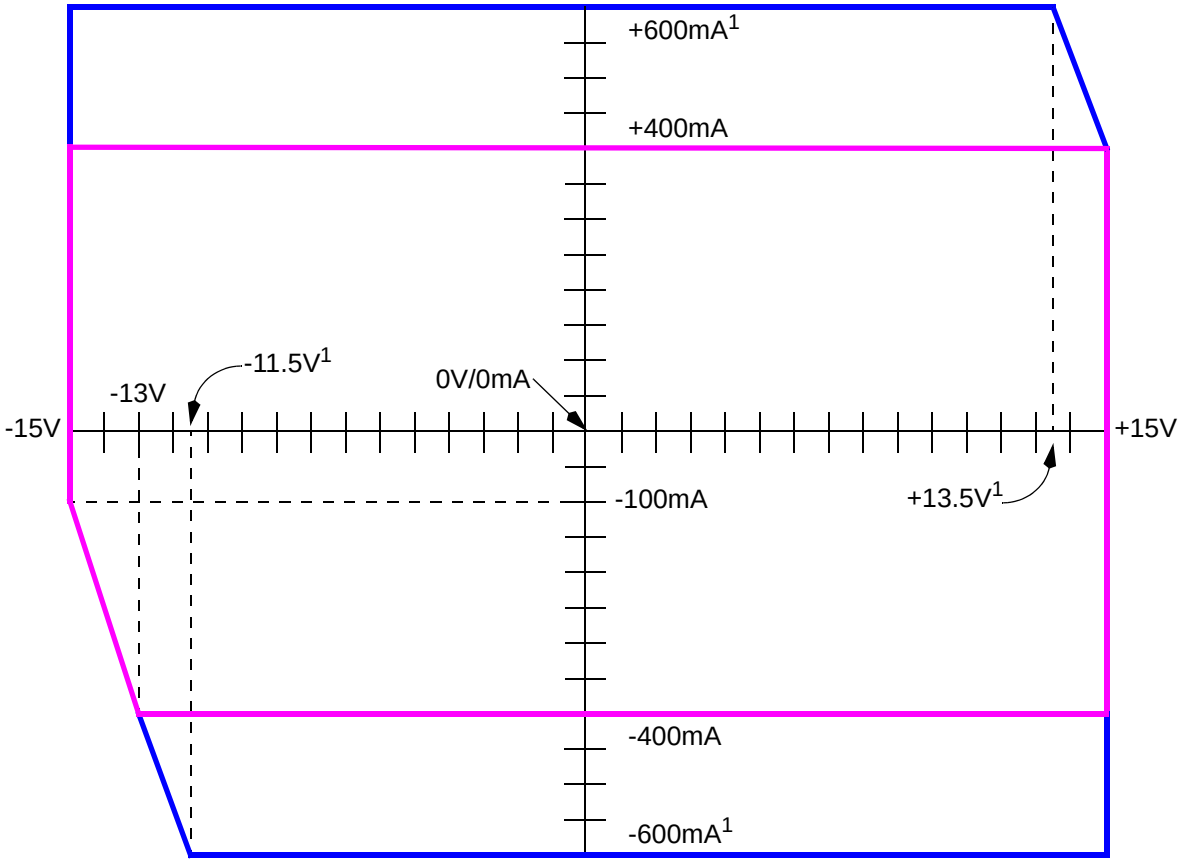
- Each DPS has two independently switchable outputs (split DPS)
- The two outputs (A&B) can operate in 2 modes (see DPS Voltage Programming Functions and DPS Output Mode):
  - In VPulse mode, both DPS outputs are set to the same voltage and may switch to an alternate voltage.
  - In Independent mode, the two DPS outputs may be programmed to different voltages but the test pattern Vpulse control cannot be used.
- A solid-state switch allows each DPS output (A&B) to connect to or disconnected from the DUT. User code must explicitly control these connections. See DPS Connect/Disconnect Functions.

- The DPS has a single force voltage range, which applies in both VPulse and Independent modes:

Table 1.6.1.0-1 DPS Force Voltage Range

Voltage Range	Max Current	Resolution	Notes
-15V to +15V -11.5V to +13.5V See DPS Operating Area diagram below	±400mA ±600mA	5mV	Only one DPS of 4 can output negative voltage. Note the ±600mA maximum current is only available with the DPS 300mA/600mA DPS Option.
The current from each output (A&B) cannot exceed the 1/2 the total available DPS output current specification.			

- The following diagram shows the operating area of the DPS:



Note-1: ±600mA (blue) applies only when using the DPS 300mA/600mA DPS Option.

Figure-10: DPS Operating Area

- The DPS output current can be tested or measured using the DC Test and Measure System. See DPS Static Current Test Functions and DPS Dynamic Current Test Functions. The DPS has the following current sense ranges (see for DPS Current Test Limit Functions details):

Table 1.6.1.0-2 DPS Test/Measure Current Ranges

Current Range	Resolution	Notes
±4uA	2nA	
±40uA	20nA	
±400uA	200nA	
±4mA	2uA	
±40mA	20uA	
±400mA	200uA	
±600mA	2mA	±600mA only when using the DPS 300mA/600mA DPS Option.
±4A		Only useable in DPS Current Sharing mode.

- Multiple DPS can be connected in parallel to obtain current greater than that available from one DPS. See DPS Current Sharing.
- Each DPS has a corresponding Parametric Measurement Unit (PMU) which can be switched to replace the DPS to perform PMU: Testing DPS Pins.

- The diagram below shows the DPS output structure. Each Site Assembly Board has 8 of these:

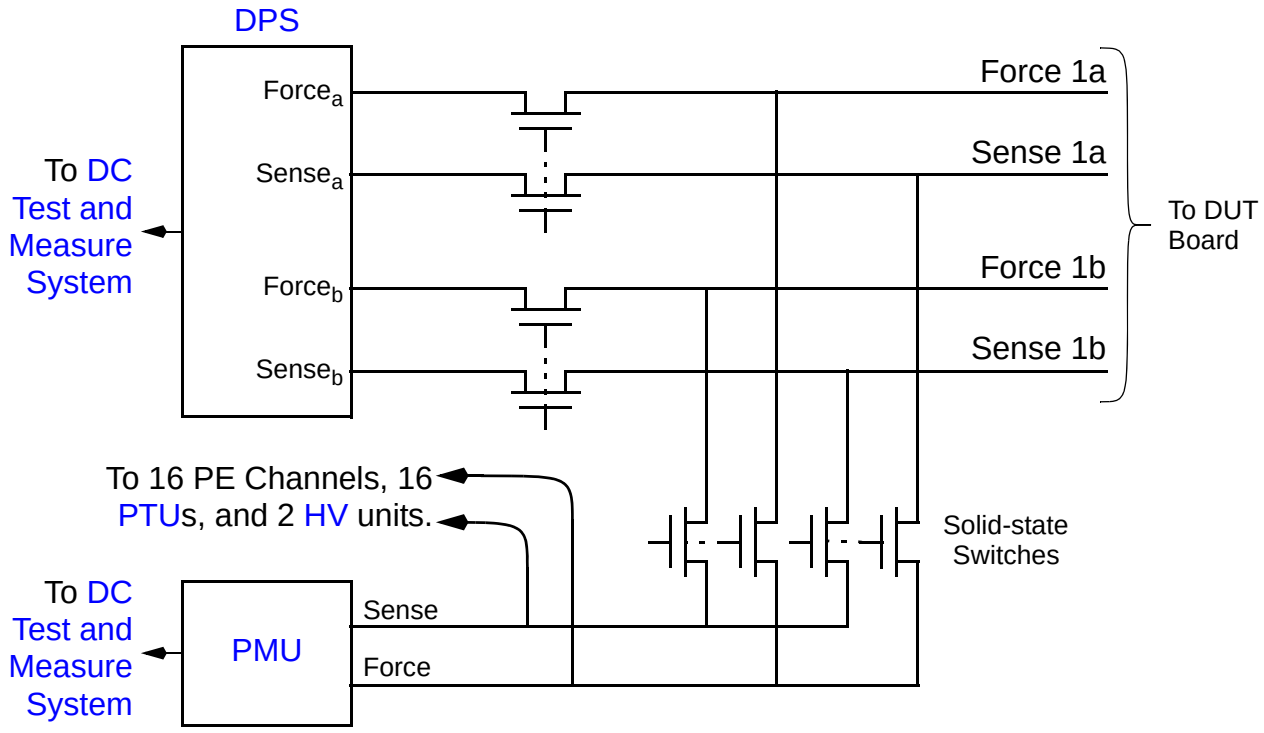


Figure-11: Magnum DPS Output Block Diagram

- Not shown is the current sense circuitry, which operates differently depending on the output mode of the DPS (see DPS Output Mode).
  - In Vpulse mode, the combined output current is sensed and can be tested or measured.
  - In independent mode, the output current from one output at a time can sensed and tested or measured.

1.6.2 High Voltage Source/Measure Unit (HV)

See Pin Electronics (PE), Pin Electronics (PE) Block Diagram.

Each Site Assembly Board contains 16 high voltage source/measure units, effectively one for each 8 pins. Note the following:

- See High Voltage Source/Measure Unit (HV) Functions.

- A solid-state switch allows each HV to connect to or disconnected from the DUT. User code must explicitly control these connections. See HV Connect/Disconnect Functions.
- The HV has a single force voltage range:

Table 1.6.2.0-1 HV Force Voltage Range

Voltage Range	Resolution	Max Current	Load
0V to +28V	2mV	8mA	Up to 0.01uF

- The HV output current can be tested or measured using the DC Test and Measure System. See HV Static Test Functions and HV Dynamic Test Functions. The HV has 1 current sense range:

Table 1.6.2.0-2 HV Test/Measure Current Range

Current Range	Resolution
0mA to 8mA	4uA

- The HV output voltage can be tested or measured using the DC Test and Measure System. See HV Static Test Functions and HV Dynamic Test Functions. The HV has 1 voltage sense range:

Table 1.6.2.0-3 HV Test/Measure Voltage Range

Voltage Range	Resolution
0V to +28V	4mV

- Each two HV units have a corresponding Parametric Measurement Unit (PMU) which can be switched to replace HV (one at a time) to perform PMU: Testing HV Pins, typically continuity tests.

- The diagram below shows the HV output structure:

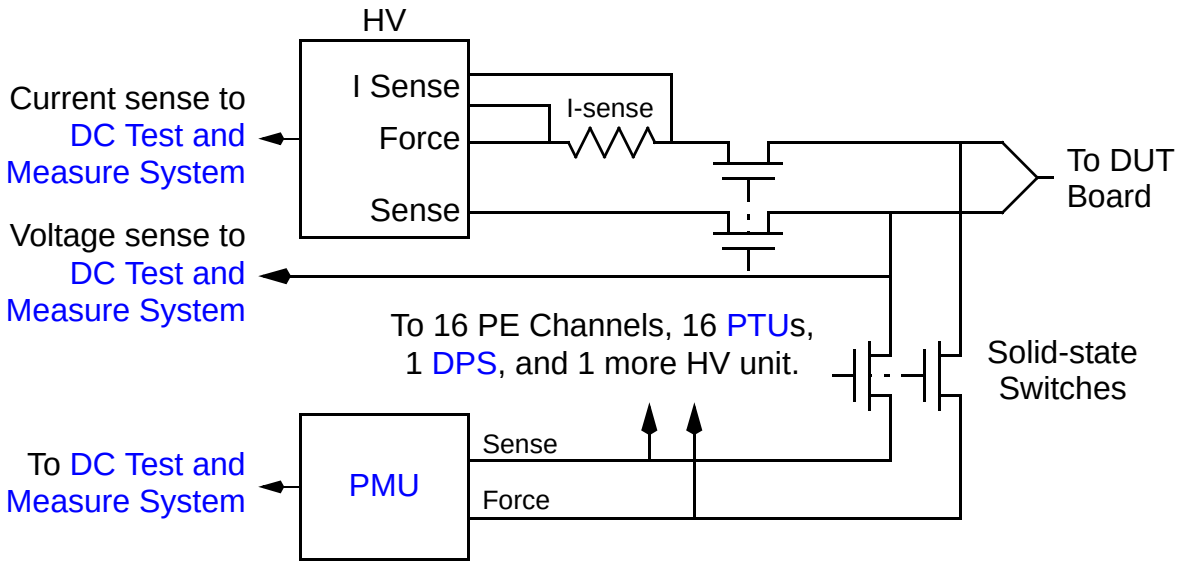


Figure-12: Magnum HV Output Block Diagram

1.6.3 Parametric Measurement Unit (PMU)

See Pin Electronics (PE), Pin Electronics (PE) Block Diagram.

Each Site Assembly Board contains eight PMUs, 1 for each 16 pins. These PMUs can be used to perform conventional DC parametric tests, and as a statically connected programmable voltage or current source.

A solid-state switch matrix allows each PMU to connect to, and test or measure, 16 PE pins, 16 Per-pin Parametric Test Unit (PTU)s (for measurements only), one DUT Power Supply (DPS), and two High Voltage Source/Measure Unit (HV)s.

The PMU has a single force voltage range:

Table 1.6.3.0-1 PMU Force Voltage Range

Voltage Range	Resolution	Max Current	Notes
-2.5V to +12.75V	1mV	±20mA	On PE Pins
-5V to +15V			On DPS Pins
-2.5V to +15V			On HV Pins
Note: while this may seem to be three voltage ranges, in hardware only one range exists. The system software will limit the force voltage if/when the PMU is connected to, or testing PE, DPS, or HV pins. The broader voltage output capabilies are only usable when the PMU is connected to replace a DUT Power Supply (DPS) (see PMU: Testing DPS Pins) or High Voltage Source/Measure Unit (HV) (see PMU: Testing HV Pins).			

The PMU has the following voltage measure ranges:

Table 1.6.3.0-2 PMU Measure Voltage Ranges

Range	Resolution	
-2.5V to +4V	1mV	On PE Pins
-2.5V to +12.75V	4mV	
-5V to +15V	4mV	On DPS Pins
-2.5V to +15V	4mV	On HV Pins
Note: the system software limits the measure voltage range to the values noted depending on the hardware connection being tested.		

PMU voltage measure range is programmed implicitly or explicitly using the PMU Voltage Test Limit Functions.

The PMU has a five force current ranges:

Table 1.6.3.0-3 PMU Force Current Ranges

Current Range	Resolution
±2uA	1nA
±20uA	10nA
±200uA	100nA
±2mA	1uA
±20mA	10uA

The diagram below illustrates the different operating areas for the PMU based on how it is connected. As indicated in the tables above, the operating range of the PMU depends upon whether it is connected to PE channels or to a DPS or HV unit:

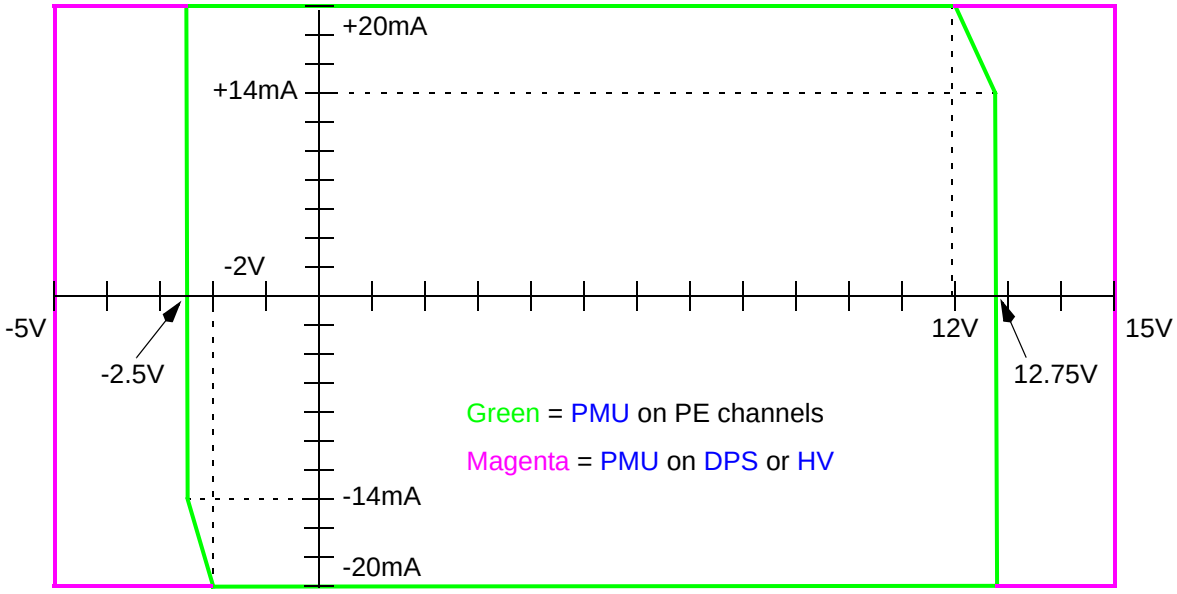


Figure-13: PMU Operating Area

PMU voltage measure range is programmed implicitly or explicitly using the PMU Voltage Test Limit Functions.

In force voltage mode, the PMU current sense circuitry generates a voltage proportional to the amount of current supplied by the PMU. This current sense signal can be tested or measured using the DC Test and Measure System. The PMU has 5 current ranges:

Table 1.6.3.0-4 PMU Test/Measure Current Ranges

Current Range	Resolution
±2uA	1nA
±20uA	10nA
±200uA	100nA
±2mA	1uA
±20mA	10uA

The PMU has a two voltage clamps:

Table 1.6.3.0-5 PMU Voltage Clamp Range

Range	Resolution	
-5V to +16V	100mV	Positive Voltage Clamp
-6V to +15V		Negative Voltage Clamp

PMU voltage clamps are programmed using PMU Voltage Clamp Functions. The voltage clamps are only active when the PMU is in the force-current mode. PMU voltage clamps can affect the PMU force voltage and PASS/FAIL limits.

See PMU as Voltage/Current Source for additional details about using the PMU as a statically connected voltage or current source. The rest of this section addresses using the PMU to perform DC parameteric tests, making measurements, etc.

Connections are made to the PMU a solid-state switch matrix. When performing standard PMU tests, connections between the PMU and the DUT are sequenced automatically; i.e. PMU Connect/Disconnect Functions are ***not needed***. When using the PMU as Voltage/ Current Source, explicit connections are controlled using the PMU Connect/Disconnect Functions.

In the force-voltage mode, the voltage is programmed using the PMU Force Voltage Functions. In the force-current mode, the current is programmed using the PMU Force

Current Functions. PMU current ranges are normally set automatically, based on the force voltage/current values programmed. It is also possible to explicitly set ranges using the PMU Current Test Limit Functions.

The `parametric_mode()` function can be used to determine if the most recently executed PMU test was testing current or voltage.

In the force-voltage mode, a voltage representing the PMU output current is routed to the DC Test and Measure System via the DC Source Select MUX. In the force-current mode, the PMU output voltage is routed to the DC Test and Measure System via the DC Source Select MUX. The DC Source Select MUX will select the PMU test voltage any time one of the PMU Static Test Functions or PMU Dynamic Test Functions is executed. The test voltage is used to perform Go/NoGo PMU current tests, using the DC Comparators and Error Logic, and can also be measured (more below). PMU current PASS/FAIL test limits are programmed using the PMU Current Test Limit Functions.

Both static and dynamic PMU tests are supported, and executed using PMU Static Test Functions or PMU Dynamic Test Functions. Also see Static DC Tests and Dynamic DC Tests.

The system software provides a Built-in Settling Time to PMU current tests. The user may use the `partime()` function to add additional settling time. See Parametric Settling Time.

It is possible to switch the PMU to temporarily replace one of the DUT Power Supply (DPS). This is typically done when the PMU's additional accuracy or voltage/current range is needed to perform a special test. See PMU: Testing DPS Pins.

The `start_ac_partest(), stop_ac_partest()` functions can be used to perform a specialized DC parametric test, where the PMU configuration and connections to the DUT are explicitly set up, user-written code executes as desired, and finally the PMU is disconnected. The user code typically executes test patterns which trigger the DC Comparators and Error Logic or DC A/D Converter, etc.

1.6.4 Parametric Background Voltage

The parametric background voltage is used to bias a set of pins while testing another [adjacent] pin. Having a background voltage facility simplifies using the Parametric Measurement Unit (PMU) to perform continuity tests (opens/shorts testing) and [adjacent channel] leakage tests.

Since the PMU is not a per-pin resource, using the PMU typically requires a sequential test, where all pins are forced to a low (or high) voltage, i.e. the background voltage, and one pin

at a time is forced to the opposite voltage and tested for current. In systems without background voltage capability, the functional drivers are used, increasing the complexity of writing a reliable and portable test program.

While the background voltage facility is not as beneficial when using the Per-pin Parametric Test Unit (PTU), a sequentual PTU test does support its use (a parallel PTU test, the default mode, does not use the background voltage mechanism.

Using the Site Assembly Board, the background voltage is generated by the Per-pin Parametric Test Unit (PTU). The background voltage value is programmed using Background Voltage Functions.

The background voltage has a single voltage range:

Table 1.6.4.0-1 Background Voltage Range

Range	Resolution
0V to +12V	1mV

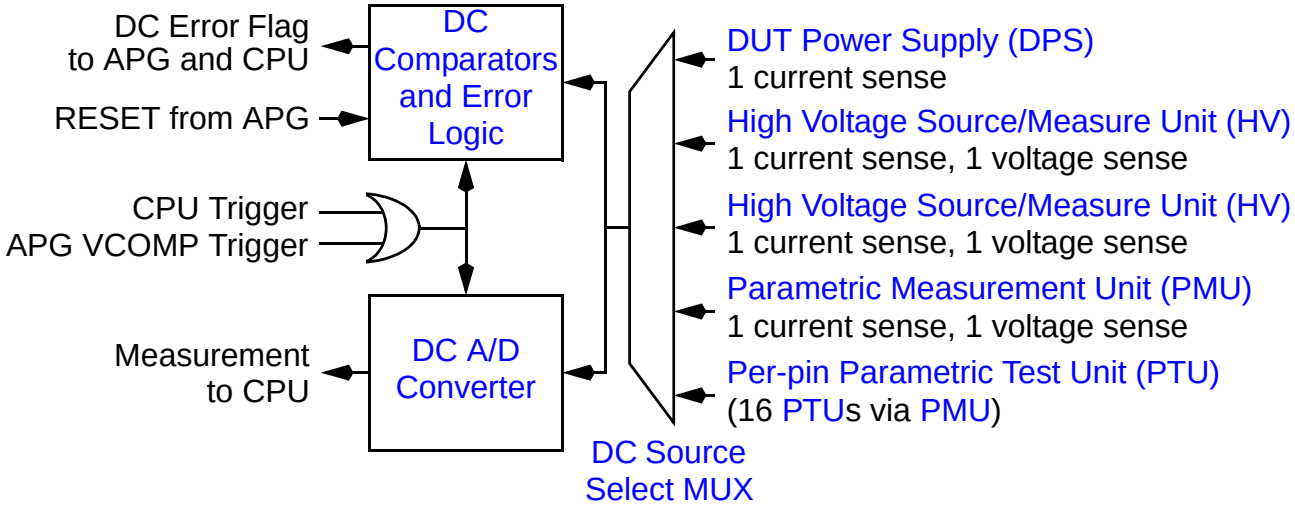
Note: when the Per-pin Parametric Test Unit (PTU) supplies the background voltage it is set to operate on the ±2uA range. The actual background voltage output can be affected by the current supplied. See PTU Operating Area.

1.6.5 DC Test and Measure System

See Site Assembly Board Block Diagram, DC Sub-System Block Diagram.

The DC test and measure system is used to test or measure voltage or current from the DUT Power Supply (DPS), High Voltage Source/Measure Unit (HV), and Parametric Measurement Unit (PMU). It is also used when measuring values from the PTU.

The diagram below shows the key elements of the DC test and measure system. There are 8 of these on each Site Assembly Board:



Also see DC Sub-System Block Diagram

Figure-14: DC Test and Measure System Block Diagram

Note the following:

- Each DC test and measure system is shared by one DUT Power Supply (DPS), two High Voltage Source/Measure Unit (HV), one Parametric Measurement Unit (PMU), and via the PMU 16 Per-pin Parametric Test Unit (PTU) (for measure only). Except for PTU Go/NoGo tests, all tests and measurements using these DC instruments is done using this subsystem.
- One input to the system is selected by the DC Source Select MUX. The MUX is automatically configured any time a DC test is executed, using DPS Static Current Test Functions, DPS Dynamic Current Test Functions, HV Static Test Functions and HV Dynamic Test Functions, PMU Static Test Functions, PMU Dynamic Test Functions or PTU Functions (measure only).
- Go/NoGo tests use the DC Comparators and Error Logic. Measurements use the DC A/D Converter. The `measure()` function switches between Go/NoGo testing and measurements.
- Static tests are triggered from the CPU. Dynamic tests can be triggered from either the CPU (Go/NoGo tests only) or from a test pattern, using the `VCOMP` pattern token.

- In Go/NoGo tests, the state of the DC Error Flag determines the final test result. When measurements are made, values are retrieved from the DC A/D Converter and analyzed to determine the final test result.
- In dynamic tests, the DC Error Flag signal allows the executing test pattern to perform branch-on-error or stop-on-error operations based on the logical OR of the DC Error Flag signal from the DC Comparators and Error Logic and the error flags from the PE Comparators and/or PTUs. The test pattern `MAR RESET` and `CHIPS RESET` instructions (Memory Test Patterns) and `VEC/RPT RESET`, `VAR RESET`, `VPINFUNC RESET` instructions (Logic Test Patterns) will reset theDC Error Flag in the DC Comparators and Error Logic.
- Regarding the PTU, the DC test and measure system is only used to make measurements (i.e. `measure()` = TRUE). In this situation, the Parametric Measurement Unit (PMU) routes one PTU at a time via the DC Source Select MUX to the DC A/D Converter.

1.6.6 DC Source Select MUX

See Magnum System Overview, DC Sub-System Block Diagram

The DC source select MUX is used, during DC parametric testing, to connect one DC instrument (at a time) to the DC Test and Measure System. The connection allows the output of the DC instrument to be tested or measured. The DC instruments include the DUT Power Supply (DPS), High Voltage Source/Measure Unit (HV), Parametric Measurement Unit (PMU), and when making a measurement (only) a Per-pin Parametric Test Unit (PTU).

The DC source select MUX is automatically configured based on the type of test being performed:

- Executing a PMU Go/NoGo test or measurement using `partest()` or `ac_partest()` will select the output of the Parametric Measurement Unit (PMU).
- Executing a PTU measurement (only) using `ptu_partest()` will select the output of the Parametric Measurement Unit (PMU) and connect one or more PTU(s) to the PMU.
- Executing a DPS current Go/NoGo test or measurement using `test_supply()` or `ac_test_supply()` will select the output of the DUT Power Supply (DPS).
- Executing a HV current Go/NoGo test or measurement using `hv_test_supply()` or `hv_ac_test_supply()` will select the output of the High Voltage Source/ Measure Unit (HV).

The operation noted above is independent of whether a Go/NoGo test is being performed or a measurement is being made. When executing a PTU test when `measure()` is TRUE, will select the output of the Parametric Measurement Unit (PMU), and connect the specified PTU to the PMU input.

1.6.6.1 DC Comparators and Error Logic

See DC Test and Measure System, DC Sub-System Block Diagram.

The DC Test and Measure System contains a dual DC comparator circuit, and associated error logic, used to perform DC parametric tests (except using PTU). Note the following:

- The DC comparators are used during Go/NoGo tests. When measurements are required the DC A/D Converter is used instead. The `measure()` function switches between Go/NoGo testing and measurements.
- Selection of which DC resource is connected to the DC comparators is made by the DC Source Select MUX, based on the type of test being performed.
- In static DC tests, the DC comparators are triggered from the CPU. In dynamic tests, the DC comparators can be enabled from the CPU (Go/NoGo tests only) or triggered from the executing test pattern, using the VCOMP pattern token.
- In dynamic tests, if test pattern triggers are disabled, the CPU enables the DC comparators for the entire duration of the executing test pattern. If, during that time, the parameter being tested fails the specified test limits, the DC error flag is set. After the test pattern stops, if the error flag remains set the test returns FAIL. The `MAR RESET` instruction (Memory Test Patterns) or `VEC/RPT RESET`, `VAR RESET`, or `VPINFUNC RESET` instruction (Logic Test Patterns) will clear the error flag.
- In dynamic tests with test pattern triggers enabled, each `MAR VCOMP` instruction (Memory Test Patterns) or `VEC/RPT VCOMP`, `VAR VCOMP`, or `VPINFUNC VCOMP` instruction (Logic Test Patterns) will trigger the DC comparators, which set the error flag if the parameter being tested fails the specified test limits at the time of the trigger. Again, after the test pattern stops, if the error flag remains set the test returns FAIL. The `MAR RESET` instruction (Memory Test Patterns) or `VEC/RPT RESET`, `VAR RESET`, or `VPINFUNC RESET` instruction (Logic Test Patterns) will clear the error flag.
- The error flag signal is routed to the APG. During dynamic tests, this allows test pattern branch-on-error or stop-on-error operations to react to this flag. The `MAR RESET` instruction (Memory Test Patterns) or `VEC/RPT RESET`, `VAR RESET`, or

VPINFUNC RESET instruction (Logic Test Patterns) do clear the error flag. Any measured values (measure() = TRUE) are only processed after pattern execution completes and thus cannot affect branch-on-error or stop-on-error (see DC A/D Converter). Note that the error flag signal is logically OR' ed with the error flag signal from the DC Comparators and Error Logic and the error flags from the PE Comparators and/or PTUs.

1.6.6.2 DC A/D Converter

See DC Test and Measure System, DC Sub-System Block Diagram.

The DC Test and Measure System contains a 16-bit A/D converter (ADC) which is used to measure and log the following:

- The output current of the DUT Power Supply (DPS). A voltage representing the DPS output current is routed to the ADC, via the DC Source Select MUX.
- The output current of any of the High Voltage Source/Measure Unit (HV). A voltage representing the HV output current is routed to the ADC, via the DC Source Select MUX.
- The output voltage of the Parametric Measurement Unit (PMU) when in force-current mode or the output current (as a voltage) when in force-voltage mode. The output of the PMU is routed to the ADC, via the DC Source Select MUX.
- The output voltage of the Per-pin Parametric Test Unit (PTU) when in force-current mode or the output current (as a voltage) when in force-voltage mode. The PTU output is routed via the Parametric Measurement Unit (PMU), which is routed to the ADC, via the DC Source Select MUX.

Note the following:

- The ADC is used to make DC measurements. When a Go/NoGo test is required the DC Comparators and Error Logic are used instead (except using PTU). The measure() function switches between Go/NoGo testing and measurements.
- The ADC only measures voltage. Those DC resources which can test or measure current output a voltage representing the current to be measured.
- Selection of which DC resource is connected to the ADC is made by the DC Source Select MUX, based on the type of test being performed.
- In static tests, the ADC is triggered from the CPU. In dynamic tests, the ADC can only be triggered from the executing test pattern, as specified using the VCOMP pattern token.

- The PASS/FAIL result of a dynamic measurement is determined after the test pattern execution ends. The CPU retrieves the measured value and compares it to the test limits to determine PASS/FAIL. Note that the test will also fail if any functional strobes fail in the test pattern.
- User code can retrieve measured values. See Retrieving DC Test Results.
- ADC measurements cannot affect test pattern branch-on-error or stop-on-error operation.

1.7 Pattern and Timing System

1.7.1 Overview

See Magnum System Overview, Site Assembly Board.

The diagram below shows the architecture of the test pattern and timing system of one Site Assembly Board. The pin electronics (PE) are simplified to show just the driver/comparator of each of four pins of the total 128 test channels on the Site Assembly Board:

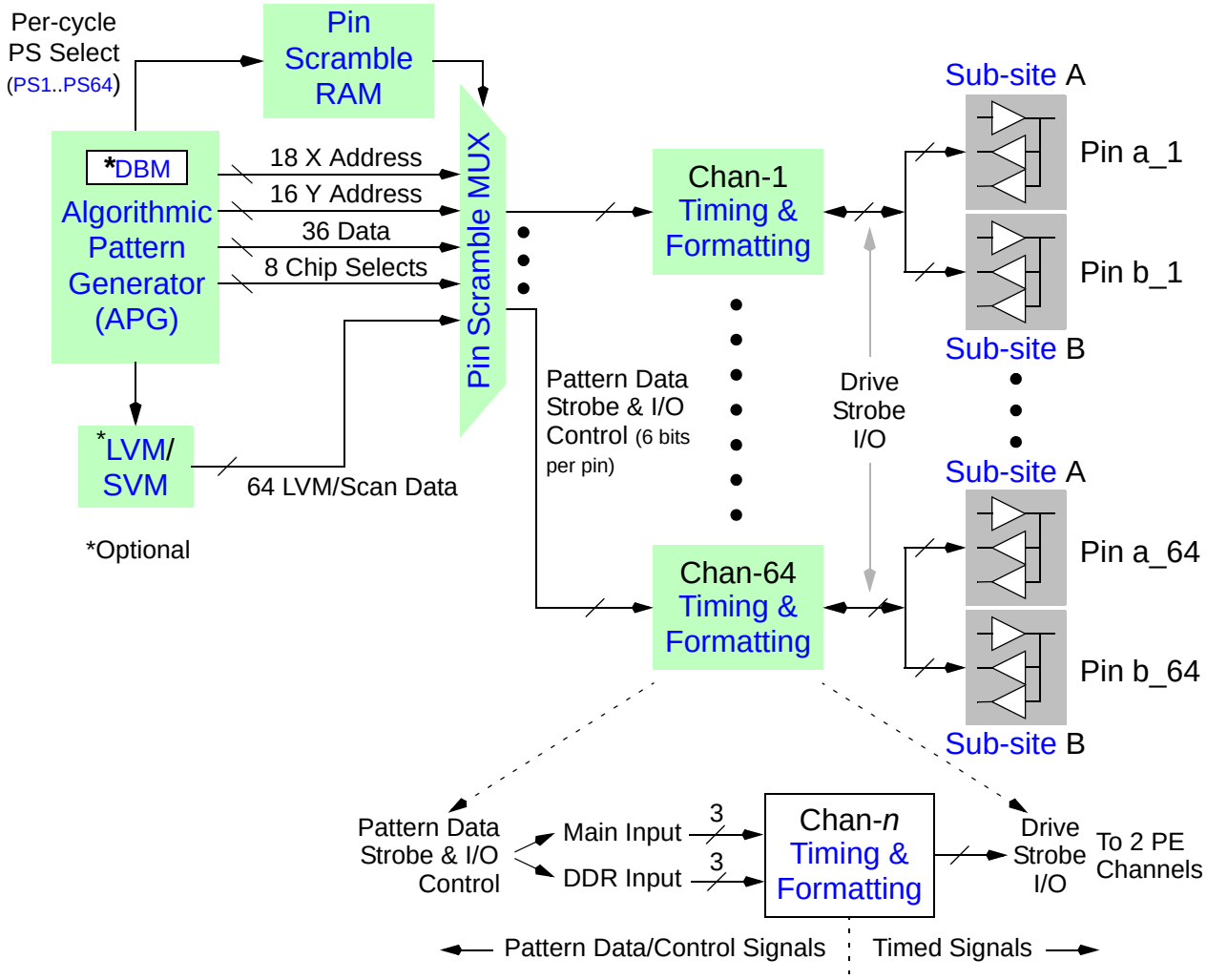


Figure-15: Magnum Pattern & Timing System

A key architectural feature of the Site Assembly Board is the combination of the Pin Scramble MUX driving the 64 channel Timing & Formatting logic. During test pattern execution, the APG, via the Pin Scramble RAM, causes the Pin Scramble MUX to select a unique source of test pattern signals, per cycle, as input to the Timing & Formatting logic of each channel. This allows an arbitrary pattern data source to control, per-cycle, any given pair of pins.

Each pin-pair (see Functional Pin-pairs) has independent Timing & Formatting logic. This is where drive waveforms are formatted and exact edge times are controlled. Similarly, strobes and I/O signals are timed and controlled here, per-pin pair, per-cycle. See Timing & Formatting for more details.

1.7.2 Pin Scramble MUX

See Site Assembly Board, Pattern and Timing System.

The Pin Scramble MUX (see Pattern and Timing System) is used, during functional test pattern execution, to select the source of test pattern data, strobe, and I/O control signals for each tester pin-pair, on-the-fly.

For instance, testing a DUT with a complex I/O interface like that found on a NAND FLASH device (with an instruction word, data, and three layers of address all multiplexed onto an eight bit bus) is simple to test using the pin scrambler. One pattern cycle scrambles the instruction to the 8-pin bus, the next cycle scrambles the first part of the address to these same pins, the next cycle scrambles the second part of the address to these same pins, etc.

The diagram below shows key details about the Magnum pin scramble architecture:

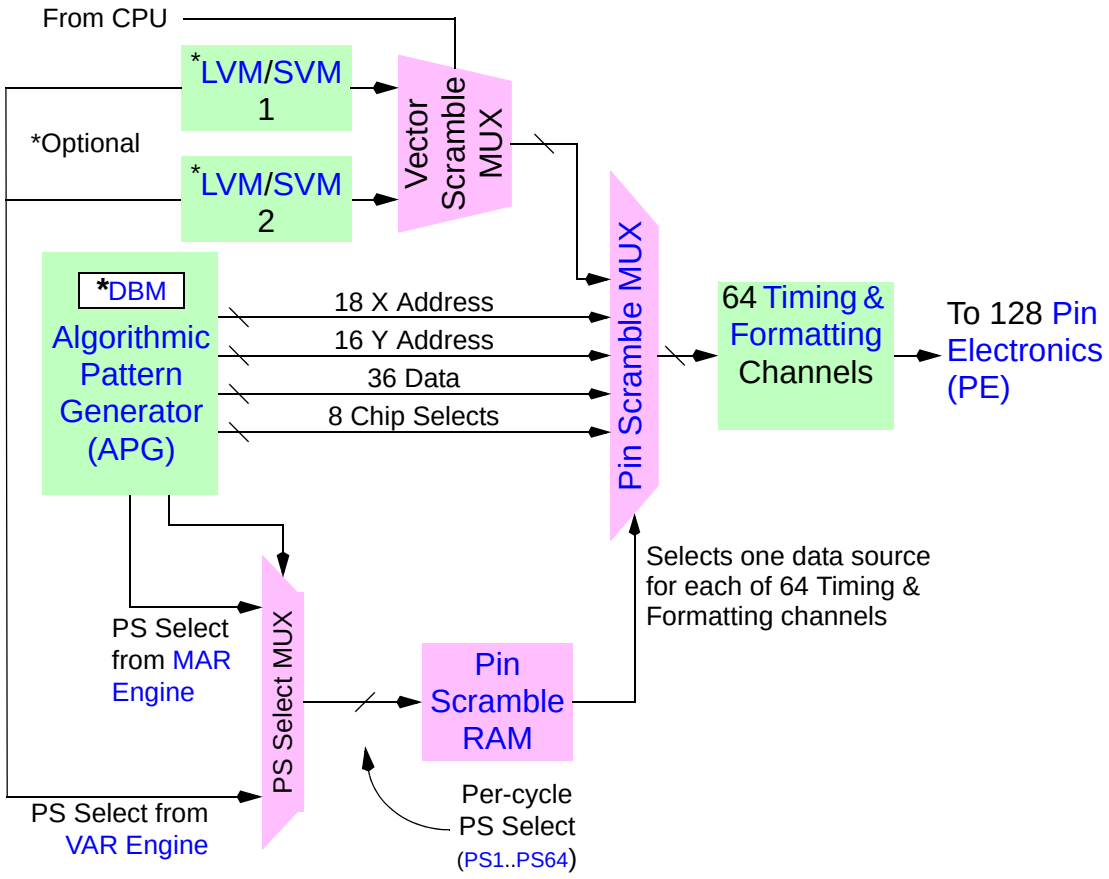


Figure-16: Pin Scramble Block Diagram

Note the following:

- The inputs to the Pin Scramble MUX consist of the various test pattern data sources including:
  - 18 X APG Address Generator outputs (X0 through X17, see note below)
  - 16 Y APG Address Generator outputs (Y0 through Y15, see note below)
  - 36 APG Data Generator outputs
  - 8 APG Chip Selects
  - 1 or 2 Logic Vector Memory (LVM) / Scan Vector Memory (SVM) data sources for each pin-pair (see Functional Pin-pairs). The Vector Scramble MUX allows a given LVM/SVM bit to be mapped any given pin-pair(s), allowing a test pattern for a single DUT to be use to test multiple DUTs without modification. The Magnum

LVM stores both logic vectors and scan vectors. The LVM architecture provides separate memories; LVM-1 for normal logic vector and scan operations, LVM-2 stores Double Data Rate (DDR) Mode B-cycle pattern data.

Note: the APG Address Generator can output up to 18 X addresses, but the combined X + Y address is limited to 36 bits. Therefore if X16 is used, Y15 can't be, and if X17 is used Y14 can't be.

- The Pin Scramble RAM contents are initialized using the Pin Scramble Macros, during the initial program load. This is where the user's software identifies a test pattern data source for each pair of DUT pins. Up to 64 sets can be defined (PS1 through PS64). Those which are not explicitly defined retain the Default Pin Scramble Map definitions.
- During functional test execution, in each pattern cycle, the APG outputs a pin scramble selection, to select one of the 64 pin scramble maps (PS1 to PS64). This equates to the PS# parameter specified using the PINFUNC PS# (Memory Test Patterns) or VEC/RPT PS# and VPINFUNC PS# instructions (Logic Test Patterns). This is the address input to the Pin Scramble RAM. Thus, in each pattern cycle, the Pin Scramble RAM outputs cause the Pin Scramble MUX to select a specific pattern data source for each pair of tester pins.
- The output of the Pin Scramble MUX is 64 sets of pattern data (PEL), strobe control (PES) and I/O control (PEE). Each set is sent to one of the 64 Timing & Formatting channels where real-time timing is added, drive format is determined, etc. See Timing & Formatting.
- The Timing & Formatting section outputs 64 sets of time-calibrated strobe and I/O control signals and 64 time-calibrated and formatted drive signals. Each set controls 2 tester pins (see Functional Pin-pairs).

1.7.3 Pin Scramble RAM

See Site Assembly Board, Pattern and Timing System.

It is the Pin Scramble RAM which translates the pin scramble map selection (PS1 through PS64), output by the APG in each pattern cycle, into the signals which cause the Pin Scramble MUX to select a specific pattern data source for each pair of DUT signal pins.

The Pin Scramble RAM contents are initialized using the Pin Scramble Macros, during the initial program load. This is where the user's software identifies a test pattern data source for each pair of DUT signal pins (see Functional Pin-pairs). Up to 64 sets (PS1 through PS64)

can be defined. Those which are not explicitly defined retain the default Default Pin Scramble Map definitions.

The Magnum APG has two pattern control engines (MAR Engine and VAR Engine) either of which can select the pin scramble map, per cycle. The pattern instructions used are PINFUNC PS# (Memory Test Patterns) or VEC/RPT PS# and VPINFUNC PS# instructions (Logic Test Patterns). In Mixed Memory/Logic Patterns, an additional instruction is required (PINFUNC VPS) to cause the VAR Engine pin scramble selection to take precedence over the MAR Engine pin scramble selection. See Pin Scramble MUX.

The Pin Scramble Block Diagram includes key details about how the pin scramble selection is made. Note the following:

- The PS Select MUX determines whether the APG MAR Engine or VAR Engine will, cycle-by-cycle, make the pin scramble selection:
  - In Memory Test Patterns, the selection is always made by the MAR Engine, using the test pattern PINFUNC PS# instruction.
  - In pure Logic Test Patterns (including Scan Test Patterns), the selection is always made by the VAR Engine, using the test pattern VEC/RPT PS#, or VPINFUNC PS# instructions.
  - In Mixed Memory/Logic Patterns, by default, the MAR Engine pin scrmable selection is used. The VAR Engine selection can be enabled, per cycle, using the PINFUNC VPS instruction.
- The Vector Scramble MUX is able to map any Logic Vector Memory (LVM)/Scan Vector Memory (SVM) data source to any combination of pin-pairs. This means, for example, that the LVM/SVM data typically mapped to a_1/b_1, can also be mapped to other pin-pairs. This supports logic pattern parallel applications, where multiple DUTs will receive identical LVM/SVM outputs, without having to write special Logic Test Patterns or duplicate pattern data into multiple LVM/SVM locations (pins). The Vector Scramble MUX is configured statically, prior to executing a test pattern, based on the Pattern Attributes.

1.7.4 Timing & Formatting

See Site Assembly Board, Pattern and Timing System, Timing and Formatting Functions.

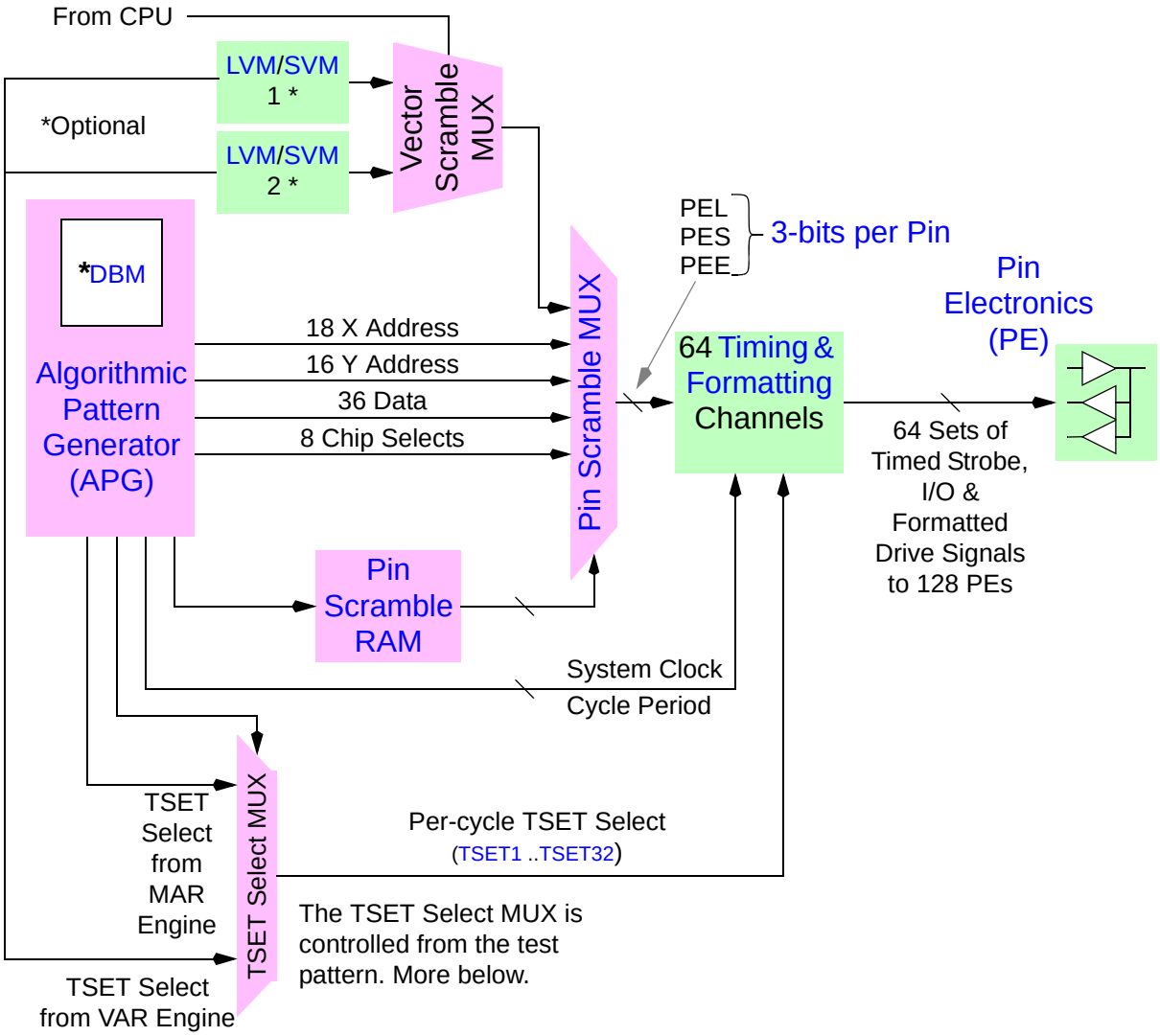
The Magnum Timing and Formatting logic provides the following AC test capabilities:

Table 1.7.4.0-1 Timing Specifications

Parameter	Range	Resolution
Time-sets	32	
Cycle Period	20nS to 10.2uS	1nS
On-the-fly Cycle Period Resolution	1ns	
Max. Data Rate	50MHz 100MHz DDR[1] 133MHz DDR MUX	1nS
Edge Placement Resolution	100pS Static 1nS On-the-fly Edges range to end of 2nd cycle	
Minimum Window Strobe Width	5nS	
Minimum Drive Pulse-width	4nS	
Timing Edges Per-Cycle	2 Drive 2 Strobe 2 I/O	
Mux Mode (lose one pin for every mux'ed pin)	Yes	

Notes:
1) Magnum Double Data Rate (DDR) Mode, no loss of pins ala MUX.

Each Magnum Site Assembly Board contains 64 timing and formatting channels. Each timing channel drives two pin electronics (PE) channels. The diagram below shows key details about the this architecture:



Note the following:

- The TSET Select MUX is determines whether the MAR Engine or VAR Engine will, cycle-by-cycle, determine the time-set selection:
  - In Memory Test Patterns, the selection is always made by the MAR Engine, using the test pattern `PINFUNC TSET#` instruction.

- In Logic Test Patterns (including Scan Test Patterns), the selection is always made by the VAR Engine, using the test pattern `VEC/RPT TS#`, or `VPINFUNC TSET#` instructions.
- In Mixed Memory/Logic Patterns, by default, the MAR Engine time-set selection is used. The VAR Engine selection can be enabled, per cycle, using the `PINFUNC VTSET` instruction.

The following diagram shows the 4 timing generators used to generate the timed drive, strobe and I/O edges used to stimulate and test the DUT during functional tests:

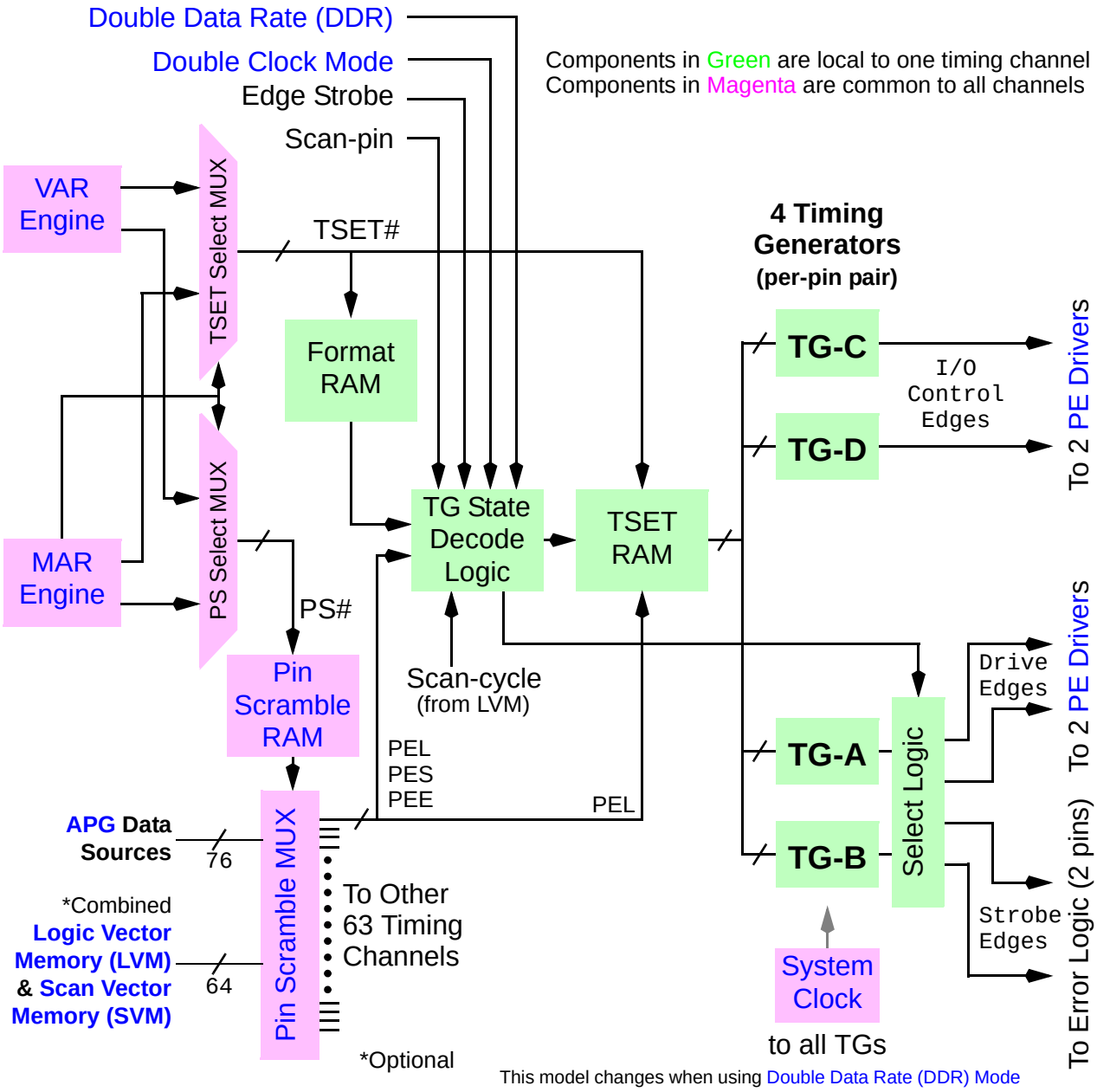


Figure-17: Timing Generator Block Diagram

Note the following:

- Each Site Assembly Board has 64 timing channel's, each with four timing generators (TG-A, TG-B, TG-C and TG-D).
- During test pattern execution, each timing channel's four timing generators generate the timed and formatted drive, strobe and I/O control signals sent to two Pin Electronics (PE) channels (see Functional Pin-pairs).
- Two timing generators (TG-C and TG-D) are used to control I/O switching i.e. at what time in the current pattern cycle the PE driver will drive and/or tri-state.
- Two timing generators (TG-A and TG-B) are used to generate either zero, one or two drive edges or zero or two strobe edges, per cycle.
- Each of the 4 TGs are independently controlled by their associated TSET RAM, which stores several edge-time and control bit combinations for each TG, for each of 32 time-sets. In each pattern cycle, the combined control bit plus edge time value determines whether a given TG issues an edge, or not, and at what time the edge is generated. For example, to produce a window strobe or RTZ/RTO drive format both TG-A and TG-B will issue an edge at a specified time. To produce an NRZ format only TG-A will issue an edge. Edge strobes generate two edges even though only the first edge is actually used as a strobe (see `edge_strobe()`).
- It is the combination of the TG State Decode Logic, Format RAM, test pattern data from the Pin Scramble MUX, and time-set selection, plus several static signals (more below) which determine how the TSET RAM is addressed and thus which edges are generated and at what time.
- During test pattern execution, in each tester cycle, the Pin Scramble MUX selects a pattern data source for each timing channel (each pin-pair). This provides 3-bits per-channel used to control drive, strobe and I/O state.
- During test pattern execution, in each tester cycle, one time-set is selected. This determines the global cycle period, and the drive, strobe and I/O timing and drive format for each timing channel (each pin-pair).
- The following signal, input to the TG State Decode Logic, is global (not per-pin) and static i.e. it does not change during test pattern execution.
  - The Double Data Rate (DDR) Mode signal configures the TG State Decode Logic to operate in DDR mode. This signal is set by the system software when preparing to execute a given test pattern based on the pattern's rate attribute (see Pattern Rate Attributes). This signal is global (not per-pin) and static i.e. it does not change during test pattern execution.
- The following signals are per-pin and static i.e. they don't change dynamically (per-cycle):

- Scan-pin: identifies those pin(s) which will react to the Scan-cycle signal. In scan cycles, those pins which are *not* scan pins will *hold*; i.e. they will use the pattern data and generate the format from the most recent non-scan cycle. The Scan-pin signal is set by the system software based on which pin(s) of the test pattern about be executed were identified in the test pattern using the SCANDEF pattern directive (see Scan Test Patterns).
- Double Clock Mode: configures the TG State Decode Logic to operate in double clock mode. These pin(s) will not tri-state or strobe, regardless of test pattern format selection. This signal is set when the most recent execution of settime(), on a given pin, specifies the DCLKPOS or DCLKNEG. Double Clock Mode also requires setting Dclk Mode using pe_driver_mode_set().
- Edge Strobe: determines whether the TG State Decode Logic generates an edge strobe or window strobe. This signal is controlled using the edge_strobe() function.

- The following signals are global (not per-pin) and change dynamically (per-cycle):
  - TSET#: selects one of 32 time-sets
  - PS#: selects one of 64 Pin Scramble Maps
- The following signals are per-pin and change dynamically (per-cycle):
  - Scan-cycle: causes pin(s) which are not Scan-pin(s) (see above) to hold; i.e. they will use the pattern data and generate the format from the most recently executed non-scan cycle. Scan-pin(s) will be controlled using the scan pattern data from the Pin Scramble MUX.

- In non-DDR mode, a Magnum timing channel can generate five drive formats and 2 strobe formats:

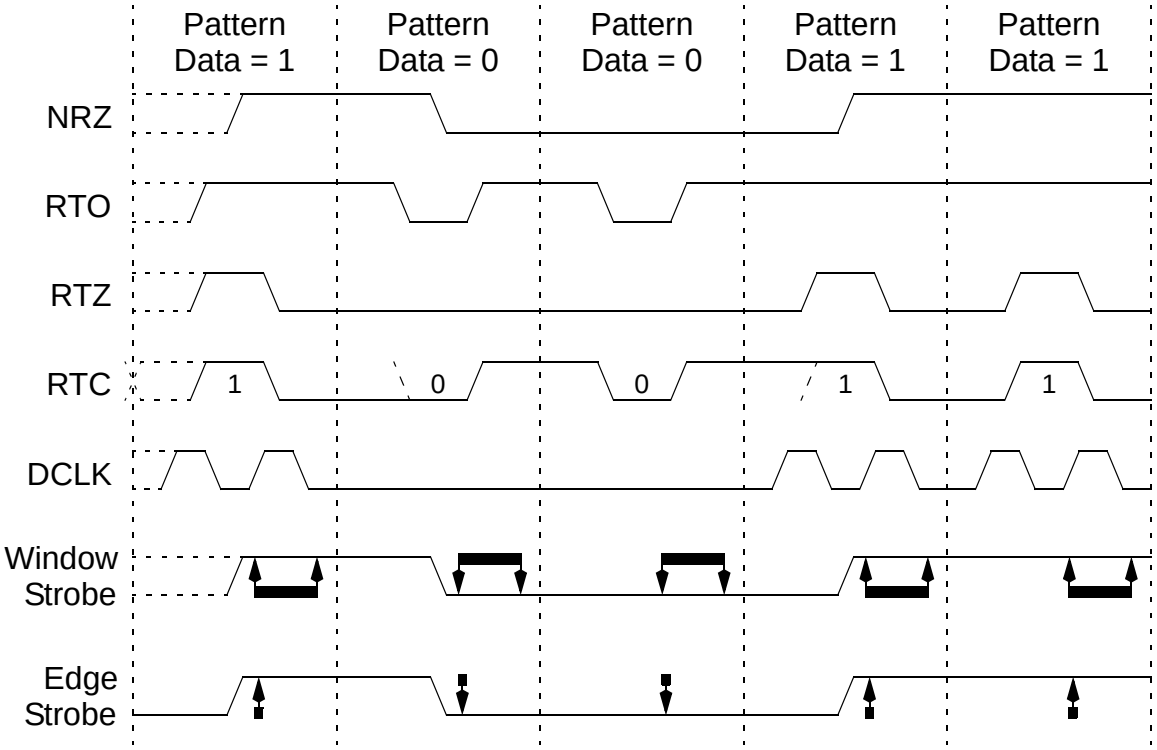


Figure-18: Non-DDR Waveform Format Options

- In DDR mode, the NRZ, DCLK and Edge Strobe formats are available:

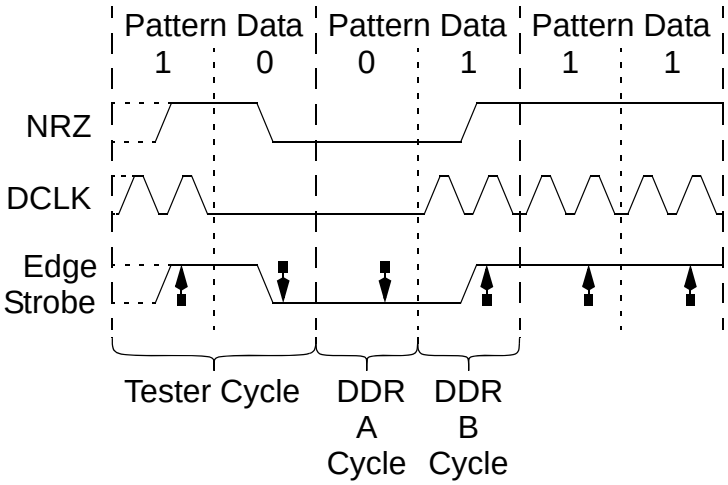
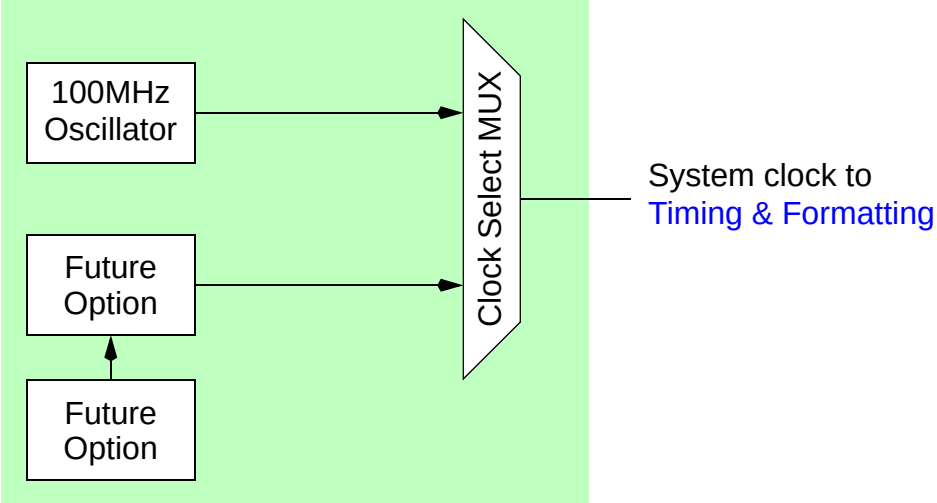


Figure-19: DDR Waveform Format Options

1.7.5 System Clock

See Site Assembly Board, Pattern and Timing System.

The following diagram shows the system clock sub-system:



Two system clock sources are available:

- A fixed 100MHz oscillator, selected when the most recently programmed cycle period value is an even multiple of 1nS.
- TBD - a possible future option.

Note: in use, the system clock source frequency is effectively multiplied by 10. Thus, using the fixed 100MHz Oscillator source (a 10nS clock) all cycle period values and (digital) edge timing values are derived by counting a 1nS clock. For documentation purposes, this effective clock source is called the X-clock (10x clock source). See Magnum Timing Rules.

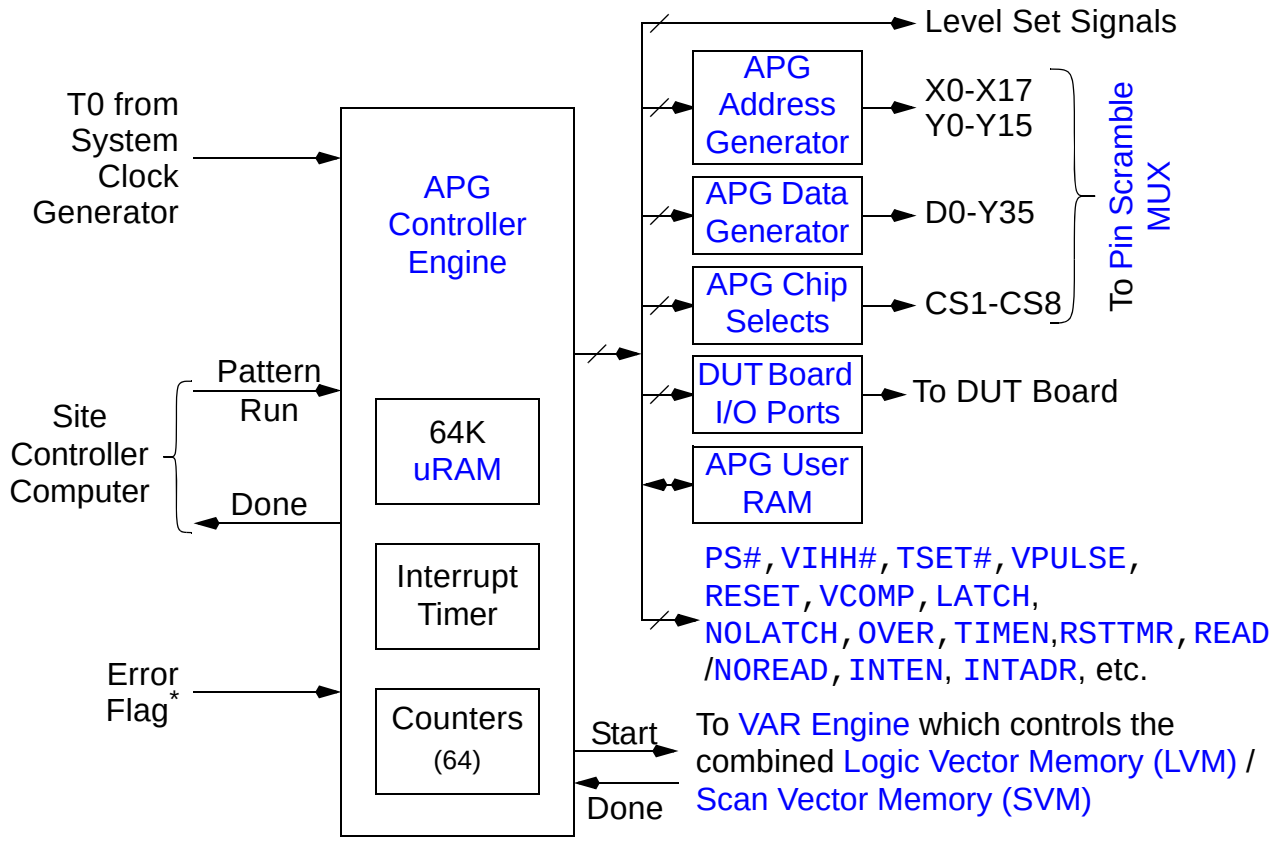
1.8 Algorithmic Pattern Generator (APG)

See Magnum System Overview.

The APG on each Site Assembly Board performs the following functions:

- Controls all functional test pattern execution (start/stop, subroutines, branch-on-error, etc.).
- Algorithmically generates address, data, and chip select test patterns used to functionally test memory devices:
  - APG Address Generator: 18 X and 16 Y address outputs (36 bits total)
  - APG Data Generator: 36 data outputs
  - APG Chip Selects: 8 independently controllable outputs, 2 can do I/O and strobe
- Generates a start signal to the VAR Engine, which controls Logic Vector Address (VAR) used to sequence logic and scan test patterns from the combined Logic Vector Memory (LVM) / Scan Vector Memory (SVM).
- Delivers per-cycle time-set selection (TSET#) to the Timing & Formatting logic.
- Delivers per-cycle pin scramble map selection (TSET#) to the Pin Scramble MUX.
- Delivers per-cycle VIHH Map selection (VIHH#) to the Pin Electronics (PE).
- Generates and controls various other miscellaneous signals used to:
  - Switch the DUT Power Supply (DPS) to the secondary voltage (VPULSE).
  - RESET the Error Flags in the DC Comparators and Error Logic and PE Comparators.
  - Trigger the DC Comparators and Error Logic or DC A/D Converter during dynamic DC tests (VCOMP).
  - Control over-programming logic used when programming some device types. See Over-programming Controls and Parallel Test.
  - Etc.

The following diagram represents the Site Assembly Board APG:



*Logical OR of DC Error Flag signals from the DC Comparators and Error Logic and all error flags from the PE Comparators. Controls branch-on-error operations.

Figure-20: APG Block Diagram

Regarding the diagram above, note the following:

- All of the logic shown resides on the Site Assembly Board.

Test pattern execution operates as follows:

- During the initial program load, test pattern(s) are loaded automatically.
  - Memory Test Patterns are loaded into the appropriate APG Controller Engine.
  - Logic Test Patterns and Scan Test Patterns are loaded into the combined Logic Vector Memory (LVM) / Scan Vector Memory (SVM).

  The system software has a record of the location of each pattern, by name.

- Prior to pattern execution, most of the APG's components may be optionally initialized from the test program, using functions documented in Memory-pattern Related Functions and/or Logic Pattern Related Functions. This includes setting initial counter values, initial values for X/Y APG Address Generators, APG Data Generator, APG counters, etc. It is also possible to initialize these components using the same functions in the Pattern Initial Conditions section of the test pattern.
- A test pattern is executed using `funtest()`, `start_pattern()`, `ac_partest()`, or `ac_test_supply()`. Various pattern execution stop options are possible. To simplify things, the rest of this section will only refer to `funtest()`.
- To execute the test pattern, the site controller computer sends the *pattern run* signal to the APG Controller Engine. Once the *pattern run* signal is received, the APG Controller Engine is in complete control of the APG and its various components. This includes controlling the Start signal sent to the VAR Engine which controls the combined Logic Vector Memory (LVM) / Scan Vector Memory (SVM).
- As each pattern instruction executes, the APG Controller Engine outputs control bits to each of the various APG components. These determine, for example, what the X/Y APG Address Generator will do in the current instruction, what the APG Data Generator will do in the current instruction, which time-set is selected, etc.
- The site controller computer then waits for a *done* signal. The APG will issue a done signal under the following conditions:
  - The last pattern instruction executes; i.e. `MAR DONE` instruction (Memory Test Patterns) or `VAR DONE` (Logic Test Patterns) executes.
  - A `MAR PAUSE` instruction (Memory Test Patterns) or `VAR PAUSE` (Logic Test Patterns) executes.
  - If the pattern execution stop condition = `error` (see `funtest()`) and an Error Flag is set. This can be any error flags from the PE Comparators and/or any of the of DC Error Flags from the DC Comparators and Error Logic on each Site Assembly Board.
  - The user interrupts execution using UI's Stop Testing button.

1.8.1 APG Controller Engine

See Algorithmic Pattern Generator (APG), MAR Engine, VAR Engine.

As noted above, the during pattern execution APG operation is controlled, at pattern data rates, by its local APG Controller Engine, which actually consists of two state machines:

- The MAR Engine is involved in all test pattern executions, whether executing Memory Test Patterns only, Logic Test Patterns only (which includes Scan Test Patterns), or Mixed Memory/Logic Patterns.
- The VAR Engine is only active when executing a test pattern which contains logic and scan instructions.

The term MAR has evolved, originally referring to the APG's MicroRAM Address Register. In this documentation, the term MAR is also used:

- When referring to the hardware engine which controls all test pattern execution; i.e. MAR Engine.
- To represent the pattern instruction which controls the MAR Engine; i.e. the `MAR` instruction.
- To represent the current value in the APG MAR register. This is the address of the pattern instruction being executed. Note that user code doesn't directly interact with literal MAR register values; all references to specific pattern instructions is done using Pattern Labels or the pattern's name.

The term VAR refers to Vector Address Register, which is consistent with the use of MAR. In this documentation, the term VAR is also used:

- When referring to the hardware engine which controls logic pattern instruction execution; i.e. VAR Engine.
- To represent the pattern instruction which controls logic pattern execution sequence; i.e. `VAR` instruction. Note that most pure Logic Test Patterns will not contain an explicit `VAR` instruction because the `VEC`/`RPT` instructions implicitly control the VAR Engine.
- The current value in the APG's VAR register. This is the address of the logic pattern instruction being executed. User code doesn't directly interact with literal VAR register values; all references to specific pattern instructions is done using Pattern Labels or the pattern's name.

Three pattern execution scenarios are possible, based on the type of instructions in the test pattern:

- Pattern contains memory instructions only. Only the MAR Engine is used, the VAR Engine is effectively disabled. Normal uRAM consumption by memory instructions.
- Pattern contains logic and scan instructions only. Only the VAR Engine is used, the MAR Engine is effectively disabled.

- Pattern contains a mix of logic and memory instructions. Both the MAR Engine and VAR Engine are used. If the test pattern does not contain any memory instructions, the MAR Engine is still used, but executes a built-in MAR instruction which loops on itself indefinitely.

Note: Maverick-I logic pattern operation is not supported using Magnum i.e. any test pattern using a logic instruction enables the VAR Engine and logic instructions do not impact uRAM consumption.

The following diagram shows the two engines and how they interact.

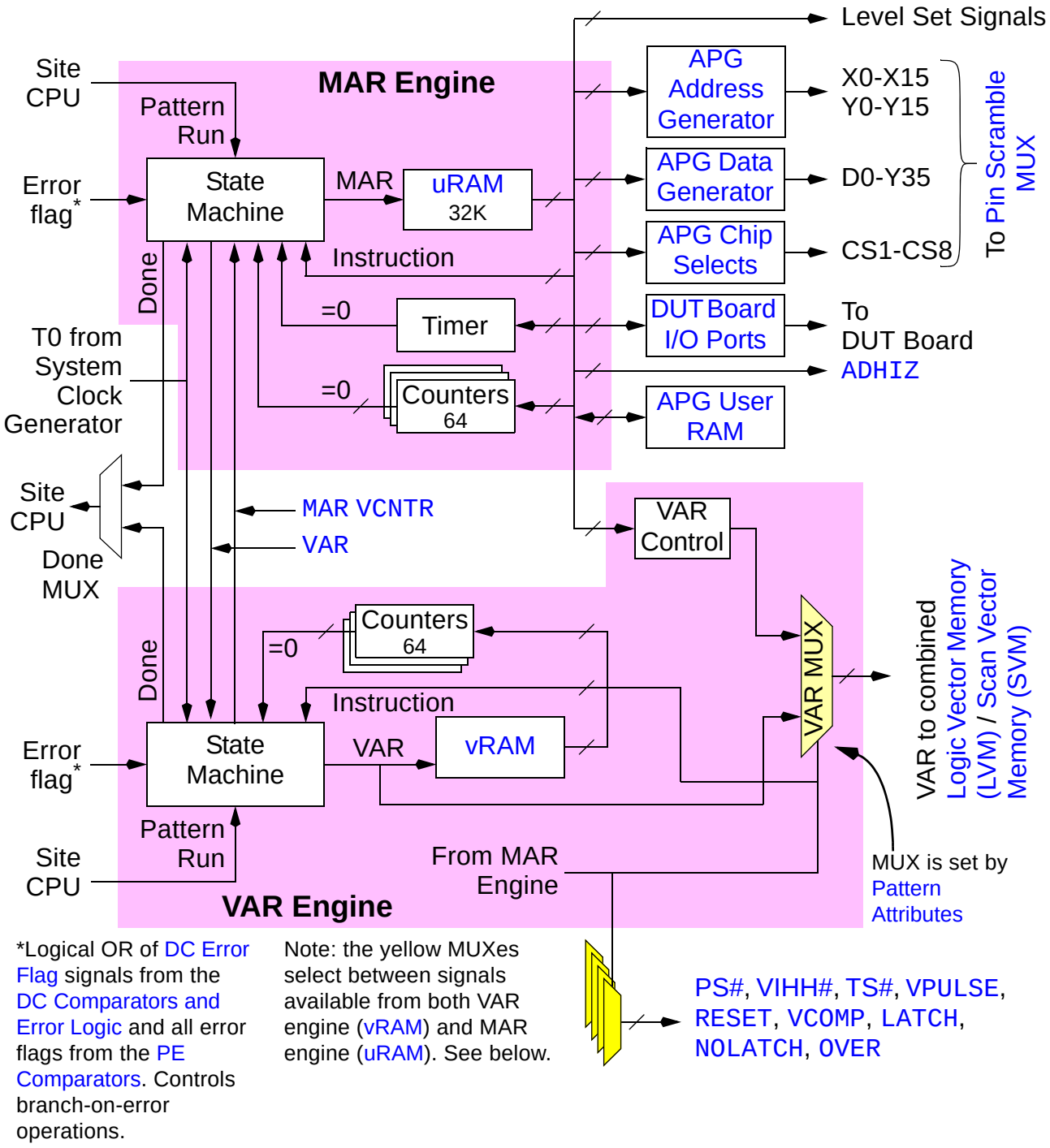


Figure-21: MAR/VAR Engine Block Diagram

Pattern execution using the model above is discussed below. But first, note the following:

- All of the logic shown in the diagram above resides in the APG.
- In the diagram above, the *pattern run* signal is shown in two places (to simplify the diagram). However, there is only one signal.
- In the diagram above, the *error flags* signal is shown in two places (to simplify the diagram). However, there is only one signal.
- There are two separate *Done* signals, one from the MAR Engine state machine, one from the VAR Engine state machine. The Done MUX selects one of these signals to be sent to the site controller computer . Operation is described in MAR DONE and/or VAR DONE.
- During the initial program load, memory test pattern(s) are loaded into the 64K APG uRAM. The system software has a record of the location of each pattern, which is used to execute that pattern. Magnum does not support executing Logic Test Patterns in Maverick-I compatibility mode; i.e. the VAR Engine vRAM are always used and uRAM never stores logic vector information (VAR, etc.)
- If the pattern contains logic vectors, these are loaded into the Logic Vector Memory (LVM). Again, the system software has a record of the location of the pattern. Logic pattern instructions (VAR instructions) are loaded into the VAR Engine vRAM.
- If the pattern contains scan vectors, these are loaded into the combined Logic Vector Memory (LVM) / Scan Vector Memory (SVM).
- When executing a test pattern that does not contain any logic instructions only the MAR Engine is used; i.e. the VAR Engine is disabled.
- When executing a test pattern which does contain logic instructions the VAR MUX selects the VAR Engine's state machine as the source of the logic vector address (VAR) output to the Logic Vector Memory (LVM). This is the same address seen at the input to the vRAM.
- Two counter related signals connect the MAR Engine's state machine and the VAR Engine's state machine. One allows the VAR Engine's state machine to see the counter = 0 signal from the MAR Engine's state machine, the other allows the MAR Engine's state machine to see the counter = 0 signal from the VAR Engine's state machine. The `VAR MCNTR` and `MAR VCNTR` instructions must be explicitly used to enable these features.
- When used, the VAR Engine is also able to select many of the per-cycle parameters previously only selectable by the MAR Engine (see below). In the diagram above, the hardware MUXs used to make these selections are shown at the bottom, in yellow. Each MUX has input(s) from both the MAR Engine's uRAM and the VAR Engine's vRAM. In a pure logic pattern (i.e. `logic` Pattern System

Attributes) these MUX are set to select inputs from the VAR Engine's vRAM. In pure Memory Test Patterns the MUXs are set to select inputs from the MAR Engine's uRAM. In Mixed Memory/Logic Patterns, default operation selects the MAR Engine inputs and an explicit pattern instruction is needed, for each parameter, to select the VAR Engine inputs, as follows:

- `PINFUNC VPS` selects `PS#` from the VAR Engine's vRAM.
- `PINFUNC VVIHH` selects `VIHH#` from the VAR Engine's vRAM.
- `PINFUNC VTSET` selects `TS#` from the VAR Engine's vRAM.
- `PINFUNC VVPULSE` selects `VPULSE` from the VAR Engine's vRAM.
- `PINFUNC VLATCHRESET` selects `RESET` from the VAR Engine's vRAM.
- `PINFUNC VVCOMP` selects `VCOMP` from the VAR Engine's vRAM.
- `PINFUNC VLATCHRESET` selects `LATCH`/`NOLATCH` from the VAR Engine's vRAM.
- `PINFUNC VOVER` selects `OVER` from the VAR Engine's vRAM.

Pattern execution operates as follows:

- During the initial program load, test patterns s are loaded into the appropriate MAR Engine uRAM and/or VAR Engine vRAM. Memory instructions go into the uRAM and logic instructions into the vRAM. The system software has a record of the location of each pattern, by name.
- At the same time, logic patterns and scan patterns are loaded into the combined Logic Vector Memory (LVM) / Scan Vector Memory (SVM).
- Prior to executing a test pattern, most of the APGs components may be optionally initialized from user C code, using functions documented in Memory-pattern Related Functions and/or Logic Pattern Related Functions. This includes setting initial counter values, initial values for APG Address Generator, APG Data Generator, etc. It is also possible to initialize these components using the same C functions in the Pattern Initial Conditions section of the test pattern. User code does not directly interact with the MAR Engine/uRAM or VAR Engine/vRAM.
- A test pattern is executed using `funtest()`, `start_pattern()`, `ac_partest()`, or `ac_test_supply()`. Various pattern execution stop options are possible. To simplify things, the rest of this discussion will only refer to `funtest()`.
- As part of the `funtest()` execution, the system software gets the APG uRAM address of the first instruction of the test pattern, and loads it into the MAR Engine state machine.

Note: all test patterns have an associated MAR pattern. In the case of pure Logic Test Patterns (i.e. `logic` Pattern System Attributes) a built-in memory pattern is used, containing a single MAR instruction (see Default Memory Pattern Instruction), which forever loops on itself.

- If the test pattern contains logic instructions, the system software gets the APG vRAM address (VAR) of the first instruction of the test pattern, and loads it into the VAR Engine's state machine. If not, the VAR Engine's is disabled and the VAR is arbitrary.
- Other housekeeping operations are performed to properly configure the various APG components prior to actually starting the test pattern.
- To execute the test pattern, the site controller computer sends the *pattern run* signal to both the MAR Engine state machine and VAR Engine state machine. Once the *pattern run* signal is received, the two state machines are in complete control of the APG, and its various components. This includes controlling the VAR sent to the Logic Vector Memory (LVM) and the SAR to the Scan Vector Memory (SVM).
- The site controller computer then waits for a *done* signal, or for the user to interrupt execution using UI's Stop Testing button. Or, if the `funtest()` was executed with the *error* option (i.e. stop on error) the APG will stop if it receives an error signal from any of the per-pin PE Comparators or from the DC Comparators and Error Logic (the DC Error Flag) on any Site Assembly Board.
- The *done* signal from the VAR Engine's state machine is only selected when executing a pure logic pattern (i.e. `logic` Pattern System Attributes) otherwise the done signal from the MAR Engine state machine is selected. See MAR DONE and/or VAR DONE.
- As each pattern instruction executes, the uRAM outputs control bits to each of the various APG components. These determine, for example, APG Address Generator operation in the current instruction, the APG Data Generator in the current instruction, which time-set is selected, etc. The uRAM has lots of output bits.
- As each pattern instruction executes, the uRAM also sends the `MAR` instruction from that instruction back to the state machine, to specify how the next instruction to be executed is determined. This can be a simple `MAR INC`, an unconditional branch (`JUMP`, `RETURN`, etc.), a subroutine call (`GOSUB`), a conditional branch (`CJMPNZ`, etc.), or a timer interrupt (see APG Interrupt Timer). It can also be `MAR DONE` or `MAR PAUSE`, which stops the APG and sends the *done* signal to the site controller computer .

- The state machine sends the address (MAR) of the next instruction to be executed to the uRAM, which then outputs the control bits for that instruction to the APG hardware, etc. The process repeats until either a MAR DONE or MAR PAUSE is detected or the user clicks UI's Stop Testing button. Or, if the `funtest()` was executed with the *error* option (i.e. stop on error) the APG will stop if it receives an error latch signal from any of the per-pin PE Comparators or from the DC Comparators and Error Logic (the DC Error Flag) on any Site Assembly Boards.
- If the test pattern contains logic or scan instructions the vRAM sends the VAR instruction at the current address to the VAR Engine's state machine, and to the combined Logic Vector Memory (LVM) / Scan Vector Memory (SVM) (more below). This specifies how the state machine is to determine which instruction will to be executed next. This can be a simple VAR INC, an unconditional branch (JUMP, RETURN, etc.), a subroutine call (GOSUB), a conditional branch (CJMPNZ, etc.). It can also be VAR DONE, which, in pure Logic Test Patterns (only), stops the APG and sends the done signal to the site computer.
- The two state machines continue to execute instructions from their associated RAMs until a DONE instruction is executed, or the user clicks UI's Stop Testing button. In Memory Test Patterns or Mixed Memory/Logic Patterns the MAR DONE instruction will stop the APG. In pure Logic Test Patterns the VAR DONE instruction will stop the APG. See MAR DONE and/or VAR DONE. The state machines will also stop if a PAUSE instruction is executed. This can be MAR PAUSE (Memory Test Patterns) or VAR PAUSE (Logic Test Patterns). Or, if the `funtest()` was executed with the *error* option (i.e. stop on error) both state machine's will stop if an error signal is received from any of the per-pin PE Comparators or from the DC Comparators and Error Logic (the DC Error Flag) on any Site Assembly Board.

1.8.1.1 uRAM

See Algorithmic Pattern Generator (APG), MAR Engine

The following diagram shows the Site Assembly Board's APG uRAM architecture:

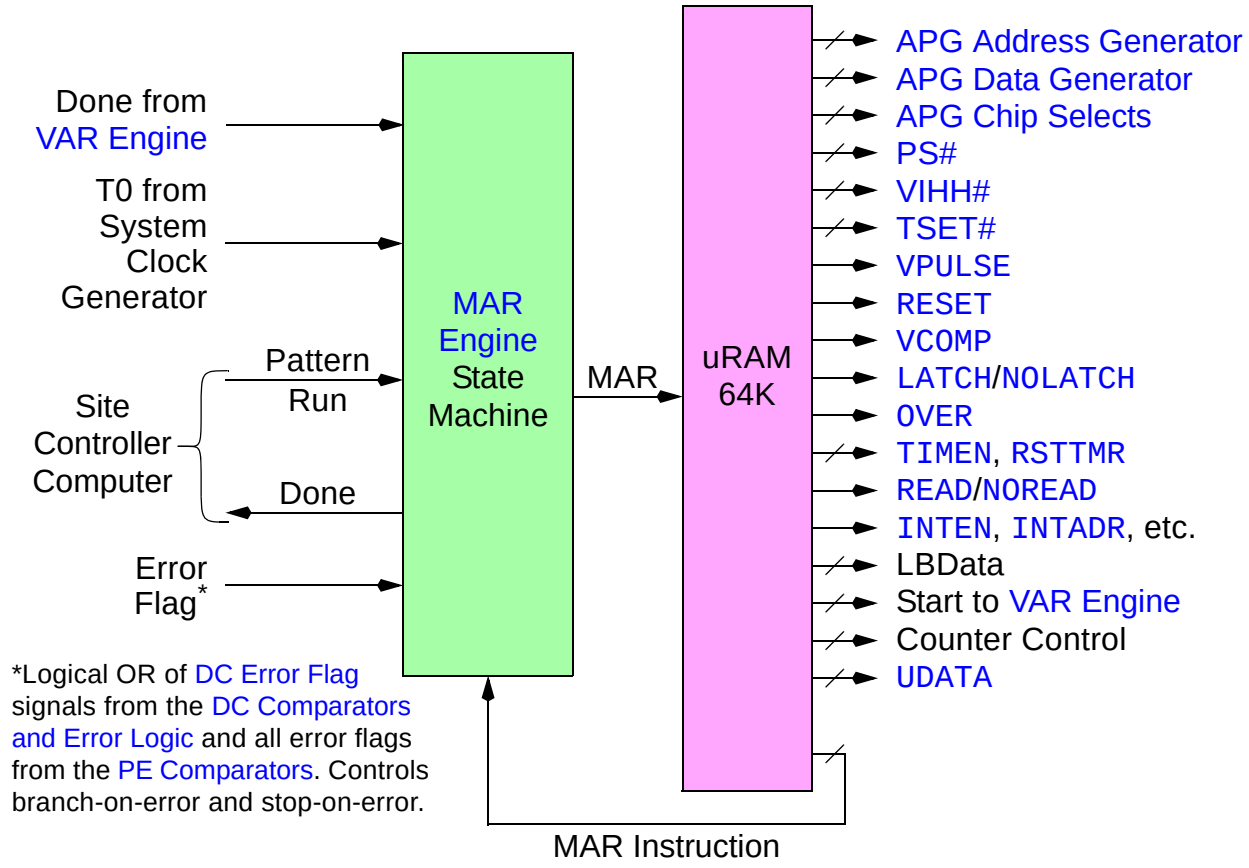


Figure-22: APG uRAM Architecture Block Diagram

In each pattern cycle, the MAR Engine state machine sends the instruction address (MAR) to the uRAM, which outputs control bits to the various APG components, and sends the next MAR instruction back to the state machine.

1.8.1.2 vRAM

See Algorithmic Pattern Generator (APG), VAR Engine.

The following diagram shows the Site Assembly Board's APG vRAM architecture:

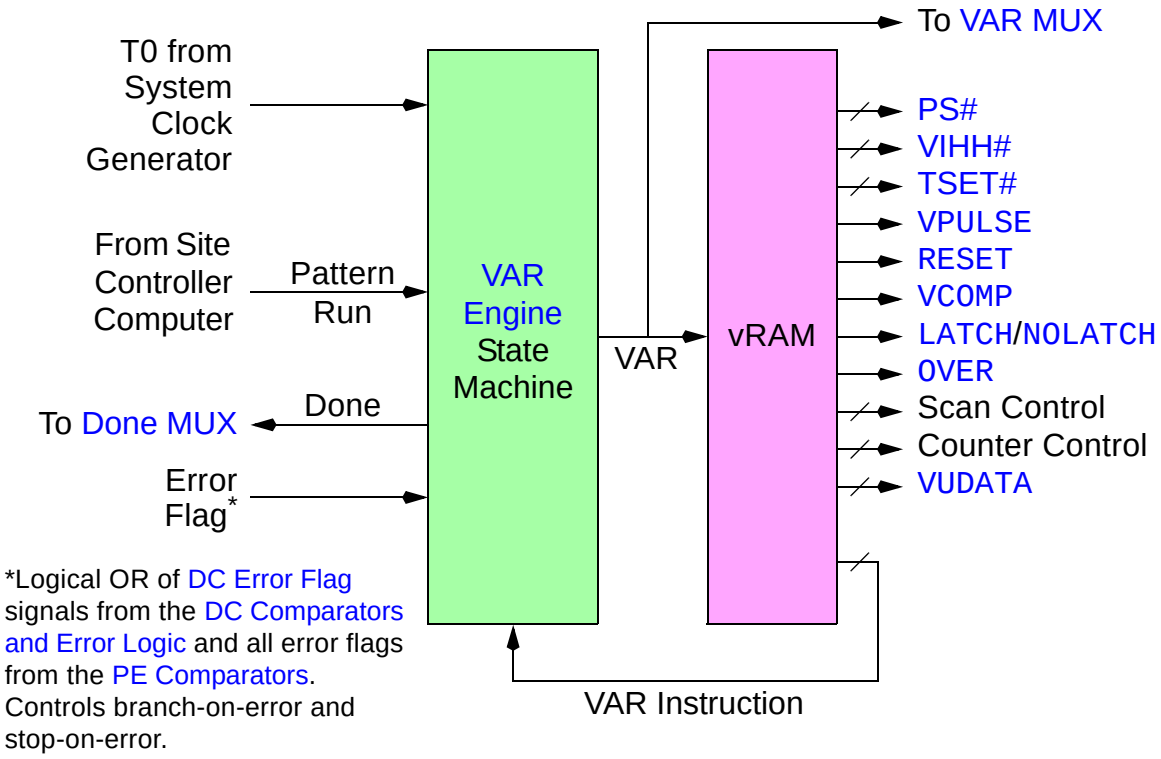


Figure-23: APG vRAM Architecture Block Diagram

In each pattern cycle, the VAR Engine state machine sends the instruction address (VAR) to the vRAM, which outputs control bits to the various APG components, and sends the next VAR instruction back to the state machine.

The vRAM is sized to match the Logic Vector Memory (LVM).

1.8.2 Branch-on-error Logic

The Magnum error logic receives one error *flag* signal from each set of 8 pins on a Site Assembly Board (see Error Flag vs. Error Latch). These signals are used by the Algorithmic Pattern Generator (APG) for test pattern *branch-on-error* operations. Much of the versatility described below applies only to Multi-DUT Test Programs; i.e. when concurrently testing multiple DUTs in parallel.

Various test pattern branch options are possible:

- Branch on any error, from any pin or DUT.
- Branch on error if any DUT(s) connected to the Sub-site-A pin have an error (or not) and no DUT(s) connected to the Sub-site-A pin have an error (or do). And, vice-versa.
- Branch on error if DUT-1 (or -2, etc.) has an error. In Multi-DUT Test Programs, up to 8 DUTs per site are supported.
- Branch if the ECR's Total Error Counters are greater than the TEC Comparator value. Similarly, for the ECR's Row Error Counters and Col Error Counters.
- Etc.

It is the error logic described below which combines and decodes the error flag signal from each set of 8 pins or the ECR Counter Comparators outputs to control these branch operations.

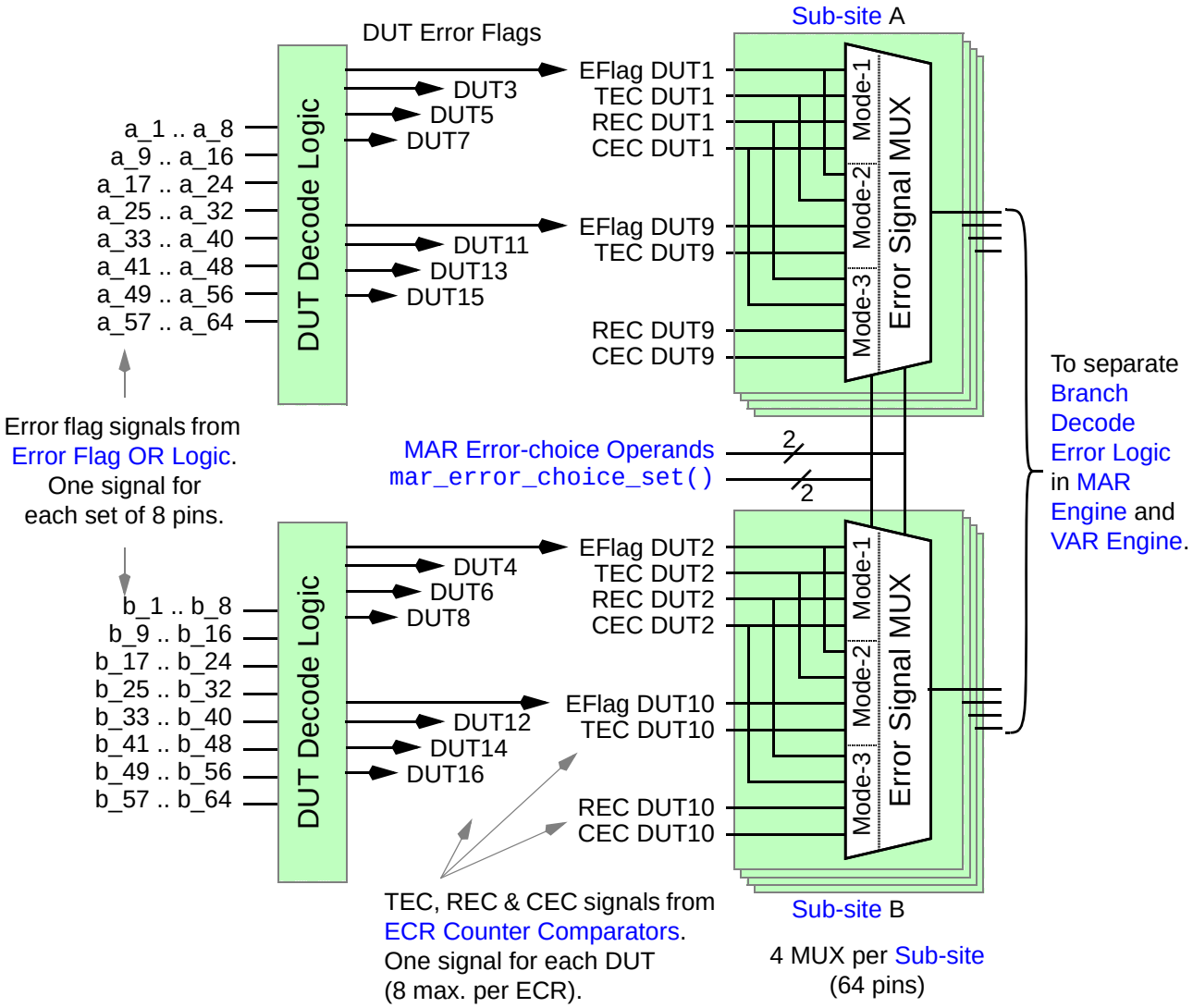
The diagrams below are divided into two sections:

- Branch Error Choice Logic - a single set of error logic shared by both the memory pattern controller (MAR Engine) and logic pattern controller (VAR Engine).
- Branch Decode Error Logic - error logic which is duplicated for both the MAR Engine and VAR Engine. In Mixed Memory/Logic Patterns this allows the memory pattern branch operation to diverge from the logic pattern branch operation.

The diagrams which follow describe the various modes and control options.

Branch Error Choice Logic

The following diagram shows the branch logic which is shared by both the memory pattern controller (MAR Engine) and logic pattern controller (VAR Engine):



Static Error Choice Functions, Branch-on-error

Figure-24: Branch Error Choice Logic

Note the following:

- The diagram above shows the error choice logic which combines the error flag signals from both Sub-sites on a given Site Assembly Board and signals from the ECR Counter Comparators for each ECR. It is this logic which determines which signals are tested when executing a conditional jump, conditional subroutine call or conditional subroutine return test pattern operations.
- The two Error Signal MUXs are controlled by two sets of inputs:
  - Two static bits are set using the `mar_error_choice_set()` function, prior to pattern execution, to select between various static modes. See Static Error Choice Functions, Branch-on-error.
  - Two bits are controlled from the test pattern by the MAR Engine, as defined using MAR Error-choice Operands. In pure Logic Test Patterns the two bits controlled from the test pattern will be set to default values since the MAR Engine is not used.

Note: proper branch operation requires that the desired error choice be continuously selected for a minimum of 4 cycles before the instruction which branches. See MAR Error-choice Operands.

Note that the default static error choice selection (`t_errmode1`) plus the default MAR Error-choice Operands (`ERRSRC1`) operation matches Maverick-I/-II memory pattern conditional branch operation, for both branch-on-error and branch-on-abort, supporting up to 4 DUTs per-64 pins.

- As indicated in MAR Error-choice Operands, there are 4 types of signals which can affect conditional branch operations:
  - Error flags, one from each set of 8 pins
  - ECR TEC Comparator signals
  - ECR REC Comparator signals
  - ECR CEC Comparator signals

  } ECR Counter Comparators

  Not all signal types can be used within a single pattern execution. Signal selection is a done in two parts, keep reading.

Note: the term *error* is used consistently, even when the ECR Counter Comparators signals are the selected error source signals. For example, `CJMPE` (jump if error) can test the REC Comparator signal and jump accordingly. See MAR Conditional Branch-condition Operands & MAR Multi-DUT Branch-condition Operands or VAR Conditional Branch-condition Operands & VAR Multi-DUT Branch-condition Operands.

- There are 4 Error Signal MUXs per Sub-site (per 64-pins), 8 per Site Assembly Board. This results in 8 error signals into the Branch Decode Error Logic. Prior to pattern execution, these MUXs are partially configured, using the `mar_error_choice_set()` function, to statically select between 1 of 4 sets of input signals. In the diagram, these are identified as Mode-1 (`t_errmode1`), Mode-2 (`t_errmode2`), Mode-3 (`t_errmode3`) and Mode-4 (`t_errmode4`). All 8 Error Signal MUXs are configured identically.
- Static option selection is primarily based on two criteria:
  - The number of DUTs being tested per Sub-site:
    - 1-4 DUTs (`t_errmode1`)
    - 5-8 DUTs (`t_errmode2`, `t_errmode3` and `t_errmode4`)
  - Whether the ECR Error Counters will control conditional branch operations, and which ECR Counter Comparators are to be tested: TEC Comparator, REC Comparator, or CEC Comparator. Each static mode has different capabilities/ limitations; i.e. which ECR Counter Comparators can be tested, see MAR Error-choice Operands.
- During pattern execution, in each pattern instruction, one of the MAR Error-choice Operands determines which of the 4 Error Signal MUX inputs (of the statically selected option group) is sent to the Branch Decode Error Logic, which further decodes the selected signals and selects which decoded signal will affect conditional branch operation. As indicated above, the Branch Decode Error Logic is duplicated for both the memory pattern controller (MAR Engine) and logic pattern controller (VAR Engine).
- In Multi-DUT Test Programs, test pattern conditional operations can test error signals organized by DUT. The number of DUTs being tested is determined by the Pin Assignment Table but the Error Signal MUX/Branch Decode Error Logic can only distinguish up to 8 DUTs per Sub-site (per 64 pins). In the diagram above, it is the DUT Decode Logic which determines which error flags represent a given DUT. Proper operation requires DUT board connections to each DUT follow the rules noted in DUT-pin to Tester-pin Connection Requirements. The ECR Counter Comparators are also configured per-DUT, with up to a maximum of 8 DUTs per ECR (16 per Site Assembly Board).

Branch Decode Error Logic

The following diagram shows how the outputs of the Branch Error Choice Logic are further decoded and selected to determine the final conditional branch signal. The logic to the right

of the magenta line is duplicated for the memory pattern controller (MAR Engine) and logic pattern controller (VAR Engine):

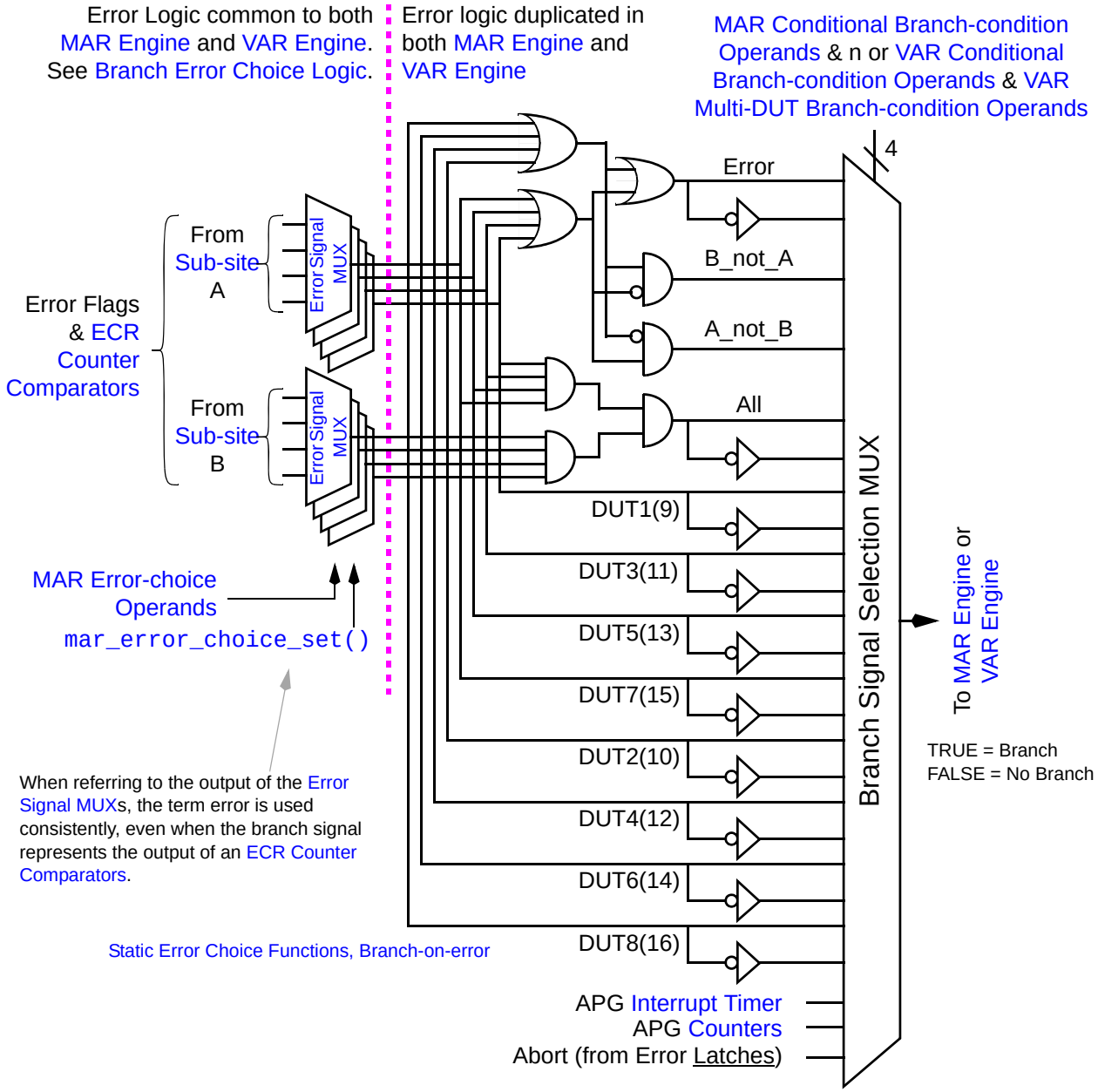


Figure-25: Branch Decode Error Logic

During pattern execution (Memory Test Patterns and Mixed Memory/Logic Patterns) each memory instruction's conditional branch operand causes the Branch Signal Selection MUX in the MAR Engine to select one input, which controls memory pattern branch operations for

that instruction. Similarly, in Logic Test Patterns and Mixed Memory/Logic Patterns, each logic instruction's conditional branch operand causes the Branch Signal Selection MUX in the VAR Engine to select one input, which controls logic pattern branch operations for that instruction. See MAR Conditional Branch-condition Operands & MAR Multi-DUT Branch-condition Operands or VAR Conditional Branch-condition Operands & VAR Multi-DUT Branch-condition Operands. In Mixed Memory/Logic Patterns using the mixedsync mode (see Pattern Type Attributes) only one engine (MAR or VAR) will control each instructions branch operation.

Note that when using some branch-on-error options, a `MAR BOE-type` operand must be specified in the instruction which executes immediately prior to the branch instruction. See MAR BOE Type Operands.

1.8.3 APG Address Generator

See Algorithmic Pattern Generator (APG), Memory Test Patterns.

To test memory devices, an APG must be able to generate complex address sequences.

The Magnum APG has two independent APG Address Generators:

- X address for the row addresses
- Y address for column addresses.

In use, the intersection of a row address and a column address determines which DUT memory cell(s) will be read or written. The dual X/Y address generation architecture provides flexibility in writing memory test patterns.

The heart of each APG Address Generators is an Arithmetic Logic Unit, or ALU. In each pattern cycle, the ALU takes one or two inputs, identified as `sourceA` and `sourceB`, from one of the three 16-bit address register(s), called `MAIN`, `BASE`, and `FIELD`, or `UDATA`, a register holding an arbitrary value, as well as one `carry/borrow` input. The ALU then operates on these inputs performing a specified function. The result of the ALU's operation is passed through a mask and then placed back in one or more of the 3 address registers. In each pattern cycle, the output of one X address register and one Y address register is selected as the address output from the APG in the current instruction.

In both the X and Y address generator, the upper three bits of the selected address source can be replaced by bits from the other address generator. These are called the Address Cross-over Bits. In the Y address generator, the Y15, Y14 and Y13 address bits can be replaced by any of the X address bits. In the X address generator, the X17, X16 and X15

address bits can be replaced by any of the Y address bits. The configuration is set statically, using Address Cross-over Bit Functions.

Finally, the address is routed to the address topological scramble RAM (Address TOPO RAM), the Data Buffer Memory (DBM), Error Catch RAM (ECR) and the DTOPO RAM (see APG Data Topological Inversion (DTOPO) Function).

The operation of the two address generators are controlled from Memory Test Patterns by the XALU and the YALU pattern instructions See Memory Test Patterns.

The following diagram shows the Y Address Generator. The X Address Generator is identical except for the Address Cross-over Bits, which can be used to replace X17, X16, and X15:

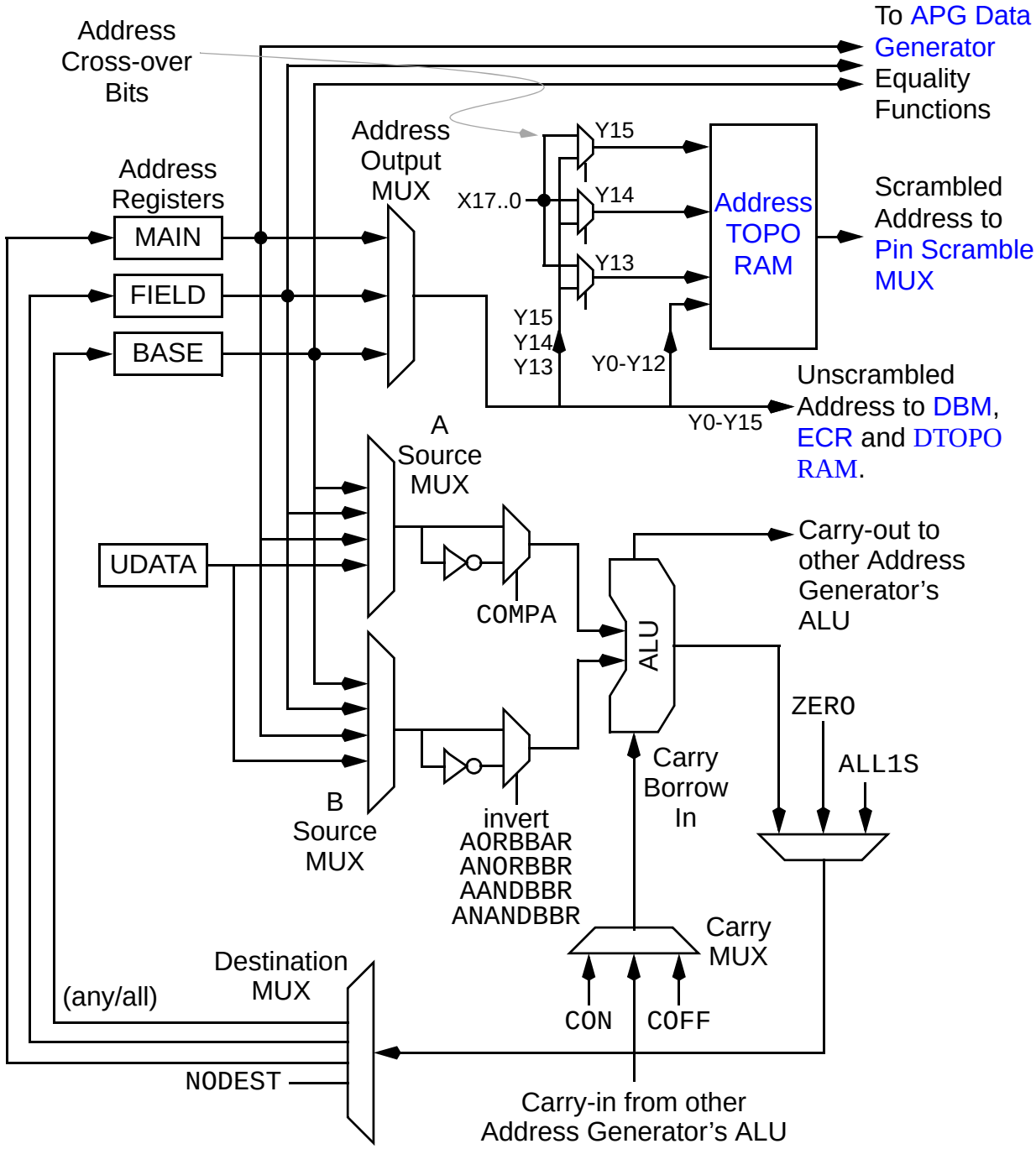


Figure-26: APG Y Address Generator Block Diagram

1.8.3.1 Address TOPO RAM

See APG Address Generator, APG Address Topo RAM Load Functions.

The X and Y APG Address Generators both have a topological scramble RAM, commonly called address TOPO RAMs.

These are used to provide address topological scrambling; i.e. a translation of logical address values, as generated by the APG Address Generator, into the topological addresses needed to correctly test the DUT. See Logical vs. Physical, vs. Electrical Addresses.

The following diagram shows a portion of the Y APG Address Generator, with an expanded view of one address TOPO RAM. The X address TOPO RAM is 256Kx18 (2^{18} X addresses), Y address TOPO RAM is 64Kx16 (2^{16} Y addresses):

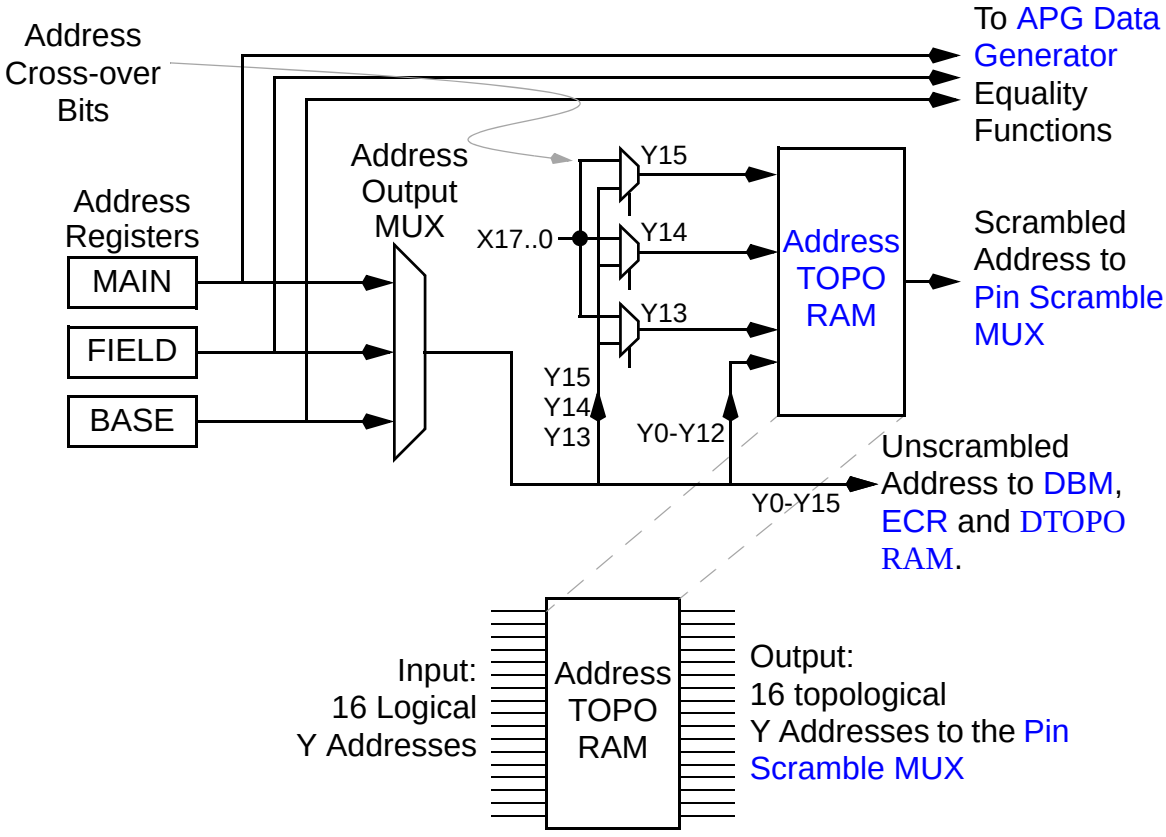


Figure-27: Address TOPO RAM Block Diagram

In simple terms, in each pattern cycle the input address selects one TOPO RAM address, which outputs its contents as the topological address. By default, the TOPO RAMs are initialized to pass a given input address through unmodified; i.e. input (logical) addess 0x0 results in output (topological) address 0, etc. User code is required to change this operation, see APG Address Topo RAM Load Functions.

1.8.4 APG Data Generator

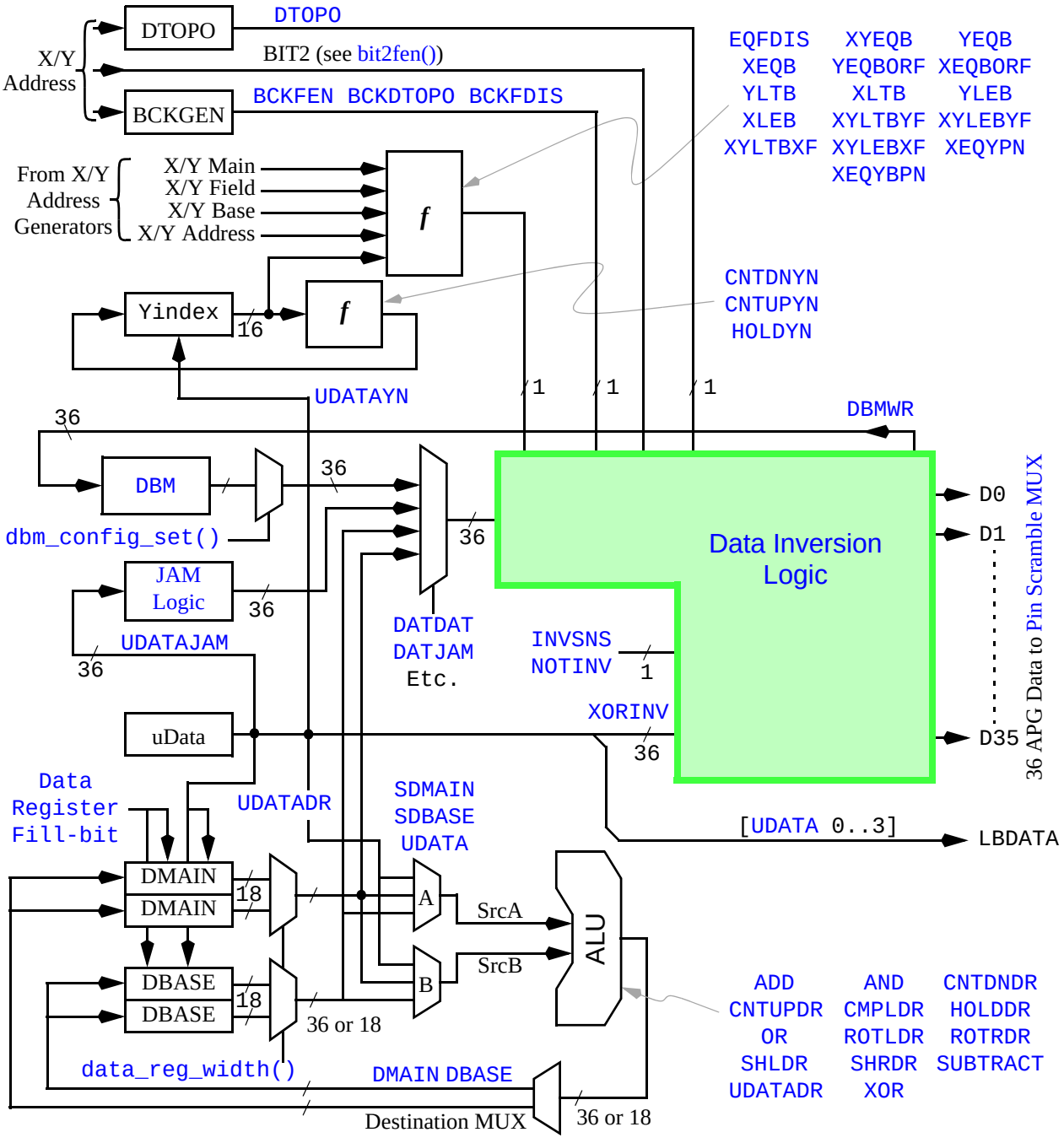


Figure-28: APG Data Generator Block Diagram

1.8.4.1 Data Inversion Logic

See APG Data Generator.

The following diagram shows the APG Data Generator inversion logic:

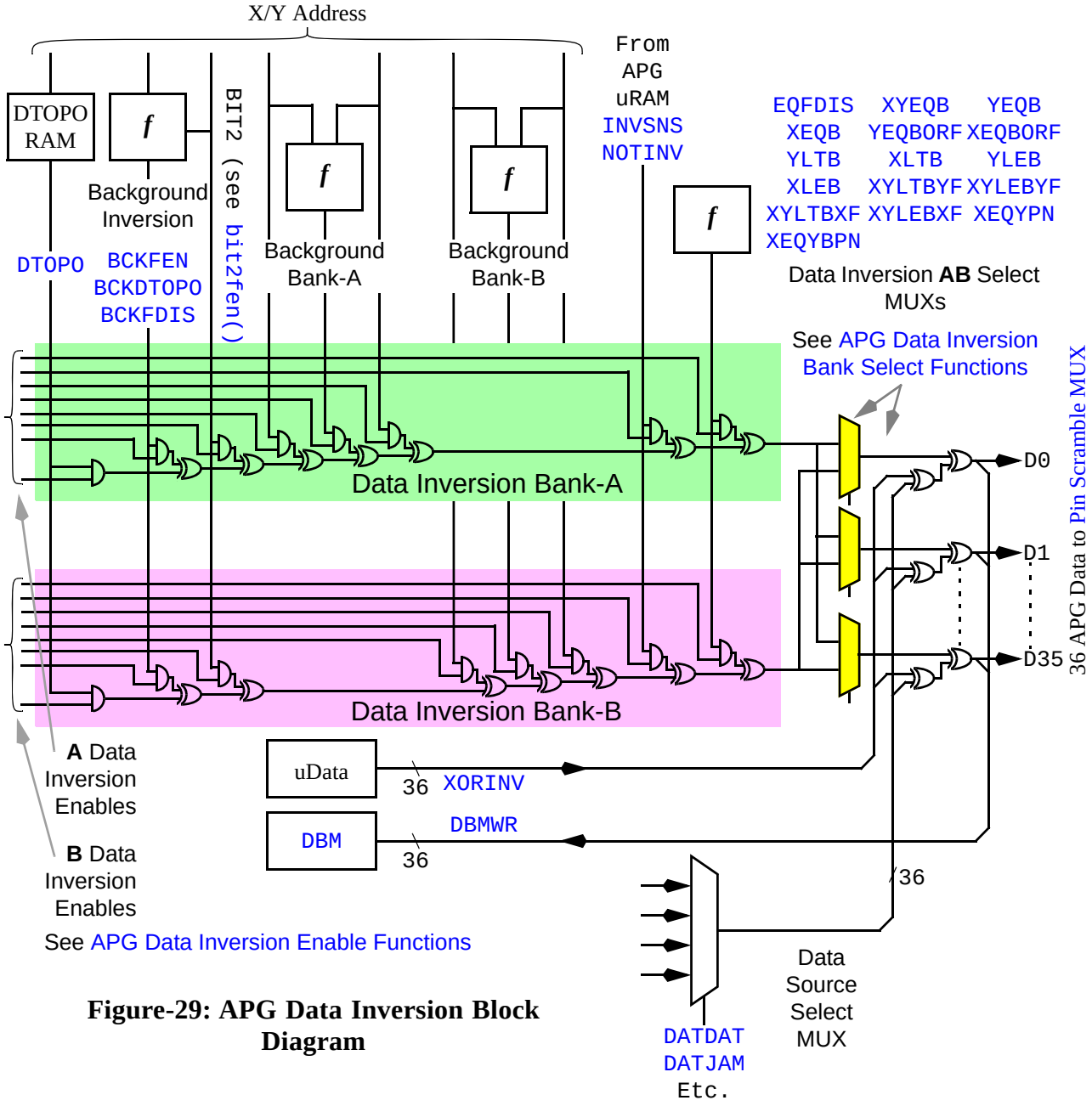


Figure-29: APG Data Inversion Block Diagram

Note the following:

- This logic will, on a per-pattern-cycle basis, perform up to 9 separate inversions on the data source selected in each cycle by the Data Source Select MUX.
  - Data topological inversion; i.e. conditionally invert data as a function of the X/Y address and the contents of two DTOPO memories, also called DTOPO RAMs. This is used when the DUT stores a logic-1 (or logic-0) in some cells inverted from other cells. See APG Data Topological Inversion (DTOPO) Function and APG Data TOPO RAM Load Functions.
  - Background data inversion; i.e. conditionally invert data as a function of X and/or Y address. The `bckfen()` function defines the desired operation, and optionally which X/Y address bit(s) are considered. A typical application of background inversion is to generate various checkerboard data patterns. See APG Background Data Inversion Function.
  - Bit-2 inversion; conditionally invert based on the logic state of a specific X or Y address bit. See `bit2fen()`.
  - APG Background Bank-A, Bank-B Inversion. This is similar to the Background data inversion except that two seperate sets of inversion logic are used with Data Inversion Bank-A outputs routed to Data Inversion Bank-A only and Background Bank-B outputs routed to Data Inversion Bank-B only. See APG Background Bank-A, Bank-B Inversion.
  - Explicit or pattern inversion. This is controlled using the test pattern `DATGEN INVSNS` and `NOTINV` instructions which control a bit issued from the APG uRAM in each pattern instruction.
  - APG address equality plus Y-index inversion; i.e. conditional inversion based on a comparison of APG X/Y address registers or X/Y address output + Y-index register. Together, these are used to generate various inverted bit, inverted row(s), inverted column(s) or diagonal data patterns. See DATGEN Equality Function Operands and DATGEN Yindex Operands.
  - Direct inversion based on the bit-wise value output from the APG uRAM in each pattern cycle. See DATGEN Invert Sense Operand. These inversions are different from the others in that they are not affected by the Data Inversion AB Select MUXs (more below).
- The Magnum APG data generator has two banks of Data Inversion Logic and 36 Data Inversion AB Select MUXs. This allows two inversion setups to be configured and selectively applied to each of the 36 APG Data Generator outputs. For example, some data outputs can have background inversion applied (see APG Background Data Inversion Function) while others use data inversion (see APG Data Topological Inversion (DTOPO) Function).

- The A Data Inversion Enables and B Data Inversion Enables are configured using the APG Data Inversion Enable Functions. Five of the six inversions have independent enable bits which must be setup before executing the test pattern.
- The Data Inversion AB Select MUXs are configured using the APG Data Inversion Bank Select Functions. These are also setup before executing the test program.

1.8.4.2 JAM Logic

See APG Data Generator, APG JAM Logic Configuration Functions.

The Magnum APG Data Generator has four data sources, selected during pattern execution, on a cycle-by-cycle basis, using the memory pattern DATGEN Instruction:

- DMAIN Register
- DBASE register
- Data Buffer Memory (DBM)
- JAM Register
- JAM RAM

The following diagram shows the JAM logic, containing the JAM Register and JAM RAM:

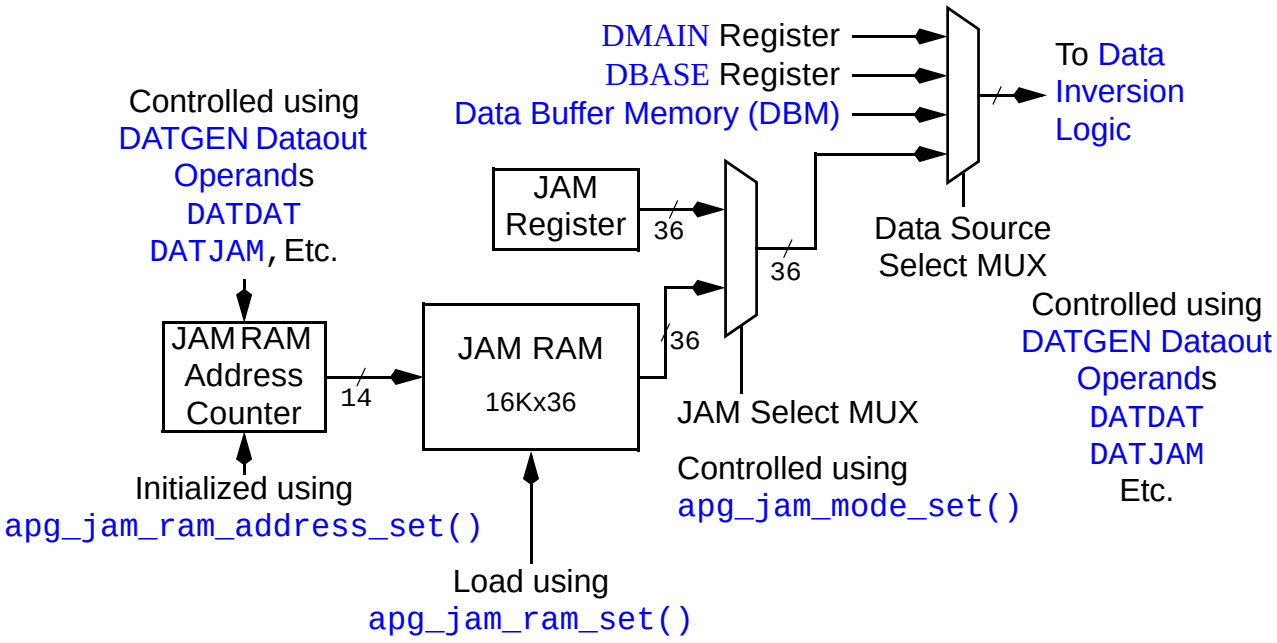
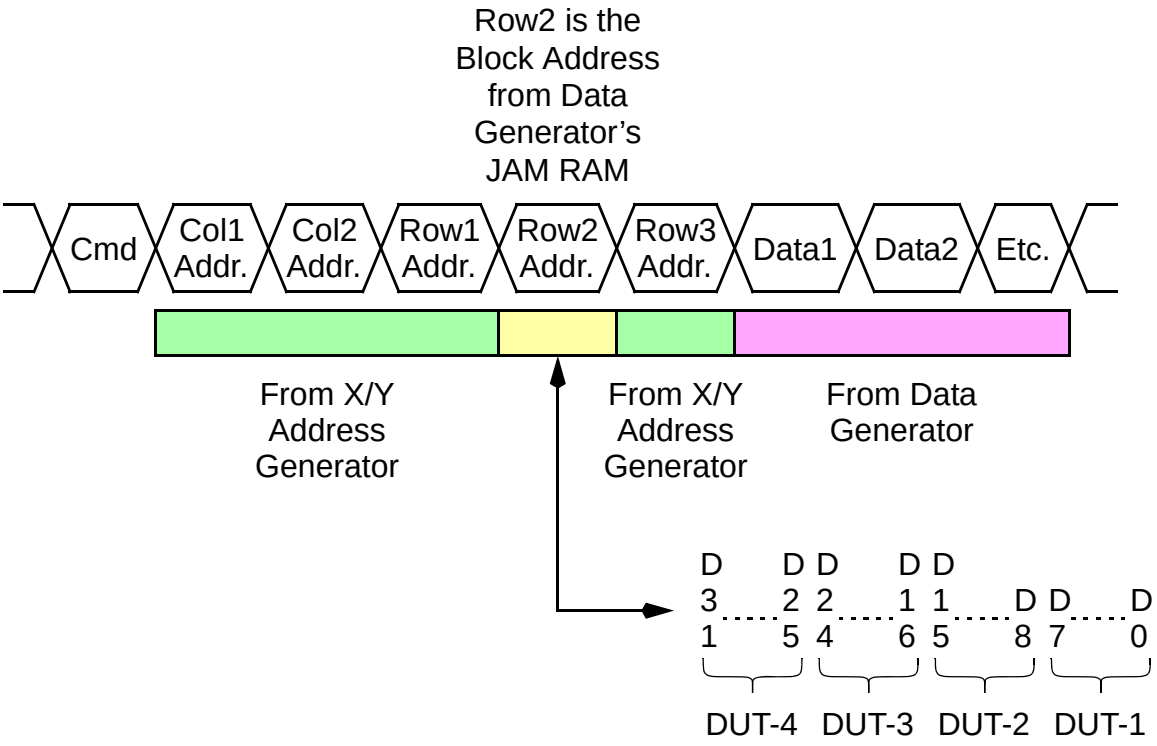


Figure-30: JAM Logic Block Diagram

The JAM Register holds a single 36-bit value. The JAM RAM, first available in in software release h1.1.23, is a 16K deep 36-bit wide memory. By default, the JAM Register is selected until `apg_jam_mode_set()` selects the JAM RAM.

The initial application for the JAM RAM was targeted at parallel testing of multiple DUTs which have:

- Known bad blocks; i.e. a series of known bad addresses which should not be tested.
- Address and data sent sequentially, with the block address being distinct from read/write data and the non-block portion of an address. For example:



To test is example device in parallel, it must be possible to concurrently send a different block address to each DUT, with the remaining portion of the address and the data being identical for each DUT. In this application the JAM RAM is used to supply the 8-bits-per-DUT block address, with each 8-bits potentially different. The Pin Scrambler makes it easy to select the JAM RAM as the source of the block address, much like a look-up table of good block addresses.

Prior to pattern execution:

- The JAM RAM is loaded with the good block table, with each JAM RAM address storing the block address for up to 4 DUTs (8 bits per DUT). See `apg_jam_ram_set()`.
- The initial JAM RAM address is set, using `apg_jam_ram_address_set()`.
- The JAM RAM is enabled, using `apg_jam_mode_set()`. By default the original JAM Register is enabled.

During pattern execution, in a given instruction the JAM RAM is selected and controlled using the DATGEN Dataout Operand, to sequence through the good blocks.

See APG JAM Logic Configuration Functions.

1.8.5 APG Chip Selects

See Algorithmic Pattern Generator (APG).

The Magnum APG has 8 chip selects, which can be selected via the Pin Scramble MUX as the source of test pattern control, per-pin, per-cycle. Note the following:

- Two of the chip selects (`t_cs1` and `t_cs2`) are bi-directional; i.e. they have I/O and strobe capability. See APG Chip Select Drive/Strobe Polarity Functions, APG Chip Select Polarity Control Function. The other six chip selects drive only.
- Chip selects are traditionally used when testing memory devices, to control the DUT's chip select, write enable, R/W pins, etc.

1.8.6 APG Interrupt Timer

See Algorithmic Pattern Generator (APG), APG Timer Functions.

Each APG contains a real-time interrupt timer, traditionally provided for testing DRAM refresh operation, but also useful when testing any device which has self-timed operation. The interrupt timer requires the use of Memory Test Patterns instructions, thus to use the timer with Logic Test Patterns instructions actually requires using Mixed Memory/Logic Patterns,

The timer consists of a counter clocked by the system clock. It is loaded with the desired time value and explicitly enabled to count-down in selected pattern instructions.

The timer can be used two ways:

- Polled i.e. explicity test for timer = zero. This uses the test pattern MAR Branch Condition Operands which test the timer (MAR CJMPT, CRETT, etc.).
- Interrupt: the timer generates a real-time interrupt during pattern execution. An interrupt can conditionally call a subroutine or cause pattern execution to conditionally branch. See MAR Interrupt Operands and MAR Timer Operands.

When the timer reaches a count of 0, an interrupt is set (pending). The timer can be used to cause an interrupt subroutine to execute at a timed interval, or to control pattern execution based on a real-time interval, both independent of the cycle period(s) currently programmed.

Timer interrupt operation is as follows:

- The timer value is programmed using the `timer()` function, which must be executed before executing the test pattern.
- The timer interrupt is cleared at the start of pattern execution.
- The interrupt address register for memory patterns (MAR Engine) is set using the `MAR INTADR` or `MAR INTENADR` operands. This identifies a pattern subroutine which may be called if an interrupt occurs (the timer may also be used without calling a subroutine, more below).
- Any pattern instruction containing the `MAR RSTTMR` operand will reset the timer to its original programmed value. Since this is the default interrupt timer operand, the timer will be reset in any pattern instruction which does not contain an explicit `MAR TIMEN` operand (next).
- The interrupt timer will count-down in any pattern instruction(s) containing the `MAR TIMEN` operand. If the timer reaches a count of 0, an interrupt will be set (pending). See previous bullet.
- The `MAR CJMPT`, `CJMPNT`, `CSUBT`, `CSUBNT`, `CRETT`, or `CRETNT`, instructions can be used at any time to perform conditional jump, subroutine call, or subroutine return based on the state of the timer (zero or not zero). Similar instructions are available for use in logic/mixed patterns: `VAR CJMPT`, `CJMPNT`, `CSUBT`, `CSUBNT`, `CRETT`, or `CRETNT`. This usage is actually more common than using the interrupt subroutine facility.

Once set, a timer interrupt will remain set (pending) until cleared by:

- Execution of an instruction containing the `MAR INTEN` or `MAR INTENADR` operands
- Execution of any of the pattern jump, subroutine, or return which is conditional based on the timer being equal or not equal to zero (see above).
- Execution of the current test pattern completes.

A pending interrupt is masked in any pattern instruction containing the `MAR NOINT` or `MAR INTADR` operands. Masked means that the interrupt subroutine will not be called in that

instruction, even it an interrupt is pending. Note that `MAR NOINT` is the default in any instruction which does not explicitly include one of the other interrupt operands.

If an interrupt is pending at the start of an instruction containing `MAR INTEN` the next instruction address is pushed onto an execution stack and the interrupt subroutine, specified in the interrupt address register, is called.

If an interrupt is pending at the start of an instruction containing `MAR INTENADR` the next instruction address is pushed on the stack and the interrupt subroutine, specified in the interrupt address register, is called. The interrupt address register is also updated with the address in the `UDATA` field, but it is the subroutine specified by the register contents at the start of the instruction that is called.

Only one timer interrupt latch exists (per APG); i.e. there is no interrupt stack or queue.

1.8.7 APG User RAM

See Algorithmic Pattern Generator (APG), APG User RAM Functions, USERRAM Instruction.

The Magnum APG contains a 4Kx32 scratch-pad memory called the User RAM, which can be used, under test pattern control, in different ways:

- Move values between select APG registers.
- Save/Restore various APG registers.
- For a given register, set multiple values from the test program to be used in the test pattern.
- Incrementally save various APG register values for retrieval after pattern execution ends.

The simplified block diagram below shows the key components of the APG User RAM:

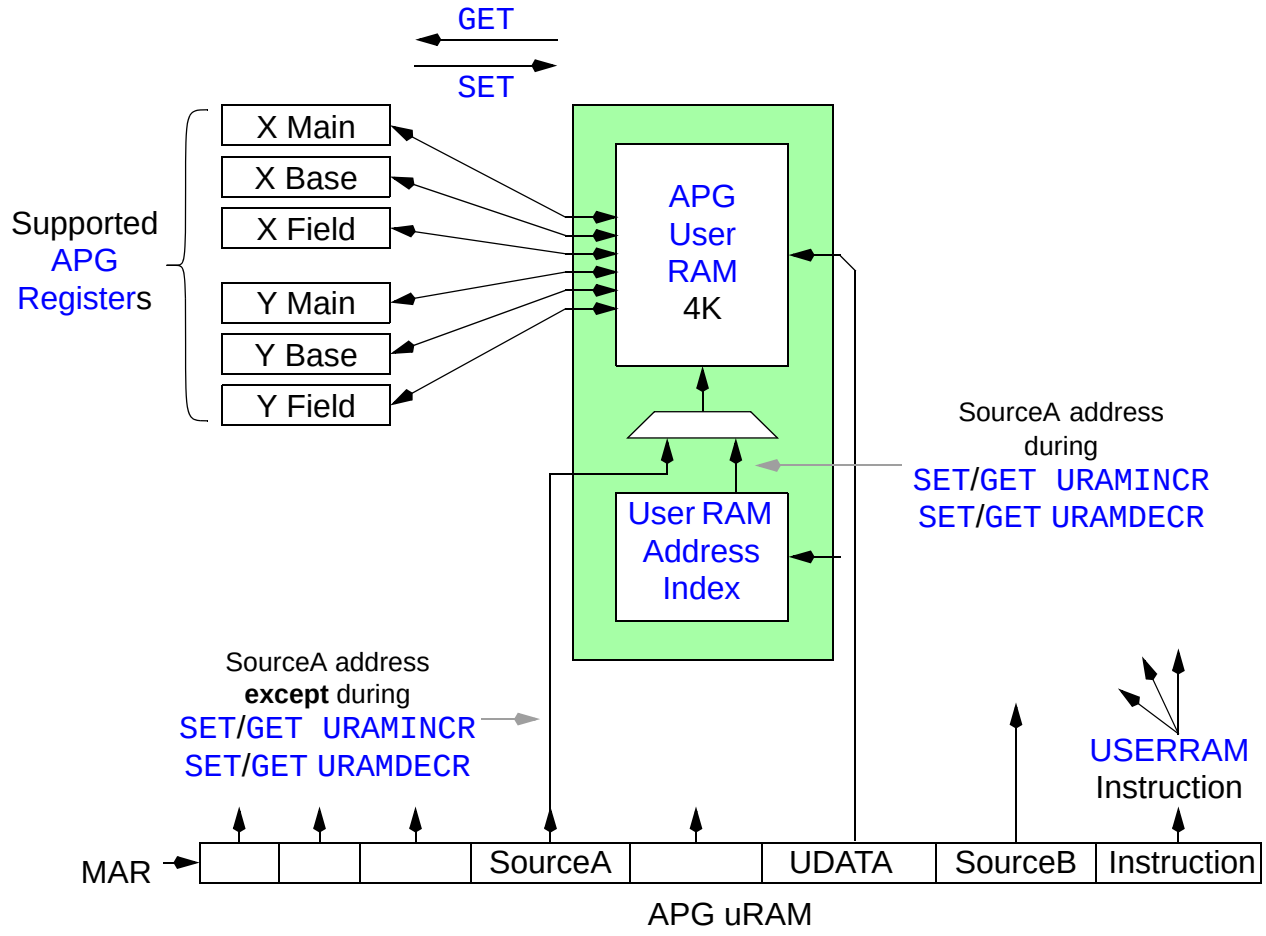


Figure-31: APG User RAM Simplified Block Diagram

Note the following:

- Before and/or after a test pattern executes the user's program can read/write values from/to the APG User RAM. See APG User RAM Functions.
- The APG User RAM is controlled, during pattern execution, from instructions stored in the APG's uRAM.

- In the pattern language, the USERRAM instruction plus various operands is used to control the APG User RAM. The following USERRAM instructions are supported and specified using USERRAM Operation Operands:

Operand	Purpose	SourceA[1]	SourceB[1]
GET	GET the value from SourceA into SourceB.	APG User RAM	APG Register
SET	SET the value from SourceB into SourceA.	APG User RAM	APG Register
Note-1: SourceA and SourceB refer to operands of the USERRAM instruction.			

1.8.7.1 User RAM Address Index Register

See Algorithmic Pattern Generator (APG), APG User RAM.

Some APG User RAM operations require selecting one or two User RAM addresses (URAM1-URAM4096), to specify the source and/or destination for the instruction. In the pattern language these are explicitly specified using the USERRAM Instruction SourceA and SourceB operands (see USERRAM SourceA Operands and USERRAM SourceB Operands).

When used, the SourceB value is always explicitly specified, however, the SourceA address can be either an explicit address (URAM1 to URAM4096) or can be the output of the *Address Index Register*. The latter is selected by specifying either URAMINCR or URAMDECR as the SourceA value (see USERRAM SourceA Operands).

The main application of the Address Index Register is to sequentially access the APG User RAM, typically in a pattern loop. Thus, when used, the Address Index Register is always either post-incremented or post-decremented.

Once the Address Index Register is loaded with an initial address (USERRAM LOAD) that address can be used (instead of an explicit address) to Get/Set a value from/to the APG User RAM. For example, to clear all or part of the APG User RAM, or set all or several addresses to 0xFF, etc. Note the following:

- If an explicit SourceA address is specified (URAM1 to URAM4096) the Address Index Register is not used or modified.

- If URAMINCR is specified as SourceA, the address in the Address Index Register determines which APG User RAM address is accessed by the current USERRAM instruction. Then, the Address Index Register is incremented in preparation for the next URAMINCR or URAMDECR instruction.
- If URAMDECR is specified as SourceA, the address in the Address Index Register determines which APG User RAM address is accessed by the current USERRAM instruction. Then, the Address Index Register is decremented in preparation for the next URAMINCR or URAMDECR instruction.
- The values in the Address Index Register do wrap, in both directions.

1.8.8 Data Buffer Memory (DBM)

See Site Assembly Board, Algorithmic Pattern Generator (APG), Data Buffer Memory Software (DBM).

The Magnum test system has two test pattern data sources:

- APG (optionally including the DBM): see Memory Test Patterns.
- The combined Logic Vector Memory (LVM) / Scan Vector Memory (SVM) see Logic Test Patterns and Scan Test Patterns.

The DBM is needed when the APG is used to test memory devices using *stored* pattern data; i.e. non-algorithmically generated drive/compare data.

The original application of the DBM was testing ROMs, where the DUT contains non-volatile data. The data pattern stored in a ROM isn't written to the DUT during testing, and is arbitrary, i.e. effectively random, non-algorithmic, etc. When testing the ROM, this read-only data is loaded into the DBM from a disk file, with each ROM code typically contained in a different file. Then, during functional testing, the DBM is selected as the data source (DATGEN BUFBUF) when doing the actual read cycles in APG pattern instructions.

Although ROM testing originally identified the need for a DBM, the DBM can also be used for testing any device type where (a) the APG is to be used as the pattern data source, and (b) stored pattern data is needed. Cycle-by-cycle, DBM data can be used as drive data, for writing to the DUT, or as compare data when strobing the DUT. Note that the APG data source is selectable on a cycle-by-cycle basis, with the DBM selection being one data source option.

See Data Buffer Memory Software (DBM) for programming information. The DBM Architecture is described further below.

1.8.8.1 DBM Architecture

See Site Assembly Board, Algorithmic Pattern Generator (APG), Data Buffer Memory (DBM).

The Data Buffer Memory (DBM) option consists of two SIMM memory modules mounted on each Site Assembly Board. Both modules must be installed to use the DBM.

The DBM is implemented using DRAM, which may be used in an interleaved mode to provide random DBM access at full speed or non-interleaved mode to increase the usable DBM size, with restrictions on either access speed or address sequencing. See DBM DRAM Interleaving and DBM Sequential Mode. The following table describes the DBM size options vs. interleave mode:

Table 1.8.8.1-1 DBM Memory Size Options

Installed DBM Size	Usable DBM Size	
	Interleaved	Non-Interleaved
72 MBits	72 MBits	576 MBits
144 MBits	144 MBits	1152 MBits
288 MBits	288 MBits	2304 MBits
576 MBits	576 MBits	4608 MBits
1152 MBits	1152 MBits	9216 MBits
2304 MBits	2304 MBits	18432 MBits
Note: not all DBM memory size options are initially available.		

See Data Buffer Memory Software (DBM) and DBM Usage Rules.

1.9 Logic Vector Memory (LVM)

The Logic Vector Memory (LVM) and Scan Vector Memory (SVM) are the same memory and consists of a set of optional memory modules mounted on each Site Assembly Board.

The LVM/SVM is used to store both Logic Test Patterns and Scan Test Patterns. The combined LVM/SVM has 64 3-bit outputs, consisting of:

- One drive/strobe data bit (PEL)
- One strobe control bit (PES)
- One I/O control bit (PEE)

The LVM/SVM stores Logic Test Patterns, commonly used to test logic devices, or portions of memory devices which can't be tested using test patterns generated algorithmically (APG) and Scan Test Patterns, used to test the DUT via, for example, an IEEE TAP port. In both Logic Test Patterns and Scan Test Patterns, these are controlled using the same Logic Vector Bit Codes.

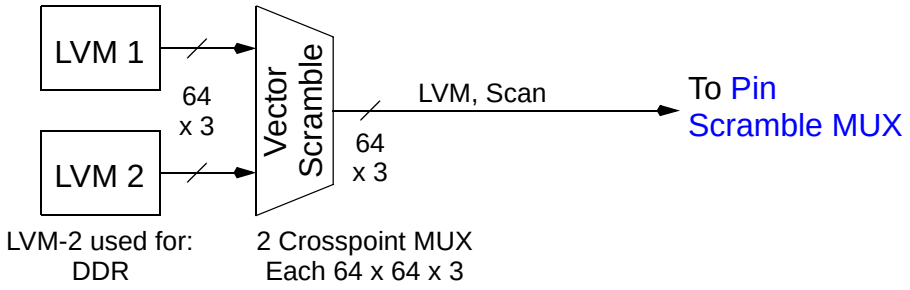
The Magnum LVM/SVM architecture has the following key features:

- DRAM + SRAM architecture, used to store both logic vectors and scan vectors. Use of DRAM provides large, cost effective LVM/SVM. The SRAM is used in special situations which require faster memory access than possible using DRAM. SRAM usage is transparent to the user; it is only noted here to help explain Magnum 1/2/2x Logic Pattern Rules.
- As noted above, each LVM/SVM address stores 3-bits per timing channel (see table below) plus other per-vector information (opcode, time-set selection, pin scramble address, etc.).
- Two banks of LVM. The 2nd bank used to:
  - Store DDR vectors
  - Store DDR scan vectors
- Up to 64 scan channels per Site Assembly Board. Scan vectors are packed, to conserve/optimize LVM/SVM use (see diagram below). Optimum scan vector storage is obtained when the user's test pattern defines scan data in sets of 1, 2, 4, 8, 16, and 64 pins.
- The Magnum Vector Scramble MUX allows any LVM/SVM output to be mapped to any DUT pin(s),

The Magnum LVM architecture supports the following logic pattern features:

- Single vector repeat loops.
- Multi-vector loops nested to 3 levels (with restrictions)
- Vector subroutines nested to 2 levels (with restrictions)
- Branch-on-error, stop-on-error (with restrictions)
- Mixed Memory/Logic Patterns test patterns

- The model below shows the architecture of the two LVM/SVM banks and how conventional logic vectors and scan vectors use the same LVM space:



	Opcode 24 bits	Drive Data (PEL) 64 bits	Strobe Data (PES) 64 bits	I/O Control (PEE) 64 bits
Parallel Vectors	VEC TSET1 PS1 VIHH1	0 1 1 0 0 1	0 0 0 0 0 0	0 0 0 0 0 0
	VEC TSET2 PS4 VIHH1	1 0 1 0 1 0	0 0 0 0 0 0	0 0 0 0 0 0
	VEC TSET2 PS4 VIHH2	1 1 1 0 1 1	1 1 1 1 1 1	1 1 1 1 1 1
	RPT 10 TSET2 PS4 VIHH3	0 0 1 0 0 0	1 0 0 1 1 0	1 1 1 1 1 1
	VEC TSET2 PS4 VIHH2	0 0 0 1 0 0	0 1 1 0 0 1	1 1 1 1 1 1
	VEC TSET1 PS1 VIHH1	1 1 0 1 1 1	1 1 1 1 1 1	1 1 1 1 1 1
	VEC TSET2 PS4 VIHH1	0 1 0 1 0 1	0 0 0 0 0 0	0 0 0 0 0 0
		1 0 0 1 1 0	0 0 0 0 0 0	0 0 0 0 0 0
Scan Vectors		0 0 0 0····	0 1 0 0····	0 1 1 1····
		1 0 0 1····	0 1 0 0····	0 1 1 1····
		0 1 0 0····	0 0 1 0····	0 1 1 1····
	Scan Cycle	0 | 1 | 2 | 3 |...	0 | 1 | 2 | 3 |...	0 | 1 | 2 | 3 |...
Parallel Vectors	VEC TSET1 PS1 VIHH1	0 1 1 0 0 1	0 0 0 0 0 0	0 0 0 0 0 0
	VEC TSET2 PS4 VIHH1	1 0 1 0 1 0	0 0 0 0 0 0	0 0 0 0 0 0
	RPT 55 TSET6 PS2 VIHH1	1 1 1 0 1 1	1 1 1 1 1 1	1 1 1 1 1 1
	RPT 10 TSET2 PS9 VIHH1	0 0 1 0 0 0	1 0 0 1 1 0	1 1 1 1 1 1
	VEC TSET4 PS8 VIHH1	0 0 0 1 0 0	0 1 1 0 0 1	1 1 1 1 1 1
	VEC TSET1 PS1 VIHH1	1 1 0 1 1 1	1 1 1 1 1 1	1 1 1 1 1 1
	VEC TSET2 PS4 VIHH1	0 1 0 1 0 1	0 0 0 0 0 0	0 0 0 0 0 0
		1 0 0 1 1 0	0 0 0 0 0 0	0 0 0 0 0 0

The arrows show the direction that pattern execution sequences through LVM contents. Note that conventional logic vectors and scan vectors are different.

Figure-32: Magnum Logic Vector Memory Architecture

The Opcode field stores:

- Vector instruction: `VEC`, `RPT`, `VPULSE`, `VCOMP`, `RESET`, `NOLATCH`, `OVER`
- Repeat count; i.e. `RPT` 12, etc.
- Pin scramble select: PS2, etc.

- Time-set select: TSET 4, etc.
- VIHH select: VIHH5, etc.

1.10 Scan Vector Memory (SVM)

See Site Assembly Board.

The Scan Vector Memory (SVM) and Logic Vector Memory (LVM) are the same memory and consists of a set of optional memory modules mounted on each Site Assembly Board.

The combined LVM/SVM stores both Logic Test Patterns and Scan Test Patterns.

See Logic Vector Memory (LVM) for more information.

1.11 Error Catch RAM (ECR)

See Site Assembly Board.

The optional Error Catch RAM (ECR) is used to capture functional failures, per-pin, per-cycle, in real time. Typical applications include:

- Memory test Bitmap support; i.e. BitmapTool.
- Memory test Redundancy Analysis (RA).
- Logic Error Catch (LEC).

Key Features:

- Each Site Assembly Board supports two ECR's, 1 ECR captures errors from Sub-site A pins and 1 ECR captures errors from Sub-site B pins (see PE Sub-site Architecture).
- Fail capture at 50Mhz. See Magnum ECR Memory Size Options.
- Built-in Logic Error Catch (LEC).
- Bit line error counter (IOC Error Counters below).
- Word line error counter (Row Error Counters/Col Error Counters below).
- Total failures error counter (Total Error Counters below).
- ECR Mini-RAM, to support block or segment oriented device testing.
- Test pattern on-the-fly branching based on Row Error Counters, Col Error Counters and Total Error Counters.

Note: the following usage restriction was added 3/25/05. This restriction is required to ensure that DUT boards and test programs written for Magnum will operate in Magnum-compatible systems being planned for future development.

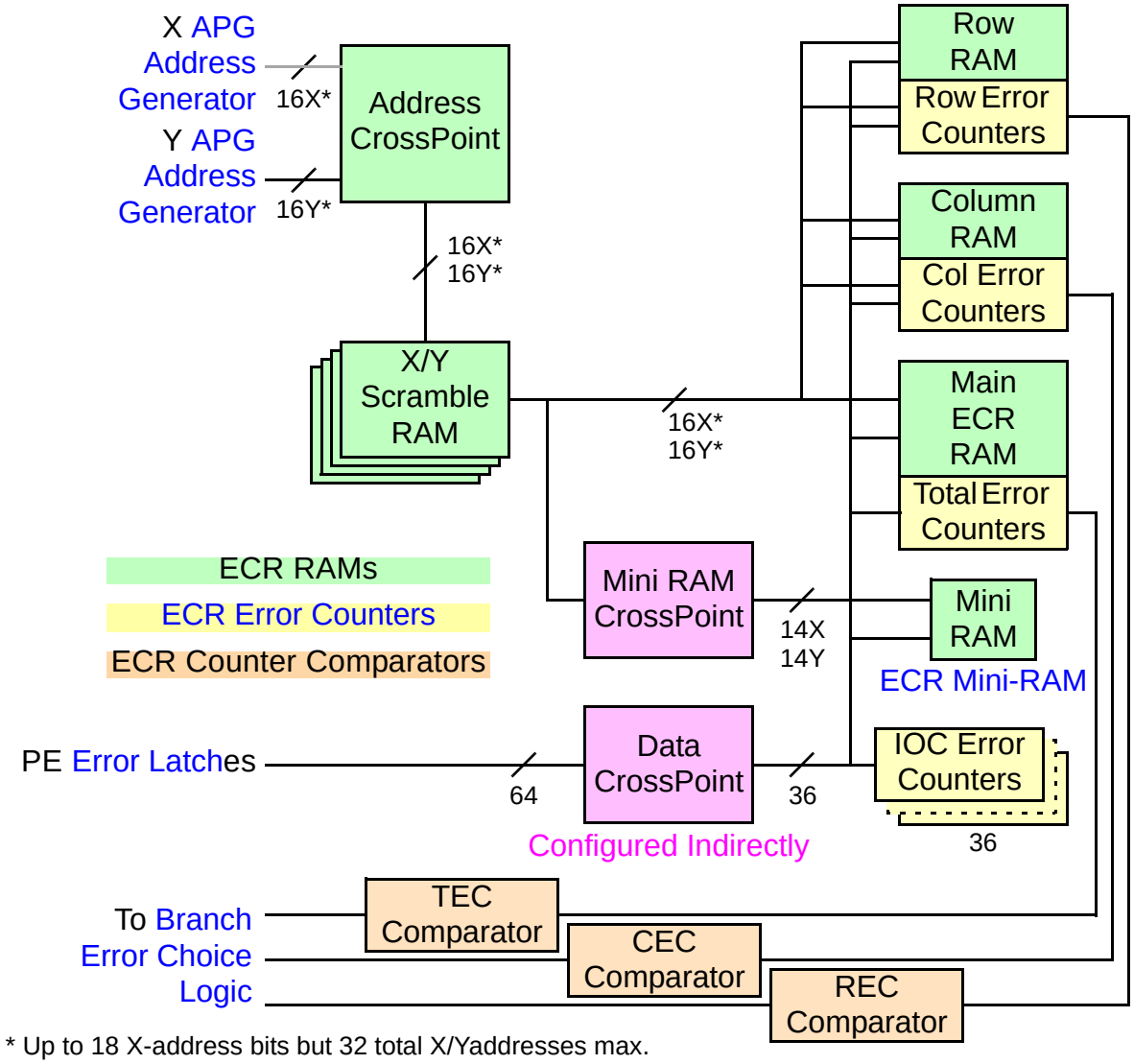
As note above, each Site Assembly Board supports two ECR's: 1 ECR captures errors from Sub-site A pins and 1 ECR captures errors from Sub-site B pins. Each ECR can capture errors from up to 36 pins in its associated Sub-site, however, to maintain DUT board and test program compatibility with future Magnum systems, at most a maximum of 18 pins from each group of 32 pins should be captured. In other words:

- From Sub-site A, capture any 18 pins from a_1 to a_32
- From Sub-site A, capture any 18 pins from a_33 to a_64

- From Sub-site B, capture any 18 pins from b_1 to b_32
- From Sub-site B, capture any 18 pins from b_33 to b_64

This requires that the user carefully consider which tester pins are connected, via the DUT board, to the DUT pins which are to be captured in the ECR.

The following diagram is used to describe the ECR architecture:



* Up to 18 X-address bits but 32 total X/Yaddresses max.

Figure-33: Magnum ECR Block Diagram

Note the following:

- The Main ECR RAM is implemented using burst DRAM, using an interleaving technique to provide random access. To capture errors at the Magnum's maximum data rate (i.e. 50MHz/20nS strobe rate) requires an interleave ratio of 8; i.e. the entire ECR main memory size is divided by 8. Conversely, as the capture data rate is reduced the interleave ratio is reduced, effectively increasing the size of the ECR. When configuring the ECR (see `ecr_config_set()`), the `fastest_cycle` argument is used to specify the maximum strobe rate (on all pins) in test pattern(s) which are to capture errors in the ECR. The system software uses this value to set up ECR interleaving, in one of 4 configurations:

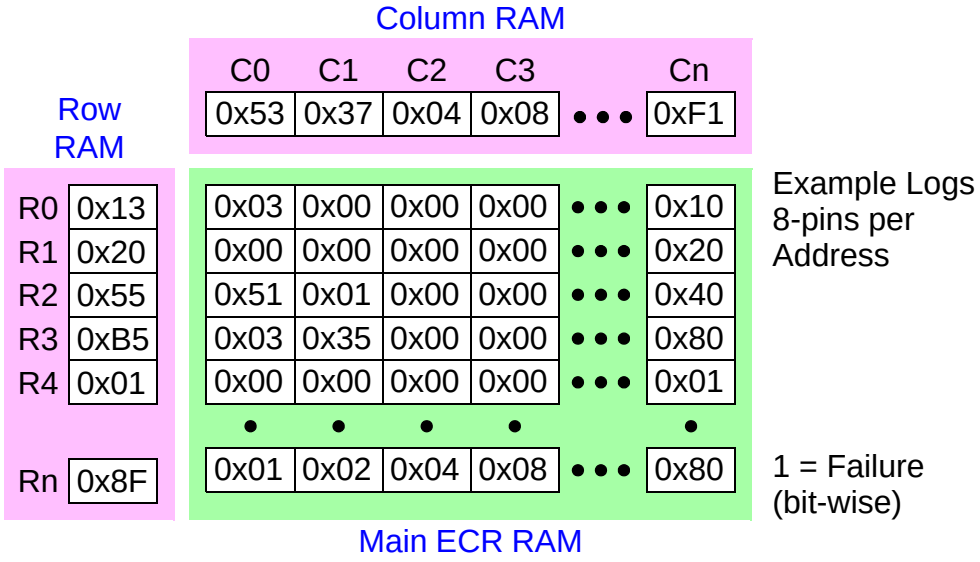
Table 1.11.0.0-1 Magnum ECR Memory Size Options

ECR Size	Max Capture Speed vs. Effective ECR Depth				First Avail.
	50MHz	25MHz	12.5MHz	6.25MHz	
72 MBit	72 MBit	144 MBit	288 MBit	576 MBit	
144 MBit	144 MBit	288 MBit	576 MBit	1152 MBit	
288 MBit	288 MBit	576 MBit	1152 MBit	2304 MBit	
576 MBit	576 MBit	1152 MBit	2304 MBit	4608 MBit	
1152 MBit	1152 MBit	2304 MBit	4608 MBit	9216 MBit	
2304 MBit	2304 MBit	4608 MBit	9216 MBit	18432 MBit	h2.0.12 h1.1.27

Note that in the product specification the ECR size is specified for full-speed operation; i.e. 8-way interleaving.

- The Address CrossPoint provides for address compression; i.e. one or more X and/or Y address bits are not used (ignored, discarded, etc.) when addressing the ECR. In effect, this results in a range of addresses which are captured to the same location in the ECR. X and Y address compression is configured using two arguments to `ecr_config_set()`. The current address compression configuration can be read using `ecr_config_get()`.
- The Data CrossPoint is used to map individual pin error flags to the ECR; i.e. which pins are captured into each ECR data location. It also provides for data compression; i.e. errors from multiple pins are captured to the same data in the ECR. The Data CrossPoint is indirectly configured using `ecr_config_set()`. The configuration pinlist can be retrieved using `ecr_config_get()`.

The following diagram shows how errors are logged to the Main ECR RAM, Row RAM, and Column RAM:



- The Row RAM and Column RAM are used to improve the ECR read (scan) performance. During test pattern execution, each time a failure is logged to the ECR at a given address, a bit is also set in the Row RAM and Column RAM, at the same address. Then, the ECR scan routine can skip rows/columns which do not have a bit set in the Row RAM or Column RAM. These RAMs can be read from user code using, `ecr_row_ram_read()`, `ecr_row_ram_scan()`, `ecr_col_ram_read()`, `ecr_column_ram_scan()`. These RAMs can be modified from user code using `ecr_row_ram_write()`, `ecr_col_ram_write()`. These RAMs can be cleared using `ecr_all_clear()`, `ecr_rams_clear()` and, optionally, `ecr_area_clear()`. Note that `ecr_config_set()` also clears these RAMs.
- The Mini RAM is used when testing block or segment oriented memory devices. In simple terms, each block or segment of the main ECR is mapped to one address in the Mini RAM. Then, during test pattern execution, any time an error is logged to the ECR the appropriate bit in the Mini RAM is also set. After the pattern completes, the contents of the Mini RAM indicate which block(s) or segment(s) contain errors. The Mini RAM is configured using `ecr_miniram_config_set()`, which indirectly configures the Mini RAM CrossPoint. This defines the scope of each Mini RAM address; i.e. how many and which X/Y addresses are mapped to a given Mini RAM address. The current Mini RAM configuration can be retrieved using `ecr_miniram_config_get()`. The Mini RAM can be read using `ecr_miniram_scan()` and `ecr_miniram_read()`. The Mini RAM can be

modified using ecr_miniram_write(). The Mini RAM can be cleared using ecr_all_clear(), ecr_rams_clear(), and, optionally, ecr_area_clear(). Note that ecr_config_set() also clears these RAMs.

1.11.1 ECR Error Counters

See Error Catch RAM (ECR).

As indicated in the Magnum ECR Block Diagram, the ECR contains several error counters, some with corresponding ECR Counter Comparators:

- Total Error Counters (TEC) : 34-bit counter(s) which count total failing bits or total failing addresses (see ecr_counters_config_set()). In Multi-DUT Test Programs, each DUT has an independent TEC (up to 8 per ECR). The TEC Comparator signal to the Branch Error Choice Logic will be TRUE when the value in the TEC counter exceeds the count specified using ecr_compare_reg_set(). This can be used to control branch operations in Memory Test Patterns using the MAR Error-choice Operands.
- Col Error Counters (CEC): 18-bit counter(s) which count the number of failing bits or failing addresses in the Column RAM. In Multi-DUT Test Programs, each DUT has an independent CEC (up to 8 per ECR). The CEC Comparator signal to the Branch Error Choice Logic will be TRUE when the value in the CEC counter exceeds the count specified using ecr_compare_reg_set(). This can be used to control branch operations in Memory Test Patterns using the MAR Error-choice Operands
- Row Error Counters (REC): 18-bit counter(s) which count the number of failing bits or failing addresses in the Row RAM. In Multi-DUT Test Programs, each DUT has an independent REC (up to 8 per ECR). The REC Comparator signal to the Branch Error Choice Logic will be TRUE when the value in the REC counter exceeds the count specified using ecr_compare_reg_set(). This can be used to control branch operations in Memory Test Patterns using the MAR Error-choice Operands

- IOC Error Counters (IOC): 32-bit counters which count the number of errors on each ECR input. There are 36 I/O Counters (IOC1 through IOC36). In Multi-DUT Test Programs, these are evenly distributed between DUTs as follows:

Table 1.11.1.0-1 ECR I/O Counter Distribution

DUTs/ECR	IOC/DUT
1	36
2	18
4	9
8	4

  For example, given 4 Duts/ECR, the system software configures the IOC Error Counters to provide 9 IOC counters per-DUT, `t_ioc1`..`t_ioc9`.
- When data compression is used (see `ecr_config_set()`), in a given pattern cycle, the various counters see a single error per compressed input, regardless of how many pin errors actually occur before compression is applied.
- ECR Error Counters are configured using `ecr_counters_config_set()`. The current configuration can be retrieved using `ecr_counters_config_get()`.
- A given counter can be read using `ecr_error_counter_get()`. A counter can be modified using `ecr_error_counter_set()`.
- Counters can be cleared using `ecr_all_clear()` and `ecr_counters_clear()`. Note that `ecr_config_set()` also clears these counters.
- Each TEC, REC and CEC counter has a corresponding comparator. The output of these comparators may be used to affect test pattern conditional branch operations, see Branch-on-error Logic and MAR Error-choice Operands. The comparator values are accessed using `ecr_compare_reg_set()`, `ecr_compare_reg_get()`. When counter/comparator is used to control branch operations, the user's test pattern must meet the Error Pipeline Requirements.

1.11.2 ECR Mini-RAM

See Error Catch RAM (ECR).

As shown in the Magnum ECR Block Diagram, the ECR contains a feature called the Mini RAM, which is used to obtain a compressed view of the Main ECR RAM. In hardware, the Mini RAM is a 16Kx1 memory, storing one bit at each address.

The target Mini RAM application is to obtain a block or segment-oriented view of the errors logged to the Main ECR RAM. Because the Mini RAM represents a [potentially highly] compressed version of the Main ECR RAM, the Mini RAM can be used to quickly determine if any errors occurred in a large chunk of the Main ECR RAM. Devices with redundant resources commonly have spare rows and/or columns. The Mini RAM is most useful when the DUT has spare segments or blocks.

Note the following:

- The Mini RAM is not configured by the system software; i.e. user code is required. This is done using `ecr_miniram_config_set()`, after configuring the ECR using `ecr_config_set()`, and prior to using the ECR. In hardware, this configures the Mini RAM CrossPoint. The `ecr_miniram_config_get()` function can be used to get the current Mini RAM configuration.
- During pattern execution, errors are concurrently logged to the Main ECR RAM, Mini RAM, Row RAM and Column RAM.
- After errors are captured to the ECR, the `ecr_miniram_scan()` function may be used to scan the Mini RAM, to determine whether any errors were captured for each range of Main ECR RAM, each range typically equating to a segment or block. This is much faster than scanning the Main ECR RAM and/or Row RAM and/or Column RAM and analyzing the results to identify which segment(s) or block(s) had errors.
- In Multi-DUT Test Programs, each DUT has an independent Mini RAM (up to 8 per ECR).

1.12 DUT Board I/O Ports

See DUT Board I/O Port Functions.

The Magnum 1, 2 and 2x have the following ports available at the DUT board to control or access external circuitry:

- I2C Bus
- SPI Port & GPIO Port
- Loadboard Board Data Bits (LBDATA)

1.12.1 I2C Bus

See DUT Board I/O Ports, I2C Bus Functions.

The following diagram shows how devices are connected to the I2Cbus:

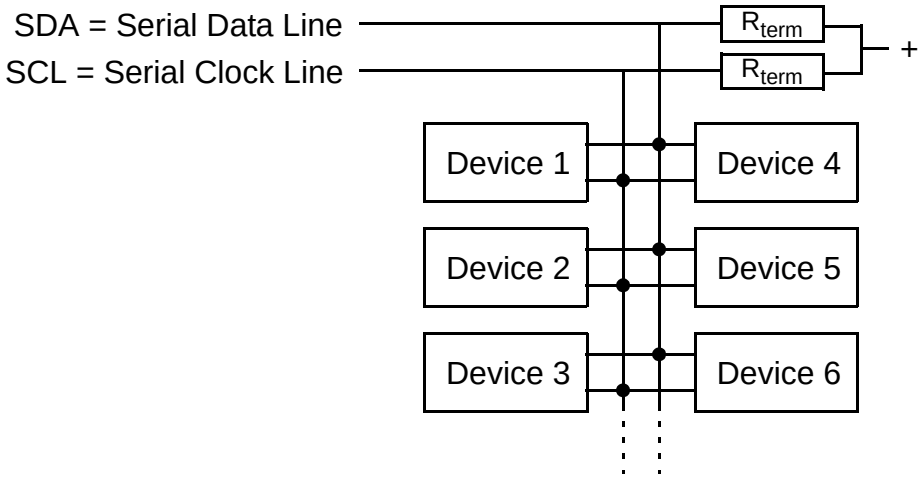


Figure-34: I2C Bus Architecture

As shown, the I2C bus is very simple, consisting of only 2 signal connections to each device. I2C is a two wire interface with one wire being a clock and the other a serial data line. Each Site Assembly Board contains an I2C interface to the DUT board.

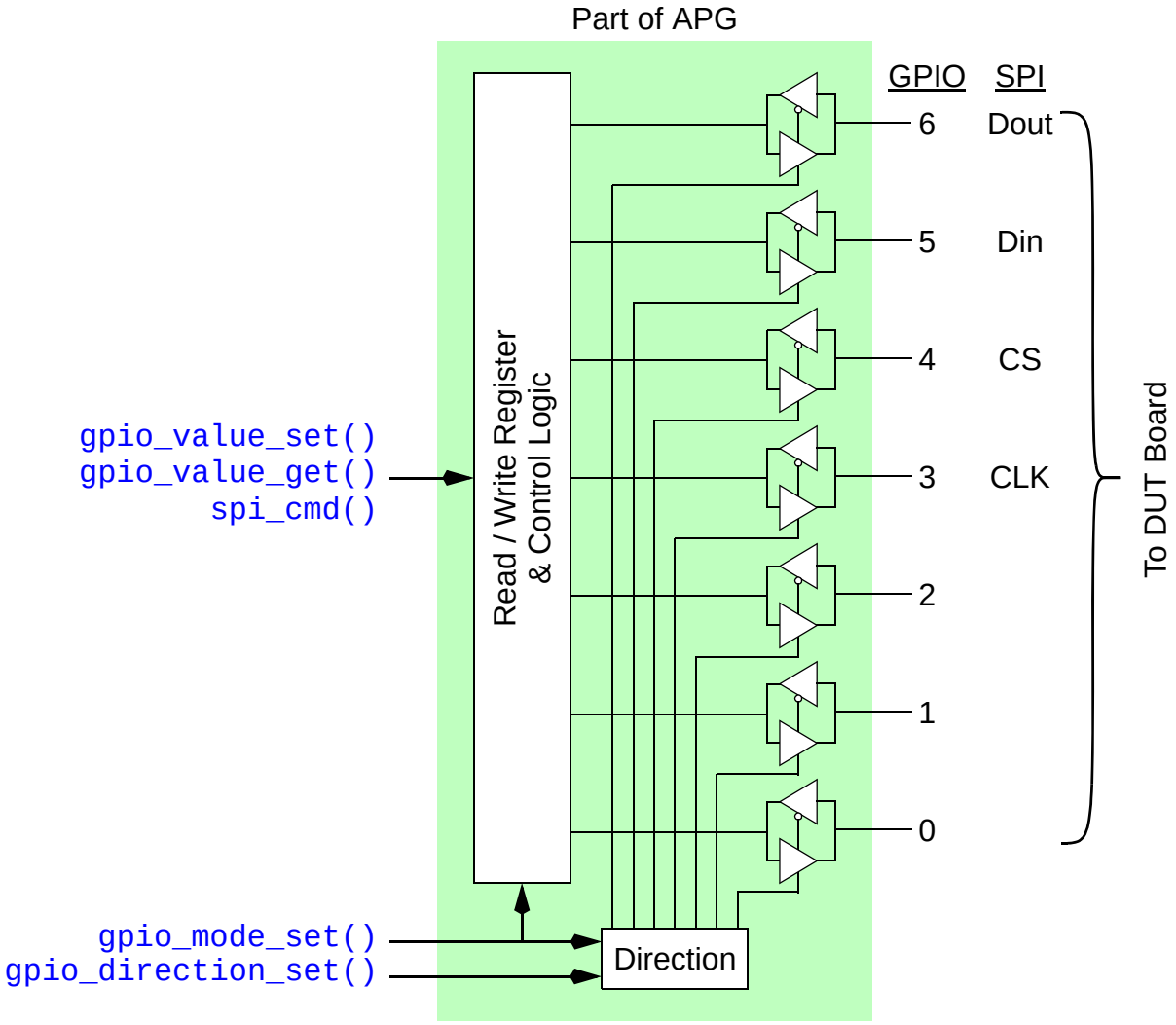
The I2C Bus Functions are used to read and/or write to devices on the I2C bus.

Note: device address 0x0 (Device 0) is reserved for Nextest use.

1.12.2 SPI Port & GPIO Port

See DUT Board I/O Ports, DUT Board I/O Port Functions.

The following diagram is used to describe the GPIO and SPI ports:



Note the following:

- The GPIO port consists of either 3 or 7 signals, as determined by the `gpio_mode_set()` function. In `t_parallel_io_mode` mode, all 7 signals are treated as GPIO signals. In `t_spi_mode` mode, the upper 4-bits are dedicated to the SPI port, leaving the low 3-bits for the GPIO port.
- GPIO signals are bi-directional, but the direction is set independent of the read/write data using the `gpio_direction_set()` function. In other words, to both read and write the GPIO port requires setting the direction to read, reading the data, then setting the direction to write, then writing the data. The read/write direction is a per-bit control, allowing some bits to be written while others are read, possibly in a dedicated configuration.
- The SPI port is accessed using the `spi_cmd()` function. A single execution of `spi_cmd()` can write, read, or write/read the SPI port. The I/O direction of the SPI pins is automatically set; i.e. the `gpio_direction_set()` function has no effect on pins in SPI mode.
- The GPIO/SPI pins are designed to drive high impedance, TTL levels (VIL <0.8V and VIH >2.0V). They are not specified to drive 3.3V CMOS levels. The GPIO/SPI pins are driven by a T.I. 74LVC1G125 IC, but each output is protected with a series 33 Ohm discrete resistor. The 74LVC1G125 uses 3.3V power. The 74LVC1G125 IC is capable of driving:
  - VOL 0.4V at 16mA or 0.55V at 24mA
  - VOH of 2.4V at -16mA or 2.3V at -24mA

  but the 33 Ohm series resistor can affect the output voltage dramatically (33mV per mA).
- A timing diagram is not defined because all bus transactions occur at computer speeds; i.e. relatively slowly as compared to the potential speed of the interface IC.

Chapter 2 Magnum 1, 2 & 2x Parallel Test

This section includes the following topics:

2.1 Overview

See Magnum 1, 2 & 2x Parallel Test.

The Magnum 1, 2 and 2x system software formally supports parallel test; i.e. concurrently testing multiple DUTs in parallel. This section provides an overview of key concepts and associated operational details.

Note: except as noted, all parallel test (multi-DUT) concepts apply to software executing in Site processes.

Except as noted below, a Multi-DUT Test Program is written as though one DUT is being tested, greatly simplifying parallel test implementation. Exceptions:

- The Pin Assignment Table, where the physical connections to each DUT are specified, identifies the maximum number of DUT(s) which can be tested by the test program, and the tester resources (pins, DPS, etc.) which are connected to each pin of each DUT.
- Standard Test Blocks are, by definition, parallel test blocks; i.e. the system software will test all enabled DUT(s) concurrently, in parallel; e.g. all DUT(s) in the Active DUTs Set (ADS). These test blocks are defined using the existing `TEST_BLOCK` macro. However, when test conditions exist which prohibit concurrently testing specific DUT(s) in parallel, a Sequential Test Block may be used. These test blocks are defined using the `TEST_BLOCK_SEQUENTIAL` macro and require that conflicting DUT(s) identified via a Conflict List.
- User designed hardware, typically on the DUT board, requires user-written control code. This code can benefit from accessing the Active DUTs Set (ADS), to identify which DUT(s) are enabled at any given time (more below).

The following key software features are used in support of Magnum 1, 2 and 2x Parallel Test Operation:

- The Active DUTs Set (ADS), or ADS
- The Ignored DUTs Set (IDS), or IDS
- Multi-DUT Test Results
- Sequential Test Blocks and Conflict Lists
- Parking Blocks

The Active DUTs Set (ADS) is the mechanism which the system software uses, during Sequence & Binning Table execution, to manage which DUT(s) are enabled at any given time; i.e. when some DUT(s) are being tested while others are disabled (parked). For example, when multiple DUTs are tested in parallel, it will be common for some to pass a given test block while others fail. When this occurs, the Sequence & Binning Table will continue to test some DUT(s), for example those that passed, while others are temporarily disabled, for example those that failed. In this context, ADS management is automatic, and optimized to reduce test time. The Magnum 1, 2 and 2x hardware design provides for disabling tester resources per-DUT.

User code may also manipulate the ADS, as required to read (get) a programmed value for a specific DUT (see Using Getter Functions) or, less often, to [temporarily] enable or disable hardware connected to specific DUT(s). Operational use of the ADS is described in Parallel Test Operation.

The Ignored DUTs Set (IDS) is the mechanism available for user code to advise the system software to ignore (don't enable) specific DUT(s). This supports, for example, the situation in which a DUT handler or prober is able to specify which test sites are active (or disabled), which can be affected by an empty input tray, a full output bin, edge of wafer, a known bad DUT socket, etc. The IDS may need to be updated for each start test signal, typically by user code included in the handler/prober control loop, executing in the Host process.

Operational use of the IDS is described in Parallel Test Operation. In general, ignored DUT(s) (i.e. those in the IDS) are not allowed in the Active DUTs Set (ADS).

In Multi-DUT Test Programs, individual test results must be continuously tracked for each DUT being tested; i.e. all DUT(s) which are not in the Ignored DUTs Set (IDS). As described in Multi-DUT Test Results, in simple cases this is can be automatic (no additional user code required), however in many programs user code will be needed. See Multi-DUT Test Results.

2.1.1 Multi-DUT Test Program

See Magnum 1, 2 & 2x Parallel Test.

A *Multi-DUT Test Program* is, by design, able to correctly and completely test multiple DUTs concurrently, in parallel.

The term *Multi-DUT Test Program* refers to a Magnum 1, 2 or 2x test program in which the Pin Assignment Table is defined using one of the multi-DUT macros; i.e. `ASSIGN_2DUT()`,

ASSIGN_32DUT(), etc. This includes the use of ASSIGN_1DUT(), even though the resulting test program will only support testing one DUT.

Conversely, when Pin Assignment Table is defined using the legacy ASSIGN() macro, the multi-DUT features described in this manual do not apply.

Test blocks in a Multi-DUT Test Program may only return the value MULTI_DUT, which causes the Sequence & Binning Table code to evaluate the correct test results.

The multi_dut_features() function can be used to determine if a given test program is a Multi-DUT Test Program.

2.1.2 Parallel Test Operation

See Magnum 1, 2 & 2x Parallel Test.

This section applies only in Multi-DUT Test Programs.

As described in Overview, parallel test operation utilizes the following key software features:

- The Active DUTs Set (ADS)
- The Ignored DUTs Set (IDS)
- Multi-DUT Test Results

Operational use of these features must be described in two contexts:

- Multi-DUT non-Sequence & Binning Table Operation
- Multi-DUT Sequence & Binning Table Operation

Multi-DUT non-Sequence & Binning Table Operation

This section describes how the ADS, IDS, and Multi-DUT Test Results operate when executing user code when the Sequence & Binning Table is not actively executing.

More specifically, these rules apply to user code executing in a Site process (the ADS and IDS only exists in the Site process) via User Tools, User Dialogs and User Variables any time the Sequence & Binning Table is NOT executing

- During initial program load, the IDS is set, by the system software, to contain zero DUTs; i.e. no DUTs are ignored. The system software does not otherwise modify the IDS; i.e. user code modifications to the IDS are persistent indefinitely.
- During initial program load, the ADS is set, by the system software, to contain all DUT(s) defined in the Pin Assignment Table.

- Operation of Nextest functions which are affected by the ADS is the same whether the Sequence & Binning Table is executing or not.
- At the end of each Sequence & Binning Table execution, the ADS is set to contain all DUT(s) which are NOT in the IDS. DUT(s) in the IDS can never be added to the ADS.
- User code modifications to the ADS when the Sequence & Binning Table is NOT executing are persistent until the next execution of the Sequence & Binning Table.
- Multi-DUT Test Results are correctly updated for all DUT(s) in the ADS by the Nextest test functions. If a Test Block is executed when the Sequence & Binning Table is NOT (i.e. `invoke(myTB)`) the rules specified in Multi-DUT Test Results still apply.
- The DUT Manager can be used to view and modify the ADS, IDS, and Multi-DUT Test Results.

Multi-DUT Sequence & Binning Table Operation

This section describes how the ADS, IDS, and Multi-DUT Test Results operate when executing user code when the Sequence & Binning Table IS actively executing.

More specifically, these rules apply to user code executing in a Site process (the ADS and IDS only exists in the Site process) via Test Blocks, User Tools, User Dialogs and User Variables, any time the Sequence & Binning Table IS executing

- During initial program load, the IDS is set, by the system software, to contain zero DUTs; i.e. no DUTs are ignored. The system software does not otherwise modify the IDS; i.e. user code modifications to the IDS are persistent indefinitely.
- At the start of Sequence & Binning Table execution, the ADS is initialized to include all DUTs which are not in the IDS. This means that all DUTs not in the IDS will be tested by the current execution of the Sequence & Binning Table. But, user code can affect this (more below).
- The previous statement also applies when using UI to execute one test block or execute starting with a selected test block.

Note: it is highly recommended that changes to the Ignored DUTs Set (IDS) be made ***before*** the Sequence & Binning Table executes the first test block; i.e. in a Before-testing Block. This ensures that the Sequence & Binning Table correctly and completely tests all DUT(s) enabled (not ignored) at the start of each execution. The IDS is set or modified using `ignored_duts_enable()` and `ignored_duts_disable()`.

- During Sequence & Binning Table execution, the system software will manage (modify) the ADS as needed to control which DUT(s) are enabled at any given time. Details follow:
- On entry to a given test block, any DUT(s) which are not to be tested will be parked. See Active DUTs Set (ADS) and Parking Blocks (and, more below).
- On entry to a given test block, the system software will set the ADS to enable only those DUT(s) to be tested by that test block. The system software does not otherwise modify the ADS during the execution of a given test block.
- In standard test blocks, the test block code will execute once, regardless of how many DUT(s) are in the ADS. This tests all DUT(s) in the ADS concurrently, in parallel. User code within the test block may change this behavior (more below) but in many applications this will not be necessary.
- Sequential Test Blocks provide for situations in which not all DUT(s) in the ADS can be tested concurrently. User code must identify conflicting DUT(s) (see Conflict List), then a Sequential Test Block code will execute two or more times, during which the system software will manipulate the ADS to control which DUT(s) are to be tested for each execution. During each execution, those DUT(s) which are waiting to be tested are parked.
- When test block execution exits, the Sequence & Binning Table code retrieves the Multi-DUT Test Results for those DUT(s) in the ADS at the start of block execution, to determine how testing proceeds. Note that Multi-DUT Test Results are NOT the value returned via the test block `return` statement.
- When test block execution returns to the Sequence & Binning Table, before proceeding to the next step, the algorithm determines which DUT(s) are to be in the ADS (enabled) for the next test block. This can cause the following additional actions:
  - If any DUT(s) need to be parked and the test block in which they are to be tested next contains a user specified Parking Block, the ADS is temporarily set to contain these DUT(s) and the Parking Block is executed. Then the default parking operations are executed; basically disabling the tester hardware connected to these DUT(s). See Active DUTs Set (ADS) for a detailed example.
  - If any DUT(s) need to be un-parked, the default un-parking operations are executed first; basically enabling the tester hardware connected to these DUT(s). Then, the system software determines whether any DUT(s) were originally parked by a user specified Parking Block and, if so, the ADS is set to these DUT(s) and that Parking Block is executed to un-park those DUT(s).

  - On entry to a Parking Block, the ADS includes only those DUT(s) which are to be parked or un-parked. When parking DUT(s), any which were previously parked (and are to remain parked) will not be included in the ADS. Within the Parking Block, the built-in variable, named `parking`, will be set by the system software to allow user code to distinguish between parking (`parking` = TRUE) and un-parking (`parking` = FALSE).
- Once any DUT(s) are tested to completion, the Sequence & Binning Table will continue testing any remaining DUT(s) which were previously parked. Testing continues until all DUTs which are not in the Ignored DUTs Set (IDS) have reached a STOP statement in the Sequence & Binning Table.
- The After-testing Block will execute only after all DUT(s) have reached `STOP` in the Sequence & Binning Table.
- As noted above, user code can modify the ADS, using `active_duts_disable()` and `active_duts_enable()`. Using these functions, changes to the ADS are scoped to the currently executing Test Block, Test Bin, or Parking Blocks, i.e. once the block execution exits the system software sets the ADS as needed to properly manage testing. This allows user code to change the ADS within the block, as needed, without affecting the operation of the Sequence & Binning Table, which uses the ADS to completely and correctly test multiple DUTs. The current ADS can be copied using `active_duts_get()` (see ADS Save/Modify/Restore Example).
- It is possible for user code to modify the Ignored DUTs Set (IDS) during Sequence & Binning Table execution. However, this is ***NOT RECOMMENDED***. Once a DUT is added to the IDS it totally ignored by the Sequence & Binning Table algorithm; i.e. it will not be completely tested, it will never reach a STOP, it will not have a bin assigned, etc. See Note:.

In Multi-DUT Test Program, the Sequence & Binning Table algorithm is optimized to reduce the overall test time of all DUTs. In general, the number of times a given test block is executed is minimized and the number of times DUT(s) are parked is minimized. For example, when a test flow branches and merges (for example TB2, TB6, and TB3 in the Example Test Flow Diagram) the algorithm is optimized to know which test block(s) should execute first. In the Example Test Flow Diagram, the algorithm will execute TB6 before TB3 because any DUT(s) which pass TB6 must be tested in TB3. This warrants the following observations:

- Any given DUT will transit the Sequence & Binning Table the same as if it was being tested alone.
- When multiple DUTs are being tested, the amount of time *between* test blocks, as seen by any given DUT, may be different based on the test results of other DUT(s) being tested. In most applications, this is not significant. When it is, the user must

design the Sequence & Binning Table to ensure test flow execution cannot vary between the critical test blocks (contiguous test blocks with no exit branches will always execute with the same relative real-time performance).

- The optimum benefit of parallel testing occurs when all DUTs pass all tests identically or fail all tests identically and no Sequential Test Blocks are used. Conversely, as the need to park DUT(s) increases, or Sequential Test Blocks execute multiple times, the benefit of parallel test is reduced. This is true on all test equipment.

2.1.3 Using Getter Functions

See Multi-DUT Test Program, Active DUTs Set (ADS), Ignored DUTs Set (IDS).

In Multi-DUT Test Programs, the Active DUTs Set (ADS) and Ignored DUTs Set (IDS) both affect the operation of Nextest *getter* functions.

In general, a getter function is one which returns a value. When the value being returned is a per-pin attribute, for example a drive voltage, PTU current, etc., the associated getter function requires that a `DutPin` be specified, to identify the target pin. However, in Multi-DUT Test Programs, the same `DutPin` will exist for each DUT, thus creating an ambiguity. In this situation, the first DUT in the Active DUTs Set (ADS) determines which value is returned. And, DUT(s) in the Ignored DUTs Set (IDS) cannot be in the Active DUTs Set (ADS).

In order to get a per-pin value from a DUT which is not currently first in the Active DUTs Set (ADS), user code must modify the Active DUTs Set (ADS).

Note: any time user code modifies the Active DUTs Set (ADS), user code becomes responsible for all subsequent operations which depend on the Active DUTs Set (ADS), for the duration of the current Test Block, Test Bin or Parking Blocks.

To simplify this, the following steps should be considered any time user code must modify the Active DUTs Set (ADS):

- Save the current state of the Active DUTs Set (ADS), using `active_duts_get()`.
- Modify the Active DUTs Set (ADS) as needed, using `active_duts_enable()` and/or `active_duts_disable()`.
- Use the modified Active DUTs Set (ADS) as needed.
- Repeat the previous 2 steps as needed.

- Restore the Active DUTs Set (ADS) to that saved earlier, using `active_duts_enable()`.

An example of this code is available in ADS Save/Modify/Restore Example.

2.2 Types, Enums, etc.

See Magnum 1, 2 & 2x Parallel Test.

Description

The following enumerated types are used in Multi-DUT Test Programs:

Usage

The `DutNum` enumerated type is used in Multi-DUT Test Programs to identify a specific DUT. These values are used when accessing the Active DUTs Set (ADS) or Ignored DUTs Set (IDS).

```
enum DutNum {
   t_dut1, t_dut2, t_dut3, t_dut4,
   t_dut5, t_dut6, t_dut7, t_dut8,
   ... snip ...
   t_dut125, t_dut126, t_dut127, t_dut128,
   t_dut_na };
```

`DutNumArray` is a C++ data type used for defining arrays of `DutNum`s; i.e. arrays of DUTs. This is the base data type of the Active DUTs Set (ADS) and Ignored DUTs Set (IDS):

```
typedef CArray< DutNum, DutNum > DutNumArray;
```

2.3 Active DUTs Set (ADS)

See Magnum 1, 2 & 2x Parallel Test, Overview, Ignored DUTs Set (IDS).

Description

The Magnum 1, 2 and 2x system software formally supports parallel test applications. A key component of this support is the *Active DUTs Set*, or ADS. See Overview and Magnum 1, 2 & 2x Parallel Test.

Note the following:

- The Magnum 1, 2 and 2x architecture includes hardware to enable and disable tester resources on a per-DUT basis. In software, it is the Active DUTs Set (ADS) which controls this.
- In simple terms, the Active DUTs Set (ADS) is the mechanism which the system software uses to manage which DUT(s) are enabled at any given time; i.e. when some DUT(s) are being tested while others are idle (parked). This is explained in more detail below.
- There is only one Active DUTs Set (ADS), established by the system software.
- At any given time, the contents (members) of the ADS consist of one or more DUT(s) (`DutNum` = `t_dut1`, etc.) up to the total number of DUTs defined in the Pin Assignment Table.
- The ADS cannot contain DUT(s) which are in the Ignored DUTs Set (IDS).
- The ADS only exists in the Site process(es).

The Active DUTs Set (ADS) has several usage contexts:

- During Sequence & Binning Table execution, within a given Test Block or Test Bin the ADS identifies those DUT(s) which are currently enabled.
- During Sequence & Binning Table execution, within a given Parking Block the ADS identifies those DUT(s) which are being parked or un-parked. See Magnum 1, 2 & 2x Parallel Test and Parking Blocks.
- During Sequence & Binning Table execution user code in a Test Block, Test Bin or Parking Block, may manipulate the Active DUTs Set (ADS), to locally change which DUT(s) are enabled at the time, and thus which will be affected by subsequent execution of certain Nextest API functions (set or get voltages, timing, power supplies, etc.), execute tests, etc. Most of the Nextest *getter* functions; i.e. functions which return a value, return a single value, which will be for the first DUT in the Active DUTs Set (ADS). In order to get a value for a different DUT, user code must manipulate the Active DUTs Set (ADS). This will typically require a save, modify, restore sequence. See ADS Save/Modify/Restore Example.

Note: the need to save/restore the Active DUTs Set (ADS) is critical to proper test results any time user code modifies the ADS during Sequence & Binning Table execution.

- During non-Sequence & Binning Table execution the Active DUTs Set (ADS) still determines which DUT(s) are enabled which, by default, will be all DUT(s) not in the Ignored DUTs Set (IDS). This will affect User Tools, User Dialogs and User Variables executed in a Site process when the Sequence & Binning Table is not executing.

The following example test flow is used to further explain the application of the Active DUTs Set (ADS) and the use of Parking Blocks:

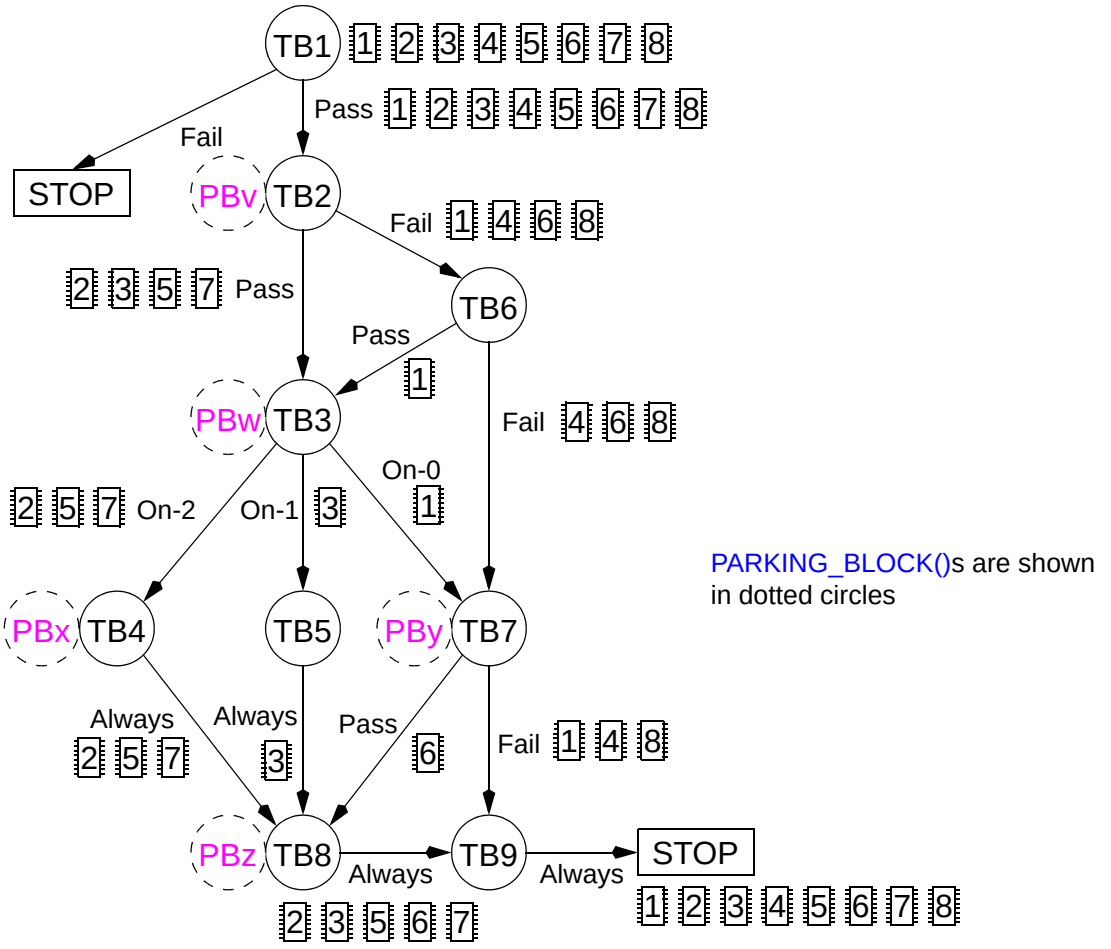


Figure-35: Example Test Flow Diagram

For reference, the following Sequence & Binning Table implements the diagram above:

```
TESTL2(   L1,  TB1, L2,  STOP           )
TESTL2P(  L2,  TB2, L3,  L6,       PBv  )
TESTL3P(  L3,  TB3, L4,  L5,  L7,  PBw  )
TESTL1P(  L4,  TB4, L8,            PBx  )
TESTL1(   L5,  TB5, L8                  )
TESTL2(   L6,  TB6, L3,  L7             )
TESTL2P(  L7,  TB7, L8,  L9,       PBy  )
TESTL1P(  L8,  TB8, NEXT           PBz  )
TESTL1(   L9,  TB9, STOP                )
```

A test program containing this example generated the following output, given the pass/fail results shown above:

```
TestBlock => TB1 : active DUTs => t_dut1 t_dut2 t_dut3 t_dut4 t_dut5 t_dut6 t_dut7 t_dut8
TestBlock => TB2 : active DUTs => t_dut1 t_dut2 t_dut3 t_dut4 t_dut5 t_dut6 t_dut7 t_dut8
PBlock (PBw) => Parking => t_dut2 t_dut3 t_dut5 t_dut7
TestBlock => TB6 : active DUTs => t_dut1 t_dut4 t_dut6 t_dut8
PBlock (PBy) => Parking => t_dut4 t_dut6 t_dut8
PBlock (PBw) => Unparking => t_dut2 t_dut3 t_dut5 t_dut7
TestBlock => TB3 : active DUTs => t_dut1 t_dut2 t_dut3 t_dut5 t_dut7
PBlock (PBx) => Parking => t_dut2 t_dut5 t_dut7
PBlock (PBy) => Unparking => t_dut4 t_dut6 t_dut8
TestBlock => TB7 : active DUTs => t_dut1 t_dut4 t_dut6 t_dut8
PBlock (PBz) => Parking => t_dut6
PBlock (PBx) => Unparking => t_dut2 t_dut5 t_dut7
TestBlock => TB4 : active DUTs => t_dut2 t_dut5 t_dut7
PBlock (PBz) => Parking => t_dut2 t_dut5 t_dut7
TestBlock => TB5 : active DUTs => t_dut3
PBlock (PBz) => Unparking => t_dut2 t_dut5 t_dut6 t_dut7
TestBlock => TB8 : active DUTs => t_dut2 t_dut3 t_dut5 t_dut6 t_dut7
TestBlock => TB9 : active DUTs => t_dut1 t_dut2 t_dut3 t_dut4 t_dut5 t_dut6 t_dut7 t_dut8
```

Note the following:

- In this example, the Ignored DUTs Set (IDS) contains no (zero) DUTs. This means that all DUTs defined in the Pin Assignment Table will be tested by the Sequence & Binning Table. Thus, when the Sequence & Binning Table table is executed, the system software will set the Active DUTs Set (ADS) to include all 8 DUTs (`t_dut1`, `t_dut2`... `t_dut8`), and test block 1 (TB1) will test all 8 DUTs.
- Any DUT(s) which fail TB1 will STOP and the system software will remove them from the ADS for the duration of the current Sequence & Binning Table execution. This is consistent with a continuity test executed at the start of the test flow. In this example, all 8 DUTs passed TB1 and will next be tested next in TB2.

- In TB2, some DUTs fail and some pass, thus the test flow must branch at this point. The Sequence & Binning Table will determine which test block to execute next, identify which DUTs must be enabled in that test block, and identify which DUTs must be disabled; i.e. parked (more below). A record is kept of these decisions to ensure that those DUT(s) which are parked will be correctly tested later. In this example, either TB3 or TB6 could be executed next. But...
- TB6 will be scheduled to execute next, not TB3. This is because any DUT(s) which pass TB6 will need to be tested in TB3 and the Sequence & Binning Table algorithm is smart enought to schedule TB6 first. This makes it necessary to park `t_dut2`, `t_dut3`, `t_dut5`, `t_dut7` while TB6 executes.
- During the execution of a given Test Block, those DUTs which are not in the ADS (and IDS) are, by definition, *parked*. The built-in park mechanism ensures that these DUTs will not see any changes in hardware state, will not receive any test stimulus, and cannot effect test PASS/FAIL results. However, ...
- For those occasions when the built-in park mechanism is inadequate, each Sequence & Binning Table statement may optionally specify a Parking Block. For example:

```
TESTL2P( L2, TB2, L3, L6, PBn )
```

- The trailing **P** in the macro name indicates that a Parking Block is being specified, using the last parameter. The example above uses 5 Parking Blocks, named PBv, PBw, ... PBz.
- A Parking Block allows user code to execute as part of the parking mechanism. This code can be used to set special voltages, power-down DUTs being parked, power-up DUTs being unparked, etc. See Parking Block for examples.
- A given Sequence & Binning Table statement identifies one Test Block which will execute when the test flow reaches that statement. If the statement also contains a Parking Block, it will be executed any time one or more DUT(s) which are to execute the specified Test Block are parked while a different Test Block is executed first. Conversely, when execution finally does reach the Sequence & Binning Table statement any DUTs which were previously parked by that statement's Parking Block will be un-parked by executing that Parking Block again. To allow user code to distinguish parking from un-parking, within the scope of a Parking Block a built-in BOOL variable named `parking` exists, and can be tested: TRUE = parking, FALSE = un-parking. See Parking Block examples.
- In the example above, since TB6 will execute before TB3, any DUTs which passed TB2 must be parked. And, since the Sequence & Binning Table statement which executes TB3 also includes a user specified Parking Block (PBw), before TB6 is executed, the system software sets the ADS to include `t_dut2`, `t_dut3`, `t_dut5`,

t_dut7 and the Parking Block named PBw is executed. The system software remembers that these DUTs were parked using PBw and later will execute PBw again to un-park these DUTs, just before TB3 executes.

- Next the system software sets the ADS to contain t_dut1, t_dut4, t_dut6 and t_dut8 and executes TB6. In this example, one DUT (t_dut1) passes TB6 thus the Sequence & Binning Table algorithm will schedule TB3 to execute before TB7, because some DUTs exiting TB3 may need to be tested by TB7; again the Sequence & Binning Table algorithm is smart, and schedules TB3 first.
- And, since the Sequence & Binning Table statement which executes TB7 also includes a user specified Parking Block (PBy) it will be executed with the ADS set to park t_dut4, t_dut6, and t_dut8. This occurs before TB3 is executed.
- Before executing TB3, it is necessary to un-park t_dut2, t_dut3, t_dut5, t_dut7. And, since the Sequence & Binning Table statement for TB3 contained a Parking Blocks (PBw) it will be executed again, with the ADS set to contain t_dut2, t_dut3, t_dut5, t_dut7. During this execution the parking variable will be FALSE.
- During TB3, the ADS is set to contain t_dut1, t_dut2, t_dut3, t_dut5 and t_dut7.
- This process repeats until all DUTs are tested and each has reached a STOP or STOPL operation.
- In this example, note that the Parking Block specified for TB2 (PBv) will never be executed. This is because no DUTs are ever parked before executing TB2.
- Also note that some Sequence & Binning Table statements don't include a Parking Block. This means that any DUTs which need to be parked before these tests will only be affected by the normal parking actions performed by the system software.
- Test Bin body code operates the same as Test Blocks; i.e. the ADS is set by the Sequence & Binning Table to identify which DUT(s) are active during the Test Bin execution.

The information above addresses the general problem of managing how DUTs are tested in parallel. However, in operation, the Active DUTs Set (ADS) actually determines (limits) which hardware resources are affected when executing many Nextest API functions. For example, executing vil(800 MV) to change a drive level will affect only those pin(s) which are connected to DUT(s) in the Active DUTs Set (ADS) at the time the function executes. Similarly, when executing funtest(), the drive, I/O, and strobe signals are only applied to DUT(s) in the Active DUTs Set (ADS). This raises the following issues:

- In most applications, program operation will be as desired; i.e. user code changes to the Active DUTs Set (ADS) will rarely be needed.

- When user code does explicitly manipulate the Active DUTs Set (ADS), it is then responsible for correct operation of all subsequent code within a given Test Block, Test Bin or Parking Block. See ADS Save/Modify/Restore Example.
- On exit from a Test Block, Test Bin or Parking Block, the system software sets the Active DUTs Set (ADS) as needed to correctly continue testing, as directed by the Sequence & Binning Table. This means that the scope of user changes to the ADS is limited to the test block.
- In Multi-DUT Test Programs, Pin Lists contain an element for every DUT defined in the Pin Assignment Table. When the pin list is used to set a parameter the Active DUTs Set (ADS) determines which pin(s) are actually affected. See Pin Lists.

2.3.1 ADS Save/Modify/Restore Example

See Magnum 1, 2 & 2x Parallel Test, Active DUTs Set (ADS), Ignored DUTs Set (IDS).

The following example is typical of that needed when user code must temporarily modify the Active DUTs Set (ADS). In this example, the `vil()` function is used to represent a Nextest *getter* function which returns a single value, from the first DUT in the Active DUTs Set (ADS). This example shows how the ADS is saved, temporarily modified, and restored:

```
DutNumArray ADSsave;

// Save the current ADS
int count = active_duts_get( &ADSsave );

// Set t_dut3 as the only DUT in the ADS
if( active_duts_enable( t_dut3 ) == FALSE)
   output(" ERROR: attempt to enable invalid DUT");

else // Use modified ADS to get vil() from pin D0 of t_dut3
   double v = vil( D0 );

// Restore ADS to continue normal testing
active_duts_enable( ADSsave );
```

Note: the need to save/restore the Active DUTs Set (ADS) is critical to proper test results any time user code modifies the ADS from within a Test Block, Test Bin or Parking Block.

2.3.2 active_duts_enable()

See Magnum 1, 2 & 2x Parallel Test, Active DUTs Set (ADS), Ignored DUTs Set (IDS).

Description

The `active_duts_enable()` function is used to completely define the Active DUTs Set (ADS) or add DUT(s) to the Active DUTs Set (ADS). Note the following:

- The Active DUTs Set (ADS) cannot contain DUT(s) which are in the Ignored DUTs Set (IDS). `active_duts_enable()` will return FALSE when attempting to add DUT(s) which are in the Ignored DUTs Set (IDS).
- The effect of using `active_duts_enable()` is limited (scoped) based on where the function is executed:
  - If executed in a Test Block, Test Bin or Parking Block the scope is limited to that test block.
  - If executed from outside Sequence & Binning Table (i.e. User Variables, etc.), the changes are persistent until the next Sequence & Binning Table executes, at which time the system software sets the Active DUTs Set (ADS) to completely and correctly test all DUT(s) which are not in the Ignored DUTs Set (IDS).
- The current Active DUTs Set (ADS) can be copied using `active_duts_get()`.
- Information about which DUT(s) are currently in the Active DUTs Set (ADS) can be obtained using `active_duts_get()`.
- One or more DUT(s) can be removed from the Active DUTs Set (ADS) using `active_duts_disable()`.

Several versions of `active_duts_enable()` are provided:

- The version of `active_duts_enable()` which takes the `DutNum` argument allows a single DUT to be specified.
- The version of `active_duts_enable()` which takes the `DutNumArray` argument allows multiple DUT(s) to be specified.
- The version of `active_duts_enable()` which takes the `mask` argument allows multiple DUT(s) to be specified using a bitmask to identify which DUT(s) are to be included.
- All versions can either completely redefine or modify the Active DUTs Set (ADS), as controlled by the `incremental` argument. See Usage.

Usage

The following functions are used to completely redefine or incrementally add DUT(s) to the Active DUTs Set (ADS):

```
BOOL active_duts_enable( DutNum dut,
                         BOOL incremental DEFAULT_VALUE( FALSE ) );
BOOL active_duts_enable( DutNumArray &duts,
                         BOOL incremental DEFAULT_VALUE( FALSE ) );
BOOL active_duts_enable( DWORD mask,
                         BOOL incremental DEFAULT_VALUE( FALSE ) );
```

where:

dut identifies one DUT which:

- When **incremental** = TRUE **dut** is added to the Active DUTs Set (ADS).
- When **incremental** = FALSE completely redefines the Active DUTs Set (ADS).

duts is a DutNumArray identifying zero or more DUTs, which:

- When **incremental** = TRUE **duts** are added to the Active DUTs Set (ADS).
- When **incremental** = FALSE completely redefines the Active DUTs Set (ADS).

incremental is optional, and if specified determines whether the current Active DUTs Set (ADS) will be modified (TRUE) or completely redefined (FALSE). Default = FALSE.

mask is a bitmask identifying zero or more DUTs, which:

- When **incremental** = TRUE DUT(s) are added to the Active DUTs Set (ADS).
- When **incremental** = FALSE completely redefines the Active DUTs Set (ADS).

In the bitmask, a 1 identifies a DUT to be added and 0 identifies a DUT which will not be added. See Examples.

active_duts_enable() returns FALSE if an invalid DUT is specified (i.e. a DUT which is not defined in the Pin Assignment Table) or if a DUT in the Ignored DUTs Set (IDS) is specified.

Example

The following example redefines the Active DUTs Set (ADS) to contain one DUT (t_dut3). If the Pin Assignment Table does not define at least 3 DUTs, active_duts_enable() will return FALSE:

```
BOOL ok = active_duts_enable( t_dut3 );
```

The following example adds one DUT (`t_dut4`) to the Active DUTs Set (ADS). If the Pin Assignment Table does not define at least 4 DUTs, `active_duts_enable()` will return FALSE:

```
BOOL ok = active_duts_enable( t_dut4, TRUE );
```

The following example completely redefines the Active DUTs Set (ADS) to contain 2 DUTs. If the Pin Assignment Table does not define at least 8 DUTs, `active_duts_enable()` will return FALSE:

```
DutNumArray enable_list;
enable_list.Add( t_dut4 );
enable_list.Add( t_dut8 );
BOOL ok = active_duts_enable( enable_list );
```

The following example adds 3 DUTs (`t_dut6`, `t_dut4`, `t_dut2`) to the Active DUTs Set (ADS):

```
BOOL ok = active_duts_enable( 0x2A, TRUE );
```

The following example redefines the Active DUTs Set (ADS) to include only 3 DUTs (`t_dut5`, `t_dut3`, `t_dut1`):

```
BOOL ok = active_duts_enable( 0x15 );
```

2.3.3 active_duts_disable()

See Magnum 1, 2 & 2x Parallel Test, Active DUTs Set (ADS), Ignored DUTs Set (IDS).

Description

The `active_duts_disable()` function is used to completely define the Active DUTs Set (ADS) or remove DUT(s) from the Active DUTs Set (ADS). Note the following:

- The Active DUTs Set (ADS) cannot contain DUT(s) which are in the Ignored DUTs Set (IDS). Using `active_duts_disable()` to remove a DUT in the IDS is silently ignored.
- The effect of using `active_duts_disable()` is limited (scoped) based on where the function is executed:
  - If executed in a Test Block, Test Bin or Parking Block the scope is limited to that test block.

  - If executed from outside Sequence & Binning Table (i.e. User Variables, etc.), the changes are persistent until the next Sequence & Binning Table executes, at which time the system software sets the Active DUTs Set (ADS) to completely and correctly test all DUT(s) which are not in the Ignored DUTs Set (IDS).
- The current Active DUTs Set (ADS) can be copied using `active_duts_get()`.
- One or more DUT(s) can be added to the Active DUTs Set (ADS) using `active_duts_enable()`.

Several versions of `active_duts_disable()` are provided:

- The version of `active_duts_disable()` which takes the `DutNum` argument allows a single DUT to be specified.
- The version of `active_duts_disable()` which takes the `DutNumArray` argument allows multiple DUT(s) to be specified.
- The version of `active_duts_disable()` which takes the `mask` argument allows multiple DUT(s) to be specified using a bitmask to identify which DUT(s) are to be removed.
- All versions can either completely redefine or modify the Active DUTs Set (ADS), as controlled by the `incremental` argument. See Usage.

Usage

The following functions are used to completely redefine or incrementally remove DUT(s) from the Active DUTs Set (ADS):

```
BOOL active_duts_disable( DutNum dut,
                          BOOL incremental DEFAULT_VALUE( FALSE ) );
BOOL active_duts_disable( DutNumArray &duts,
                          BOOL incremental DEFAULT_VALUE( FALSE ) );
BOOL active_duts_disable( DWORD mask,
                          BOOL incremental DEFAULT_VALUE( FALSE ) );
```

where:

`dut` identifies one DUT which:

- When **`incremental`** = TRUE is removed from the Active DUTs Set (ADS).
- When **`incremental`** = FALSE the Active DUTs Set (ADS) is set to include all DUT(s) not in the Ignored DUTs Set (IDS), then **`dut`** is removed from the Active DUTs Set (ADS).

`duts` is a `DutNumArray` identifying zero or more DUTs, which:

- When **incremental** = TRUE will be removed from the Active DUTs Set (ADS).
- When **incremental** = FALSE the Active DUTs Set (ADS) is set to include all DUT(s) not in the Ignored DUTs Set (IDS), then **duts** are removed from the Active DUTs Set (ADS).

incremental is optional, and if specified determines whether the current Active DUTs Set (ADS) will be modified (TRUE) or completely redefined (FALSE). Default = FALSE.

mask is a bitmask identifying zero or more DUTs, which:

- When **incremental** = TRUE are removed from the Active DUTs Set (ADS).
- When **incremental** = FALSE completely redefines the Active DUTs Set (ADS).

In the bitmask, a 1 identifies a DUT to be removed and 0 identifies a DUT which will not be removed. See Examples.

`active_duts_disable()` returns FALSE if an invalid DUT is specified; i.e. a DUT which is not defined in the Pin Assignment Table. `active_duts_disable()` returns TRUE if a DUT in the Ignored DUTs Set (IDS) is specified.

Example

The following example redefines the Active DUTs Set (ADS) to contain all non-ignored DUTs except one (`t_dut3`). If the Pin Assignment Table does not define at least 3 DUTs, `active_duts_disable()` will return FALSE:

```
BOOL ok = active_duts_disable( t_dut3 );
```

The following example removes one DUT (`t_dut4`) from the Active DUTs Set (ADS). If the Pin Assignment Table does not define at least 4 DUTs, `active_duts_disable()` will return FALSE:

```
BOOL ok = active_duts_disable( t_dut4, TRUE );
```

The following example completely redefines the Active DUTs Set (ADS) to contain all non-ignored DUTs except two (`t_dut4` and `t_dut8`). If the Pin Assignment Table does not define at least 8 DUTs, `active_duts_disable()` will return FALSE:

```
DutNumArray disable_list;
disable_list.Add( t_dut4 );
disable_list.Add( t_dut8 );
BOOL ok = active_duts_disable( disable_list );
```

The following example removes 3 DUTs (`t_dut6`, `t_dut4`, `t_dut2`) from the Active DUTs Set (ADS):

```
BOOL ok = active_duts_disable( 0x2A );
```

The following example redefines the Active DUTs Set (ADS) to include only 5 DUTs (`t_dut8`, `t_dut7`, `t_dut6`, `t_dut4`, `t_dut2`), assuming that 8 DUTs are defined in the Pin Assignment Table:

```
BOOL ok = active_duts_disable( 0x15, TRUE);
```

2.3.4 active_dut_get()

See Magnum 1, 2 & 2x Parallel Test, Active DUTs Set (ADS), Ignored DUTs Set (IDS).

Description

The `active_dut_get()` function is used to get the first DUT in the Active DUTs Set (ADS).

Usage

```
DutNum active_dut_get();
```

where:

`active_dut_get()` returns the first DUT in the Active DUTs Set (ADS).

Example

```
DutNum d = active_dut_get();
```

2.3.5 active_duts_get()

See Magnum 1, 2 & 2x Parallel Test, Active DUTs Set (ADS), Ignored DUTs Set (IDS).

Description

The `active_duts_get()` function is used to get information about which DUT(s) are currently in the Active DUTs Set (ADS).

`ignored_duts_get()` may be used to access the Ignored DUTs Set (IDS).

Usage

The following function determines whether the specified DUT is currently in the Active DUTs Set (ADS):

```
BOOL active_duts_get( DutNum dut );
```

The following function returns an array containing the DUT(s) which are currently in the Active DUTs Set (ADS):

```
int active_duts_get( DutNumArray *duts );
```

The following function returns a bitmask which indicates which DUT(s) are currently in the Active DUTs Set (ADS):

```
int active_duts_get( DWORD *mask );
```

where:

dut identifies one DUT (DutNum) to be checked for inclusion in the Active DUTs Set (ADS).

duts is a pointer to an existing DutNumArray used to return a copy of the Active DUTs Set (ADS).

mask is a pointer to an existing DWORD used to return a bitmask of the Active DUTs Set (ADS). In the returned bitmask, a 1 identifies a DUT which is in the Active DUTs Set (ADS) and 0 identifies a DUT which is not in the Active DUTs Set (ADS).

The version of active_duts_get() which returns BOOL, returns TRUE if the specified **dut** is in the Active DUTs Set (ADS), otherwise FALSE is returned.

The versions of active_duts_get() which returns int, return the number of DUT(s) which are in the Active DUTs Set (ADS).

Example

The following example checks if t_dut2 is currently in the Active DUTs Set (ADS):

```
if( active_duts_get( t_dut2 ) )
   output(" t_dut2 IS in the ADS");
else
   output(" t_dut2 is NOT in the ADS");
```

The following example returns an array containing those DUT(s) currently in the Active DUTs Set (ADS):

```
DutNumArray ADS_copy;
int count = active_duts_get( &ADS_copy );
```

The following example returns a bitmask indicating which DUT(s) are currently in the Active DUTs Set (ADS) and, using the mask, checks to see if `t_dut3` is in the Active DUTs Set (ADS):

```
DWORD mask;
int count = active_duts_get( &mask );
if( mask && 0x4 )
   output(" t_dut3 IS in the ADS");
else
   output(" t_dut3 is NOT in the ADS");
```

2.3.6 max_dut()

See Pin Assignment Table.

Description

The `max_dut()` function is used in Multi-DUT Test Programs to determine the number of DUTs which are defined in the Pin Assignment Table. Also see `multi_dut_features()`. The value returned is not affected by the Active DUTs Set (ADS) or Ignored DUTs Set (IDS).

Usage

```
DutNum max_dut();
```

where:

`max_dut()` returns the `DutNum` of the highest DUT specified in the Pin Assignment Table. This is a zero based value; i.e. 0 is returned when only 1 DUT is supported.

Examples

```
DutNum d = max_dut();
output("%d DUTs are specified in the Pin Assignments Table", d+1 );
for( DutNum dn = t_dut1; dn <= max_dut(); ++dn ) {
   // Do something per DUT
}
```

2.3.7 multi_dut_features()

See Pin Assignment Table.

Description

The `multi_dut_features()` function is used to determine if the test program is a Multi-DUT Test Program.

Usage

```
BOOL multi_dut_features();
```

where:

`multi_dut_features()` returns TRUE if the test program is a Multi-DUT Test Program, otherwise FALSE is returned.

Example

```
LPCTSTR s = multi_dut_features() ? "IS" : "Is NOT";
output(" This program %s a Multi-DUT Test Program", s);
```

2.3.8 Active DUTs Set Iterators

See Magnum 1, 2 & 2x Parallel Test, Active DUTs Set (ADS), Ignored DUTs Set (IDS).

Note: `SoftwareOnlyActiveDutIterator` was first available in software release h1.1.23. Additional iterator capabilities were added in software release h1.1.23. A new overload of the `More()` member function was added in software release h2.2.7/h1.2.7.

Description

The `ActiveDutIterator` and `SoftwareOnlyActiveDutIterator` macros are used to iterate over all DUTs currently in the Active DUTs Set (ADS). Note the following:

- The difference between `ActiveDutIterator` and `SoftwareOnlyActiveDutIterator` is that the latter does not modify hardware state; i.e. it does not enable/disable per-DUT test resources as the iterator executes. Thus, `SoftwareOnlyActiveDutIterator` should only be used when *getting* per-DUT values, not for *setting* any values which should affect hardware state. `SoftwareOnlyActiveDutIterator` may execute faster than `ActiveDutIterator`.
- Before the iteration loop begins the current Active DUTs Set (ADS) is saved.
- During the iteration loop, the Active DUTs Set (ADS) is modified within the body of the `ActiveDutIterator` or `SoftwareOnlyActiveDutIterator` block. One originally active DUT at a time is set active for each iteration. The `active_duts_get()` function can be used to identify which DUT is active during each loop iteration. Note that only DUTs in the Active DUTs Set (ADS) at the start of the iteration loop will ever be set active within the loop.
- `ActiveDutIterator` and `SoftwareOnlyActiveDutIterator` have the following member functions:
  - The `More()` member function returns TRUE as long as more iterations are to occur, then FALSE is returned.
  - By default, when the iteration loop completes, the original Active DUTs Set (ADS) is the restored and the associated variables are destroyed. However, beginning in softwware release h2.2.7/h1.2.7 the `More()` member function has an overload which takes a `BOOL` variable which, if set TRUE, resets the iterator pointer without destroying these variables. Setting this argument to FALSE or specifying no argument results in the original operation. Using this version of `More()` will result in faster iteration loop execution, which is beneficial when `SoftwareOnlyActiveDutIterator` is used within an outer loop. Example 4: includes additional information.
  - ~~The `Restore()` member function can be used to restore the original Active DUTs Set (ADS) (although this is normally not required).~~

Note: the previous reference to the `Restore()` member function was documented in error. It is not necessary or desirable for `Restore()` to be used. Support for this function may be removed in a future software release.

- When the iteration loop ends the original Active DUTs Set (ADS) is the restored. In software releases prior to h1.1.23 this occurred when leaving the scope of the iterator macro (see Note:). Beginning in software release h1.1.23 this occurs when the `More()` member function returns FALSE; i.e. immediately after the closing parenthesis of the iterator `while` loop.

- Beginning in software release h1.1.23, it is possible to (usefully) nest calls to `ActiveDutIterator` (and `SoftwareOnlyActiveDutIterator`), with each nesting level starting with the original ADS contents; i.e. the ADS contents prior to starting the outer level iterator loop. Previously, the inner loop would only see the ADS currently set by the outer loop, which would only contain one DUT (which was not useful). See examples.

Note: this note becomes obsolete using software release h1.1.23 or later. In earlier releases this remains critical to proper program operation.

if `ActiveDutIterator` is to be used multiple times within a function, proper operation requires that each instance be delimited in its own block, using an extra set of curly braces. This is important to program reliability. For example:

```
{ // Start block delimiter
   ActiveDutIterator duts;
   while ( duts.More() ) {
      [user code]
   }
} // End block delimiter
```

Usage

The following executes [user code] once for each DUT in the Active DUTs Set (ADS). The hardware for each DUT not in the Active DUTs Set (ADS) is disabled for each loop iteration:

```
ActiveDutIterator iID;
while ( iID.More() ) {
   [user code]
}
```

The following executes [user code] once for each DUT in the Active DUTs Set (ADS). The hardware state is not modified:

```
SoftwareOnlyActiveDutIterator iID;
while ( iID.More() ) {
   [user code]
}
```

The following executes [user code] once for each DUT in the Active DUTs Set (ADS). The hardware state is not modified. When iteration ends the ADS pointer is reset to the start of the ADS and the iterator variables are not destroyed, see Description and Example 4: below:

```
SoftwareOnlyActiveDutIterator iID;
while ( iID.More( BOOL wrap ) ) {
   [user code]
}
```

where:

iID is the iterator variable, not used except as shown.

The `More()` member function returns TRUE as long as additional DUT(s) remain to be processed, otherwise FALSE is returned. Beginning in softwware release h2.2.7/h1.2.7 the `More( BOOL wrap )` overload allows the iterator pointer to be reset without destroying the iterator variables, see Description. Using this version of `More()` will result in faster iteration loop execution when `SoftwareOnlyActiveDutIterator` is used within an outer loop. See Example 4: below.

When using software release h1.1.23 or later see Note: regarding program reliability.

Examples

Example 1:

The following example prints a list of DUTs currently in the Active DUTs Set (ADS). The extra curly braces are explained in Note:

```
{
   ActiveDutIterator duts;
   while ( duts.More() )
      output(" DUT-%d is in the Active DUTs Set (ADS)",
                active_dut_get() +1);
}
```

Example 2:

The following example retrieves and prints per-DUT test results after executing a test:

```
SoftwareOnlyActiveDutIterator ad1;
while ( ad1.More() ) {
   DutNum dut = active_dut_get( );
   output(" Test Result for DUT-%d => %d",
        dut +1,
        result_get( dut ) );
}
```

Example 3:

The following example demonstrates nested iterator use, only useful beginning in software release h1.1.23:

```
DWORD d;
output( "The ADS contains %d DUTs.", active_duts_get(&d) );

ActiveDutIterator ad1;
while ( ad1.More() ) { // Start Outer
   DutNum dut = active_dut_get();
   output( "Outer: t_dut%d", dut + 1 );
   ActiveDutIterator ad2;
   while ( ad2.More() ) {
      DutNum dut = active_dut_get();
      output( "  Inner-1: t_dut%d", dut + 1 );
   }
   ActiveDutIterator ad3;
   while ( ad3.More() ) {
      DutNum dut = active_dut_get();
      output( "  Inner-2: t_dut%d", dut + 1 );
   }
} // End Outer
```

Given a test program configured to test two DUTs, the example produces the following output:

```
The ADS contains 2 DUTs.
Outer: t_dut1
   Inner-1: t_dut1
   Inner-1: t_dut2
   Inner-2: t_dut1
   Inner-2: t_dut2
Outer: t_dut2
   Inner-1: t_dut1
   Inner-1: t_dut2
   Inner-2: t_dut1
   Inner-2: t_dut2
```

Example 4:

The following examples demonstrate the use of the `More(BOOL wrap)` overload added in software release h2.2.7/h1.2.7. The first two code segments have the same result but the *Enhanced* version will execute faster. The difference is that

`SoftwareOnlyActiveDutIterator` is defined outside the `for()` loop, which requires the use of the TRUE argument to the `More()` member function. Note that in the 3rd example since the `SoftwareOnlyActiveDutIterator` is defined outside the `for()` loop the ADS iteration will not execute as expected. This occurs because the `More(TRUE)` option was not used, thus the iteration pointer is not reset and only one ADS loop iteration can occur:

Original:

```
for ( int i = 0; i < 3; ++i ) {
   SoftwareOnlyActiveDutIterator iID;
   while ( iID.More() ) {
      output( "  Iteration %d for t_dut%d",
              i,
              active_dut_get() + 1 );
   }
}
```

Enhanced:

```
SoftwareOnlyActiveDutIterator iID;
for ( int i = 0; i < 3; ++i ) {
   while ( iID.More( TRUE ) ) {
      output( "  Iteration %d for t_dut%d",
              i,
              active_dut_get( ) + 1 );
   }
}
```

Incorrect:

```
SoftwareOnlyActiveDutIterator iID;
for ( int i = 0; i < 3; ++i ) {
   while ( iID.More() {
      output( "  Iteration %d for t_dut%d",
              i,
              active_dut_get( ) + 1 );
   }
}
```

2.4 Ignored DUTs Set (IDS)

See Magnum 1, 2 & 2x Parallel Test, Active DUTs Set (ADS).

Description

As noted earlier, during Sequence & Binning Table execution the Active DUTs Set (ADS) identifies which DUT(s) are enabled for testing at any given time. The *Ignored DUTs Set*, or IDS, defines a set of DUT(s) which will be ignored by the Sequence & Binning Table; i.e. not tested. The IDS only exists in the Site process.

Ignored DUT(s) are never included in the Active DUTs Set (ADS), During Sequence & Binning Table execution, the system software manipulates the Active DUTs Set (ADS) as needed to correctly and completely test the all DUTs which are not in the *Ignored DUTs Set*.

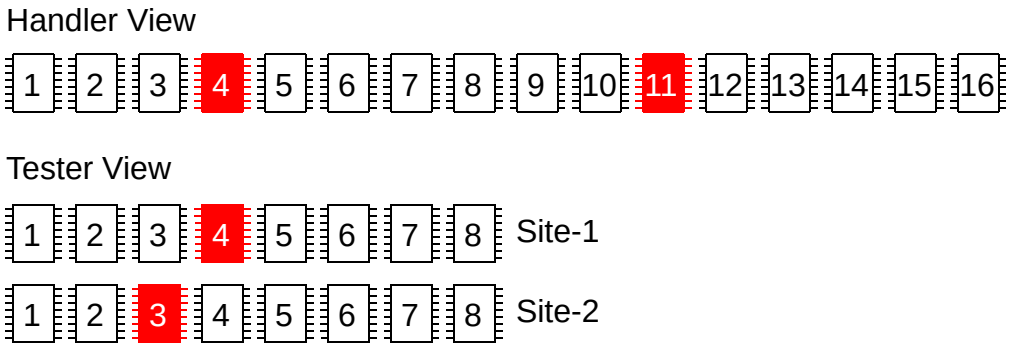
During initial program load, the Ignored DUTs Set is initialized to include zero DUTs. The system software does not otherwise set/modify the IDS. Thus, if no changes to the IDS are made by user code, all DUTs defined in the Pin Assignment Table are tested by each execution of the Sequence & Binning Table.

User code can modify the IDS using `ignored_duts_enable()` or `ignored_duts_disable()`. This supports the situation in which an IC handler or prober is able to specify which test sites are active or disabled (which can be affected by an empty input tray, a full output bin, a known bad DUT socket, etc.). And, the value can change for each start test signal issued by the hander or prober. In this situation, the *IDS* must be updated for each start test signal, typically using code included in the handler/prober control loop, executing in the Host process (more below).

Each time the Sequence & Binning Table is executed, the IDS is consulted to determine which DUT(s) should be tested.

Note: it is recommended that the Ignored DUTs Set functions (`ignored_duts_enable()`, `ignored_duts_disable()`) be executed ***before*** the Sequence & Binning Table executes the first test block; i.e. a Before-testing Block, User Variables code, etc. This ensures that the Sequence & Binning Table correctly and completely tests all DUT(s) enabled at the start of each execution.

When defining the IDS, user code is responsible for mapping a given handler/prober DUT to the appropriate tester site and DUT number. For example:



In this example, the handler specifies that DUTs 4 and 11 must be ignored. This translates into `t_dut4` on Site-1 and `t_dut3` on Site-2. User code must correctly identify both the DUT number and the site testing each DUT, for each DUT being ignored. See Examples.

2.4.1 ignored_duts_enable()

See Magnum 1, 2 & 2x Parallel Test, Active DUTs Set (ADS), Ignored DUTs Set (IDS).

Description

The `ignored_duts_enable()` function is used to completely redefine or add DUT(s) to the Ignored DUTs Set (IDS).

Note: it is recommended that `ignored_duts_enable()` be executed before the Sequence & Binning Table executes the first test block; i.e. a Before-testing Block, User Variables code, etc. This ensures that the Sequence & Binning Table correcly and completely tests all DUT(s) enabled at the start of execution.

When defining the IDS, user code is responsible for mapping a given handler/prober DUT to the appropriate tester site and DUT number. See example in Ignored DUTs Set (IDS).

Several versions of `ignored_duts_enable()` are provided:

- The version of `ignored_duts_enable()` which takes the `DutNum` argument allows a single DUT to be specified.
- The version of `ignored_duts_enable()` which takes the `DutNumArray` argument allows multiple DUT(s) to be specified.

- The version of `ignored_duts_enable()` which takes the `mask` argument allows multiple DUT(s) to be specified using a bitmask to identify which DUT(s) are to be included.
- All versions can either completely redefine or modify the Ignored DUTs Set (IDS), as controlled by the `incremental` argument. See Usage.

Usage

The following functions are used to completely redefine or incrementally add DUT(s) to the Ignored DUTs Set (IDS):

```
BOOL ignored_duts_enable( DutNum dut,
                          BOOL incremental DEFAULT_VALUE( FALSE ) );
BOOL ignored_duts_enable( DutNumArray &duts,
                          BOOL incremental DEFAULT_VALUE( FALSE ) );
BOOL ignored_duts_enable( DWORD mask,
                          BOOL incremental DEFAULT_VALUE( FALSE ) );
```

where:

`dut` identifies one DUT which:

- When **`incremental`** is TRUE, is added to the Ignored DUTs Set (IDS) .
- When **`incremental`** is FALSE, completely redefines the Ignored DUTs Set (IDS) .

`duts` is a `DutNumArray` identifying zero or more DUTs, which:

- When **`incremental`** is TRUE, are added to the Ignored DUTs Set (IDS)
- When **`incremental`** is FALSE, completely redefines the Ignored DUTs Set (IDS) .

`incremental` is optional, and if specified determines whether the current Ignored DUTs Set (IDS) will be modified (TRUE) or completely redefined (FALSE). Default = FALSE.

`mask` is a bitmask identifying zero or more DUTs, which:

- When **`incremental`** = TRUE DUT(s) are added to the Ignored DUTs Set (IDS).
- When **`incremental`** = FALSE completely redefines the Ignored DUTs Set (IDS).

In the bitmask, a 1 identifies a DUT to be added and 0 identifies a DUT which will not be added.

`ignored_duts_enable()` returns FALSE if an invalid DUT is specified; i.e. a DUT which is not defined in the Pin Assignment Table.

Examples

The following example is designed to execute in the Host process, as part of the handler/prober control loop. For each start test signal from the handler/prober, this code gets the list of DUT(s) which are to be ignored, translates it from the handler view to the tester view (see example in Ignored DUTs Set (IDS)) and sends an ignore list to each Site:

```
// Definitions this code needs to operate correctly
#define NUM_SITES 2
#define DUTS_PER_SITE 8
DutNumArray MAG_ignore_dut_list, Handler_ignore_dut_list;

// // User must write my_handler_get_dut_ignore_function()... too
// handler specific. May also need to define appropriate data
// structure(s) and map handler DUT IDs to DutNums.
my_handler_get_dut_ignore_function( &Handler_ignore_dut_list );

// Translate handler/prober view to tester view, and update the
// Ignored DUTs Set (IDS) on each site.
for( int site = 0; site < NUM_SITES; ++site ) {
   MAG_ignore_dut_list.RemoveAll(); // Clear it for each site

   // Process the entire handler list, extract DUTs for one site at
   // a time.
   for( int i = 0; i < Handler_ignore_dut_list.GetSize(); ++i ){
      DutNum dn = Handler_ignore_dut_list.GetAt(i);
      if(  dn <= (t_dut8 + DUTS_PER_SITE * site) &&
           dn >= (t_dut1 + DUTS_PER_SITE * site)
        ){

         // Current dn maps to the current site, translate and put
         // into the ignore list
         DutNum di = (DutNum) (dn - (DUTS_PER_SITE * site));
         MAG_ignore_dut_list.Add( di );
      }
   }

   // All DUT(s) for the current site have been identified & put
   // into the MAG_ignore_dut_list. Send the list to the current
   // site. This redefines the Ignored DUTs Set (IDS) for this site.
   SiteMask mask( 1 << site );
   ignored_duts_enable(mask,
                       MAG_ignore_dut_list,
                       FALSE );
}
```

2.4.2 ignored_duts_disable()

See Magnum 1, 2 & 2x Parallel Test, Active DUTs Set (ADS), Ignored DUTs Set (IDS).

Description

The `ignored_duts_disable()` function is used to completely define or remove DUT(s) from the Ignored DUTs Set (IDS).

Note: it is recommended that `ignored_duts_disable()` be executed before the Sequence & Binning Table executes the first test block; i.e. in a Before-testing Block, User Variables code, etc. This ensures that the Sequence & Binning Table correcly and completely tests all DUT(s) enabled at the start of execution.

When defining the IDS, user code is responsible for mapping a given handler/prober DUT to the appropriate tester site and DUT number. See example in Ignored DUTs Set (IDS).

Several versions of `ignored_duts_disable()` are provided:

- The version of `ignored_duts_disable()` which takes the `DutNum` argument allows a single DUT to be specified.
- The version of `ignored_duts_disable()` which takes the `DutNumArray` argument allows multiple DUT(s) to be specified.
- The version of `ignored_duts_disable()` which takes the `mask` argument allows multiple DUT(s) to be specified using a bitmask to identify which DUT(s) are to be included.
- All versions can either completely redefine or modify the Ignored DUTs Set (IDS), as controlled by the `incremental` argument. See Usage.

Usage

The following functions are used to completely redefine or incrementally remove DUT(s) from the Ignored DUTs Set (IDS):

```
BOOL ignored_duts_disable( DutNum dut,
                           BOOL incremental DEFAULT_VALUE( FALSE ) );
BOOL ignored_duts_disable( DutNumArray &duts,
                           BOOL incremental DEFAULT_VALUE( FALSE ) );
BOOL ignored_duts_disable( DWORD mask,
                           BOOL incremental DEFAULT_VALUE( FALSE ) );
```

where:

dut identifies one DUT which:

- When **incremental** is TRUE, is removed from the Ignored DUTs Set (IDS).
- When **incremental** is FALSE, completely redefines the Ignored DUTs Set (IDS) .

duts is a DutNumArray identifying zero or more DUTs, which:

- When **incremental** is TRUE, are removed from the Ignored DUTs Set (IDS)
- When **incremental** is FALSE, completely redefines the Ignored DUTs Set (IDS) .

incremental is optional, and if specified determines whether the current Ignored DUTs Set (IDS) will be modified (TRUE) or completely redefined (FALSE). Default = FALSE.

mask is a bitmask identifying zero or more DUTs, which:

- When **incremental** = TRUE are removed from the Ignored DUTs Set (IDS)..
- When **incremental** = FALSE completely redefines the Ignored DUTs Set (IDS).

In the bitmask, a 1 identifies a DUT to be removed and 0 identifies a DUT which will not be removed.

ignored_duts_disable() returns FALSE if an invalid DUT is specified; i.e. a DUT which is not defined in the test program.

Examples

This example creates an empty DutNumArray, which when used by ignored_duts_disable() has the effect of removing all DUTs from the Ignored DUTs Set (IDS):

```
#define NUM_SITES 2
DutNumArray restore_list;
for( int site = 0; site < NUM_SITES; ++site)
   ignored_duts_disable( site+1, restore_list, FALSE );
```

Also see Examples.

2.4.3 ignored_duts_get()

See Magnum 1, 2 & 2x Parallel Test, Active DUTs Set (ADS), Ignored DUTs Set (IDS).

Description

The `ignored_duts_get()` function is used to get information about which DUT(s) are currently in the Ignored DUTs Set (IDS).

`active_duts_get()` may be used to access the Active DUTs Set (ADS).

Usage

The following function determines whether the specified DUT is currently in the Ignored DUTs Set (IDS):

```
BOOL ignored_duts_get( DutNum dut );
```

The following function returns an array containing the DUT(s) which are currently in the Ignored DUTs Set (IDS):

```
int ignored_duts_get( DutNumArray *duts );
```

The following function returns a bitmask which indicates which DUT(s) are currently in the Ignored DUTs Set (IDS):

```
int ignored_duts_get( DWORD *mask );
```

where:

`dut` identifies one DUT (`DutNum`) to be checked for inclusion in the Ignored DUTs Set (IDS).

`duts` is a pointer to an existing `DutNumArray` used to return a copy of the Ignored DUTs Set (IDS).

`mask` is a pointer to an existing `DWORD` used to return a bitmask of the Ignored DUTs Set (IDS). In the returned bitmask, a 1 identifies a DUT which is in the Ignored DUTs Set (IDS) and 0 identifies a DUT which is not in the Ignored DUTs Set (IDS).

The version of `ignored_duts_get()` which returns `BOOL`, returns TRUE if the specified **`dut`** is in the Ignored DUTs Set (IDS), otherwise FALSE is returned.

The versions of `ignored_duts_get()` which returns `int`, return the number of DUT(s) which are in the Ignored DUTs Set (IDS).

Example

The following example checks if `t_dut2` is currently in the Ignored DUTs Set (IDS):

```
if( ignored_duts_get( t_dut2 ) )
   output(" t_dut2 IS in the IDS");
else
   output(" t_dut2 is NOT in the IDS");
```

The following example returns an array containing those DUT(s) currently in the Ignored DUTs Set (IDS):

```
DutNumArray IDS_copy;
int count = ignored_duts_get( &IDS_copy );
```

The following example returns a bitmask indicating which DUT(s) are currently in the Ignored DUTs Set (IDS) and, using the mask, checks to see if `t_dut3` is in the Ignored DUTs Set (IDS):

```
DWORD mask;
int count = ignored_duts_get( &mask );
if( mask && 0x4 )
   output(" t_dut3 IS in the IDS");
else
   output(" t_dut3 is NOT in the IDS");
```

2.5 Multi-DUT Test Results

In Multi-DUT Test Programs, the following test functions automatically store a test result for each DUT in the Active DUTs Set (ADS) (ADS) at the time the test is executed:

- `funtest()`
- `test_supply()` and `ac_test_supply()`
- `partest()` and `ac_partest()`
- `hv_test_supply()` and `hv_ac_test_supply()`
- `ptu_partest()`

These results are stored in a system defined data structure which is used by the system software to properly track and manage per-DUT testing via the Sequence & Binning Table.

User code may utilize the following functions to access these same test results:

- `result_set(), result_get()`
- `results_set(), results_get()`

- `all_results_match()`
- `any_results_match()`

During the test execution the system software determines which DUT(s) failed the test and executes `result_set()` to store a result for each DUT in the ADS. When a given Test Block only executes one of these tests (once), user code is typically not required to manage test results. However, if more than one test function is executed in a given Test Block, the results of the last one executed will over-write those from earlier tests. In this situation, it will be necessary for user code to track and combine the test results from all test(s) executed within the Test Block before the test block exits. This requirement is no different than when testing a single DUT however the methods used are different. (more below).

In Multi-DUT Test Programs, when a given Test Block exits, the Sequence & Binning Table needs a test result for each DUT that was in the ADS at the time the test block execution started. The test functions manage this correctly, provided user code has not modified the ADS in the Test Block and only one test function executes in the test block. When user code executes `result_set()` or `results_set()` or modifies the ADS in a Test Block the following rules apply:

- The default per-DUT test result = FAIL.
- All DUT(s) in the ADS at the start of test block execution must have a test result specified, using `result_set()` or `results_set()`, before the test block exits. If this rule is violated a warning is displayed, in the appropriate controller output window, and the undefined DUT(s) are assigned the default test result.

Also:

- Using `result_set()` or `results_set()`, any test results specified for DUT(s) which are parked will generate a warning message but are otherwise ignored.
- Using `result_set()` or `results_set()`, any test results specified for DUT(s) which are in the Ignored DUTs Set (IDS) will generate a warning message but are otherwise ignored.

As noted above, any Test Block which executes more than one test requires user-written code to track and merge the test results for all test(s) before the Test Block exits. Examples follow.

Note: the following examples assume a Multi-DUT Test Program.

The following example requires no user test result code:

```
TEST_BLOCK( myTB1 ){ // Or TEST_BLOCK_SEQUENTIAL
   // Other test setup code as desired
   PFState result = funtest( myPat1, ... );
   // Other code as desired (which doesn't execute a test)
   return MULTI_DUT;
}
```

In this example, since only one test function is executed the correct parallel test results are recorded by the function and are available to the Sequence & Binning Table; i.e. no user code is needed to manage per-DUT test results. However, note the following:

- In Multi-DUT Test Programs the test functions still return PASS/FAIL, reflecting the overall test result of all DUT(s) tested (in the Active DUTs Set (ADS)). The individual
  per-DUT test results are stored as noted above.
- In Multi-DUT Test Programs, the only valid return value from a Test Block is `MULTI_DUT`. If this rule is violated, a warning is displayed, in the appropriate controller output window, but testing proceeds as though `MULTI_DUT` was returned.
- Combining the two previous bullets, the following code will cause a warning but will also operate correctly:

  ```
  return funtest( ... ); // Any test function
  ```

- Two new test block macros, `MULTI_DUT_TEST_BLOCK` and `TEST_BLOCK_SEQUENTIAL`, are available which implicitly return `MULTI_DUT`, eliminating the need for user code to return a value. These are only usable in Multi-DUT Test Programs. See Test Block Macros.

In the following example, only the test results from the 2nd test will be seen by the Sequence & Binning Table because the results from the 1st test are over-written. This is likely not what the user wanted:

```
TEST_BLOCK( myTB1 ){ // Or TEST_BLOCK_SEQUENTIAL
   // Other test setup code as desired
   PFState result;
   result = funtest( myPat1, error );
   result &= funtest( myPat2, error );
   return MULTI_DUT;
}
```

The following example shows how user code can explicitly manage test results when more than one test is executed in a test block:

```
TEST_BLOCK( myTB1 ){ // Or TEST_BLOCK_SEQUENTIAL
   // Other test setup code as desired
   PFState result = funtest( myPat1, finish );
   IntArray r1, r2; // To store local test results
   int count = results_get( &r1 ); // Save results from 1st test
   result = funtest( myPat2, finish );
   count = results_get( &r2 ); // Save results from 2nd test
   // Combine test results into r1
   for( int dut = 0; dut < count; ++dut )
      r1[ dut ] = r1[ dut ] && r2[ dut ] ;
   results_set( r1 );
   return MULTI_DUT;
}
```

Regarding this example note the following:

- The fact that `results_get()` returns values in the order DUT(s) are listed in the Active DUTs Set (ADS) ensures that, in this example, the `results_set()` loop inherently sets values in the correct order.
- If user code modifies the Active DUTs Set (ADS) between the two tests this example will not work. In most applications, the need to change the Active DUTs Set (ADS) is temporary, and it will be appropriate to save and restore the original ADS to ensure overall operation remains as described above. See ADS Save/ Modify/Restore Example.

2.5.1 result_set(), result_get()

See Overview, Multi-DUT Test Results, Program Execution Control.

Description

The `result_set()` function is used, in Multi-DUT Test Program only, to explicitly set a test result for one DUT. As described Multi-DUT Test Results, it is sometimes not necessary for user code to use `result_set()`; i.e. the system software may do all the work.

The `result_get()` function is used to get the current test result value for one DUT. Again, in Multi-DUT Test Programs only.

The `results_set(), results_get()` functions support setting or getting test results for multiple DUTs in Multi-DUT Test Programs.

Functions, MACROs & Keyword

```
result_set()
result_get()
```

Usage

```
BOOL result_set( DutNum dut, int result );
int result_get( DutNum dut );
```

where:

`dut` identifies one DUT for which the test result is being set or retrieved.

`result` specifies the test result for **`dut`**. Legal values are any non-negative integer, where 0 = FAIL, 1 = PASS, and other values are also PASS. Note that the Sequence & Binning Table macros support values of 0 through 7, which correspond to the `On0`, `On1`,... `On7` parameter positions of the Test Block Macros. Other values may be useful within the test block (UI's Breakpoint Monitor supports up to 63), or user code may extend the Test Block Macros to support values > 7 in the Sequence & Binning Table.

`result_set()` returns FALSE if an invalid `DutNum` is specified; i.e. a `DutNum` which is not defined in the Pin Assignment Table or not in the current Active DUTs Set (ADS).

`result_get()` returns the test result for one specified DUT.

Example

The following example set a PASS test result for `t_dut1`:

```
result_set( t_dut1, PASS );
```

2.5.2 results_set(), results_get()

See Overview, Multi-DUT Test Results, Program Execution Control.

Description

The `results_set()` function is used, in Multi-DUT Test Program only, to explicitly set a test result for one or more DUT(s). As described below, it is sometimes not necessary for user code to use `results_set()`; i.e. the system software may do all the work.

The `results_get()` function is used to get the current test result value for one or more DUT(s). Again, in Multi-DUT Test Programs only.

The `result_set(), result_get()` functions support setting or getting test results for one DUT in Multi-DUT Test Programs.

Functions, MACROs & Keyword

```
results_set()
results_get()
```

Usage

```
int results_set( int result );
BOOL results_set( IntArray &results );
int results_get( IntArray *results );
```

where:

`result` specifies the test result assigned to all DUT(s) in the Active DUTs Set (ADS). Legal values are any non-negative integer, where 0 = FAIL, 1 = PASS, and other values are also PASS. Note that the Sequence & Binning Table macros support values of 0 through 7, which correspond to the `On0`, `On1`,... `On7` parameter positions of the Test Block Macros. Other values may be useful within the test block (UI's Breakpoint Monitor supports up to 63), or user code may extend the Test Block Macros to support values > 7 in the Sequence & Binning Table.

`results` is used in two contexts:

- Using `results_set()`, **`results`** is an `IntArray` containing a test result for one or more DUT(s). Values MUST be ordered to match the order of DUT(s) in the Active DUTs Set (ADS). See **`result`** above for legal values.
- Using `results_get()`, **`results`** is a pointer to an existing `IntArray` used to return a test result for one or more DUT(s). The results will be ordered to match the order of DUT(s) in the Active DUTs Set (ADS). See **`result`** above for legal values.

The version of `results_set()` which returns a `BOOL` returns FALSE if the size of `results` does not match the number of DUTs in the Active DUTs Set (ADS), or if a test result is specified for a DUT which was not in the Active DUTs Set (ADS) at the start of the current test block.

The version of `results_set()` which returns an `int` returns the number of DUTs currently in the Active DUTs Set (ADS).

`results_get()` returns the number of values in the **`results`** array.

Example

See Multi-DUT Test Results for an example using `results_set()` and `results_get()`.

2.5.3 all_results_match()

See Overview, Multi-DUT Test Results, Program Execution Control.

Description

In Multi-DUT Test Programs the system software manages a set of test results, one value for each DUT defined in the Active DUTs Set (ADS). See Multi-DUT Test Results.

The `all_results_match()` function can be used to test whether the individual test result for each DUT(s) in the Active DUTs Set (ADS) match a specified value.

Usage

```
BOOL all_results_match( int value );
```

where:

`value` is the test result to be checked.

`all_results_match()` returns TRUE if the current test results for all DUT(s) in the Active DUTs Set (ADS) all match **`value`**, otherwise FALSE is returned.

Example

```
if( all_results_match( FAIL ) )
   output(" All active DUTs are FAILing");
else
   output(" One or more active DUTs are not FAILing");
```

2.5.4 any_results_match()

See Overview, Multi-DUT Test Results, Program Execution Control.

Description

In Multi-DUT Test Programs the system software manages a set of test results, one value for each DUT defined in the Active DUTs Set (ADS). See Multi-DUT Test Results.

The `any_results_match()` function can be used to determine if the current test results contain one or more values matching a specified value.

Usage

```
BOOL any_results_match( int value );
```

where:

`value` is the test result to be checked.

`any_results_match()` returns TRUE if any value in the current test results match **`value`**, otherwise FALSE is returned.

Example

```
if( any_results_match( FAIL ) )
   output(" At least one DUT is FAILing");
else
   output(" No DUTs are FAILing");
```

2.6 Functional Test Pattern Execution

In Multi-DUT Test Programs, functional test pattern execution has an important additional consideration:

- In most cases, the pattern execution stop option must be set to execute the entire test pattern !

The alternative stops pattern execution on detection of an error (failing strobe). If this is done, the system software will correctly report which DUT(s) had a failure and which DUT(s) passed.

- ***HOWEVER***, test pattern execution was stopped ***EARLY***, thus the test was ***INCOMPLETE*** on DUT(s) which didn't have a failure.

This means selecting the `PatStopCond` = `finish` (NOT `error`) when executing:

- `funtest()` (and usually `start_pattern()`)
- `ac_test_supply()`
- `ac_partest()`
- `hv_ac_test_supply()`

2.7 Functional Pin-pairs

See Magnum 1, 2 & 2x Parallel Test.

As shown in the Site Assembly Board Block Diagram, each Magnum 1/2 Site Assembly Board in has 128 signal pins but only 64 timing channels. This means that each timing channel is shared by two (2) signal pins.

This concept is called *functional pin-pairs; i.e.* 2 pins get a the same timed drive/strobe/IO signals under control of the same test pattern data source, per-cycle. This architecture supports most parallel test applications while reducing the overall cost-per-pin of the test system. Note that all 128 pins have independent PE Drivers, PE Comparators, and Per-pin Parametric Test Unit (PTU) and independent DC levels.

The functional pin-pairs architecture does require the user consider the following:

- Using a Multi-DUT Test Program.
- In most applications, a given pin on each DUT can use the same timing and pattern control. When this is not acceptable, i.e. when idependent timing and/or or pattern control is required for a given pin DUT 4 pins will effectvely be consumed; the 2 pins which are used at the DUT and the 2 other pins of each pin-pair which are not.
- The DUT-to-pin definitions in the Pin Assignment Table must match the DUT board design. `HDTesterPin`s are identified using `a_1`, `b_1`, `b_2`, etc. It should be obvious that `a_1` and `b_1` a pin-pair. During functional pattern execution both pins of a given pin-pair will receive the same timing signals, controlled by the same pattern data source, etc.

- In Multi-DUT Test Programs, in the Pin Assignment Table, a specific set of MACROs are used to specify which tester resource is connected to each DUT's pins. By definition, DUT-1 and DUT-2 will share functional pin-pairs, DUT-3 and DUT-4 will share functional pin-pairs, etc. As indicated above, it is also possible for [some] pins to not use pin-pairs. See Pin Assignment Table.
- Both pins of a pin-pair must use the same PE driver mode. See Magnum PE Driver Modes.
- Only the Magnum 1/2 versions of the Redundancy Analysis (RA) software supports the upper 64 pins (B-pins). See Note:.

Chapter 3 Software

- Software Architecture Overview
- Test System Macros
- Specifying Units
- Special Data Types
- Types, Enums, etc.
- output(), warning(), fatal(), vFormat()
- Configuring the Tester to the DUT
- Program Execution Control
- DPS Functions
- PMU Functions
- Pin Electronics Voltages/Currents
- Timing and Formatting Functions
- Test Patterns
- Functional Tests
- Data Buffer Memory (DBM)
- Error Catch RAM (ECR)
- Manipulating Tester Hardware

3.1 Software Architecture Overview

See Software.

A simple software hierarchy model is shown below:

Windows OS	
Win32 DLLs	Windows Libraries
MFC DLLs	Microsoft Foundation Class Libraries
Nextest DLLs	Libraries of Functions
User code	Test program, test patterns, User Tools

This manual documents the software used to write the test program, test patterns, and User Tools.

3.1.1 Test Program Overview

See Software.

Magnum 1/2/2x test programs are composed of:

- One or more user-created C-code source files. The code in these files will invoke various Nextest functions and macros to define program data and operation. In addition, all standard C/C++ facilities are available to implement user-defined functionality.
- Zero or more user-written test pattern source files. Test Pattern Programming consists of Memory Test Patterns and/or Logic Test Patterns and/or Scan Test Patterns, written using a proprietary pattern language. The pattern compiler (Patcom) generates C code from the pattern language source files, which then becomes part of the Visual C++ project that is built into a complete test program.
- Miscellaneous files required by or created by Developer Studio to manage each project (makefile, project workspace definition file, etc.). These are not documented here.

- Libraries of Nextest functions and macroS, and associated data types, are used to communicate with tester hardware, and to use other features designed by Nextest. These are incorporated into the test program using one `#include` statement (`#include "TestProgApp/public.h"`), typically in a file named *tester.h*.

3.1.2 Test Program Wizards

See Software.

Two Nextest wizards are included with Microsoft Visual C++, as shipped with the test system software:

- MinimalTestProgram
- New Test Program

The *MinimalTestProgram* option creates a single file containing one `#include` statement. This `#include` causes the complete Nextest software libraries (functions, macros, data types) to be accessible to code in the file. This option allows experienced Magnum 1/2/2x users a quick way to create a program, but does not provide any indication, or guidance, for completing the other features typically required in a usable test program.

The *New Test Program* option creates a test program shell, containing a number of source files, and one test pattern (see Test Program Wizard Files). These files contain comments which outline the basic purpose of that file, and example code showing typical, but very basic, contents. This option can be used at any time to create a basic test program shell which contains standard components, compiles correctly, but does little else.

3.1.2.1 Test Program Wizard Files

See Software.

When the *New Test Program* Test Program Wizard is invoked it creates a test program template containing the following files:

Table 3.1.2.1-1 Wizard Created Program Source Files

File Name	Reference
address_topo.cpp	See APG Address Topo RAM Load Functions
data_topo.cpp	See APG Data Topological Inversion (DTOPO) Function
host_begin.cpp	See Host Begin Block
pin_assignments.cpp	See Pin Assignment Table
pin_lists.cpp	See Pin Lists
pin_scramble.cpp	See Pin Scramble Map
seq_and_bin.cpp	See Sequence & Binning Table
site_begin.cpp	See Site Begin Block
test_blocks.cpp	See Test Blocks
vihh_maps.cpp	See VIHH Maps
dialogs.cpp	Example dialog source code. See User Dialogs.
MultiDialog.rc	Example user-dialogs, as created by the dialog editor
pattern1\ folder	Stores temporary pattern compiler files
pattern1.pat	Test Pattern Source File
patterns.h	Auto-generated external declarations file
resource.h	External declarations file for dialog resources.
tester.h	External declarations file. Includes Nextest libraries.
*.mak	Autogenerated make file (Pre MsDev 6)
*.mdp	MSDS Project Workspace File (Pre MsDev 6)
*.dsw	MSDS Project Workspace File (MsDev 6)
*.dsp	Autogenerated make file (MsDev 6)
*.ncb	Autogenerated program database file
*.dep	MSDS Project Dependencies File

The wizard only creates a template. The source files created by the wizard have simple code examples embedded in them as comments. To create a complete test program, simply fill in the files following the form of the examples in each file.

Complete descriptions of the functions and macros used in these files can be found in this manual.

The user is free to [re]partition the code in these files in other ways, but following this basic standard is recommended. The test block and pattern files are often expanded into multiple files for ease of editing and tracking.

Note: a user-created `main()` program file will not be executed in the traditional sense. The `main()` of a Magnum 1/2/2x test program is a hidden part of the Nextest software.

Note: users familiar with MFC must ***NOT*** define their own `CWinApp`-derived class. Doing this will cause fatal side effects.

A brief description of the intended purpose of each file in the Test Program Wizard follows. Few test programs require all of these files, which if not needed can be deleted from the project. The files are listed alphabetically:

- `address_topo.cpp`: maps the logical X and Y addresses from the APG address generators into topologically scrambled (DUT physical) addresses. See APG Address Topo RAM Load Functions.
- `data_topo.cpp`: define APG data topological scramble code, used to invert APG data generator outputs, to generate physically correct data when testing devices that store data in inverted form as a function of address. See APG Data Topological Inversion (DTOPO) Function.
- `dialogs.cpp`: example user dialogs and related User Variables.
- `host_begin.cpp`: conventionally the file where the following macros are used: `HOST_CONFIGURATION()`, `HOST_BEGIN_BLOCK()`, `HOST_END_BLOCK()`. Code in the Host Begin Block is often used to communicate outside the tester to operators, networks, external equipment, etc.
- `pin_assignments.cpp`: conventionally the file where the Pin Assignment Table is defined. This maps DUT pin names to DUT pin numbers, and tester pin numbers.

- `pin_list.cpp`: conventionally the location where Pin Lists are defined. Since pin lists are typically global, a corresponding `pin_list.h` file is used to declare each pin list external.
- `pin_scramble.cpp`: conventionally the file where Pin Scramble Maps are defined. These map test pattern data sources to DUT pins.
- `seq_and_bin.cpp`: conventionally the file where the Sequence & Binning Table is defined. This controls the flow of the test execution, by calling test blocks one at a time. After executing a test block, a branch decision to another test block is made based on the result returned from the current test block just executed. Binning is also managed here.
- `site_begin.cpp`: conventionally the file where the following macros are used: `SITE_CONFIGURATION()`, `SITE_BEGIN_BLOCK()`, `SITE_END_BLOCK()`. Code in the Site Begin Block is typically used initialize test parameters which are set once.
- `test_blocks.cpp`: conventionally the file where the Test Blocks are defined. These are the core of a test program, where tests are executed, tester hardware is configured, etc. For instance, a low level input leakage test may be in one test block, a high level input leakage test in another test block, a gross functional AC test in another, etc. Setups, such as voltage and timing, are performed within the test blocks.
- `tester.cpp`: a boiler-plate file, containing the #include statement which enables the Nextest libraries. This ensures the program is a Nextest test program, loads all the requisite DLLs, etc.
- `vihh_maps.cpp`: conventionally the file where the VIHH Maps are defined. These specify which DUT pins are driven to the VIHH voltage level under pattern generator control.
- `*.pat`: these are test pattern source files. Magnum 1/2/2x supports Memory Test Patterns, Logic Test Patterns, Mixed Memory/Logic Patterns and Scan Test Patterns. Patterns are compiled automatically, along with the test program, using a special purpose pattern compiler. See Test Pattern Programming.

3.1.3 Program Loading and Execution Order

See Software.

When the test program is first loaded, the system software processes the various Test System Macros, as specified by the user's program code. This builds various lists and

tables, sets the tester hardware to its default state, and automatically executes several key initialization routines.

During this process, one key operation is reading the DUT Board calibration data from the ASIC on the DUT Board.

Note: to ensure calibration data is properly loaded, the DUT Board must be properly installed on the test system before the test program load is started.

The general order of execution, and the process in which each of the major pieces of tester software execute, is shown in the table below. See Overview in Program Execution Control for a description of the different program processes (Host, Site, etc.).

The CONFIGURATION block(s) always execute first on the Host, before any code is executed on the Site(s). After this, even though the table shows some steps happening side-by-side on the Host and Sites, there is no synchronization between these processes, unless explicitly implemented in user code:

Table 3.1.3.0-1 Program Load Execution Order per Process

	Host Process	Site Process(es)	Tool Process(es)
CONFIGURATION() (on Host)	X		
HOST_CONFIGURATION()	X		
DUT Board Status Check	X		
CONFIGURATION() (on Site)		X	
SITE_CONFIGURATION()		X	
CONFIGURATION() (on Tools)			X
TOOL_CONFIGURATION()			X
PIN_ASSIGNMENTS()	X	X	
CURRENT_SHARE()		X	
PINLIST()		X	

Table 3.1.3.0-1 Program Load Execution Order per Process *(Continued)*

	Host Process	Site Process(es)	Tool Process(es)
VIHH_MAP()		X	
PIN_SCRAMBLE()		X	
HOST_BEGIN_BLOCK()	X*		
SITE_BEGIN_BLOCK()		X	
TOOL_BEGIN_BLOCK()			X
INITIALIZATION_HOOK()	X	X	X

* The HOST_BEGIN_BLOCK() (only) executes in a separate thread, allowing code there to execute continuously as needed for handler/prober control, etc. For this reason, output(), warning(), and fatal() messages from the HOST_BEGIN_BLOCK() may appear to occur out of sequence relative to other code.

The Sequence & Binning Table is executed each time **Start Testing** is invoked.

Additional details can be obtained using the ui_ResourceInitialized User Variables. The Example Output shown there (ui_ResourceInitialized) was executed on the Site only.

3.1.4 DUT Board Status Check

See Software.

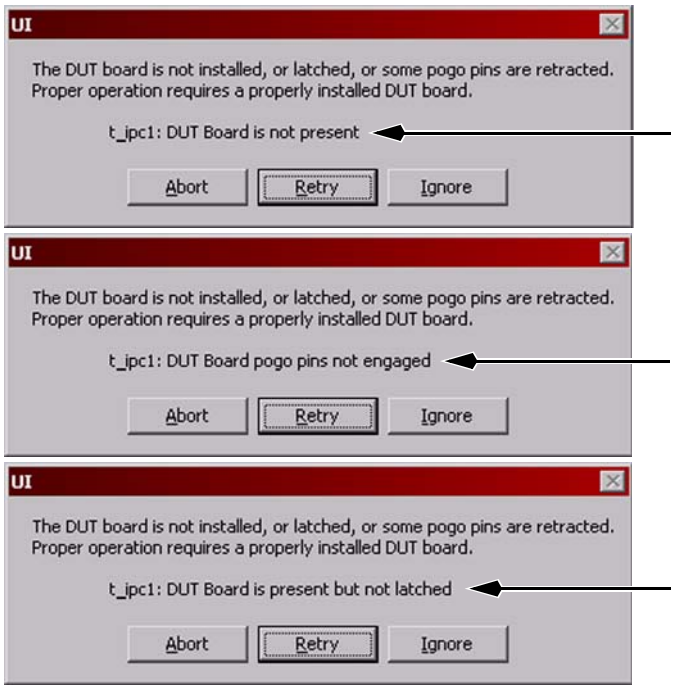
Note: first available in software release h1.1.23.

During the program load process, the system software checks the DUT board installation status. Most test programs require that a DUT board be properly installed. And, if the DUT board EEPROM stores TDR data the DUT board must be correctly installed before the test program loads on the Site controllers.

This check was added to reduce the potential confusion caused by pogo pins left retracted after executing system diagnostics or calibration. Beginning in software h1.1.23, the system diagnostics and calibration programs were modified to automatically retract the pogo pins, allowing these programs to operate with a DUT board installed and latched. However, these

programs do not extend the pogo pins, requiring that the user do so using the Latch button on the Manipulator Control Box.

If the DUT board status check detects that a DUT board is not installed, or not latched or that any pogo pins used by the program are retracted the user is presented with a warning dialog indicating the DUT board status:



Note that the warning message is identical in each dialog. The details are displayed just above the buttons.

The following options are available:

- Retry = check the DUT board status again. This is the default.
- Ignore = continue the program load.
- Abort = abort the program load.

The first option allows the user to install and latch a DUT board, and ensure that the pogo pins are extended and then perform the DUT board status check again.

By default, the DUT board status check is performed when any test program is loaded on a Magnum 1/2/2x Test System. The check can be disabled using `ui_DutBoardStatusCheckDisable`.

3.1.5 Program Un-Loading and Execution Order

See Software.

When the test program is *Closed* (unloaded), the system software will execute the following code if included in the test program, or User Tools. Each of the END_BLOCK macros executes only in the process indicated. Note that, unless explicitly implemented in user code, there is no synchronization between processes as implied by the table:

Table 3.1.5.0-1 Program Un-Load Execution Order per Process

	Host Process	Site Process(es)	Tool Process(es)
HOST_END_BLOCK()	X		
SITE_END_BLOCK()		X	
TOOL_END_BLOCK()			X

Note that output(), warning(), and fatal() messages generated in END_BLOCKs may not be displayed in the various UI - User Interface output windows. This is because the communications process used to send the messages from the Host, Site or Tool processes is terminated as the program is unloaded.

3.1.6 Program Working Directory

See Software.

The *working directory* of a loaded test program is the directory containing the **.exe** file, regardless of how the test program was loaded.

This is important if test program code does file I/O and uses *relative path names*, or invokes User Tools using *relative path names*. The working directory can be changed using the Win32 SetCurrentDirectory() function.

3.1.7 Retrieving the Nextest Software Version

See Software.

Description

The `current_release()` function can be used to get the version of the Nextest software currently in use. This may be useful when generating test reports, etc.

Also see `builtin_what_exe`.

Usage

```
CString current_release();
```

where:

`current_release()` returns the current Nextest software release as a `CString`.

Example

```
CString s = current_release();
output(" Release => %s", s );
```

3.2 Test System Macros

See Software.

Many of the lists and configurations (Resources) are specified in the test program using C macros, defined by Nextest. Macros are used to make test program generation simpler. They hide much of the underlying C-code implementation details and allow the user to focus on the testing issues. And, as Resources, additional capabilities are standardized.

3.2.1 Macro Syntax

See Software.

Nextest-defined macros are named using all upper case letters.

All of the Nextest-defined macros require an argument, which acts as the name of the macro. In many cases, this name will be referenced again. For example, a Sequence & Binning Table or Site Begin Block is selected using one of the `USE_xxx` macros (see Configuration Macros). In many cases, this name will be referenced as a pointer to a particular Resource type; for example `PinList*`, `DutPin*`, etc.

All of the Nextest-defined macros which have an argument must be typed with no spaces between the macro name and the open (left side) parentheses. Spaces are allowed anywhere else in the macro argument list. For example:

```
TEST( ... is correct
TEST ( ...) is not correct because of the space between TEST and
            the open parenthesis.
```

Nextest-defined macros never require a trailing semicolon but including one is harmless.

Many of the Nextest-defined macros may only be defined at global scope; i.e. not within the body of another macro or a C-function. Others are only usable within the body of another Nextest-defined macro. These are documented as needed.

Nextest-defined macros may now, or in the future, expand to more than one statement. This means that, for example:

```
if ( BoardPresent( t_pe1 ) )
   PINS2( ... )
else if ( BoardPresent( t_pe2 ) )
   PINS2( ... )
```

will not compile. Instead use (note the added braces):

```
if ( BoardPresent( t_pe1 ) ) {
   PINS2( ... )
}
else if ( BoardPresent( t_pe2 ) ) {
   PINS2( ... )
}
```

3.3 Specifying Units

See Software.

All of the Nextest API functions that set a voltage, current or time value allow the use of a units modifier. This takes the following form:

```
value units
```

where `value` is a floating point number (usually a `double`) and `units` is one of the Nextest-defined macros used as a multiplier for `value`. The space between `value` and `units` ***is*** required.

Technically, the use of `units` is optional, but highly recommended, not only for program readability but to ensure the program will operate correctly with or without MKS Units enabled (more below).

As the test program is compiled, when the C preprocessor detects a units macro it multiplies the preceeding `value` by the `units` value, yielding a scaled value. As indicated in the table below, multiplier values change when using MKS Units (in fact it is the units macro mechanism which allows a test program to function correctly whether using MKS Units or not).

The `units` macros are shown in the table below:

Table 3.3.0.0-1 Voltage, Current, and Timing Units

Units Macro	Definition	Multiplier	
		Non-MKS Units	MKS Units
UV	Microvolts	1	1.0e-6
MV	Millivolts	1.0e3	1.0e-3
V	Volts	1.0e6	1
NA	Nanoamps	1	1.0e-9
UA	Microamps	1.0e3	1.0e-6
MA	Milliamps	1.0e6	1.0e-3
A	Amps	1.0e9	1
PS	Picoseconds	1	1.0e-12
NS	Nanoseconds	1.0e3	1.0e-9
US	Microseconds	1.0e6	1.0e-6

Table 3.3.0.0-1 Voltage, Current, and Timing Units *(Continued)*

Units Macro	Definition	Multiplier	
		Non-MKS Units	MKS Units
MS	Milliseconds	1.0e9	1.0e-3
S	Seconds	1.0e12	1
Note: MKS Units were first available in software release v2.8.7.			

The units macros allow user code to program voltages, currents, and time related functions using familiar terminology. For example, to set the pin electronics drive high voltage level to three volts, the following function is used:

```
vih( 3 V );
```

This could also be written as

```
vih( 3000 MV );
```

or...

```
vih( 3000000 );
vih( 3.0e-6 );
```

But... these latter two methods SHOULD NOT BE USED because this code WILL NOT FUNCTION correctly when switching from non-MKS units to MKS Units. And, it is less readable and forces others reading the test program to know the default units value for voltages vs. currents vs. timing values (which prior to MKS Units were different).

The units macros are also usable (recommended) when variables are used to store voltage, current and/or time values. However, careful attention to the scope of the macro is required to ensure desired results: use parentheses:

	Expression	Non-MKSUnits	MKSunits
1	`double d = 0;`	0	0
2	`d += 1;`	1	1
3	`d += 1 V;`	1000001	2
4	`d += 1 + 1 V;`	2000002	4

	Expression	Non-MKSUnits	MKSunits
5	`d += 1 + (1 V);`	3000003	6
6	`d += (1 + 1) V;`	5000003	8
The output was generated using: `output("%0.0f", d );`			

Regarding these results, note the following:

- These examples use the V units macro, suggesting a voltage value.
- When using non-MKS Units, the default voltage units are uV. In the examples above note that it is not until the V macro is used that the output is scaled to uV.
- When using MKS Units, the default voltage units are volts (V). Thus 1 = 1 Volt as does (1 V).
- Note that except for the use of parenthesis that the last 3 expressions are much the same. However, the result is quite different.

The following example shows why the units macro should always be used:

	Expression	Non-MKSUnits	MKSunits
1	d = 1 V;	1000000.000000	1.000000
2	d += 1 UV;	1000001.000000	1.000001
3	d += 1;	1000002.000000	2.000001
The output was generated using: `output(" #8 => %0.6f", d );`			

In this example, the intent is to twice add 1uV to the initial 1V value. When using non-MKS Units voltage values are in uV and the desired operation is obtained i.e. 1 = (1 UV); When this program is migrated to use MKS Units 1 = 1V and as can be seen, the result of last expression is quite different than desired.

3.3.1 MKS Units

With the introduction of the Lightning Test System, a new (as compared to Maverick-I/-II) form of mathematical units, called MKS Units, was used when programming the voltage, current and time parameter values. The following topics are discussed in this section:

Background

MKS stands for Meters, Kilograms, Seconds, one of two recognized metric units systems (the other is CGS and is not used).

With the introduction of the Lightning Test System, the software for all system types supports MKS units. The method used to migrate legacy programs (more below) is quite simple. Once migrated, all existing Nextest functions support values specified in MKS units.

Note: initially, only the Lightning software requires the use of MKS Units. However, as Lightning features are migrated to other system types this requirement will become more common. For the most part, migration of existing programs is automatic and transparent. However, depending on coding styles used in the older test program, manual intervention may be required. See Migration Issues.

Legacy Units

Prior to adding MKS, three non-natural (non-MKS) base units were used in Nextest software:

- uV (for voltages)
- nA (for currents)
- pS (for timing)

Units support consisted of a set of C macros, which were used as units tokens. For example, the following function set VIL = 350mV:

```
vil( 350 MV ); // Prototype = void vil( double Value );
```

In this example, the MV token is actually a Nextest defined C macro which multiplies the base value (350) by 1,000 (1e3) to obtain 350,000uV. Note that several other methods could be used to obtain the same result:

```
vil( 350000 UV );
vil( 350000 );    // BAD form, more below
vil( 3.5 * 1e5 ); // BAD form, more below
vil( 350 * 1e3 ); // BAD form, more below
```

Note that from a program migration standpoint, if the methods used in the last 3 examples are used, migration requires manual intervention. More below.

The Lightning Test System introduced two key units requirements:

- MKS units support is needed for effective (and conventional) use in mixed signal test applications. The number of parameter types has increased (i.e. frequency, dB, radians, etc.), as has the range of some existing parameter types. Substantial mathematical operations are common and the use of MKS units greatly simplifies decimal point management.
- Maverick-I/-II and Magnum 1/2/2x test programs must operate correctly, without modification, even if recompiled.

The solution documented here addresses both requirements (unless the wrong notation was used, more below).

MKS Units

The software defines the following new data types for MKS units support. These are used as expected by the name:

`MKSVolts`	for Voltage values, in Volts
`MKSAmps`	for Current values, in Amps
`MKSFrequency`	for Frequency values, in Hz
`MKSTime`	for Time values, in Seconds
`MKSPeriod`	for Period values, in Seconds

Definitions

`Legacy Units Mode`	The test program only uses legacy units: uV, nA, pS.
`MKS Units Mode`	The test program only uses MKS units.

Enabling MKS Units

The following steps outline how MKS units are enabled:

1. A new preprocessor symbol is defined to enable MKS Units Mode:

   ```
   #define USE_MKS_UNITS
   ```

   For backwards compatibility, Legacy Units Mode is the default i.e. `USE_MKS_UNITS` must be explicitly added to the test program to enable MKS Units Mode.

2. If used, `USE_MKS_UNITS` MUST be defined before *TestProgApp/public.h* is `#include`'ed in a given source file. This is required for the system software to adapt to which units are used. For this reason, it is recommended that `USE_MKS_UNITS` be defined in the test program's *tester.h* file or equivalent. For example, in a typical *tester.h* file:

   ```
   #define USE_MKS_UNITS
   #include "TestProgApp/public.h"
   ```

   This also ensures that all source files use the same units methodology, which is required. All user source files must be compiled in the same mode or a fatal load-time error will be reported.

Conditional Definition of the Legacy Units Macros

Beginning with the introduction of the Lightning Test System, the definition of the various legacy units macros (`V`, `MV`, `US`, etc.) depends on whether the test program defines `USE_MKS_UNITS`. As noted above, the default is to use the Legacy Units Mode.

As an example, consider (`2.4 V`). In Legacy Units Mode, the `V` macro multiplies 2.4 * 1e6 = 2,400,000uV. If `USE_MKS_UNITS` is defined, the operation of the `V` macro is redefined, to not modify the base value, thus 2.4 V = 2.4 Volts.

Migration Issues

The one migration issue which cannot be automatically handled relates to how the user might scale values in their C code. Consider the following:

```
output (" Vih = %f Volts", vih( t_pe1 ) / 1e6 );     // Bad !!!
output (" Vih = %f Volts", vih( t_pe1 ) / (1 V) );   // Good
```

Both examples scale the value returned by `vih( )` (in uV) to obtain the desired value in Volts. Using the latter form i.e. divide by (1 V) makes program migration transparent (and is also better form, more readable, etc.). While dividing by `1e6` only works correctly in the Legacy Units Mode, in MKS Units Mode the value printed will be off by a factor of `1e6`.

Usage Issues

As noted above, the MKS units implementation provides for easy migration of Maverick-I/-II test programs. Fundamentally, all MKS units (`MKSVolts`, `MKSFrequency`, etc.) are a `double`, however, because the MKS units data types are actually C++ objects, additional considerations exist. Note the following:

- Assignments between `double` variables and MKS units variables (`MKSVolts`, etc.) may require that a cast be performed:

```
double d1 = 1.0;
MKSAmps a1 = 1.0;
a1 = (MKSAmps) d1;     // Cast required
d1 = a1;               // Cast not required, class handles it
```

- The C mathematical operators (+, -, *, /, etc.) operate only on `double`, `int`, `float`, etc. These operators are not overloaded to support `MKSVolts`, `MKSFrequency`, etc. Below are several examples showing correct and incorrect usage.

  The following each generate a compile-time error, "*operator =' is ambiguous*"

```
MKSAmps a1;
a1 = 1 MA + 2 MA;                 // Bad
a1 = (1 MA + 2 MA);               // Bad
a1 = (MKSAmps) 1 + 2;             // Bad
a1 = (MKSAmps) 1 + (MKSAmps) 2;  // Bad
```

  The following all compile OK and provide the expected result. Note both the cast and the use of parenthesis:

```
a1 = (MKSAmps) (1 + 2 );       // 3Amps (3.000)
a1 = (MKSAmps) (1 MA + 2 MA );// 3mA (0.003)
a1 = (MKSAmps) (1 + 2 ) MA;   // 3mA (0.003)
```

  The following compile OK and provide the expected result:

```
MKSAmps a1, a2, a3, a4;
a1 = 1.0 MA;                // 1mA (0.001)
a2 = 2.0 MA;                // 2mA (0.002)
a3 = (MKSAmps) (a1 + a2);   // 3mA (0.003)
```

The following all generate a compile-time error, "*operator =' is ambiguous*"

```
a3 = a1 + a2;                       // Bad
a3 = (MKSAmps) a1 + a2;             // Bad
a3 = (MKSAmps) a1 + (MKSAmps) a2;   // Bad
```

The following compiles OK, but the result may not be as expected:

```
a4 = (MKSAmps) (a1 MA + a2 MA);// Result = 3uA, not 3mA
```

The following compiles OK and provide the expected result:

```
double d1, d2, d3, d4;
d1 = 1.0;
d2 = 2.0;
d3 = d1 + d2;                       // 3.000
d4 = d3 MA;                         // 0.003

// Protos: void ms_dps_current_high( MKSAmps value, ... );
ms_dps_current_high( d3,    ... );    // 3.0
ms_dps_current_high( d3 MA, ... );    // 0.003
ms_dps_current_high( d4,    ... );    // 0.003
ms_dps_current_high( d4 MA, ... );    // 0.000003
```

The following examples are included here NOT because the problems they represent are caused by MKS units, but rather because the error messages seem to indicate a problem with MKS values. The root issue here is the improper use of the value '0' (zero) for the last argument, where the proper method is to use an appropriate enumerated type value (`id_msdps1`, etc.):

```
ms_dps_current_high( d3 MA, t_mspc4, id_msdps1 ); // OK
ms_dps_current_high( d3 MA, t_mspc4, 0 );         // Bad
```

```
Error Message: none of the 6 overloads can convert parameter 1
               from type 'class MKSAmps'
```

```
MKSAmps a1 = ms_dps_current_high( t_mspc4, id_msdps1 );// OK
MKSAmps a1 = ms_dps_current_high( t_mspc4, 0 );     // Bad
```

```
Error Message: binary '=' : no operator defined which takes a
               right-hand operand of type 'void' (or there is no
               acceptable conversion)
```

3.4 Special Data Types

3.4.1 __int64

See Special Data Types.

Some of the tester registers require an integer value greater than 32 bits. Below is an example of using the `__int64` data type:

```
__int64 my_variable;
```

There is a double underscore before the `int64`. You can search for `__int64` in the Visual C++ manual for more information.

By default, `__int64` is a signed value. To obtain unsigned operation, use:

```
unsigned __int64 my_variable;
```

To print `__int64` values use format(s) which include "`I64`" plus one of the other integer format characters (d, x, etc.). For example:

```
__int64 val = 99;
output (" %I64d %-0.3I64d 0x%I64x\n", val, val, val );
```

3.5 Types, Enums, etc.

See Software.

Description

The following enumerated types are used in support of various software functions.

Usage

The `HDTesterPin` enumerated type is used to identify individual tester pins. This includes signal pins, DPS and HV pins:

Note: HDTesterPin
```
a_1,a_2,... snip ...,a_640,
b_1,b_2,... snip ...,b_640,
a_dps1a,a_dps2a,... snip ...,a_dps40a,
a_dps1b,a_dps2b,... snip ...,a_dps40b,
b_dps1a,b_dps2a,... snip ...,b_dps40a,
b_dps1b,b_dps2b,... snip ...,b_dps40b,
a_hv1, a_hv2,... snip ...,a_hv80,
b_hv1, b_hv2,... snip ...,b_hv80,
```
the declaration above is simplified. The maximum values shown are those actually usable given the maximum number of Sites-per-Controller supported (10).

The HSBBoard enumerated type is used to identify a specific Site Assembly Board (HSB):

```
enum HSBBoard { t_hsb1, t_hsb2, t_hsb3, t_hsb4,
                t_hsb5, t_hsb6,t_hsb7, t_hsb8,
                t_hsb9, t_hsb10, t_hsb11, t_hsb12,
                t_hsb13, t_hsb14, t_hsb15, t_hsb16,
                t_hsb17, t_hsb18, t_hsb19, t_hsb20,
                t_hsb21, t_hsb22, t_hsb23, t_hsb24,
                t_hsb25, t_hsb26, t_hsb27, t_hsb28,
                t_hsb29, t_hsb30, t_hsb31, t_hsb32,
                t_hsb33, t_hsb34, t_hsb35, t_hsb36,
                t_hsb37, t_hsb38, t_hsb39, t_hsb40, t_hsb_na };
```

The PFState enumerated type is used as the return value from the various test functions (funtest(), etc.):

```
enum PFState { FAIL = 0, PASS = 1, MULTI_DUT = -1 };
```

The PatStopCond enumerated type is used to specify a test pattern execution stop option using funtest(), ac_partest(), ac_test_supply(), hv_test_supply(). Note that only the values shown below are implemented (other values are defined but are not shown):

```
enum PatStopCond {  finish, error, fullec, LEC_only_errors,
                    LEC_first_vectors, LEC_last_vectors,
                    LEC_before_error, LEC_after_error,
                    LEC_center_error };
```

The TesterFunc enumerated type is used to identify a specific test pattern data source:

```
enum TesterFunc {
   t_cs1, t_cs2, t_cs3, t_cs4, t_cs5, t_cs6, t_cs7, t_cs8,
   t_d0,  t_d1,  t_d2,  t_d3,  t_d4,  t_d5,  t_d6,  t_d7,  t_d8,
   t_d9,  t_d10, t_d11, t_d12, t_d13, t_d14, t_d15, t_d16, t_d17,
   t_d18, t_d19, t_d20, t_d21, t_d22, t_d23, t_d24, t_d25, t_d26,
   t_d27, t_d28, t_d29, t_d30, t_d31, t_d32, t_d33, t_d34, t_d35,
   t_y0,  t_y1,  t_y2,  t_y3,  t_y4,  t_y5,  t_y6,  t_y7,
   t_y8,  t_y9,  t_y10, t_y11, t_y12, t_y13, t_y14, t_y15,
   t_x0,  t_x1,  t_x2,  t_x3,  t_x4,  t_x5,  t_x6,  t_x7,
   t_x8,  t_x9,  t_x10, t_x11, t_x12, t_x13, t_x14, t_x15,
   t_x16, t_x17
   t_drive_low, t_drive_high, t_strobe_low, t_strobe_high,
   t_strobe_valid, t_strobe_mid, t_tri_state, t_scan,
   t_lvm, t_tf_na };
```

The PSNumber enumerated type is used to identify a specific Pin Scramble Map entry. The value PS_na is not normally used. See Pin Scramble Macros, data_strobe(), set_ps(), var_pinfunc():

```
enum PSNumber {
   PS1,  PS2,  PS3,  PS4,  PS5,  PS6,  PS7,  PS8,
   PS9,  PS10, PS11, PS12, PS13, PS14, PS15, PS16,
   PS17, PS18, PS19, PS20, PS21, PS22, PS23, PS24,
   PS25, PS26, PS27, PS28, PS29, PS30, PS31, PS32,
   PS33, PS34, PS35, PS36, PS37, PS38, PS39, PS40,
   PS41, PS42, PS43, PS44, PS45, PS46, PS47, PS48,
   PS49, PS50, PS51, PS52, PS53, PS54, PS55, PS56,
   PS57, PS58, PS59, PS60, PS61, PS62, PS63, PS64,
   PS_na };
```

The PatternState enumerated type is used by pattern_state() to identify the execution state of the last test pattern executed:

```
enum PatternState { PATTERN_RUNNING,
                    PATTERN_PAUSED,
                    PATTERN_DONE,
                    PATTERN_STOPPED };
```

The following array data types are provided to simplify function prototype syntax. These are all specializations of the C++ CArray data type:

```
typedef CArray< BYTE,BYTE > ByteArray;
typedef CArray< short,short > ShortArray;
typedef CArray< int,int > IntArray;
typedef CArray< WORD,WORD > WordArray;
typedef CArray< long,long > LongArray;
typedef CArray< DWORD,DWORD > DWordArray;
typedef CArray< double,double > DoubleArray;
typedef CArray< float, float > FloatArray;
typedef CArray< __int64,__int64 > Int64Array;
typedef CArray< CString,CString > CStringArray;
```

3.6 Magnum System Type Get Function

See Software.

Description

The `is_magnum()` function may be used on Magnum Test Systems to determine the system type (Magnum 1, 2 or 2x).

Usage

```
BOOL is_magnum( DWORD *version = 0 );
```

where **`version`** is a pointer to an existing `DWORD` variable used to return the Magnum system type value. The value returned will be:

- 1 when `is_magnum()` is executed on a Magnum 1.
- 2 when `is_magnum()` is executed on a Magnum 2.
- 3 when `is_magnum()` is executed on a Magnum 2x.

`is_magnum()` returns TRUE when executed on any Magnum system type, otherwise FALSE is returned. If FALSE is returned the value in **`version`** is invalid.

Example

```
DWORD mtype;
if( is_magnum( &mtype ))
   output( "Executed on Magnum-%d", mtype );
else
   output( "Executed on NON-Magnum system );
```

3.7 output(), warning(), fatal(), vFormat()

See Software.

Description

The `output()`, `warning()` and `fatal()` functions are used to generate messages in the UI - User Interface output window(s).

> Note: the standard C `printf()` function can not be used for this application. For file I/O applications `fprintf()` operates as desired.

With one notable exception, the format option arguments for these functions are the same as for `printf()`. The exception is that all three functions automatically include a *newline* character ('\n'). To suppress this, two back slash characters ("\\") must be used (the first to escape the second).

By default, all output uses a fixed font and font attributes. Beginning in software release h3.3.xx some font attributes may be manipulated (color, font size, bold, italic, underline, etc.). See Output/Warning/Fatal Text Format Options. In addition, messages generated by `warning()` and `fatal()` will be displayed using red text. Some right-mouse selection options in UI Output Window allow saving messages in Rich-text format (*.rtf*) to allow saved text to include these font attributes (saving as plain text does not).

Messages can be logged to a text file using `ui_OutputFile`. However, a more versatile mechanism is described in Redirecting Output Messages.

A user-defined prefix can be added to any message using `ui_OutputFormat`. However, a more versatile mechanism is described in Redirecting Output Messages.

The `vFormat()` function can be used to create a `CString` using syntax similar to `printf()`. This is typically used to create a `CString` containing a combination of constant characters, integers, floats, etc.

Usage

```
void output( const char *format, ... );
void warning( const char *format, ... );
void fatal( const char *format, ... );
CString vFormatVA( LPCTSTR format, va_list va );
CString vFormat( LPCTSTR format, ... );
```

where:

`output()` is a C-function, with arguments consistent with the standard C `printf()` function, except that an automatic end-of-line is included by `output()`. **`output()`** generates no special message prefix.

`warning()` operates the same as **`output()`** except that it automatically generates the following prefix message: `Warning:`. Beginning in software release h3.3.xx all warnings (including those generated by the system software) will be displayed using red text.

`fatal()` operates the same as **`output()`** except that it automatically generates the following prefix message: `Error:` and, if executed in the SITE process, the test program is terminated. Beginning in software release h3.3.xx all fatal messages (including those generated by the system software) will be displayed using red/bold text.

`vFormat()` assembles a `CString` from the formatting tokens and corresponding argument list. See examples.

Examples

```
output (" Hello World"); // Prints: "Hello World"
CString warn_msg = "Will Robinson";
warning ("%s", msg); // Prints: "Warning: Will Robinson"
int val = 10;
CString msg = " Max value is => ";
CString s = vFormat("Msg includes %d(int) %f(float)", 13, 1.333 );
// Prints: "Error: Max value is => 10” and unloads the program
fatal ("%s %n", msg, val );
```

3.7.1 Output/Warning/Fatal Text Format Options

See `output(), warning(), fatal(), vFormat()`.

Note: first available in software release h3.3.xx.

Description

The `output()`, `warning()` and `fatal()` functions may be used in test programs to generate user-defined messages in UI's output window(s).

By default these messages use the system default font style (font, size, color, not-bold, not-italic, not-underlined, etc.). Using the controls noted below it is possible to manipulate some font style attributes applied to user-defined messages. This includes font-size, color, bold, italic and under-line format attributes (but not the specific font used). For example:

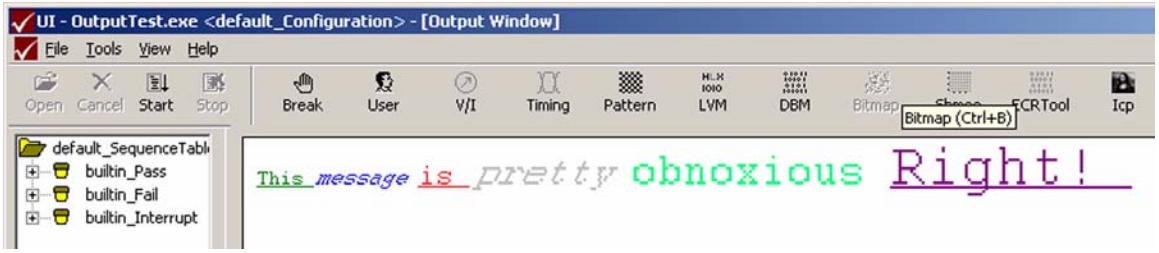


Figure-36: Example Output

Note the following:

- During the initial program load, the format styles for all UI output windows are reset to default values. Except as noted below for `warning()` and `fatal()` the system software does not otherwise change any format styles.
- The syntax used to manipulate text format styles is similar to HTML tags. The supported UI Output/Warning Format Tags must be enclosed in angle braces (i.e. `<tag>`). The following table describes the supported format tags:

Table 3.7.1.0-1 UI Output/Warning Format Tags

Open Tag	Close Tag	Attribute
<I>	</I>	*Italic* Font

Table 3.7.1.0-1 UI Output/Warning Format Tags *(Continued)*

Open Tag	Close Tag	Attribute
<B>	</B>	**Bold** Font
<U>	</U>	Underline Font
<H6> <H5> <H4> <H3> <H2> <H1>	</H6> </H5> </H4> </H3> </H2> </H1>	H = Heading Size (font size) H6 Font (default font size) H5 Font H4 Font H3 Font H2 Font H1 Font
<COLOR=color> where color = BLACK BLUE GRAY GREEN LIME MAROON PURPLE RED SILVER WHITE YELLOW	</COLOR>	COLOR = Font color BLACK Font BLUE Font GRAY Font GREEN Font LIME Font MAROON Font PURPLE Font RED Font SILVER Font Font (white = can't see it at all) YELLOW Font (yellow = hard to see)

- Tags and values are not case sensitive.
- Tags can be opened (<tag>) and closed (</tag>) independently, in separate calls to `output()` and/or `warning()` and/or `fatal()`.

- A given style attribute (color, bold, italic, etc.) is a property of the UI output window which displays a given message. Once opened (i.e. <tag>) a given attribute will be applied to all user-defined messages in that window until closed (i.e. </tag>).
- User-defined font styles do not affect messages generated by Nextest software.
- Format tags can be nested and overlapped. For example:

```
output("<B> Start bold text from here...");
output("Continues with bold text…<I>and italic...");
output("Closing bold…</B> and closing italic...</I>");
output("End of the message.");
```

  Generates the following output:

  Start bold text from here...
  Continues with bold text…*and italic...*
  Closing bold… *and closing italic...*
  End of the message.

- When a format tag is opened the previous value for that attribute is saved on a stack. When a given tag is closed that attribute is popped from its stack and the next most recent value for that tag will take effect. For example, given the following:

```
output("<COLOR=maroon>Some text");
output("<COLOR=blue>More text");
output("This text will be blue");
output("</COLOR>"); // Pops color stack, could be anywhere
output("This text will be maroon");
```

- The font style attribute stacks are limited to 1000 entries. When a given stack is full, no additional changes to that font attributes will saved to the stack until one or more related tags are popped from the stack.
- When the `warning()` function executes it automatically pre-pends "`Warning:`" to the text specified by the user. As shown, the `warning()` function also sets <COLOR=RED>, which causes all warning test to be RED, but it does not put this on the font attribute stack. Then, at the end of the warning text the `warning()` function also closes the red color, using </COLOR>. However, if the user includes a <COLOR> tag in the body of warning text two things occur:
  - The specified color will be applied to the appropriate text. For example:

```
warning("<COLOR=green>My green Warning");
```

  generates:

  Warning: My green Warning

  - The `warning()` function does not issue the close color tag (i.e. no </COLOR>). This means that any color specified in warning text strings will carry-over to subsequent executions of `output()`, `warning()` and `fatal()` unless the tag is closed by user code.
- Similarly, fatal messages are displayed using **RED-bold** text, and the color operations described in the previous bullet apply (although since fatal unloads the test program this mostly doesn't matter).
- Closing a tag which is not currently open is ignored.
- The back-slash character can be used to *escape* a tag to allow the tag's text to be seen in the output text. For example, try:

```
output("Use \\<B> to open the bold text tag");
```

- Some right-mouse selection options in UI Output Window allow saving messages in Rich-text format (*.rtf*) to allow saved text to include these font attributes (saving as plain text does not).

Example

The following example generates the Example Output:

```
output("<COLOR=Green><U>This </COLOR></U>\\");
output("<COLOR=Blue><H5><I>message </COLOR></H5></I>\\");
output("<COLOR=Red><H4><U>is </COLOR></H4></U>\\");
output("<COLOR=Silver><H3><I>pretty </COLOR></H3></I>\\");
output("<COLOR=Lime><H2>obnoxious </COLOR></H2>\\");
output("<COLOR=Purple><H1><U>Right! </COLOR></H1></U>");
```

This example closes each tag used within each `output()` statement.

The same text can be obtained using a single output statement (line-breaks added for clarity):

```
output("<COLOR=Green><U>This </U>
       <COLOR=Blue><H5><I>message </I>
       <COLOR=Red><H4><U>is </U>
       <COLOR=Silver><H3><I><B>pretty </I></B>
       <COLOR=Lime><H2>obnoxious
       <COLOR=Purple><H1><U>Right! ");
```

Note, that the 2nd example (unlike the 1st) does not close some attribute tags thus some of the attribute stacks will not end the same. This can affect subsequent text.

3.7.2 Redirecting Output Messages

See `output(), warning(), fatal(), vFormat()`.

Description

In simple terms, the `intercept()` function was designed to redirect messages from the UI - User Interface display to a text file. The `fumble()` function is used to restore default operation.

More correctly, the `intercept()` function is used to register a user-written call-back function which will then be called each time a message would be send to a UI - User Interface output window. Then, user-written code in the call-back function determines the fate of the message. For example, code in the call-back function could cause the message to be output to the UI display and/or to a text file, to a User Dialog, or discarded. And, `output()`, `warning()`, and `fatal()` messages can be treated separately.

The call-back function name and internal code is determined by the user programmer. The prototype definitions of the call-back function are defined by Nextest (see Usage).

The `fumble()` function is used to un-register the user-written call-back and cause the built-in functions to be called.

A registered call-back function is scoped to the process which calls `intercept()`. This means that messages may be separately handled for messages generated in Host vs. Site vs. User Tools processes. Using conditional code in the call-back it is possible to process messages uniquely for each `site_num()`.

It is possible to register more than one call-back function. In operation, the most recently registered call-back function is called first. If it returns TRUE the next most recently registered call-back function is then called, etc. If any call-back returns FALSE the process stops. This allows a series of call-backs to be sequentially executed.

If the earliest registered user call-back function returns TRUE the built-in function is also called, causing the messages to appear normally in the UI - User Interface display. If the earliest registered user call-back function returns FALSE the built-in function is not called.

The currently executing user call-back does not intercept messages generated by calling `output()`, `warning()`, or `fatal()` from within the call-back function code. The messages are intercepted by any other registered call-backs which execute after the currently executing call-back completes.

Note: different versions of `intercept()` and `fumble()` functions are also used to to intercept User Variables. See Intercepting User Variables.

Usage

```
void intercept( BOOL (*func)( char type, CString string ) );
void intercept(BOOL (*func)(  char type,
                              CString string,
                              void *data ),
               void *data );
BOOL fumble( BOOL (*func)( char type, CString string ) );
BOOL fumble( BOOL (*func)(  char type,
                            CString string,
                            void *data ) );
```

where:

`*func` is a pointer to the user-written callback function.

`fumble()` returns TRUE if the specified call-back function is successfully un-registered, otherwise FALSE is returned. The arguments to `fumble()` identify which call-back to un-register; the variation using the optional `void *data` argument is needed when the call-back name is overloaded with different `*data` arguments.

The call-back prototype definitions include:

`BOOL` call-back return value. See Description.

`type` is a single character, and will be one of '**`o`**', '**`w`**', or '**`f`**', corresponding to `output()`, `warning()`, or `fatal()`. This allows code in the user call-back to identify the nature of the message and conditionally treat each type of message differently. See Examples.

`string` is the message. This allows the call-back code to access to the message, and determine its fate. See Examples.

`*data` is optional, and is not used by the system software in any way. It is a generic pointer, allowing additional information to be passed to the call-back as needed by the user. Within the call-back code this pointer should be cast to the proper data type before use.

Examples

Example 1:

The function my_output_callback() is registered from code called in the Host Begin Block, thus my_output_callback() will be called each time there is a output(), warning(), or fatal() message from code executing in the Host process:

```
// Call-back function for intercept(). All output from output(),
// warning(),and fatal() is intercepted here and output to UI
// with a custom prefix. The TRACE macro is useful when running in
// UI Site or Host debug mode. It generates output in the
// Developer Studio output window.
BOOL my_output_callback( char type, CString string ) {
   // ID if message was output(), warning(), or fatal()
   char *id = 0;
   if ( type == 'w' ) id = "My Warning: ";
   if ( type == 'f' ) id = "My Fatal: ";
   if ( type == 'o' ) id = "My Output: ";
   // Display the message
   TRACE( "%s%s", id, string ); // For Developer Studio Output only
   output("%s%s", id, string ); // For UI output
   return TRUE; // Pass string to UI
}
HOST_BEGIN_BLOCK( HBB1) { // See HOST_BEGIN_BLOCK()
   // ... other code here ...
   // Register intercept() callback function in the Host processes
   intercept( my_output_callback );
   // ... other code here ...
}
```

Example 2:

This example saves just the warning(), and fatal() messages, produced during the execution of problematic_function(), to the text file D:/temp/my_output.txt. Since intercept() is executed in the Site Begin Block only messages generated when problematic_function() is executed in a Site process will be saved:

```
void problematic_function() { // Any written code...
   output ( "This message is not saved to the file" );
   warning( "This message is saved to the file" );
}
```

```
BOOL my_output_callback ( char type, CString string, void *data ) {
   if ( type != 'o' ) { // ID warning() and fatal() messages
      // Cast the void arg pointer to the proper type
      FILE *file = (FILE *) data;
      fprintf ( file, "Type => %c: %s", type, string );
   }
   return TRUE; // Send all messages to UI too
}
SITE_BEGIN_BLOCK( sb1 ) { // See Site Begin Block
   // ... other code here ...
   FILE *fp = fopen( "D:/temp/my_output.txt", "w" ); // Open file
   if ( fp ) {
      intercept( my_output_callback , fp ); // Register call-back
      problematic_function();
      fumble( my_output_callback ); // Un-register call-back
      fclose( fp ); // Close output file
      output("Output is back to default");
   }
   // ... other code here ...
}
```

Example 3:

This example uses the `output()` function from within the `my_output_callback()` code to replace `warning()` with `output()`.

```
BOOL my_output_callback ( char type, CString string ) {
   if ( type == 'w' ) {
      // Suppress final newline, because the original warning()
      // includes it
      output( "This would have been a warning but %s\\", string );
      return FALSE; // Don't pass warning()s to UI
   }
   return TRUE; // Do pass output() and fatal() to UI
}
SITE_BEGIN_BLOCK( SBB1 ) { // See Site Begin Block
   // ... other code here ...
   intercept( my_output_callback ); // Register call-back
   output( "This line output as-is" ); // Outputs to UI normally
```

```
    warning( " my_output_callback() modified this one." );
    // ... other code here ...
}
```

3.8 Configuring the Tester to the DUT

The following topics are covered in this section:

- DUT Board Connection Considerations
- DUT Pins
- Pin Assignment Table
  - ASSIGN_64DUT Work-around
  - Sites-per-Controller
  - Shared Tester Pins
  - `testerpin_name()`
  - `testerpin_value()`
  - `testerpin_offset()`
  - Pin Iteration
- Pin Lists
  - DUT-specific Pin Lists
  - `pinlist_create(), pinlist_destroy()`
  - `pin_info()`
  - `testerpin_name()`
  - `testerpin_value()`
  - `testerpin_offset()`
  - Pin Iteration
  - `all_dps()`
  - `no_dps()`
  - `all_hv()`
  - `no_hv()`
  - `all_pe()`
  - `no_pe()`
- Pin Scramble Functions & Macros
  - Overview
  - Pin Scramble Macros
  - Default Pin Scramble Map
- VIHH Maps

3.8.1 DUT Board Connection Considerations

See Configuring the Tester to the DUT, Magnum 1, 2 & 2x Parallel Test.

Using Magnum 1/2/2x, the following issues must be considered when designing a new DUT board. These issues reflect various constraints imposed by the tester hardware architecture:

1. Using Magnum 1/2, the 128 digital PE pins of a given Site Assembly Board are divided into 2 groups: A-pins and B-pins, see PE Sub-site Architecture. Each pin of sub-site A is paired with one corresponding pin of sub-site B. For example, a_1 and b_1, a_13 and b_13, etc. During functional tests, each pin of a given pin-pair (see Functional Pin-pairs) receives identical drive, strobe and I/O signals and pattern data. This normally means that, for example, when pin a_1 is connected to a given pin of DUT-1 that pin b_1 is connected to the same pin of DUT-2. See Pin Assignment Table, Multi-DUT Test Program, Parallel Test Operation, etc. When the expected pin assignments are used odd numbered DUTs will be connected to A-pins and even numbered DUTs to B-pins. A given DUT will never have connections to both A-pins and B-pins.
2. Using Magnum 1/2, each DUT Power Supply (DPS) has two switchable outputs, which can be programmed to identical voltages or separately to different voltages. Some of the DPS hardware is shared between the two outputs, which might affect the user's DUT board design considerations.
3. Proper operation of DUT-based test pattern conditional branch operations depends upon certain DUT-to-Pin relationships. See MAR Multi-DUT Branch-condition Operands, DUT-pin to Tester-pin Connection Requirements.
4. When using pins in MUX Mode, the odd pins of a pin-pair(for example, a_1 and b_1) are connected to the DUT and the next higher even numbered pins (for example, a_2 and b_2) cannot be used at the DUT.
5. In Double Data Rate (DDR) Mode, hardware constraints affect how DDR A-cycle vs. DDR B-cycle errors are captured. This affects memory patterns when logging errors to the Error Catch RAM (ECR) and both Memory and Logic patterns when datalogging; specifically some pins cannot be captured/logged at the same time as other pins, which might affect DUT board design considerations. See DDR Fail Signal MUX, DDR Fail Signal MUX: Logic Error Catch, DDR Fail Signal MUX: Memory Error Catch.
6. Each Site Assembly Board has two Error Catch RAM (ECR). Each ECR can capture errors from up to 36 PE channels (36 DUT pins). In order to maintain DUT board and program compatibility with future Magnum-compatible systems, at most 18 DUT pins should be assigned to each of the following pin groups:

- a_1 to a_32
- b_1 to b_32
- a_33 to a_64
- b_33 to b_64

7. Additional DUT board layout design rules are available from the Nextest Interface Solutions Group.

3.8.2 DUT Pins

See Magnum 1, 2 & 2x Parallel Test, Pin Assignment Table.

Description

Using the Magnum 1/2/2x Test System, each DUT pin must be defined once, using the `DUT_PIN` macro. This creates a `DutPin` resource which may be used elsewhere in the program. Note that this is different than, and in addition to, the methods used when programming Maverick-I/-II.

Note the following:

- The `DUT_PIN` macro will be used once, for each unique DUT pin of the device being tested, to create/define a corresponding `DutPin` resource. The `DutPin` data type is actually a *resource*, like a `PinList`, `TestBlock`, etc. See Resources.
- A given DUT pin is defined only once, regardless of how many DUT(s) are being tested in parallel.
- Each entry in the Pin Assignment Table requires a corresponding `DutPin`, including pins which are not connected to tester resources; i.e. Gnd, etc.
- The `DUT_PIN` macro is global in scope; i.e. it may not be used within the body of another macro or C-function.
- `DutPins` are used throughout the test program. Functions which require a single pin or return a single pin will specify the parameter as a `DutPin*`.
- Using Magnum 1/2/2x, the `TesterPin` data type used with using Maverick-I/-II is replaced by `HDTesterPin`. This is required due to increased number of pins available using Magnum 1, Magnum 2 and Magnum 2xI.
- `HDTesterPin` is only used in the Pin Assignment Table, where `DutPin` to tester pin mapping is specified. Except for the Pin Assignment Table, `DutPin*` is used everywhere else in the program to identify a single pin. The Nextest functions

which, using Maverick-I/-II, took a `TesterPin` argument to identify one pin are not supported using Magnum 1/2/2x. Instead, an equivalent function must be used, which uses `DutPin` instead of `TesterPin`.

- In Multi-DUT Test Programs, the Active DUTs Set (ADS) affects the operation of many functions which take a `DutPin` argument. In general, setter functions only affect pin(s) of DUT(s) which are currently in the Active DUTs Set (ADS). Getter functions return a value from the first DUT in the Active DUTs Set (ADS) (see Using Getter Functions).

Usage

```
DUT_PIN( name ){}
```

where:

name specifies the name for this `DutPin`. **name** must be a valid C identifier.

Example

The following example defines 4 pins:

```
DUT_PIN( A0 ){}
DUT_PIN( WEbar ){}
DUT_PIN( Vcc ){}
DUT_PIN( Gnd ){}
```

3.8.2.1 dutpin_info()

See Pin Lists.

Description

The `dutpin_info()` function may be used to iterate over the `DutPin`s actually used in the Pin Assignment Table selected during the initial program load. `dutpin_info()` does not report any `DutPin`(s) which, although defined, are not actually used in the currently defined Pin Assignment Table.

Usage

```
BOOL dutpin_info( int index, DutPin **dut_pin );
```

where:

index is used to select which element of the Pin Assignment Table is being accessed. DutPins are returned in the order they are entered in the Pin Assignment Table. See Example.

dut_pin is a pointer to an existing DutPin* variable used to return the indexth DutPin of the current Pin Assignment Table.

dutpin_info() returns TRUE if a valid DutPin exists at the indexth position of the Pin Assignment Table, otherwise FALSE is returned.

Example

```
DutPin *dut_pin;
for ( int index = 0; dutpin_info( index, &dut_pin ); ++index )
   output(" Index-%d => %s", index, resource_name( dut_pin ) );
```

3.8.3 Pin Assignment Table

See Configuring the Tester to the DUT, Magnum 1, 2 & 2x Parallel Test.

Description

The Pin Assignment Table specifies the relationship between DUT pin names (DutPin), DUT pin numbers, and tester pin numbers. Using Magnum 1/2I, it also determines if the resulting test program is a Multi-DUT Test Program and which DUT pins share each set of Functional Pin-pairs..

Note the following:

- The Pin Assignment Table describes the connections made, on the DUT board, between the DUT and the tester hardware. This includes signal pin connections, HV connections and DPS connections.
- A Pin Assignment Table is created using the PIN_ASSIGNMENTS macro noted below. DutPins are added to the Pin Assignment Table using the various ASSIGN macros noted below.
- Pin Assignment Tables can only be defined globally (i.e not nested, and outside of any C-function. The Test Program Wizards locates Pin Assignment Tables in the *pin_assignments.cpp* file. This form must be followed if the test program will use Logic Test Patterns which contain a VECDEF Compiler Directive.

- Once defined, the DutPins used in the Pin Assignment Table may be used throughout the test program, to refer to these connections using DUT pin names rather than pin numbers or tester channels. These same DutPins are used to define Pin Lists.
- DPS and HV pins are not automatically connected to the DUT by the system software- see `dps_connect()` and `hv_connect()`.
- Multiple Pin Assignment Tables may be defined in a test program. This allows a single test program to test DUTs with different packaging or pinout options but otherwise the same electrical characteristics. If only one Pin Assignment Table is defined in the test program it will automatically be selected as the program loads, otherwise either user-code must make the selection, using the `USE_PIN_ASSIGNMENTS()` macro in the body of a `CONFIGURATION()` macro, or the system software will require the operator to select from a list of tables as the program loads (this is not user friendly on multi-site systems).
- When the DUT to be tested requires more than 64 signal pins, the `SITES_PER_CONTROLLER` macro must be used to specify how many sites are required to test one DUT. See Sites-per-Controller.
- The `PinAssignments_find()` function can be used to get a pointer to a Pin Assignments Table. This function has little practical use.

A Magnum 1/2/2x Pin Assignment Table design supports testing multiple DUTs, in parallel. Different MACROs are used to add entries to the Pin Assignment Table and additional rules apply, as follows:

- If the `ASSIGN` macro noted above is used, the test program is, by definition, NOT a Multi-DUT Test Program, the rules noted above apply, the concept of Functional Pin-pairs does not apply and the b_nnn pins are not usable. It is not legal to mix the use of the `ASSIGN` macro with the `ASSIGN_nnn` macros noted below.

Note: the Magnum 1, Magnum 2 and Magnum 2x Programmer's Manuals do not otherwise document non-Multi-DUT Test Program programming; i.e. it is expected that Functional Pin-pairs and the associated programming methods are always used (if not, half of the available tester channels are not usable).

- To define the Pin Assignments Table for a Multi-DUT Test Program requires the use of the macros developed specifically for parallel test (`ASSIGN_2DUT`, `ASSIGN_4DUT`, etc.). When any of these MACROs are used the test program is, by definition, a Multi-DUT Test Program and the b_nnn pins of the Functional Pin-pairs are usable.

- The name of the macro used determines how many DUT(s) the program can test in parallel. For example, ASSIGN_4DUT is used when the test program will test 4 DUTs in parallel. Only one version of these macros can be used in a given Pin Assignments Table; i.e. if ASSIGN_4DUT is used ASSIGN_2DUT, ASSIGN_6DUT, etc. cannot be used.
- The appropriate macro is used once for each DutPin, to specify which tester channel is connected to that DutPin of each DUT. This is the only place in the test program in which HDTesterPins are used. For example:

```
   DUT_PIN( dp1 )

PIN_ASSIGNMENTS( example ) {
   SITES_PER_CONTROLLER( 1 );
   //          DutPin    t_dut1    t_dut2    t_dut3    t_dut4
   ASSIGN_4DUT( dp1,      a_1,      b_1,      a_23,     b_23 )
}
```

- Using these macros, note that tester signal pins are assigned in pairs (see Functional Pin-pairs). This means that, using the previous example, during functional tests, pin dp1 of both t_dut1 and t_dut2 will receive the same timing and test pattern stimulus, and pin dp1 of both t_dut3 and t_dut4 will receive the same timing and test pattern stimulus. All 4 pins do have independent PE voltages, and t_dut1/t_dut2 and t_dut3/t_dut4 do have independent timing/pattern stimulus. For example:

```
DUT_PIN( D0 ){}
DUT_PIN( D1 ){}
DUT_PIN( Cs ){}
DUT_PIN( Rd ){}
DUT_PIN( A1 ){}
DUT_PIN( A0 ){}
DUT_PIN( VCC ){}
DUT_PIN( GND ){}

PIN_ASSIGNMENTS( example ) {
   SITES_PER_CONTROLLER( 1 );
      //        DUT     DUT-1   DUT-2     DUT-3     DUT-4
      //        Pin     Tester  Tester    Tester    Tester
      //        Name    Pin #   Pin #     Pin #     Pin #
      //        ------  ------  ------    -------   -----
   ASSIGN_4DUT (D0,     a_1,    b_1,      a_17,     b_17 )
   ASSIGN_4DUT (D1,     a_3,    b_3,      a_19,     b_19 )
   ASSIGN_4DUT (Cs,     a_11,   b_11,     a_31,     b_31 )
   ASSIGN_4DUT (GND,    a_na,   b_na,     a_na,     b_na )
```

```
   ASSIGN_4DUT (Rd,     a_9,     b_9,       a_25,      b_25 )
   ASSIGN_4DUT (A1,     a_7,     b_7,       a_23,      b_23 )
   ASSIGN_4DUT (A0,     a_5,     b_5,       a_21,      b_21 )
   ASSIGN_4DUT (VCC, a_dps1a,  b_dps1b,   a_dps2a,   b_dps2b )
}
```

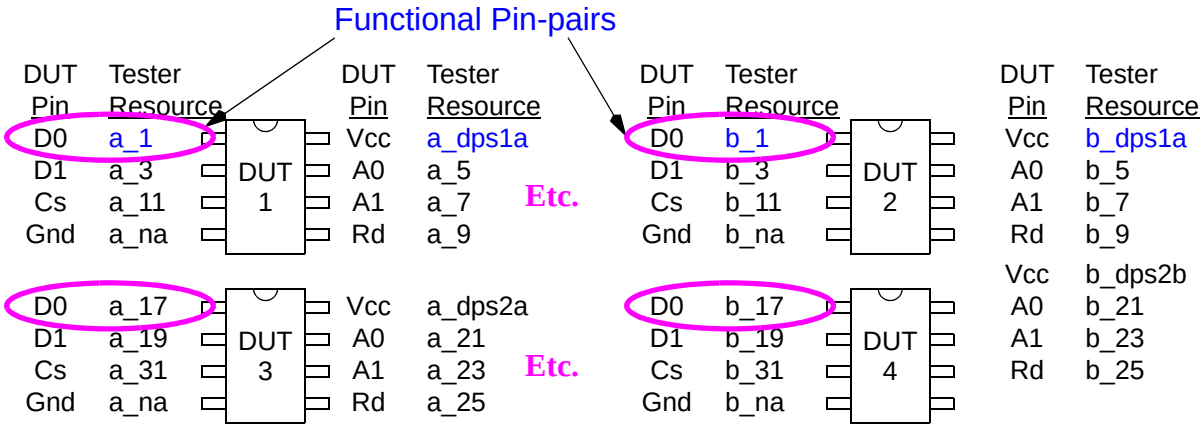


Figure-37: Multi-DUT Test Program Pin Assignments (pin_pairs)

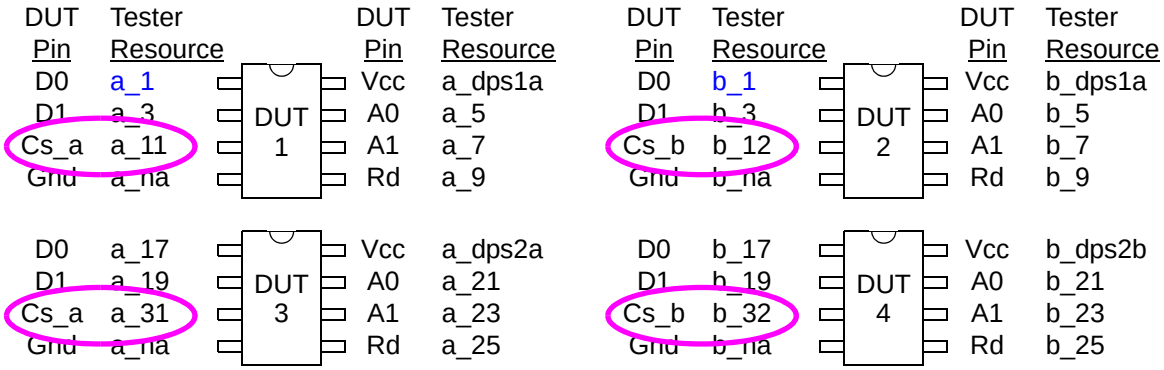
In the diagram above note the following:

- DUT-1 (`t_dut1`) and DUT-2 share Functional Pin-pairs. DUT-3 and DUT-4 share Functional Pin-pairs. Etc. As noted in the example below, it is also possible, in a Multi-DUT Test Program, for DUTs which nominally share Functional Pin-pairs to have [some] pin(s) which are independent.
- Each Pin Scramble Map will have one entry for each DUT, but the pattern data source specified for both DUTs which share Functional Pin-pairs must be identical.
- The concept of Functional Pin-pairs does not apply to pins connected to DPS or HV. The example above does follow the recommended convention.

- It may sometimes be necessary when testing some devices which, for the most part, can be tested using Functional Pin-pairs, but need a few pin(s) with independent timing and/or pattern stimulus.For each of these special DUT pins, two `DutPin`s must be defined and two `ASSIGN_xxxDUT` statements must be used. For instance, considering the earlier example, what if each DUT's chip select pin (Cs) requires completely independent timing and/or test pattern control? The following example shows the changes needed to support this:

```
DUT_PIN( Cs_a )
DUT_PIN( Cs_b )
//....
```

```
//              DutPin t_dut1   t_dut2    t_dut3     t_dut4
ASSIGN_4DUT( Cs_a,     a_11,    b_na,     a_31,      b_na )
ASSIGN_4DUT( Cs_b,     a_na,    b_12,     a_na,      b_32 )
```



In the diagram above note the following:

- For the Cs pin, two `DutPin`s are defined; `Cs_a` and `Cs_b`. Only two are ever needed (per DUT pin), regardless of how many DUTs are being tested.
- The Cs pin of 2 DUTs is connected to the primary pin of a pin-pair (`a_11`, `a_12`) and the other 2 DUTs to the secondary pin of the adjacent pins (`b_12`, `b_32`). The other pin of each of these pin-pairs is not usable for functional testing; i.e. `b_11`, `a_12`, `b_31`, and `a_32`.
- The `ASSIGN_4DUT` MACRO is used twice, once for `Cs_a` and once for `Cs_b`. When defining `Cs_a`, the Cs connections to DUT-2 and DUT-4 are specified as `b_na`. When defining `Cs_b`, the Cs connections to DUT-1 and DUT-3 are specified as `a_na`.
- Throughout the program, two `DutPin`s are used to represent the Cs pin; `Cs_a` for odd numbered DUTs and `Cs_b` for even numbered DUTs.
- Each Pin Scramble Map will have two entries for the Cs pin, one for `Cs_a` and `Cs_b`.
- The other functional pins are treated as Functional Pin-pairs.

Usage

The following macro creates a named Pin Assignment Table:

```
PIN_ASSIGNMENTS( name )
```

The following macro selects one, of possibly multiple, Pin Assignment Tables to be used. This must be done in a `CONFIGURATION()` block:

```
USE_PIN_ASSIGNMENTS( name )
```

The following macro allows the inclusion of an existing [partial] Pin Assignment Table in the definition of the Pin Assignment Table being defined:

```
INCLUDE_PIN_ASSIGNMENTS( name )
```

The following macro is used to make an external or forward declaration:

```
EXTERN_PIN_ASSIGNMENTS( name )
```

The following macros are used to insert DutPins into the Pin Assignment Table being defined, and associate one or more HDTesterPin value(s) with each DutPin. This describes the physical connections between one or more DUTs and the tester resources. The name of the macro identifies the number of required HDTesterPins arguments and thus the number of DUT(s) the program can test in parallel. A given Pin Assignment Table can use only one form of these macros By definition, using any of these macros results in a Multi-DUT Test Program:

```
ASSIGN_1DUT( DutPin, t1 )
ASSIGN_2DUT( DutPin, t1, t2 )
ASSIGN_4DUT( DutPin, t1, t2, t3, t4 )
ASSIGN_6DUT( DutPin, t1, t2, t3, t4, t5, t6 )
... snip ...
ASSIGN_32DUT(  DutPin,
               t1, t2, t3, t4, t5, t6, t7, t8, t9, t10,
               t11, t12, t13, t14, t15, t16, t17, t18, t19, t20,
               t21, t22, t23, t24, t25, t26, t27, t28, t29, t30,
               t31, t32 )
```

where:

name identifies the name of the Pin Assignment Table. Must be a valid C identifier.

DutPin identifies one DutPin.

t1, **t2**, ... **t32**, each identify one HDTesterPin value. When multiple **t**n values are specified, the first value corresponds to DUT-1, the second to DUT-2, etc.:

Value	Purpose
a_*n* b_*n*	Tester signal pin connection, where ***n*** is an integer from 1 to 64 (for Sites-per-Controller = 1) or up to 640 (for Sites-per-Controller = 10). The value ***n*** represents one tester pin electronics channel. Each Magnum 1/2 Site Assembly Board has 128 signal pins, half identified using the ***a_n*** notation, the other half using the ***b_n*** notation, see Functional Pin-pairs.
a_dps*n*a a_dps*n*b b_dps*n*a b_dps*n*b	Tester DPS connection, where **n** is an integer from 1 to 4 (for Sites-per-Controller = 1) or up to 40 (for Sites-per-Controller = 10), identifying one DPS. Each Site Assembly Board has 8 DPS, half are identified using the ***a_n*** prefix notation, the other half using the ***b_n*** prefix notation (this is to be consistent with how signal pins are numbered using the Magnum 1/2 Functional Pin-pairs architecture). The ***a*** and ***b*** suffix identifies one DPS output (each DPS has 2 outputs, see Magnum DPS Output Block Diagram).
a_hv*n* b_hv*n*	Tester HV connection, where *n* is an integer from 1 to 8 (for Sites-per-Controller = 1) or up to 80 (for Sites-per-Controller = 10), identifying one HV. Each Site Assembly Board has 16 HV, half are identified using *the* ***a_n*** notation, the other half using the ***b_n*** notation (this is to be consistent with how signal pins are numbered using the Magnum 1/2 Functional Pin-pairs architecture).
a_na b_na	No connection to test resources. Used to identify DUT pins which are not connected to the tester hardware but which the user wishes to show in the Pin Assignments Table anyway. This is most commonly used to identify ground pins.

Examples

The following is a Assignment Table for a memory device . The name of this Pin Assignment Table is soic_package. The DUT pin named data_in is connected to tester channel a_10. The DUT pin named data_out is connected to tester channel a_5. And so on. The gnd pin is not connected to any tester hardware resource thus a_na or b_na (not applicable) is used in the tester pin field for that pin. The DUT power pin, vcc, is connected to a_dps1 as shown. This is a Multi-DUT Test Program, even though it supports testing only one DUT:

```
DUT_PIN( data_in ){}
DUT_PIN( dat_out ){}
DUT_PIN( clk ){}
DUT_PIN( cs_bar ){}
DUT_PIN( vcc ){}
DUT_PIN( gnd ){}

PIN_ASSIGNMENTS( soic_package ) {
   SITES_PER_CONTROLLER( 1 ) // See Sites-per-Controller
             //   DUT Pin   Tester
             //   Name      Resource
             // --------    --------
   ASSIGN_1DUT( data_in,     a_10    )
   ASSIGN_1DUT( dat_out,     a_5     )
   ASSIGN_1DUT( clk,         a_3     )
   ASSIGN_1DUT( cs_bar,      a_1     )
   ASSIGN_1DUT( vcc,         a_dps1a )
   ASSIGN_1DUT( gnd,         a_na    )
}
```

The following Pin Assignment Table is designed to test 4 DUTs in parallel. Each DUT has a different tester resource identified for each DUT pin name (DutPin):

```
DUT_PIN( dp1 ){}
DUT_PIN( dp2 ){}
DUT_PIN( vcc ){}
DUT_PIN( vpp ){}
DUT_PIN( gnd ){}

PIN_ASSIGNMENTS( myPinAssignmentsTable ) {
   SITES_PER_CONTROLLER( 1 ) // See Sites-per-Controller
   //          DutPin    t_dut1    t_dut2     t_dut3    t_dut4
   ASSIGN_4DUT( dp1,      a_1,       b_1,      a_23,      b_23 )
   ASSIGN_4DUT( dp2,     a_21,      b_21,       a_3,       b_3 )
   ASSIGN_4DUT( vcc,  a_dps1a,   b_dps1a,   a_dps1b,   b_dps1b )
   ASSIGN_4DUT( vpp,    a_hv1,     b_hv1,     a_hv2,     b_hv2 )
   ASSIGN_4DUT( gnd,    a_na,      b_na,      a_na,      b_na )
}
```

Note the following:

- This is a Magnum 1/2 Multi-DUT Test Program, testing 4 DUTs, each with four DutPins (dp1, dp2, vcc, vpp) plus ground.

- DUT #1 (`t_dut1`) and DUT #2 (`t_dut2`) share Functional Pin-pairs, as do DUTs #3 and #4. In this example, DUT #1 and DUT #2 will receive the same functional timing edges, as will DUT #3 and DUT #4.

3.8.3.1 ASSIGN_64DUT Work-around

See Pin Assignment Table.

This section is needed because Visual Studio limits the number of arguments to a given macro to 64 and violating this limit is fatal.

Note: the information in this section was added to support testing 64 DUTs on Magnum 2x (first available in software release h3.5.xx). However, the methods noted below may be used in any Magnum test program, thus this section appears in all Magnum manuals.

In software release h3.5.xx, the number of DUTs which can be tested in parallel using Magnum 2x was increased to a maximum of 64. However, given the Visual Studio limitation noted above, it is not possible to use an `ASSIGN_64DUT` macro to support this number of DUTs in the Pin Assignment Table.

The work-around to this limitation (below) expands the underlying code represented by the macro. In the following example, `NUM_DUTS` is user-defined, to aid in seeing where the value is used (it may also be useful defining the Pin Scramble tables, see SCRAMBLE_32DUT Work-around). Regarding the `add()` function in the example:

- `add()` is a member function of the `PIN_ASSIGNMENTS` class.
- `add()` is used to specify the `HDTesterPin`s mapped to one DUT pin, one `HDTesterPin` for each DUT.
- The first argument, `obj`, is defined by the PIN_ASSIGNMENTS class and represents the pin assignments table being defined (local scope). This is used as-is as shown below.
- The 2nd argument is the DUT pin (`DutPin`) being defined.
- The 3nd argument is the count of subsequent `HDTesterPin` arguments (the `add()` function uses a variable length argument list).
- The remaining arguments are `HDTesterPin`s, specified in numerical DUT order (`t_dut1`, `t_dut2`, etc.). The quantity of `HDTesterPin` arguments must match the count specified for the 3rd argument.

The following example has 4 DUT pins to test 64 DUTs. This example is only valid on Magnum 2x.

```
#include "tester.h"

DUT_PIN( dp1 ){}
DUT_PIN( dp2 ){}
DUT_PIN( dp3 ){}
DUT_PIN( dp4 ){}

#define NUM_DUTS 64

PIN_ASSIGNMENTS( PA64_duts ){
   add( obj, dp1,
      NUM_DUTS,
      a_1, b_1, c_1, d_1, a_5, b_5, c_5, d_5,
      a_9, b_9, c_9, d_9, a_13, b_13, c_13, d_13,
      a_17, b_17, c_17, d_17, a_21, b_21, c_21, d_21,
      a_25, b_25, c_25, d_25, a_29, b_29, c_29, d_29,
      a_33, b_33, c_33, d_33, a_37, b_37, c_37, d_37,
      a_41, b_41, c_41, d_41, a_45, b_45, c_45, d_45,
      a_49, b_49, c_49, d_49, a_53, b_53, c_53, d_53,
      a_57, b_57, c_57, d_57, a_61, b_61, c_61, d_61
   );
   add( obj, dp2,
      NUM_DUTS,
      a_2, b_2, c_2, d_2, a_6, b_6, c_6, d_6,
      a_10, b_10, c_10, d_10, a_14, b_14, c_14, d_14,
      a_18, b_18, c_18, d_18, a_22, b_22, c_22, d_22,
      a_26, b_26, c_26, d_26, a_30, b_30, c_30, d_30,
      a_34, b_34, c_34, d_34, a_38, b_38, c_38, d_38,
      a_42, b_42, c_42, d_42, a_46, b_46, c_46, d_46,
      a_50, b_50, c_50, d_50, a_54, b_54, c_54, d_54,
      a_58, b_58, c_58, d_58, a_62, b_62, c_62, d_62
   );
   add( obj, dp3,
      NUM_DUTS,
      a_3, b_3, c_3, d_3, a_7, b_7, c_7, d_7,
      a_11, b_11, c_11, d_11, a_15, b_15, c_15, d_15,
      a_19, b_19, c_19, d_19, a_23, b_23, c_23, d_23,
      a_27, b_27, c_27, d_27, a_31, b_31, c_31, d_31,
      a_35, b_35, c_35, d_35, a_39, b_39, c_39, d_39,
```

```
      a_43, b_43, c_43, d_43, a_47, b_47, c_47, d_47,
      a_51, b_51, c_51, d_51, a_55, b_55, c_55, d_55,
      a_59, b_59, c_59, d_59, a_63, b_63, c_63, d_63
   );
   add( obj, dp4,
      NUM_DUTS,
      a_4, b_4, c_4, d_4, a_8, b_8, c_8, d_8,
      a_12, b_12, c_12, d_12, a_16, b_16, c_16, d_16,
      a_20, b_20, c_20, d_20, a_24, b_24, c_24, d_24,
      a_28, b_28, c_28, d_28, a_32, b_32, c_32, d_32,
      a_36, b_36, c_36, d_36, a_40, b_40, c_40, d_40,
      a_44, b_44, c_44, d_44, a_48, b_48, c_48, d_48,
      a_52, b_52, c_52, d_52, a_56, b_56, c_56, d_56,
      a_60, b_60, c_60, d_60, a_64, b_64, c_64, d_64
   );
}
```

3.8.3.2 Sites-per-Controller

See Pin Assignment Table.

Description

When the DUT to be tested requires more than 64 signal pins, the `SITES_PER_CONTROLLER()` Test System Macro is used, in the Pin Assignment Table, to specify how many test sites are required to test the DUT.

In general, when testing a DUT with 64 signal pins or less, the sites-per-controller value should be 1, for DUTs with 65 to 128 signal pins the value should be 2, etc.

Using Magnum 1/2 the Functional Pin-pairs architecture does not change the purpose of sites-per-controller, however the definition of a *site* is somewhat more complex as compared to Maverick-I/-II. A Magnum 1/2 site consists of 128 signal pins; 64 pins on sub-site A (a_1 through a_64) and 64 pins on sub-site B (b_1 through b_64). In most applications, it is not practical to use pins from both sub-site A and B to functionally test a given DUT. This is because both pins of a given pin-pair receive exactly the same pattern data and waveforms and it is a rare device type which can use this in real world testing. Thus, when specifying the sites-per-controller value, it is likely that two sites will always be required to test a DUT which needs more than 64 but less than 128 signal pins. However, in this situation, it will

also be possible to test 2 DUTs on each site; DUT-1 using pins a_1 through a_128 and DUT-2 using pins b_1 through b_128.

In situations where the DUT being tested has multiple package types with different pin requirements, for example 64 pins or less in one package type and 65 signal pins or more in another, the Pin Assignment Table for each package type will specify a different sites-per-controller value.

When sites-per-controller > 1, test pattern branch-on-error operations (MAR CJMPE, CJMPNE, etc.) require additional pipeline clock counts, see Error Pipeline Requirements.

The sites_per_controller() function may be used to determine the sites-per-controller value currently set in the test program.

Usage

```
PIN_ASSIGNMENTS( table_name ) { // See PIN_ASSIGNMENTS()
   SITES_PER_CONTROLLER( #_sites )
   // ... Pin Assignment Table macros as needed.
}
int sites_per_controller();
```

where:

SITES_PER_CONTROLLER is a Test System Macro used to specify the number of 64-pin sites required to test one DUT. When **SITES_PER_CONTROLLER** is not explicitly specified it defaults to 1.

#_sites is the integer number of 64-pin sites required to test one DUT. Using Magnum 1/2/2x, the legal values are 1 to 10. The value specified must be consistent with the number of site-assembly boards installed in any test system which will execute the test program.

sites_per_controller() returns the value set using SITES_PER_CONTROLLER.

Example

The following example shows the beginning of a Pin Assignment Table for a DUT which has between 65 and 128 signal pins:

```
DUT_PIN( dp1 ){}
DUT_PIN( dp2 ){}
PIN_ASSIGNMENTS( myPinAssignmentsTable ) {
   SITES_PER_CONTROLLER( 2 )
   //          DutPin    t_dut1    t_dut2     t_dut3    t_dut4
```

```
   ASSIGN_4DUT( dp1,     a_1,      b_1,      a_23,     b_23 )
   ASSIGN_4DUT( dp2,    a_96,     b_96,     a_111,    b_111 )
}
```

3.8.3.3 Shared Tester Pins

See Pin Assignment Table.

Note: this section and related information requires software release h1.0.32 or later.

Note: this section describes a specific application which is not commonly used, quite restricted in capabilities and requires the user to manage program details normally handled by the system software.

It is possible to connect a given tester pin (PE channel) to more than one DUT, allowing more DUTs to be tested in parallel than is possible without sharing pins. Normally, this application requires user-designed DUT board circuitry to connect/disconnect DUTs at selected times in the test flow.

The Magnum 1/2/2x software only support this capability in two places:

- Pin Assignment Table, allowing a given pin (PE channel) to be assigned to the same `DutPin` of more than one DUT. This is described and documented below. And, as indicated above, support for this capability is very specific and limited.
- Error Catch RAM Software, specifically the `ecr_dut_number_set()`, `ecr_dut_number_get()` functions. This section also describes how pin lists are affected, which affects ECR use.

Note: other Magnum 1/2/2x software does not comprehend or compensate for sharing tester pins between DUTs; i.e. the user is responsible for proper operation.

Rules:

1. The various `ASSIGN_nDUT` macros (see Pin Assignment Table) allow a given `HDTesterPin` (PE channel) to be mapped to a given `DutPin` more than once, but the usage rules are very specific. Examples are used below to explain the rules.

For example, the following program tests 16 DUTs in parallel with the first 8 DUTs sharing pins with the second 8 DUTs:

```
DUT_PIN(D0){}
DUT_PIN(D1){}
...
PIN_ASSIGNMENTS( pin_assign_16_duts ){
//              PIN  DUT1 DUT2 DUT3 ...snip...   DUT9 DUT10 Etc...
ASSIGN_16DUT(  D0,  a_1, b_1, a_17,...snip..., a_1,  b_1, Etc...
ASSIGN_16DUT(  D1,  a_2, b_2, a_18,...snip..., a_2, b_2, Etc...
...
}
```

Note the following:

- Each set of DUTs must mirror the first set. Using the previous example, two sets of DUTs are tested: the second set begins with DUT-9. The `HDTesterPin` associated with DUT-9 pin D0 must match DUT-1 pin D0, and the `HDTesterPin` associated with DUT-9 pin D1 must match DUT-1 pin D1. Etc.
- All DUT pins must be configured identically; i.e. all must share pins in a similar manner. Using the previous example, the following is illegal because it violates this rule :

```
ASSIGN_16DUT(  D1,  a_2, b_2, a_18,...snip..., a_50, b_50, Etc...
```

3.8.3.4 testerpin_name()

See Configuring the Tester to the DUT, Pin Assignment Table, DUT Pins.

Description

The `testerpin_name()` function is used to return the name of a specified tester pin.

Usage

```
CString testerpin_name( HDTesterPin pin );
```

where:

`pin` identifies the target tester pin. Legal values are of the `HDTesterPin` enumerated type.

`testerpin_name()` returns the pin name as a `CString`.

Example

The following example prints the name of each tester pin connected to each DUT defined in the Pin Assignment Table:

```
DutPin *dp = dp1;
HDTesterPin tpin;
for( DutNum dutnum = t_dut1;
   pin_info( dp, dutnum, &tpin );
   ++dutnum )
      output(" %s of t_dut%d connects to %s",
         resource_name( dp ),
         (dutnum + 1),
         testerpin_name( tpin) );
```

3.8.3.5 testerpin_value()

See Configuring the Tester to the DUT, Pin Assignment Table, DUT Pins.

Description

The `testerpin_value()` function converts a tester pin name specified as a `CString` into a `HDTesterPin`.

`testerpin_value()` is the complement of `testerpin_name()`.

Usage

```
BOOL testerpin_value( CString name, HDTesterPin *pin );
```

where:

`name` specifies the target pin name as a `CString`.

`pin` is a pointer to an existing `HDTesterPin` variable used to return the pin identified by **`name`**. If **`name`** is invalid **`pin`** will return NULL.

`testerpin_value()` returns TRUE if the specified pin **`name`** is valid, otherwise FALSE is returned.

Example

```
HDTesterPin htpin;
```

```
BOOL ok;
ok = testerpin_value( "a_257", &htpin );
ok = testerpin_value( "a_dps1a", &htpin );
ok = testerpin_value( "a_hv2", &htpin );
```

3.8.3.6 testerpin_offset()

See Configuring the Tester to the DUT, Pin Assignment Table, DUT Pins.

Note: first available in software release h1.1.23.

Description

Given a specified HDTesterPin, the testerpin_offset() function will return:

- The first pin (base pin) of the same type (signal pin, DPS pin, etc.).
- The zero-based numerical offset of the specified pin from its base pin.

If the specified target pin is a valid HDTesterPin the HDTesterPin pin value returned will be one of:

- a_1 (first signal pin from sub-site A)
- b_1 (first signal pin from sub-site B)
- a_dps1a (first DPS pin from sub-site A)
- b_dps1a (first DPS pin from sub-site B)
- a_hv1 (first HV pin from sub-site A)
- b_hv1 (first HV pin from sub-site B)
- a_na

For example:

- Given HDTesterPin a_49, the base pin is a_1 and the offset is 48.
- Given HDTesterPin b_hv4, the base pin is b_hv1 and the offset is 3.

Usage

```
BOOL testerpin_offset( HDTesterPin *pin,
                       DWORD *offset = 0);
```

where:

pin both specifies the target input HDTesterPin and returns the base pin. It must be a pointer to an existing HDTesterPin variable, previously initialized with the target input value which will be replaced by the base pin. See examples.

offset is a pointer to an exising DWORD variable used to return the **pin**'s offset value from its base pin.

testerpin_offset() returns TRUE if the value specified for pin is valid, otherwise FALSE is returned.

Example

The following example will return a pin value = a_1 and offset = 102:

```
DWORD offset;
HDTesterPin pin = a_103;
BOOL ok = testerpin_offset( &pin, &offset );
if( ! ok ) output("ERROR: invalid pin input value");
```

3.8.3.7 Pin Iteration

See Configuring the Tester to the DUT, Pin Assignment Table, Magnum 1, 2 & 2x Parallel Test, See DUT Pins.

This section discusses methods which can be used to reliably iterate over pins of a particular type.

Fundamental to this topic are the following built-in pin lists, defined by the system software. These can be used to reliably iterate pins:

- PinList* builtin_UsedPins; // All signal pins used in Pin Assignment Table
- PinList* builtin_UsedHVPins; // All HV pins used in Pin Assignment Table
- PinList* builtin_UsedDPS; // All DPS pins used in Pin Assignment Table

These pin lists are initialized by the system software during the initial program load, to reflect the pins actually used by the test program being loaded.

To iterate over all signal pins used in the current program:

```
HDTesterPin tpin;
for ( int i = 0; pin_info( builtin_UsedPins, i, &tpin ); ++i ) {
   // Use tpin
}
```

It is also possible to iterate by DutNum, for example to determine which pins are associated with a DUT:

```
DutPin *dpin;
for( int i = 0; pin_info( builtin_UsedPins, i, &dpin ); ++i ){
   HDTesterPin tpin;
   for( DutNum dut = t_dut1;
        pin_info( dpin, dut, &tpin );
        ++dut ) {
      // Use tpin and dpin
   }
}
```

It is also possible to iterate through specific pin ranges. However, any loops through an entire range of a particular type of pin should be treated with suspicion! It is generally better to loop over the built-in pin lists, so don't use the idioms in the following example unless you have a reason why the built-in pin lists don't work for your application:

```
for ( HDTesterPin tp = a_1; tp < a_pe_max; ++tp ) {
   // In software releases prior to h1.2.xx, tp will range from a_1
   // to a_512. Beginning in h1.2.xx, tp will range from a_1 to
   // a_2560
}
for ( HDTesterPin tp = b_1; tp < b_pe_max; ++tp ) {
   // In software releases prior to h1.2.xx, tp will range from a_1
   // to a_512. Beginning in h1.2.xx, tp will range from b_1 to
   // a_2560
}
```

3.8.4 Pin Lists

See Configuring the Tester to the DUT, Pin Assignment Table, Magnum 1, 2 & 2x Parallel Test, See DUT Pins.

Description

A pin list is a named list of DUT pins. Many Nextest functions take a pin list (`PinList*`) argument, to identify one or more pin(s) to be affected by the function; for example, to program a per-pin voltage, current, or timing value, etc.

Note the following:

- Pin lists can be defined statically (most common) or dynamically.
- The `PINLIST` Test System Macro is used to create a static pin list. The `PINLIST` macro can only be used at global scope (i.e. outside of any C-function, not nested, etc.), but otherwise can appear in any test program source file. For instance, it might improve program readability to define a pin list used only once just before the code that uses it, but to define commonly used pin lists all in one file. The convention implemented by the Test Program Wizards has pin lists stored in the *pin_lists.cpp* file.
- The `EXTERN_PINLIST` macro is provided to simplify making an external pin list declaration, conventionally in the *pin_lists.h* file, which is then included into all other source files, typically via the *tester.h* file.
- The `INCLUDE_PINLIST` macro can be used to add an existing pin list as an element of a new pin list definition.
- Dynamic pin lists are created, and destroyed, using `pinlist_create()`, `pinlist_destroy()`.
- In use, a pin list is formally represented as a `PinList*`.
- Several additional macros are used to add members to a pin list. See Usage.
- Pin list members are specified using `DutPin` names, previously defined using the `DUT_PIN` macro. These are the same names used in the Pin Assignment Table.
- When using Magnum 1/2/2x, most functions which take a `Pinlist*` argument also have an overload which will accept a `DutPin` argument.
- The order of pins in a pin list is typically not important. However when it is, the general rule is that the first element is processed first, etc.
- The `pin_info()` function can be used to iterate through the pins in a pin list.
- Given the name of a pin list (as a `char*`, `LPCTSTR` or `CString`) the `PinList_find()` function can be used to get a pointer to the pin list (`PinList*`).
- The system software predefines a number of built-in pin lists, all of which start with the prefix `builtin_`. This prefix is reserved for Nextest use.

```
builtin_UsedPins
builtin_UsedHVPins
builtin_UsedDPS
```

Using Magnum 1/2/2x Multi-DUT Test Programs, the underlying contents of a pin list is more complex than in non-Multi-DUT Test Program. The following example code will be used to explain this:

```
DUT_PIN( dp1 ){}
DUT_PIN( dp2 ){}
PIN_ASSIGNMENTS( myPinAssignmentsTable ) {
   SITES_PER_CONTROLLER( 1 ) // See Sites-per-Controller
   //          DutPin    t_dut1   t_dut2    t_dut3    t_dut4
   ASSIGN_4DUT( dp1,      a_1,       b_1,      a_23,      b_23 )
   ASSIGN_4DUT( dp2,     a_21,      b_21,       a_3,       b_3 )
}
PINLIST( plAllPins ){
   PINS2 ( dp1, dp2 )
}
```

Using this example, note the following:

- `plAllPins` actually represents 8 members, 2 pins for each DUT defined in the example Pin Assignment Table.
- When `plAllPins` is used to program (set) a value, the Active DUTs Set (ADS) determines which pin(s) are actually affected. If, for example, the Active DUTs Set (ADS) contains all DUTs except `t_dut2`, any Nextest function which programs a parameter using `plAllPins` will not affect the two pins mapped to DUT-2; i.e. pins `b_1` and `b_21` will not be affected. And, the other 6 pins mapped to DUTs 1, 3 and 4, will be affected.

Also see DUT-specific Pin Lists.

Usage

`PinList` is the resource (see Resources) created using the `PINLIST` macro.

The following Test System Macros are used to create and add pins to a `PinList`.

```
PINLIST( list_name ) {
   PINS1( p1 )
   PINS2( p1, p2 )
   PINS3( p1, p2, p3 )
   PINS4( p1, p2, p3, p4 )
```

```
   PINS5( p1, p2, p3, p4, p5 )
   PINS6( p1, p2, p3, p4, p5, p6 )
   PINS7( p1, p2, p3, p4, p5, p6, p7 )
   PINS8( p1, p2, p3, p4, p5, p6, p7, p8 )
   PINS( p1, p2, p3, p4, p5, p6, p7, p8 )
   INCLUDE_PINLIST( existing_list )
}
```

where:

`list_name` identifies the pin list being created. Must be a valid C identifier.

`PINS1` through **`PINS8`**, and **`PINS`** are Test System Macros used to add members to the pin list. The number of arguments to each macro must match the name of the macro; i.e. **`PINS4`** requires 4 arguments. **`PINS`** operates the same as **`PINS8`**.

`p1` through **`p8`** are DutPins. These are the same as used in the DUT Pin Name column in the Pin Assignment Table.

Examples

Example 1:

The following example creates a pin list called `data_bus`, containing 8 members (`d0,..,d7`):

```
PINLIST( data_bus ) {
   PINS8( d0, d1, d2, d3, d4, d5, d6, d7 )
}
```

Example 2:

The following example creates a pin list called `address_bus`, containing 16 members. Different macros are used here for variety:

```
PINLIST( address_bus ) {
   PINS5( a15, a14, a13, a12, a11, a10 )
   PINS4( a9, a8, a7, a6 )
   PINS6( a5, a4, a3, a2, a1, a0 )
}
```

Example 3:

In the following example, the pin lists from Example 1: and Example 2: are used as elements in a new pin list, called `all_pins`. This example demonstrates the use of the `INCLUDE_PINLIST` macro:

```
PINLIST( all_pins ) {
   INCLUDE_PINLIST( data_bus )
   INCLUDE_PINLIST( address_bus )
   PINS3( WE, CS, OE )
}
```

3.8.4.1 DUT-specific Pin Lists

See Pin Lists.

Note: first available in software release h1.1.26 and h2.0.11.

Description

In Multi-DUT Test Programs, when defining Pin Lists, by default, a given `DutPin` added to the pin list actually represents the same pin of all DUTs in the test program. This is transparent to the user and, in most applications, is desirable: these pin lists make it easy to, for example, set identical test parameter values for a set of pins for all DUTs in the test program (usually all DUTs in the Active DUTs Set (ADS)). And, when the pins of a sub-set of DUTs must be manipulated/tested the Active DUTs Set (ADS) can be modified to control which DUT(s) will be affected.

It is also possible to create a pin list which contains pin elements for specific DUT(s), excluding pins of some DUT(s). These can be useful when Controlling PE Levels from the Test Pattern.

When Controlling PE Levels from the Test Pattern, when a conventional pin list is used in the LSENABLE pattern instruction, the level being modified will change on the same pin(s) of all DUT(s) in the Active DUTs Set (ADS) (this is usually desirable). But, since it is not possible to modify the ADS during the execution of a test pattern, to modify a PE level for pins of some active DUT(s) (maybe only one) while the other active DUT(s) remain unaffected requires using a pin list which contains only pins of specific DUTs. The following macros may be used to create these pin lists.

Usage

The following Test System Macros are used to add pin(s) for specific DUT(s) to a `PinList`. These are only usable within the body of the `PINLIST()` macro (see example below):

```
PINS_OF_1DUT(  dp, d1 )
PINS_OF_2DUT(  dp, d1, d2 )
PINS_OF_3DUT(  dp, d1, d2, d3 )
... snip ...
PINS_OF_32DUT( dp, d1, d2, d3, ... snip ..., d32 )
```

where:

dp identifies the `DutPin` to be added to the pin list.

d1 through **d32** identifies which DUT(s) will have their pins included in the pin list. Legal values are of the `DutNum` enumerated type. The number of `DutNum` values which must be specified is determined by the macro name.

Example

The following example creates 8 pin lists, each with a unique name and each identifying one pin (VDD) from one DUT:

```
PINLIST( pl_vcc_DUT1 ) { PINS_OF_1DUT( VDD, t_dut1) } // PINLIST()
PINLIST( pl_vcc_DUT2 ) { PINS_OF_1DUT( VDD, t_dut2) }
PINLIST( pl_vcc_DUT3 ) { PINS_OF_1DUT( VDD, t_dut3) }
PINLIST( pl_vcc_DUT4 ) { PINS_OF_1DUT( VDD, t_dut4) }
PINLIST( pl_vcc_DUT5 ) { PINS_OF_1DUT( VDD, t_dut5) }
PINLIST( pl_vcc_DUT6 ) { PINS_OF_1DUT( VDD, t_dut6) }
PINLIST( pl_vcc_DUT7 ) { PINS_OF_1DUT( VDD, t_dut7) }
PINLIST( pl_vcc_DUT8 ) { PINS_OF_1DUT( VDD, t_dut8) }
```

3.8.4.2 pinlist_create(), pinlist_destroy()

See Pin Lists.

Description

The `pinlist_create()` function is used, during program execution, to dynamically create a pin list and define its `PinList*`. The `pinlist_destroy()` function is used to destroy a pin list, freeing any associated memory, and undefine its `PinList*`.

Usage

The following function creates a pin list containing one DutPin:

```
PinList* pinlist_create( DutPin *dutpin );
```

The following function creates a pin list containing one or more DutPin(s):

```
PinList* pinlist_create( DutPin **pins,
                         int size,
                         LPCTSTR name DEFAULT_VALUE( 0 ) );
void pinlist_destroy( PinList* &obj );
```

where:

dutpin identifies one pin to be in the pin list. When this pin list is accessed, the corresponding pin of all DUT(s) in the Active DUTs Set (ADS) are accessed.

pins is a pointer to an existing DutPin* or array of DutPin* values which define the members of the pin list being created. When this pin list is accessed, the corresponding pin(s) of all DUT(s) in the Active DUTs Set (ADS) are accessed.

size is used when **pins** repesents an array of DutPins and specifies the number of DutPin* values to add to the pin list from the **pins** argument.

name is optional, and is used to specify a name for the PinList* being defined.

obj is a pointer to the PinList to be destroyed.

pinlist_create() returns a pointer to the PinList created (PinList*).

Examples

```
DutPin* myPins[]={ D0, D1 };
PinList* pl_myPins = pinlist_create( myPins, 2, "myPins");
// Use pl_myPins as desired
pinlist_destroy( pl_myPins );
```

3.8.4.3 pin_info()

See Pin Lists.

Description

The `pin_info()` function has several purposes:

- To identify the `DutPin` at a specified position in a `PinList`:
- In Multi-DUT Test Program, to identify the tester pin connected to a specific pin of a specified DUT.
- In Multi-DUT Test Program, to identify the `HDTesterPin` associated with each pin in a pin list, for each DUT in the test program.

`pin_info()` is often used within a loop, to iterate over each member pin of the specified `PinList`. This usage can be seen in the various datalogging applications found in most test programs. The example below shows this usage.

Usage

The following function is used to obtain the `HDTesterPin` connected to a specified `DutPin` for a specified DUT. This is used in Multi-DUT Test Programs, where a given `DutPin` actually represents an `HDTesterPin` for each DUT in the program:

```
BOOL pin_info( DutPin *obj, DutNum dut, HDTesterPin *pin );
```

The following function is used to obtain a pointer to the `DutPin` at the indexth position of the specified `PinList`:

```
BOOL pin_info( PinList* obj, int dutpin_index, DutPin **dutpin );
```

The following function is used to obtain the `HDTesterPin` at the `pin_index`th position in the specified pin list. In Multi-DUT Test Programs, the Active DUTs Set (ADS) is ignored. The `HDTesterPin` at `pin_index` = 0 is for `t_dut1`, the `HDTesterPin` at `pin_index` = 1 is for `t_dut2`, etc.

```
BOOL pin_info( PinList* obj, int pin_index, HDTesterPin *p_pin );
```

where:

`obj` is used in two contexts:

- As a `DutPin*`, **`obj`** identifies the target `DutPin`.
- As a `PinList*`, **`obj`** identifies the `PinList` containing the **`dutpin`** of interest.

`dutpin_index` is the zero based position of the **`dutpin`** of interest in the specified `PinList`.

`dutpin` is a pointer to a `DutPin` pointer variable used to return the address of the `DutPin` at the index'th position in the specified `PinList`.

dut identifies the DutNum for which the information is to be returned. This is used in Multi-DUT Test Programs, where a given DutPin actually represents a HDTesterPin for each DUT in the program.

pin_index is the zero based position of the pin of interest in the specified PinList.

p_pin is a pointer to an existing variable of type HDTesterPin. It is used to return the tester pin of the pin at the pin_index'th position in the specified PinList.

The first version of pin_info() above returns TRUE if the specified DUT is currently in the Active DUTs Set (ADS), otherwise FALSE is returned.

The second and third versions of pin_info() above return TRUE when the index'th position of the specified PinList contains a valid pin, otherwise FALSE is returned. This is useful when using pin_info() as the control statement in a C for loop (as seen the Example below).

Example

The following example iterates over all members of the pin list named allpins and prints the name of each member:

```
DutPin *dutpin;
for( int i = 0;pin_info( allpins, i, &dutpin ); ++i )
   output(" Pinlist member-%d => %s", p, resource_name( dutpin ) );
```

The following example iterates over all DUT(s) defined in the Pin Assignment Table and, for each DUT, returns the HDTesterPin connected to that DUT's WE_bar pin:

```
DutPin *dp = WE_bar;
HDTesterPin tpin;
for( DutNum dutnum = t_dut1;
     pin_info( dp, dutnum, &tpin );
     ++dutnum )
        output(" %s of t_dut%d connects to %s",
             resource_name( dp ),
             (dutnum + 1),
             testerpin_name( tpin) );
```

3.8.4.4 all_dps()

See Pin Lists.

Description

The `all_dps()` function can be used to confirm that all members of a specified `PinList` are DUT Power Supply (DPS) pins (i.e. not digital test pins or High Voltage Source/Measure Unit (HV) pins).

See also `no_dps()`, `all_pe()`, `no_pe()`, `all_hv()`, `no_hv()`.

Usage

```
BOOL all_dps( PinList* obj );
```

where:

`obj` identifies the `PinList` of interest.

`all_dps()` returns TRUE if all pins in **`obj`** are DPS pins, otherwise FALSE is returned. Beginning in software release h2.2.xx/h1.2.xx `all_dps()` returns FALSE if the **`obj`** argument is NULL or contains no pins.

Example

```
if ( all_dps ( pl_some_pins ) )
   dps ( 3.3 V, pl_some_pins );
else
   output(" ERROR: invalid PinList for use with dps() function");
```

3.8.4.5 no_dps()

See Pin Lists.

Description

The `no_dps()` function can be used to determine whether any members of a specified `PinList` are DUT Power Supply (DPS) pins.

See also `all_dps()`, `all_pe()`, `no_pe()`, `all_hv()`, `no_hv()`.

Usage

```
BOOL no_dps( PinList* obj );
```

where:

obj identifies the PinList of interest.

no_dps() returns TRUE if none of the pins in **obj** are DPS pins, otherwise FALSE is returned. Beginning in software release h2.2.xx/h1.2.xx no_dps() returns FALSE if the **obj** argument is NULL or contains no pins.

Example

```
if ( no_dps ( pl_some_pins ) )
   vol ( 1 V, pl_some_pins );
else
   output(" ERROR: invalid PinList using vol() function");
```

3.8.4.6 all_hv()

See Pin Lists.

Description

The all_hv() function can be used to confirm that all members of a specified PinList are High Voltage Source/Measure Unit (HV) pins (i.e. not digital test pins or DUT Power Supply (DPS) pins).

See also all_dps(), no_dps(), all_pe(), no_pe(), no_hv().

Usage

```
BOOL all_hv( PinList* *obj );
```

where:

obj identifies the PinList of interest.

all_hv() returns TRUE if all pins in **obj** are HV pins, otherwise FALSE is returned. Beginning in software release h2.2.xx/h1.2.xx all_hv() returns FALSE if the **obj** argument is NULL or contains no pins.

Example

```
if ( all_hv ( pl_some_pins ) )
   hv_voltage_set( 3.3 V, pl_some_pins );
else
   output(" ERROR: invalid PinList for use with hv_voltage_set()");
```

3.8.4.7 no_hv()

See Pin Lists.

Description

The `no_hv()` function can be used to determine whether any members of a specified `PinList` are High Voltage Source/Measure Unit (HV) pins.

See also `all_dps()`, `no_dps()`, `all_pe()`, `no_pe()`, `all_hv()`.

Usage

```
BOOL no_hv( PinList* obj );
```

where:

`obj` identifies the `PinList` of interest.

`no_hv()` returns TRUE if none of the pins in **`obj`** are HV pins, otherwise FALSE is returned. Beginning in software release h2.2.xx/h1.2.xx `no_hv()` returns FALSE if the **`obj`** argument is NULL or contains no pins.

Example

```
if ( no_hv( pl_some_pins ) )
   vol ( 1 V, pl_some_pins );
else
   output(" ERROR: invalid PinList using vol() function");
```

3.8.4.8 all_pe()

See Pin Lists.

Note: first available in software release h2.2.xx/h1.2.xx.

Description

The `all_pe()` function can be used to confirm that all members of a specified `PinList` are digital test pins; i.e. not DUT Power Supply (DPS) or High Voltage Source/Measure Unit (HV) pins).

See also `all_dps()`, `no_dps()`, `no_pe()`, `all_hv()`, `no_hv()`.

Usage

```
BOOL all_pe( PinList* obj );
```

where:

`obj` identifies the `PinList` of interest.

`all_pe()` returns TRUE if all pins in **`obj`** are digital test pins pins, otherwise FALSE is returned. `all_pe()` returns FALSE if the **`obj`** argument is NULL or contains no pins.

Example

```
if ( all_pe ( pl_some_pins ) )
   vol( 800 MV );
else
   output(" ERROR: invalid PinList using vol() function");
```

3.8.4.9 no_pe()

See Pin Lists.

Note: first available in software release h2.2.xx/h1.2.xx.

Description

The `no_pe()` function can be used to determine whether a specified `PinList` contains any digital test pins.

See also `all_dps()`, `no_dps()`, `all_pe()`, `all_hv()`, `no_hv()`.

Usage

```
BOOL no_pe( PinList* obj );
```

where:

obj identifies the PinList of interest.

no_pe( ) returns TRUE if none of the pins in **obj** are digital test pins, otherwise FALSE is returned. no_pe( ) returns FALSE if the **obj** argument is NULL or contains no pins.

Example

```
if ( no_pe ( pl_some_pins ) )
   output("pl_some_pins contains no digital test pins");
else
   output("pl_some_pins contains at least 1 digital test pin");
```

3.9 Program Execution Control

See Software.

- Overview
- Execution Context Functions
- Configuration Macros
- Host Begin Block, Host End Block
- Site Begin Block, Site End Block
- Tool Begin Block, Tool End Block
- Initialization Hook
- Sequence & Binning Table
- Parking Blocks
- Test Flow Synchronization
- Test Blocks
- `Delay()`
- Error Line Reset from CPU: reset_error()
- Control of Branch on Error Flag
- Over-programming Control Stimulus Selection

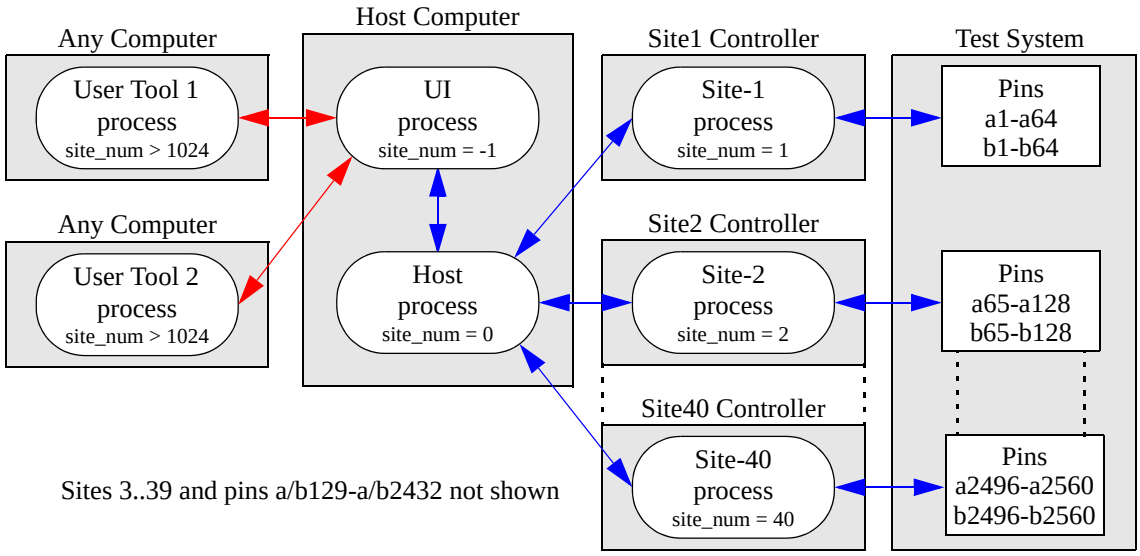
3.9.1 Overview

See Program Execution Control.

Executing a Magnum 1/2/2x test program invokes the following processes:

- UI - User Interface. Used to select, load, and unload a test program. It is UI which invokes the Host process and Site process(es). Communicates with Host, Site, and User Tools. The UI process `site_num()` = -1.
- Host process. Executes selected test program code on the Host computer (see `CONFIGURATION()`, `HOST_CONFIGURATION()`, `HOST_BEGIN_BLOCK()`, `HOST_END_BLOCK()`, `INITIALIZATION_HOOK()`). Can communicate with UI, Sites and User Tools. The Host process `site_num()` = 0.

- Site process(es). Executes selected test program code on the Site computer(s). (see `CONFIGURATION()`, `SITE_CONFIGURATION()`, `SITE_BEGIN_BLOCK()`, `SITE_END_BLOCK()`, `INITIALIZATION_HOOK()`). The Sequence & Binning Table, Test Blocks, and all functions which access tester hardware execute in the Site process. Can communicate with UI, Host, and User Tools. The Site process `site_num()` = 1 through 40.
- Tool process(es). Optional. User Tools execute independent of the Host/Site processes, but can communicate with each of them, and each other, though UI. Tool processes have `site_num()` > 1024:



The diagram above shows the various process communication paths. The blue paths are established when a test program is loaded - these are the standard program communications paths. The red paths are established when User Tools are started.

user-written code in each process can communicate with the other processes using various functions documented in Host / Site / Tool Communication. The `site_num()` values noted above play a key role in this communications.

When a test program is loaded, the Host process and each Site process load the same test program executable file. The program executes different code on Host vs. Site process as determined by command line arguments used when the program is loaded. These command line arguments are normally invisible to the user, and managed by UI - User Interface. The exception is when Host/Site/Tool Debug Mode(s).

As noted above, different test program code executes in Host vs. Site process(es). The table below indicates some of the differences in Host vs. Site processes. Note that there is no synchronization between Host or Site or Tool, as might be implied by the table:

Table 3.9.1.0-1 Host vs. Site vs. Tool Execution

Host	Site(s)	Tool
Communicates directly with UI and Sites. Communicates with User Tools via UI.	Communicates with UI via Host process. Communicates with User Tools via UI.	Communicates directly with UI. Communicates with Host and Sites via UI.
Execute CONFIGURATION(), and HOST_CONFIGURATION()	Wait for Host to signal *ready*. Then Initialize tester hardware defaults	Execute CONFIGURATION(), and TOOL_CONFIGURATION() to configure tool options.
Execute PIN_ASSIGNMENTS() to get Sites-per-Controller value	Execute CONFIGURATION(), and SITE_CONFIGURATION() to set up Site options.	Execute TOOL_BEGIN_BLOCK() to configure tool options.
Execute HOST_BEGIN_BLOCK() (in a separate thread) to invoke and interact with user dialogs and interact with prober/ handler software.	Executing PIN_ASSIGNMENTS(), CURRENT_SHARE(), PINLIST(), VIHH_MAP(), PIN_SCRAMBLE()	Execute INITIALIZATION_HOOK()
The Host does **NOT** execute the Sequence & Binning Table.	Execute SITE_BEGIN_BLOCK() to set up hardware per test program configuration.	Communicate with UI, Host, and/or Site using Host / Site / Tool Communication methods.
Execute INITIALIZATION_HOOK()	Execute INITIALIZATION_HOOK()	Execute TOOL_END_BLOCK() when terminating the tool.
Execute HOST_END_BLOCK() when Closing the program	For each Start Testing signal, execute the Sequence & Binning Table. Report test results to UI.	
	Execute SITE_END_BLOCK() when Closing the program	

3.9.2 Execution Context Functions

See Program Execution Control.

Description

The `OnHost()` function may be used to determine if the code being executed is executing in the host process.

The `OnSite()` function may be used to determine if the code being executed is executing in a site process.

The `OnTool()` function may be used to determine if the code being executed is executing in the tool process.

The site_num() function may be used to determine the site number where the function is executed.

These functions can be used to determine the current execution context (process) from user-written C-code. See Overview.

By design, execution context is mostly transparent in a Magnum 1/2/2x test program. However, there are several situations where user-written code explicitly references execution context. This is in the form of a *site number*. In these cases the functions documented here can be useful:

- Using the `CONFIGURATION()` macro, the macro body code will execute in Host, Site, and User Tools processes, thus conditional statements are sometimes used to differentiate code which is to execute in each context. However, there is a better way (older programs didn't have this). Rather than use `CONFIGURATION()` plus conditional code it is recommended that the Host/Site/Tool specific configuration macros be used instead; i.e. use `HOST_CONFIGURATION()`, `SITE_CONFIGURATION()`, `TOOL_CONFIGURATION()`.
- User Dialogs executed in the Host process often need to communicate with Site process(es).
- User Dialogs executed in a User Tools process often need to communicate with Host and/or Site processes (sometimes other tools).
- Using `ui_OutputFile` or `ui_OutputFormat`.

- User Tools execute in processes separate from the test program. Code in user tools often communicates with Host and/or Site processes. Similarly, if the test program contains C-code which is to communicate with a user tool, that code must be able to obtain the site number of the tool (as an ID), which can be different each time the tool is started. However, this is not good design and is discouraged.

Usage

```
BOOL OnHost();
BOOL OnSite();
BOOL OnTool();
```

These functions each return TRUE when they are executed in the context after which they are named. For example, `OnHost()` returns TRUE when executed in the Host process, etc.

```
int site_num();
```

The `site_num()` function is more general, and returns one of the following:

- -1 = UI execution context
- 0 = Host process execution context
- 1 through 40 = Site process execution context
- > 1024 = User Tools execution context

Example

```
if ( OnHost() ) {
   // Host process code here
}
else if ( OnSite() ) {
   // Site process code here. Will execute on all sites
}
if (site_num() == 0) {
   // Host process code here
}
else if ( (site_num() >0) && (site_num() < 1024) ){
   // Site process code. Will execute on all sites
}
else if ( (site_num() == 3) ){
   // Site process code. Will execute on site #3 only
}
```

```
else if (site_num() > 1024) {
   // User Tools process code
}
```

Including the last `else` statement is misleading... code implementing User Tools is almost always separate from test program code, thus this example would never be used.

3.9.3 Configuration Macros

See Program Execution Control.

Description

The macros documented below are all Test System Macros.

An important part of writing a test program is creating the various tables that configure the tester software and hardware. Most of these tables are more generically called Single Resource Types. For example:

- Pin Assignment Table
- Pin Scramble Map
- VIHH Maps
- Sequence & Binning Table
- ... etc. ... See Single Resource Types for a complete list.

To enable reuse, and increase program versatility, the system software supports the creation of multiple instances of each type. However, as the program loads, only one instance of each Single Resource Types can be active.

As a convenience, if only a single definition of a given Single Resource Types exists (for example, one Sequence & Binning Table), then that table will be automatically selected. The configuration macros may also be used in the case of a single instance to make the test program more explicit.

When multiple instances of a Single Resource Types have been defined one instance must be selected at runtime. The macros documented here can be used to make this selection, or the system software will present a dialog prompting the user to select from a list. Note, however, that this dialog is presented for each Site, which is not user friendly. Thus the configuration macros are a better solution.

The Test Program Wizard provided with Visual C++ uses the CONFIGURATION macro in an example. It can be found in the *site_begin.cpp* file.

One drawback to hard coding the configuration using the configuration macros is that making a selection at runtime can require additional user-written code. See Single Resource Types for an alternate method for selecting the configuration at runtime.

The Configuration_find() function can be used to get a pointer to a Configuration Block. The HostConfiguration_find() function can be used to get a pointer to a Host Begin Block. The SiteConfiguration_find() function can be used to get a pointer to a Site Begin Block. The ToolConfiguration_find() function can be used to get a pointer to a Tool Begin Block.

Usage

The following macro executes in each of the Host, Site, and Tool process(es):

```
CONFIGURATION(){ body-code }
```

The following macro executes only in the Host process:

```
HOST_CONFIGURATION(){ body-code }
```

The following macro executes only in site processes:

```
SITE_CONFIGURATION(){ body-code }
```

The following macro executes only in User Tools processes:

```
TOOL_CONFIGURATION(){ body-code }
```

Each entry in the CONFIGURATION blocks is optional, see above. This example only shows the CONFIGURATION() macro, however, usage is similar for each of HOST_CONFIGURATION, SITE_CONFIGURATION, and TOOL_CONFIGURATION.

```
CONFIGURATION( name ) {
   USE_PIN_ASSIGNMENTS( name )    // See USE_PIN_ASSIGNMENTS()
   USE_PIN_SCRAMBLE( name )       // See USE_PIN_SCRAMBLE()
   USE_VIHH_MAP( name )           // See USE_VIHH_MAP()
   USE_SEQUENCE_TABLE( name )     // See USE_SEQUENCE_TABLE()
   USE_CURRENT_SHARE( name )      // See USE_CURRENT_SHARE()
}
```

where **name** is the name of the configuration that you want to use.

Example

```
CONFIGURATION( plcc ) {
   USE_PIN_ASSIGNMENTS( plcc )
   USE_PIN_SCRAMBLE( plcc )
   USE_VIHH_MAP( special_vihh_modes )
   USE_SEQUENCE_TABLE( wafer_sort )
}
```

3.9.3.1 Single Resource Types

See Configuration Macros, Program Execution Control.

Certain test program structures are only allowed to have a single active instance. For example, a test program may *define* multiple Sequence & Binning Tables, but at runtime only one table can be active at any given time.

These are structures called **`Single Resource`** data types. See Resource Types.

The table below shows key information about each Single Resource type:

Table 3.9.3.1-1 Single Resource Macros and Functions

Creation Macro	Run-time Selection Function	Load-time Selection Macro	Resource Name
CONFIGURATION()	Configuration_use	USE_CONFIGURATION	S_Configuration
CURRENT_SHARE()	CurrentShare_use	USE_CURRENT_SHARE	S_CurrentShare
HOST_BEGIN_BLOCK()	HostBeginBlock_use	USE_HOST_BEGIN_BLOCK	S_HostBeginBlock
HOST_END_BLOCK()	HostEndBlock_use	USE_HOST_END_BLOCK	S_HostEndBlock
PIN_ASSIGNMENTS()	PinAssignments_use	USE_PIN_ASSIGNMENTS()	S_PinAssignments
PIN_SCRAMBLE()	PinScramble_use	USE_PIN_SCRAMBLE()	S_PinScramble
SEQUENCE_TABLE()	SequenceTable_use	USE_SEQUENCE_TABLE()	S_SequenceTable
SITE_BEGIN_BLOCK()	SiteBeginBlock_use	USE_SITE_BEGIN_BLOCK	S_SiteBeginBlock
SITE_END_BLOCK()	SiteEndBlock_use	USE_SITE_END_BLOCK	S_SiteEndBlock
TOOL_BEGIN_BLOCK()	ToolBeginBlock_use	USE_TOOL_BEGIN_BLOCK	S_ToolBegin
TOOL_END_BLOCK()	ToolEndBlock_use	USE_TOOL_END_BLOCK	S_ToolEnd
VIHH_MAP()	VihhMap_use	USE_VIHH_MAP()	S_VihhMap

3.9.3.2 Single Resource Runtime Selection

See Configuration Macros, Program Execution Control.

The runtime software requires that only one instance of each Single Resource Types be active. This is resolved during the program load sequence.

In many cases, the test program may not define even one instance of a given resource. In this scenario, the system software sets up a default, and the test program may not contain a `CONFIGURATION` block.

In many cases, the test program only defines one instance of a given Single Resource Types. In this scenario, the test program may not contain a `CONFIGURATION` block; by design, the runtime software uses the one instance.

In the case where multiple instances of one, or more, Single Resource types is defined one of the following scenarios occurs:

- User C-code manages the selection, using one of the `CONFIGURATION()` block types.
- The runtime software presents the user with a selection dialog. Note: the user will be prompted once by each site controller, for each `Single Resource` which has multiple instances defined. This can be very annoying.
- User C-code can sometimes switch between or select the desired instance using `resource_select()`.

The example below demonstrates how user-written C-code, in the host process, can acquire a `Single Resource` selection from the user and pass it to the site(s). The selection is stored in a user variable (type `CSTRING_VARIABLE`) that the site(s) can subsequently access using the `remote_get()` function calls. This method has the advantage that:

1. No explicit synchronization is needed between host and site(s)
2. All activity takes place in the `CONFIGURATION` block.

Example

```
// If there is more than one SEQUENCE_TABLE, select one at runtime.
// Ditto for all Single Resource types. This example doesn't
// consider the content of the resource, it only shows the
// mechanism for allowing the host to control a selection at
// runtime.

#include "TestProgApp/public.h"
```

```
// These represent existing tables defined in various source files

SEQUENCE_TABLE( seq1 ) { ... }
SEQUENCE_TABLE( seq2 ) { ... }
SEQUENCE_TABLE( seq3 ) { ... }

PIN_ASSIGNMENTS( pa1 ) { ... }
PIN_ASSIGNMENTS( pa2 ) { ... }

SITE_BEGIN_BLOCK( sbb1 ) { ... }
SITE_BEGIN_BLOCK( sbb2 ) { ... }

// Host code will set these, and each site will remote_get() them.

CSTRING_VARIABLE( selected_sequence_table, "", "" ) { }
CSTRING_VARIABLE( selected_pin_assignments, "", "" ) { }
CSTRING_VARIABLE( selected_site_begin_block, "", "" ) { }

// The CONFIGURATION macro executes first on the host, since the
// host is started before the sites. In this case the host saves
// the names of the selected SEQUENCE_TABLE, PIN_ASSIGNMENTS, and
// SITE_BEGIN_BLOCK so that each site can retrieve it as needed.
// The CONFIGURATION setup on the site(s) occurs only after the
// host's CONFIGURATION has completed. That means that the values
// that the host saved will always be valid by the time the site(s)
// request them.

CONFIGURATION( example ) {
   if ( OnHost() ) {
      // The function resource_select() displays a simple dialog
      // within UI. In a real application, a user-defined dialog
      // that sets the following three user-variables would be
      // invoked instead.
   selected_sequence_table = resource_select( S_SequenceTable );
   selected_pin_assignments = resource_select( S_PinAssignments );
   selected_site_begin_block = resource_select( S_SiteBeginBlock);
      // The host needs to know which PIN_ASSIGNMENTS table to use,
      // since it needs the SITES_PER_CONTROLLER value (which may
      // vary between PIN_ASSIGNMENTS TABLES).
      PinAssignments_use( selected_pin_assignments );
   }
   if ( OnSite() ) {
```

```
        // Get the names of the selected SEQUENCE_TABLE,
        // PIN_ASSIGNMENTS, and SITE_BEGIN_BLOCK from the host, and
        // select those to be used.
        // If there is, for instance, more than on SEQUENCE_TABLE and
        // we don't tell the system which one to use, each site will
        // end up querying the operator to select a SEQUENCE_TABLE.
    SequenceTable_use( remote_get( selected_sequence_table, 0 ) );
    PinAssignments_use( remote_get( selected_pin_assignments, 0 ));
    SiteBeginBlock_use( remote_get( selected_site_begin_block, 0));
    }
}
```

3.9.4 Host Begin Block

See Host End Block, Site Begin Block, Program Execution Control.

Description

Several blocks of user-written code are automatically executed by the test system software when a test program is loaded:

- Host Begin Block, executes on the host computer and typically controls user input from dialog boxes and drives handlers, probers, and networks.
- Site Begin Block, executes locally on each test site controller and is used for configuring the hardware on each test site.
- Also see Configuration Macros, used to define `CONFIGURATION` blocks which can execute on both Sites and Host.

All configurations of the Magnum 1/2/2x use a host computer which is separate from each of the test site controller(s). They communicate using a TCP/IP connection.

The Host Begin Block is defined using the `HOST_BEGIN_BLOCK` macro. It is automatically called once by the system software; see Program Loading and Execution Order.

The following tasks are appropriately done in the C-code within the `HOST_BEGIN_BLOCK`:

- User dialogs.
- Initialization of handlers, probers, or external instruments
- Any initialization of user C variables, global binning information, or other data structures

See also Host End Block.

The `HostBeginBlock_find()` function can be used to get a pointer to a Host Begin Block.

Usage

```
HOST_BEGIN_BLOCK( name ) {
   // Initializations and C code go here
}
```

where:

`HOST_BEGIN_BLOCK` is a Test System Macro used to denote the beginning of the Host Begin Block.

`name` is a user-defined name.

Example

```
HOST_BEGIN_BLOCK( 64M_flash ) {
   invoke( OPERATOR_DIALOG );
}
```

3.9.4.1 Host Waiting for Site to Load

See Host Begin Block, Program Execution Control.

There are situations in which some Host user code must not execute until all of the Sites have completed loading the test program. The example code below can be used in the Host Begin Block to wait for the Sites to complete loading the test program :

```
// File: host_begin.cpp

// UI invokes when ui_ProgLoaded body code when all site have
// completed initial program load
VOID_VARIABLE( ui_ProgLoaded, "" ) {
   // Send signal to Host process when all sites are done loading
   remote_signal( "AllSitesLoaded", site_num() );
}
```

```
HOST_BEGIN_BLOCK( my_host_block_name ) {
   ... other code here ...
   // Wait here for signal...
   remote_wait( "AllSitesLoaded", INFINITE );
   ... other code here ...
}
```

3.9.5 Host End Block

See Host Begin Block, Site End Block, Program Execution Control.

Description

Similar in purpose to the Host Begin Block, the Host End Block may optionally be defined by in user-written C-code. If defined, the code will execute as part of the program unload process. See also Site End Block.

The `HostEndBlock_find()` function can be used to get a pointer to a Host Begin Block.

Note: user C-code must ***NOT*** call `HOST_END_BLOCK`.

Usage

```
HOST_END_BLOCK( block_name ) {
   // C code goes here
}
```

where:

`HOST_END_BLOCK` is a Test System Macro used to denote the beginning of the Host End Block.

`name` is a user-defined name.

3.9.6 Site Begin Block

See Site End Block, Host Begin Block, Program Execution Control.

Description

The purpose of the SITE_BEGIN_BLOCK is to configure the test site hardware in an appropriate manner for your needs if that configuration is different than the default hardware state set by the system software at program load time. The tester hardware and much of the system software is initialized to default states before the SITE_BEGIN_BLOCK is called. Aside from a few simple mandatory actions, user code in the SITE_BEGIN_BLOCK is to override the system-defined defaults for various tester resources.

The SITE_BEGIN_BLOCK will automatically be called once by the system software; you must not call the SITE_BEGIN_BLOCK by yourself. The SITE_BEGIN_BLOCK typically runs in conjunction with the Host Begin Block to process dialog box information input by users. See Program Loading and Execution Order.

The following tasks are appropriately done in the SITE_BEGIN_BLOCK to configure the test site hardware for the application:

1. Configuration of the algorithmic pattern generator
2. Changing other hardware parameters from their default values
3. Any initialization of your own variables, binning information, or other data structures

The system software initializes much of the tester hardware before calling the SITE_BEGIN_BLOCK. The hardware initialization states are identified for all functions and macros in this manual under the sub-heading of **Default State**. The system software writes all tester hardware to the default state and then calls the SITE_BEGIN_BLOCK, allowing user code to modify the hardware state before testing begins. Thus, any hardware with a default state that is not initialized or set by the user will be in its default state for the duration of the test program.

Note: user C-code must ***NOT*** call SITE_BEGIN_BLOCK.

See also Site End Block.

Usage

```
SITE_BEGIN_BLOCK( name ) {
   /* Initializations and C code go here. */
}
```

where:

SITE_BEGIN_BLOCK is a Test System Macro used to denote the beginning of the Site Begin Block.

name is a user-defined name.

Example

```
SITE_BEGIN_BLOCK( boot_block ) {
     numx( 11 );  // Enable X0 - X10
     numy( 7 );   // Enable Y0 - Y6
     x_fast_axis( TRUE );   // X is the fast address
     data_reg_width( 36 );  // Data register is 36 bits
     initialize_my_variables();
}
```

3.9.7 Site End Block

See Site Begin Block, Host End Block, Program Execution Control.

Description

Similar in purpose to the Site Begin Block, the Site End Block may optionally be defined by the user. If defined, the Site End Block will be executed prior to the sites unloading the test program. See also Host End Block.

Note: user C-code must ***NOT*** call `SITE_END_BLOCK`.

Usage

```
SITE_END_BLOCK( block_name ) {
   // C code goes here
}
```

where:

SITE_END_BLOCK is a Test System Macro used to denote the beginning of the Site End Block.

name is a user-defined name.

3.9.8 Tool Begin Block

See Site Begin Block, Host Begin Block, Program Execution Control.

Description

User-created tools optionally have both begin and end blocks, similar in purpose to Host Begin Block and Site Begin Block. In general, if a `TOOL_BEGIN_BLOCK` is defined in the tool code it will be executed when the tool is started. The `TOOL_BEGIN_BLOCK` only executes in the process of the tool; i.e. not in the Host process or Site process(es).

Note: user C-code must ***NOT*** call `TOOL_BEGIN_BLOCK`.

Usage

```
TOOL_BEGIN_BLOCK( name ) {
   // Initializations and C code go here
}
```

where:

`TOOL_BEGIN_BLOCK` is a Test System Macro used to denote the beginning of the Tool Begin Block.

`name` is a user-defined name.

Example

```
TOOL_BEGIN_BLOCK( my_tool ) {
   // Tool specific code here
}
```

3.9.9 Tool End Block

See Site End Block, Host End Block, Program Execution Control.

Description

User-created tools optionally have a begin block, similar in purpose to Host Begin Block and Site Begin Block. Similarly, a TOOL_END_BLOCK can be defined, similar in purpose to Host End Block, and Site End Block.

When a TOOL_END_BLOCK is defined in the tool code it will be executed when the tool is terminated. The TOOL_END_BLOCK only executes in the tool process; i.e. not in the Host process or Site process(es).

Note: user C-code must ***NOT*** call TOOL_END_BLOCK.

Usage

```
TOOL_END_BLOCK( name ) {
   // Initializations and C code go here
}
```

where:

TOOL_END_BLOCK is a Test System Macro used to denote the beginning of the Tool End Block.

name specifies the desired block name.

Example

```
TOOL_END_BLOCK( my_tool ) {
   // Tool specific code here
}
```

3.9.10 Initialization Hook

See Program Execution Control.

Description

The INITIALIZATION_HOOK facility provides a way to execute user-written code *just because it exists*. This may sound funny, but it is a valuable feature.

If an INITIALIZATION_HOOK has been defined in the test program it will be executed, in the Host process, and all Site processes (see Program Execution Control: Overview). Similarly, if an INITIALIZATION_HOOK has been defined in User Tools code it will be executed, in the tool process. The example shows how OnSite() and site_num() can be used to distinguish between Host, Site, or Tool contexts.

It is up to the code within the INITIALIZATION_HOOK to recognize the context in which it will execute (Host, Site or Tool) and provide conditional statements, as needed, to execute the appropriate code in each context. See Execution Context Functions and the example below.

The INITIALIZATION_HOOK code executes after any Configuration Macros, Host Begin Block, Site Begin Block, and Tool Begin Block.

The INITIALIZATION_HOOK macro must be defined at the global level; i.e. not within the body of another macro or within a C-function.

Usage

```
INITIALIZATION_HOOK( name ) {
   // C code goes here
}
```

where:

INITIALIZATION_HOOK is a Test System Macro used to denote the beginning of the Initialization Hook code.

name is a user-defined name.

Example

```
INITIALIZATION_HOOK( name ) {
   if ( OnHost() ) {
      // Host process code here
   else if ( OnSite() ){
      // Code will execute on all sites
   }
   else if ( site_num() == 1){
      // Code will execute on site-1 only
   }
   else if ( site_num() > 1024) {
```

```
      // Code will execute in User Tools
   }
}
```

3.9.11 Sequence & Binning Table

See Program Execution Control.

Description

The *Sequence and Binning Table* implements a software state machine which controls test execution and binning. Each time **`Start Testing`** is invoked, the Sequence and Binning Table is executed, which in turn, executes Test Blocks, Parking Blocks, and Test Bins, as specified in the Sequence & Binning Test Flow.

- The term *Sequence* refers to the control of the sequence of Test Block, Parking Blocks, and Test Bin execution.
- The term *Binning* refers to the control of software bins used to count test results, as specified in Test Bins.

The Sequence and Binning Table documentation is divided into several parts:

- The Sequence & Binning Table Creation section documents the Test System Macros used to create one or more Sequence and Binning Tables. Note that these macros do not *execute* a Sequence and Binning Table, they are used to create it.
- The Sequence & Binning Test Flow section documents the Test System Macros used to define the test flow and binning portion of the table. Several examples are included.
- The Binning section documents the Test System Macros used to create Test Bins, and Test Bin Groups, which are the built-in software bins supported by the Sequence and Binning Table. The available C-functions which interact with Test Binss are covered in the Test Bin Functions section. The available C-functions which interact with Test Bin Groupss are covered in the Test Bin Group Functions section.
- A given test program can define Multiple Sequence & Binning Tables, allowing a single test program to implement different test flows and/or binning strategies. The Multiple Sequence & Binning Tables section documents the methods used to select one Sequence and Binning Table for use when multiple tables have been defined, and how this selection can be changed while the program remains loaded.

- The macros used to define a Sequence and Binning Table normally execute once, while the test program loads. This occurs after `SITE_CONFIGURATION()` execution has completed and before `SITE_BEGIN_BLOCK()` executes. In order to modify an existing Sequence and Binning Table these macros must be executed again, which is covered in Modifying Sequence & Binning Tables.
- A summary of Test Bins and Test Bin Groups can be displayed using SummaryTool, invoked from UI - User Interface. This displays any bins which have non-zero values.

Once the test program is loaded, UI - User Interface displays a graphical view of the selected Sequence and Binning Table in the UI Sequence and Binning sub-window. Various controls are available there to interactively execute Test Blocks and Test Bins, modify the table (modify the test execution sequence), reset the sequence to the original configuration, set breakpoints at Test Blocks and/or Bins, etc.

3.9.11.1 Sequence & Binning Table Creation

See Sequence & Binning Table.

Definition

Sequence & Binning Tables are created in user-written C-code, using the Test System Macros defined here.

Note: these macros do NOT *execute* a Sequence and Binning Table, they are used to create (define) one.

The `SEQUENCE_TABLE()` macro is used to create a new table, each with a user specified name. This name may subsequently be used:

- With `INCLUDE_SEQUENCE()` to include one [partial] Sequence and Binning Table within another table.
- When Multiple Sequence & Binning Tables are defined, to specify which table is to be enabled using `USE_SEQUENCE_TABLE()`.

The `SEQUENCE_TABLE_INIT()` macro is required by the system software.

The `INCLUDE_SEQUENCE_TABLE` macro can be used to insert an existing [partial] Sequence & Binning Table at the location this macro appears. In effect, it operates as though the macros defining a separate Sequence & Binning Test Flow were copied and

pasted in the same location as the INCLUDE_SEQUENCE_TABLE macro. The most common application for INCLUDE_SEQUENCE_TABLE is to define standard tests (continuity, leakage, etc.) and include them as components of multiple other tables which implement different test flows, typically sort vs. class vs. final, etc.

The USE_SEQUENCE_TABLE() macro is used, within the body of the CONFIGURATION() macro, to select one table when the test program defines multiple Sequence and Binning Tables.

The EXTERN_SEQUENCE_TABLE() macro is used to make an external or forward declaration.

These macros execute once, while the test program loads. This occurs after SITE_CONFIGURATION() execution has completed and before the SITE_BEGIN_BLOCK() executes. In order to modify an existing Sequence and Binning Table these macros must be executed again, which is covered in Modifying Sequence & Binning Tables.

The convention used by the Test Program Wizards puts the Sequence and Binning Table code in the file named *seq_and_bin.cpp*.

Usage

```
SEQUENCE_TABLE( table_name ) {
   SEQUENCE_TABLE_INIT
   // One or more macros defining the Sequence & Binning Test Flow
   // and Binning
   INCLUDE_SEQUENCE_TABLE( other_table ) // Alt method
}
```

where:

SEQUENCE_TABLE is the Test System Macro used to create a Sequence and Binning Table. The SEQUENCE_TABLE macro can only be used at global scope (outside of any C function, not nested, etc.).

table_name is the name of the Sequence and Binning Table being defined. Must be a valid C identifier.

SEQUENCE_TABLE_INIT is a Test System Macro required to initialize the Sequence and Binning Table. Do not omit this from your Sequence and Binning Table. This macro can only be used within the body of the SEQUENCE_TABLE macro, and must occur before any of the macros used to define the Sequence & Binning Test Flow are used.

INCLUDE_SEQUENCE_TABLE is optionally used to insert the specified existing table at the location this macro is placed. This macro can only be used within the body of the SEQUENCE_TABLE macro.

EXTERN_SEQUENCE_TABLE can be used to create an external declaration for the specified Sequence & Binning Table. This is rarely needed.

Example

See Examples

3.9.11.2 Sequence & Binning Test Flow

See Sequence & Binning Table.

Definition

This section documents the Test System Macros used to specify the order of execution of Test Blocks, Parking Blocks, and Test Bins in a Sequence & Binning Table, executed each time **Start Testing** is invoked. These macros are:

Macro	Executes	Branch Action
BIN() BINL()	Test Bins	Unconditional action
CALL() CALLL()	Test Blocks	Unconditional action = execute next statement
STOP() STOPL()	Test Bins	Unconditional Stop

Macro	Executes	Branch Action
TEST() TESTL()	Test Blocks	Conditional pass action, fail action. See Branch Table Actions.
TEST0() thru TEST8() and TESTL0() thru TESTL8()	Test Blocks	Conditional action, using Test Block Integer Return Values. See Branch Table Actions.
TESTP() TEST0P() thru TEST8P() and TESTLP() TESTL0P() thru TESTL8P()	Parking Blocks	Applies only in Multi-DUT Test Programs when the test flow must park some DUT(s) while testing continues on other DUT(s). See Parking Blocks.

These macros are used to:

- Specify the first Test Blocks executed after both the built-in and the optional Before-testing Block(s) have executed.
- Specify the name and sequence of all subsequent Test Blocks, Parking Blocks, and Test Bins executed.
- For each Test Blocks or Test Bins executed what to do next; i.e. what *action* to take. *Actions* can be unconditional, or based on the PASS/FAIL or Test Block Integer Return Values returned from each Test Blocks execution. *Actions* can include executing the NEXT statement in the table, SKIP over the next statement in the table, branch to a label, or STOP. See Branch Table Actions.
- When STOP is encountered the optional (if any) and built-in After-testing Block(s) are executed, before testing actually stops.

The macros used to create the test flow are shown below. These macros have several forms:

- Macros which include a label as the first parameter (TESTL(), CALLL(), BINL(), STOPL())
- Macros which don't include a label (TEST(), CALL(), BIN(), STOP).

- Macros which include a Parking Blocks as the last parameter (TEST0P( ), TESTL0P( ), etc.). These are available with and without the label option.

Labels which are never referenced are OK. The action options are documented in Branch Table Actions.

As noted, eight macros exist which provide for specifying up to 8 unique *action(s)* to be taken when Test Block Integer Return Values (0, 1, through 7) are used. These are further documented under Usage.

These macros are only usable within the scope of the SEQUENCE_TABLE( ) macro body code, and must occur after the SEQUENCE_TABLE_INIT( ) macro.

These macros can be used in any quantity (within computer memory limitations) and in any order. There is no required correspondence between the quantity of each statement used, i.e., there does not have to be a BIN macro for each TEST macro, etc.

Note: STOP exists as both a macro and as an *action*:

- As a *macro*, STOP specifies the last Test Bin to be executed before execution branches to the After-testing Block(s). When the STOP macro is used, the builtin_Pass and builtin_Fail Test Bins are not incremented (see Binning).
- As an *action*, STOP causes execution to branch to After-testing Block(s), without specifying a final Test Bin. One of the built-in bins (builtin_Pass or builtin_Fail) are incremented.

Branch Table Actions

As noted above, some macros require that one or more *actions* be specified. During Sequence & Binning Table execution, except as modified using *action* options, the test flow follows the order that the statements are coded. The conditional actions options are:

Action	Operation
NEXT	Execute the next statemebt in the Sequence & Binning Table
SKIP	Skip over the next statement to execute the following one
label	Branch to the statement containing the specified label
STOP	Exit the Sequence & Binning Table and execute the After-testing Block(s). STOP exists as both a macro and as an *action*, see above.

Usage

The following macros execute the specified Test Block. Execution then flows unconditionally to the next statement in the Sequence & Binning Table. In Multi-DUT Test Programs,the system software determines which DUT(s) are in the Active DUTs Set (ADS) at the time the Test Bin is executed (see Overview):

```
CALL ( Test Block )
CALLL ( label, Test Block )
```

The following macros execute the specified Test Block. In Multi-DUT Test Programs,the system software determines which DUT(s) are in the Active DUTs Set (ADS) at the time the `Test Block` is executed (see Overview). Execution continues by evaluating the value returned from the test block and performing the specified `pass_action` or `fail_action` (see Branch Table Actions). In Multi-DUT Test Programs, a unique `pass_action` and `fail_action` value exists for each DUT in the Active DUTs Set (ADS) at the time the Test Block was executed. The system software tracks and manages Sequence & Binning Table execution for each DUT (see Overview):

```
TEST(          Test Block,  pass_action, fail_action  )
TESTL(label,   Test Block,  pass_action, fail_action  )
```

The following macros execute the specified Test Bin. This does two things:

- The Test Bin is incremented
- If the Test Bin has body code, that code is executed. See Binning.

Execution continues by performing the specified action unconditionally (see Branch Table Actions). In Multi-DUT Test Programs,the system software determines which DUT(s) are in the Active DUTs Set (ADS) at the time the Test Bin is executed (see Overview):

```
BIN(           Test Bin, unconditional_action )
BINL(label,    Test Bin, unconditional_action )
```

The following macros execute the specified Test Bin then exit the Sequence & Binning Table and execute the After-testing Block(s). In Multi-DUT Test Programs,the system software determines which DUT(s) are in the Active DUTs Set (ADS) at the time the Test Bin is executed (see Overview):

```
STOP(          Test Bin  )
STOPL(label,   Test Bin  )
```

The following macros execute the specified Test Block. In Multi-DUT Test Programs,the system software determines which DUT(s) are in the Active DUTs Set (ADS) at the time the Test Bin is executed (see Overview). Execution continues by evaluating the value returned from the test block and performing the action specified for each return value, which can

range from 0 to 7 (see Branch Table Actions). In Multi-DUT Test Programs, a unique `return` value exists for each DUT in the Active DUTs Set (ADS) at the time the Test Block was executed. The system software tracks and manages Sequence & Binning Table execution for each DUT (see Overview):

```
TEST0(Test Block) // Same as CALL
TEST1(Test Block,on0)
TEST2(Test Block,on1,on0)
TEST3(Test Block,on2,on1,on0)
TEST4(Test Block,on3,on2,on1,on0)
TEST5(Test Block,on4,on3,on2,on1,on0)
TEST6(Test Block,on5,on4,on3,on2,on1,on0)
TEST7(Test Block,on6,on5,on4,on3,on2,on1,on0)
TEST8(Test Block,on7,on6,on5,on4,on3,on2,on1,on0)

TESTL0(label,Test Block) // Same as CALLL
TESTL1(label,Test Block,on0)
TESTL2(label,Test Block,on1,on0)
TESTL3(label,Test Block,on2,on1,on0)
TESTL4(label,Test Block,on3,on2,on1,on0)
TESTL5(label,Test Block,on4,on3,on2,on1,on0)
TESTL6(label,Test Block,on5,on4,on3,on2,on1,on0)
TESTL7(label,Test Block,on6,on5,on4,on3,on2,on1,on0)
TESTL8(label,Test Block,on7,on6,on5,on4,on3,on2,on1,on0)
```

The following macros are similar to those above except that a Parking Blocks is specified as the last argument. These macros can only be used in a Magnum 1/2/2x Multi-DUT Test Program:

```
TEST0P(Test Block,Parking Blocks )// Same as CALL
TEST1P(Test Block,on0, Parking Blocks )
TESTP(Test Block,on1,on0, Parking Blocks )
TEST2P(Test Block,on1,on0, Parking Blocks )
TEST3P(Test Block,on2,on1,on0, Parking Blocks )
TEST4P(Test Block,on3,on2,on1,on0, Parking Blocks )
TEST5P(Test Block,on4,on3,on2,on1,on0, Parking Blocks )
TEST6P(Test Block,on5,on4,on3,on2,on1,on0, Parking Blocks )
TEST7P(Test Block,on6,on5,on4,on3,on2,on1,on0, Parking Blocks )
TEST8P(Test Block,on7,on6,on5,on4,on3,on2,on1,on0, Parking Blocks )
TESTL0P(label,Test Block,  Parking Blocks ) // Same as, CALLL
TESTL1P(label,Test Block,on0,Parking Blocks )
TESTLP(label,Test Block,on1,on0, Parking Blocks )
```

```
TESTL2P(label,Test Block,on1,on0, Parking Blocks )
TESTL3P(label,Test Block,on2,on1,on0, Parking Blocks )
TESTL4P(label,Test Block,on3,on2,on1,on0, Parking Blocks )
TESTL5P(label,Test Block,on4,on3,on2,on1,on0, Parking Blocks )
TESTL6P(label,Test Block,on5,on4,on3,on2,on1,on0, Parking Blocks )
TESTL7P(label,Test Block,on6,on5,on4,on3,on2,on1,on0, Parking Blocks )
TESTL8P(label,Test Block,on7,on6,on5,on4,on3,on2,on1,on0, Parking Blocks )
```

where:

Macro names which contain the **L** character require a **label** as the first argument (**CALLL**, **TESTL**, **BINL**, and **STOPL**, etc.) These operate identically to their non-label version. The **label** is user-defined, and acts as a branch target; i.e. a place to branch-to. Each label must be unique within a given Sequence and Binning Table and must be a legal C identifier.

Macro names which contain the **P** character require a Parking Blocks as the last argument (**TEST0P**, **TESTL1P**, etc.) These are only usable in Magnum 1/2/2x Multi-DUT Test Program programs.

Examples

- Example 1: and Example 2: show two simple Sequence & Binning Tables which function identically. They are implemented using two different styles to show the flexibility available.
- Example 3: is more complex, and more typical of a production test flow. It contains a flow chart, and a step-by-step narrative description of how several key features execute, tied to the macro used to define each step. Last is the Sequence & Binning Tables code which implements the entire table.
- Using Magnum 1/2/2x, additional information applies in Multi-DUT Test Programs. See Active DUTs Set (ADS) and Ignored DUTs Set (IDS).

Example 1:

This example uses a common form which groups TEST macros together, and STOP macros together. This style has the advantage of making it easy to see the order in which Test Blocks are executed on good DUT(s). When a fail occurs, execution branches to one of the STOP macros. If no failures occur, the first STOP macro calls the pass_bin.

```
// Bin declarations
TEST_BIN( shorts_bin   ){ check_for_consecutive_shorts(); }
TEST_BIN( opens_bin    ){ check_for_consecutive_opens(); }
```

```
TEST_BIN( gross_bin    ){}
TEST_BIN( pass_bin     ){}

TEST_BIN_GROUP( continuity_fails ) {
   BINS2( shorts_bin, opens_bin )
}
SEQUENCE_TABLE()( sort_1M ) {
   SEQUENCE_TABLE_INIT()
        //  label    block/bin pass action  fail action
        // -------   --------- -----------  -----------
   CALL(             TB_setup                          )
   TEST(             shorts,      NEXT,     l_shorts   )
   TEST(             opens,       NEXT,     l_opens    )
   TEST(             gross_func,  NEXT,     l_gross    )
   STOP(             pass_bin                          )
   STOPL(l_shorts,   shorts_bin                        )
   STOPL(l_opens,    opens_bin                         )
   STOPL(l_gross,    gross_bin                         )
}
```

The first statement executes the Test Block named `TB_setup`. Regardless of the value returned by `TB_setup`, execution unconditionally moves to the next statement. Since no *actions* in this table reference this `CALL`, a label is not necessary.

The `TEST` macros each execute one Test Block. If `PASS` is returned execution proceeds to the `NEXT` statement. If `FAIL` is returned, execution branches to the `STOPL` statement containing the label specified in the `TEST` macro fail action field. At these `STOPL` macros, the specified Test Bin is incremented and execution proceeds to the After-testing Block(s).Note that the two Test Bins `shorts_bin` and `opens_bin` each have body code. If execution reaches these bins the specified function is executed. Also, `shorts_bin` and `opens_bin` are members of the `Test Bin Groups` named `continuity_fails`. If execution reaches either of these bins the `continuity_fails` bin is also incremented.

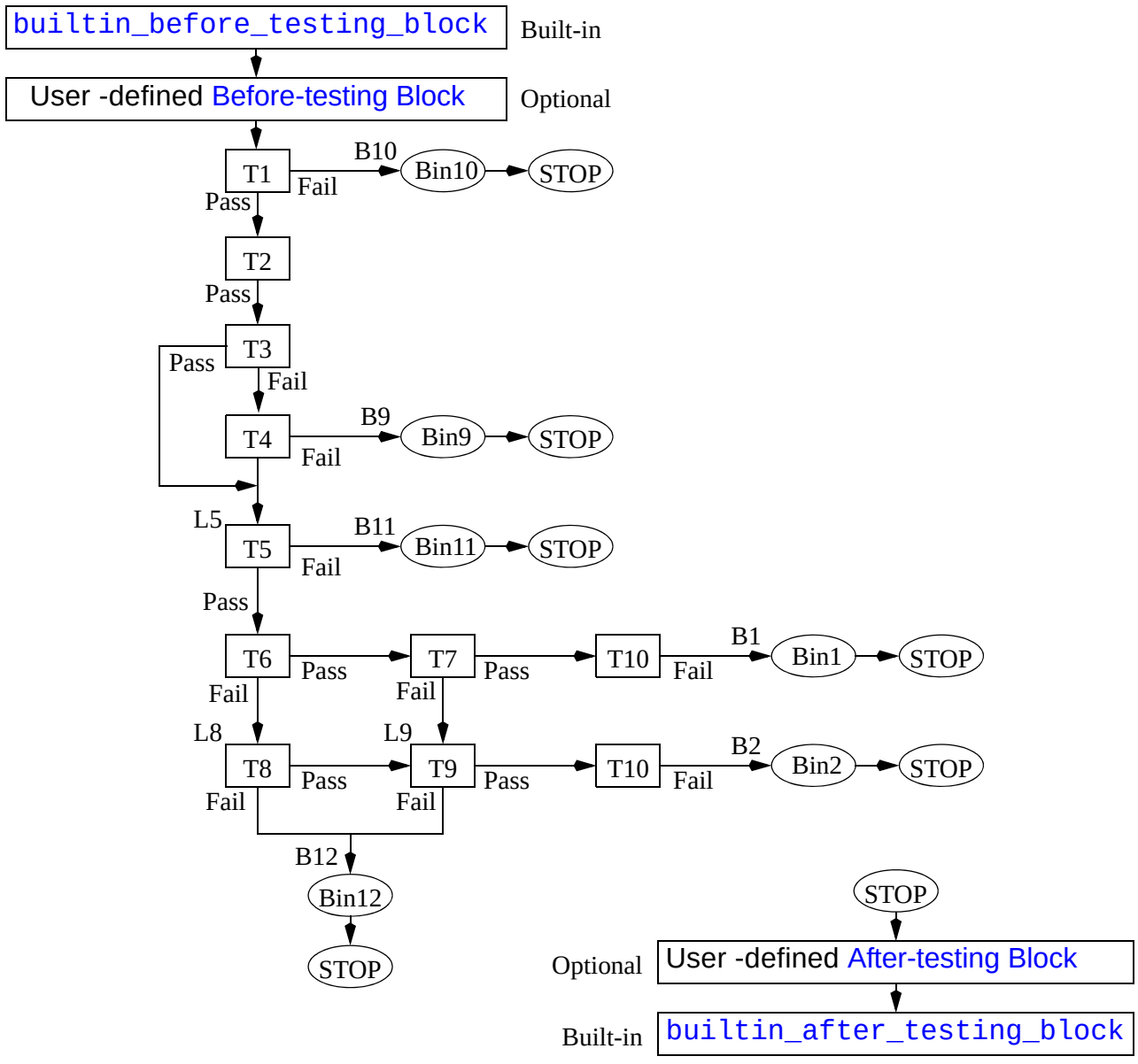
Example 2:

This Sequence and Binning Table is identical in function to the previous one, but uses a different style. In this example, each `TEST` macro is followed by a `BIN` macro. If the Test Block executed by the `TEST` returns `PASS`, execution `SKIP`'s over the `BIN` to the next `TEST`. If the Test Block returns `FAIL`, execution flows to the `NEXT` statement, i.e. the `BIN` immediately following the `TEST` is executed. Notice that no labels are used.

```
// Bin declarations
TEST_BIN( shorts_bin   ){ check_for_consecutive_shorts(); }
TEST_BIN( opens_bin    ){ check_for_consecutive_opens(); }
TEST_BIN( gross_bin    ){}
TEST_BIN( pass_bin     ){}

TEST_BIN_GROUP( continuity_fails ) {
   BINS2( shorts_bin, opens_bin )
}
SEQUENCE_TABLE( sort_1M ) {
   SEQUENCE_TABLE_INIT
        // label   block/bin pass label   fail label
        // -----   --------- ----------   ----------
     CALL(        set_up                           )
     TEST(        shorts,      SKIP,     NEXT      )
     BIN(         shorts_bin,  STOP                )
     TEST(        opens,       SKIP,     NEXT      )
     BIN(         opens_bin,   STOP                )
     TEST(        gross_func,  SKIP,     NEXT      )
     STOP(        gross_bin                        )
     STOP(        pass_bin                         )
}
```

Example 3:

The diagram below shows a flow chart view of a test flow containing the commonly used test flow options. The rectangular boxes represent Test Blocks and the ovals represent Test Bins (see Binning). Branch labels on Test Block use the **L** prefix, and **B** on Bins.



The complete Sequence & Binning table code representing this example follows the descriptions below.

- When **`Start Testing`** is invoked a Before-testing Block named `builtin_before_testing_block` is always executed first. This Before-testing Block is defined by system software.
- If the test program contains one or more user-defined Before-testing Block(s) they are executed next.
- The Test Block or Test Bin specified in the first statement executes next. In the example, Test Block **`T1`** executes next (the name **`T1`**, and all other Test Block names, labels, and bin names are user-defined). The macro which specifies this is:

  ```
  TEST( T1, NEXT , B10 ) // T1 = Test Block, B10 = Label
  ```

  The pass action specified is `NEXT`, thus if **`T1`** PASSes execution continues to the next statement, which executes Test Block **`T2`**. If **`T1`** `FAILs` execution branches to the statement at the label **`B10`**. This must be a macro which supports the label parameter; i.e.:

  ```
  STOPL( B10, Bin10 ) // B10 = label, Bin10 = Test Bin
  ```

  This specifies that execution will `STOP` after executing the Test Bin `Bin10` (see Binning).
- If execution proceeded to the next statement:

  ```
  CALL( T2 ) // T2 = Test Block
  ```

  This specifies that Test Block **`T2`** will execute. Then the test flow proceeds *unconditionally* to Test Block **`T3`**. The `CALL()` macro is used to specify an unconditional test; i.e. **`T2`** has no PASS/FAIL results.
- Test Block **`T3`** executes next.

  ```
  TEST( T3, L5, NEXT ) // T3 = Test Block, L5= Label
  ```

  If **`T3`** `FAILs` the test flow proceeds to the `NEXT` statement; i.e. execute Test Block **`T4`** (more below). If **`T3`** PASSes the test flow branches to the statement containing the label **`L5`**. This must be a macro which supports the label parameter; i.e.:

  ```
  TESTL( L5, T5, NEXT, B11 ) // L5/B11 = Labels, T5 = Test Block
  ```

  This causes Test Block **`T5`** to execute. If **`T5`** PASSes execution continues to the `NEXT` statement, which executes Test Block **`T6`**. If **`T5`** `FAILs` the test flow branches to the statement containing the label **`B11`**; i.e.:

  ```
  STOPL( B11, Bin11 ) // B11 = Label, Bin11 = Test Bin
  ```

  This executes the Test Bin **`Bin11`** (see Binning), then execution `STOP`s
- Earlier, **`T4`** was executed:

```
TEST( T4, SKIP, NEXT ) // T4 = Test Block
BIN( Bin9, STOP )
```

If **T4** PASSes execution SKIPs over the following statement (in this case, BIN()) to the next one (in this case the TESTL() macro containing the **L5** label). If **T4** FAILed execution goes NEXT to the following macro (BIN()), which executes TEST_BIN() Bin9, then execution STOPs.

- Etc.
- When execution reaches a STOP, if test program contains one or more user-defined After-testing Block(s) these execute next.
- The last thing that (always) executes is the After-testing Block named builtin_after_testing_block. This Test Block is defined by system software

The complete definition of this Sequence & Binning table is:

```
// Bin definitions
TEST_BIN( Bin1 ) {}
TEST_BIN( Bin2 ) {}
TEST_BIN( Bin9 ) {}
TEST_BIN( Bin10 ){}
TEST_BIN( Bin11 ){}
TEST_BIN( Bin12 ){}

SEQUENCE_TABLE( my_sort_table ) {
   SEQUENCE_TABLE_INIT
        // label   block/bin pass action  fail action
        // -----   --------- ----------   ----------
   TEST(          T1,       NEXT,        B10        )
   CALL(          T2                                  )
   TEST(          T3,       L5,          NEXT       )
   TEST(          T4,       SKIP,        NEXT       )
   BIN(           Bin9,     STOP                    )
   TESTL( L5,     T5,       NEXT,        B11        )
   TEST(          T6,       NEXT,        L8         )
   TEST(          T7,       NEXT,        L9         )
   CALL(          T10                                 )
   STOP(          Bin1                                )
   TESTL( L8,     T8,       NEXT,        B12        )
   TESTL( L9,     T9,       NEXT,        B12        )
   CALL(          T10                                 )
   STOP(          Bin2                                )
```

```
    STOPL(  B10,    Bin10                               )
    STOPL(  B11,    Bin11                               )
    STOPL(  B12,    Bin12                               )
}
```

3.9.11.3 Multiple Sequence & Binning Tables

See Sequence & Binning Table.

Multiple Sequence & Binning Table can be defined, allowing a single test program to implement different test flows and/or binning strategies.

When a test program defines only one Sequence & Binning Table it is automatically selected when the program is loaded.

When multiple Sequence & Binning Tables are defined one must be selected for use. Several methods are available to make this selection:

- User-written C-code can select a Sequence & Binning Table, by name, using the `USE_SEQUENCE_TABLE()` macro, within the body of the `CONFIGURATION()` macro.
- If, during program load, none is selected by test program code the system software will automatically display a dialog requiring the user to select a table.
- Once the test program is loaded, if multiple Sequence and Binning Tables exist it is possible to select a different table. This can only be done when the currently selected table is not actually executing, which means the selection code must be invoked from code executing in the Host process, User Tools and/or User Dialogs.

The code below shows how this latter case can be implemented:

```
// Disable the currently selected Seq/Bin table
resource_deallocate( S_SequenceTable ); // resource_deallocate()

// Invoke dialog to select new Seq/Bin table
// See resource_select()
CString SBT_cs = resource_select( S_SequenceTable );

// Convert CString to pointer for use by resource_initialize()
// See SequenceTable_find()
SequenceTable *SBT = SequenceTable_find( SBT_cs );
```

```
// Enable the selected Seq/Bin table
   if ( SBT ) resource_initialize ( SBT ); //resource_initialize()
   else output("ERROR: invalid Seq/Bin table [%s]", SBT_cs );
```

Note that this example is only suitable for a single Site system because in multi-site Maverick-I/-II systems and all Magnum 1/2/2x systems the `resource_deallocate()`, `SequenceTable_find()`, and `resource_initialize()` code must be executed on each active Site.

3.9.11.4 Modifying Sequence & Binning Tables

See Sequence & Binning Table.

Normally, the macros used for Sequence & Binning Table Creation and to define the Sequence & Binning Test Flow execute only one time, while the test program loads. This occurs after `SITE_CONFIGURATION()` execution has completed and before the `SITE_BEGIN_BLOCK()` executes.

However, `resource_initialize()` can be used to execute these macros again. This has the effect of enabling conditional code within the SEQUENCE_TABLE macro. For example:

```
// Variable used to conditionally modify Seq/Bin configuration
// This variable can be modified using User Tools, User Dialogs, or
// User Variables Tool, as any variable in C-code.
INT_VARIABLE( SBT_pass, 0, "" ) {}

SEQUENCE_TABLE( SBT1 ) {
   SEQUENCE_TABLE_INIT
   if( SBT_pass_num == 0 ) {
      TEST( tb_1,NEXT,STOP )
      TEST( tb_2,NEXT,STOP )
      TEST( tb_3,STOP,STOP )
      SBT_pass_num++;
   }
   else if (SBT_pass_num == 1 ) {
      TEST( tb_2,NEXT,STOP )
      TEST( tb_3,NEXT,STOP )
      TEST( tb_1,STOP,STOP )
      SBT_pass_num++;
```

```
    }
    else {
       TEST( tb_2,STOP,STOP )
       SBT_pass_num = 0;
    }
 }
```

It is important to note that these conditional statements only execute in the following situations:

- When the test program load initializes the Sequence and Binning Table the first time.
- Each time `resource_initialize(SBT1)` is executed.

Conversely, these conditional statements do NOT execute during the execution of the Sequence and Binning Table; i.e. during Test Block execution. One other note, it is not legal to modify a Sequence and Binning Table while it is executing; i.e. Test Block code cannot modify a Sequence and Binning Table.

3.9.12 Parking Blocks

See Sequence & Binning Test Flow, Overview, Active DUTs Set (ADS).

Overview

The `PARKING_BLOCK()` macro is used to define a Parking Block.

In Multi-DUT Test Programs, during the Sequence & Binning Table execution, it will be common for some DUT(s) to be tested (active) while others are temporarily disabled (parked). For example, when multiple DUTs are tested in parallel, it will be common for some to pass a given test block while others fail. When this occurs, the Sequence & Binning Table will continue to test some DUT(s), for example those that passed, while others are temporarily disabled (parked), for example those that failed.

Detailed operation is described in Overview.

The Magnum 1/2/2x hardware design provides for disabling tester resources per-DUT. The system software manages all the details, at the Sequence & Binning Table level, using the Active DUTs Set (ADS).

By default, when a DUT is parked, the system software disables the hardware connected to it. This means no change of state (PE voltages don't change, DPS voltages don't change, etc.), no test simulus is applied, etc. Later, when appropriate, the DUT is un-parked to resume testing, at which time the system software re-enables the hardware connected to it.

In anticipation of special testing requirements, it may be necessary for user code to be executed as part of parking or unparking a DUT. This is the purpose of the Parking Block, supporting, for example, setting DPS voltages to 0V, controlling PE connections, DUT board relays, etc.

Note the following:

- A Parking Block is defined using the `PARKING_BLOCK` macro. Any number of Parking Blocks may be defined.
- User code is written in the body of the `PARKING_BLOCK`, much like test block code is written in the body of Test Block Macros.
- To be executed, a Parking Block must be specified as an argument to an entry in the Sequence & Binning Table. Specific macros are used, which require a Parking Block argument (`TESTL0P()`, etc.). Note these macros all have a **P** as the last character in the macro name. These macros are only valid in Multi-DUT Test Programs.
- Within a Parking Block, the system defined variable `parking` is available to allow user code to execute conditionally, as determined by whether the parking block is being executed to park the DUT(s) or un-park the DUT(s). Again, see Overview.

See Active DUTs Set (ADS) for a detailed example of Parking Block use.

Usage

```
PARKING_BLOCK( name ){
   // User code
}
```

where:

name identifies the name of the Parking Block. These names are used in the Sequence & Binning Table to specify which Parking Block is executed by a given Sequence & Binning Table statement.

Example

The following example outputs whether parking or un-parking is being performed, followed by a list of DUT(s) being parked/un-parked. Last, is a conditional statement suitable for adding user code which executes differently when parking vs. un-parking:

```
PARKING_BLOCK( myPB ) {
   output( " %sing => \\", parking ? " Park" : " Unpark" );
   DutNumArray duts;
   int size = active_duts_get( &duts ); // active_dut_get()
   for ( int i = 0; i < size; ++i )
      output( "t_dut%d \\", duts[ i ] + 1 );
   output("");
   if( parking ) {
        // Parking code here as desired
   }
   else {
      // UN-Parking code here as desired
   }
}
```

3.9.13 Test Flow Synchronization

See Sequence & Binning Test Flow.

Note: first available in software release h2.2.7/h1.2.7.

Overview

Normally, when executing the Sequence & Binning Table, program execution on each site and on the host are *in-sync* only at the time that start-testing is first invoked. Subsequently, no execution synchronization is attempted (or typically required) between the individual program instances executing on each site.

The synchronization facility described here may be used to re-synchronize test flow execution on all sites, with or without host involvement. This might be required, for example, in wafer testing situations where all the sites need to by synchronized before changing the substrate voltage. Or when all sites need to be at a particular place before controlling some external instrument connected to the host. This test flow synchronization facility is designed to provide a simple, reliable, and efficient built-in mechanism to address the most common synchronization needs.

Executing `remote_synchronize()` causes site execution to pause and wait until execution on all *active* sites has reach the same point (inactive sites are those which have

already stopped Sequence & Binning Table execution). Then, depending on the type of the `block` argument specified, a synchronization block will execute, either on the host or on all active sites:

- When synchronization requires host code execution, a `HostSynchronizationBlock` is specified as the `block` argument to `remote_synchronize()`. A `HostSynchronizationBlock` is defined using the `HOST_SYNCHRONIZATION_BLOCK()` macro.
- When synchronization requires only site code execution, a `SiteSynchronizationBlock` is specified as the `block` argument to `remote_synchronize()`. A `SiteSynchronizationBlock` is defined using the `SITE_SYNCHRONIZATION_BLOCK()` macro.

During Sequence & Binning Table execution, executing `remote_synchronize(sync_block)` on a given site causes execution to pause and wait on that site. Execution will remain paused until all active sites have each executed `remote_synchronize(sync_block)` (the same block name, here called `sync_block`, is required), at which time the body code of the specified synchronization block will execute. Two scenarios are possible:

- The specified synchronization block is a `HostSynchronizationBlock`.
- The specified synchronization block is a `SiteSynchronizationBlock`.

For example:

```
TEST_BLOCK( id ){
   \\... other code as desired...
   UNIT64 m = remote_synchronize( sync_block ); // Pause/wait here
   \\ If desired, examine m (bit-mask of synchronized sites ) here.
   \\... other code as desired...
   return( ... ); // Appropriate return value
}
```

Given the `TEST_BLOCK` example above, if `sync_block` is a `HostSynchronizationBlock`:

- The `HostSynchronizationBlock` body code is executed only after Sequence & Binning Table execution on all active sites has stopped because `remote_synchronize(sync_block)` was executed on every active site.
- `HostSynchronizationBlock` body code then executes only on the host.
- Execution on all active sites waits until the `HostSynchronizationBlock` body code execution ends. Then, execution restarts on all active sites, beginning with the statement immediately following `remote_synchronize(sync_block)`.

Alternatively, given the TEST_BLOCK example above, if sync_block is a SiteSynchronizationBlock:

- The SiteSynchronizationBlock body code is executed only after test flow execution on all active sites has stopped because remote_synchronize(sync_block) was executed on every active site.
- The SiteSynchronizationBlock body code then executes on all active sites.
- When the SiteSynchronizationBlock body code execution completes, execution continues on all active sites beginning with the statement immediately following remote_synchronize(sync_block).

Also note the following:

- remote_synchronize() is only useful when executed in site code.
- remote_synchronize() is only effective when the Sequence & Binning Table is executing. As a practical matter, this means executed directly or indirectly from a Test Block.
- A site which has stopped executing the Sequence & Binning Table (possibly because all DUTs on that site have finished testing) before remote_synchronize() has executed on any other sites will not affect synchronization.
- The Active DUTs Set (ADS) has no effect on synchronization, except that a given site may be idle if all DUTs on that site are inactive.
- Within the body code (scope) of both the HostSynchronizationBlock and SiteSynchronizationBlock an implicitly declared UINT64 variable named mask is available. This is a bit-wise mask indicating which sites wait for synchronization. See examples below.
- Invoking UI's Stop-testing control will interrupt any pending synchronization and stop testing as expected.
- HostSynchronizationBlock and SiteSynchronizationBlock are Resources, just like PINLIST, TEST_BLOCK, etc.

Usage

This synchronization facility uses the following:

```
HOST_SYNCHRONIZATION_BLOCK( block ){ [body code ] }
SITE_SYNCHRONIZATION_BLOCK( block ){ [body code ] }
```

```
UINT64 remote_synchronize(
          HostSynchronizationBlock *block,
          DWORD timeout DEFAULT_VALUE( INFINITE ) );
UINT64 remote_synchronize(
          SiteSynchronizationBlock *block,
          DWORD timeout DEFAULT_VALUE( INFINITE ) );
```

where:

The HOST_SYNCHRONIZATION_BLOCK macro defines a HostSynchronizationBlock. The SITE_SYNCHRONIZATION_BLOCK macro defines a SiteSynchronizationBlock.

block identifies the HostSynchronizationBlock or SiteSynchronizationBlock.

body-code is user-written code which executes as described above.

timeout is optional and, if used, specifies the amount of time to wait for all active sites to pause. Default = INFINITE.

Note: INFINITE is the recommended **timeout** value. When a non-INFINITE time-out value is specified and a time-out occurs remote_synchronize() returns 0. When this occurs, a corresponding HostSynchronizationBlock will not receive notice that all sites have paused and its body code will not execute. A corresponding SiteSynchronizationBlock will be executed but the site mask value will be = 0x0.

---

remote_synchronize() returns a bit-wise mask indicating which sites were synchronized. Sites which had stopped Sequence & Binning Table execution will not be in this mask.

Example

Two test blocks are used below to demonstrate the two synchronization scenarios:

```
HOST_SYNCHRONIZATION_BLOCK( hsb1 ){
   output("Sites[%I64x] are waiting.", mask );
   output("None of these sites will return from their call to
remote_synchronize() until this HOST_SYNCHRONIZATION_BLOCK() code
ends." );
}
```

```
SITE_SYNCHRONIZATION_BLOCK( ssb1 ){
   output("Sites[%I64x] were waiting and started executing this
code in parallel.", mask );
}
// This test block demonstrates synchronization with Host code
// execution. Program execution on all sites will stop below, and
// wait until all active sites have executed
// remote_synchronize( hsb1 ).
// Then, HOST_SYNCHRONIZATION_BLOCK(hsb1) body code will be
// executed on the host only. When this ends, execution will be
// restarted on all active sites, with the instruction immediately
// following remote_synchronize( hsb1 ).
TEST_BLOCK( host_synchronization_example ){
   // ... other code as desired...
   UNIT64 m = remote_synchronize( hsb1 );
   // ... other code as desired...
   return MULTI_DUT;
}
// This test block demonstrates synchronization with no Host code
// execution. Program execution on all active sites will stop
// below, and wait until all these sites have all executed
// remote_synchronize( ssb1 ).
// Then, SITE_SYNCHRONIZATION_BLOCK( ssb1 ) will be executed on all
// active sites, in parallel. Execution will continue on each site
// independently.
TEST_BLOCK( site_synchronization_example ){
   // ... other code as desired...
   UNIT64 m = remote_synchronize( ssb1 );
   // ... other code as desired...
   return MULTI_DUT;
}
```

3.9.14 Test Blocks

See Program Execution Control.

- Overview
- Test Block Macros
- Sequential Test Block
- Test Block Integer Return Values
- Before-testing Block, After-testing Block
- Conflict List
- Conflict List Macros
- Test Numbers
- Setup Numbers

3.9.14.1 Overview

See Test Blocks, Program Execution Control.

Conceptually, a test block is a unit of user-written code which programs the tester hardware and executes one or more tests.

Each time a **`Start Testing`** is invoked, the Sequence & Binning Table executes zero or more test blocks to test DUT(s). In general, test blocks contain user code which executes in the Site processes any time the Sequence & Binning Table executes.

Statements in the Sequence & Binning Table, determine which test block(s) are executed, in which order they are executed, and how the next step of Sequence & Binning Table execution is determined.

Using Magnum 1/2/2x, two forms of test blocks are available:

- Standard test block, defined using the `TEST_BLOCK` macro.
- Sequential Test Block, defined using the `TEST_BLOCK_SEQUENTIAL` macro.

A standard test block is used when test conditions allow all enabled DUT(s) to be tested concurrently, in parallel. This is the only type of test block supported using Maverick-I/-II.

A Sequential Test Block is used when test conditions are such that some enabled DUT(s) conflict with other enabled DUT(s); i.e. the test block code must be executed more than once, with a subset of enabled DUT(s) tested for each execution. See Sequential Test Block and Conflict List.

Test blocks are created using Test Block Macros.

3.9.14.2 Test Block Macros

See Test Blocks, Program Execution Control.

Description

The macros documented in this section are used to define Test Blocks:

- The `TEST_BLOCK()` macro defines a standard test block.
- The `TEST_BLOCK_SEQUENTIAL` macro defines a Sequential Test Block.
- The `EXTERN_TEST_BLOCK()` macro is used to declare a test block or Sequential Test Block as external.
- The `MULTI_DUT_TEST_BLOCK()` macro defines a parallel test block, which is the same as defined using `TEST_BLOCK` except that the test block implicitly returns `MULTI_DUT`, eliminating the need for user code to return a value. See Overview. Only usable when the Pin Assignment Table uses the parallel test macros; i.e. `ASSIGN_1DUT()`, `ASSIGN_2DUT()`, etc.
- The `MULTI_DUT_TEST_BLOCK_SEQUENTIAL()` macro defines a Sequential Test Block, which is the same as defined using `TEST_BLOCK_SEQUENTIAL` except that the test block implicitly returns `MULTI_DUT`, eliminating the need for user code to return a value. See Overview. Only usable when the Pin Assignment Table uses the parallel test macros; i.e. `ASSIGN_1DUT()`, `ASSIGN_2DUT()`, etc.
- The `MULTI_DUT_CALL_BLOCK()` macro defines a test block intended to be executed using the `CALL()` or `CALLL()` macro. Unlike `MULTI_DUT_TEST_BLOCK`, a `MULTI_DUT_CALL_BLOCK` does not require that a test result value be returned.

It is the user-written code within a test block which controls how the hardware is programmed, tests are executed, test results are used, etc.

The `TestBlock_find()` function can be used to get a pointer to a Test Block, given its name. Using this pointer, the Test Block can be explicitly executed using `invoke()`,

typically from a User Tool or User Dialog. Note: it is ***NOT necessary*** to use `invoke()` when executing test blocks via the Sequence & Binning Table.

The name of the currently executing test block can be obtained using the `current_test_block()` function.

Usage

```
TEST_BLOCK( name ){
   // Test block code
   return( int );
}
TEST_BLOCK_SEQUENTIAL( name, Conflict_List ) {
   // Test block code
   return( MULTI_DUT );
}
MULTI_DUT_TEST_BLOCK( name ){
   // Test block code
}
MULTI_DUT_TEST_BLOCK_SEQUENTIAL( name, Conflict_List ) {
   // Test block code
}
MULTI_DUT_CALL_BLOCK( name ){
   // Test block code
   // No return value
}
EXTERN_TEST_BLOCK( name )
```

where:

`name` identifies the name of the test block and must be a legal C identifier.. These names are used in the Sequence & Binning Table to specify which test block is executed by a given Sequence & Binning Table statement.

`Conflict_List` identifies the Conflict List to be used during the execution of the Sequential Test Block. A conflict list is defined using the Conflict List Macros.

The `TEST_BLOCK*(` macro requires that user code return an integer value. The default Sequence & Binning Table operation supports return values of 0 through 7, where 0 = FAIL, and 1 through 7 = PASS. See Test Block Integer Return Values.

The TEST_BLOCK_SEQUENTIAL() macro is only usable in Multi-DUT Test Programs and thus must return the value MULTI_DUT.

Both MULTI_DUT_TEST_BLOCK() and MULTI_DUT_TEST_BLOCK_SEQUENTIAL() implicitly return MULTI_DUT. These macros are only usable in test programs in which the Pin Assignment Table uses the parallel test macros; i.e. ASSIGN_1DUT(), ASSIGN_2DUT(), etc.

The MULTI_DUT_CALL_BLOCK() macro requires that no value be returned.

Example

The following example creates a test block named shorts:

```
TEST_BLOCK( shorts ) {
   output("Executing => %s", current_test_block() );
   dps( 0 V, pl_all_Vcc );
   vpar_force( 2 V );
   ipar_high( 100 UA );
   ipar_low( -100 UA );
   back_voltage( 0 V );
   back_voltage_enable( TRUE );
   PFState pf = partest( passnicl, all_signal_pins );
   // Other code as desired: datalogging, etc.
   return pf;
}
```

3.9.14.3 current_test_block()

See Test Blocks, Program Execution Control.

Description

The current_test_block() function is used to get the name of the currently executing test block.

current_test_block() does *not* report Before-testing Block, After-testing Block.

current_test_block() returns the name of a TEST_BIN() if executed within the optional Test Bin body code.

If a call-back function is registered using `install_debug_hook()`, any time the call-back executes the name of the test block being executed can be obtained using the `current_test_block()`.

Usage

```
CString current_test_block();
```

where `current_test_block()` returns the name of the currently executing test block, as a `CString`.

Example

```
TEST_BLOCK( myTB1 ){
   // Other code as desired
   output(" Executing => %s", current_test_block() );
   // Other code as desired
}
```

3.9.14.4 Sequential Test Block

See Test Blocks, Conflict List, Program Execution Control.

Using Magnum 1/2/2x, when executing the Sequence & Binning Table, by default each Test Block is executed once, concurrently testing all DUT(s) in the Active DUTs Set (ADS).

In situations where it is not appropriate to test all active DUTs, a Conflict List may be defined and used in conjunction with a sequential test block, to selectively test subsets of the active DUTs. A sequential test block is executed the number of times necessary to test the active DUTs but without testing any DUTs which are in conflict.

A sequential test block is defined using the `TEST_BLOCK_SEQUENTIAL` macro. A sequential test block differs from the normal test block in that the sequential test block requires that a Conflict List be associated. Then, as the Sequence & Binning Table is executed, when a sequential test block is encountered the specified conflict list controls how many times the sequential test block executes, with each execution testing a subset of active DUTs which are not in conflict.

3.9.14.5 Test Block Integer Return Values

See Test Blocks, Program Execution Control.

Description

Test Blocks typically return a PASS or FAIL value. However, they can also return an integer between 0 and 7.

This enables increased flexibility in the Sequence & Binning Test Flow where the test flow can branch to a different label based on the return value. For example:

```
// Test Block Code
TEST_BLOCK( TB_1 ) {
   // Other test block code
   return result_int;
}
// Sequence & Binning Code
TEST5( TB_1, label_4, label_3, label_2, label_1, STOP )
TESTL(label_4, TB_on4, STOP, STOP );
TESTL(label_3, TB_on3, STOP, STOP );
TESTL(label_2, TB_on2, STOP, STOP );
TESTL(label_1, TB_on1, STOP, STOP );
```

In this example, during Sequence & Binning Table execution, the TEST5 macro evaluates the return value from TB_1 and takes the following action based on that value:

- 0 = STOP
- 1 = branch to *label_1*, which then executes the TEST_BLOCK named TB_01.
- 2 = branch to *label_2*, which then executes the TEST_BLOCK named TB_02.
- 3= branch to *label_3* which then executes the TEST_BLOCK named TB_03.
- 4= branch to *label_4*, which then executes the TEST_BLOCK named TB_04.

The TEST1 through TEST7 macros are covered in the Usage section of Sequence & Binning Test Flow.

3.9.14.6 Before-testing Block, After-testing Block

See Test Blocks, Program Execution Control.

Description

The `BEFORE_TESTING_BLOCK()` macro defines a before-testing block, which is a user-defined block of code that will always be executed immediately before the first test block specified in the Sequence & Binning Table. This is a good place to perform any final preparations for device testing, such as closing switches to connect DPS and tester pins to the DUT (`dps_connect()`, etc.), or doing some final set up of the APG, initializing variables, etc.

Similarly, the `AFTER_TESTING_BLOCK()` macro defines an after-testing block, which is a user-defined block of code that will always be executed immediately after the first test block specified in the Sequence & Binning Table. This is a good place to perform any special power down, or binning activities.

Note: any user-defined before-testing block(s) and/or after-testing block(s) do execute when executing one test block using mouse controls in the UI Sequence and Binning sub-window.

Note the following:

- After-testing blocks are guaranteed to execute, even if *Stop Testing is asserted from UI*.
- Before-testing blocks and after-testing blocks do not (and must not) appear in the Sequence & Binning Table; just the fact they are defined causes them to execute automatically. And, unlike other test blocks, before-testing and after-testing blocks do not return a PASS/FAIL value.
- Multiple before-testing block(s) and/or after-testing block(s) may be defined. The block names are stored in a list, and will be executed in the order they were defined.
- In every test program, the system software automatically defines one before-testing block and one after-testing block. These are named:
  - `builtin_before_testing_block`
  - `builtin_after_testing_block`
- The `builtin_before_testing_block` contains the following:

```
BEFORE_TESTING_BLOCK( builtin_before_testing_block ) {
   // Closes pin relays on all used signal pins
   // Does NOT close any DPS switches
   pin_connect( builtin_UsedPins );
}
```

where **builtin_UsedPins** is a pin list automatically created by the system software and contains all of the signal pins defined in the test program.

- The builtin_after_testing_block contains the following:

```
   AFTER_TESTING_BLOCK( builtin_after_testing_block ) {
   pmu_disconnect( );
   vihh( 0.0, builtin_UsedPins );
   vih( 0.0, builtin_UsedPins );
   voh( 0.0, builtin_UsedPins );
   vil( 0.0, builtin_UsedPins );
   vol( 0.0, builtin_UsedPins );
   vtt( 0.0, builtin_UsedPins );
   hv_pmu_disconnect( );
   hv_voltage_set( 0.0 );
   dps( 0.0, builtin_UsedDPS );
   ptu_disconnect( builtin_UsedPins );
   disconnect( builtin_UsedPins );
}
```

where **builtin_UsedPins**, **builtin_UsedAVS** and **builtin_UsedDPS** and are pin lists automatically created by the system software based on the signal pins, DPS pins, and ATC pins used in the test program. The Analog Test Channel (ATC) is documented in a separate manual.

- The system software always executes the builtin_before_testing_block first, even when the user has also defined a before-testing block. User-code can override the builtin_before_testing_block by defining a before-testing block with the same name. The user may also create additional before-testing block(s) which will be executed after the builtin_before_testing_block. These will have names created by the user.
- Similarly, the system software always executes the builtin_after_testing_block last, even when the user has also defined a after-testing block. User-code can override the builtin_after_testing_block by defining a after-testing block with the same name. The user may also create additional after-testing block(s) which will be executed before the builtin_after_testing_block. These will have names created by the user.

- The `BeforeTestingBlock_find()` function can be used to get a pointer to a before-testing block, given its name. The `AfterTestingBlock_find()` function can be used to get a pointer to an after-testing block, given its name. Using this pointer, the before-testing block or after-testing block can be explicitly executed using `invoke()`. However, it is not necessary to use `invoke()` to execute a before-testing block or after-testing block during typical testing operations.

Usage

```
BEFORE_TESTING_BLOCK( name ) { body-code }
AFTER_TESTING_BLOCK( name ) { body-code }
```

where:

`BEFORE_TESTING_BLOCK` and **`AFTER_TESTING_BLOCK`** are Test System Macros used to define a before-testing block or after-testing block.

`name` is the name assigned to the before-testing block or after-testing block, and must be a legal C identifier.

`body-code` represents the user-written body-code to be executed when the before-testing block or after-testing block executes.

Example

```
BEFORE_TESTING_BLOCK( tb_initialize ) {
   dps_connect( pl_vcc );
   data_strobe( pl_data );
   vclamp( 7 V, -3 V );
   // Test flow setup – for parallel test routines
   curr_dut_mask = init_dut_mask;
   clear_dut_bins();// user function in datalog.cpp
   // Do NOT return a value from a Before or After Testing Block
}
AFTER_TESTING_BLOCK( my_after_testing_block ) {
   my_power_down_func();
   my_binning_eval_func();
}
```

3.9.14.7 Conflict List

See Test Blocks, Program Execution Control.

Description

Using Magnum 1/2/2x, when executing the Sequence & Binning Table, by default each Test Block is executed once, concurrently testing all DUT(s) in the Active DUTs Set (ADS).

In situations where it is not appropriate to test all active DUTs, a conflict list may be defined and used in conjunction with a Sequential Test Block, to selectively test subsets of the active DUTs. A Sequential Test Block is executed the number of times necessary to test the active DUTs but without testing any DUTs which are in conflict.

A conflict list is defined, using Conflict List Macros, to identify sets of DUTs which cannot be tested concurrently. Then, as the Sequence & Binning Table is executed, when a Sequential Test Block is encountered the specified conflict list controls how many times the Sequential Test Block executes, with each execution testing a subset of active DUTs which are not in conflict.

3.9.14.8 Conflict List Macros

See Test Blocks, Sequential Test Block, Conflict List, Program Execution Control.

Note: first available in software release h1.1.23.

Description

User code creates and names a Conflict List using the `CONFLICT_LIST` and related macros documented below.

Within the body of the `CONFLICT_LIST` macro the `CONFLICTn` macros or `COMPATIBLEn` macros are used to identify DUT(s) which cannot be concurrently tested; i.e. are in conflict. The name of the macro used (`CONFLICT2`, `CONFLICT3`, or `COMPATIBLE3`, `COMPATIBLE4`, etc.) is determined by the number of DUTs in each conflict, which corresponds to the number of `DutNum` arguments specified in the macro. For example, to specify that even numbered DUTs are in conflict with odd numbered DUTs the `CONFLICT2` macro could be used:

```
CONFLICT_LIST( EvenOddDutsConflict ) {
   CONFLICT2( t_dut1, t_dut2 )
   CONFLICT2( t_dut1, t_dut4 )
   CONFLICT2( t_dut2, t_dut1 )
   CONFLICT2( t_dut2, t_dut3 )
   CONFLICT2( t_dut3, t_dut2 )
   CONFLICT2( t_dut3, t_dut4 )
   CONFLICT2( t_dut4, t_dut1 )
   CONFLICT2( t_dut4, t_dut3 )
   // Etc.
}
```

Note that since `CONFLICT2( x, y ) == CONFLICT2( y, x )`, the above can be simplified to:

```
CONFLICT_LIST( EvenOddDutsConflict ) {
   CONFLICT2( t_dut1, t_dut2 )
   CONFLICT2( t_dut2, t_dut3 )
   CONFLICT2( t_dut3, t_dut4 )
   CONFLICT2( t_dut4, t_dut1 )
   // Etc.
}
```

Or, the same result can be obtained using:

```
CONFLICT_LIST( EvenOddDutsConflict ) {
   COMPATIBLE2( t_dut1, t_dut3 )
   COMPATIBLE2( t_dut2, t_dut4 )
   // Etc.
}
```

Using the `COMPATIBLEn` macros, each line lists DUT(s) that can be tested concurrently; i.e. DUT(s) which are NOT listed are, by definition, in conflict.

Conflict Lists must be defined at global scope; i.e. the `CONFLICT_LIST` macro cannot be used within the body of another macro, C function, etc.

A Conflict List is used in conjunction with the `TEST_BLOCK_SEQUENTIAL` macro, by specifying its name as the last parameter. For example:

```
TEST_BLOCK_SEQUENTIAL( my_test_block, EvenOddDutsConflict ){
   // Test block code
}
```

As described in Conflict List, the code in a Sequential Test Block may execute more than once; the specific number of times is determined by which DUT(s) are enabled in the Active

DUTs Set (ADS) and the relationships defined in the specified Conflict List. For example, given the `EvenOddDutsConflict` Conflict List example above, the table below shows how many times the Sequential Test Block will execute for various combinations of active DUT(s):

Active DUTs Set Members	Test Block Code Execution Count
t_dut1	1
t_dut2	1
t_dut1 t_dut3	1
t_dut2 t_dut4	1
t_dut1 t_dut2 t_dut3	2
t_dut1 t_dut2 t_dut4	2
t_dut1 t_dut3 t_dut4	2
t_dut2 t_dut3 t_dut4	2
t_dut1 t_dut2 t_dut3 t_dut4	2

Note the following:

- If only `CONFLICT2` is used to define a Conflict List, the test block code will never execute more than 2 times, regardless of how many DUT(s) are in the Active DUTs Set (ADS). Similarly, if only `CONFLICT4` is used to define a Conflict List, the test block code will never execute more than 4 times, regardless of how many DUT(s) are in the Active DUTs Set (ADS). Etc.
- If a mix of `CONFLICTn` macros are used, complex combinations are possible. However, this is unlikely to be useful in the real world, and thus is not further documented here.

Usage

The following macro is used to create a new Conflict List, with the specified **name**:

```
CONFLICT_LIST( name )
```

The following macro is used to make a forward or external Conflict List declaration:

```
EXTERN_CONFLICT_LIST( name )
```

The following macros are used to identify specific DUT conflicts:

```
CONFLICT1( d1 )
CONFLICT2( d1, d2 )
CONFLICT3( d1, d2, d3 )
... snip ...
CONFLICT32(  d1, d2, d3, d4, d5, d6, d7, d8, d9, d10,
             d11, d12, d13, d14, d15, d16, d17, d18, d19, d20,
             d21, d22, d23, d24, d25, d26, d27, d28, d29, d30, d31,
             d32 )
```

The following macros are used to identify specific DUT compatibilities:

```
COMPATIBLE1( d1 )
COMPATIBLE2( d1, d2 )
COMPATIBLE3( d1, d2, d3 )
... snip ...
COMPATIBLE32(d1, d2, d3, d4, d5, d6, d7, d8, d9, d10,
             d11, d12, d13, d14, d15, d16, d17, d18, d19, d20,
             d21, d22, d23, d24, d25, d26, d27, d28, d29, d30, d31,
             d32 )
```

where:

The name of the macro identifies how many `DutNum` arguments must be specified.

d1, **d2**, through **d32** identify the DUTs which are in conflict or are compatible. Legal values for are of the `DutNum` enumerated type (`t_dut1`, etc.).

Example:

See Description.

3.9.14.9 Test Numbers

See Test Blocks, Program Execution Control.

Description

The `test_number()` function may be used to explicitly set a test number or retrieve the current test number value.

Every test in a test program can be uniquely identified by test block name and a test number within each test block. This is important when gathering test data and when setting interactive breakpoints, as described in Breakpoint Monitor.

Test Blocks names are assigned by the user when test blocks are created. `Test numbers` are assigned automatically by the system software within each test block for the predefined functions that execute parametric and functional tests. A test number counter, starting at zero when the test block is entered, is automatically incremented by the system software at each execution of the following functions:

- `partest()`
- `ac_partest()`
- `funtest()`
- `test_supply()`
- `ac_test_supply()`
- `hv_test_supply()`
- `hv_ac_test_supply()`
- `ptu_partest()`
- `ptu_ac_partest()`

For example, if a `test block` named `leakage` executes ten `partest()` functions, the first one would be test number one, the second would be test number two, etc. The fourth `partest()` in this test block would be identified as: Test Block: leakage, Test # 4.

In certain instances, the user may want to manipulate the test number. Functions are provided that allow the user to retrieve the current test number or set the test number to a value. Incrementing the test number can also be done using these functions.

Also note that there are set-up numbers assigned by the system software for all tester defined functions in a test block. See the description under Setup Numbers for more information.

Usage

```
int test_number();              // Return current test number
int test_number( int value ); // Set test number with value.
                                // Return previous test number
```

where **`value`** is an integer (int).

Examples

Example 1:

```
test_number( 50 );
```

The test number is set to 50.

Example 2:

```
int curr_tn = test_number();
```

This example retrieves the current test number and stores it in the user-defined variable named `curr_tn`.

Example 3:

```
test_number( test_number() + 1 );
```

This increments the test number by retrieving the current test number and adding one to it. This is also automatically done by the system software throughout a test block unless overridden by the explicit use of `test_number()`.

3.9.14.10 Setup Numbers

See Test Blocks, Program Execution Control.

Description

The `setup_number()` function may be used to explicitly set a setup number or retrieve the current a setup number value.

Every C function documented in this manual except the five test functions shown below are uniquely identified within each test block by a *setup number*.

The following five test functions are identified by Test Numbers, not setup numbers.

- partest()
- ac_partest()
- funtest()
- test_supply()
- ac_test_supply()

Executing any of the other C functions documented in this manual causes the setup number to be incremented. Each time one of the five test functions above is executed, the setup number is reset to zero.

Note: only functions created by Nextest cause the setup number to increment; i.e. user-created functions have no effect on setup number unless they call the `setup_number()` function documented here.

During interactive debug, using the Breakpoint Monitor, breakpoints can be set to act on a specific setup number; i.e. on any function documented in this manual. This allows execution to halt before or after executing a specific C function.

Usage

The following function changes the setup number to the value specified:

```
void setup_number( int value );
```

The following function retrieves the current setup number:

```
int setup_number();
```

where **`value`** specifies the desired setup number.

Examples

Example 1:

```
setup_number( 50 );
```

The setup number is set to 50.

Example 2:

```
int my_sn = setup_number();
```

This example retrieves the current setup number and stores it in the user-defined variable called `my_sn`.

Example 3:

```
setup_number( setup_number() + 1 );
```

This increments the setup number by retrieving the current value and adding one to it.

3.9.15 Delay()

See Program Execution Control.

Description

The `Delay()` function may be used to pause test program execution for a specified time. This uses the same hardware timer as `partime()`, and that used by the system software for internal relay and voltage settling time delays.

Usage

```
void Delay( double Value );
```

where:

`Value` specifies the desired delay. Legal values are 100uS to 1S. Units may be used (see Specifying Units).

Example

```
// Other code as desired...
Delay( 10 MS ); // Wait 10mS before proceeding
// Other code as desired...
```

3.9.16 Error Line Reset from CPU: reset_error()

See Program Execution Control.

Description

The `reset_error()` function will reset the Pin Electronics error *latches*, for all tester channels on the Site (this is not a per-pin or per-PE board function). See Error Flag vs. Error Latch.

Once the error latches are reset the `test_pin()` and `test_pin_first_error()` functions will return PASS.

Usage

```
void reset_error();
```

Example

```
reset_error();
```

All pin electronics error latches are cleared.

3.9.17 Control of Branch on Error Flag

See Over-programming Controls and Parallel Test, Over-programming Control Stimulus Selection.

Description

See Over-programming Controls and Parallel Test for a detailed overview of the purpose and operation of the `error_flag_enable()` function.

Note: the `error_flag_enable()` function documented here is *not* needed when testing a single DUT per-site, including when using `MAR OVER` to inhibit over-programming.

Note: the features noted below assume that all pins of a given set of 8 are only used by one DUT; i.e. `a_1` through `a_8` are used by one DUT, `b_1` through `b_8` are used by one DUT, etc. Note that other restrictions exist: see Over-programming Controls and Parallel Test.

`error_flag_enable(TRUE)` is the global default set during initial program load. The system software does not otherwise modify this setting.

Executing `error_flag_enable(TRUE)` in user code restores default error flag and error latch operation. More below.

`error_flag_enable(FALSE)` does the following:

- Globally, for all DUT(s), prevents the corresponding error *flags* from that DUT from affecting test pattern branch-on-error operation when an error *latch* is set for that DUT. Thus, when a DUT is defective (i.e some associated error latches are set), the adaptive test pattern will ignore that DUT when making branch-on-error decisions. See Over-programming Controls and Parallel Test.
- Inhibits the programming stimulus, identified using `over_inhibit()`, for any DUT(s) which have an error latch set. Again, see Over-programming Controls and Parallel Test.

The following table summarizes this operation:

Table 3.9.17.0-1 error_flag_enable() Operation and Options

Function Option	Error Latches	Error Flags	
`error_flag_enable(TRUE)`	Enabled	Enabled	
`error_flag_enable(FALSE)`	Enabled	Conditionally Disabled[1]	

Notes:
1) Error flags are disabled on DUT(s) which have an error latch set, in real-time, as the test pattern executes.

Note: the Maverick-I/-II software includes a second overload of `error_flag_enable()` with a `PEBoard` argument, used to provide a per-DUT control. The equivalent overload is not available using Magnum 1/2/2x because the Active DUTs Set (ADS) provides a more fully integrated level of control and should be used.

Usage

```
void error_flag_enable( BOOL State );
```

where:

`State` is either TRUE or FALSE. See Description.

Example

```
error_flag_enable( FALSE );
```

3.9.18 Over-programming Control Stimulus Selection

See Over-programming Controls and Parallel Test, Control of Branch on Error Flag.

Description

The `over_inhibit()` function is used to identify the programming stimulus which will be affected (inhibited) as described in Over-programming Controls and Parallel Test.

All three mode options noted below are deselected at test program initialization. This is equivalent to `over_inhibit(0,0,0)`; The system software does not otherwise modify this configuration.

Using Magnum 1/2/2x, the special hardware noted in Over-programming Controls and Parallel Test is replicated for every 8 pins and one DPS (i.e. a_1 through a_8, b_1 through b_8, etc). This restricts the minimum DUT size to 8 pins but also works when a DUT spans more than 8 pins. However, additional restrictions apply depending on the programming stimulus being used:

- If using VIHH as the programming stimulus, no additional restrictions exist.
- If using DPS as the programming stimulus, the over-programming inhibit hardware requires that the DPS Output Mode be configured/used in VPulse mode (`t_dps_vpulse`). The over-programming inhibit hardware inhibits the DPS programming stimulus by forcing the DPS to its primary output voltage, over-riding (inhibiting) the secondary output voltage (vpulse). In VPulse mode, both the A and B DPS outputs will drive the same voltage (unless one is disconnected).
- If using a normal PE driver signal as the programming stimulus, the over-programming inhibit hardware forces the drive state to VIH (presuming the active programming state to be active-low). This affects BOTH the A/B tester channels which share a given timing generator.

Usage

The function options below provide for several usage variations. In general, argument names that begin with `No` allow that argument to be effectively ignored by entering 0 as the argument value. Only the combinations documented below are supported.

```
// Specify 0 for all arguments. Restores default operation
void over_inhibit(int NoVihhPinList,
                  int NoVihPinList,
                  int NoDPSPinList );
```

```
// Specify a pin list for all arguments
void over_inhibit(PinList* pVihhPinList,
                  PinList* pVihPinList,
                  PinList* pDPSPinList );

// Specify 0 for argument 3 only
void over_inhibit(PinList* pVihhPinList,
                  PinList* pVihPinList,
                  int NoDPSPinList );

// Specify 0 for arguments 1 & 3
void over_inhibit(int NoVihhPinList,
                  PinList* pVihPinList,
                  int NoDPSPinList );

// Specify 0 for arguments 2 & 3
void over_inhibit(  PinList* pVihhPinList,
                    int NoVihPinList,
                    int NoDPSPinList );
```

where the following apply to all pin electronics boards in a test site:

pVihhPinList is a pin list identifying one or more pins on which VIHH will be inhibited. With VIHH inhibited, the PE driver drives the logic state last set from the test pattern. Specify 0 for this parameter when VIHH is not being used as programming stimulus.

pVihPinList is a pin list on which the drive-state will be inhibited. This causes the PE driver to drive-high (VIH) independent of the test pattern. Specify 0 for this parameter when PE pins are not used as programming stimulus. Note that this operation presumes that the pins are being used as active-low chip-select-like DUT inputs. The active-high state is not supported. Note that restrictions exist: see Description.

pDPSPinList is a pin list identifying which DPS will be inhibited. When a DPS is inhibited, it is forced to the primary voltage (not `dps_vpulse()`). Specify 0 for this parameter when not using a DPS as programming stimulus.

Note: using Magnum 1/2/2x, each DPS has two independently switchable output connections. This adds capabilities but also requires the user understand the potential effects when using the **pDPSPinList**option. See DUT Power Supply (DPS).

Example

The following example will cause VIHH on the pins in the `pl_cs` pin list to be inhibited:

```
over_inhibit( pl_cs, 0, 0 );
```

3.9.19 Binning

See Program Execution Control, Software.

The Magnum 1/2/2x software contains support for software binning. Note the following:

- All automated binning operations occur during Sequence & Binning Table execution. User code can manipulate or access binning structures at any time.
- Individual software bins are called Test Bins, and are created using the `TEST_BIN()` macro.
- During Sequence & Binning Table execution, if the test flow executes a Test Bin that Test Bin will automatically be incremented.
- Each Test Bin may include optional user body code which is executed any time that bin is encountered in a `BIN()`, `BINL()`, `STOP`, and `STOPL()` during Sequence & Binning Table execution.
- The `TEST_BIN_GROUP()` macro is used to create a software bin which accumulates the sum of one or more Test Bins. See Test Bin Groups.
- The available C-functions which interact with Test Bins are covered in the Test Bin Functions section.
- The available C-functions which interact with Test Bin Groups are covered in the Test Bin Group Functions section.
- Two built-in Test Bins, named `builtin_Pass` and `builtin_Fail`, are defined by the system software. One will be incremented when Sequence & Binning Table execution reaches a `STOP` *action* (not a `STOP` macro, see Sequence & Binning Test Flow). Conversely, neither of these built-in bins will be incremented when Sequence & Binning Table execution stops in a BIN() or BINL() statement.
- In Multi-DUT Test Programs, the `TEST_BIN()` and `TEST_BIN_GROUP()` macros actually create a separate bin for each DUT. Then, as the Sequence & Binning Table executes, the system software automatically increments the correct Test Bin and, if appropriate Test Bin Group, independently for each active DUT. In other words, when execution reaches a `BIN()`, `BINL()`, `STOP`, or `STOPL()` statement, the system software takes the correct action for each DUT currently in the Active DUTs Set (ADS).

3.9.19.1 Test Bins

See Binning, Program Execution Control.

Definition

The TEST_BIN( ) macro is used to create software test bin(s), for use in the traditional way, to count test results. In software, a test bin is represented as a TestBin data type.

When test bins are created, the BIN( ), BINL( ), STOP, and STOPL( ) macros are used in the Sequence & Binning Test Flow definition to specify which test bin to increment when execution reaches that statement. It is possible to increment any number of test bin(s) during a single execution of a Sequence & Binning Table.

Each test bin may optionally include body code i.e. user-written C-code. During Sequence & Binning Table execution, when a BIN( ) or BINL( ) statement is executed which references a test bin which has body code, that code is also executed. This enables user code to perform binning related actions which are not available in the built-in binning software. Iin Multi-DUT Test Programs, for a given BIN( ) or BINL( ) execution, the test bin body code is executed once, independent on the number of DUT(s) currently in the Active DUTs Set (ADS).

Test bin(s) are defined outside the scope of the SEQUENCE_TABLE( ) macro body code, however, the convention used by the Test Program Wizards puts this code in the file named *seq_and_bin.cpp*.

The EXTERN_TEST_BIN macro is used to create an external test bin declaration. This should only be necessary when code outside the *seq_and_bin.cpp* file needs access a test bin.

current_test_block( ) returns the name of the test bin if executed within the test bin's body code.

Usage

```
TEST_BIN( bin_name ) { optional body code }
EXTERN_TEST_BIN( bin_name )
```

where:

TEST_BIN is a Test System Macro used to create a test bin.

EXTERN_TEST_BIN is a Test System Macro used to create an external test bin declaration.

bin_name is the user-defined name of the test bin being created. Names must be valid C identifiers.

optional body code allows user-written C code to be executed each time the test bin is encounterd in a `BIN()`, `BINL()`, `STOP`, and `STOPL()` during Sequence & Binning Table execution.

Example

The following example creates 5 test bins and 2 Test Bin Groups:

```
TEST_BIN( OpensFail ){ output("msg"); }
TEST_BIN( ShortsFail ){}
TEST_BIN( SpeedFail ){}
TEST_BIN( PassBin1 ){}
TEST_BIN( PassBin2 ){}

TEST_BIN_GROUP( PassBins ) {
   BINS2( PassBin1, PassBin2 )
}

TEST_BIN_GROUP( FailBins ) {
   BINS3( OpensFail, ShortsFail, SpeedFail )
}
```

3.9.19.2 Test Bin Groups

See Binning, Program Execution Control.

Definition

The `TEST_BIN_GROUP` macro is used to create a *Test Bin Group*, essentially a software bin which accumulates the sum of one or more Test Bins.

During Sequence & Binning Table execution, a Test Bin Group is incremented when any of its member Test Bins is incremented. For example, it might be useful to add individual Test Bins named `opens_fail_bin`, `shorts_fail_bin`, `vcc_open_fail_bin`, and `vcc_open_fail_bin` to a Test Bin Group named `continuity_fails`. Then, SummaryTool can be used to display both the individual Test Bin counts and the sum of these bins i.e. the total count of the Test Bin Group.

Test bin group are created at a global level, normally in the same location as Test Bins. The convention used by the Test Program Wizards puts this code in the file named *seq_and_bin.cpp*.

Individual Test Bins are added to a Test Bin Group using the `BINS1()` through `BINS8()` macros, documented below. An existing Test Bin Group can be added to a new Test Bin Group being created using the `INCLUDE_TEST_BIN_GROUP` macro.

As noted above, Test Bin Groups are incremented automatically, as a side effect of incrementing one of its member Test Bins. The Test Bin Group Functions can be used to explicitly manipulate a Test Bin Group from user-written C-code.

The `EXTERN_TEST_BIN_GROUP` macro is used to make a forwarsd or external Test Bin Group declaration.

The *Units Passed* counter in FrontPanelTool will only be updated if the test program defines a Test Bin Group named *units_passed* (case insensitive). See page 2013 for an example.

Usage

```
TEST_BIN_GROUP( group_name ) {
   BINS1( bin_name1 )
   BINS2( bin_name2, bin_name3 )
   INCLUDE_TEST_BIN_GROUP( other_group )
}
EXTERN_TEST_BIN_GROUP( group_name )
```

The following macros are used to add up to 8 Test Bins to a Test Bin Group.

```
BINS1(group_name,tbin0)
BINS2(group_name,tbin1,tbin0)
BINS3(group_name,tbin2,tbin1,tbin0)
BINS4(group_name,tbin3,tbin2,tbin1,tbin0)
BINS5(group_name,tbin4,tbin3,tbin2,tbin1,tbin0)
BINS6(group_name,tbin5,tbin4,tbin3,tbin2,tbin1,tbin0)
BINS7(group_name,tbin6,tbin5,tbin4,tbin3,tbin2,tbin1,tbin0)
BINS8(group_name,tbin7,tbin6,tbin5,tbin4,tbin3,tbin2,tbin1,tbin0)
```

where:

`TEST_BIN_GROUP` is a Test System Macro used to create a software bin containing the sum its member Test Bins.

`group_name` is a user-defined name of the Test Bin Group being created or being externally declared. Names must be valid C identifiers.

bin_name1, **bin_name2**, **bin_name3**, and **tbin0** through **tbin8** are the names of existing Test Bins being added to the Test Bin Group.

INCLUDE_TEST_BIN_GROUP is a Test System Macro used to insert the contents of an existing Test Bin Group (**other_group** in the example above) into the Test Bin Group currently being defined.

BINS1 through **BINS8** are Test System Macros used to add from 1 to 8 Test Bins to a **TEST_BIN_GROUP**.

Example

See Example.

3.9.19.3 Test Bin Functions

See Binning, Program Execution Control.

The following functions may be used to access Test Bins:

- Test Bin set()/get() Functions - used to set the value of a specified Test Bin or get the value of a specified Test Bin.
- Test Bin increment()/decrement() Functions - used to increment or decrement the value of a specified Test Bin.
- Test Bin reset_all_bins() Function - used to reset the value of all defined Test Bins.
- Test Bin total_all_bins() Function - used to obtain the sum of all defined Test Bins for a specified DUT.
- Test Bin set_bin()/get_bin() Functions - used to set or get a final Test Bin.
- Test Bin invoke() Function - executes the user-written C code (body code) associated with a specified Test Bin.

Also see Test Bin Group Functions.

3.9.19.4 Test Bin set()/get() Functions

See Test Bin Functions, Test Bins, Binning, Program Execution Control.

Description

The `set()` function is used to set the value of a specified Test Bin.

The `get()` function is used to get the current value of a specified Test Bin.

Usage

The following function sets the value of a specified Test Bin for a specific DUT:

```
int set( TestBin* testBin, DutNum dut, int newValue );
```

The following function gets the current value of a specified Test Bin for a specific DUT:

```
int get( TestBin* testBin, DutNum dut );
```

where:

`testBin` identifies the target Test Bin.

`newValue` specifies the desired value.

`dut` is used in Multi-DUT Test Programs to identify the **`testBin`** for a specific DUT. The Active DUTs Set (ADS) has no effect on this function.

`set()` returns the previous value of the specified **`testBin`**.

`get()` returns the current value of the specified **`testBin`**.

Example

Given:

```
TEST_BIN( OpensFail ){}
```

The following functions set and get the value of the `OpensFail` Test Bin:

```
int prev_value = set( OpensFail, t_dut1, 0 );
int curr_value = get( OpensFail, t_dut1 );
```

3.9.19.5 Test Bin increment()/decrement() Functions

See Test Bin Functions, Test Bins, Binning, Program Execution Control.

Description

The `increment()` function is used to increment the value of a specified Test Bin.

The `decrement()` function is used to decrement the value of a specified Test Bin.

Note: normal Test Bin usage in the Sequence & Binning Table automatically increments a Test Bin.

Usage

The following function increments the value of a specified Test Bin:

```
int increment( TestBin* testBin );
```

The following function decrements the value of a specified Test Bin:

```
int decrement( TestBin* testBin );
```

where:

`testBin` identifies the target Test Bin. In Multi-DUT Test Programs, the specified **`testBin`** for all DUT(s) in the Active DUTs Set (ADS) are incremented or decremented.

`increment()` returns the value of the specified **`testBin`** after it is incremented. In Multi-DUT Test Programs, the return value is the sum of the specified **`testBin`** of all DUT(s) in the Active DUTs Set (ADS).

`decrement()` returns the value of the specified **`testBin`** after it is decremented. In Multi-DUT Test Programs, the return value is the sum of the specified **`testBin`** of all DUT(s) in the Active DUTs Set (ADS).

Example

Given:

```
TEST_BIN( OpensFail ){}
```

The following functions increment and decrement the `OpensFail` Test Bin:

```
int new_value = increment( OpensFail );
int new_value = decrement( OpensFail );
```

3.9.19.6 Test Bin reset_all_bins() Function

See Test Bin Functions, Test Bins, Binning, Program Execution Control.

Description

The `reset_all_bins()` function is used to reset the value of all defined Test Bins.

Usage

```
BOOL reset_all_bins( DutNum dut );
```

where:

`dut` is used in Multi-DUT Test Programs to identify a specific DUT. `reset_all_bins()` resets all defined Test Bins for this DUT. The Active DUTs Set (ADS) has no effect on this function.

`reset_all_bins()` returns FALSE if an invalid **`dut`** is specified otherwise TRUE is returned.

Example

```
ok = reset_all_bins( t_dut1 );
```

3.9.19.7 Test Bin total_all_bins() Function

See Test Bin Functions, Test Bins, Binning, Program Execution Control.

Description

The `total_all_bins()` function is used to obtain the sum of all defined Test Bins for a specified DUT.

`total_all_bins()` has no effect on Test Bin Groups, see `group_total()`.

Usage

```
int total_all_bins( DutNum dut );
```

where:

dut is used in Multi-DUT Test Programs to identify a specific DUT. `total_all_bins()` returns the sum of all defined Test Bins for this DUT. The Active DUTs Set (ADS) has no effect on this function.

`total_all_bins() returns the sum of all defined` Test Bins for the specified **dut**.

Example

```
int Dut1_total = total_all_bins( t_dut1 );
```

3.9.19.8 Test Bin set_bin()/get_bin() Functions

See Test Bin Functions, Test Bins, Binning, Program Execution Control.

Description

The `set_bin()` function is used, during Sequence & Binning Table execution, to set a final Test Bin. By default, the final Test Bin is that executed in the last test block executed, which defaults to either `builtin_Pass` or `builtin_Fail` based on the test block result.

The `get_bin()` function is used to get the final Test Bin, which is either the Test Bin executed in the last test block executed by the Sequence & Binning Table or the Test Bin set by `set_bin()`.

Usage

In Multi-DUT Test Programs, the following function sets the same bin for all DUTs in the Active DUTs Set (ADS):

```
BOOL set_bin( TestBin* bin );
```

The following function sets the bin for a specified DUT:

```
BOOL set_bin( TestBin* bin, DutNum dut );
```

The following function is used to get the currently set bin for a specified DUT:

```
TestBin* get_bin( DutNum dut );
```

where:

bin identifies the target Test Bin to be set.

dut is used in Multi-DUT Test Programs to identify a specific DUT. `get_bin()` returns the last bin executed for this DUT. The Active DUTs Set (ADS) has no effect on this function.

`set_bin()` returns TRUE if no errors occur, otherwise FALSE is returned.

`get_bin()` returns a pointer to the last Test Bin executed.

Example

The following code is targeted for execution in an After-testing Block:

```
output("Final Bin Results");
TestBin *tbin;
for( DutNum dut = t_dut1; dut <= max_dut(); ++dut ){
   tbin = get_bin( dut );
   output(" t_dut%d bin = %s", dut+1, resource_name( tbin ) );
}
```

3.9.19.9 Test Bin invoke() Function

See Test Bin Functions, Test Bins, Binning, Program Execution Control.

Description

The `invoke()` function executes the user-written C code (body code) associated with a specified Test Bin.

Note: the `invoke()` function has several overloads used with other data types, each documented separately.

Usage

```
int invoke( TestBin* obj );
```

where:

obj identifies the target Test Bin. In Multi-DUT Test Programs, the Active DUTs Set (ADS) has no effect on this function.

`invoke()` returns the final value assigned to the specified Test Bin. This value can also be obtained using `get()`.

Example

Given:

```
void myFunc(){
   if( get( OpensFail ) > 10 ){
      output(" Resetting OpensFail");
      set( OpensFail, 0 );
   }
}
TEST_BIN( OpensFail ){ myFunc() }
```

The following causes `myFunc()` to be executed returns the value of the `OpensFail` Test Bin:

```
int val = invoke( OpensFail );
```

3.9.19.10 Test Bin Group Functions

See Test Bins, Binning, Program Execution Control.

The following functions may be used to access Test Bins:

- Test Bin Group group_total() Function - used to get the current value of the specified Test Bin Group i.e. the sum of all Test Bins which are members of the specified Test Bin Group
- Test Bin Group group_bin() Function - used to identify the *index*'th Test Bin in a specified Test Bin Group.

Also see Test Bin Functions.

3.9.19.11 Test Bin Group group_reset() Function

See Test Bin Groups, Test Bin Group Functions, Binning, Program Execution Control.

Description

The `group_reset()` function is used to reset all Test Bins represented by a specified Test Bin Group. This also, in effect, resets the value of the specified Test Bin Group.

In Multi-DUT Test Programs, the version of `group_reset()` which takes a `DutNum` argument must be used. Using the version without a `DutNum` generates a warning similar to the following (the bin name `tbin1` will change, depending on the bins in the Test Bin Group being reset):

```
Warning: Calling "set(tbin1,0)" is probably a mistake.  Call the
version that takes a DutNum instead.
```

Usage

```
BOOL group_reset( TestBinGroup *obj );
BOOL group_reset( TestBinGroup *obj, DutNum dut );
```

where:

`obj` identifies the target Test Bin Group.

`dut` is used in Multi-DUT Test Programs to identify a specific DUT for which the bin(s) in the Test Bin Group are to be reset.

`group_reset()` returns FALSE if an error occurs otherwise TRUE is returned.

Example

```
TEST_BIN( tbin0 ){}
TEST_BIN( tbin1 ){}
TEST_BIN( tbin2 ){}
TEST_BIN( tbin3 ){}
TEST_BIN( tbin4 ){}
TEST_BIN_GROUP( myBinGroup ){
   BINS5( tbin0, tbin1, tbin2, tbin3, tbin4 )
}
group_reset( myBinGroup, t_dut1 );
```

3.9.19.12 Test Bin Group group_total() Function

See Test Bin Groups, Test Bin Group Functions, Binning, Program Execution Control.

Description

The `group_total()` function is used to get the current value of the specified Test Bin Group i.e. the sum of all Test Bins which are members of the specified Test Bin Group.

Usage

```
int group_total( TestBinGroup *obj, DutNum dut );
```

where:

`obj` identifies the target Test Bin Group.

`dut` is used in Multi-DUT Test Programs to identify a specific DUT.

`group_total()` returns the sum of the Test Bins represented by the specified Test Bin Group (**`obj`**). The Active DUTs Set (ADS) has no effect on this function.

Example

```
int total = group_total( PassBins, t_dut1 );
```

3.9.19.13 Test Bin Group group_bin() Function

See Test Bin Groups, Test Bin Group Functions, Binning, Program Execution Control.

Description

The `group_bin()` function is used to identify the *index*'th Test Bin in a specified Test Bin Group. This can be useful to iterate over the Test Bin members of a Test Bin Group, see Example.

Usage

```
TestBin* group_bin( TestBinGroup *obj, int index );
```

where:

`obj` identifies the target Test Bin Group.

`index` is the zero-based index into the Test Bin Group.

`group_bin()` returns a pointer to the **`index`**'th Test Bin in a specified Test Bin Group. The Active DUTs Set (ADS) has no effect on this function.

Example

```
TestBin *bin;
for ( int i = 0; bin = group_bin( PassBins, i ); ++i )
   output(" Bin => %s", resource_name( bin ) );
```

3.10 DC Functions

See Software.

This section covers topics which are common to using the DUT Power Supply (DPS), High Voltage Source/Measure Unit (HV), Per-pin Parametric Test Unit (PTU), and Parametric Measurement Unit (PMU). See DPS Functions, High Voltage Source/Measure Unit (HV) Functions, PTU Functions, and PMU Functions,

This section includes the following topics:

- Overview of hardware resources.
- Static DC Tests
- Dynamic DC Tests
- Types, Enums, etc.
- Parametric Settling Time
  - Built-in Settling Time
- `measure()`
- Measurement Average Count Functions
- Retrieving DC Test Results

3.10.1 Overview

See DC Functions.

Each Site Assembly Board contains the following DC test resources (see Site Assembly Board Block Diagram):

- Eight DUT Power Supply (DPS)s, programmed using the DPS Functions.
- Sixteen High Voltage Source/Measure Unit (HV)s, programmed using the High Voltage Source/Measure Unit (HV) Functions.
- Eight Parametric Measurement Unit (PMU)s, programmed using the PMU Functions.
- 128 Per-pin Parametric Test Unit (PTU)s, programmed using the PTU Functions.

- Eight DC Test and Measure Systems, used to perform both Go/NoGo tests and measurements using the DC resources above. Details are outlined in Static DC Tests and Dynamic DC Tests.

Key voltages from each of these DC resources are routed, one at a time via the DC Source Select MUX, to the DC Test and Measure System, which is used to both perform Go/NoGo tests and to make measurements. Both Static DC Tests and Dynamic DC Tests are supported, as Go/NoGo tests (faster) or measurements (more information).

The `measure()` function is used to switch between Go/NoGo test mode and measurement mode. Measured values are used by the system software to determine PASS/FAIL and may also be retrieved for evaluation by user code, see Retrieving DC Test Results.

Both Go/NoGo tests and measurements can be triggered from the site computer or by triggers from an executing test pattern.

3.10.2 Static DC Tests

See Overview, Dynamic DC Tests.

Static DC tests are executed using:

- DPS Static Current Test Functions; i.e. `test_supply()`
- HV Static Test Functions; i.e. `hv_test_supply()`
- PMU Static Test Functions; i.e. `partest()`
- PTU Static Test Functions; i.e. `ptu_partest()`.

Two static test options are possible:

- If measurements are disabled (see `measure()`) a static Go/NoGo test is performed. For all DC instruments except the PTUs, the site computer triggers the DC Comparators and Error Logic, one or more times, and reads the DC Error Flag to determine the test result. Using the PTU, the site computer triggers one or more PTU(s) internal DC comparators and reads the per-pin PTU error flag(s) to determine the test result.
- If measurements are enabled (see `measure()`) a static measurement is made. For all DC instruments, including the PTUss, the site computer triggers the DC A/D Converter, one or more times, and reads the measured value(s). The system software then compares the measured value(s) against the appropriate PASS/FAIL limits to determine the test result. When more than one measurement is made, the

average of all measurements is compared to the PASS/FAIL limits to determine the test result. Measured values can be retrieved by user code, see Retrieving DC Test Results.

In software release h1.1.23, support for measurement averaging was enhanced, which affects the Magnum 1 static DC parametric tests which support the `iacc` argument. The `test_supply()` and `ptu_partest()` operations were also enhanced at the same time. See Measurement Average Count Functions.

When performing Static DC Tests a user-specified Parametric Settling Time can be added to the Built-in Settling Time, both of which occur after the test stimulus is applied and before the Go/NoGo test or measurement is performed.

3.10.3 Dynamic DC Tests

See Overview, Dynamic DC Tests.

Dynamic DC tests include the execution of a functional test pattern.

Dynamic DC tests are executed using:

- DPS Dynamic Current Test Functions; i.e. `ac_test_supply()`
- HV Dynamic Test Functions; i.e. `hv_ac_test_supply()`
- PMU Dynamic Test Functions; i.e. `ac_partest()`
- PTU Dynamic Test Functions; i.e. `ptu_ac_partest()`

During a dynamic DC test a specified test pattern is executed, to functionally stimulate the DUT and to trigger the DC portion of the test.

Dynamic DC tests have two basic forms:

- Go/NoGo tests
- Measurement tests

The `measure()` function is used to enable or disable measurements vs. Go/NoGo tests.

A dynamic DC Go/NoGo test has two variations. The selection is made by including or excluding the `vcomp` argument to the test function being used:

- The executing test pattern triggers the DC Comparators and Error Logic. If, after pattern execution ends, if a DC Error Flag or a PE error latch is set the test fails (see Error Flag vs. Error Latch). Note that dynamic PTU Go/NoGo tests (i.e.

ptu_ac_partest() with measure() = FALSE) do not use the DC Error Flag. Instead, each PTU has its own set of DC comparators and local error flag which are used to determine PASS/FAIL. See Dynamic PTU Go/NoGo Test.

- The site computer enables the DC Comparators and Error Logic for the entire time the test pattern is executing. If, during this time, the parameter being tested fails either PASS/FAIL test limit the DC Error Flag is set and, if not reset (more below) the test fails. The test also fails if any PE error latch is set. This option is not usable in dynamic PTU Go/NoGo tests, see Dynamic PTU Go/NoGo Test..

In order to perform a dynamic DC measurement, the executing test pattern must trigger the DC A/D Converter (ADC); i.e. the computer can't trigger dynamic measurements. After pattern execution completes, the system software retrieves the last measurement value and compares it to the PASS/FAIL test limits to determine the test result. Measured values can also be retrieved by user code, see Retrieving DC Test Results.

As noted above, the test pattern executed during dynamic DC tests may trigger the DC portion of the test. This requires two things:

- Except for ptu_ac_partest(), the DC test function must specify vcomp as the optional the CompCond argument. See ac_partest(), hv_ac_test_supply(), ac_test_supply(). Excluding this argument or specifying no_vcomp means that the site computer enables the test hardware and any pattern triggers (next) have no effect on the test.
- The test pattern must include one or more VCOMP instructions. One DC trigger will be generated for each executed pattern instruction which contains the MAR VCOMP instruction (Memory Test Patterns) or VEC/RPT VCOMP, VAR VCOMP, or VPINFUNC VCOMP instructions (Logic Test Patterns).

Note: when using vcomp, if the pattern does not generate a DC trigger ac_partest(), ptu_ac_partest(), hv_test_supply() and ac_test_supply() will return an invalid DC test result. If measure = FALSE, the DC Comparators and Error Logic will not be triggered, which means the DC Error Flags will remain cleared = PASS.
If measure() = TRUE, no new measurements will occur, the measured value will be invalid (old, stale, etc.) and the test may return PASS or FAIL based on the invalid measurement value. The system software cannot check for this error; i.e. it is the user's responsibility to ensure that at least one trigger is issued by the pattern.

Any number of test pattern triggers can generated by a given test pattern, however, when measurements are enabled, additional considerations exist:

- The hardware can only store one measurement for each DC A/D Converter, thus only the measured value acquired by the last trigger is used to determine PASS/FAIL.
- The DC A/D Converter requires a minimum of 15 uS to complete a conversion. The user's test pattern is responsible for ensuring proper operation.

Note: using test pattern triggers with `measure()` = TRUE it is possible (easy?) to trigger the DC A/D Converter faster than it can correctly convert and store a measurement. Neither the hardware nor the system software can detect this condition; i.e. it is the user's responsibility.

As noted, whether using test pattern triggers or not, the state of the DC Error Flags will affect the test pattern's branch-on-error operations. The test pattern `MAR RESET` instruction (Memory Test Patterns) or `VEC/RPT RESET`, `VAR RESET`, or `VPINFUNC RESET` instruction (Logic Test Patterns) will clear the DC Error Flags on all Site Assembly Boards. If measurements are disabled, after pattern execution stops, the state of the DC error flags (and PE error flags, more below) determine tests overall PASS/FAIL result.

During the execution of the dynamic DC test, if any test pattern functional strobes fail (i.e. any digital PE error latches are set) the DC test will return FAIL, even if the DC portion of the test passed.

Note: all dynamic DC test functions execute a functional test pattern, thus for proper operation, it is necessary to correctly set up AC timing, DUT power, and PE drive, compare, and load voltages/currents prior to executing a dynamic DC test.

The functions which execute a dynamic DC test do not return until the functional test pattern execution ends. These functions all require that the test pattern termination option be specified, using the `PatStopCond` argument (see `funtest()` for reference). Most tests will use the `error` or `finish` options.

PASS/FAIL test limits are set using DPS Current Test Limit Functions, PMU Current Test Limit Functions and PMU Voltage Test Limit Functions, HV Current Test Limit Functions and HV Voltage PASS/FAIL Limit Functions, and PTU Current Test Limit Functions and PTU Voltage Test Limit Functions.

The information above applies to dynamic PTU measurement tests but not to dynamic PTU Go/NoGo tests, which operate differently, see Dynamic PTU Go/NoGo Test.

3.10.4 Types, Enums, etc.

See DC Functions.

Description

The following data types are used in support of various DC Functions:

Usage

The `Range` enumerated type is used to get or set a range option in DPS Current Test Limit Functions and PMU Force Current Functions. Note that the `norange` value is not normally used in test program but equates to auto-range operation:

```
enum Range { norange, range1, range2, range3,
             range4, range5, range6, range7, range8};
```

The `PassCond` enumerated type is used to specify test conditions for DC parametric tests. See DPS Static Current Test Functions, DPS Dynamic Current Test Functions, High Voltage Source/Measure Unit (HV) Functions, PTU Functions, PMU Static Test Functions and PMU Dynamic Test Functions:

The `PassCond` enumerated type is used to specify test conditions for DC parametric tests. See PMU Static Test Functions, PMU Dynamic Test Functions, HV Static Test Functions, HV Dynamic Test Functions, PTU Static Test Functions, DPS Static Current Test Functions, DPS Dynamic Current Test Functions:

```
enum PassCond {pass_pcl, pass_ncl, pass_nicl,
               pass_vg, pass_vl, pass_nivl };
```

The `CompCond` enumerated type is used in DPS Dynamic Current Test Functions and PMU Dynamic Test Functions to enable *test pattern triggers*, used to trigger the DC Comparators and Error Logic from the test pattern (see Dynamic DC Tests). Excluding the `vcomp` argument from the test function disables test pattern triggers.

```
enum CompCond { vcomp, no_vcomp };
```

Test pattern `MAR VCOMP` (Memory Test Patterns) and `VEC`/`RPT VCOMP`, `VAR VCOMP`, or `VPINFUNC VCOMP` (Memory Test Patterns) instructions are used to generate each trigger.

The `PartestOpt` enumerated type is used to select various options when executing PMU Static Test Functions,PMU Dynamic Test Functions, PTU Static Test Functions, HV Static Test Functions and HV Dynamic Test Functions:

```
enum PartestOpt { sequential,
                  parallel,
                  iacc,
                  no_iacc };
```

3.10.5 Parametric Settling Time

See DC Functions.

Description

The `partime()` function is used to add additional settling time to the Built-in Settling Time, applied during DC parametric tests.

During parametric test execution, a fixed amount of Built-in Settling Time is applied by the system software. Settling time is an intentional time delay which occurs after the specified parametric voltage or current is forced by the, DUT Power Supply (DPS), High Voltage Source/Measure Unit (HV), Per-pin Parametric Test Unit (PTU), Parametric Measurement Unit (PMU), and before the specified Go/NoGo test or measurement is performed. This settling time allows circuits internal to the tester to stabilize, decoupling capacitors to charge, etc. The `partime()` function can be used to add any additional settling time needed to compensate for DUT board circuitry, DUT requirements, etc.

In most applications, this settling time is only useful when performing static tests; i.e. `test_supply()`, `hv_test_supply()`, `ptu_partest()` or `partest()`. The additional delay *will* occur during dynamic tests but is usually not useful because it occurs after the hardware is connected to the specified pins, and after the force parameter is applied, but before the test pattern is executed and, since it is the test pattern which induces the DC parameter being tested to be dynamic, any settling time occuring before the pattern executes is not typically useful.

The system software sets the default `partime()` value to 0pS during the initial program load. The value is not otherwise modified by the system software; i.e. any additonal settling time programmed using `partime()` remains in effect until another value is programmed using `partime()`.

In sequential mode tests (the default for all DC tests except PTU tests) the additional delay repeats for each pin tested.

`partime()` is a global value; i.e. it can not be specified on a per-pin or per-DUT basis. As indicated above, any additonal settling time programmed using `partime()` remains in

effect until another value is programmed using `partime()`. This will impact the operation of ***all*** subsequent:

- DPS Static Current Test Functions; i.e. `test_supply()`
- HV Static Test Functions; i.e. `hv_test_supply()`
- PMU Static Test Functions; i.e. `partest()`
- PTU Static Test Functions; i.e. `ptu_partest()`
- DPS Dynamic Current Test Functions; i.e. `ac_test_supply()`
- HV Dynamic Test Functions; i.e. `hv_ac_test_supply()`
- PMU Dynamic Test Functions; i.e. `ac_partest()`

Note: when executing `ptu_partest()` in force-voltage mode a maximum additonal settling time of 5mS is enforced. In this situation, if the currently programmed `partime()` value is >5mS a warning is issued and `partime(5 MS)` is executed. This remains in effect until subsequently changed by user code.

To restore the minimum parametric settling time, execute `partime( 0 )`.

Usage

The function below is used to program a new `partime` value:

```
void partime( double value );
```

The following function returns the currently programmed `partime` value:

```
double partime();
```

where, **`value`** specifies the desired additional settling time. Units may be used (see Specifying Units). Legal values are:

Table 3.10.5.0-1 partime() Range & Resolution

Range	Resolution
0, or 0.1mS to 1000mS	0.1mS

The `partime()` getter function returns the currently programmed value.

Examples

```
partime( 5 MS ); // Delay = 5mS + Built-in Settling Time
partime( 0 MS ); // Reset to minimum value
double t = partime();
```

3.10.5.1 Built-in Settling Time

See DC Functions.

The system software causes a built-in Parametric Settling Time to occur during DC parametric tests.

The built-in settling time is designed to compensate for the internal DC circuitry to reach a stable operating condition before the actual test or measurement is made. The built-in settling time does not compensate for external requirements imposed by the DUT board design or DUT, etc., see Parametric Settling Time. The following DC instruments have a non-zero built-in settling time applied when performing DC parametric tests (details follow):

- Parametric Measurement Unit (PMU).
- DUT Power Supply (DPS).
- High Voltage Source/Measure Unit (HV).
- Per-pin Parametric Test Unit (PTU).

In force-voltage mode (DPS, HV, PTU or PMU) the specific amount of built-in settling time depends upon which current range is selected and, when applicable, which compensation capacitor is being used. In force voltage mode the built-in settling time is normally adequate to assure that tester specifications are met as long as a compensation capacitor consistent with the DUT capacitance has been selected to assure stability. See PMU Compensation Capacitors and DPS Compensation Capacitors for information about selecting compensation capacitors.

In force-current mode (PMU and PTU) the built-in settling time is fixed at 100uS for all ranges and all compensation capacitors. The user ‘s program should provide *additional* settling time based on the capacitance at the node being tested and the amount of current being forced. The user can either estimate the time using a mathematical model or using an oscilloscope to observe the voltage waveform at the DUT under both normal and fault conditions.

Built-in PMU Settling Time

The following table summarizes the built-in settling times for each PMU current range and compensation capacitor value. These apply when the PMU is in force-voltage mode. Refer to PMU Compensation Capacitors for more information about the function of the compensation capacitors.

Table 3.10.5.1-1 PMU Built-in Force Voltage Settling Time

PMU Current Range	Comp-cap 0	Comp-cap 1	Comp-cap 2
range1	601uS	18mS	180mS
range2	108uS	1.8mS	18mS
range3	91uS	201uS	1.8mS
range4	90uS	92uS	201uS
range5	90uS	90uS	92uS

Built-in DPS Settling Time

The following table summarizes the built-in settling times for each DPS current range and compensation capacitor value of the DPS. Refer to DPS Compensation Capacitors for more information about the function of the compensation capacitors:

Table 3.10.5.1-2 DPS Built-in Settling Time

DPS Range	Comp-cap 0	Comp-cap 1	Comp-cap 2
range1	9mS	90mS	990mS
range2	905uS	9mS	90mS
range3	127uS	905uS	9mS
range4	91uS	127uS	1.27mS
range5	90uS	91uS	91uS
range6	90uS	90uS	91uS

Built-in HV Settling Time

Since the High Voltage Source/Measure Unit (HV) has a single voltage range, has no compensation capacitor options, and has a low current output capability the built-in settling time is a constant 100uS.

Built-in PTU Settling Time

The following table summarizes the built-in settling times for each PTU current range. These apply when the PTU is in force-voltage mode:

Table 3.10.5.1-3 PTU Built-in Force Voltage Settling Time

PTU Current Range	Settling Time
range1	2mS
range2	1mS
range3	400uS
range4	200uS
range5	150uS
range6	
range7	
range8	

3.10.6 measure()

See DC Functions.

Description

The `measure()` *setter* function is used to switch the global DC test mode between Go/NoGo test mode and measurement mode.

During initial program load the measure mode is set to FALSE. The mode is not otherwise changed by the system software.

The effect of `measure()` is covered in some detail in Static DC Tests and Dynamic DC Tests.

The `measure()` *getter* function can be used to get the currently programmed measure mode state.

User code can retrieve a measured values, see Retrieving DC Test Results.

Note: measured values should be retrieved immediately after performing a test to prevent subsequent tests from over-writing useful information.

Usage

```
void measure( BOOL state );
BOOL measure();
```

where:

`state` is TRUE or FALSE to enable or disable measure mode.

The `measure()` *get* function returns the currently programmed measure mode state.

Example

```
measure( TRUE );       // Enable measure mode
BOOL state = measure();// Retrieve current measure state
```

3.10.7 Measurement Average Count Functions

Note: first available in software release h1.1.23.

Description

The Magnum 1/2/2x static DC parametric tests noted below support measurement averaging, enabled by including the `iacc` argument in the function parameter list.

The `iacc_count_set()` function may be used to specify the number of measurements (i.e. `iacc count`) made to determine each average. Prior to the availability of `iacc_count_set()` the count was fixed at 10.

The `iacc_count_get()` function can be used to retrieve the value last set using `iacc_count_set()`.

Note the following:

- A value set using `iacc_count_set()` affects the static DC parametric tests which accept the `PartestOpt iacc` argument. This are:
  - `partest()` testing signal pins, DPS pins and HV pins.
  - `hv_test_supply()`
  - `test_supply()`
  - `ptu_partest()`

  The value set using `iacc_count_set()` has no effect if the `iacc` argument is not included in the parametric test function's parameter list.
- During the initial program load `iacc_count_set()` is set = 10. The system software does not otherwise change this value.
- The value set using `iacc_count_set()` is a global value; i.e. it cannot be set per-pin, per-DUT, per-site, etc.
- The value set using `iacc_count_set()` is only used when `measure()` = TRUE.
- When Retrieving DC Test Results only the average value is returned, regardless of the value set using `iacc_count_set()`.

Usage

```
void iacc_count_set( int value );
int iacc_count_get();
```

where:

`value` specifies the desired `iacc` count value.

`iacc_count_get()` returns the currently programmed `iacc` count value.

Example

```
iacc_count_set( 13 );
int value = iacc_count_get();
```

3.10.8 Retrieving DC Test Results

See Static DC Tests, Dynamic DC Tests.

Description

The parametric test functions `partest()`, `ac_partest()`, `hv_test_supply()`, `hv_ac_test_supply()`, `ptu_partest()`, `test_supply()`, and `ac_test_supply()`return a PASS/FAIL result, indicating the overall outcome of the test. In Multi-DUT Test Programs, a test result is separately stored for each DUT in the Active DUTs Set (ADS), which can be retrieved using `result_get()` or `results_get()`.

Each of the parametric test functions noted above are able to test multiple pins, DPS, or HV with a single execution, thus additional test results may be available and which can be retrieved using the following functions:

- `Pin_meas()` - returns an array of measurement results. Used when testing signal pins using `partest()`, `ac_partest()`, `ptu_partest()` with `measure()` = TRUE. Values in the array are ordered based on the pin list tested. Values are returned for the first DUT in the Active DUTs Set (ADS).
- `Pin_pf()` - returns an array of PASS/FAIL values. Used when testing signal pins using `partest()`, `ac_partest()`, `ptu_partest()`. Values in the array are ordered based on the pin list tested. Values are returned for the first DUT in the Active DUTs Set (ADS).
- `Dps_meas()` - returns an array of measurement results. Used when testing DPS pins using `partest()`, `ac_partest()`, `test_supply()` and `ac_test_supply()`with `measure()` = TRUE. Values in the array are ordered based on the pin list tested. Values are returned for the first DUT in the Active DUTs Set (ADS).
- `Dps_pf()` - returns an array of PASS/FAIL values. Used when testing DPS pins using `partest()`, `ac_partest()`, `test_supply()` and `ac_test_supply()`. Values in the array are ordered based on the pin list tested. Values are returned for the first DUT in the Active DUTs Set (ADS).
- `Hv_meas()` - returns an array of measurement results. Used when testing HV pins using `partest()`, `ac_partest()`, `hv_test_supply()` and `hv_ac_test_supply()`with `measure()` = TRUE. Values in the array are ordered based on the pin list tested. Values are returned for the first DUT in the Active DUTs Set (ADS).

- `Hv_pf()` - returns an array of PASS/FAIL values. Used when testing HV pins using `partest()`, `ac_partest()`, `hv_test_supply()` and `hv_ac_test_supply()`. Values in the array are ordered based on the pin list tested. Values are returned for the first DUT in the Active DUTs Set (ADS).

These test results should be read immediately after the test function execution completes. This ensures subsequent tests do not over-write and cause a loss of information.

Note: when any of the parametric test functions terminate due to an error the previous PASS/FAIL and measurement values are invalid; i.e. old, stale, etc.

Measured current and voltage values are stored in base units (see Specifying Units). When accessing the PASS/FAIL results, 0 represents a FAIL and 1 represents a PASS.

Note: each Magnum 1/2/2x DPS has two independently switchable output connections.

Using `partest()`, when the `parallel` or `parallel_pmu` test options are specified, identical values are returned for each pin tested.

Usage

```
void Pin_meas( CArray<double, double>& data );
void Pin_pf( CArray<PFState, PFState>& data );
void Dps_meas( CArray<double, double>& data );
void Dps_pf( CArray<PFState, PFState>& data );
void Hv_meas( CArray<double, double>& data );
void Hv_pf( CArray<PFState, PFState>& data );
```

where:

`data` is used in two contexts:

- When retrieving PASS/FAIL test results, **`data`** is a previously declared `CArray`, defined to store `PFState` information. The array is automatically cleared and resized as necessary to store the appropriate test results. See Example.
- When retrieving measured values, **`data`** is a previously declared `CArray`, defined to store `double`. The array is automatically cleared and resized as necessary to store the appropriate test results. See Example.

Example

The following example may be used in Multi-DUT Test Programs to retrieve per-DUT measured values and PASS/FAIL results. There are two functions below:

- `cnvt_result()` : scale and outputs voltage and current values with the appropriate units. Only ever called by `pdatalog()`.
- `pdatalog()` : collect and output various test results. Calls `cnvt_result()`.

```
#include "math.h"  // Needed for fabs() below
enum e_log_type { LOG_I, LOG_V, LOG_T };

// Support function: scale and output result
CString cnvt_result( double value, e_log_type log_type ){
   CString cstr;
   switch( log_type ) {
   case LOG_V: // Convert to voltage
          if ( fabs( value ) >= ( 1 V ) ) {
             cstr.Format( "%8.3f v", value / ( 1 V ) );
          }
          else {
             cstr.Format( "%8.0f mv",value / ( 1 MV ) );
          }
     break;
   case LOG_I:  // Convert to current
          if( fabs( value ) >= ( 1 MA ) ) {
             cstr.Format( "%8.3f ma", value / ( 1 MA ) );
          }
          else if ( fabs( value ) >= ( 1 UA ) ) {
             cstr.Format( "%8.3f ua", value / ( 1 UA ) );
          }
          else {
             cstr.Format( "%8.0f na", value / ( 1 NA ) );
          }
          break;
   case LOG_T: break;
   default:cstr.Format( "############" );
   }
   return ( cstr );
}

void pdatalog( PinList* pPinList ) {
```

```
// Determine the mode of the parametric test last executed
e_log_type log_type;
switch( parametric_mode()) {
   case 0:log_type = LOG_I; break;
   case 2:log_type = LOG_I; break;
   case 1:log_type = LOG_V; break;
   default: output("Error: invalid mode: %d",parametric_mode());
}

// Output datalog header
output( "TestBlock = %-15s   TestNumber = %d",
          current_test_block(),
          test_number() );

if ( measure() == TRUE ) {
   output( "-----------------------------------------------" );
   output( "Tester   DUT        Pin      Pin      Measured" );
   output( " Pin     Pin        Name     Result   Value   " );
   output( "-----------------------------------------------" );
}
else {
   output( "-----------------------------------" );
   output( "Tester   DUT        Pin      Pin   " );
   output( " Pin     Pin        Name     Result" );
   output( "-----------------------------------" );
}

CArray<double,double> meas_results;
CArray<PFState,PFState> pf_results;

// Get/save the current Active DUTs Set
DutNumArray active_duts;
int count = active_duts_get(&active_duts);

// For each active DUT...
for( int dut = 0; dut < count; dut++) {
   DutNum dut_num = active_duts[ dut ];
   active_duts_enable( dut_num ); // Enable one DUT at a time

   // Check the pinlist for either dps pins or signal pins
   if( all_dps( pPinList )){
      Dps_pf( pf_results );
      if( measure()) Dps_meas( meas_results );
   }
   else if( all_hv( pPinList )){
```

```
      Hv_pf( pf_results );
      if( measure()) Hv_meas( meas_results );
    }
    else {
      Pin_pf( pf_results );
      if( measure()) Pin_meas( meas_results );
    }
    output("Dut-%d : %s", active_duts[dut]+1,
      result_get(active_duts[dut]) ? "PASS" : "FAIL");
    int size = pf_results.GetSize(); // Get num of results
    // Loop through all tested pins and output results
    for ( int pin = 0; pin < size; pin++) {
      DutPin * dpin;
      pin_info( pPinList, pin, &dpin);
      TesterPin tpin;
      pin_info( dpin, dut_num, &tpin );
      CString pin_name = resource_name( dpin );
      if( measure() )
        output("t_%-4d  %5d  %-14s   %4s  %s",
          ( tpin + 1 ),
          0,
          pin_name,
          pf_results[ pin ] ? "PASS" : "FAIL",
          cnvt_result( meas_results[ pin ], log_type ) );
      else
        output( "t_%-4d  %5d  %-14s   %4s",
          ( tpin + 1 ),
          0,
          pin_name,
          pf_results[ pin] ? "PASS" : "FAIL");
    }
  }
  // Restore Active_DUTs_Set
  active_duts_enable( active_duts );
}
```

3.11 DPS Functions

See DUT Power Supply (DPS).

This section includes the following topics:

- Overview
- Types, Enums, etc.
- DPS Connect/Disconnect Functions
- DPS Voltage Programming Functions
- DPS Output Mode
- DPS Current Test Limit Functions
- DPS Static Current Test Functions
- DPS Dynamic Current Test Functions
- VPulse Function
- DPS Current Sharing
- DPS Compensation Capacitors
- DPS 300mA/600mA DPS Option
  - `dps_ilimit_set(), dps_ilimit_get()`

Other related information includes:

- Static DC Tests and Dynamic DC Tests
- Parametric Settling Time and Built-in Settling Time
- `measure()`
- Retrieving DC Test Results

3.11.1 Overview

See DPS Functions, DUT Power Supply (DPS).

Using Magnum 1/2, there are 8 DUT Power Supply (DPS) per Site Assembly Board (i.e. one DPS per sixteen signal pins but each DPS has two independently switchable outputs). See DUT Power Supply (DPS).

By default, all DPS are disconnected from the DUT. The DPS Connect/Disconnect Functions must be used to connect all used DPS to the DUT.

Each DPS has two programmable voltage values. Both are set using the DPS Voltage Programming Functions. The secondary output voltage (VPulse) is dynamically enabled from an executing test pattern using the `PINFUNC VPULSE` pattern instruction (Memory Test Patterns) or `VEC/RPT VPULSE`, `VAR VPULSE` and `VPINFUNC VPULSE` instructions (Logic Test Patterns) but only DPS which have been enabled to respond to the VPULSE signal will be affected, see VPulse Function. Two output modes are available, more below. The VPULSE signal only affects DPS in `t_dps_vpulse` mode.

The DPS output current can be tested (i.e. a Go/NoGo test) or measured:

- The DPS uses the DC Test and Measure Systems to test or measure DPS current. The DC Comparators and Error Logic are used when performing Go/NoGo tests, the DC A/D Converter is used when measuring DPS current. The DC Test and Measure Systems is also used (shared) by the Parametric Measurement Unit (PMU), High Voltage Source/Measure Unit (HV) and Per-pin Parametric Test Unit (PTU).
- The DPS current test PASS/FAIL limits are programmed using DPS Current Test Limit Functions.
- Both static and dynamic DPS current tests are supported. See DPS Static Current Test Functions and DPS Dynamic Current Test Functions. Also review Static DC Tests and Dynamic DC Tests.
- The `measure()` function is used to switch between Go/NoGo tests vs. measurements.
- Review Parametric Settling Time and Built-in Settling Time; both apply to DPS current tests.
- Also review Retrieving DC Test Results.

In the Magnum 1/2 DPS hardware design, each DPS has 2 outputs, which be configured in two ways, using the DPS Output Mode:

- The *vpulse mode* allows an executing test pattern to cause the DPS(s) output to switch between the 2 voltage levels, typically for power supply noise immunity tests (VBump tests), or to enable special test modes. Additional steps are required to enable the Vpulse signal at each DPS, see VPulse Function.
- The *independent output mode* allows the two outputs (A&B) of each DPS to be programmed to a different voltage. In this mode the Vpulse signal has no effect. The *independent* output mode cannot be used when DPS Current Sharing.

- In *vpulse mode*, the combined current of both outputs cannot exceed the total available DPS output current specification. In *independent mode*, each output is limited to 1/2 the total available DPS output current specification.

To test devices requiring DPS current exceeding that available from one DPS, up to 10 DPS can be shared; i.e. electrically connected together on the DUT board. When this is done, the system software must be advised using the `CURRENT_SHARE()` macro. See DPS Current Sharing.

The DPS is a closed-loop feedback system, with stability that is affected by the circuit load. The DPS have several internal compensation capacitors, which are selected based on information supplied using the `dps_comp_cap()` function. See DPS Compensation Capacitors.

3.11.2 Types, Enums, etc.

See DPS Functions.

Description

The following enumerated types are used in support of various DPS Functions:

Usage

The `DpsOutputMode` enumerated type is used to control DPS Output Mode options:

```
enum DpsOutputMode {t_dps_vpulse, t_dps_independent };
```

The `DpsILimit` enumerated type is used to select whether the DPS 300mA/600mA DPS Option is enabled, using `dps_ilimit_set()`, and as a return value from `dps_ilimit_get()`:

```
enum DpsILimit { t_dps_default_ilimit, t_dps_high_ilimit };
```

3.11.3 DPS Connect/Disconnect Functions

See Overview, DUT Power Supply (DPS).

Description

The `dps_connect()` function is used to close the solid-state switch(es) connecting DPS(s) to DUT pins.

The `dps_connected()` function may be used to determine whether one specified pin is connected. This function was first available in software release h3.4.xx.

During initial program load, all DPS connect switches are opened. The system software does not otherwise change the configuration.

It is common to use `dps_connect()` in the Site Begin Block, to connect all used DPS for the duration of the test program.

Usage

The following function connects one DPS:

```
void dps_connect( DutPin *pDutPin );
```

The following function connects one or more DPS:

```
void dps_connect( PinList* pPinList );
```

The following function disconnects one DPS:

```
void dps_disconnect( DutPin *pDutPin );
```

The following function disconnects one or more DPS:

```
void dps_disconnect( PinList* pPinList );
```

The following function determines if the specified DPS is currently connected:

```
BOOL dps_connected( DutPin *pDutPin );
```

where:

`pDutPin` is used in two contexts:

- Using `dps_connect()` and `dps_disconnect()` identifies the target DPS to be manipulated. In Multi-DUT Test Programs, the DPSs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a DPS in the Pin Assignment Table.
- Using `dps_connected()`, identifies one DPS for which the current connect state is to be read.

pPinList specifies one or more target DPS(s). Only the DPS in the pin list, including any DPS connected by DPS Current Sharing, are affected. In Multi-DUT Test Programs, the DPSs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **pPinList** must only contains pins mapped to a DPS in the Pin Assignment Table.

Note: each DPS has two independently switchable output connections. This adds capabilities but also requires the user understand the potential effects on the functions documented in the DPS related sections.

`dps_connected()` returns TRUE if the specified DPS is currently connected, otherwise FALSE is returned.

Example

The example below causes all DPS specified in the pin list named VCC to be disconnected. Then some test program code is executed, and these same DPS are reconnected.

```
dps_disconnect( VCC );
// Other code executes here with all VCC DPS disconnected
dps_connect( VCC );
```

3.11.4 DPS Voltage Programming Functions

See Overview, DUT Power Supply (DPS).

Description

The `dps()` function is used to program the primary output voltage for one or more DUT Power Supply (DPS) or get the currently programmed value for one DPS. It is also used to program both outputs (A&B) independently for DPS(s) in `t_dps_independent`, more below.

The `dps_vpulse()` function is used to program the secondary output voltage (VPulse) for one or more DPS or get the currently programmed value for one specified DPS. See Overview and VPulse Function.

Each DPS has two programmable voltage values, set using the `dps()` and `dps_vpulse()` functions:

Table 3.11.4.0-1 DPS Voltage Range

Function	Range[3]	Max. Current	LSB
`dps()`	-15V to +15V -11.5V to +13.5V	±400mA[1] ±600mA[1,2]	5mV
`dps_vpulse()`			

Notes:
1) The current from each output (A&B) cannot exceed 1/2 the total available DPS output current specification.
2) The ±600mA maximum current is only available with the DPS 300mA/600mA DPS Option.
3) See DPS Operating Area.

Note the following:

- During the initial program load, all DPS output voltages are set to 0V and the primary voltage is selected.
- Executing `dps()` takes effect immediately (presuming a previously executed test pattern has not left the secondary (VPulse) voltage enabled).
- Once programmed, DPS voltage and/or current values remain in effect until:
  - Reprogrammed by user code.
  - Modified from a test pattern. See Controlling PE Levels from the Test Pattern.
  - Sequence & Binning Table execution stops, at which time the `builtin_after_testing_block` sets all DPS voltages to 0V.
  - Changing the DPS Output Mode may also cause the output voltage to change, more below.
- Except as noted below, the primary DPS voltage is not affected when setting the secondary voltage, and vice versa.
- The secondary DPS voltage (VPULSE) may be set greater than, less than or equal to the primary DPS voltage.
- The secondary voltage (VPulse) is enabled from an executing test pattern, using the `PINFUNC VPULSE` instruction (Memory Test Patterns) or `VEC/RPT VPULSE`, `VAR VPULSE`, or `VPINFUNC VPULSE` instructions (Logic Test Patterns). However, by default, all DPS are configured to not respond to the VPULSE signal from the test

pattern. The VPulse Function may be used to enable selected DPS to respond to this signal. Beginning in software release h3.4.xx, the DPS Vpulse Enable Functions may be used to enable selected DPS to respond to this signal. This is the only method which is supported in Magnum 2x but may also be used on Magnum 1/2.

As noted above, each DPS has two programmable voltage values, and can operate in two modes, selected using DPS Output Mode functions. The following diagram is used to explain DPS Output Mode operation:

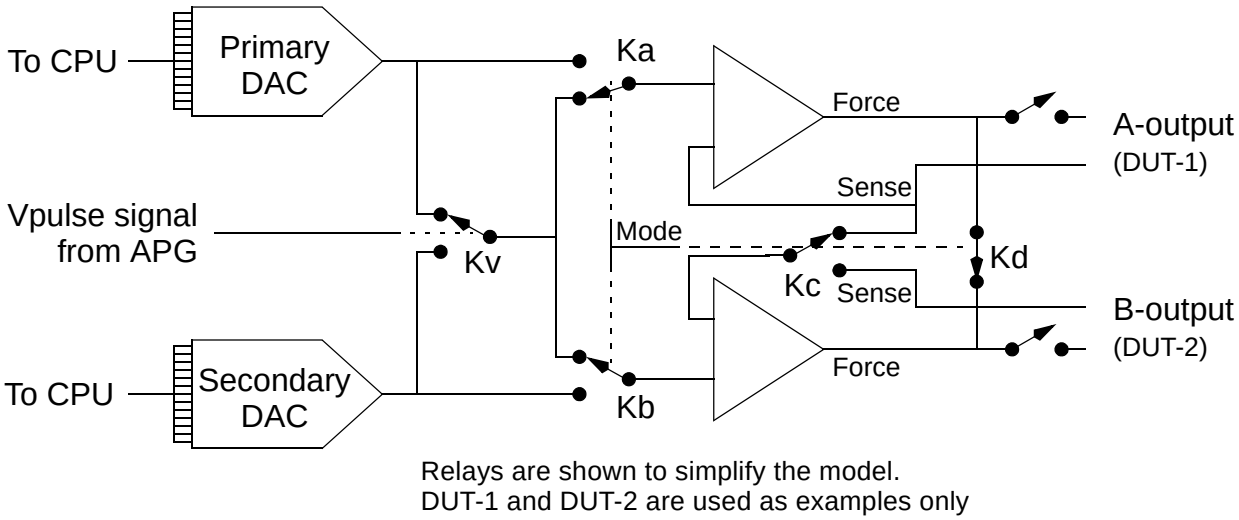


Figure-38: Simplified DPS Model

Note the following:

- The diagram is shown configured in `t_dps_vpulse` mode.
- Selecting `t_dps_vpulse` mode has the following effects:
  - The Ka, Kb, Kc and Kd relays are switched to the positions shown in the diagram.
  - The outputs of the A&B output amplifiers are connected together via Kd. Thus, in `t_dps_vpulse` mode, the A and B outputs will always be at the same voltage level, and the combined current of both A&B DPS outputs cannot exceed the total available DPS output current specification.
  - The output sense line connection for the B DPS output is disabled; i.e. the sense connection for both amplifiers is the A sense connection.

- The current sense circuitry (not shown) senses the total current of both output amplifiers, thus DPS current tests test/measure the combined current. User code can connect/disconnect the A&B outputs independently as needed to test/measure current for one output. See DPS Connect/Disconnect Functions.
- The Vpulse signal from the APG causes both DPS outputs to switch between the primary voltage, set using `dps()`, and the secondary voltage, set using `dps_vpulse()`. This allows an executing test pattern to cause both DPS outputs (A&B) to switch between the 2 voltage levels. Additional steps are required to completely enable the Vpulse signal at each DPS, see VPulse Function.

- Selecting `t_dps_independent` mode has the following effects:
  - The Ka, Kb, Kc and Kd relays switch to the opposite state.
  - The connection between the A&B output amplifiers is opened, allowing the A&B outputs to be set to different voltage levels (more below).
  - The sense line connection for the B DPS output is enabled.
  - Each DPS output is limited to 1/2 the total available DPS output current specification. The current sense circuitry (not shown) senses the current from one output (A or B) e.g. DPS current tests test/measure the A&B output current individually.
  - In `t_dps_independent` mode, the `dps()` function is used to program the output voltage value for both outputs. The Active DUTs Set (ADS) determines which output(s) of a given DPS are programmed by any given execution of `dps()`. Remember, a given DPS's A&B outputs are each independently mapped to a DUT in the Pin Assignment Table.
  - The effect of the Vpulse signal from the APG is disabled. Executing `dps_vpulse()` has no effect on DPS(s) in `t_dps_independent` mode.

The following text describes how the DPS Output Mode affects DPS operation. The diagram above is used for example:

- The system software keeps a record of 3 voltage values for each DPS:
  - DPS A primary voltage
  - DPS B primary voltage
  - VPULSE voltage (secondary voltage)
- During initial program load, the DPS Output Mode is set to `t_dps_vpulse`. The system software does not otherwise modify the mode.
- During initial program load, all three recorded voltage values are set = 0V. And, both DACs are set to output 0V.

- For DPS(s) in `t_dps_vpulse` mode, executing `dps()` sets only the primary DAC. The system software records only the A primary voltage value.
- For DPS(s) in `t_dps_vpulse` mode, executing `dps_vpulse()` sets only the secondary DAC value. However, the DPS output doesn't change until the test pattern outputs the Vpulse signal, selecting the secondary voltage. The system software records only the VPULSE voltage value.
- For DPS(s) in `t_dps_independent` mode, executing `dps()` can program either or both DACs, depending on which DUT(s) are in the Active DUTs Set (ADS) at the time `dps()` is executed. This works because each DPS output (A&B) is mapped to a specific DUT in the Pin Assignment Table. In hardware, the DPS's A output (DUT-1) will be at the A primary voltage (Primary DAC) and the B output (DUT-2) at the B primary voltage (SecondaryDAC). The system software records the changes appropriately.
- For DPS in `t_dps_independent` modes, executing `dps_vpulse()` updates the recorded VPULSE value only; i.e. the secondary DAC is not affected.
- When a given DPS is switched from `t_dps_vpulse` to `t_dps_independent`, the DPS's A output is connected to the primary DAC but the DAC is not reprogrammed, which is significant if a prior test pattern left the DPS connected to the secondary DAC. The DPS's B output is connected to the secondary DAC, which is set to the B primary voltage last programmed for the DPS. This operation may or may not cause the voltage at the two DPS outputs to change or to be different. The recorded values do not change.
- When a given DPS is switched from `t_dps_independent` back to `t_dps_vpulse` the secondary DAC is programmed to the recorded VPULSE value and the relays are switched as shown in the diagram. This connects the DAC last selected from the test pattern to both outputs; i.e. the DPS's A output will match the B output. The recorded values do not change.
- At the end of Sequence & Binning Table execution, the `builtin_after_testing_block` sets all DPS voltages to 0V. The DPS Output Mode is not modified.
- When executing `dps()` to get a value, the value is returned for the specified DPS connected to the first DUT in the Active DUTs Set (ADS). The value returned depends upon the DPS Output Mode of the DPS being read:
  - If the DPS read is in `t_dps_vpulse` mode, the value returned is the A primary voltage.

- If the DPS read is in `t_dps_independent` mode, the value returned depends upon whether the specified DPS is an A output or a B output. If connected to an A DPS output the value returned will be the A primary voltage. If connected to a B DPS output the value returned will be the B primary voltage.

- When executing `dps_vpulse()` to get a value, the value returned will always be the last VPULSE value programmed.

Usage

The following function programs the primary output voltage for the DPS(s) connected to the specified `DutPin` of all DUT(s) in the Active DUTs Set (ADS). The effect on each DPS depends upon the DPS Output Mode of each affected DPS, see Description:

```
void dps( double value, DutPin *pDutPin );
```

The following function programs the primary output voltage for the DPS(s) connected to the specified pin(s) of all DUT(s) in the Active DUTs Set (ADS). The effect on each DPS depends upon the DPS Output Mode of each affected DPS, see Description:

```
void dps( double value, PinList* pPinList );
```

The following function retrieves the currently programmed primary DPS voltage for one specified DPS. The value actually returned depends upon the DPS Output Mode of the DPS read. The value is returned for the first DUT in the Active DUTs Set (ADS). See Description:

```
double dps( DutPin *pDutPin );
```

The following function programs the secondary DPS output voltage (VPulse) for one DPS. In Multi-DUT Test Programs, only DPSs of DUTs currently in the Active DUTs Set (ADS) are affected.:

```
void dps_vpulse( double value, DutPin *pDutPin );
```

The following function programs the secondary output voltage (VPulse) for the DPS(s) connected to the specified `DutPin` of all DUT(s) in the Active DUTs Set (ADS). The effect on each DPS depends upon the DPS Output Mode of each affected DPS, see Description:

```
void dps_vpulse( double value, PinList* pPinList );
```

The following function retrieves the currently programmed secondary voltage for one specified DPS. The value actually returned depends upon the DPS Output Mode of the DPS read. The value is returned for the first DUT in the Active DUTs Set (ADS). See Description:

```
double dps_vpulse( DutPin *pDutPin );
```

where:

`value` specifies the desired voltage. Units may be used, see Specifying Units.

pDutPin is used in two contexts:

- In the setter functions, identifies one DPS to be programmed. In Multi-DUT Test Programs, the DPSs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a DPS in the Pin Assignment Table.
- In the getter functions, identifies one DPS to be read.

pPinList specifies which DPS(s) are to be programmed. In Multi-DUT Test Programs, the DPSs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **pPinList** must only contains pins mapped to a DPS in the Pin Assignment Table.

The `dps()` and `dps_vpulse()` getter functions return the currently programmed value for one DPS. In Multi-DUT Test Programs, the value is retrieved from the first DUT in the Active DUTs Set (ADS). Operation is affected by the DPS Output Mode, see Description.

Examples

The following example programs the primary voltage for each DPS included in the pin list named VCC to 5V. :

```
dps( 5 V, VCC );
```

The following example programs the secondary voltage for each DPS included in the pin list named VCC to 5.5V.

```
dps_vpulse( 5.5 V, VCC );
```

3.11.5 DPS Output Mode

See Overview, DUT Power Supply (DPS).

Description

The `dps_output_mode_set()` function is used to select the DPS Output Mode for one or more DPS.

The `dps_output_mode_get()` function is used to determine the currently selected DPS Output Mode for one DPS.

Each DPS has two programmable voltage values and two outputs. The two outputs (A&B) can operate in 2 modes, set using `dps_output_mode_set()`:

- In VPulse mode (t_dps_vpulse), both DPS outputs are set to the same voltage and may be switched to an alternate voltage, called VPULSE. The voltage value is programmed using dps_vpulse().The switch is controlled from the test pattern using the PINFUNC VPULSE pattern instruction (Memory Test Patterns) or the VPINFUNC VPULSE, VEC/RPT VPULSE or VAR VPULSE pattern instruction (Logic Test Patterns).
- In Independent mode (t_dps_independent), the two DPS outputs may be programmed to different voltages but the test pattern Vpulse control cannot be used.

Note the following:

- The DPS Output Mode affects the operation of dps() and dps_vpulse(), both the setter versions and getter versions. Detailed operation is described in DPS Voltage Programming Functions.
- The DPS Output Mode affects the total current available on each DPS output.
  - In t_dps_vpulse mode the maximum combined current from both DPS outputs is ±400mA.
  - In t_dps_independent mode the maximum current from each DPS output is ±200mA.
  - Using the DPS 300mA/600mA DPS Option, the maximum output current is increased to ±600mA (t_dps_vpulse) and ±300mA (t_dps_independent). See DPS Operating Area.
- The DPS Output Mode affects how DPS current sensing is done, which impacts how values are determined when using DPS Current Test Limit Functions.
- During initial program load, the DPS Output Mode is set to t_dps_vpulse. The system software does not otherwise modify the mode.
- The t_dps_independent mode cannot be used when DPS Current Sharing.
- The over-programming inhibit facilities only support t_dps_vpulse mode when using the DPS as the programming stimulus. See Control of Branch on Error Flag and Over-programming Control Stimulus Selection.

Usage

The following function sets the DPS Output Mode for the DPS connected to the specified DutPin:

```
void dps_output_mode_set( DutPin *pDutPin,
                DpsOutputMode mode DEFAULT_VALUE( t_dps_vpulse ) );
```

The following function sets the DPS Output Mode for the DPS connected to the specified pins:

```
void dps_output_mode_set( PinList* pPinList,
                DpsOutputMode mode DEFAULT_VALUE( t_dps_vpulse ) );
```

The following function returns the DPS Output Mode for the DPS connected to the specified DutPin:

```
DpsOutputMode dps_output_mode_get( DutPin *pDutPin );
```

where:

pDutPin is used in two contexts:

- In the setter function, identifies one DPS to be programmed. In Multi-DUT Test Programs, the DPSs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified DutPin must be mapped to a DPS in the Pin Assignment Table.
- In the getter function, identifies one DPS to be read.

mode is optional and, if specified, determines whether the the target DPS(s) are configured for VPULSE operation (default, t_dps_vpulse) or independent output (t_dps_independent).

pPinList specifies which DPS(s) are to be programmed. In Multi-DUT Test Programs, the DPSs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **pPinList** must only contains pins mapped to a DPS in the Pin Assignment Table.

dps_output_mode_get() returns the DPS Output Mode for one specified DPS. In Multi-DUT Test Programs, the value is retrieved from the first DUT in the Active DUTs Set (ADS).

Example

```
dps_output_mode_set( Vcc, t_dps_vpulse );
DpsOutputMode m = dps_output_mode_get( Vcc );
```

3.11.6 DPS Current Test Limit Functions

See Overview, DUT Power Supply (DPS).

Description

The `dps_current_high()` and `dps_current_low()` functions are used to specify PASS/FAIL test limits used during DPS current tests (i.e. `test_supply()` and `ac_test_supply()`). Optionally, a current sense range may be explicitly specified, more below.

Note:

- `dps_current_high()` and `dps_current_low()` values are set to 0 during initial program load, but are otherwise not modified by the system software.
- The DPS current sense circuitry operates differently depending on each DPS's output mode, set using DPS Output Mode functions:
  - In `t_dps_vpulse` mode, the combined current from both DPS outputs (A&B) is sensed and can be tested or measured.
  - In `t_dps_independent`, the output current from one output at a time can sensed and tested or measured. And, the maximum current available on each output is limited to 1/2 the total current specification.
- The `range` argument is optionally used to explicitly specify the current-sense range to be used during subsequent executions of DPS current test(s) (using DPS Static Current Test Functions and DPS Dynamic Current Test Functions). For a given DPS, the most recent execution of `dps_current_high()` or `dps_current_low()` determines whether an explicit range is used during subsequent DPS current tests. When an explicit `range` value is not specified, the system software will select the most accurate range (auto-range) based on the PASS/FAIL limit values used for each DPS. If the high and low compare limits fall into different ranges, the coarser (lower resolution) range is used.
- The PASS/FAIL limit value set using `dps_current_high()` and `dps_current_low()` may be manipulated (set/tweaked) from an executing test pattern, see Controlling PE Levels from the Test Pattern. However, this is only useful when the test pattern is being executed as part of a dynamic DPS current test (see DPS Dynamic Current Test Functions). Proper operation requires that the actual range used during the test is identical to the range specified in any `LEVELSET` pattern instruction which manipulates these levels. This is the user's responsibility.
- Some Magnum 1/2 systems include the DPS 300mA/600mA DPS Option. When this option is enabled the upper current sense limit and resolution change, see DPS Current Measurement Ranges.

Note: using Magnum 1/2 in vpulse output mode, each DPS has two independently switchable output connections (A/B). It is the total current of all connected DPS which are tested/measured by the DPS Static Current Test Functions and DPS Dynamic Current Test Functions.

Usage

The following functions program the PASS/FAIL current test limits for one DPS:

```
void dps_current_high( double value, DutPin *pDutPin );
void dps_current_low( double value, DutPin *pDutPin );
```

Note: the following two funtions were first available in software release h2.xx.yy:

```
void dps_current_high( double value,
                       DutPin *pDutPin,
                       Range range );
void dps_current_low( double value, DutPin *pDutPin, Range range );
```

The following functions program the PASS/FAIL current test limits for one or more DPS(s):

```
void dps_current_high( double value, PinList* pPinList );
void dps_current_low( double value, PinList* pPinList );
void dps_current_high( double value,
                       PinList* pPinList,
                       Range range );
void dps_current_low(  double value,
                       PinList* pPinList,
                       Range range );
```

The following function retrieves the currently programmed PASS/FAIL current test limit for one DPS:

```
double dps_current_high( DutPin *pDutPin );
double dps_current_low( DutPin *pDutPin );
```

where:

`value` specifies the desired high or low current value. Units may be used (see Specifying Units).

`pDutPin` is used in two contexts:

- In the setter function, identifies one DPS to be programmed. In Multi-DUT Test Programs, the DPSs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a DPS in the Pin Assignment Table.
- In the getter function, identifies one DPS to be read.

`pPinList` specifies which DPS(s) are to be programmed. In Multi-DUT Test Programs, the DPSs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **`pPinList`** must only contains pins mapped to a DPS in the Pin Assignment Table.

`range` is used to explicitly select a DPS current range. Legal values are of the `Range` enumerated type and are used as follows:

Table 3.11.6.0-1 DPS Current Measurement Ranges

Range	Current Range	LSB	Comments
`range1`	±4uA	2nA	
`range2`	±40uA	20nA	
`range3`	±400uA	200nA	
`range4`	±4mA	2uA	
`range5`	±40mA	20uA	
`range6`[1]	±400mA	200uA	
`range6`[1]	±600mA	2mA	±600mA only when using the DPS 300mA/600mA DPS Option.
	±4A		Usable only when DPS Current Sharing.
Note-1: `range6` selects the high current range. When the PASS/FAIL current test limits exceed ±400mA additional circuitry is used to scale the current sense signal accordingly.			

The `dps_current_high()` and `dps_current_low()` getter functions return the currently programmed high or low test limit for one specified DPS. In Multi-DUT Test Programs, the value is retrieved from the first DUT in the Active DUTs Set (ADS).

Example

The following example sets the high and low current test limits of all DPS(s) included in the pin list named VCC:

```
dps_current_high( 30 MA, VCC );
dps_current_low( 100 UA, VCC );
```

The following example sets the high current test limit to 500mA on all DPS(s) included in the pin list named VCC. The largest DPS measure range will be automatically selected. This example will only be valid if if `t_dps_high_ilimit` mode is set using `dps_ilimit_set()`, or if the the specified DPS(s) are DPS Current Sharing. If neither of these conditions are true, the setting is invalid and a fatal error will result.

```
dps_current_high( 500 MA, VCC );
```

3.11.7 DPS Static Current Test Functions

See Overview, DUT Power Supply (DPS).

Description

The `test_supply()` function is used to execute a static DPS current test. DPS Dynamic Current Test Functions are documented separately

See Static DC Tests.

The current of any number of DPS can be tested simultaneously, each with independent PASS/FAIL test limits. It is the pin list argument to `test_supply()` which identifies which DPS are tested.

In software release h1.1.23, two additional `test_supply()` overloads were added to support measurement averaging. Note the following:

- Measurement averaging is enabled by including the `iacc` argument.
- The number of measurements made to obtain the average is set using `iacc_count_set()`. Default = 10.
- The value set using `iacc_count_set()` is only used when `measure()` = TRUE.
- When averaging is enabled the measurement average is compared to the PASS/FAIL test limits, set using DPS Current Test Limit Functions, to determine whether `test_supply()` returns PASS or FAIL.

- When Retrieving DC Test Results only the average value is returned, regardless of the number of measurements made.

Prior to executing `test_supply()` the following parameters must be set up:

- DPS connections. See DPS Connect/Disconnect Functions.
- DPS output voltage. See DPS Voltage Programming Functions.
- PASS/FAIL current test limits (and sense range, optional). See DPS Current Test Limit Functions.
- The system software provides a Built-in Settling Time to DPS tests. The user may use the `partime()` function to add additional settling time. See Parametric Settling Time.
- Enable/disable measurements using `measure()`. Measured values can be retrieved by user code, see Retrieving DC Test Results.

Note: using Magnum 1/2 in vpulse mode, each DPS has two independently switchable output connections. It is the total current of all connected DPS which are tested/ measured by the DPS Static Current Test Functions and DPS Dynamic Current Test Functions.

It is also possible to connect the Parametric Measurement Unit (PMU) to DPS pins, temporarily disconnecting the DPS. See PMU: Testing DPS Pins.

Usage

```
PFState test_supply( PassCond pass_cond, PinList* pPinList );
PFState test_supply( PassCond pass_cond, DutPin DutPin* pDutPin );
```

Note: the following two overloads were first available in software release h1.1.23:

```
PFState test_supply(PassCond pass_cond,
                    DutPin *pDutPin,
                    PartestOpt opt );
PFState test_supply(PassCond pass_cond,
                    PinList* pPinList,
                    PartestOpt opt );
```

where:

pass_cond specifies how the PASS/FAIL test limits are used. Legal values are of the PassCond enumerated type, but only the values in the table below may be used with test_supply():

Table 3.11.7.0-1 DPS Current Test PASS/FAIL Limit Options

pass_cond	Operation
pass_pcl	PASS if the DPS current is greater than dps_current_high()
pass_ncl	PASS if the DPS current is less than dps_current_low()
pass_nicl	PASS if the DPS current is between dps_current_low() and dps_current_high()

pPinList identifies one or more DPS to be tested. In Multi-DUT Test Programs, the DPSs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **pPinList** must only contains pins mapped to a DPS in the Pin Assignment Table.

pDutPin identifies one DPS to be tested. In Multi-DUT Test Programs, the DPSs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified DutPin must be mapped to a DPS in the Pin Assignment Table.

opt is used to enable or disable measurement averaging, see Description. Legal values are of the PartestOpt enumerated type but only iacc and no_iacc are valid in this context:

Optional Arguments	Description
iacc	Enable measure averaging. Applies only when measure() = TRUE. See Description and Measurement Average Count Functions.
no_iacc	Default. Disable measure averaging.

test_supply() returns the overall PASS/FAIL result. A PASS result will only be returned if all DPS(s) tested PASS (i.e. no DC Error Flags are set). In Multi-DUT Test Programs, only DPSs of DUTs currently in the Active DUTs Set (ADS) affect the test result.

Examples

The following example tests the current of all DPS included in the pin list named VCC. For the test to pass, the current of each DPS tested must be between (pass_nicl) 800uA and 100mA.

```
dps( 5 V, VCC );
dps_current_high( 100 MA, VCC );
dps_current_low( 800 UA, VCC );
PFState result = test_supply( passnicl, VCC );
```

The example below illustrates setting the voltage on two DPS to output the same voltage (5V), setting the PASS/FAIL test limits on two DPS to different values, and performing a static DPS current test on both DPS simultaneously:

```
dps( 5 V, pl_Vcc_Vpp );
dps_current_high( 100 MA, VCC );
dps_current_low( 800 UA, VCC );
dps_current_high( 20 MA, VPP );
dps_current_low( 60 UA, VPP );
PFState result = test_supply( passnicl, pl_Vcc_Vpp );
```

3.11.8 DPS Dynamic Current Test Functions

See Overview, DUT Power Supply (DPS)

Description

The `ac_test_supply()` function is used to execute a dynamic DPS current test. DPS Static Current Test Functions are documented separately.

See Dynamic DC Tests.

The `ac_test_supply()` function executes a test pattern, and the associated pattern execution stop condition must be specified. Options are listed in Usage, and include terminating pattern execution if there is a functional failure or executing the pattern to completion, regardless of functional failures.

Note: in dynamic DC parametric tests, the test pattern can perform branch-on-error based on the state of the DC Error Flag in each DC Test and Measure System. The `MAR RESET`, or `CHIPS RESET` instructions (Memory Test Patterns) or `VEC/RPT RESET`, `VAR RESET` and `VPINFUNC RESET` instructions (Logic Test Patterns) will clear the DC Error FlagDC Error Flag, which can affect the overall PASS/FAIL result of the `ac_test_supply()`.

The ac_test_supply() function has two trigger modes, controlled using the comp_cond argument, which determines whether test pattern triggers are enabled. Four scenarios are possible:

- comp_cond = default (no_vcomp) and measure() = FALSE: the DC Comparators and Error Logic are used and enabled for the entire duration of the executing test pattern.
- comp_cond = vcomp and measure() = FALSE: the DC Comparators and Error Logic are used and triggered by the VCOMP token from the executing test pattern. If any one or more triggered samples fails the test returns FAIL.
- comp_cond = vcomp and measure() = TRUE: the DC A/D Converter is used and triggered by the VCOMP token from the executing test pattern. This causes a measurement to be made.
- comp_cond = default (no_vcomp) and measure() = TRUE. Not supported in DC tests. No warning is issued, testing continues, and ac_test_supply() operates as though measure() = FALSE. Any DC measurements which are retrieved are invalid.

When using comp_cond = vcomp, one trigger will be generated in each test pattern cycle which contains the MAR VCOMP instruction (Memory Test Patterns) or VEC/RPT VCOMP, VAR VCOMP or VPINFUNC VCOMP instructions (Logic Test Patterns). See Dynamic DC Tests.

Note: when using vcomp, if the pattern does not generate any triggers, ac_test_supply() will return invalid results. If measure = FALSE, the DC Comparators and Error Logic will never get triggered, which will result in a PASS condition. If measure() = TRUE, the DC A/D Converter will never be triggered, the retrieved measurement values will be invalid (old, stale, etc.) and the test may return PASS or FAIL based on the invalid measurement data. The system software cannot check for this error; i.e. it is the user's responsibility to ensure that at least one vcomp trigger is issued by the pattern.

---

ac_test_supply() will not return until test pattern execution terminates.

ac_test_supply() will also fail if any PE error latches are set (latched functional strobes failed).

Prior to executing ac_test_supply() the following parameters must be set up:

- DPS connections. See DPS Connect/Disconnect Functions.
- DPS output voltage. See DPS Voltage Programming Functions.

- PASS/FAIL current test limits (and sense range, optional). See DPS Current Test Limit Functions.
- The `partime()` function, used to add additional settling time to the Built-in Settling Time, is not normally useful in dynamic DC tests. This is because this delay will occur after programming the DC circuitry but before executing the functional test pattern; i.e. at a non-useful time. See Parametric Settling Time.
- Enable/disable measurements using `measure()`. Measured values can be retrieved by user code, see Retrieving DC Test Results.
- For dynamic tests, the test will also fail if any functional strobes fail. Thus proper digital PE levels and timing will affect test results, as does the test pattern executed.

The `ac_test_supply()` function returns an overall PASS/FAIL result, independent of how many DPS(s) are tested. In Multi-DUT Test Programs, only DPSs of DUTs currently in the Active DUTs Set (ADS) affect the test result. Also see Retrieving DC Test Results.

Note: using Magnum 1/2 in vpulse mode, each DPS has two independently switchable output connections. It is the total current of all connected DPS which are tested/ measured by the DPS Static Current Test Functions and DPS Dynamic Current Test Functions.

It is also possible to connect the Parametric Measurement Unit (PMU) to DPS pins, temporarily disconnecting theDPS. See PMU: Testing DPS Pins.

Usage

```
PFState ac_test_supply(
                    PassCond pass_cond,
                    PinList* pPinList,
                    Pattern *pPattern,
                    PatStopCond stop_cond,
                    CompCond comp_cond DEFAULT_VALUE( no_vcomp ));
```

Note: the following two overloads were first available in software release h2.xx.yy:

```
PFState ac_test_supply(
                    PassCond pass_cond,
                    DutPin *pDutPin,
                    Pattern *pPattern,
                    PatStopCond stop_cond );
```

```
PFState ac_test_supply(
                    PassCond pass_cond,
                    DutPin *pDutPin,
                    Pattern *pPattern,
                    PatStopCond stop_cond,
                    CompCond comp_cond );
```

where:

pass_cond specifies how the PASS/FAIL test limits are used. Legal values are of the PassCond enumerated type, but only the values in the table below may be used with ac_test_supply():

Table 3.11.8.0-1 DPS Current Test PASS/FAIL Limit Options

pass_cond	Operation
pass_pcl	PASS if the DPS current is greater than dps_current_high()
pass_ncl	PASS if the DPS current is less than dps_current_low()
pass_nicl	PASS if the DPS current is between dps_current_low() and dps_current_high()

pPinList identifies one or more DPS to be tested. In Multi-DUT Test Programs, the DPSs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **pPinList** must only contains pins mapped to a DPS in the Pin Assignment Table.

pPattern identifies the functional test pattern to be executed.

stop_cond specifies the pattern execution stop condition. Legal values are of the PatStopCond enumerated type, but only those in the table below are valid for this test:

Table 3.11.8.0-2 Pattern Execution Stop Condition Options

Stop Condition	Summary Description
finish	Execute pattern to completion, regardless of errors. When execution finishes, the PE error latches and DC Error Flags are examined. If an error was latched, the result of ac_test_supply() is FAIL, otherwise the result is PASS
error	Stop pattern execution on first functional or DC Error Flag error and sets the result of ac_test_supply() to FAIL. Note that the pattern generator may continue for one or more cycles past where the error occurred, depending on the cycle time and where in the cycle the error was detected.
fullec	Execute pattern to completion. Enable full ECR, row error catch, and column error catch to capture errors during pattern execution. This argument should be used when performing Redundancy Analysis (RA) or using BitmapTool.
LEC_only_errors	Enable full ECR, row error catch, and column error catch. Capture the first 2Meg (2^{21}-6) failing vectors. Intended for use when the ECR is used as an Logic Error Catch (LEC).
LEC_first_vectors	Enable full ECR, row error catch, and column error catch. Capture the first 2Meg (2^{21}-6) vectors executed. Ignores PASS/FAIL. Intended for use when the ECR is used as an Logic Error Catch (LEC).
LEC_last_vectors	Enable full ECR, row error catch, and column error catch. Capture the last 2Meg (2^{21}-6) vectors executed. Ignores PASS/FAIL. Intended for use when the ECR is used as an Logic Error Catch (LEC).
LEC_before_error	Enable full ECR, row error catch, and column error catch. Capture the first failing vector plus the previous 2Meg (2^{21}-6) vectors executed. Intended for use when the ECR is used as an Logic Error Catch (LEC).

Table 3.11.8.0-2 Pattern Execution Stop Condition Options *(Continued)*

Stop Condition	Summary Description
LEC_after_error	Enable full ECR, row error catch, and column error catch. Capture the first failing vector plus the next 2Meg (2^{21}-6) vectors executed. Intended for use when the ECR is used as an Logic Error Catch (LEC).
LEC_center_error	Enable full ECR, row error catch, and column error catch. Capture the first failing vector plus up to 512K vectors executed before the failure and up to 512K vectors executed after the failure. Intended for use when the ECR is used as an Logic Error Catch (LEC).
Note: in parallel test applications, the test pattern must be executed to completion, to ensure that DUT(s) which don't fail are completely tested. In other words, halting the pattern early (error) because one or more DUT(s) failed prevents DUT(s) which PASS from being completely tested. This is BAD.	

comp_cond is optional, and if used must be the reserved word vcomp. Including this argument enables test pattern triggers. See Description and Dynamic DC Tests.

pDutPin identifies one DPS to be tested. In Multi-DUT Test Programs, the DPSs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified DutPin must be mapped to a DPS in the Pin Assignment Table.

ac_test_supply() returns the overall PASS/FAIL result. A PASS result will only be returned if:

- All DPS(s) tested pass (no DC Error Flags are set).
- No functional error latches are set (no functional strobes failed)
- In Multi-DUT Test Programs, only DPSs of DUTs currently in the Active DUTs Set (ADS) affect the test result.

Examples

In the following example, the output voltage of all DPS in the pin list named VCC is set to 5V. The min/max DPS current test PASS/FAIL limits are set to 100mA and 1mA. The ac_test_supply() function executes the test pattern named myPat to completion and tests the that the DPS current is between the min/max limits. Since the vcomp option is not specified, the DPS current is monitored during the entire pattern execution. Once test pattern execution completes, the DC Error Flags and the PE error latches are read by the system software to determine the overall test result:

```
measure( FALSE );
dps( 5 V, VCC );
dps_current_high( 100 MA, VCC );
dps_current_low( 1 MA, VCC );
PFState result = ac_test_supply( pass_nicl, VCC, myPat, finish );
```

The following example uses the same test conditions and test pattern but uses the vcomp option. The DC Comparators and Error Logic are strobed in those test pattern cycles containing the pattern instructions noted in Description:

```
PFState result = ac_test_supply( pass_nicl,
                                 VCC,
                                 myPat,
                                 finish,
                                 vcomp);
```

The example below shows how to perform dynamic power supply tests simultaneously on two different power supplies, named VCC and VPP. Each supply uses unique voltages and currents limit values. Three pin lists are needed, VCC, VPP, and both_supplies.

The first three lines of code set the voltage and current test limits for VCC. The second three lines of code set different voltage and current test limits for VPP. The last line executes the test, simultaneously testing VCC and VPP. The myPat test pattern executes to completion (finish). The DPS current sense compare window will be open for the entire pattern execution (no vcomp). In this example, ac_test_supply() will return FAIL if VCC and/or VPP fail or if the myPat test pattern fails functionally:

```
dps( 5 V, VCC );
dps_current_high( 100 MA, VCC );
dps_current_low( 0.5 MA, VCC );
dps( 5.5 V, VPP );
dps_current_high( 20 MA, VPP );
dps_current_low( 100 UA, VPP );
int result = ac_test_supply(  pass_nicl,
                              both_supplies,
                              myPat,
                              finish);
```

3.11.9 DPS Vpulse Enable Functions

See Overview, DUT Power Supply (DPS).

Note: first available in software release h3.5.xx.

Description

The `dps_vpulse_enable()` function is used to manage which DUT Power Supply (DPS)s will respond to the VPULSE signal issued from an executing test pattern.

The `dps_vpulse_enabled()` function may be used to determine whether a given DPS will respond to the VPULSE signal issued from an executing test pattern.

Note the following:

- During the initial program load, all DPS are set to not respond to the test pattern VPULSE signal. The system software does not otherwise affect this operation.
- This function is supported on Magnum 1/2/2x but not Maverick-I/-II. The VPulse Function used on Maverick-I/-II to enable/disable VPulse operations is not supported on Magnum 2x but remains usable on the older system types.

Usage

```
void dps_vpulse_enable(  DutPin *pDutPin,
                         BOOL enable DEFAULT_VALUE( TRUE ));
void dps_vpulse_enable(  PinList* pPinList,
                         BOOL enable DEFAULT_VALUE( TRUE ));
BOOL dps_vpulse_enabled( DutPin *pDutPin );
```

where:

`pDutPin` is used in two contexts:

- Using `dps_vpulse_enable()`, identifies one DPS to be enabled or disabled. In Multi-DUT Test Programs, the DPSs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a DPS in the Pin Assignment Table.
- Using `dps_vpulse_enabled()`, identifies the DPS for which the enable state is to be returned.

`enable` is optional and, if used, specifies whether the specified DPS will respond to the test pattern VPULSE signal (TRUE) or not response (FALSE). Default = TRUE.

pPinList specifies which DPS(s) are to be enabled or disabled. In Multi-DUT Test Programs, the DPSs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **pPinList** must only contains pins mapped to a DPS in the Pin Assignment Table.

Example

```
dps_vpulse_enable( Vcc, TRUE );
dps_vpulse_enable( pl_allDPS, TRUE );
BOOL enabled = dps_vpulse_enabled( Vcc );
```

3.11.10 VPulse Function

See Overview, DUT Power Supply (DPS).

Description

The `vpulse()` function is used to enable the secondary DPS voltage, VPulse. Specific test pattern instructions are also required to actually cause the DPS to switch to this voltage (more below).

Each DPS has two programmable voltage values, both set using DPS Voltage Programming Functions. The secondary voltage, called VPulse, is typically used in the following applications:

- Power supply noise immunity tests.
- Memory V-bump tests.
- As a programming stimulus, when testing devices which use an alternate DPS voltage as the programming mechanism.

During the initial test program load, all DPS are switched to select the primary output voltage and the `vpulse()` option is disabled for all DPS.

To enable the DPS to switch to the VPulse level, under control of the executing test pattern, requires the following:

- Enable the VPULSE signal to specified DPS using the `vpulse()` function. If this step is skipped, the next step has no effect.
- Enable the VPULSE signal from the test pattern using the `PINFUNC VPULSE` instruction (Memory Test Patterns) or `VEC/RPT VPULSE`, `VAR VPULSE`, or `VPINFUNC VPULSE` instructions (Logic Test Patterns).

- Ensure the DPS(s) to be used are in `t_dps_vpulse` mode (see DPS Output Mode). VPULSE operations are disabled on DPS in `t_dps_independent`.

In hardware, there is only one VPULSE enable signal, distributed and shared by all DPS, thus all DPS which are enabled will switch to the secondary voltage when the test pattern enables the VPULSE signal. The signal changes state when the pattern instruction containing VPULSE executes, however additional time is required for this signal to reach the DPS(s) and for those DPS to respond and slew to the secondary voltage value. The user must design their test patterns to achive the desired effect at the DUT. This may require some experimentation and tuning of the test pattern.

When a DUT Power Supply (DPS)is in `t_dps_vpulse` mode, the secondary voltage affects both DPS outputs (A&B) identically. Conversely, the secondary voltage value has no effect on any DPS configured in `t_dps_independent`. See DPS Output Mode.

Both DPS voltage levels can be manipulated from the test pattern. See Controlling PE Levels from the Test Pattern.

Usage

The following function enables the specified DPS to react to the VPULSE signal from the test pattern:

```
void vpulse( DutPin *pDutPin );
```

The following function enables the one or more DPS to react to the VPULSE signal from the test pattern:

```
void vpulse( PinList* pPinList );
```

The following function disables the VPULSE signal on all DPS:

```
void vpulse();
```

where:

`pDutPin` identifies one DPS. In Multi-DUT Test Programs, the DPS(s) of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a DPS in the Pin Assignment Table.

`pPinList` specifies which DPS(s) are to be programmed. In Multi-DUT Test Programs, the DPS(s) of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **`pPinList`** must only contains pins mapped to a DPS in the Pin Assignment Table.

Example

The following example enables the secondary DPS output voltage option of all DPS(s) in the pin list named VPP. This does not, by itself, select the secondary output voltage (see Description). In Multi-DUT Test Programs, only DPSs of DUTs currently in the Active DUTs Set (ADS) are affected:

```
vpulse( VPP );
```

The following example disables the secondary DPS output voltage option of all DPS(s) and switches all DPS to output the primary voltage:

```
vpulse();
```

3.11.11 DPS Current Sharing

See Overview, DUT Power Supply (DPS).

Description

The `CURRENT_SHARE()` and `SHARE()` macros are used to configure DPS current sharing, by creating and defining a `CurrentShare` resource (see Resource Types). When the test program defines multiple `CurrentShare` resources, the `USE_CURRENT_SHARE()` macro should be used, within the body code of the `CONFIGURATION()` macro, to select one.

To test devices requiring DPS current values exceeding that available from one DPS, up to 10 DPS can be connected to the DUT in parallel. When this is done, the system software must be advised, using the `CURRENT_SHARE` macro, about which DPS are shared. When 2 or more DPS are shared the set is called a *current share group*. By default, no DPS are in current share mode.

The following rules apply when using current sharing:

- Current sharing begins by electrically wiring multiple DPS together at the DUT.
- Each Magnum 1/2 Site Assembly Board contains 8 DPS, in two groups (A/B) of four. Each DPS has two outputs, also identified using A and B. A given DPS output is formally identified using `HDTesterPin` values i.e. `a_dps1a`, `b_dps1b`, `a_dps3b`, etc. The prefix indicates the group, the suffix identifies one output of a given DPS.

- When current sharing, the force outputs of each shared DPS must be electrically connected to the same point, typically at the DUT. Shared DPS can only be configured in `t_dps_vpulse` mode, see DPS Output Mode and only the DPS A output can be specified in the Pin Assignment Table (more below)
- An A-prefix DPS (i.e. `a_dps1a`, a_dps2b, etc.) can only be shared with other A-prefix DPSs. Similarly, a B-prefix DPSs (i.e. `b_dps1a`, b_dps2b, etc.) can only be shared with other B-prefix DPSs. It is not possible to share an A-prefix DPS with a B-prefix DPS, etc.
- The underlying DPS hardware design only supports current sharing between sequentially numbered DPS. For example, it is legal to current share `a_dps1a` with a_dps2a and a_dps3a, but not legal to share `a_dps1a` with a_dps3a without also including a_dps2a.
- It is not necessary to begin current sharing with any specific DPS. For example, it is legal to current share a_dps2a with a_dps3a, without also including `a_dps1a`.
- It is legal to current share across Site Assembly Board boundaries. For example, it is legal to current share a_dps8a with a_dps9a. This does require Sites-per-Controller be > 1 in the Pin Assignment Table.
- A given DPS can only be included in one current share group.
- Only the sense line of the lowest numbered DPS in each DPS share group is used. The sense line(s) of the other shared DPS are not used and should not be connected (to anything).
- The Magnum 1/2 hardware natively supports sharing up to 10 DPS, to source and test/measure up to 4A maximum. This requires Sites-per-Controller = 3 (4 DPS from board-1, 4 from board-2, 2 from board-3).
- When using shared DPS, an additional current test/measure range becomes available, allowing `dps_current_high()` and `dps_current_low()` values up to ±4A.
- The CURRENT_SHARE macro is used in the test program to define a named current share group. The SHARE macro is used, in the body of the CURRENT_SHARE macro, to specify which DPS pin(s) are connected to shared DPS and the number of shared DPS connected to each pin. In the Pin Assignment Table, for each of these power pins, the DPS resource assigned must be the lowest numbered shared DPS, as described above. The other shared DPS will not (must not) be specified anywhere the Pin Assignment Table.
- In Multi-DUT Test Programs, when a given DUT power pin is identified in a current share group, the specified number of shared DPSs must be available for each DUT. For example, given the following Pin Assignment Table entry...

```
ASSIGN_2DUT( Vcc, a_dps1a, b_dps1a )
```

... the following DPS current share definition will consume 6 DPS:

```
CURRENT_SHARE( only_1_group ) {
   SHARE( Vcc, 3 )
}
```

In this example, a_dps1a, a_dps2a, and a_dps3a will supply up to 1.2A to DUT-1's Vcc pin, and b_dps1a, b_dps2a, and b_dps3a to supply up to 1.2A to DUT-2's Vcc pin. Notice that in Multi-DUT Test Programs, using DPS current sharing will quickly reduce the number of DUTs which can be used. For example, when each DUT has a single power input requiring 400mA-800mA, only 4 DUTs per-Site Assembly Board can be tested (using 6 of the 8 DPS on the board). All other DPS current sharing applications will typically limit the number of DUTs to 2 per-Site Assembly Board or require Sites-per-Controller > 1 i.e. multiple Site Assembly Boards.

- As indicated above, both outputs (A and B) of each shared DPS must be electrically connected to the same point, typically very close to the DUT. Because of this, it is not legal, in the Pin Assignment Table, to assign the B output of any shared DPS to a DUT pin i.e. it is not legal, for example, to assign a_dps1b to a DUT pin if a_dps1a is shared.
- The CURRENT_SHARE macro can be used multiple times, to define more than one current share group, but only one can be used. A named current share group is selected using the USE_CURRENT_SHARE() macro, in a CONFIGURATION() block.
- When using current shared DPS, to provide the best current test resolution possible, the system software may temporarily disconnect one or more DPS from the DUT. The software considers the programmed dps_current_high() and dps_current_low() values, plus the programmed PASS condition (pass_nicl, pass_ncl, etc.) to determine the number of current share supplies that will actually be used to perform the test. When the test limits are less than ±400mA only the master DPS (lowest numbered DPS of a share group) will be used. As the test limits increase past these values, additional DPS will be used, and the corresponding current sense resolution decreases. If *n* DPSs are used for the test, the resolution of the measured value will be *n* times the resolution of a single DPS.

Usage

```
CURRENT_SHARE( share_group_name ) {
   SHARE( dps_pin, count )
   . . .
   SHARE( dps_pin, count )
}
```

where:

CURRENT_SHARE is a Test System Macro used to denote the start of the current share group definition.

name is the name of the current share group being created. Each current share group can define one or more shared DPS configurations, each with a unique **dps_pin**.

SHARE is a Test System Macro used to add an entry to the current share group.

dps_pin identifies the DUT power pin connected to the master DPS of a group of shared DPS. **dps_pin** is a DutPin (VCC, VDD, etc.) from the Pin Assignment Table. See Description.

count is the total number of shared DPS connected to **dps_pin**; i.e. the number of DPS electrically wired together to each **dps_pin**.

Examples

Example 1:

In the following example, the current share group being created is named `ProdPower`. In this group, two sets of shared DPS are specified. VCC will have 3 shared DPS physically wired together at the DUT's VCC pin, with the lowest numbered DPS assigned to the VCC pin in the Pin Assignment Table. VCCO will have 2 shared DPS physically wired together at the DUT's VCCO pin, with the lowest numbered DPS assigned to the VCCO pin in the Pin Assignment Table.

```
CURRENT_SHARE( ProdPower ) {
   SHARE( VCC, 3 )
   SHARE( VCCO, 2 )
}
```

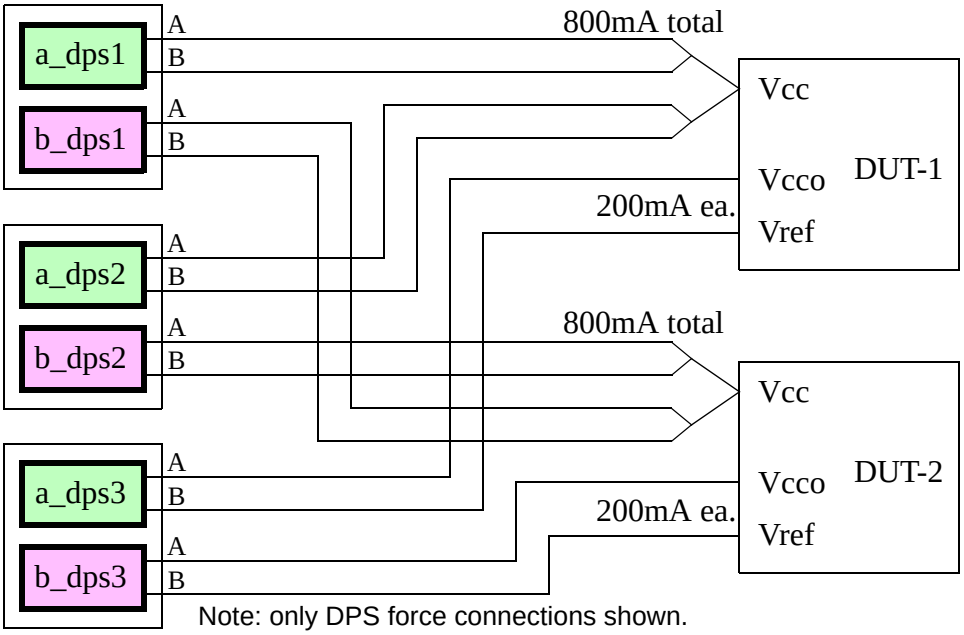
Assuming that the DUT VCC pin is assigned to a_dps1a in the Pin Assignment Table and with a SHARE count of 3, then a_dps1a, a_dps2a, and a_dps3a must be wired to the VCC pin on the DUT board. Similarly, if the DUT VCCO pin is assigned to b_dps1a, then b_dps1a and b_dps2a must be wired to the VCCO pin on the DUT board.

Example 2:

The following example tests two DUTs, each with 3 independent power inputs. One input, Vcc, requires up to 800mA. The other two inputs, Vcco and Vref, only require up to 200mA

each:

One Site Assembly Board (partial)



The following (partial) code snippets indicate how this would be defined in the test program:

```
// Pin Assignment Table:
ASSIGN_2DUT( Vcc, a_dps1a, b_dps1a )
ASSIGN_2DUT( Vcco, a_dps3a, b_dps3a )
ASSIGN_2DUT( Vref, a_dps3b, b_dps3b )

CURRENT_SHARE( only_1_group ) {
   SHARE( Vcc, 2 )
}
```

Note the following:

- This example represents a Multi-DUT Test Program, testing 2 DUTs in parallel.
- The `ASSIGN` statements are part of a partial Pin Assignment Table definition, included to restate the rules regarding how DPS hardware is allocated when DPS current sharing is used in a Multi-DUT Test Program.
- Vcc is specified to share 2 DPS (per DUT). The master DPS for DUT-1 is `a_dps1a` (per the Pin Assignment Table), thus `a_dps1b`, `a_dps2a` and `a_dps2b` cannot appear in the Pin Assignment Table, and DPS `a_dps1b`, `a_dps2a` and `a_dps2b` are automatically used as the slave shared DPS for Vcc on DUT-1.

- The master DPS for DUT-2 is `b_dps1a`, thus `b_dps1b`, `b_dps2a`, and `b_dps2b` cannot appear in the Pin Assignment Table, and `b_dps1b`, `b_dps2a`, and `b_dps2b` are automatically used as the slave shared DPS for Vcc on DUT-2.

3.11.12 DPS Compensation Capacitors

See DPS Functions, DUT Power Supply (DPS) .

Description

The `dps_comp_cap()` function is used to select a compensation capacitor for one or more DPS or to retrieve the currently selected DPS compensation capacitor for one DPS.

During initial program load the minimum compensation value is selected. The system software does not otherwise change the selection.

The DPS is basically a closed-loop, unity-gain, power operational amplifier (OpAmp), with a high power output driver. OpAmps configured in a closed loop configuration, like the DPS, use negative feedback, which consists of connecting the OpAmp output, through a network, to the OpAmp's inverting input. At low frequencies signals at the inverting input are reproduced at the output with a 180 degree phase-shift, due to the inherent inverting relationship between the inverting input and the output. Thus, as the sensed voltage falls the OpAmp output voltage will increase to compensate, and vice versa.

However, because the propagation delay through the amplifier is not zero, at higher frequencies there is some phase shift between the sense voltage and the OpAmp (DPS) output. At high enough frequencies the phase shift due to the propagation delay adds to the 180 degree phase-shift inherent in the inverting input and will approach 360 degrees. In other words, the signal can become in-phase, resulting in positive feedback. If this is allowed to occur the DPS output will oscillate. Even as the phase shift approaches 360 degrees the DPS output may over-shoot and exhibit ringing.

The purpose of the DPS compensation capacitors is to modify the gain and phase shift of the DPS (OpAmp) so that, for a particular application, the DPS output does not oscillate or overshoot/ring. This is required when the DPS must drive a load that includes decoupling capacitance.

The DPS hardware has three internal compensation capacitors, used to adjust the feedback loop to compensate for different capacitive loads; a single fixed compensation scheme can not effectively balance the trade-off between stability and settling time. Compensation of the

DPS feedback loop is accomplished by selecting a compensation capacitor of appropriate size, see table below.

In general, the lowest value compensation option that provides DPS stability should be used. Using a larger than needed value will slow down measurements, particularly on the lower current sense ranges, and require more DPS slew/settling time when voltage levels are changed.

Note: when using the PMU to test DPS pins in a force current test, the PMU connection to the DPS pins is initially made in voltage-force mode, then the PMU is switched to force current mode to complete the test. Even though the PMU compensation capacitors are disconnected automatically when the PMU is in force-current mode, the initial connection is being made in force-voltage mode, thus the appropriate PMU compensation capacitor selection should be made, see PMU Compensation Capacitors.

Usage

Note: the `dps_comp_cap()` function (only) is used in all Maverick system programs and for Magnum 1 and Magnum 2 programs. Magnum 2x programs only use the `dps_comp_cap_set()` and `dps_comp_cap_get()` functions. Executing `dps_comp_cap()` on Magnum 2x will generate a warning and the compensation capacitor selection will not be changed.

The following function sets the DPS compensation capacitor value of all DPS:

```
void dps_comp_cap( int state );
```

The following function sets the DPS compensation capacitor value for one DPS:

```
void dps_comp_cap( int state, DutPin *pDutPin );
```

The following function sets the DPS compensation capacitor value for one or more DPS(s):

```
void dps_comp_cap( int state, PinList* pPinList );
```

The following function returns the currently programmed compensation capacitor value for one DPS:

```
int dps_comp_cap( DutPin *pDutPin );
```

The following function sets the DPS compensation capacitor value of all DPS:

```
void dps_comp_cap( int state );
```

The following function sets the DPS compensation capacitor value for one DPS:

```
void dps_comp_cap( int state, DutPin *pDutPin );
```

The following function sets the DPS compensation capacitor value for one or more DPS(s):

```
void dps_comp_cap( int state, PinList* pPinList );
```

The following function returns the currently programmed compensation capacitor value for one DPS:

```
int dps_comp_cap( DutPin *pDutPin );
```

where:

`state` specifies the desired compensation capacitor value:

Table 3.11.12.0-1 DPS Compensation Capacitor Selection

State	Purpose
0	For bypass caps less than 0.1uF (Default)
1	For bypass caps greater than 0.1uF but less than 1.0uF
2	For bypass caps greater than 1.0uF but less than 10uF.

`pDutPin` is used in two contexts:

- In the setter function, identifies one DPS to be programmed. In Multi-DUT Test Programs, the DPSs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a DPS in the Pin Assignment Table.
- In the getter function, identifies one DPS to be read.

`pPinList` specifies which DPS(s) are to be programmed. In Multi-DUT Test Programs, the DPSs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **`pPinList`** must only contains pins mapped to a DPS in the Pin Assignment Table.

The getter version of `dps_comp_cap()` returns the currently selected compensation capacitor for one DPS. Legal return values are noted in the table above. In Multi-DUT Test Programs, the value is retrieved from the first DUT in the Active DUTs Set (ADS).

Examples

The following example performs a current test on the DPS(s) included in the pin list named VCC. The DPS compensation capacitance is set assuming a DPS capacitive load between 0.1uF and 1.0uF:

```
dps_comp_cap( 1 );
result = test_supply ( pass_nicl, VCC );
```

3.11.13 DPS 300mA/600mA DPS Option

Note: first available in software release h1.1.23.

Description

An optional Magnum 1/2 hardware configuration allows DUT Power Supply (DPS) to output up to ±600mA in VPulse mode or ±300mA per output in Independent Mode (see DPS Output Mode and DPS Operating Area). This extends the DPS output current capability from ±400mA (VPulse mode) and ±200mA (Independent Mode).

This option requires the correct Magnum 1/2 hardware configuration, which can be evaluated two ways:

- The BoardRevs program will indicate when this option is usable.
- The `dps_ilimit_set()` function will generate an error if an attempt is made to use the expanded current capability on systems which do not have the proper hardware configuration.

When using Magnum 1/2 systems with this hardware configuration all DPS operation defaults to the original system's current capabilities; i.e. ±400mA/±200mA max. The `dps_ilimit_set()` function must be executed to enable the higher current capability. This affects all DPS.

3.11.13.1 dps_ilimit_set(), dps_ilimit_get()

See DPS 300mA/600mA DPS Option.

Description

The `dps_ilimit_set()` function is used to enable the high-current option of DUT Power Supply (DPS) operation, available via the DPS 300mA/600mA DPS Option. See DPS Operating Area.

Executing `dps_ilimit_set()` does the following:

- Checks to confirm that the system in use has the required hardware configuration. If not, a warning is generated and the standard DPS output current ranges apply.
- Enables the DPS hardware to output the higher maximum current.
- Enables the DPS software to allow use of the ±4A DPS current sense range. The current sense range is selected using DPS Current Test Limit Functions (`dps_current_high()` and `dps_current_low()`).

The `dps_ilimit_get()` function is used to identify whether the high-current option is currently enabled.

Note the following:

- `dps_ilimit_set()` sets a global state which affects all DPS identically.
- During initial program load, the system software sets the mode to `t_dps_default_ilimit`. The system software does not otherwise change this mode.

Usage

The following function is used to set the DPS maximum current operation of all DPS:

```
void dps_ilimit_set( DpsILimit limit );
```

The following function is used to get the DPS maximum current selection:

```
DpsILimit dps_ilimit_get();
```

where:

`limit` specifies which DPS current option is to be set. Legal values are of the `DpsILimit` enumerated type.

`dps_ilimit_get()` returns the currently enabled DPS current selection.

Example:

```
dps_ilimit_set( t_dps_high_ilimit );
DpsILimit l = dps_ilimit_get();
```

3.12 High Voltage Source/Measure Unit (HV) Functions

See Site Assembly Board Block Diagram, High Voltage Source/Measure Unit (HV).

This section includes the following:

- Overview
- HV Connect/Disconnect Functions
- HV Voltage Programming Functions
- HV Current Test Limit Functions
- HV Voltage PASS/FAIL Limit Functions
- HV Static Test Functions
- HV Dynamic Test Functions

Other related information includes:

- Static DC Tests and Dynamic DC Tests
- Parametric Settling Time and Built-in Settling Time
- `measure()`
- Retrieving DC Test Results

3.12.1 Overview

See High Voltage Source/Measure Unit (HV) Functions, High Voltage Source/Measure Unit (HV).

Each Site Assembly Board contains 16 high voltage source/measure units, effectively one for each 8 pins. See High Voltage Source/Measure Unit (HV). Note the following:

- A solid-state switch allows each HV to connect to or disconnected from the DUT. User code must explicitly control these connections. See HV Connect/Disconnect Functions.
- The HV can be used to supply a positive DC voltage to one or more DUT pin(s). HV Voltage Programming Functions are used to program the HV output voltage.
- Both HV output current and voltage can be tested using HV Static Test Functions and HV Dynamic Test Functions. Both Go/NoGo tests and measurements are supported. However, as noted, each Site Assembly Board has 16 HV units but only

8 DC Test and Measure Systems. This means that when testing or measuring HV output current or voltage it may not always be possible to perform all HV tests in parallel.

- When testing HV output current the PASS/FAIL test limits are set using HV Current Test Limit Functions.
- When testing HV output voltage the PASS/FAIL test limits are set using HV Voltage PASS/FAIL Limit Functions.
- The HV has no voltage or current clamps.

3.12.2 HV Connect/Disconnect Functions

See High Voltage Source/Measure Unit (HV), High Voltage Source/Measure Unit (HV) Functions.

Description

The `hv_connect()` function is used to close the solid-state switch(es) connecting High Voltage Source/Measure Unit (HV) to DUT pins. The `hv_disconnect()` function is used to open these switch(es).

During initial program load, all HV connect switches are opened. The system software does not otherwise control these switches; i.e. once they are closed (or opened), they remain closed (or opened) until user code changes them again. This is consistent with DPS operation.

It is common to use `hv_connect()` in the `SITE_BEGIN_BLOCK()`, to set HV connections which are constant for the duration of the test program.

The voltage test/measure signal between the HV and the DC Test and Measure System is connected to the output-side of the solid-state switch which connects the HV to the DUT. This means that it is possible to test/measure this voltage, from the DUT, even though the HV is not connected. See HV Static Test Functions and HV Dynamic Test Functions.

Usage()

The following function connects one specified HV:

```
void hv_connect( DutPin *pDutPin );
```

The following function connects one or more specified HV(s):

```
void hv_connect( PinList* pPinList );
```

The following function disconnects one specified HV:

```
void hv_disconnect( DutPin *pDutPin );
```

The following function disconnects one or more specified HV(s):

```
void hv_disconnect( PinList* pPinList );
```

where:

`pDutPin` identifies one HV to be programmed. In Multi-DUT Test Programs, the HVs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a HV in the Pin Assignment Table.

`pPinList` specifies which pins are to be affected. In Multi-DUT Test Programs, the HVs of each DUT currently in the Active DUTs Set (ADS) are affected. The pin list must only contain pins which are mapped to HV in the Pin Assignment Table.

Example

The example below causes all HV specified in the pin list named `pl_HVpins` to be disconnected. Then some test program code is executed, and these same HV are reconnected.

```
hv_disconnect( pl_HVpins );
// Other code executes here with all pl_HVpins HV disconnected
hv_connect( pl_HVpins );
```

3.12.3 HV Voltage Programming Functions

See High Voltage Source/Measure Unit (HV), High Voltage Source/Measure Unit (HV) Functions.

Description

The `hv_voltage_set()` function is used to program the output voltage for one or more High Voltage Source/Measure Unit (HV).

The `hv_voltage_get()` function is used to get the currently programmed output voltage for one HV.

The HV has a single output voltage range:

Table 3.12.3.0-1 HV Voltage Range

Range	LSB
0V to +28V	2mV

HV voltages are typically programmed in Test Blocks, or other C-code called from test blocks.

All HV output voltages are set to 0V during initial program load. Executing the functions below takes effect immediately. Once programmed, HV output voltage remains in effect until:

- Reprogrammed by user code.
- Modified by a test pattern, see Controlling PE Levels from the Test Pattern.
- Sequence & Binning Table execution stops, at which time the `builtin_after_testing_block` sets the HV output voltage to 0V.

Usage

The following function programs the output voltage for all HVs. In Multi-DUT Test Programs, only HVs of DUTs currently in the Active DUTs Set (ADS) are affected:

```
void hv_voltage_set( double value );
```

The following function programs the output voltage for one specifed HV:

```
void hv_voltage_set( double value, DutPin *pDutPin );
```

The following function programs the output voltage for all HV(s) in the specified pin list:

```
void hv_voltage_set( double value, PinList* pPinList );
```

The following function returns the currently programmed output voltage for one specified HV:

```
double hv_voltage_get( DutPin *pin );
```

where:

`value` specifies the desired output voltage. Units may be used, see Specifying Units.

`pDutPin` is used in two contexts:

- In the setter function, identifies one HV to be programmed. In Multi-DUT Test Programs, the HVs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a HV in the Pin Assignment Table.
- In the getter function, identifies one HV to be read.

`pPinList` identifies one or more HV units to be programmed. In Multi-DUT Test Programs, only pin(s) of DUT(s) currently in the Active DUTs Set (ADS) are affected. The pin list must only contain pins which are mapped to a HV in the Pin Assignment Table.

`pin` identifies one `DutPin` to be read. Must be a `DutPin` mapped to HV in the Pin Assignment Table.

`hv_voltage_get()` returns the currently programmed output voltage of the specified v unit of the first DUT in the Active DUTs Set (ADS).

Examples

The following example programs the output voltage for each HV included in the pin list named `pl_HVpins` to 15V. In Multi-DUT Test Programs, only pin(s) of DUT(s) currently in the Active DUTs Set (ADS) are affected.

```
hv_voltage_set( 15 V, pl_HVpins );
```

The following example gets the current output voltage for one HV, named `HVpin`. The value is returned for the first DUT in the Active DUTs Set (ADS):

```
double v = hv_voltage_get( HVpin );
```

3.12.4 HV Current Test Limit Functions

See High Voltage Source/Measure Unit (HV), High Voltage Source/Measure Unit (HV) Functions.

Description

The `hv_ipar_high()` and `hv_ipar_low()` functions are used to set or get the high and low PASS/FAIL test limits for HV current tests. These are test limits used by `hv_test_supply()` and `hv_ac_test_supply()`.

`hv_ipar_high()` and `hv_ipar_low()` must be programmed before executing HV Static Test Functions and HV Dynamic Test Functions which test or measure HV current.

are set to zero at test program initialization, but are not otherwise modified by the system software.

The HV has a single current range:

Table 3.12.4.0-1 HV Current Range

Range	LSB
0mA to 8mA	4uA

Both limits can be modified from a test pattern. See Controlling PE Levels from the Test Pattern.

Note: a commonly made mistake is to assume that programming HV force value or PASS/FAIL test limit also defines the type of HV test which will execute next. It is the arguments passed to `hv_test_supply()` and `hv_ac_test_supply()` which define the type of test the HV actually performs (test/measure current or voltage), and thus which force and test limits will be used.

Usage

The following functions program the high/low current test limit for all HVs. In Multi-DUT Test Programs, only HVs of DUTs currently in the Active DUTs Set (ADS) are affected:

```
void hv_ipar_high( double value );
void hv_ipar_low( double value );
```

The following functions program the high/low current test limit for one specified HV:

```
void hv_ipar_high( double value, DutPin *pDutPin );
void hv_ipar_low( double value, DutPin *pDutPin );
```

The following functions program the high/low current test limit for one or more HV(s):

```
void hv_ipar_high( double value, PinList* pPinList );
void hv_ipar_low( double value, PinList* pPinList );
```

The following functions get the currently programmed high/low current test limit for one HV:

```
double hv_ipar_high( DutPin *pin );
double hv_ipar_low( DutPin *pin );
```

where:

value specifies the desired high or low current value. Units may be used (see Specifying Units).

pDutPin is used in two contexts:

- In the setter function, identifies one HV to be programmed. In Multi-DUT Test Programs, the HVs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a HV in the Pin Assignment Table.
- In the getter function, identifies one HV to be read.

pPinList identifies one or more HV units to be programmed. In Multi-DUT Test Programs, only pin(s) of DUT(s) currently in the Active DUTs Set (ADS) are affected. The pin list must only contain pins which are mapped to HV in the Pin Assignment Table.

pin identifies one `DutPin` to be read. Must be a `DutPin` mapped to HV in the Pin Assignment Table.

The `hv_ipar_high()` and `hv_ipar_low()` getter functions return the currently programmed value. The value is retrieved from the first DUT in the Active DUTs Set (ADS).

Example

The following example sets the high and low current test limits all HV(s) included in the pin list named `pl_HVpins`:

```
hv_ipar_high( 10 MA, pl_HVpins );
hv_ipar_low( 30 UA, pl_HVpins );
```

The following example gets the currently programmed high current test limit for one HV named `hvpin`:

```
double v = hv_ipar_high( hvpin );
```

3.12.5 HV Voltage PASS/FAIL Limit Functions

See High Voltage Source/Measure Unit (HV), High Voltage Source/Measure Unit (HV) Functions.

Description

The `hv_vpar_high()` and `hv_vpar_low()` functions are used to set or get the high and low PASS/FAIL test limits for High Voltage Source/Measure Unit (HV) voltage tests. These

are test limits used by `hv_test_supply()` and `hv_ac_test_supply()` when testing HV output voltage.

`hv_vpar_high()` and `hv_vpar_low()` must be programmed before executing HV Static Test Functions and HV Dynamic Test Functions which test or measure HV voltage.

Both limits are set to zero at test program initialization, but are not otherwise modified by the system software.

The HV has a single output voltage range:

Table 3.12.5.0-1 HV Voltage Range

Range	LSB
0V to +28V	4mV

Both limits can be modified from a test pattern. See Controlling PE Levels from the Test Pattern.

Note: a commonly made mistake is to assume that programming HV force value or PASS/FAIL test limit also defines the type of HV test which will execute next. It is the arguments passed to `hv_test_supply()` and `hv_ac_test_supply()` which defines the type of test the HV actually performs (test/measure current or voltage), and thus which force and test limits will be used.

Usage

The following functions set the high/low voltage test limit for all HV(s). In Multi-DUT Test Programs, only HVs of DUTs currently in the Active DUTs Set (ADS) are affected:

```
void hv_vpar_high( double value );
void hv_vpar_low( double value );
```

The following functions set the high/low voltage test limit for one or more HV(s):

```
void hv_vpar_high( double value, PinList* pPinList );
void hv_vpar_low( double value, PinList* pPinList );
```

The following functions get the currently programmed high/low voltage test limit for one HV:

```
double hv_vpar_high( DutPin *pin );
double hv_vpar_low( DutPin *pin );
```

where:

`value` specifies the desired high or low voltage value. Units may be used (see Specifying Units).

`pPinList` identifies one or more HV units to be programmed. In Multi-DUT Test Programs, only pin(s) of DUT(s) currently in the Active DUTs Set (ADS) are affected. The pin list must only contain pins which are mapped to HV in the Pin Assignment Table.

`pin` identifies one `DutPin` to be read. Must be a `DutPin` mapped to HV in the Pin Assignment Table.

The `hv_vpar_high()` and `hv_vpar_low()` getter functions return the currently programmed value. The value is retrieved from the first DUT in the Active DUTs Set (ADS).

Example

The following example sets the high and low voltage test limits all HV(s) included in the pin list named `pl_HVpins`:

```
hv_vpar_high( 7.5 V, pl_HVpins );
hv_vpar_low( 6.5 V, pl_HVpins );
```

The following example gets the currently programmed high voltage test limit for one HV named `hvpin`:

```
double v = hv_vpar_high( hvpin );
```

3.12.6 HV Static Test Functions

See High Voltage Source/Measure Unit (HV), High Voltage Source/Measure Unit (HV) Functions, DC Sub-System Block Diagram, Static DC Tests.

Description

The `hv_test_supply()` function is used to statically test or measure HV current or output voltage. HV Dynamic Test Functions are documented separately.

See Static DC Tests.

As noted in Overview, each Site Assembly Board has 16 HV units and 8 DC Test and Measure Systems; i.e. in hardware, two HV units are associated with a given DC Test and Measure System. This means that when testing or measuring HV output current or voltage it

may not always be possible to perform all HV tests in parallel. Thus, when executing `hv_test_supply()`, if the specified pin list contains HV units which share a given DC Test and Measure System, the test will be performed in two steps, testing one HV at a time.

In software release h1.1.23, `hv_test_supply()` support for measurement averaging was enhanced. Note the following:

- Measurement averaging is enabled by including the `PartestOpt iacc` argument.
- The number of measurements made to obtain the average is set using `iacc_count_set()`. Default = 10.
- The value set using `iacc_count_set()` is ignored when `measure()` = FALSE.
- When averaging is enabled the measurement average is compared to the PASS/FAIL test limits, set using HV Current Test Limit Functions, to determine whether `hv_test_supply()` returns PASS or FAIL.
- When Retrieving DC Test Results only the average value is returned, regardless of the number of measurements made.

Prior to executing `hv_test_supply()` the following parameters must be set up:

- HV connections. See HV Connect/Disconnect Functions. Note that the voltage (but not current) test/measure signal between the High Voltage Source/Measure Unit (HV) and the DC Test and Measure System is connected to the output-side of the solid-state switch which connects the HV to the DUT. This means that it is possible to test/measure this voltage, from the DUT, even though the HV is not connected.
- HV output voltage. See HV Voltage Programming Functions.
- PASS/FAIL voltage or current test limits. See HV Current Test Limit Functions and HV Voltage PASS/FAIL Limit Functions.
- The system software provides a Built-in Settling Time to HV tests. The user may use the `partime()` function to add additional settling time. See Parametric Settling Time.
- Enable/disable measurements using `measure()`. Measured values can be retrieved by user code, see Retrieving DC Test Results.

Usage

The following function executes a static HV test. In Multi-DUT Test Programs, only HV(s) connected to DUT(s) currently in the Active DUTs Set (ADS) are tested:

```
PFState hv_test_supply( PassCond pass_cond,
                        PinList* pPinList,
                        PartestOpt test_type DEFAULT_VALUE( no_iacc ));
```

```
PFState hv_test_supply(
                PassCond pass_cond,
                DutPin *pDutPin,
                PartestOpt test_type DEFAULT_VALUE( no_iacc ));
```

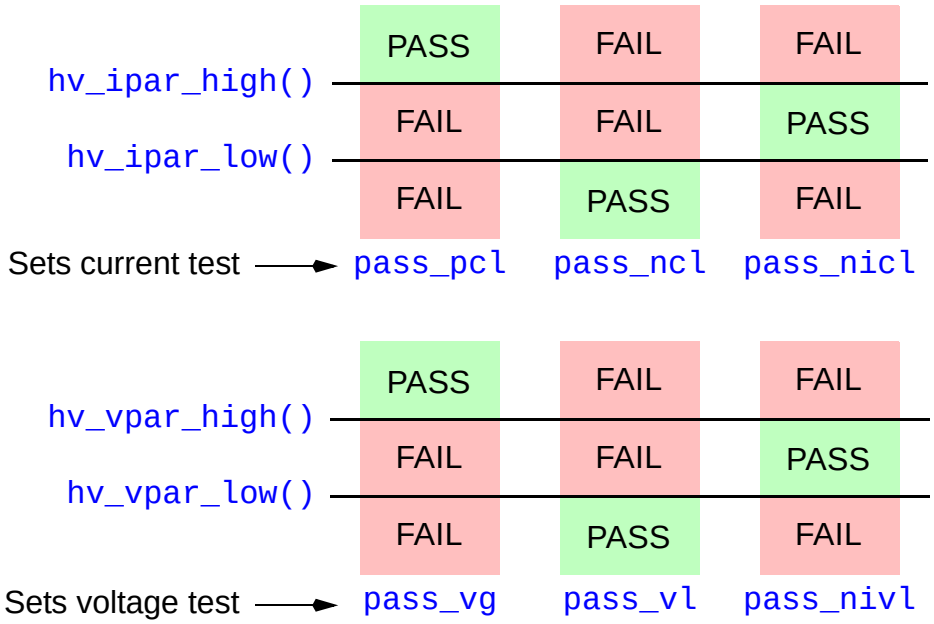
where:

pass_cond determines whether `hv_test_supply()` tests/measures output current or voltage, and therefore which PASS/FAIL test limits will be used. **pass_cond** values are defined using the `PassCond` enumerated type. Operation is defined in the following table:

Table 3.12.6.0-1 HV Test Options and PASS/FAIL Limit Selection

Pass Condition	Comments
`pass_pcl`	*pcl* = Positive Current Limit. Test/measure HV output current. Pass if current is greater than the value set using `hv_ipar_high()`.
`pass_ncl`	*ncl* = Negative Current Limit. Test/measure HV output current. Pass if current is less than the value set using `hv_ipar_low()`.
`pass_nicl`	*nicl* = Not In Current Limit. Test/measure HV output current. Pass if current is between the values set using `hv_ipar_high()` and `hv_ipar_low()`.
`pass_vg`	*vg* = Voltage Greater. Test/measure HV output voltage. Pass if voltage is greater than the value set using `hv_vpar_high()`.
`pass_vl`	*vl* = Voltage Less. Test/measure HV output voltage. Pass if voltage is less than the value set using `hv_vpar_low()`.
`pass_nivl`	*nivl* = Not In Voltage Limit. Test/measure HV output voltage. Pass if voltage is between the values set using `hv_vpar_high()` and `hv_vpar_low()`.

The diagrams below show this same information graphically. The upper diagram applies when testing current, the lower diagram applies when testing voltage:



pPinList identifies which HV are to be tested. In Multi-DUT Test Programs, only pin(s) of DUT(s) currently in the Active DUTs Set (ADS) are tested. The pin list must only contain pins which are mapped to HV in the Pin Assignment Table. When **pPinList** contains both HV units which share a given DC Test and Measure System, the test will be performed in two steps, testing one HV at a time, in the order they are listed in **pPinList**.

pDutPin identifies one HV to be tested. In Multi-DUT Test Programs, only pin(s) of DUT(s) currently in the Active DUTs Set (ADS) are tested. The pin must by mapped to an HV in the Pin Assignment Table.

test_type is optional, and is used to select one HV test option. Legal values are of the PartestOpt enumerated type, but only the options noted in the table below are valid:

Table 3.12.6.0-2 HV Test Optional Arguments

Optional Arguments	Comments
iacc	Enable measure averaging. Applies only when measure() = TRUE. See Description and Measurement Average Count Functions.
no_iacc	Default. Disable measure averaging.

`hv_test_supply()` returns the overall PASS/FAIL result. A PASS result will only be returned if all HV(s) tested PASS (i.e. no DC Error Flags are set). In Multi-DUT Test Programs, only HVs of DUTs currently in the Active DUTs Set (ADS) affect the test result.

Example

The following example tests output current on all HV units included in the pin list named `pl_HVpins`. The test will pass if the current is between the PASS/FAIL limits set using `hv_ipar_high()` and `hv_ipar_low()`:

```
PFState pf = hv_test_supply( pass_nicl, pl_HVpins );
```

3.12.7 HV Dynamic Test Functions

See High Voltage Source/Measure Unit (HV), High Voltage Source/Measure Unit (HV) Functions, DC Sub-System Block Diagram, Dynamic DC Tests.

Description

The `hv_ac_test_supply()` function is used to dynamically test or measure HV current or output voltage. HV Static Test Functions are documented separately.

See Dynamic DC Tests.

The `hv_ac_test_supply()` function executes a test pattern, and the associated pattern stop condition must be specified. Options are listed in Usage, and include terminating pattern execution if there is a functional failure or executing the pattern to completion, regardless of functional failures.

Note: in dynamic DC parametric tests, the test pattern can perform branch-on-error based on the state of the DC Error Flag in each DC Test and Measure System. The `MAR RESET`, or `CHIPS RESET` instructions (Memory Test Patterns) or `VEC/RPT RESET`, `VAR RESET` and `VPINFUNC RESET` instructions (Logic Test Patterns) will clear the DC Error Flag, which can affect the overall PASS/FAIL result of the `hv_ac_test_supply()`.

The `hv_ac_test_supply()`function has two trigger modes, controlled using the `comp_cond` argument, which determines whether test pattern triggers are enabled. Four scenarios are possible:

- comp_cond = default (no_vcomp) and measure() = FALSE: the DC Comparators and Error Logic are used and enabled for the entire duration of the executing test pattern.
- comp_cond = vcomp and measure() = FALSE: the DC Comparators and Error Logic are used and triggered by the VCOMP token from the executing test pattern. If any one or more triggered samples fails the test returns FAIL.
- comp_cond = vcomp and measure() = TRUE: the DC A/D Converter is used and triggered by the VCOMP token from the executing test pattern. This causes a measurement to be made.
- comp_cond = default (no_vcomp) and measure() = TRUE. Not supported in DC tests. hv_ac_test_supply() operates as though measure() = FALSE. Any DC measurements which are retrieved are invalid.

When using comp_cond = vcomp, one trigger will be generated in each test pattern cycle which contains the MAR VCOMP instruction (Memory Test Patterns) or VEC/RPT VCOMP, VAR VCOMP and VPINFUNC VCOMP instruction s (Logic Test Patterns). See Dynamic DC Tests.

Note: when using vcomp, if the pattern does not generate any triggers, hv_ac_test_supply() will return invalid results. If measure = FALSE, the DC Comparators and Error Logic will never get triggered, which will result in a PASS condition. If measure() = TRUE, any measured values will be invalid (old, stale, etc.) and the test may return PASS or FAIL based on the invalid measurement data. The system software cannot check for this error; i.e. it is the user's responsibility to ensure that at least one vcomp trigger is issued by the pattern.

---

hv_ac_test_supply() will not return until test pattern execution terminates.

hv_ac_test_supply() will also fail if any PE error latches are set (latched functional strobes failed).

As noted in Overview, each Site Assembly Board has 16 HV units and 8 DC Test and Measure Systems; i.e. in hardware, two HV units are associated with a given DC Test and Measure System. When executing hv_ac_test_supply(), it is an error if the specified pin list contains HV units which share a given DC Test and Measure System:

- The test will return immediately
- The test result for all DUT(s) tested will be FAIL.
- Any test results retrieved will be invalid (stale, etc.) See Retrieving DC Test Results.

Prior to executing `hv_ac_test_supply()` the following parameters must be set up:

- HV connections. See HV Connect/Disconnect Functions. Note that the voltage (but not current) test/measure signal between the HV and the DC Test and Measure System is connected to the output-side of the solid-state switch which connects the HV to the DUT. This means that it is possible to test/measure this voltage, from the DUT, even though the HV is not connected.
- HV output voltage. See HV Voltage Programming Functions.
- PASS/FAIL voltage or current test limits. See HV Current Test Limit Functions and HV Voltage PASS/FAIL Limit Functions.
- The `partime()` function, used to add additional settling time to the Built-in Settling Time, is not normally useful in dynamic DC tests. This is because this delay will occur after programming the DC circuitry but before executing the functional test pattern; i.e. at a non-useful time. See Parametric Settling Time.
- Enable/disable measurements using `measure()`. Measured values can be retrieved by user code, see Retrieving DC Test Results.
- For dynamic tests, the test will also fail if any functional strobes fail. Thus proper digital PE levels and timing will affect test results, as does the test pattern executed.

Usage

The following function is used to perform dynamic current or voltage test on one or more HVs. In Multi-DUT Test Programs, only HV(s) connected to DUT(s) currently in the Active DUTs Set (ADS) are tested:

```
PFState hv_ac_test_supply(PassCond pass_cond,
                          PinList* pPinList,
                          Pattern *pPattern,
                          PatStopCond stop_cond,
                     CompCond comp_cond DEFAULT_VALUE( no_vcomp ));
```

where:

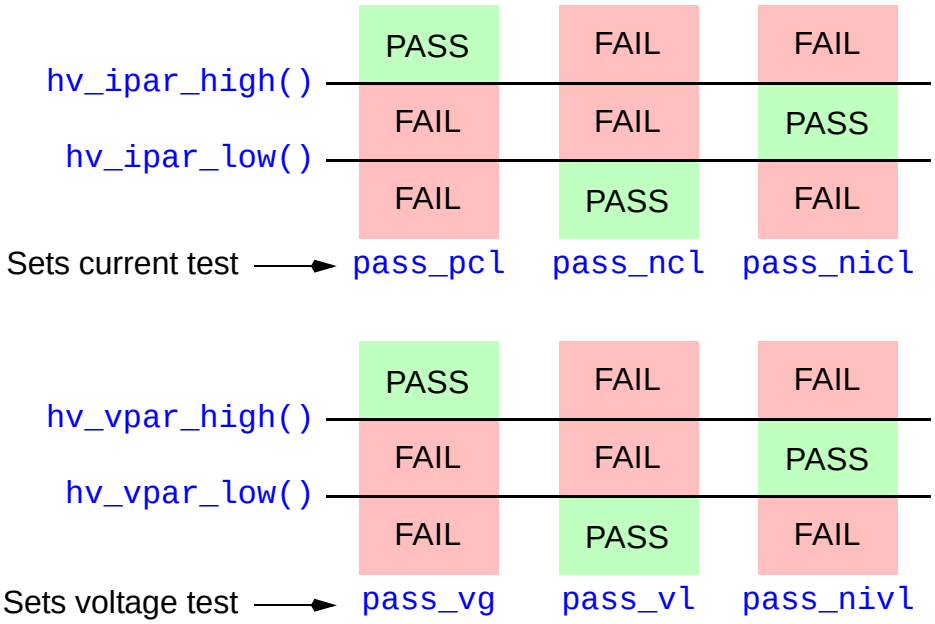
pass_cond determines whether `hv_ac_test_supply()` tests or measures output current or voltage, and therefore which PASS/FAIL test limits will be used. **pass_cond**

values are defined using the PassCond enumerated type. Operation is defined in the following table:

Table 3.12.7.0-1 HV Test Options and PASS/FAIL Limit Selection

Pass Condition	Comments
pass_pcl	*pcl* = Positive Current Limit. Test/measure HV output current. Pass if current is greater than the value set using hv_ipar_high().
pass_ncl	*ncl* = Negative Current Limit. Test/measure HV output current. Pass if current is less than the value set using hv_ipar_low().
pass_nicl	*nicl* = Not In Current Limit. Test/measure HV output current. Pass if current is between the values set using hv_ipar_high() and hv_ipar_low().
pass_vg	*vg* = Voltage Greater. Test/measure HV output voltage. Pass if voltage is greater than the value set using hv_vpar_high().
pass_vl	*vl* = Voltage Less. Test/measure HV output voltage. Pass if voltage is less than the value set using hv_vpar_low().
pass_nivl	*nivl* = Not In Voltage Limit. Test/measure HV output voltage. Pass if voltage is between the values set using hv_vpar_high() and hv_vpar_low().

The diagrams below shows this same information graphically. The upper diagram applies when testing current, the lower diagram applies when testing voltage:



`pPinList` identifies which HV are to be tested. In Multi-DUT Test Programs, only pin(s) of DUT(s) currently in the Active DUTs Set (ADS) are tested. The pin list must only contain pins which are mapped to HV in the Pin Assignment Table. When **`pPinList`** contains both HV units which share a given DC Test and Measure System, the test will be performed in two steps, testing one HV at a time, in the order they are listed in **`pPinList`**. When this occurs, the test pattern will be executed two times.

`pPattern` identifies the test pattern to be executed by `hv_ac_test_supply()`.

`stop_cond` controls how test pattern execution terminates. Legal **`stop_cond`** values are defined using the `PatStopCond` enumerated type. Note that majority of tests will use the

`error` or `finish` options: **`comp_cond`** is optional, and controls whether test pattern

Table 3.12.7.0-2 Pattern Execution Stop Condition Options

Stop Condition	Summary Description
`finish`	Execute pattern to completion, regardless of errors.
`error`	Stop pattern execution on first functional or DC Error Flag error and set the result of `ac_partest()` to `FAIL`.
`fullec`	Execute pattern to completion. Enable the ECR to capture errors during pattern execution. This argument should be used when performing Redundancy Analysis (RA) or using BitmapTool.
`LEC_only_errors`	Enable the ECR to capture errors during pattern execution. Capture the first 2Meg (2^{21}-6) failing vectors. Intended for use when the ECR is used as an Logic Error Catch (LEC).
`LEC_first_vectors`	Enable the ECR to capture errors during pattern execution. Capture the first 2Meg (2^{21}-6) vectors executed. Ignores PASS/FAIL. Intended for use when the ECR is used as an Logic Error Catch (LEC).
`LEC_last_vectors`	Enable the ECR to capture errors during pattern execution. Capture the last 2Meg (2^{21}-6) vectors executed. Ignores PASS/FAIL. Intended for use when the ECR is used as an Logic Error Catch (LEC).
`LEC_before_error`	Enable the ECR to capture errors during pattern execution. Capture the first failing vector plus the previous 2Meg (2^{21}-6) vectors executed. Intended for use when the ECR is used as an Logic Error Catch (LEC).

Table 3.12.7.0-2 Pattern Execution Stop Condition Options *(Continued)*

Stop Condition	Summary Description
LEC_after_error	Enable the ECR to capture errors during pattern execution. Capture the first failing vector plus the next 2Meg (2^{21}-6) vectors executed. Intended for use when the ECR is used as an Logic Error Catch (LEC). .
LEC_center_error	Enable the ECR to capture errors during pattern execution. Capture the first failing vector plus up to 512K vectors executed before the failure and up to 512K vectors executed after the failure. Intended for use when the ECR is used as an Logic Error Catch (LEC).
Note: in parallel test applications, the test pattern must be executed to completion, to ensure that DUT(s) which don't fail are completely tested. In other words, halting the pattern early (error) because one or more DUT(s) failed prevents DUT(s) which PASS from being completely tested. This is BAD.	

triggers are enabled. See Description. Default = no_vcomp = disabled.

hv_ac_test_supply() returns the overall PASS/FAIL result. A PASS result will only be returned if:

- All HV(s) tested pass (no DC Error Flags are set).
- No functional error latches are set (no functional strobes failed)
- In Multi-DUT Test Programs, only HVs of DUTs currently in the Active DUTs Set (ADS) affect the test result.

Example

The following example tests output current on all HV units included in the pin list named pl_HVpins. The test pattern named myPat will be executed, to completion. Test pattern triggers are enabled. The test will pass if the current is between the PASS/FAIL limits set using hv_ipar_high() and hv_ipar_low() AND no functional strobes fail:

```
PFState pf = hv_ac_test_supply(pass_nicl,
                               pl_HVpins,
                               myPat,
                               finish,
                               vcomp );
```

If the test program is a Multi-DUT Test Program, the return value will be `MULTI_DUT`, see above.

3.13 PMU Functions

See Parametric Measurement Unit (PMU).

This section includes the following topics:

- Overview
- Types, Enums, etc.
- PMU Connect/Disconnect Functions
- PMU Force Current Functions
- PMU Current Test Limit Functions
- PMU Force Voltage Functions
- PMU Voltage Test Limit Functions
- PMU Voltage Clamp Functions
- Background Voltage Functions
- PMU Static Test Functions
- PMU Dynamic Test Functions
- `start_ac_partest(), stop_ac_partest()`
- `ac_partest_results_store()`
- PMU: Testing DPS Pins
- PMU: Testing HV Pins
- `parametric_mode()`
- PMU as Voltage/Current Source (`pmu_connect()`, `pmu_disconnect()`)
- PMU Compensation Capacitors

Other related information includes:

- Static DC Tests and Dynamic DC Tests
- Parametric Settling Time and Built-in Settling Time
- `measure()`
- Retrieving DC Test Results

3.13.1 Overview

See Parametric Measurement Unit (PMU), PMU Functions, Overview.

Each Magnum 1/2/2x Site Assembly Board board contains 8 Parametric Measurement Unit (PMU)s.

The PMU can be used to:

- Force voltage, test or measure current
- Force current, test or measure voltage
- Act as a static voltage source
- Act as a static current source

PMU test options include:

- Perform a Go/NoGo test or make a measurement
- Perform a static PMU test or dynamic PMU test
- Dynamic PMU tests can be triggered by the site controller computer or by triggers from an executing test pattern.

Note: these options interact. Detailed operation and usage rules are described in Static DC Tests and Dynamic DC Tests. The user must understand the information in these sections to obtain valid test results.

The PMU Force Current Functions are used to set or get a PMU force current value.

The PMU Force Voltage Functions are used to set or get a PMU force voltage value.

These parameters can both be set, and do not supersede each other. The actual use of these values, in hardware, does not occur until a PMU test is executed (using PMU Static Test Functions or PMU Dynamic Test Functions), or until explicit PMU connections are made using PMU Connect/Disconnect Functions.

PMU current test PASS/FAIL limits are set using PMU Current Test Limit Functions.

PMU voltage test PASS/FAIL limits are set using PMU Voltage Test Limit Functions.

Again, these parameters can both be set, and do not supersede each other. The actual use of these values, in hardware, does not occur until a PMU test is executed, using PMU Static Test Functions or PMU Dynamic Test Functions.

The PMU Current Test Limit Functions also set the PMU current range, for both force current and sense current operation. By default, range setting is implicit, however, an explicit range can be specified using an argument to the PMU Current Test Limit Functions.

The PMU has two voltage ranges which only apply to PASS/FAIL test limits. By default, range setting is implicit, however, an explicit range can be specified using an argument to the PMU Voltage Test Limit Functions.

Explicit PMU range programming is required when changing voltage or current values from an executing test pattern. See Controlling PE Levels from the Test Pattern.

PMU tests are set up and executed using PMU Static Test Functions and PMU Dynamic Test Functions. Again, lots of important information is covered in Static DC Tests and Dynamic DC Tests.

PMU tests are performed on a user specified pin list, which can consist of signal pins, HV pins, or DPS pins (one type at a time). By default, PMU tests are executed sequentially, testing one pin at a time, testing pins in the order they occur in the specified pin list. The parallel options (there are 2) only apply when testing signal pins.

As indicated, the PMU can be used to test DPS pins. See PMU: Testing DPS Pins.

The PMU can be used to test HV pins. See PMU: Testing HV Pins.

The PMU has two voltage clamps, programmed using PMU Voltage Clamp Functions. When the PMU is forcing current (into a high resistance DUT circuit) the resulting PMU voltage may reach the maximum positive or negative voltage available from the PMU. The PMU voltage clamps can be used to independently limit the maximum and minimum voltage generated by the PMU.

When performing a static PMU test, to sequentially test multiple pins, the other pins can be forced to a user specified background voltage. This can greatly simplify continuity tests and leakage tests. See Background Voltage Functions.

The PMU can also be used as a statically connected voltage or current source. See PMU as Voltage/Current Source. This is normally the only reason for user code to use the PMU Connect/Disconnect Functions.

Various PMU voltages and currents can be set or modified from an executing test pattern. See Controlling PE Levels from the Test Pattern.

The PMU is closed-loop feedback system, with stability that is affected by the circuit load. The PMU has selectable internal compensation capacitors, which are controlled using `pmu_comp_cap()` function. The selection is based on the amount of load capacitance connected to pin(s) being tested or connected to the PMU. See PMU Compensation Capacitors.

3.13.2 Types, Enums, etc.

Description

The following enumerated types are used in support of various PMU Functions:

Usage

The `PMUSense` enumerated type is used to set the PMU connection when using the PMU as Voltage/Current Source:

```
enum PMUSense { REM, LOCAL };
```

The `PMUMode` enumerated type is used to specify the PMU or PTU operating mode; i.e. force voltage or force current:

```
enum PMUMode { FORCEV, FORCEI };
```

3.13.3 PMU Connect/Disconnect Functions

See Parametric Measurement Unit (PMU), PMU Functions, Overview.

The `pmu_connect()` and `pmu_disconnect()` functions are used to explicitly control connections between an PMU and one or more pin(s). These functions are normally only used when the PMU is used as a statically connected voltage or current source and are thus documented in the section titled PMU as Voltage/Current Source.

Note: standard PMU tests using `partest()` and `ac_partest()` ***automatically*** control PMU connections and disconnections to pins; i.e. it is ***NOT*** necessary to use `pmu_connect()` and `pmu_disconnect()` when performing these tests.

3.13.4 PMU Force Current Functions

See Parametric Measurement Unit (PMU), PMU Functions, Overview.

Definition

The `ipar_force()` function is used to set or get the PMU force current value. Note the following:

- All PMU voltages and currents are set to 0V/0A during the initial program load, *except* for the PMU voltage clamps, which are set to maximum values, effectively disabling both voltage clamps.
- During Sequence & Binning Table execution, PMU voltage and current values are managed by user-written C-code.
- When Sequence & Binning Table execution stops, all PMUs are set to 0V/0A and disconnected from all pins. This is done in the `builtin_after_testing_block`.
- The PMU has multiple force current ranges. See Usage.
- PMU force current has multiple ranges. See Usage. By default, the range is implicitly selected, by the system software, based on the force current value programmed using `ipar_force()`. Optionally, an explicit range can be specified using the `range` argument to `ipar_force()`.
- PMU voltage and current parameters can be modified from an executing test pattern. See Controlling PE Levels from the Test Pattern. A current range value must be explicitly specified if the PMU force current will be set or modified from an executing test pattern. See Controlling PE Levels from the Test Pattern.
- Programming the PMU force current value has no effect on a previously programmed force voltage, and vice versa. The actual use of these values, in hardware, does not occur until a PMU test is executed (using PMU Static Test Functions or PMU Dynamic Test Functions), or until explicit PMU connections are made using PMU Connect/Disconnect Functions.
- It is possible to change the PMU force current value while the PMU is statically connected as a current source, BUT the current range cannot be changed (implicitly or explicitly). Violating this rule will generate a warning in the appropriate controller window, and the force current will not be modified.

Note: a commonly made mistake is to assume that programming a PMU force value or PASS/FAIL test limits also defines the type of PMU test which will execute next. It is the arguments passed to `partest()` or `ac_partest()` which define the type of test the PMU will perform (force current/measure voltage, etc.), and which force value and test limits will be used.

Usage

The following function programs the PMU force current value for all PMUs. Range selection is implicit:

```
void ipar_force( double force_value );
```

The following function programs the PMU force current value for one PMU. Range selection is implicit:

```
void ipar_force( double force_value, DutPin *pDutPin );
```

The following function programs the PMU force current value for one or more PMU(s). Range selection is implicit:

```
void ipar_force( double force_value, PinList* pPinList );
```

The following function programs the PMU force current value and current range for all PMUs:

```
void ipar_force( double force_value, Range range );
```

The following function programs the PMU force current value and current range for one PMU:

```
void ipar_force(  double force_value,
                  DutPin *pDutPin,
                  Range range );
```

The following function programs the PMU force current value and current range for one or more PMU(s):

```
void ipar_force(  double force_value,
                  PinList* pPinList,
                  Range range );
```

The following function will return the currently programmed PMU force current value for one PMU:

```
double ipar_force( DutPin *pDutPin );
```

where:

`force_value` specifies the desired force current value. Units may be used (see Specifying Units).

`pDutPin` is used in two contexts:

- In the setter functions, identifies one PMU to be programmed. In Multi-DUT Test Programs, the PMUs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.
- In the getter functions, identifies one PMU to be read.

`pPinList` specifies which PMU(s) are to be programmed. In Multi-DUT Test Programs, the PMUs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **`pPinList`** must only contains pins mapped to signal pins in the Pin Assignment Table.

`range` is used to explicitly select the force current range. Legal values are of the `Range` enumerated type and are used as follows:

Table 3.13.4.0-1 PMU Force Current Ranges

Current Range	LSB	Range
±2uA	1nA	range1
±20uA	10nA	range2
±200uA	100nA	range3
±2mA	1uA	range4
±20mA	10uA	range5

The getter version of `ipar_force()` returns the currently programmed force current value for one PMU. In Multi-DUT Test Programs, the value is retrieved from the first DUT in the Active DUTs Set (ADS).

Examples

```
ipar_force ( -100 UA );
ipar_force ( -100 UA, D0 );
ipar_force ( -100 UA, myPinList );
double value = ipar_force( D0 );
```

3.13.5 PMU Current Test Limit Functions

See Parametric Measurement Unit (PMU), PMU Functions, Overview.

Definition

The `ipar_high()` or `ipar_low()` functions are used to set or get the PMU current test PASS/FAIL test limits. These limits apply when using the PMU to force-voltage and test or measure current.

Note the following:

- All PMU voltages and currents are set to 0V/0A during the initial program load, *except* for the PMU voltage clamps, which are set to maximum values, effectively disabling both voltage clamps.
- During Sequence & Binning Table execution, PMU voltage and current values are managed by user-written C-code.
- When Sequence & Binning Table execution stops, all PMUs are set to 0V/0A and disconnected from all pins. This is done in the `builtin_after_testing_block`.
- PMU current test limits have multiple ranges. See Usage. By default, the range is implicitly selected, by the system software, based on the current test limit values programmed using `ipar_high()` and `ipar_low()`. Optionally, an explicit range can be specified using the `range` argument to `ipar_high()` or `ipar_low()`. When a range is not explicitly specified the system software selects the most accurate range based on the combination of test limits programmed. If the high and low test limits fall into different ranges, the system software selects the lower resolution range, but only if a double-ended test such as `pass_nicl` is being performed.
- Both test limits can be modified from a test pattern. See Controlling PE Levels from the Test Pattern.
- The range value must be explicitly specified if either PMU current test limit will be set or modified from an executing test pattern. See Controlling PE Levels from the Test Pattern.
- Programming the PMU current test limits value has no effect on a previously programmed voltage test limit, and vice versa. The actual use of these values, in hardware, does not occur until a PMU test is executed (using PMU Static Test Functions or PMU Dynamic Test Functions).

- These test limits are not required to be symmetric around zero.

Note: a commonly made mistake is to assume that programming a PMU force value or PASS/FAIL test limits also defines the type of PMU test which will execute next. It is the arguments passed to `partest()` or `ac_partest()` which define the type of test the PMU will perform (force current/measure voltage, etc.), and which force value and test limits will be used.

Usage

The following function programs the PMU current test high/low test limit for all PMUs. Range selection is implicit. In Multi-DUT Test Programs, the PMU of each DUT currently in the Active DUTs Set (ADS) are affected:

```
void ipar_high( double value );
void ipar_low( double value );
```

The following function programs the PMU current test high/low test limit for one PMU. Range selection is implicit:

```
void ipar_high(double value,
               DutPin *pDutPin );
void ipar_low( double value,
               DutPin *pDutPin );
```

The following function programs the PMU current test high/low test limit for one or more PMU(s). Range selection is implicit:

```
void ipar_high(double value,
               PinList* pPinList );
void ipar_low( double value,
               PinList* pPinList );
```

The following function programs the PMU current test high/low test limit and range for all PMUs. In Multi-DUT Test Programs, the PMUs of each DUT currently in the Active DUTs Set (ADS) are affected:

```
void ipar_high( double value, Range range );
void ipar_low( double value, Range range );
```

The following function programs the PMUs current test high/low test limit and current range for one PMU:

```
void ipar_high(   double value,
                  Range range,
                  DutPin *pDutPin );
void ipar_low(    double value,
                  Range range,
                  DutPin *pDutPin );
```

The following function programs the PMUs current test high/low test limit and current range for one or more PMU(s):

```
void ipar_high(   double value,
                  Range range,
                  PinList* pPinList );
void ipar_low(    double value,
                  Range range,
                  PinList* pPinList );
```

The following function will return the currently programmed PMUs current test high test limit or low test limit value for one PMU:

```
double ipar_high( DutPin *pDutPin );
double ipar_low( DutPin *pDutPin );
```

where:

`value` specifies the current value. Units may be used (see Specifying Units).

`pDutPin` is used in two contexts:

- In the setter functions, identifies one PMUs to be programmed. In Multi-DUT Test Programs, the PMUss of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.
- In the getter functions, identifies one PMUs to be read.

`pPinList` identifies which PMU(s) are to be programmed. In Multi-DUT Test Programs, the PMUss of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **`pPinList`** must only contains pins mapped to signal pins in the Pin Assignment Table.

range is used to explicitly select a PMU current range. Legal values are of the Range enumerated type and are used as follows:

Table 3.13.5.0-1 PMU Current Test Limit Ranges

Current Range	LSB	Range
±2uA	1nA	range1
±20uA	10nA	range2
±200uA	100nA	range3
±2mA	1uA	range4
±20mA	10uA	range5

The getter versions of `ipar_high()` and `ipar_low()` return the currently programmed current test high test limit or low test limit value for one PMU. In Multi-DUT Test Programs, the value is retrieved from the first DUT in the Active DUTs Set (ADS).

Examples

```
ipar_high( 3.2 MA );
ipar_low( 0.1 MA );
ipar_high( 3.2 MA, D0 );
ipar_low( 0.1 MA, D0 );
ipar_high( 3.2 MA, myPinList );
ipar_low( 0.1 MA, myPinList );
double high_val = ipar_high( D0 );
double low_val = ipar_low( D0 );
```

3.13.6 PMU Force Voltage Functions

See Parametric Measurement Unit (PMU), PMU Functions, Overview.

Definition

The `vpar_force()` function is used to set or get the PMU force voltage value. Note the following:

- All PMU voltages and currents are set to 0V/0A during the initial program load, *except* for the PMU voltage clamps, which are set to maximum values, effectively disabling both voltage clamps.
- During Sequence & Binning Table execution, PMU voltage and current values are managed by user-written C-code.
- When Sequence & Binning Table execution stops, all PMUs are set to 0V/0A and disconnected from all pins. This is done in the `builtin_after_testing_block`.
- The PMU force voltage has a single range but the minimum/maximum value is limited based on the type of pins being programmed (signal pins, DPS pins, HV pins). See Uage.
- Programming the PMU force voltage value has no effect on a previously programmed force current, and vice versa. The actual use of these values, in hardware, does not occur until a PMU test is executed (using PMU Static Test Functions or PMU Dynamic Test Functions), or until explicit PMU connections are made using PMU Connect/Disconnect Functions.
- The PMU force voltage can be modified from a test pattern. See Controlling PE Levels from the Test Pattern.
- It is possible to change the PMU force voltage value while the PMU is statically connected as a current source.

Note: the programmed `vclamp()` value *will* limit the PMU force voltage. A warning message will be displayed when the user's test program attempts to program this condition.

Note: a commonly made mistake is to assume that programming a PMU force value or PASS/FAIL test limits also defines the type of PMU test which will execute next. It is the arguments passed to `partest()` or `ac_partest()` which define the type of test the PMU will perform (force current/measure voltage, etc.), and which force value and test limits will be used.

Usage

The following function programs the PMU force voltage for all PMUs:

```
void vpar_force( double force_value );
```

The following function programs the PMU force voltage value for one PMU:

```
void vpar_force( double force_value, DutPin *pDutPin );
```

The following function programs the PMU force voltage value for one or more PMU(s):

```
void vpar_force( double force_value, PinList* pPinList );
```

The following function will return the currently programmed PMU force voltage for one PMU:

```
double vpar_force( DutPin *pDutPin );
```

where:

force_value specifies the PMU force voltage value. Units may be used (see Specifying Units). Legal values are:

Table 3.13.6.0-1 PMU Force Voltage Range

Range	LSB	Max Current	
-2.5V to +12.75V	1mV	±20mA	On PE Pins
-5V to +15V			On DPS pins
-2.5V to +15V			On HV pins
Note: while this may seem to be two voltage ranges, in hardware only one range exists. The system software will limit the force voltage if/when the PMU is connected to, or testing PE pins. The broader voltage output capabilies are only usable when the PMU is connected to replace a DUT Power Supply (DPS) (see PMU: Testing DPS Pins) or High Voltage Source/Measure Unit (HV) (see PMU: Testing HV Pins).			

pDutPin is used in two contexts:

- In the setter functions, identifies one PMU to be programmed. In Multi-DUT Test Programs, the PMUs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified DutPin must be mapped to a signal pin in the Pin Assignment Table.
- In the getter functions, identifies one PMU to be read.

pPinList specifies which PMU(s) are to be programmed. In Multi-DUT Test Programs, the PMUs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **pPinList** must only contains pins mapped to signal pins in the Pin Assignment Table.

The getter version of `vpar_force()` returns the currently programmed force voltage value for one PMU. In Multi-DUT Test Programs, the value is retrieved from the first DUT in the Active DUTs Set (ADS).

Examples

```
vpar_force( 7 V );
vpar_force( 7 V, D0);
vpar_force( 7 V, myPinList );
double val = vpar_force( D0 );
```

3.13.7 PMU Voltage Test Limit Functions

See Per-pin Parametric Test Unit (PTU), PMU Functions, Overview.

Definition

The `vpar_high()` or `vpar_low()` functions are used to set or get the PMU voltage test PASS/FAIL test limits. These limits apply when using the PMU to force-current and test or measure voltage.

Note the following:

- All PMU voltages and currents are set to 0V/0A during the initial program load, *except* for the PMU voltage clamps, which are set to maximum values, effectively disabling both voltage clamps.
- During Sequence & Binning Table execution, PMU voltage and current values are managed by user-written C-code.
- When Sequence & Binning Table execution stops, all PMUs are set to 0V/0A and disconnected from all pins. This is done in the `builtin_after_testing_block`.
- The PMU voltage PASS/FAIL test limits have multiple ranges, see Usage. The minimum/maximum values are also limited based on the type of pins being programmed (signal pins, DPS pins, HV pins). By default, the range is implicitly selected, by the system software, based on the voltage test limit values programmed using `vpar_high()` and `vpar_low()`. Optionally, an explicit range can be specified using the `range` argument to `vpar_high()` and `vpar_low()`. When a range is not explicitly specified the system software selects the most

accurate range based on the combination of test limits programmed. If the high and low test limits fall into different ranges, the system software selects the lower resolution range, but only if a double-ended test such as `pass_nivl` is being performed.

- Both test limits can be modified from a test pattern. See Controlling PE Levels from the Test Pattern.
- The range value must be explicitly specified if either PMU voltage test limit will be set or modified from the test pattern (see Controlling PE Levels from the Test Pattern).
- Programming the PMU voltage test limits value has no effect on a previously programmed current test limit, and vice versa. The actual use of these values, in hardware, does not occur until a PMU test is executed (using PMU Static Test Functions or PMU Dynamic Test Functions).
- These test limits are not required to be symmetric around zero.

Note: the programmed `vclamp()` value *will* limit the PMU voltage test limits. A warning message will be displayed when the user's test program attempts to program this condition.

Note: a commonly made mistake is to assume that programming a PMU force value or PASS/FAIL test limits also defines the type of PMU test which will execute next. It is the arguments passed to `partest()` or `ac_partest()` which define the type of test the PMU will perform (force current/measure voltage, etc.), and which force value and test limits will be used.

Usage

The following function programs the PMU voltage test high/low test limit value for all PMUs. Range selection is implicit. In Multi-DUT Test Programs, the PMU of each DUT currently in the Active DUTs Set (ADS) are affected:

```
void vpar_high( double value );
void vpar_low( double value );
```

The following function programs the PMU voltage test high/low test limit for one PMU. Range selection is implicit:

```
void vpar_high(double value,
               DutPin *pDutPin );
```

```
void vpar_low( double value,
               DutPin *pDutPin );
```

The following function programs the PMU voltage test high/low test limit for one or more PMU(s). Range selection is implicit:

```
void vpar_high(double value,
               PinList* pPinList );
void vpar_low( double value,
               PinList* pPinList );
```

The following function programs the PMU voltage test high/low test limit and range for all PMUs. In Multi-DUT Test Programs, the PMU of each DUT currently in the Active DUTs Set (ADS) are affected:

```
void vpar_high( double value, Range range );
void vpar_low( double value, Range range );
```

The following function programs the PMU voltage test high/low test limit and range for one PMU:

```
void vpar_high(double value,
               Range range,
               DutPin *pDutPin );
void vpar_low( double value,
               Range range,
               DutPin *pDutPin );
```

The following function programs the PMU voltage test high/low test limit and range for one or more PMU(s):

```
void vpar_high(double value,
               Range range,
               PinList* pPinList );
void vpar_low( double value,
               Range range,
               PinList* pPinList );
```

The following function will return the currently programmed PMU voltage test high/low test limit for one PMU:

```
double vpar_high( DutPin *pDutPin );
double vpar_low( DutPin *pDutPin );
```

where:

value specifies the test limit voltage value. Units may be used (see Specifying Units). Legal values are:

Table 3.13.7.0-1 PMU Measure Voltage Ranges

Range	Measure LSB	Range	
-2.5V to +4V	1mV	range1	On PE Pins
-2.5V to +12.75V	4mV	range2	
-5V to +15V	4mV	range2	On DPS Pins. See PMU: Testing DPS Pins.
-2.5V to +15V	4mV	range2	On HV Pins. See PMU: Testing HV Pins.

pDutPin is used in two contexts:

- In the setter functions, identifies one PMU to be programmed. In Multi-DUT Test Programs, the PMUs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.
- In the getter functions, identifies one PMU to be read.

pPinList specifies which PMU(s) are to be programmed. In Multi-DUT Test Programs, the PMUs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **pPinList** must only contains pins mapped to signal pins in the Pin Assignment Table.

range is used to explicitly select the test limit voltage range. Legal values are of the `Range` enumerated type. See Description.

The getter versions of `vpar_high()` and `vpar_low()` return the currently programmed voltage test high/low test limit for one PMU. In Multi-DUT Test Programs, the value is retrieved from the first DUT in the Active DUTs Set (ADS).

Example

```
vpar_high( 3.2 V );
vpar_low( 800 MV );
vpar_high( 3.2 V, D0 );
vpar_low( 800 MV, D0 );
```

```
vpar_high( 3.2 V, myPinList );
vpar_low( 800 MV, myPinList );
double high_val = vpar_high( D0 );
double low_val = vpar_low( D0 );
```

3.13.8 PMU Voltage Clamp Functions

See Parametric Measurement Unit (PMU), PMU Functions, Overview.

Definition

The `vclamp()` function is used to set the two PMU voltage clamp values.

The `positive_clamp()` and `negative_clamp()` functions are used to get the currently programmed values.

When performing a PMU force current test, the voltage clamps limit the compliance of the PMU, with independently programmable high and low clamp limits. This limits the voltage output by the PMU, to protect the DUT. And, as noted below, if carelessly programmed the voltage clamps can also affect subsequent PMU test results.

Note the following:

- All PMU voltages and currents are set to 0V/0A during the initial program load, *except* for the PMU voltage clamps, which are set to maximum values, effectively disabling both voltage clamps.
- During Sequence & Binning Table execution, PMU voltage and current values are managed by user-written C-code.
- When Sequence & Binning Table execution stops, all PMUs are set to 0V/0A and disconnected from all pins. This is done in the `builtin_after_testing_block`.
- The PMU voltage clamps have a single range, see Usage.
- It is possible to program the negative PMU voltage clamp to a positive voltage, but this is not useful. It is possible to program the positive PMU voltage clamp to a negative voltage, but this is not useful.
- It is illegal to program the negative voltage clamp to a value greater than the positive voltage clamp.

- The PMU voltage clamp values can be modified from a test pattern. See Controlling PE Levels from the Test Pattern.

Note: the programmed `vclamp()` value can affect the PMU force voltage set using `vpar_force()` and PASS/FAIL limits set using `vpar_high()` and `vpar_low()`. When the user's code attempts to program this situation a warning message is generated.

Usage

The following function programs both PMU voltage clamps for all PMUs. In Multi-DUT Test Programs, only PMUs of DUTs currently in the Active DUTs Set (ADS) are affected:

```
void vclamp( double positive_clamp, double negative_clamp );
```

The following function programs both PMU voltage clamps for one PMU:

```
void vclamp( double positive_clamp,
             double negative_clamp,
             DutPin *pDutPin );
```

The following function programs both PMU voltage clamps for one or more PMU(s):

```
void vclamp( double positive_clamp,
             double negative_clamp,
             PinList* pPinList );
```

The following function will return the currently programmed PMU positive voltage clamp value for one PMU:

```
double positive_clamp( DutPin *pDutPin );
```

The following function will return the currently programmed PMU negative voltage clamp value for one PMU:

```
double negative_clamp( DutPin *pDutPin );
```

where:

positive_clamp and **negative_clamp** specify the positive and negative voltage clamp value. Units may be used (see Specifying Units). Legal values are:

Table 3.13.8.0-1 PMU Voltage Clamp Range

Range	LSB	
-5V to +16V	100mV	Positive PMU Voltage Clamp
-6V to +15V		Negative PMU Voltage Clamp

pDutPin is used in two contexts:

- In the setter functions, identifies one PMU to be programmed. In Multi-DUT Test Programs, the PMU of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.
- In the getter functions, identifies one PMU to be read.

pPinList specifies which PMU(s) are to be programmed. In Multi-DUT Test Programs, the PMUs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **pPinList** must only contains pins mapped to signal pins in the Pin Assignment Table.

`positive_clamp()` and `negative_clamp()` return the currently programmed clamp value for one PMU. In Multi-DUT Test Programs, the value is retrieved from the first DUT in the Active DUTs Set (ADS).

Examples

In this example, `vclamp()` limits the maximum voltage the PMU can output, the PASS/FAIL voltage test limits, and any PMU voltage measurements to +8V.

```
vclamp(8.0 V, 0 V );
vclamp(8.0 V, 0 V, D0 );
vclamp(8.0 V, 0 V, myPinList );
double pos_val = positive_clamp( D0 );
double neg_val = negative_clamp( D0 );
```

3.13.9 Background Voltage Functions

See Parametric Measurement Unit (PMU), PMU Functions, Overview.

Definition

The `back_voltage()` function is used to set or get the Parametric Background Voltage value.

The `back_voltage_enable()` function is used to globally enable or disable the background voltage facility.

When sequentially testing multiple pins using a static PMU test (see PMU Static Test Functions), one pin is tested at a time and the other pins can optionally be forced to a user specified background voltage. This can greatly simplify continuity tests and leakage tests. See Parametric Background Voltage.

To use the background voltage feature requires the following:

- Globally enable the background voltage facility using `back_voltage_enable()`. By default, the background voltage is disabled.
- Program the desired background voltage value using `back_voltage()`.
- Execute a static PMU test (see PMU Static Test Functions), to sequentially test multiple signal pins. The background voltage is not used when using the PMU to test DPS pins (see PMU: Testing DPS Pins) or HV pins (see PMU: Testing HV Pins).

Also note the following:

- The background voltage facility is enabled or disabled globally, i.e. the enable state is not programmable per-pin, per-PMU, etc.
- The background voltage value can be set globally or on a per-pin basis.
- Changing the background voltage value does not affect the enable/disable state of the background voltage facility, and vice versa.
- During the static PMU test, any/all DUT pins not specified in the pin list being tested are left in the state to which they were last set by the test program, and are unaffected by the background voltage.
- All PMU voltages and currents are set to 0V/0A during the initial program load, *except* for the PMU voltage clamps, which are set to maximum values, effectively disabling both voltage clamps.
- During Sequence & Binning Table execution, PMU voltage and current values are managed by user-written C-code.
- When Sequence & Binning Table execution stops, all PMUs are set to 0V/0A and disconnected from all pins. This is done in the `builtin_after_testing_block`.

- The background voltage has a single voltage range, see Usage.

Using Magnum 1/2/2x, the background voltage is generated by the Per-pin Parametric Test Unit (PTU), and is used both as noted above for PMU tests and when executing PTU Static Test Functions to test multiple pins sequentially (the default is parallel). When the background voltage is enabled, the PTU connection switches of all pins in the pin list being tested are switched to connect the background voltage to each pin (the PE driver is disconnected from these pins). Then, for sequential PMU tests, one pin at a time is switched to PMU and tested or measured, leaving the other pins connected to the background voltage. For sequential PTU tests, the PTU for one pin at a time is programmed to the appropriate test value and tested or measured, leaving the other pins connected to the background voltage. After each pin is tested it is switched back to the background voltage before the next pin is tested. The process repeats until all pins in the pin list are tested.

Note: when the Per-pin Parametric Test Unit (PTU) supplies the background voltage it is set to operate on the ±2mA range. The actual background voltage output can be affected by the current supplied. See PTU Operating Area.

With background voltages *enabled*, as the test executes:

- Those pins in the pin list being tested which are not currently connected to the PMU will be set to the background voltage.
- Pins not included in the pin list being tested are not affected.

Usage

The following function globally enables or disables the background voltage facility:

```
void back_voltage_enable( BOOL state );
```

The following function returns the current background voltage enable state:

```
BOOL back_voltage_enable();
```

The following function is used to set the background voltage value, for all PMUs. In Multi-DUT Test Programs, only PMUs of DUTs currently in the Active DUTs Set (ADS) are affected:

```
void back_voltage( double value );
```

The following function programs the background voltage value for one PMU:

```
void back_voltage( double value, DutPin *pDutPin );
```

The following function programs the background voltage value for one or more PMU(s):

```
void back_voltage( double value, PinList* pPinList );
```

The following function will return the currently programmed background voltage value for one PMU:

```
double back_voltage( DutPin *pDutPin );
```

where:

state specifies whether the background voltage is enabled (TRUE) or disabled (FALSE).

value specifies the background voltage value. Units may be used (see Specifying Units). Legal values are:

Table 3.13.9.0-1 Background Voltage Range

Voltage Range	LSB
0V to +12V	1mV

pDutPin is used in two contexts:

- In the setter functions, identifies one PMU to be programmed. In Multi-DUT Test Programs, the PMUs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.
- In the getter functions, identifies one PMU to be read.

pPinList specifies which PMU(s) are to be programmed. In Multi-DUT Test Programs, the PMUs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **pPinList** must only contains pins mapped to signal pins in the Pin Assignment Table.

The getter version of `back_voltage_enable()` returns the current background voltage enable state.

The getter version of `back_voltage()` returns the currently programmed background voltage, for one PMU. In Multi-DUT Test Programs, the value is retrieved from the first DUT in the Active DUTs Set (ADS).

Examples

```
back_voltage ( 0 V );
back_voltage ( 0 V, D0 );
back_voltage ( 0 V, myPinList );

back_voltage_enable ( TRUE );
if ( back_voltage_enable() )
   output("Background voltage is ENABLED");
```

3.13.10 PMU Static Test Functions

See Overview, Parametric Measurement Unit (PMU), Static DC Tests.

Description

The `partest()` function is used to execute a static PMU test. PMU Dynamic Test Functions are documented separately.

See Static DC Tests.

Also note the following:

- It is the `pass_cond` argument to `partest()` which determines whether the PMU will force current or voltage, and which PASS/FAIL test limits are used.
- Static PMU tests are performed on the DUT pins specified in the pin list or `pDutPin` argument to the `partest()` function, which can contain signal pins, HV pins, or DPS pins, but only one type at a time.
- When testing multiple signal pins, by default, the test is executed sequentially, testing one pin at a time, in the order they occur in the specified pin list. The parallel options (there are 2) only apply when testing signal pins. This is controlled by an argument to `partest()`.
- When testing DPS pins, each PMU supports 2 DPS outputs. This means that when testing or measuring DPS output current it may not always be possible to test all DPS pins concurrently. When the `pPinList` argument contains 2 DPS pins which share a given PMU, the test will be performed in two steps, testing one DPS output at a time. See PMU: Testing DPS Pins.
- When testing HV pins, each PMU supports 2 HV outputs. This means that when testing or measuring HV output current or voltage it may not always be possible to test all HV pins concurrently. When the `pPinList` argument contains 2 HV pins which share a given PMU, the test will be performed in two steps, testing one HV at a time. See PMU: Testing HV Pins.
- All pins which are not specified in the `pPinList` argument or the `pDutPin` argument remain connected to the tester hardware in the state to which they were last set by the test program.

`partest()` supports measurement averaging (first available in software release h1.1.23). Note the following:

- This information applies when `measure()` = TRUE.

- Measurement averaging is enabled by including the `PartestOpt iacc` argument.
- The number of measurements made to obtain the average is set using `iacc_count_set()`. Default = 10.
- The value set using `iacc_count_set()` is only used when `measure()` = TRUE.
- When averaging is enabled the measurement average is compared to the PASS/FAIL test limits, set using PMU Current Test Limit Functions or PMU Voltage Test Limit Functions, to determine whether `partest()` returns PASS or FAIL.
- When Retrieving DC Test Results only the average value is returned, regardless of the number of measurements made.

PMU Test Checklist

- Set the PMU force current or voltage. See PMU Force Current Functions and PMU Force Voltage Functions.
- Set the PASS/FAIL test limits. See PMU Current Test Limit Functions and PMU Voltage Test Limit Functions.
- If performing a force-current test, the PMU voltage clamps should be set. See PMU Voltage Clamp Functions.
- If performing a sequential test of multiple pins the background voltage may be useful. See Background Voltage Functions.
- Settling time in addition to the Built-in Settling Time may be needed. See Parametric Settling Time. Don't forget to set it back to 0 when done.
- The `measure()` function is used to switch between measurements and Go/NoGo tests.
- Don't forget DUT power, and any required functional set up. Static PMU tests do not execute a test pattern.
- Specific rules apply when testing pins which are statically connected to the PMU (which is not common). See PMU as Voltage/Current Source.
- Operation is somewhat more complex when using the PMU to test DPS pins. See PMU: Testing DPS Pins.
- Operation is somewhat more complex when using the PMU to test HV pins. See PMU: Testing HV Pins.
- After the test is complete, user code may retrieve test results for further processing, datalogging, etc. See Retrieving DC Test Results.

Note: the system software automatically controls connecting and disconnecting the PMU, or DPS, to DUT pins. It is ***not necessary*** for user code to do this.

Usage

```
BOOL partest( PassCond pass_cond, DutPin *pDutPin );
BOOL partest( PassCond pass_cond, PinList* pPinList );
BOOL partest(  PassCond pass_cond,
               DutPin *pDutPin,
               PartestOpt opt );
BOOL partest(PassCond pass_cond,
             PinList* pPinList,
             PartestOpt opt );
BOOL partest(  PassCond pass_cond,
               DutPin *pDutPin,
               PartestOpt type,
               PartestOpt accuracy );
BOOL partest(PassCond pass_cond,
             PinList*pPinList,
             PartestOpt type,
             PartestOpt accuracy );
```

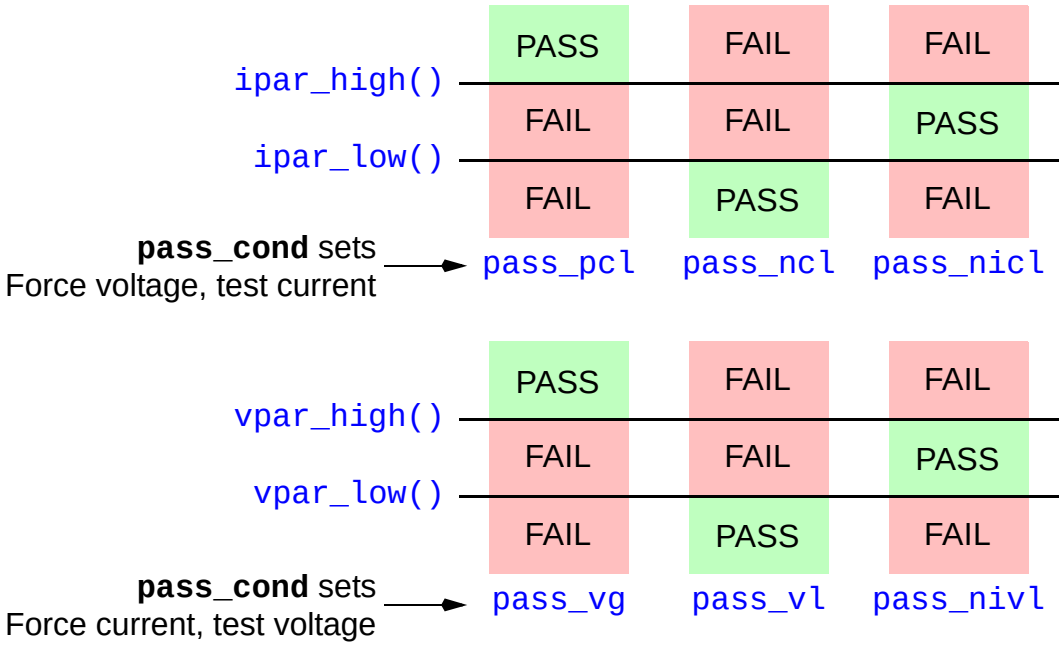
where:

pass_cond determines whether the PMU test will force current or voltage, which PASS/FAIL test limits are used and how they are applied. Legal values are of the PassCond enumerated type, and operate as described below:.

Table 3.13.10.0-1 Parametric Test Force & Pass/Fail Limit Specification

Pass Condition	Comments
pass_pcl	**pcl** = Positive Current Limit PMU forces voltage (vpar_force()). Test passes if the tested/measured current is greater than ipar_high().
pass_ncl	**ncl** = Negative Current Limit PMU forces voltage (vpar_force()). Test passes if the tested/measured current is less than ipar_low().
pass_nicl	**nicl** = Not In Current Limit PMU forces voltage (vpar_force()). Test passes if the tested/measured current is between ipar_low() and ipar_high().
pass_vg	**vg** = Voltage Greater Than PMU forces current (ipar_force()). Test passes if the tested/measured voltage is greater than vpar_high().
pass_vl	**vl** = Voltage Less Than PMU forces current (ipar_force()). Test passes if the tested/measured voltage is less than vpar_low().
pass_nivl	**nivl** = Not In Voltage Limit PMU forces current (ipar_force()). Test passes if the tested/measured voltage is between vpar_low() and vpar_high().

The diagram below shows this operation graphically:



pDutPin identifies one pin to be tested. In Multi-DUT Test Programs, the pins of each DUT currently in the Active DUTs Set (ADS) are tested. **pDutPin** may be a signal pin, HV pin or DPS pin.

pPinList specifies the pin(s) to test. In serial mode tests (see **opt** and **type** below), the pins are tested in the order they are listed in the pin list. The **pinlist** may only include one type of pin: signal pins, HV pins, or DPS pins. In Multi-DUT Test Programs, only pin(s) of DUTs currently in the Active DUTs Set (ADS) are tested.

opt is optional and, if specified, controls one of two PMU test options. Legal values are of the PartestOpt enumerated type:

Table 3.13.10.0-2 Parametric Test Optional Arguments

Optional Arguments	Comments
sequential	Default. The pins in **pPinList** are tested sequentially, in the order listed in the pin list. PMU sense is done at the tester channel of each pin tested.
parallel	The pins in **pPinList** are tested in parallel. PMU sense is done at the tester channel of the first pin in **pPinList**. Only applies when testing signal pins.
parallel_pmu	The pins in **pPinList** are tested in parallel. PMU sense is done at the PMU. Only applies when testing signal pins.
iacc	Enable measure averaging. Applies only when measure() = TRUE. See Description and Measurement Average Count Functions. When iacc is specified alone the test uses the sequential option.
no_iacc	Default. Disable measure averaging.

To specify both options it is necessary to use both the **type** and **accuracy** arguments.

- **type** specifies the execution option (sequential, parallel, parallel_pmu).
- **accuracy** specifies iacc.

As noted above, iacc is only usable when measure() = TRUE.

The **parallel** and **parallel_pmu** options are used to test a group of signal pins in parallel. A common application is testing leakage current, using a PASS/FAIL specification for a single pin. Since most DUTs have significantly lower leakage than their specification, this type of test usually passes for good DUTs. If it fails, then a sequential leakage test is performed. Additional considerations exist when testing using **parallel** or **parallel_pmu**:

- If **pPinList** contains pins which share a given PMU multiple PMUs will automatically be used to perform the test, but only those pins which share a given PMU can be electrically tested in parallel. However, by default, the PASS/FAIL test limits are identical for each PMU. It is possible to program these test limits on a per PMU basis.

- When using the **parallel** option, PMU sensing is done based on the pins in the specified pin list. When multiple PMUs are involved, sensing is done at the tester channel of first *tested* pin in the pin list on each PMU.
- When using the **parallel** or **parallel_pmu** options with measure() = TRUE, a single measured value is recorded for the entire group of pins. When retrieving measured values only the value stored for the first pin in the pin list is valid; i.e. the measured value is invalid for the other pins in the pin list, and may be a value from a previous measurement. See Retrieving DC Test Results.

When Retrieving DC Test Results, the measured value obtained using the **iacc** option is the average of ten measurements.

Note: the iacc option is intended for use when forcing voltage and measuring current in the low current measurement ranges. The iacc option can however be specified for any range and either voltage or current force. Additionally, iacc can be specified simultaneously with either of the parallel options. Using iacc increases test time ~1mS for each pin tested.

---

partest() returns TRUE (PASS) or FALSE (FAIL). All pin(s) tested must PASS otherwise FAIL is returned. In Multi-DUT Test Programs, only DUT(s) in the Active DUTs Set (ADS) can affect test results.

Examples

Example 1:

The following example performs a static Go/Nogo PMU test on the pin list named Input_pins. If this pin list contains more than one pin which share a given PMU, each pin is tested sequentially. Since the PassCond specifies pass_nicl the PMU will force voltage (+5V) and test current. No additional settling time has been programmed (here). The test will pass if the current for each pin tested is between +1uA and +25nA:

```
measure( FALSE );   // Set Go/NoGo testing
vpar_force( 5 V );
ipar_high( 1 UA );
ipar_low( 25 NA );
BOOL result = partest( pass_nicl, Input_pins ); // Force V, test I
```

Example 2:

The following example performs a static Go/Nogo PMU test on the pin list named DBus. If this pin list contains more than one pin which share a given PMU these pins are all tested in parallel; i.e. electrically connected and tested as a group. Since the PassCond specifies

pass_nicl the PMU will force voltage and test current. An additional 2mS settling time has been programmed. The test will FAIL if the total current drawn by all pins which share a given PMU is more than ±1uA. PMU sensing at the first tested pin which share a given PMU:

```
measure( FALSE );   // Set Go/NoGo testing
vpar_force( 5 V );
ipar_high( 1 UA );
ipar_low( -1 UA );
partime ( 2 MS );   // Additional settling time
BOOL result = partest( pass_nicl, DBus, parallel );
```

Example 3:

The following example enables the measure() mode, and performs a sequential static PMU test on all pins in the pin list named IO_pins. The high accuracy option (iacc) is selected which results in 10 measurements being made for each pin tested, with the average compared to the test limits to determine if the test passes or fails.

```
measure( TRUE );    // Set measure mode testing
vpar_force( 5 V );
ipar_high( 50 NA );
ipar_low( -50 NA );
BOOL result = partest( pass_nicl, IO_pins, iacc );
```

Example 4:

The following example performs a PMU force voltage test. The force voltage, PASS/FAIL test limits, background voltage, parametric settling time, etc. are specified elsewhere in the program, which is very common!

```
BOOL result = partest( pass_nicl, Input_pins );
```

3.13.11 PMU Dynamic Test Functions

See Overview, Parametric Measurement Unit (PMU), Dynamic DC Tests.

Description

The ac_partest() function is used to execute a dynamic PMU test. PMU Static Test Functions are documented separately .

See Dynamic DC Tests.

Also note the following:

- Dynamic PMU test perform a DC test while a test pattern is executing (more below).
- It is the `pass_cond` argument to `ac_partest()` which determines whether the PMU will force current or voltage, and which PASS/FAIL test limits are used.
- Dynamic PMU tests are performed on the DUT pins specified in the pin list or `pDutPin` argument to `ac_partest()`, which can contain signal pins, HV pins, or DPS pins, but only one type at a time.
- When testing signal pins, unlike static PMU tests, dynamic PMU tests do not support a sequential test mode. During dynamic PMU tests, before the test pattern is executed, all pins in the specified pin list which share a given PMU are electrically connected in parallel and connected to the PMU. This means that the PMU test is performed on all pins at the same time, and that pins which don't share a given PMU are tested by different PMU(s). In most applications, `ac_partest()` is used to test a single pin.
- When testing DPS pins, each PMU supports 2 DPS outputs. This means that when testing or measuring DPS output current it may not always be possible to test all DPS pins concurrently. When the `pPinList` argument contains 2 DPS pins which share a given PMU, the test will be performed in two steps, testing one DPS output at a time. This will require executing the test pattern two times. See PMU: Testing HV Pins.
- When testing HV pins, each PMU supports 2 HV outputs. This means that when testing or measuring HV output current or voltage it may not always be possible to test all HV pins concurrently. When the `pPinList` argument contains 2 HV pins which share a given PMU, the test will be performed in two steps, testing one HV at a time. This will require executing the test pattern two times. See PMU: Testing HV Pins.
- All pins which are not specified in the `pPinList` argument or the `pDutPin` argument remain connected to the tester hardware in the state to which they were last set by the test program.

As indicated, a dynamic PMU test executes a test pattern. Note the following:

- A pattern execution stop option must be specified, Options include terminating the pattern if there is a functional failure or executing the pattern to completion, regardless of functional failures.

- When the test pattern terminates, the system software interrogates the pin electronics error latches checking for functional failures and the DC Error Flags checking for DC failures. If ***either*** of these indicate a failure, then the `ac_partest()` function returns FAIL.

Note: in dynamic DC parametric tests, the test pattern can perform branch-on-error based on the state of the DC Error Flag in each DC Test and Measure System. The `MAR RESET` or `CHIPS RESET` instructions (Memory Test Patterns) or `VEC/RPT RESET`, `VAR RESET` and `VPINFUNC RESET` instructions (Logic Test Patterns) will clear all DC Error Flags, which can affect the overall PASS/FAIL result of the `hv_ac_test_supply()`.

The `ac_partest()`function has two trigger modes which determine whether test pattern triggers are enabled. This is controlled using the `comp_cond` argument. Four scenarios are possible:

- `comp_cond` = default (`no_vcomp`) and `measure()` = FALSE: the DC Comparators and Error Logic are used and enabled for the entire duration of the executing test pattern. If at any time during pattern execution the tested parameter fails the test returns FAIL.
- `comp_cond` = `vcomp` and `measure()` = FALSE: the DC Comparators and Error Logic are used and are triggered by the `VCOMP` token from the executing test pattern. If any one or more triggered samples fails the test returns FAIL.
- `comp_cond` = `vcomp` and `measure()` = TRUE: the DC A/D Converter is used and triggered by the `VCOMP` token from the executing test pattern. Each trigger causes a measurement to be made.
- `comp_cond` = default (`no_vcomp`) and `measure()` = TRUE. Not supported in DC tests. `ac_partest()` operates as though `measure()` = FALSE. Any DC measurements which are retrieved are invalid.

When using `comp_cond` = `vcomp`, one trigger will be generated in each test pattern cycle which contains the `MAR VCOMP` instruction (Memory Test Patterns) or `VEC/RPT VCOMP`, `VAR VCOMP` and `VPINFUNC VCOMP` instructions (Logic Test Patterns). See Dynamic DC Tests.

Note: when using vcomp, if the pattern does not generate any triggers, ac_partest() will return invalid results. If measure = FALSE, the DC Comparators and Error Logic will never be triggered, which will result in a PASS condition. If measure() = TRUE, any measured values will be invalid (old, stale, etc.) and the test may return PASS or FAIL based on the invalid measurement data. The system software cannot check for this error; i.e. it is the user's responsibility to ensure that at least one vcomp trigger is issued by any pattern used by ac_partest().

Note: whether using test pattern triggers or not, the DC Error Flag will affect test pattern's branch-on-error operations (proper pipelining is required for intentional use). The test pattern MAR RESET and CHIPS RESET instructions (Memory Test Patterns) and VEC/RPT RESET, VAR RESET and VPINFUNC RESET instructions (Logic Test Patterns) will clear all DC Error Flags. If measurements are disabled, after pattern execution stops, the DC Error Flag remains set the dynamic PMU test will return FAIL.

Note: the system software automatically controls connecting and disconnecting the PMU to DUT pins. It is ***not necessary*** for user code to do this.

PMU Test Checklist

- Set the PMU force current or voltage. See PMU Force Current Functions and PMU Force Voltage Functions.
- Set the PASS/FAIL test limits. See PMU Current Test Limit Functions and PMU Voltage Test Limit Functions.
- If performing a force-current test, the PMU voltage clamps should be set. See PMU Voltage Clamp Functions.
- Dynamic PMU tests do not utilize the background voltage.
- Setting a non-zero Parametric Settling Time is typically not useful (waste of test time) when performing dynamic PMU tests. See Parametric Settling Time.
- Don't forget DUT power. And, since functional failures will also cause ac_partest() to return FAIL the AC timing and PE levels are important.
- Specific rules apply when testing pins which are statically connected to the PMU (which is not common). See PMU as Voltage/Current Source.

- Additional rules apply when using the PMU to test DPS pins. See PMU: Testing DPS Pins.
- Additional rules apply when using the PMU to test HV pins. See PMU: Testing HV Pins.
- After the test is complete, user code may retrieve test results for further processing, datalogging, etc. See Retrieving DC Test Results.

Note: the system software automatically controls connecting and disconnecting the PMU, or DPS, to DUT pins. It is ***not necessary*** for user code to do this.

Usage

```
BOOL ac_partest(  PassCond pass_cond,
                  DutPin *pDutPin,
                  Pattern *pPattern,
                  PatStopCond stop_cond );
BOOL ac_partest(  PassCond pass_cond,
                  PinList* pPinList,
                  Pattern *pPattern,
                  PatStopCond stop_cond );
BOOL ac_partest(  PassCond pass_cond,
                  DutPin *pDutPin,
                  Pattern *pPattern,
                  PatStopCond stop_cond,
                  CompCond comp_cond );
BOOL ac_partest(  PassCond pass_cond,
                  PinList* pPinList,
                  Pattern *pPattern,
                  PatStopCond stop_cond,
                  CompCond comp_cond );
```

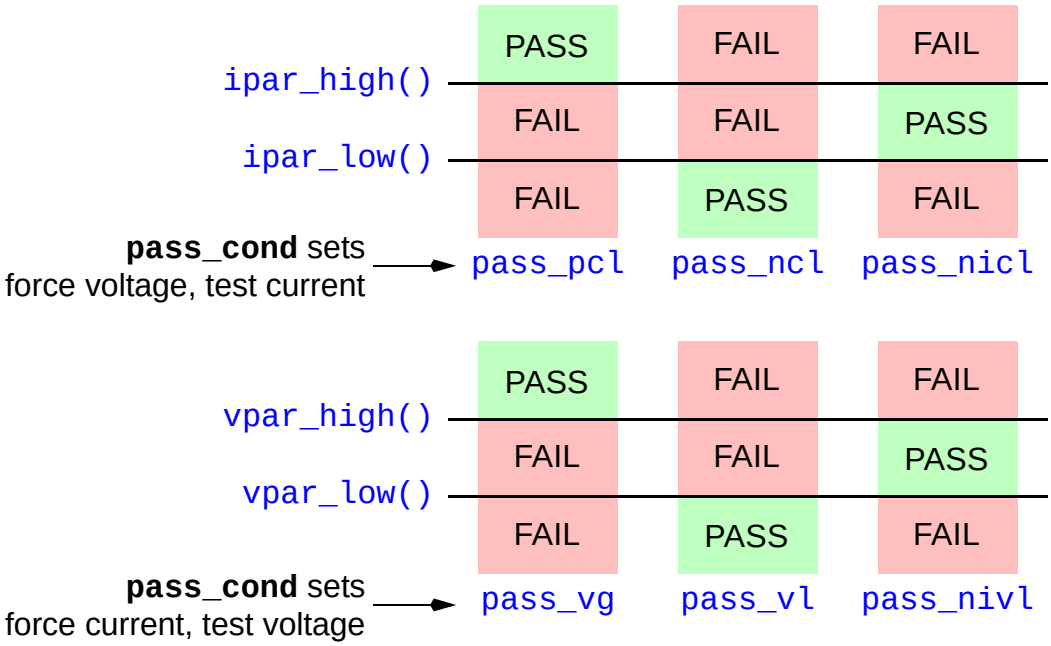
where:

pass_cond determines whether the PMU test will force current or voltage, which PASS/FAIL test limits are used and how they are applied. Legal values are of the PassCond enumerated type, and operate as described below:.

Table 3.13.11.0-1 Parametric Test Force & Pass/Fail Limit Specification

Pass Condition	Comments
pass_pcl	**pcl** = Positive Current Limit PMU forces voltage (vpar_force()). Test passes if the tested/ measured current is greater than ipar_high() AND all functional strobes PASS.
pass_ncl	**ncl** = Negative Current Limit PMU forces voltage (vpar_force()). Test passes if the tested/ measured current is less than ipar_low() AND all functional strobes PASS.
pass_nicl	**nicl** = Not In Current Limit PMU forces voltage (vpar_force()). Test passes if the tested/ measured current is between ipar_low() and ipar_high() AND all functional strobes PASS.
pass_vg	**vg** = Voltage Greater Than PMU forces current (ipar_force()). Test passes if the tested/ measured voltage is greater than vpar_high() AND all functional strobes PASS.
pass_vl	**vl** = Voltage Less Than PMU forces current (ipar_force()). Test passes if the tested/ measured voltage is less than vpar_low() AND all functional strobes PASS.
pass_nivl	**nivl** = Not In Voltage Limit PMU forces current (ipar_force()). Test passes if the tested/ measured voltage is between vpar_low() and vpar_high() AND all functional strobes PASS.

The diagram below shows this operation graphically:



pDutPin identifies one pin to be tested. In Multi-DUT Test Programs, the same pin of each DUT currently in the Active DUTs Set (ADS) is tested.

pPinList specifies the pin(s) to test. When testing signal pins, *all pins* in **pPinList** are connected in parallel before the pattern is executed, see Description. **pPinList** may only include one type of pin: signal pins, HV pins or DPS pins.

pPattern identifies the test pattern to be executed.

stop_cond specifies the pattern execution stop condition. Legal values are of the enumerated type, but only those in the table below are valid for this test:

Table 3.13.11.0-2 Pattern Execution Stop Condition Options

Stop Condition	Summary Description
finish	Execute pattern to completion, regardless of errors.
error	Stop pattern execution on first functional or DC Error Flag error and set the result of ac_partest() to FAIL.
fullec	Execute pattern to completion. Enable the ECR to capture errors during pattern execution. This argument should be used when performing Redundancy Analysis (RA) or using BitmapTool.
LEC_only_errors	Enable the ECR to capture errors during pattern execution. Capture the first 2Meg (2^{21}-6) failing vectors. Intended for use when the ECR is used as an Logic Error Catch (LEC).
LEC_first_vectors	Enable the ECR to capture errors during pattern execution. Capture the first 2Meg (2^{21}-6) vectors executed. Ignores PASS/FAIL. Intended for use when the ECR is used as an Logic Error Catch (LEC).
LEC_last_vectors	Enable the ECR to capture errors during pattern execution. Capture the last 2Meg (2^{21}-6) vectors executed. Ignores PASS/FAIL. Intended for use when the ECR is used as an Logic Error Catch (LEC).
LEC_before_error	Enable the ECR to capture errors during pattern execution. Capture the first failing vector plus the previous 2Meg (2^{21}-6) vectors executed. Intended for use when the ECR is used as an Logic Error Catch (LEC).

Table 3.13.11.0-2 Pattern Execution Stop Condition Options *(Continued)*

Stop Condition	Summary Description
LEC_after_error	Enable the ECR to capture errors during pattern execution. Capture the first failing vector plus the next 2Meg (2^{21}-6) vectors executed. Intended for use when the ECR is used as an Logic Error Catch (LEC). .
LEC_center_error	Enable the ECR to capture errors during pattern execution. Capture the first failing vector plus up to 512K vectors executed before the failure and up to 512K vectors executed after the failure. Intended for use when the ECR is used as an Logic Error Catch (LEC).
Note: in parallel test applications, the test pattern must be executed to completion, to ensure that DUT(s) which don't fail are completely tested. In other words, halting the pattern early (error) because one or more DUT(s) failed prevents DUT(s) which PASS from being completely tested. This is BAD.	

comp_cond is optional, and if used must be the reserved word vcomp. Including this argument enables test pattern triggers. See Description and Dynamic DC Tests for more details.

ac_partest() returns TRUE (PASS) or FALSE (FAIL). All pin(s) tested must PASS otherwise FAIL is returned. All latched functional strobes must PASS. In Multi-DUT Test Programs, only DUT(s) in the Active DUTs Set (ADS) can affect test results.

Examples

Example 1:

The following example performs a dynamic Go/Nogo PMU test on the pin list named Vcc (likely a DPS pin). Since the PassCond specifies pass_nicl the PMU will force voltage (+5V) and test current. The PASS/FAIL test limits are set to +10mA and +100uA. The test pattern named myPat is executed to completion. Since the vcomp option is not specified, the DC Comparators and Error Logic are enabled for the duration of the test pattern. Once pattern execution terminates, the DC Error Flags and the PE error latches are examined to determine if the test passed or failed. Failures in any of these will cause ac_partest() to return FAIL.

```
// Not shown are PE level setups and AC timing setup, both critical
// for proper functional test PASS/FAIL operation
vpar_force( 5 V );
ipar_high( 10 MA );
ipar_low( 100 UA );
int test_result = ac_partest( pass_nicl, VCC, myPat, finish );
```

Example 2:

This example performs a similar test, but since the `vcomp` option is specified, the DC Comparators and Error Logic are triggered in test pattern cycles which contain a VCOMP instruction. One DC trigger will occur for each pattern instruction which contains the `MAR VCOMP` instruction (memory patterns) or `VEC`/`RPT VCOMP`, `VAR VCOMP`, or `VPINFUNC VCOMP` instructions (logic patterns).

```
test_result = ac_partest(pass_nicl,
                         VCC,
                         myPat,
                         finish,
                         vcomp );
```

3.13.12 start_ac_partest(), stop_ac_partest()

See Overview, Parametric Measurement Unit (PMU).

Description

The `start_ac_partest()` and `stop_ac_partest()` functions enable a specialized AC PMU test capability, as described below.

The `start_ac_partest()` function does the following:

- Records the current connection state of the pins specified in the `pPinlist` or `pDutPin` argument.
- Sets up the appropriate PMUs to perform the specified type of PMU test (force voltage and test current or force current and test voltage).
- Connects the PMU to the pins in the specified `pPinlist` or `pDutPin` argument. Multiple PMUs may be involved.
- Starts execution of a specified test pattern (using `start_pattern()`), then immediately returns control to the user's test program code.

- Note: the start_ac_partest() function does not trigger a PMU test or measurement or otherwise control the test pattern. More below.

The stop_ac_partest() function does the following:

- Stops any executing test pattern, if any.
- Restores the PMU and tester pins to the configuration prior to executing start_ac_partest() (this is the same operation that occurs after performing a standard ac_partest()).

After executing start_ac_partest() the PMU can effectively perform a Go/NoGo test by having a test pattern trigger the DC Comparators and Error Logic using a VCOMP instruction. The test pattern can be that specified as the pPattern argument to the start_ac_partest() function or, assuming that pattern execution completed, any other functional pattern(s) executed between start_ac_partest() and stop_ac_partest(). Each occurrence of the MAR VCOMP instruction (Memory Test Patterns) or VEC/RPT VCOMP, VAR VCOMP, or VPINFUNC VCOMP instructions (Logic Test Patterns) will trigger the DC Comparators and Error Logic once, setting the DC Error Flag if the DC parameter being tested fails the selected PASS/FAIL limits.

As noted above, immediately after executing start_ac_partest(), control returns to the test program C-code. This allows the following:

- If the pattern executed by start_ac_partest() contains a MAR PAUSE instruction (Memory Test Patterns) or VAR PAUSE (Logic Test Patterns) user C-code can be executed before restart() is used to restart that pattern. This code can modify APG registers, counters, etc., or modify timing, pin/DPS voltages, etc.
- Once execution of the pattern started by start_ac_partest() completes, additional test patterns can be executed using funtest(). These patterns may also use VCOMP to trigger the DC Comparators and Error Logic.

There are some rules and limitations to what can be done between the start_ac_partest() and stop_ac_partest():

- start_ac_partest() and stop_ac_partest() must be used as a pair. That is, each start_ac_partest() must have a corresponding stop_ac_partest(). The system software will not allow another DC parametric test to execute if stop_ac_partest() has not been executed, and if detected, the error is fatal.
- In this mode, it is not possible to cause the PMU to make a measurement.

- After executing the `start_ac_partest()` function and before executing the `stop_ac_partest()` function, the PMU remains configured and connected to the specified pin(s). During this time, the only way to trigger the DC Comparators and Error Logic is from an executing test pattern, using the `VCOMP` instructions noted above.
- After executing the `start_ac_partest()` function and before executing the `stop_ac_partest()` function it is a fatal error to try to change the PMU force parameter *type*, high/low current test `PASS/FAIL` limits, or the pins connected to the PMU. It is OK to modify the voltage test `PASS/FAIL` limits (because no range changes are possible).
- After executing the `start_ac_partest()` function and before executing the `stop_ac_partest()` function it is legal to modify the PMU force *value*, from the test pattern, see Controlling PE Levels from the Test Pattern.
- After executing the `start_ac_partest()` function and before executing the `stop_ac_partest()` function it is possible to modify other voltages from user code or by Controlling PE Levels from the Test Pattern.
- The test pattern(s) can branch-on-error based on the current state of the DC Error Flag or the PE error flags. And, the DC Error Flag and PE error flags will be cleared using the `MAR RESET` and `CHIPS RESET` instructions (Memory Test Patterns) and `VEC/RPT RESET`, `VAR RESET` and `VPINFUNC RESET` instructions (Logic Test Patterns).
- Use APG counter(s) as flags if the `PASS/FAIL` results of a given `VCOMP` trigger must be known after the test pattern execution completes. In the pattern, increment the desired APG counter based on a branch-on-error execution decision. After pattern execution completes, read the desired counter(s) using the `count()` function.

Usage

```
void start_ac_partest( PassCond pass_cond,
                       DutPin *pDutPin,
                       Pattern *pPattern,
                       PatStopCond stop_cond,
                       CompCond comp_cond );
void start_ac_partest( PassCond pass_cond,
                       PinList* pPinList,
                       Pattern *pPattern,
                       PatStopCond stop_cond,
                       CompCond comp_cond );
```

```
void stop_ac_partest();
```

where:

pass_cond determines whether the PMU test is forcing current or voltage, and how the PASS/FAIL test limits are to be used. For details, see ac_partest().

pDutPin identifies one pin to be connected to a PMU. In Multi-DUT Test Programs, the same pin of each DUT currently in the Active DUTs Set (ADS) is tested.

pPinList specifies the pin(s) to be connected to a PMU. When testing signal pins, all pins in the pin list are connected in parallel before the pattern is executed, see Description. **pPinList** may only include one type of pin: signal pins, HV pins, or DPS pins. In Multi-DUT Test Programs, only pin(s) of DUT(s) currently in the Active DUTs Set (ADS) are tested.

pPattern identifies the test pattern to be started.

stop_cond specifies the pattern execution stop condition. For details, see ac_partest().

comp_cond must specify vcomp to enable the test pattern to trigger the DC Comparators and Error Logic. See Description.

3.13.13 ac_partest_results_store()

See Overview, Parametric Measurement Unit (PMU).

Note: first available in software release h3.4.xx.

The ac_partest_results_store() function is used to retrieve values measured by Parametric Measurement Unit (PMU) in the following specific situation:

- A functional test pattern is executed using start_ac_partest().
- The test pattern executes an instruction containing MAR VCOMP (Memory Test Patterns) or VEC/RPT VCOMP, VAR VCOMP or VPINFUNC VCOMP (Logic Test Patterns) to trigger a PMU measurement.
- measure() = TRUE.
- Pattern execution is paused by executing a MAR PAUSE (Memory Test Patterns) or VAR PAUSE (Logic Test Patterns) pattern instruction.
- User code must retrieve/save the just-measured value(s) before restarting pattern execution using restart(). More below.

- The test pattern contains at least one additional VCOMP trigger which, if executed, will over-write any previously measured values; i.e. the values retrieved above.

To retrieve the measured values requires using the combination of:

- `ac_partest_results_store()`, which reads the measured values from hardware and puts them into the system software's (private) PMU measurement data structure.
- `Pin_meas()` to retrieve these values allowing them to be stored in user-allocated data structures.

Executing `ac_partest_results_store()` after a test pattern ends normally (i.e. is not paused) is harmless.

Usage

```
void ac_partest_results_store();
```

Example

```
CArray<double,double> meas_results;

// Setup in preparation for pattern-triggered PMU measurement.
// Includes AC/DC/DPS setup, etc.
measure( TRUE );
start_ac_partest( pass_nivl, testPins, myPat, finish, vcomp );

// Execution will continue after myPat pauses.

// The following are repeated as necessary
ac_partest_results_store(); // Move values from HW to SW
Pin_meas( meas_results ); // Get values into user struct
// Do something with values...
restart(); // Presumably to trigger another PMU measurement
// Repeat previous 3 steps as desired.
```

3.13.14 PMU: Testing DPS Pins

See Overview, Parametric Measurement Unit (PMU).

Overview

Static PMU tests are executed using the `partest()` function. Dynamic PMU tests are executed using the `ac_partest()` function. Both functions require a pin list argument to specify which DUT pins are to be tested. This pin list must contain only one type of pin: signal pins, or HV pins, or DPS pins. This section describes operation when the pin list contains DPS pins.

The model below shows how the Magnum 1/2 PMU connects to a DPS pin:

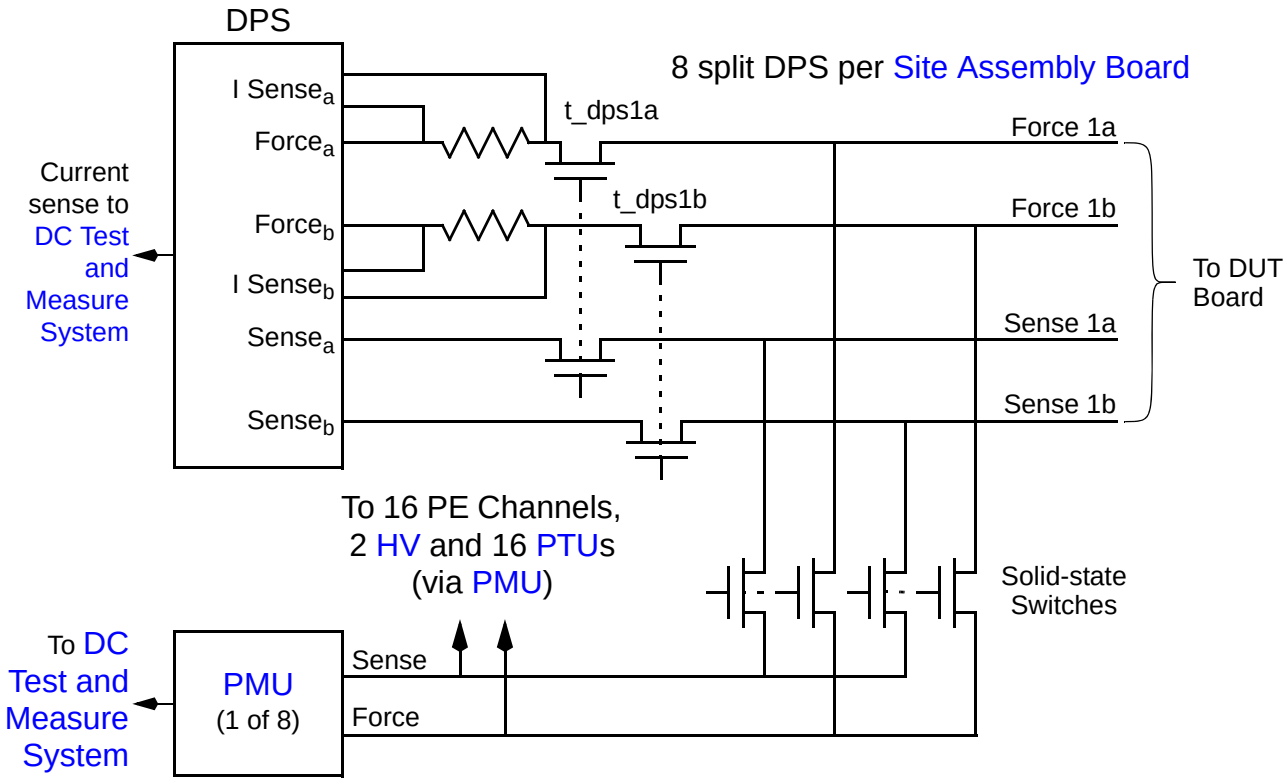


Figure-39: Magnum 1/2 PMU on DPS Block Diagram

When the pin list contains DPS pins the system software controls a sequence of events, which is different for Static PMU Tests on DPS Pins vs. Dynamic PMU Tests on DPS Pins, as noted below.

Note: the sequences below only apply when testing DPS pins using the PMU. In both static and dynamic tests, the connections between the DPS(s) and PMU are automatically controlled by the system software during the test; i.e. user code should NOT to do this.

Static PMU Tests on DPS Pins

Note: if a decoupling capacitor exists on the DPS pin(s) to be tested, the DPS will be used to pre-charge these pin(s) to the PMU force voltage. This is required because the PMU cannot drive highly capacitive nodes. The sequence below changes based on whether `pmu_comp_cap()` is set = 0, which indicates that DPS the pin(s) do NOT have any added capacitance.

Note: using Magnum 1 and Magnum 2, each DPS has 2 outputs (split DPS). When the DPS is configured in `t_dps_vpulse` mode (default, see DPS Output Mode) both outputs are derived from a single D/A converter (DAC) and will always be at the same voltage (and the test pattern VPULSE signal affects both outputs identically, see DPS Output Mode). In `t_dps_independent` mode the outputs may be at different voltages and VPULSE cannot be used. This is important here because the following sequence affects both outputs of a given DPS when it is configured in `t_dps_vpulse` mode. This does not apply to Magnum 2x or Maverick-I/-II.

1. The currently programmed DPS voltage of the pin(s) to be tested is saved. This step is skipped if `pmu_comp_cap()` = 0.
2. The DPS(s) on the pin(s) to be tested is programmed to the PMU force-voltage value (the value currently set by `vpar_force()`), to pre-charge these pin(s) to the desired PMU force voltage. See Note:. This step is skipped if `pmu_comp_cap()` = 0.
3. The PMU is set to force `vpar_force()`.
4. The PMU(s) are connected to the pin(s) to be tested, in parallel with the DPS.
5. The DPS is then disconnected from these pin(s). This leaves the PMU connected as the sole power source on the pin(s) being tested.
6. The PMU is set to force the programmed force value. This may change the PMU mode from force voltage to force current, as determined by the first argument to `partest()`.
7. The Built-in Settling Time time occurs.
8. The user programmed Parametric Settling Time occurs.
9. The PMU performs the specified Go/NoGo test or measurement (see `measure()`).
10. The PMU mode is switched back to force voltage. It is still forcing `vpar_force()`.

11. The DPS is programmed to the current PMU force-voltage value. This step is skipped if `pmu_comp_cap()` = 0.
12. The DPS is reconnected to the DUT pin(s), in parallel with the PMU.
13. The PMU is disconnected from the DUT.
14. If a sequential PMU test is being performed the process repeats for each pin in the pin list argument to `partest()`.
15. Once all pins are tested, PASS/FAIL is determined, and any measured values can be retrieved.
16. The DPS is programmed back to the voltage value saved in step-1. This step is skipped if `pmu_comp_cap()` = 0.

Dynamic PMU Tests on DPS Pins

In dynamic tests, the DC Comparators and Error Logic can be enabled for the duration of the test pattern execution (default) or triggered from the test pattern. See Dynamic DC Tests.

As with static tests, the sequence below is different if `pmu_comp_cap()` = 0 (see Note:)

1. The currently programmed DPS voltage of the pin(s) to be tested is saved. This step is skipped if `pmu_comp_cap()` = 0.
2. The DPS(s) on the pin(s) to be tested is programmed to the PMU force-voltage value (the value currently set by `vpar_force()`), to pre-charge these pin(s) to the desired PMU force voltage. This step is skipped if `pmu_comp_cap()` = 0.
3. The PMU is set to force `vpar_force()`.
4. The PMU(s) are connected to the pin(s) to be tested, in parallel with the DPS.
5. The DPS is then disconnected from these pin(s). This leaves the PMU connected as the sole power source on the pin(s) being tested.
6. The PMU forces the programmed force value. This may change the PMU mode from force voltage to force current, as determined by the first argument to `ac_partest()`.
7. The Built-in Settling Time time occurs.
8. The user programmed Parametric Settling Time occurs. A non-zero value is typically not useful during dynamic tests.
9. Four scenarios are possible in the next step, based on the state of `measure()` and whether the `vcomp` argument is specified as an argument to `ac_partest()`:

- If `measure()` = FALSE and `vcomp` argument is *not* specified: the DC Comparators and Error Logic are enabled by the site computer and remain enabled for the entire duration of test pattern execution (next).
- If `measure()` = FALSE and `vcomp` argument *is* specified: the DC Comparators and Error Logic will be (must be) triggered from the test pattern using the `MAR VCOMP` instruction (Memory Test Patterns) or `VEC`/`RPT VCOMP`, `VAR VCOMP`, or `VPINFUNC VCOMP` instructions (Logic Test Patterns). If no `VCOMP` triggers are received from the APG the DC portion of the test cannot fail.
- If `measure()` = TRUE and `vcomp` argument *is* specified: the DC A/D Converter will be (must be) triggered from the test pattern using the `MAR VCOMP` instruction (Memory Test Patterns) or `VEC VCOMP`, `VAR VCOMP`, or `VPINFUNC VCOMP` instructions (Logic Test Patterns). If no `VCOMP` triggers are received from the APG the DC portion of the test cannot fail.
- If `measure()` = TRUE and `vcomp` argument is *not* specified: this is invalid.. A warning is issued, testing continues, and the test operates as though `measure()` = FALSE. Any DC measurements which are retrieved are invalid.

When using `vcomp`, the DC Error Flag will affect test pattern branch-on-error operations, and will be cleared using the `MAR RESET` and `CHIPS RESET` instructions (Memory Test Patterns) and `VEC RESET`, `VAR RESET` and `VPINFUNC RESET` (Logic Test Patterns).

10. The test pattern is executed, and will stop based on the pattern stop options specified to `ac_partest()`.
11. The PMU mode is switched back to force voltage.
12. The DPS is programmed to the PMU force-voltage value. This step is skipped if `pmu_comp_cap()` = 0.
13. The DPS is reconnected to the DUT pin(s), in parallel with the PMU.
14. The PMU is disconnected from the DUT.
15. The DPS is programmed back to the voltage value saved in step-1. This step is skipped if `pmu_comp_cap()` = 0.
16. `PASS`/`FAIL` is determined by reading both the DC Error Flags and the PE error latches.

3.13.15 PMU: Testing HV Pins

See Overview, Parametric Measurement Unit (PMU), High Voltage Source/Measure Unit (HV).

Overview

Static PMU tests are executed using the `partest()` function. Dynamic PMU tests are executed using the `ac_partest()` function. Both functions require a pin list argument to specify which DUT pins are to be tested. This pin list must contain only one type of pin: signal pins, or HV pins, or DPS pins. This section describes operation when the pin list contains HV pins.

The model below shows how the PMU connects to an HV pin:

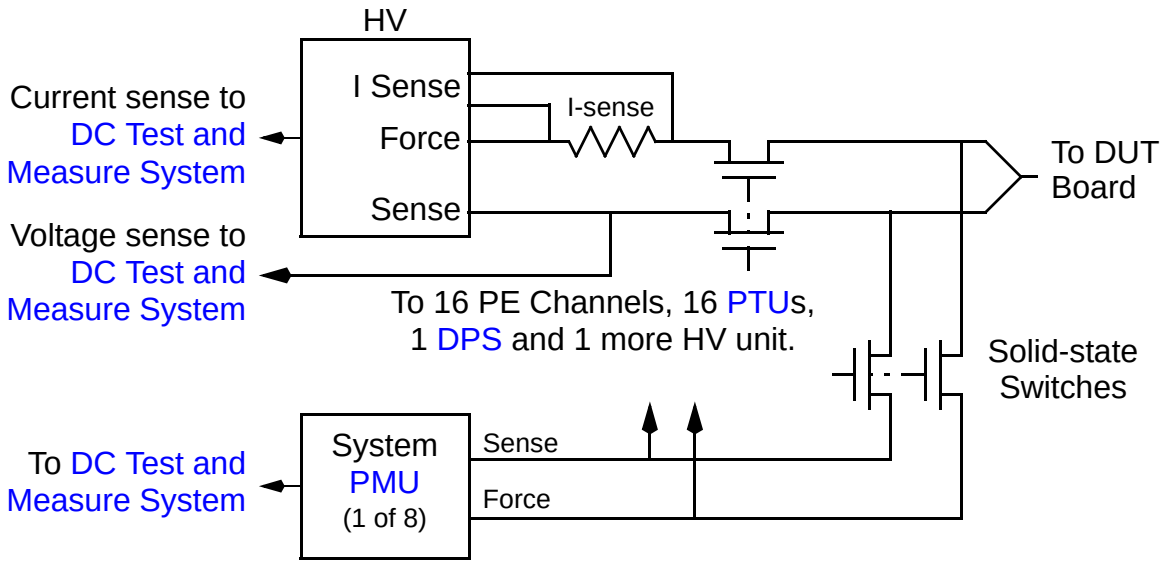


Figure-40: Magnum 1/2 PMU-on-HV Block Diagram

Each Site Assembly Board has 16 HV units and 8 PMUs; i.e. in hardware, two HV units are associated with a given PMU. Neither the HV hardware nor the PMU hardware is designed to connect a given PMU to two HV units at the same time. Thus when using `partest()` or `ac_partest()` to test HV pins if the specified pin list contains HV units which share a given PMU, the test will be performed in two steps, connecting the PMU to one HV at a time. When using `ac_partest()`, the test pattern will be executed two times.

When using the PMU to test HV pins the sequence of events is different for Static PMU Tests on HV Pins vs. Dynamic PMU Tests on HV Pins, as noted below.

Note: the sequences below only apply when testing HV pins using the PMU. Also, in both static and dynamic testing, the HV(s) and PMU are automatically connected and disconnected by the system software during the test; i.e. user code should ***NOT*** to do this.

Static PMU Tests on HV Pins

1. The currently programmed output voltage of each HV being tested is saved.
2. The HV being tested is programmed to the PMU force-voltage value (the value currently set by `vpar_force()`), to pre-charge these pin(s) to the desired PMU force voltage.
3. The PMU is set to force the current `vpar_force()` value.

Note: the High Voltage Source/Measure Unit (HV) cannot source a negative voltage. It is a fatal error to attempt a PMU-on-HV test with a negative force voltage (`vpar_force()`) or negative test limits (`vpar_high()`, `vpar_low()`).

---

4. In parallel with the HV, the PMU is connected to the HV being tested.
5. The HV is disconnected, leaving only the PMU connected to the HV pin.
6. The PMU is set to force the parameter specified by `partest()`. This may change the PMU mode from force voltage to force current.
7. The Built-in Settling Time time occurs.
8. The user programmed Parametric Settling Time occurs.
9. The PMU performs the specified Go/NoGo test or measurement. See `measure()`.
10. The PMU mode is switched back to force voltage.
11. The HV is programmed to the current PMU force-voltage value.
12. The HV is reconnected, in parallel with the PMU.
13. The PMU is disconnected from the HV pin.
14. The HV is programmed back to the voltage value saved in step-1.
15. As noted above, a given `partest()` may connect to and test one or two HV units. Once all HVs are tested, PASS/FAIL is determined, and any measured values can be retrieved. PASS/FAIL is determined by reading either the DC Error Flag (`measure()` = FALSE) or retrieving the measured value and comparing it with the PMU PASS/FAIL test limits.

Dynamic PMU Tests on HV Pins

Note that steps 1 through 7 are identical to static tests.

1. The currently programmed voltage of the HV being tested is saved.

2. The HV being tested is programmed to the PMU force-voltage value (the value currently set by `vpar_force()`), to pre-charge these pin(s) to the desired PMU force voltage.

Note: the High Voltage Source/Measure Unit (HV) cannot source a negative voltage. It is a fatal error to attempt a PMU-on-HV test with a negative force voltage (`vpar_force()`) or negative test limits (`vpar_high()`, `vpar_low()`).

---

3. The PMU is set to force the current `vpar_force()` value.
4. In parallel with the HV, the PMU is connected to the HV pins being tested.
5. The HV is disconnected, leaving only the PMU connected to the HV pin.
6. The PMU is set to force the value specified by `partest()`. This may change the PMU mode from force voltage to force current.
7. The Built-in Settling Time time occurs.
8. The user programmed Parametric Settling Time occurs. A non-zero value is typically not useful during dynamic tests.
9. Four scenarios are possible in the next step, based on the state of `measure()` and whether the `vcomp` argument is specified as an argument to `ac_partest()`:
   - If `measure()` = FALSE and `vcomp` argument is *not* specified: the DC Comparators and Error Logic are enabled by the site computer and remain enabled for the entire duration of test pattern execution (next).
   - If `measure()` = FALSE and `vcomp` argument *is* specified: the DC Comparators and Error Logic will be (must be) triggered from the test pattern using the `MAR VCOMP` instruction (memory patterns) or `VEC VCOMP`, `VAR VCOMP`, or `VPINFUNC VCOMP` instructions (logic patterns. If no `VCOMP` triggers are received from the APG the DC portion of the test cannot fail.
   - If `measure()` = TRUE and `vcomp` argument *is* specified: the DC A/D Converter will be (must be) triggered from the test pattern using the `MAR VCOMP` instruction (Memory Test Patterns) or `VEC`/`RPT VCOMP`, `VAR VCOMP`, or `VPINFUNC VCOMP` instructions (Logic Test Patterns). If no `VCOMP` triggers are received from the APG the DC portion of the test cannot fail.
   - If `measure()` = TRUE and `vcomp` argument is *not* specified: this is invalid.. A warning is issued, testing continues, and the test operates as though `measure()` = FALSE. Any DC measurements which are retrieved are invalid.

When using vcomp, the DC Error Flag will affect test pattern branch-on-error operations, and will be cleared using the MAR RESET and CHIPS RESET instructions (Memory Test Patterns) and VEC/RPT RESET, VAR RESET and VPINFUNC RESET (Logic Test Patterns).

10. The test pattern is executed, and will stop based on the pattern stop options specified to ac_partest().
11. The PMU mode is switched back to force voltage.
12. The HV is programmed to the current PMU force-voltage value.
13. The HV is reconnected, in parallel with the PMU.
14. The PMU is disconnected from the HV pin.
15. The HV is programmed back to the voltage value saved in step-1.
16. PASS/FAIL is determined by reading either the DC Error Flag (measure() = FALSE) or retrieving the measured value and comparing it with the PMU PASS/FAIL test limits. And, if any PE error latches are set the test fails, regardless of the DC test result.

3.13.16 parametric_mode()

See Overview.

Description

The parametric_mode() function can be used to determine the test/measurement mode of the most recently executed partest(), ac_partest(), hv_test_supply(), hv_ac_test_supply(), ptu_partest() test_supply() or ac_test_supply().

Note the following:

- Two modes are possible: test current or test voltage. The mode of each test is controlled by the PassCond argument passed to the test function:
  - pass_pcl, pass_ncl, and pass_nicl test current.
  - pass_vg, pass_vl, and pass_nivl test voltage.
- Only one global mode state exists; i.e. a separate state is not independently kept for partest() vs. test_supply() or per pin, per DUT, etc.

Usage

```
int parametric_mode();
```

where 0 is returned if the last executed parametric test tested/measured current, or 1 if the last executed test tested/measured voltage.

Example

```
int result = partest( pass_nicl, pl_pins );

output(" The last parametric test executed was a \\");
if ( parametric_mode() == 0 ) output("current test");
else                          output("voltage test");
```

3.13.17 PMU as Voltage/Current Source

See Overview, Parametric Measurement Unit (PMU).

Description

The PMU can be statically connected to DUT pin(s) for use as a voltage or current source. The `pmu_connect()` function is used to connect the PMU and `pmu_disconnect()` to disconnect the PMU.

Note: standard PMU tests, i.e. `partest()` or `ac_partest()`, automatically control PMU connections and disconnections to DUT pins; i.e. it is NOT necessary to use `pmu_connect()` or `pmu_disconnect()` when performing these tests.

Note: see Limitations.

Arguments passed to the `pmu_connect()` function are used to:

- Specify which DUT pins are to be connected to the PMU. Only signal pins and HV pins are legal.
- Optionally specify whether to force voltage or current. When not specified, the default connection is force-voltage mode.
- Optionally specify a current range. If not specified, the highest voltage or current range is selected.
- Optionally specify whether the PMU is to sense at a tester channel or at the PMU. If not specified, the sensing is at the tester channel.

Note the following:

- `pmu_connect()` statically connects the specified pin(s) to their associated PMU.
- Using `pmu_connect()`, the pins to be connected are identified using the `pPinlist` or `pDutPin` argument. Each execution of `pmu_connect()` records the specified `pPinlist` or `pDutPin`. The version of `pmu_disconnect()` which includes a `pPinlist` or `pDutPin` argument will only accept a `pPinlist` or `pDutPin` previously passed to `pmu_connect()`. Using `pmu_disconnect()` with any other argument value generates a warning and no pins are disconnected.
- It is illegal to attempt to connect a pin to its PMU when it is already connected to its PMU. A warning is issued.
- The PMU will remain connected to the specified pins until disconnected by calling `pmu_disconnect()`. Pins will remain connected across test blocks, and are ***not*** disconnected at the end of the Sequence & Binning Table execution.
- Executing `pmu_connect()` does not disconnect any pin(s) previously connected to their associated PMU. Executing `pmu_disconnect()` only disconnect the specified pin(s)from their associated PMU.
- It is not legal to connect a PMU to a pin which is currently statically connected to a PTU. If this is attempted, a warning is issued for each violating pin and the PMU connection to these pins is not completed. Other pins which are not violating the rule will be connected to their associated PMU.
- It is possible to use `pmu_connect()` prior to `partest()` or `ac_partest()`. However, when this is done the system software requires that the pins passed to `pmu_connect()` and `partest()` or `ac_partest()` contain *exactly* the same pin members. At the time `partest()` or `ac_partest()` are executed if any pin discrepancies are detected a warning message is generated and the test is not executed. And, if PMU measurements were enabled (see `measure()`) any values retrieved will be invalid (see Retrieving DC Test Results).
- When pins are connected using `pmu_connect()`, subsequent executions of `partest()` or `ac_partest()` do ***not*** manage PMU-to-pin connections as is done when executing these tests normally. Executing `pmu_disconnect()` causes the PMU to be *restored* to normal operation.
- The PMU force parameter value must be programmed prior to executing `pmu_connect()` or default values will be used (0V/0mA). The `vpar_force()` function is used to set a PMU force voltage level. The `ipar_force()` function is used to set a PMU force current level.
- Whether forcing voltage or current, the PMU voltage clamps, set using `vclamp()`, will limit the output voltage of the PMU.

Note: *careful consideration* must be given to setting proper PMU voltage clamp values, especially when using the PMU to force current. The PMU output voltage range is -2.5V to +12.75V. When forcing current, if the DUT doesn't adequately load the PMU, only the voltage clamps, set using `vclamp()`, will prevent the PMU from reaching one of these voltages. **This is true even when forcing 0nA current**.

The `pmu_connect_at()` function determines how subsequent PMU connections are sequenced to the DUT pins and whether connections are made in force voltage mode prior to forcing current. Once set, the mode does not change until `pmu_connect_at()` is executed again. Two options exist:

- A *break-before-make* sequence, in which the DUT pin(s) are first disconnected from the PE channel, then the PMU is switched to the pins, and finally the PMU force voltage or current is programmed. This is the default and set using `pmu_connect_at(FALSE)`.
- A *make-before-break* sequence, in which the PMU voltage or current is first programmed, then the PMU connections are made to the DUT pins, and finally the PE channel is disconnected from the pins.
  This is set using `pmu_connect_at(TRUE)`.
- Executing `pmu_connect_at(TRUE)` also sets a flag which causes all subsequent PMU connections to be made in force-voltage mode, at the voltage last programmed using `vpar_force()`. This occurs even if the PMU is subsequently changed to force-current mode. Executing `pmu_connect_at(FALSE)` clears this flag, allowing the PMU connections to be made with the PMU in force-current mode.

Using `pmu_disconnect()`, the sequence of disconnecting the PMU and reconnecting pins to PE channels is the reverse of the `pmu_connect()` sequence. The version of `pmu_disconnect()` which includes a `pPinlist` or `pDutPin` argument will only accept a `pPinlist` or `pDutPin` previously passed to `pmu_connect()`. Using `pmu_disconnect()` with any other argument value generates a warning and no pins are disconnected.

PMU force voltage sensing can be at the PMU (local sensing) or at a PE channel (remote sensing). If not specified, remote sensing is used.

Each PMU can connect to 16 signal pins. Thus the pin list passed to `pmu_connect()` may include pins which connect to different PMUs. In this scenario, if remote sensing is used, sensing occurs at the first pin of the specified pin list which connects to a given PMU. Note that when the PMU is forcing current, voltage sensing is always internal to the PMU.

System software does *not* allow the PMU force mode to be changed while the PMU is connected. Thus, it is *not* possible to execute `pmu_connect(..FORCEV..)` following by `pmu_connect(..FORCEI..)` without executing `pmu_disconnect()` between them.

It is possible to change the force voltage or current *value* while the PMU is connected to PE channels. However, when forcing a current, it is not possible to change the current *range* while the PMU is connected to the DUT. Therefore, it is appropriate to use an explicit range value for:

- Any `ipar_force()` done in preparation for using `pmu_connect()`
- The `pmu_connect(... FORCEI..., range...)` option
- Any `ipar_force()` done while the PMU is connected to the DUT.

Limitations

As of 5/2008 the following limitations exist.:

- `pmu_connect()` only supports connections to PE channels (signal pins) and HV pins; i.e. not DPS pins.
- It is *not* possible to set up `FORCEV` and `FORCEI` at the same time, even though using PMUs on different channels. It is possible to set up different conditions for `FORCEV` - ***or*** - `FORCEI` by calling `pmu_connect()` multiple times, each with a pin list constrained to a single single PMU(via the specified pin list).
- Do not use `pin_connect()` to re-connect signal pins which are currently connected to the PMU.

Usage

Three functions are documented below. The multiple overloads of `pmu_connect()` provide for different combinations of argument usage. Only those combinations defined below are supported.

```
void pmu_connect_at( BOOL state );
```

Note: the `pmu_connect()` function overloads below which do *NOT* accept the **`force_type`** argument will, by design, force a voltage.

```
void pmu_connect( DutPin *pDutPin );
void pmu_connect( PinList* pPinList );
void pmu_connect( DutPin *pDutPin, Range range );
void pmu_connect( PinList* pPinList, Range range );
```

```
void pmu_connect( DutPin *pDutPin, PMUSense sense_type );
void pmu_connect( PinList* pPinList, PMUSense sense_type );
void pmu_connect( DutPin *pDutPin,
                  PMUSense sense_type,
                  Range range );
void pmu_connect( PinList* pPinList,
                  PMUSense sense_type,
                  Range range );
 void pmu_connect( DutPin *pDutPin, PMUMode force_type );
void pmu_connect( PinList* pPinList, PMUMode force_type );
void pmu_connect( DutPin *pDutPin,
                  PMUMode force_type,
                  Range range );
void pmu_connect( PinList* pPinList,
                  PMUMode force_type,
                  Range range );
void pmu_disconnect();
void pmu_disconnect( DutPin *pDutPin );
void pmu_disconnect( PinList*pPinList ); // See Limitations
```

where:

`state` specifies how subsequent PMU connections, using `pmu_connect()`, and disconnections, using `pmu_disconnect()`, are made. If **`state`** is **FALSE** the *break-before-make* mode is used. If **`state`** is **TRUE** the *make-before-break* mode is used. See Description.

`pDutPin` identifies one pin to be connected to or disonnected from its associated PMU. In Multi-DUT Test Programs, the same pin of each DUT currently in the Active DUTs Set (ADS) is affected. **`pDutPin`** may only include one type of pin; i.e. only signal pins or only HV pins.

Note: prior to executing `pmu_connect()` or `pmu_disconnect()`, user code is responsible for programming a PMU force voltage/current value that is consistent with the voltage state of the DUT pin at the time the PMU connection is made. This is especially true when **`state`** = TRUE because the DUT pin is not yet disconnected from the pin channel when the PMU is connected, potentially allowing the PMU and tester channel to be at different voltages. Using **`state`** = FALSE causes the DUT pin to float before the PMU is connected, potentially reducing any voltage differences. In general, any voltage difference between the PMU voltage and the actual voltage on the DUT pin(s) can cause undesirable voltage transients as the PMU connection is made.

`pPinList` specifies which pins to connected to, or disconnected from, the PMU. The pin list can not contain DPS pins, see Limitations.

`range` specifies a PMU current range only, but affects both force current mode, and the current test/measure range when in force voltage mode. Legal values are of the `Range` enumerated type and are used as follows:

Table 3.13.17.0-1 PMU Force Current Ranges

Current Range	LSB	Range
±2uA	1nA	range1
±20uA	10nA	range2
±200uA	100nA	range3
±2mA	1uA	range4
±20mA	10uA	range5

`sense_type` is used to specify the PMU sense mode when forcing voltage. Legal values are of the `PMUSense` enumerated type: `REM` selects *remote* PMU sensing (at the tester channel); `LOCAL` selections *local* PMU sense (at the PMU). If **`sense_type`** is not specified, default operation is remote sense, at the first pin in **`pPinList`**, per PE board. **`sense_type`** affects force voltage sensing only; i.e. test/measure voltage sensing is always done at the tester channel.

force_type is used to specify the PMU force mode. Legal values are of the PMUMode enumerated type: FORCEV selects the force voltage mode; FORCEI selects the force current mode. If **force_type** is not specified, default operation is force voltage.

Example

In the following example, the PMU is programmed to force -2V and is connected in parallel to all pins defined by the pin list called Data_bus. Note that multiple PMU(s) may be used if some pins of Data_bus connect to a different PMU than other pins. Additional test program code is executed, then the PMU is disconnected. PMU sensing is done remotely, at the DUT pin(s). The PMU connection mode is based on the most recent execution of pmu_connect_at().

```
vpar_force( -2 V );      // Set force voltage
pmu_connect( Data_bus ); // Connect PMU(s) to all Data_bus pin(s)
// Other test program code executed here
pmu_disconnect();        // Disconnect PMU from all pins
```

3.13.18 PMU Compensation Capacitors

See Overview, Parametric Measurement Unit (PMU).

Description

Using the PMU, the pmu_comp_cap( ) function is used to select an internal compensation capacitor based on the anticipated load on pin(s) to be tested.

The PMU is a closed-loop feedback system with stability that is affected by the circuit load. Because of the wide range of possible load capacitance, a single fixed compensation scheme can not effectively balance the trade-off between stability and output settling time. PMU feedback loop compensation is accomplished by selecting between three compensation options (more below).

In general, the smallest compensation capacitor value that assures stability for the given load capacitance should be used. Using a larger than needed compensation capacitor will only slow down measurements, particularly on the lower PMU current ranges, requiring increased slewing/settling time when voltage levels are changed.

The minimum compensation capacitor value is selected during initial program load. The selection is not otherwise changed by the system software.

PMU compensation capacitors are disabled when performing force current tests.

Note: when performing PMU: Testing DPS Pins, or PMU: Testing DPS Pins, in a force current mode, the PMU connection to the DPS pins is initially made in voltage force mode, then the PMU is switched to force current mode to complete the test. Even though the PMU compensation capacitors are disconnected automatically when the PMU is in force current mode, the initial connection is being made in force voltage mode, thus the appropriate compensation capacitor selection should be made.

Note: connecting the PMU in force voltage mode to a DUT pin, whether a DPS pin or signal pin, having a total capacitance greater than 1.0uF is not advised.

Usage

The following function is used to set the PMU compensation capacitor for all PMUs:

```
void pmu_comp_cap( int value );
```

The following function is used to set the PMU compensation capacitor for a specified PMU:

```
void pmu_comp_cap( int value, DutPin* dutpin );
```

The following function is used to set the PMU compensation capacitor for one or more PMUs:

```
void pmu_comp_cap( int value, PinList* pPinList );
```

The following function returns the currently selected PMU compensation capacitor for one PMU:

```
int pmu_comp_cap( DutPin *pDutPin );
```

where:

value selects the desired compensation capacitor. Legal values are described in the table below:

Table 3.13.18.0-1 PMU Compensation Capacitor Selection

Value	Purpose
0	Intended for normal measurements on tester signal pins. This value should be specified for DUT capacitance values up to ~ 3000pF. Default.
1	Intended for use when the PMU, in force voltage mode, connects to a DPS pin. This value should be specified for any pin with capacitance greater than 3000pF but less than 0.1uF.
2	Intended for use when the PMU, in force voltage mode, connects to a DPS pin. This value should be specified when connecting to any pin with capacitance greater than 0.1uF but less than 1.0uF.

pDutPin is used in two contexts:

- In the setter functions, identifies one PMU to be programmed. In Multi-DUT Test Programs, the PMUs of each DUT currently in the Active DUTs Set (ADS) are affected.
- In the getter functions, identifies one PMU to be read.

pPinList specifies which PMU(s) are to be programmed. In Multi-DUT Test Programs, the PMUs of each DUT currently in the Active DUTs Set (ADS) are affected.

The getter version of `pmu_comp_cap()` returns one of the values noted in the table above. In Multi-DUT Test Programs, the value is retrieved from the first DUT in the Active DUTs Set (ADS).

Example

In this example the continuity of the Vcc power pin of a device having a .05 uF bypass capacitor is tested using the PMU. Because the PMU is first connected to the power pin in voltage force mode, a PMU comp cap is specified. After the test, the default capacitor is selected so that subsequent PMU tests will not have excessively long settling times.

```
pmu_comp_cap( 1 );
BOOL result = partest( pass_nivl, vcc_pin );
pmu_comp_cap( 0 );
```

3.14 PTU Functions

See Per-pin Parametric Test Unit (PTU), Site Assembly Board, Pin Electronics (PE) Block Diagram, PE Driver Block Diagram.

This section includes the following:

- Overview
- PTU Usage
- PTU Connect/Disconnect Functions
- PTU Force-current Functions
- PTU Current Test Limit Functions
- PTU Force-voltage Functions
- PTU Voltage Test Limit Functions
- PTU Voltage Clamp Functions
- PTU Static Test Functions
- PTU as Voltage/Current Source

Other related information includes:

- Static DC Tests and Dynamic DC Tests
- Parametric Settling Time and Built-in Settling Time
- `measure()`
- Retrieving DC Test Results

3.14.1 Overview

See Per-pin Parametric Test Unit (PTU), PTU Functions.

As the name implies, each of the 128 pins on a Site Assembly Board has an independent Per-pin Parametric Test Unit (PTU). The PTU provides the following:

- Per-pin DC parametric tests. Force voltage and test current or force current and test voltage.
- A statically connected voltage or current source.

- The PTU is the source of the VZ voltage used in certain Magnum PE Driver Modes.
- The PTU is the source of the VIHH voltage used in certain Magnum PE Driver Modes.
- The PTU is the source of the Parametric Background Voltage used in certain PMU and PTU test modes.

The PTU output voltage and current capabilities are affected by the selected current range. See PTU Operating Area.

The PTU provides for concurrent DC Go/NoGo tests on multiple pins, in parallel. However, DC *measurements* using the PTU also utilize the Parametric Measurement Unit (PMU), to route the parameter being measured to the DC Sub-System. Since there are 8 PMUs per Site Assembly Board PTU measurements may not be done concurrently. More below.

As noted, each PTU can force current and test voltage, or force voltage and test current. See:

- PTU Force-current Functions
- PTU Current Test Limit Functions
- PTU Force-voltage Functions
- PTU Voltage Test Limit Functions

The PTU(s) can be used to perform static parametric tests. See PTU Static Test Functions.

Each PTU has a positive and negative programmable voltage clamp. See PTU Voltage Clamp Functions.

Normally, the system software automatically manages PTU(s) connect and disconnect operations during PTU tests. For those special occasions, it is also possible to manually manage PTU connections. See PTU Connect/Disconnect Functions.

The PTU(s) can also be used as statically connected voltage or current source(s). See PTU as Voltage/Current Source.

As noted above, to make a PTU measurement (as opposed to a Go/NoGo test) the Parametric Measurement Unit (PMU) is involved. The hardware model is shown in the DC Sub-System Block Diagram. When making a PTU measurement (`measure()` = TRUE) one PTU at a time is connected to its associated PMU. The output of the PMU is selected by the DC Source Select MUX and routed to the DC Test and Measure System, where the DC A/D Converter makes the measurement. See Static DC Tests, Dynamic DC Tests, and Retrieving DC Test Results.

3.14.2 PTU Usage

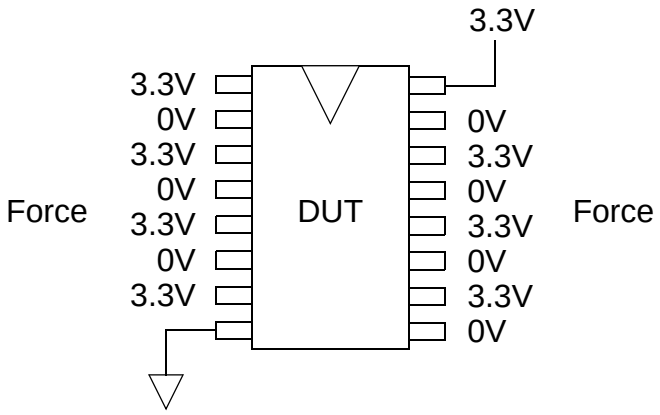
See Per-pin Parametric Test Unit (PTU), PTU Functions.

The two most common applications for the PTU are continuity tests (opens and shorts tests) and (adjacent channel) leakage tests.

In both applications, it is common that one group of pins is forced to a low voltage while other (adjacent) pins are forced to a high voltage. Then the amount of current is tested on each pin. Or, all pins are forced to a negative current and the voltage on each pin is tested.

In these applications, using the system PMU typically requires a sequential test, where all pins are forced to a low (or high) voltage by a PE driver (or background voltage) and one pin at a time is forced to the opposite voltage and tested for current. This method of sequential testing is not required when the test system has a PTU, which makes it possible to test all pins concurrently.

The diagram below illustrates this graphically:



The following step are used to configure the PTUs to perform this test:

- Define 3 pin lists:
  - `pl_even_pins`: to force 0V
  - `pl_odd_pins`: to force 3.3V
  - `pl_test_pins`: includes pins from both `pl_even_pins` and `pl_odd_pins`
- Using `ptu_vpar_force_set()`, program the force-voltage for `pl_even_pins` to 0V.
- Using `ptu_vpar_force_set()` again, program the force-voltage for `pl_odd_pins` to 3.3V.

- Using `ptu_ipar_high_set()`, program the high test limit for `pl_odd_pins` to the desired value. Using `ptu_ipar_low_set()`, program the low test limit for `pl_odd_pins` to the desired value.
- Repeat this for `pl_even_pins`.
- Using `ptu_partest()`, execute the PTU test and specify `pl_test_pins` as the pins to be tested.
- Reverse the force-voltage and test limits for the `pl_even_pins` and `pl_odd_pins`, and execute `ptu_partest()` again.

3.14.3 PTU Connect/Disconnect Functions

See Per-pin Parametric Test Unit (PTU), PTU Functions, PTU as Voltage/Current Source.

The `ptu_connect()` and `ptu_disconnect()` functions are used to explicitly control static connections between a PTU and its associated pin. These functions are normally only used when using PTU(s) as a statically connected voltage or current source and are thus documented in the section titled PTU as Voltage/Current Source.

Note: standard PTU tests using `ptu_partest()` ***automatically*** control PTU connections and disconnections to pins; i.e. it is ***NOT*** necessary to use `ptu_connect()` or `ptu_disconnect()` when performing these tests.

3.14.4 PTU Force-current Functions

See Per-pin Parametric Test Unit (PTU), PTU Functions.

Description

The `ptu_ipar_force_set()` function is used to set a PTU force-current value, for one or more pin(s).

The `ptu_ipar_force_get()` function is used to read the currently programmed force-current value from one specified PTU.

The PTU has 8 force-current ranges:

Table 3.14.4.0-1 PTU Force-Current Ranges

Current Range	Range	Resolution
±2uA	range1	1nA
±8uA	range2	4nA
±32uA	range3	16nA
±128uA	range4	64nA
±512uA	range5	256nA
±2mA	range6	1uA
±8mA	range7	4uA
±32mA	range8	16uA
Note: the PTU output voltage and current capabilities are affected by the selected current range. See PTU Operating Area.		

`ptu_ipar_force_set()` is used for both static force-current PTU applications (see PTU as Voltage/Current Source) and for DC parametric tests in which the PTU will force current (`ptu_partest()`).

By default, the current range is selected automatically based on the force-current value programmed, however it is also possible to explicitly set the current range using the range argument to `ptu_ipar_force_set()`. Once explicitly set, the range value doesn't change except as follows:

- User code explicitly selects the a different range using `ptu_ipar_force_set()`.
- User code executes `ptu_ipar_force_set()` without specifying an explicit range value.

If the PTU is in the force-current mode and is currently connected to the pin (see PTU as Voltage/Current Source), executing `ptu_ipar_force_set()` causes the output current of the PTU to change immediately, provided the current range does not need to change. Once the PTU is connected to its pin (see `ptu_connect()`), no voltage range or current range changes are permitted. If a range change is attempted, a warning is issued and no changes are made to the hardware.

The system software programs the PTU force-current value to 0 during initial test program load. When execution of the Sequence & Binning Table stops, the `builtin_after_testing_block` programs all PTU to force 0V @ 0nA.

The PTU force-current can be modified from a test pattern. See Controlling PE Levels from the Test Pattern.

Note: a commonly made mistake is to assume that programming PTU force value or PASS/FAIL test limit also defines the type of PTU test which will execute next. It is the arguments passed to `ptu_partest()` which defines the type of test the PTU actually performs (force-current/measure-voltage, etc.), and thus which force and test limits will be used.

Usage

The following function programs the force-current value for all PTUs:

```
void ptu_ipar_force_set( double value,
                         Range range DEFAULT_VALUE( norange ));
```

The following function programs the force-current value for one specified PTU:

```
void ptu_ipar_force_set( double value,
                         DutPin *pDutPin,
                         Range range DEFAULT_VALUE( norange ));
```

The following function programs the force-current value for one or more specified PTU:

```
void ptu_ipar_force_set( double value,
                         PinList* pPinlist,
                         Range range DEFAULT_VALUE( norange ));
```

The following functions gets the currently programmed force-current value from the specified PTU:

```
double ptu_ipar_force_get( DutPin *pDutPin );
```

where:

`value` is the desired PTU force-current. Units may be used (see Specifying Units).

`pDutPin` is used in two contexts:

- In the setter function, identifies one PTU to be programmed. In Multi-DUT Test Programs, the PTUs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.
- In the getter function, identifies one PTU to be read.

`pPinList` specifies which PTU(s) are to be affected. In Multi-DUT Test Programs, only pin(s) of DUT(s) currently in the Active DUTs Set (ADS) are affected. The pin list must only contain `DutPin`s mapped to a signal pins in the Pin Assignment Table.

`range` is optional, and if used explicitly selects a PTU current range. See Description. Legal range values must be one of the `Range` enumerated types, but the preferred method is to use values from Table 3.14.4.0-1.

`ptu_ipar_force_get()` returns the currently programmed PTU force-current value for the specified **`pin`**. The value is returned for the first DUT in the Active DUTs Set (ADS) (see Using Getter Functions).

Example

```
ptu_ipar_force_set( 1 MA, pl_even_pins );
ptu_ipar_force_set( 1 MA, pl_even_pins, range6 );
double i = ptu_ipar_force_get( D0 );
```

3.14.5 PTU Current Test Limit Functions

See Per-pin Parametric Test Unit (PTU), PTU Functions.

Description

The `ptu_ipar_high_set()` and `ptu_ipar_low_set()` functions are used to set PTU current-test PASS/FAIL limits. These are the high/low test limits used by `ptu_partest()` when performing a force-voltage/test-current test.

The `ptu_ipar_high_get()` function reads the currently programmed upper current-test limit for one specified PTU. The `ptu_ipar_low_get()` function reads the currently programmed low current-test limit for one specified PTU.

`ptu_ipar_high_set()` and `ptu_ipar_low_set()` must be programmed before executing the test.

Both limits are set to zero at test program initialization, and are otherwise not modified by the system software.

Both limits can be modified from a test pattern. See Controlling PE Levels from the Test Pattern.

The PTU has 8 force-current ranges:

Table 3.14.5.0-1 PTU Force-Current Ranges

Current Range	Range	Resolution
±2uA	range1	1nA
±8uA	range2	4nA
±32uA	range3	16nA
±128uA	range4	64nA
±512uA	range5	256nA
±2mA	range6	1uA
±8mA	range7	4uA
±32mA	range8	16uA
Note: the PTU output voltage and current capabilities are affected by the selected current range. See PTU Operating Area.		

When a range is not explicitly programmed, the system software selects the most accurate range based on the values programmed using `ptu_ipar_high_set()` and `ptu_ipar_low_set()`. If the high and low limits fall into different ranges, the system software sets the range to the coarser (lower resolution) range.

Note: a commonly made mistake is to assume that programming PTU force value or PASS/FAIL test limit also defines the type of PTU test which will execute next. It is the arguments passed to `ptu_partest()` which defines the type of test the PTU actually performs (force-current/measure-voltage, etc.), and thus which force and test limits will be used.

Usage

The following functions program the high/low current-test limit value for all PTUs:

```
void ptu_ipar_high_set(double value,
                       Range range DEFAULT_VALUE( norange ) );
void ptu_ipar_low_set( double value,
                       Range range DEFAULT_VALUE( norange ) );
```

The following functions program the high/low current-test limit value for one specified PTU:

```
void ptu_ipar_high_set(double value,
                       DutPin *pDutPin,
                       Range range DEFAULT_VALUE( norange ) );
void ptu_ipar_low_set( double value,
                       DutPin *pDutPin,
                       Range range DEFAULT_VALUE( norange ) );
```

The following functions program the high/low current-test limit for one or more PTU(s):

```
void ptu_ipar_high_set(double value,
                       PinList* pPinlist,
                       Range range DEFAULT_VALUE( norange ) );
void ptu_ipar_low_set( double value,
                       PinList* pPinlist,
                       Range range DEFAULT_VALUE( norange ));
```

The following functions get the currently programmed high/low current-test limit for one specified PTU:

```
double ptu_ipar_high_get( DutPin *pDutPin );
double ptu_ipar_low_get( DutPin *pDutPin );
```

where:

`value` is the desired test limit. Units may be used (see Specifying Units).

`pDutPin` is used in two contexts:

- In the setter function, identifies one PTU to be programmed. In Multi-DUT Test Programs, the PTUs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.
- In the getter function, identifies one PTU to be read.

pPinList specifies which PTU(s) are to be affected. In Multi-DUT Test Programs, only pin(s) of DUT(s) currently in the Active DUTs Set (ADS) are affected. The pin list must only contain DutPins mapped to a signal pins in the Pin Assignment Table.

range is optional, and if used explicitly selects a PTU current range. See Description. Legal range values must be one of the Range enumerated types, but the preferred method is to use values from Table 3.14.4.0-1.

ptu_ipar_high_get() and ptu_ipar_low_get() return the currently programmed PTU high/low current-test limit value for the specified **pin**. The value is returned for the first DUT in the Active DUTs Set (ADS) (see Using Getter Functions).

Example

```
ptu_ipar_high_set( 138 UA, pl_tested_pins );
ptu_ipar_low_set( 0 UA, pl_tested_pins );
double v = ptu_ipar_low_get( D0 );
```

3.14.6 PTU Force-voltage Functions

See Per-pin Parametric Test Unit (PTU), PTU Functions.

Description

The ptu_vpar_force_set() function is used to set a PTU force-voltage value.

The ptu_vpar_force_get() function is used to read the currently programmed force-voltage value from one specified PTU.

ptu_vpar_force_set() is used for both static force-voltage PTU usage (see PTU as Voltage/Current Source) and for DC parametric tests in which the PTU will force voltage (ptu_partest()).

The PTU has one force-voltage range:

Table 3.14.6.0-1 PTU Force-voltage Range

Voltage Range	LSB
-2V to +12V	1mV

Note: the PTU output voltage and current capabilities are affected by the selected current range. See PTU Operating Area.

If the PTU is in the force-voltage mode and is currently connected to its pin (see PTU as Voltage/Current Source), executing `ptu_vpar_force_set()` causes the output voltage of the PTU to change immediately.

The system software programs the PTU force-voltage value to 0V during initial test program load. When execution of the Sequence & Binning Table stops, the `builtin_after_testing_block` set the value to 0V for all PTUs.

The PTU force-voltage can be modified from a test pattern. See Controlling PE Levels from the Test Pattern.

Note: `ptu_vclamp_set()` *will* limit the PTU force-voltage. A warning message will be displayed when the user's test program attempts to program this condition, but testing will otherwise continue.

Note: a commonly made mistake is to assume that programming PTU force-value or PASS/FAIL test limit also defines the type of PTU test which will execute next. It is the arguments passed to `ptu_partest()` which defines the type of test the PTU actually performs (force-current/measure-voltage, etc.), and thus which force and test limits will be used.

Usage

The following function programs the force-voltage value for all PTUs:

```
void ptu_vpar_force_set( double value );
```

The following function programs the force-voltage value for one specified PTU:

```
void ptu_vpar_force_set( double value,
                         DutPin *pDutPin );
```

The following function programs the force-voltage value for one or more PTU(s):

```
void ptu_vpar_force_set( double value,
                         PinList* pPinlist );
```

The following function returns the currently programmed force-voltage value from the specified PTU:

```
double ptu_vpar_force_get( DutPin *pDutPin );
```

where:

value is the desired PTU force-voltage. Units may be used (see Specifying Units).

pDutPin is used in two contexts:

- In the setter function, identifies one PTU to be programmed. In Multi-DUT Test Programs, the PTUs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.
- In the getter function, identifies one PTU to be read.

pPinList specifies which PTU(s) are to be affected. In Multi-DUT Test Programs, only pin(s) of DUT(s) currently in the Active DUTs Set (ADS) are affected. The pin list must only contain `DutPin`s mapped to a signal pins in the Pin Assignment Table.

`ptu_vpar_force_get()` returns the currently programmed PTU force-voltage value for the specified **pin**. The value is returned for the first DUT in the Active DUTs Set (ADS) (see Using Getter Functions).

Example

```
ptu_vpar_force_set( 3.4 V, pl_tested_pins );
double value = ptu_vpar_force_get( D0 );
```

3.14.7 PTU Voltage Test Limit Functions

See Per-pin Parametric Test Unit (PTU), PTU Functions.

Description

The `ptu_vpar_high_set()` and `ptu_vpar_low_set()` functions are used to set PTU voltage-test PASS/FAIL limits. These are the high/low test limits used by `ptu_partest()` when performing a force-current/test-voltage test.

The `ptu_vpar_high_get()` and `ptu_vpar_low_get()` functions are used to read the currently programmed PASS/FAIL limit values from one specified PTU.

`ptu_vpar_high_set()` and `ptu_vpar_low_set()` must be programmed before executing the test.

Both `ptu_vpar_high_set()` and `ptu_vpar_low_set()` are set to 0V at test program initialization, but are not otherwise changed by the system software.

Both limits can be modified from a test pattern. See Controlling PE Levels from the Test Pattern.

The PTU has one voltage range for PASS/FAIL test limits:

Table 3.14.7.0-1 PTU Voltage-test Limit Ranges

Voltage Range	LSB
-2V to +12V	2mV

Note: the PTU output voltage and current capabilities are affected by the selected current range. See PTU Operating Area.

Note: `ptu_vclamp_set()` *will* limit the PTU voltage-test limits. A warning message will be displayed when the user's test program attempts to program this condition, but testing will otherwise continue.

Note: a commonly made mistake is to assume that programming PTU force value or PASS/FAIL test limit also defines the type of PTU test which will execute next. It is the arguments passed to `ptu_partest()` which defines the type of test the PTU actually performs (force-current/measure-voltage, etc.), and thus which force and test limits will be used.

Usage

The following functions program the high/low voltage-test limit for all PTUs:

```
void ptu_vpar_high_set( double value );
void ptu_vpar_low_set( double value );
```

The following functions program the high/low voltage-test limit for one specified PTU:

```
void ptu_vpar_high_set( double value, DutPin *pDutPin );
void ptu_vpar_low_set( double value, DutPin *pDutPin );
```

The following functions program the high/low voltage-test limit for one or more PTU(s):

```
void ptu_vpar_high_set( double value, PinList* pPinlist );
```

```
void ptu_vpar_low_set( double value, PinList* pPinlist );
```

The following functions get the currently programmed high/low voltage-test limit for one specified PTU:

```
double ptu_vpar_high_get( DutPin *pDutPin );
double ptu_vpar_low_get( DutPin *pDutPin );
```

where:

`value` is the desired test limit. Units may be used (see Specifying Units).

`pDutPin` is used in two contexts:

- In the setter function, identifies one PTU to be programmed. In Multi-DUT Test Programs, the PTUs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.
- In the getter function, identifies one PTU to be read.

`pPinList` specifies which PTU(s) are to be affected. In Multi-DUT Test Programs, only pin(s) of DUT(s) currently in the Active DUTs Set (ADS) are affected. The pin list must only contain `DutPin`s mapped to a signal pins in the Pin Assignment Table.

`ptu_vpar_high_get()` and `ptu_vpar_low_get()` return the currently programmed PTU high/low voltage-test limit value for the specified **`pin`**. The value is returned for the first DUT in the Active DUTs Set (ADS) (see Using Getter Functions).

Example

```
ptu_vpar_high_set( 3.3 V, pl_tested_pins );
ptu_vpar_low_set( 0 V, pl_tested_pins );
double value = ptu_vpar_low_get( D0 );
```

3.14.8 PTU Voltage Clamp Functions

See Per-pin Parametric Test Unit (PTU), PTU Functions.

Description

The `ptu_vclamp_set()` function is used to program the voltage clamps for one or more PTU(s).

The `ptu_positive_vclamp_get()` and `ptu_negative_vclamp_get()` functions are used to get the currently programmed positive and negative voltage clamp values for a specified PTU.

The `ptu_vclamp_enable()` function is used to enable and disable both PTU voltage clamps. By default, PTU voltage clamps are ***disabled***.

The `ptu_vclamp_enabled()` function is used to get the PTU voltage clamp enable state for one pin.

The PTU has programmable high/low voltage clamps:

Table 3.14.8.0-1 PTU Voltage Clamp Range & LSB

Range	LSB	
0.5V to +12V	4mV	Positive Clamp
-2V to +11V		Negative Clamp

Note the following:

- PTU voltage clamps are enabled only when the PTU is in the force-current mode.
- The negative clamp value must be programmed to a value less than the positive clamp value.
- The PTU voltage clamps affect the following:
  - The voltage output by the PTU, programmed using `ptu_vpar_force_set()`.
  - Voltage-test limits, programmed using `ptu_vpar_high_set()` and `ptu_vpar_low_set()`.

  If any of these are programmed outside the `ptu_vclamp_set()` range, a warning is output in the appropriate controller output window, but testing otherwise continues.
- At test program initialization, the voltage clamps are ***disabled***. The clamps are not otherwise set by the system software.
- Both clamp values can be modified from a test pattern. See Controlling PE Levels from the Test Pattern.

Note: *careful consideration* must be given to setting proper clamp values when using the PTU to force current. The PTU output voltage range is -2V to +12V. When forcing current, if the DUT doesn't adequately load the PTU, only the voltage clamps will prevent the PTU from reaching one of these voltages.

Usage

The following function programs both voltage clamps for all PTUs:

```
void ptu_vclamp_set(double positive_clamp,
                    double negative_clamp );
```

The following function programs both voltage clamps for one specified PTU:

```
void ptu_vclamp_set(double positive_clamp,
                    double negative_clamp,
                    DutPin *pDutPin );
```

The following function programs both voltage clamps for one or more PTU(s):

```
void ptu_vclamp_set(double positive_clamp,
                    double negative_clamp,
                    PinList* pPinlist );
```

The following functions get the currently programmed voltage clamp value for one PTU. Separate functions are used to get the positive vs. negative clamp:

```
double ptu_negative_vclamp_get( DutPin *pDutPin );
double ptu_positive_vclamp_get( DutPin *pDutPin );
```

The following function is used to enable or disable both voltage clamps for all PTU:

```
void ptu_vclamp_enable( BOOL state );
```

The following function is used to enable or disable both voltage clamps for one specified PTU:

```
void ptu_vclamp_enable( BOOL state, DutPin *pDutPin );
```

The following function is used to enable or disable both voltage clamps for one or more PTU(s):

```
void ptu_vclamp_enable( BOOL state, PinList* pPinlist );
```

The following function is used to get the enable state of the voltage clamps for one PTU:

```
BOOL ptu_vclamp_enabled( DutPin *pDutPin );
```

where:

positive_clamp and **negative_clamp** specify the desired positive and negative voltage clamp values. Units may be used (see Specifying Units).

pDutPin is used in two contexts:

- In the setter functions, identifies one PTU to be programmed. In Multi-DUT Test Programs, the PTUs of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.
- In the getter function, identifies one PTU to be read.

state specifies whether the voltage clamps are to be enabled (TRUE) or disabled (FALSE).

pPinList specifies which PTU(s) are to be affected. In Multi-DUT Test Programs, only pin(s) of DUT(s) currently in the Active DUTs Set (ADS) are affected. The pin list must only contain `DutPin`s mapped to a signal pins in the Pin Assignment Table.

`ptu_negative_vclamp_get()` and `ptu_positive_vclamp_get()` return the currently programmed PTU voltage clamp value for the specified **pin**. The value is returned for the first DUT in the Active DUTs Set (ADS) (see Using Getter Functions).

`ptu_vclamp_enabled()` returns TRUE if the voltage clamps of the specified **pin** are enabled, otherwise FALSE is returned. The value is returned for the first DUT in the Active DUTs Set (ADS).

Example

```
ptu_vclamp_set( 3.4 V, -0.5 V, pl_tested_pins );
double v = ptu_negative_vclamp_get( D0 );

ptu_vclamp_enable( TRUE, pl_tested_pins );
BOOL state = ptu_vclamp_enabled( D0 );
```

3.14.9 PTU Static Test Functions

See Per-pin Parametric Test Unit (PTU), PTU Functions, DC Sub-System Block Diagram, Static DC Tests.

Description

The `ptu_partest()` function is used to perform a static DC parametric test using the PTUs. PTU Dynamic Test Functions are documented separately.

As the name implies, each Magnum 1/2/2x test channel has an independent PTU, allowing tests on multiple pins concurrently. The two most common applications are continuity tests and (adjacent channel) leakage tests. See PTU Usage.

Also review Static DC Tests.

With static PTU tests it is possible to test the specified pins sequentially; i.e. one at a time, or in parallel. This is controlled by the `PartestOpt` argument to `ptu_partest()`. Using the PTU, the most benefit is obtained when testing multiple pins in parallel, thus the default is parallel. With static PTU tests, the Parametric Background Voltage may be used, if enabled (see Background Voltage Functions).

Note: when using `ptu_partest()`, the system software automatically controls connecting and disconnecting the PTU to pins. It is ***not necessary*** for user code to do this.

In software release h1.1.23, two new `ptu_partest()` overloads were added to support measurement averaging. Note the following:

- Measurement averaging is enabled by including the `PartestOpt iacc` argument.
- The number of measurements made to obtain the average is set using `iacc_count_set()`. Default = 10.
- The value set using `iacc_count_set()` is ignored when `measure()` = FALSE.
- When averaging is enabled the measurement average is compared to the PASS/FAIL test limits, set using PTU Current Test Limit Functions or PTU Voltage Test Limit Functions, to determine whether `ptu_partest()` returns PASS or FAIL.
- When Retrieving DC Test Results only the average value is returned, regardless of the number of measurements made.

Prior to executing the test the following parameters must be set up:

- Force-voltage or current value. See PTU Force-voltage Functions and PTU Force-current Functions.
- PASS/FAIL voltage or current-test limits. See PTU Current Test Limit Functions and PTU Voltage Test Limit Functions. These functions also set the test/measure range values.

- The system software provides a Built-in Settling Time to PTU current-tests. The user may use the `partime()` function to add additional settling time. See Parametric Settling Time.
- PTU voltage clamps. See PTU Voltage Clamp Functions.
- Enable/disable measurements using `measure()`. Note that PTU measurements use the system PMU(s) and typically must be performed sequentially. See Overview. Measured values can be retrieved by user code, see Retrieving DC Test Results.
- Background voltage. Applies only when `ptu_partest()` is executed with the `sequential PartestOpt`. In this mode, one pin at a time in the specified pin list is tested (its PTU is set to the force-voltage or current value) while the PTU(s) of the other pins in the specified pin list are set to the background voltage. See Background Voltage Functions.
- If any PTU(s) are statically connected to their pin(s) additional rules apply. See PTU Tests on Statically Connected Pins.

Usage

The following functions execute a static PTU test:

```
PFState ptu_partest(PassCond pass_cond,
                    DutPin *pDutPin,
                PartestOpt test_type DEFAULT_VALUE( parallel ) );
PFState ptu_partest(PassCond pass_cond,
                    PinList* pPinList,
                  PartestOpt test_type DEFAULT_VALUE( parallel ));
```

Note: the following functions were first available in software release h1.1.23.

```
PFState ptu_partest(PassCond pass_cond,
                    PinList* pPinList,
                    PartestOpt type,
                    PartestOpt accuracy );
PFState ptu_partest(PassCond pass_cond,
                    DutPin *pDutPin,
                    PartestOpt type,
                    PartestOpt accuracy );
```

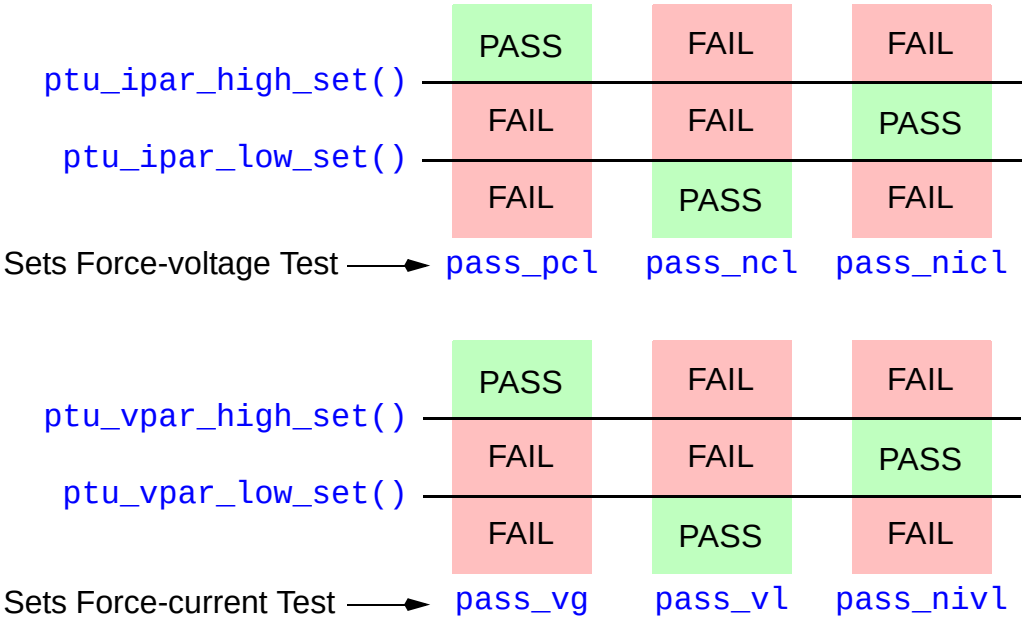
where:

pass_cond determines whether the PTU test is forcing current or voltage, and how the PASS/FAIL test limits will be used. **pass_cond** values are defined using the PassCond enumerated type. Operation is defined in the following table:

Table 3.14.9.0-1 PTU Test Force & PASS/FAIL Limit Options

Pass Condition	Comments
pass_pcl	*pcl* = Positive Current Limit. Force the voltage set using ptu_vpar_force_set(). Test/measure PTU output current. Pass if current is greater than the value set using ptu_ipar_high_set().
pass_ncl	*ncl* = Negative Current Limit. Force the voltage set using ptu_vpar_force_set(). Test/measure PTU output current. Pass if current is less than the value set using ptu_ipar_low_set().
pass_nicl	*nicl* = Not In Current Limit. Force the voltage set using ptu_vpar_force_set(). Test/measure PTU output current. Pass if current is between the values set using ptu_ipar_high_set() and ptu_ipar_low_set().
pass_vg	*vg* = Voltage Greater. Force the current set using ptu_ipar_force_set(). Test/measure PTU output voltage. Pass if voltage is greater than the value set using ptu_vpar_high_set().
pass_vl	*vl* = Voltage Less. Force the current set using ptu_ipar_force_set(). Test/measure PTU output voltage. Pass if voltage is less than the value set using ptu_vpar_low_set().
pass_nivl	*nivl* = Not In Voltage Limit. Force the current set using ptu_ipar_force_set(). Test/measure PTU output voltage. Pass if voltage is between the values set using ptu_vpar_high_set() and ptu_vpar_low_set()

The diagrams below shows this same information graphically. The upper diagram applies when forcing voltage and testing current. The lower diagram applies when forcing current and testing voltage:



pDutPin identifies one pin to be tested. In Multi-DUT Test Programs, only pins of DUT(s) currently in the Active DUTs Set (ADS) are affected. The specified DutPin must be mapped to a signal pin in the Pin Assignment Table.

pPinList specifies which pins(s) are to be tested. In Multi-DUT Test Programs, only pin(s) of DUT(s) currently in the Active DUTs Set (ADS) are affected. The pin list must only contain DutPins mapped to a signal pins in the Pin Assignment Table.

test_type is optional, and is used to select several PTU test options. Legal values are of the PartestOpt enumerated type, but only the options noted in the table below are valid. Default = parallel:

Table 3.14.9.0-2 PTU Test Optional Arguments

Optional Arguments	Comments
parallel	Default. Test all pins in **pPinList** in parallel. Background voltage is not used. sequential is always enabled when measure() = TRUE
parallel_pmu	Not supported in PTU tests. Causes a fatal runtime error.
sequential	Sequentially test each pin in **pPinList**. Background voltage is used, if enabled (see Background Voltage Functions). sequential is always enabled when measure() = TRUE .
iacc	Enable measure averaging. Applies only when measure() = TRUE. See Description and Measurement Average Count Functions. When iacc is specified alone the test uses the sequential option.
no_iacc	Default. Disable measure averaging.

To specify both options it is necessary to use both the **type** and **accuracy** arguments (this capability was added in software release h1.1.23):

- **type** specifies the execution option (sequential, parallel, parallel_pmu)
- **accuracy** specifies iacc.

As noted above, iacc is only usable when measure() = TRUE.

ptu_partest() returns TRUE (PASS) or FALSE (FAIL). All pin(s) tested must PASS otherwise FAIL is returned. In Multi-DUT Test Programs, only DUT(s) in the Active DUTs Set (ADS) can affect test results.

Example

```
PFState result = ptu_partest( pass_nicl,
                              pl_leak_pins,
                              parallel );
```

3.14.10 PTU Dynamic Test Functions

See Per-pin Parametric Test Unit (PTU), PTU Functions, DC Sub-System Block Diagram.

Note: first available in software release h2.2.7/h1.1.7.

Description

The `ptu_ac_partest()` function is used to perform a dynamic DC parametric test using the PTUs. PTU Static Test Functions are documented separately.

In a dynamic PTU test, while a specified test pattern is executing each PTU will force a voltage or current and test or measure the opposite parameter.

Note: the information in Dynamic DC Tests is not correct for dynamic PTU tests. The correct operation described below. For reference, some of the differences between dynamic PTU and dynamic PMU tests are also noted below.

The PTU sub-system hardware design makes the operation of dynamic PTU Go/NoGo tests vs. dynamic PTU measurements different in significant ways. The switch between dynamic PTU Go/NoGo tests and measurements is controlled using the `measure()` function:

- `ptu_ac_partest()` when `measure()` = FALSE = Go/NoGo test.
- `ptu_ac_partest()` when `measure()` = TRUE = measurement test.

The remainder of this section includes:

- The Basics
- Dynamic PTU Measurement Test
- Dynamic PTU Go/NoGo Test
- Setup Checklist
- PTU vs. PMU Differences
- Usage
- Example

The Basics

The `ptu_ac_partest()` function executes a specified test pattern, and the desired pattern stop condition must be specified using the `stop_cond` argument. Options are listed in Usage, and include terminating pattern execution if a functional failure occurs or executing the pattern to completion, regardless of functional failures. Note that when executing a Dynamic PTU Go/NoGo Test, the functional strobes are routed to the DC comparators of the PTU(s) involved in the test. In this scenario, any latched PTU failure will be treated the same as latched functional fails, stopping pattern execution if `stop_cond = error`. See Error Flag vs. Error Latch.

`ptu_ac_partest()` will not return until test pattern execution terminates.

The `ptu_ac_partest()` function does not provide a sequential option; i.e. all pins in the specified pin list are tested concurrently, using the PTU associated with each pin.

Note: when using `ptu_ac_partest()` the system software automatically controls connecting and disconnecting the PTU to pins. It is ***not necessary*** for user code to do this.

`ptu_ac_partest()` will fail if any functional strobes fail.

Dynamic PTU Measurement Test

A dynamic PTU measurement test is performed when `ptu_ac_partest()` is executed and `measure()` = TRUE. Dynamic PTU Go/NoGo Tests are documented separately.

In general, the output of one PTU may be routed to the A/D converter (ADC) in each DC Test and Measure System, and the test pattern VCOMP signal triggers the ADC to make a measurement. This is the same ADC used for PMU tests, DPS and HV current tests and operation is as described in Dynamic DC Tests.

The `pPinList` argument to `ptu_ac_partest()`, identifies which PTU(s) (pin(s)) are to be tested. When measurements are enabled, functional strobes on these pins can't fail.

The `pass_cond` argument controls several things:

- It determines the PTU force parameter applied during the test (force voltage vs. current). This operates the same during Dynamic PTU Go/NoGo Tests.
- It determines the test parameter type; i.e. measure current or voltage. This will always be the opposite of the force parameter type. This operates the same during Dynamic PTU Go/NoGo Tests.

- It determines how the PASS/FAIL test limits are used. See Usage. This operates differently during Dynamic PTU Go/NoGo Tests.

When executing `ptu_ac_partest()` and `measure()` = TRUE the ADC will receive one trigger for each executed pattern instruction containing the `MAR VCOMP` instruction (Memory Test Patterns) or `VEC`/`RPT VCOMP`, `VAR VCOMP` or `VPINFUNC VCOMP` instruction (Logic Test Patterns). See Dynamic DC Tests.

Note: when executing `ptu_ac_partest()` and `measure()` = TRUE, if the test pattern does not trigger the ADC, `ptu_ac_partest()` will return invalid results. The system software cannot check for this error; i.e. it is the user's responsibility to ensure that at least one `VCOMP` trigger is issued by the pattern.

To determine the test result, once the test pattern execution ends, the system software retrieves the measured value(s) and compares them to the PASS/FAIL test limits, set using PTU Current Test Limit Functions and PTU Voltage Test Limit Functions. These same measurements can be retrieved, see Retrieving DC Test Results.

The ADC only stores one value. If the test pattern triggers the ADC more than once:

- Only the last measurement is used to determine PASS/FAIL
- Only the last measurement is returned when Retrieving DC Test Results.

As noted in Overview, each Site Assembly Board has 128 PTUs (one per pin) and 8 DC Test and Measure Systems; i.e. in hardware, 16 PTUs are associated with a given DC Test and Measure System. When executing `ptu_ac_partest()` and `measure()` = TRUE, it is an error if the specified pin list contains multiple PTUs which share a given DC Test and Measure System. When this rule is violated:

- `ptu_ac_partest()` will return immediately, without actually performing the test.
- The test result for all DUT(s) tested will be FAIL.
- Any measured values retrieved will be invalid (stale, etc.) See Retrieving DC Test Results

Dynamic PTU Go/NoGo Test

A dynamic PTU Go/NoGo test is performed when `ptu_ac_partest()` is executed and `measure()` = FALSE. Also see Dynamic PTU Measurement Test.

In general, during the pattern execution, the PTU pins being tested must be strobed from the test pattern. Each strobe causes the PTU's DC Comparators and Error Logic to sample the test parameter, comparing it to the PTU PASS/FAIL test limits to determine the test result. If the pattern strobe is latched (see Error Flag vs. Error Latch) and the comparator indicates a

fail condition the pin's error logic will latch the failure, which will result in the test failing. Note that a dynamic PTU Go/NoGo test operates differently than a similar dynamic PMU Go/NoGo test, for two reasons:

- Each PTU has its own set of DC comparators; i.e. a PTU DC comparator per-pin.
- The test pattern VCOMP signal is not routed to the PTU comparators and thus can't be used to trigger a PTU Go/NoGo test. Instead, a given PTU's DC comparator is triggered by the functional strobes for that pin. These are the same signals which normally strobe a pin's functional comparators. More below.

The PASS/FAIL test limits, set using PTU Current Test Limit Functions and PTU Voltage Test Limit Functions, are the DC comparator references used during the test.

The `pPinList` argument to `ptu_ac_partest()`, identifies which PTU(s) (pin(s)) are to be tested. As indicated below, these pins will not be functionally tested during the PTU test.

When executing `ptu_ac_partest()` and `measure()` = FALSE the `pass_cond` argument controls the following:

- It determines the PTU force parameter applied during the test (force voltage vs. current). This operates the same during Dynamic PTU Measurement Tests.
- It determines the test parameter type; i.e. test current or voltage. This will always be the opposite of the force parameter type. This operates the same during Dynamic PTU Measurement Tests.

As described below, the `pass_cond` argument does not affect how the PASS/FAIL test limits are used when performing a dynamic PTU Go/NoGo test. This is different than when executing Dynamic PTU Measurement Tests.

Regarding the use of functional strobes to trigger the PTU DC comparators, note the following:

- During a pattern execution, a given pin's functional strobe signal can be used for one purpose: either to strobe for functional fails, using the pin's functional comparators, or to strobe that pin's PTU DC comparator. The test pattern strobe operation and programming methods are identical for both applications. Strobe timing, timing rules, and window vs. edge strobe operation is also identical.
- The `pPinList` argument to `ptu_ac_partest()`, which identifies which PTU(s) (pin(s)) are to be tested, also configures a MUX for those pin(s), to route that pin's strobe signal to its PTU DC comparators. This means that when executing `ac_ptu_partest()`, the pins in the `pPinList` will not be strobed for functional failures. The MUX is controlled automatically and restored to normal functional operation by `ptu_ac_partest()`.

- The output of a PTU's DC comparator is routed to the same error logic that is used for functional testing. The remainder of the system does not know or care whether a given error (failing strobe) originated from a pin's PTU DC comparators or from the pin's functional comparators. This means that the following operate the same for both applications:
  - Error latch vs. error flag operation (see Error Flag vs. Error Latch).
  - Test pattern branch-on-error/abort, for both Memory Test Patterns and Logic Test Patterns.
  - MAR RESET/NOLATCH instructions (Memory Test Patterns).
  - VEC/RPT RESET/NOLATCH and VAR RESET/NOLATCH instructions (Logic Test Patterns).
  - The Error Catch RAM (ECR) will capture PTU strobe fails the same as functional fails.
- When ptu_ac_partest() is executed and measure() = FALSE, the PassCond argument only determines the PTU force parameter (force current vs. voltage) and tested parameter; i.e. it has no effect on how the PASS/FAIL test limits, set using PTU Current Test Limit Functions and PTU Voltage Test Limit Functions, are used. It is the strobe type(s) generated from the test pattern, for each pin, which determine how the PASS/FAIL test limits are used. Note that it is possible, with a single execution of ptu_ac_partest(), to test any or all of the 4 possible test conditions, on a per-PTU (per-in) basis. The following table shows how the various strobe types operate:

Equivalent Voltage Test Condition	Equivalent Current Test Condition	Pattern Strobe Type
pass_vg	pass_pcl	Strobe-high
pass_nivl	pass_nicl	Strobe-Z
pass_vl	pass_ncl	Strobe-low
n/a	n/a	Strobe-V (valid). This strobe will pass if the test parameter is either above the high limit or below the low limit. There is no equivalent for the other DC test types.

- If a given PTU is strobed by more than one strobe type it is not possible to determine which one(s) failed; i.e. did a strobe-high fail (equivalent to `pass_vg`) or strobe-Z fail (equivalent to `pass_nivl`), etc.
- The strobes generated by the APG data generator and APG chip selects only provide two strobe types: strobe-high and strobe-low; i.e. not strobe-Z or strobe-V. This might seem to indicate that a pure Memory Test Pattern cannot to do the equivalent of `pass_nicl`/`pass_nivl` using these APG data sources. However, these strobe types can be obtained in a memory pattern by using the pin scrambler, to select a scramble map which has `t_strobe_valid` and/or `t_strobe_mid` mapped to pins involved in the PTU test. In Logic Test Patterns, the Z and V tokens enable these strobe types.

Setup Checklist

Prior to executing `ptu_ac_partest()` the following parameters must be setup:

- Force-voltage or current value. See PTU Force-voltage Functions or PTU Force-current Functions.
- The PTU voltage clamps (if forcing current). See PTU Voltage Clamp Functions
- PASS/FAIL voltage or current-test limits. See PTU Current Test Limit Functions and PTU Voltage Test Limit Functions. When forcing voltage, the PTU Current Test Limit Functions functions also set the current range value.
- The `partime()` function, used to add additional settling time to the Built-in Settling Time, is not normally useful in dynamic DC tests. This is because this delay will occur after programming the DC circuitry but before executing the functional test pattern; i.e. at a non-useful time. See Parametric Settling Time.
- Enable/disable measurements using `measure()`.
- For dynamic tests, the test will also fail if any functional strobes fail. Thus, proper digital PE levels and timing will affect test results, as does the test pattern executed.
- If any PTU(s) are statically connected to their pin(s) additional rules apply. See PTU Tests on Statically Connected Pins.

PTU vs. PMU Differences

Some important differences between PMU and PTU operation are outlined here, purely for reference.

- Errors from a given PTU's DC comparator can set both that pin's error flag and error latch, depending on the same `LATCH`/`NOLATCH` test pattern options used during functional testing, see Error Flag vs. Error Latch. Once any error latch is set

that pin will fail, regardless of the use of MAR/VEC/VAR RESET. This operation is different than the dynamic PMU, DPS and HV Go/NoGo tests, which only affect a DC error flag, which can be RESET from the test pattern.

- Also unlike PMU, DPS and HV Go/NoGo tests, it is not possible to *enable* the PTU DC comparators from the computer. Thus, it is not possible to enable the comparators, execute the pattern, and afterwards check to see if any transients tripped any comparator thresholds; e.g. all PTU Go/NoGo tests are triggered by functional strobes from an executing test pattern. This is the reason the `comp_cond` argument is not used by `ptu_ac_partest()`.
- As indicated above, when performing dynamic PMU, DPS and HV DC tests, how the PASS/FAIL test limits are used is determined by the `PassCond` argument to the corresponding test function. This is also true when executing Dynamic PTU Measurement Tests but not during Dynamic PTU Go/NoGo Tests. During the former, only one test condition can be tested for each test pattern execution: `pass_ncl`, `pass_nivl`, `pass_vl`, etc. During Dynamic PTU Go/NoGo Test, any combination of these limits can be used, on a per-PTU (per-pin) basis. See Dynamic PTU Go/NoGo Test.

Usage

The following function is used to perform dynamic PTU tests on one or more pins:

```
PFState ptu_ac_partest(PassCond pass_cond,
                       PinList* pPinList,
                       Pattern *pPattern,
                       PatStopCond stop_cond );
```

where:

`pass_cond` determines the PTU force parameter type (current or voltage) and test/measure parameter type (opposite of the force type). And, during Dynamic PTU Measurement Tests (but not Dynamic PTU Go/NoGo Tests) determines how the PASS/FAIL test limits will be

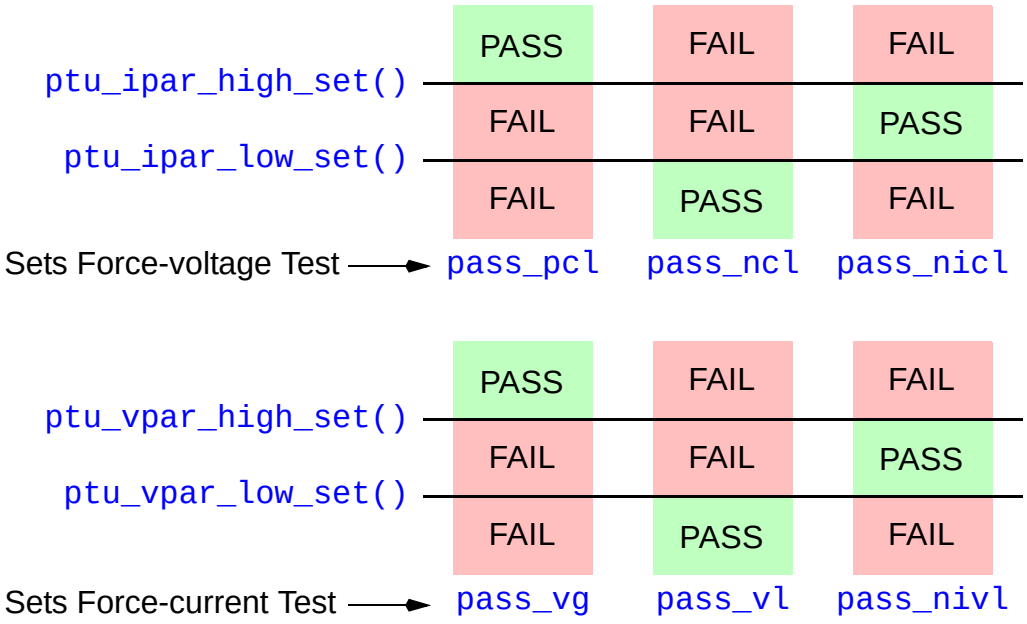
used. **pass_cond** values are defined using the PassCond enumerated type. Operation is defined in the following table:

Table 3.14.10.0-1 PTU Test Force & PASS/FAIL Limit Options

Pass Condition	Comments
pass_pcl	*pcl* = Positive Current Limit. Force the voltage set using ptu_vpar_force_set() and test/measure PTU output current. Measurements pass if current is greater than the value set using ptu_ipar_high_set().
pass_ncl	*ncl* = Negative Current Limit. Force the voltage set using ptu_vpar_force_set() and test/measure PTU output current. Measurements pass if current is less than the value set using ptu_ipar_low_set().
pass_nicl	*nicl* = Not In Current Limit. Force the voltage set using ptu_vpar_force_set() and test/measure PTU output current. Measurements pass if current is between the values set using ptu_ipar_high_set() and ptu_ipar_low_set().
pass_vg	*vg* = Voltage Greater. Force the current set using ptu_ipar_force_set() and test/measure PTU output voltage. Measurements pass if voltage is greater than the value set using ptu_vpar_high_set().
pass_vl	*vl* = Voltage Less. Force the current set using ptu_ipar_force_set() and test/measure PTU output voltage. Measurements pass if voltage is less than the value set using ptu_vpar_low_set().
pass_nivl	*nivl* = Not In Voltage Limit. Force the current set using ptu_ipar_force_set() and test/measure PTU output voltage. Measurements pass if voltage is between the values set using ptu_vpar_high_set() and ptu_vpar_low_set()

The diagrams below shows how the PASS/FAIL limits are used during Dynamic PTU Measurement Tests. The upper diagram applies when forcing voltage and testing current.

The lower diagram applies when forcing current and testing voltage. See Dynamic PTU Go/NoGo Test for equivalent operation using functional strobes:



`pPinList` specifies which PTU(s) are to be affected. In Multi-DUT Test Programs, only pin(s) of DUT(s) currently in the Active DUTs Set (ADS) are affected. The pin list must only contain `DutPin`s mapped to a signal pins in the Pin Assignment Table. When performing s Dynamic PTU Measurement Test **`pPinList`** may only contain one pin for each ADC, see Dynamic PTU Measurement Test.

`pPattern` identifies the test pattern to be executed.

stop_cond controls how test pattern execution terminates. Legal **stop_cond** values are defined using the PatStopCond enumerated type. Note that majority of tests will use the error or finish options:

Table 3.14.10.0-2 Pattern Execution Stop Condition Options

Stop Condition	Summary Description
finish	Execute pattern to completion, regardless of errors. When execution finishes, the PE error latches and DC Error Flags are examined. If an error was latched, the result of ptu_ac_partest() is FAIL, otherwise the result is PASS
error	Stop pattern execution on first functional or DC Error Flag error and sets the result of ptu_ac_partest() to FAIL. Note that the pattern generator may continue for one or more cycles past where the error occurred, depending on the cycle time and where in the cycle the error was detected.
fullec	Execute pattern to completion. Enable full ECR, row error catch, and column error catch to capture errors during pattern execution. This argument should be used when performing Redundancy Analysis (RA) or using BitmapTool.
LEC_only_errors	Enable full ECR, row error catch, and column error catch. Capture the first 2Meg (2^{21}-6) failing vectors. Intended for use when the ECR is used as an Logic Error Catch (LEC).
LEC_first_vectors	Enable full ECR, row error catch, and column error catch. Capture the first 2Meg (2^{21}-6) vectors executed. Ignores PASS/FAIL. Intended for use when the ECR is used as an Logic Error Catch (LEC).
LEC_last_vectors	Enable full ECR, row error catch, and column error catch. Capture the last 2Meg (2^{21}-6) vectors executed. Ignores PASS/FAIL. Intended for use when the ECR is used as an Logic Error Catch (LEC).
LEC_before_error	Enable full ECR, row error catch, and column error catch. Capture the first failing vector plus the previous 2Meg (2^{21}-6) vectors executed. Intended for use when the ECR is used as an Logic Error Catch (LEC).

Table 3.14.10.0-2 Pattern Execution Stop Condition Options *(Continued)*

Stop Condition	Summary Description
LEC_after_error	Enable full ECR, row error catch, and column error catch. Capture the first failing vector plus the next 2Meg (2^{21}-6) vectors executed. Intended for use when the ECR is used as an Logic Error Catch (LEC).
LEC_center_error	Enable full ECR, row error catch, and column error catch. Capture the first failing vector plus up to 512K vectors executed before the failure and up to 512K vectors executed after the failure. Intended for use when the ECR is used as an Logic Error Catch (LEC).
Note: in parallel test applications, the test pattern must be executed to completion, to ensure that DUT(s) which don't fail are completely tested. In other words, halting the pattern early (error) because one or more DUT(s) failed prevents DUT(s) which PASS from being completely tested. This is BAD.	

ptu_ac_partest() returns TRUE (PASS) or FALSE (FAIL). All pin(s) tested must PASS otherwise FAIL is returned. All latched functional strobes must PASS. In Multi-DUT Test Programs, only DUT(s) in the Active DUTs Set (ADS) can affect test results.

Example

```
PFState pf = ptu_ac_partest(pass_nicl,
                            pl_leakpins,
                            leaksetup_pat,
                            finish );
```

3.14.11 PTU as Voltage/Current Source

See Per-pin Parametric Test Unit (PTU), PTU Functions, PTU Connect/Disconnect Functions.

Description

The `ptu_connect()` function is used to close the solid-state switch(es) connecting a PTU to its associated DUT pin. The `ptu_disconnect()` function is used to open these switch(es).

Note: standard PTU tests, i.e. `ptu_partest()` ***automatically*** control PTU connections and disconnections to DUT pins; i.e. it is ***NOT*** necessary to use `ptu_connect()` or `ptu_disconnect()` when performing these tests. These functions are used when the PTU is being used as a statically connected voltage or current source.

Arguments passed to the `ptu_connect()` function are used to:

- Specify which pins are to be connected to their corresponding PTU.
- Optionally specify whether to force-voltage or current.
- Optionally specify a current range (applies to force-current use only).

Connection rules:

- A given PTU can only be connected or disconnected from its associated pin.
- When `ptu_connect()` connects a given PTU to its pin, it will remain connected until disconnected by calling `ptu_disconnect()`. Connections remain across test blocks, and are not disconnected at the end of the Sequence & Binning Table execution.
- Executing `ptu_connect()` does not disconnect any pin(s) previously connected to their associated PTU. Executing `ptu_disconnect()` only disconnect the specified pin(s) from their associated PTU.
- It is not legal to connect a PTU to a pin which is currently statically connected to a PMU. If this is attempted, a warning is issued for each violating pin and the PTU connection to these pins is not completed. Other pins which are not violating the rule will be connected to their associated PTU.
- By default, `ptu_connect()` sets the force mode to force voltage (`FORCEV`). The `force_type` argument can be used to explicitly specify the PTU force mode.
- It is not possible to change the force mode (`FORCEV` vs. `FORCEI`) while a PTU is connected to its pin.

- Executing `ptu_vpar_force_set()` or `ptu_ipar_force_set()` has no immediate effect on any PTU(s) which are not currently connected to their pins. The force value is saved but no hardware changes occur. For any PTU(s) which are currently connected to their pin(s) the force value changes at the pin immediately.
- Once a given PTU is connected to its pin using `ptu_connect()`, no current range changes are permitted. If a range change is attempted, a warning is issued and no changes are made to the hardware, including setting a different force value. Testing will otherwise continue
- PTU voltage clamps, set using `ptu_vclamp_set()`, will limit the output voltage of the PTU. The voltage clamp is only active when the PTU is in the `FORCEI` mode. At test program initialization the voltage clamps are disabled; user code must execute `ptu_vclamp_set()` to set desired clamp values.

Note: *careful consideration* must be given to setting proper clamp values when using the PTU to force current. The PTU output voltage range is -2V to +12V. When forcing current, if the DUT doesn't adequately load the PTU, only the voltage clamps will prevent the PTU from reaching one of these voltages.

PTU voltage/current ranges operate as follows:

- The PTU has a single voltage range:

Table 3.14.11.0-1 PTU Force-voltage Range

Voltage Range	LSB
-2V to +12V	1mV

Note: the PTU output voltage and current capabilities are affected by the selected current range. See PTU Operating Area.

The information below applies to current range operation.

- By default, when forcing voltage the current range is set to the coarsest value.
- By default, when forcing current the force range is set based on the last executed `ptu_ipar_force_set()`.

- By default, the measure-current range defaults to that set by the last execution of ptu_ipar_high_set() and ptu_ipar_low_set(). Optionally, the meas_range argument can be used to specify a current measure range value, as follows:

Table 3.14.11.0-2 PTU Force-Current Ranges

Current Range	Range	Resolution
±2uA	range1	1nA
±8uA	range2	4nA
±32uA	range3	16nA
±128uA	range4	64nA
±512uA	range5	256nA
±2mA	range6	1uA
±8mA	range7	4uA
±32mA	range8	16uA
Note: the PTU output voltage and current capabilities are affected by the selected current range. See PTU Operating Area.		

PTU Tests on Statically Connected Pins

Normally, to perform conventional DC parametric tests using ptu_partest() it is NOT necessary to use ptu_connect() or ptu_disconnect(), the system software automatically controls connections, as needed. However, it is possible to use ptu_connect() prior to ptu_partest(). The following rules apply:

- When (some) pins have been explicitly connected to their PTU, the system software supports two test scenarios:
  - The members of the pin list specified in ptu_partest() exactly match the set of pins currently statically connected to their PTU.
  - The members of the pin list specified in ptu_partest() contain NO pins currently statically connected to their PTU.

  - If these rules are violated, a warning is issued and `ptu_partest()` is terminated early, returning FAIL for each DUT in the Active DUTs Set (ADS). And, if measured values were enabled (see `measure()`) any values retrieved will be invalid (see Retrieving DC Test Results).
- Executing `ptu_partest()` has no effect on PTU(s) which are statically connected to pin(s).
- When executing `ptu_partest()`, if any PTU(s) are currently connected to their pin(s), it is an error to specify a `pass_cond` which sets a force mode which is different than the mode currently being forced.
- When a Force-V/Measure-I `ptu_partest()` is invoked on PTU(s) which are connected to their pin(s), if any currently set values of `ptu_ipar_high_set()` and/or `ptu_ipar_low_set()` will cause a range change, a warning will be issued, the `ptu_partest()` exits, returning FAIL for each DUT in the Active DUTs Set (ADS). And, if measured values were enabled (see `measure()`), any values retrieved will be invalid (see Retrieving DC Test Results). Note that this restriction does not apply to Force-I/Measure-V tests because the PTU has a single voltage range and a range change cannot occur.

Limitations

The following limitations exist:

- It is *not* possible to set up `FORCEV` and `FORCEI` at the same time, even using PTUs on different pins. It is possible to set up different conditions for `FORCEV` - ***or*** - `FORCEI` by calling `ptu_connect()` multiple times, each with a pin list which does not intersect the other.

Usage

The following functions are used to statically connect the PTU(s) to the specified pin(s):

```
void ptu_connect( DutPin *pDutPin );
void ptu_connect( PinList* pPinList );
void ptu_connect( DutPin *pDutPin, Range meas_range );
void ptu_connect( PinList* pPinList, Range meas_range );
void ptu_connect( DutPin *pDutPin, PMUMode force_type );
void ptu_connect( PinList* pPinList, PMUMode force_type );
void ptu_connect( DutPin *pDutPin,
                  PMUMode force_type,
                  Range meas_range );
```

```
void ptu_connect( PinList* pPinList,
                  PMUMode force_type,
                  Range meas_range );
```

The following functions are used to disconnect PTU(s) from the specified pin(s):

```
void ptu_disconnect( DutPin *pDutPin );
void ptu_disconnect( PinList* pPinlist );
```

where:

pDutPin identifies one pin to be connected or disconnected to/from their corresponding PTU. In Multi-DUT Test Programs, only pins of DUT(s) currently in the Active DUTs Set (ADS) are affected. The specified DutPin must be mapped to a signal pin in the Pin Assignment Table.

pPinList specifies which pin(s) will be statically connected or disconnected to/from their corresponding PTU. In Multi-DUT Test Programs, only pin(s) of DUT(s) currently in the Active DUTs Set (ADS) are affected. The pin list must only contain DutPins mapped to signal pins in the Pin Assignment Table.

range is used to select a PTU current measure range. See Description. Values must be of the Range enumerated types, but the preferred method is to use values from Table 3.14.4.0-1. This parameter is ignored when a given PTU is in the force-voltage mode.

force_type is used to specify whether the PTU will force current (FORCEI) or voltage (FORCEV). See Description.

Example

```
ptu_connect( pl_vref_pin );
ptu_disconnect( pl_vref_pin );
```

3.15 Pin Electronics Voltages/Currents

See Software.

This section documents the various DC voltage and current parameters associated with the Pin Electronics (PE) circuitry used during functional tests:

- Pin Electronics Levels
- Types, Enums, etc.
- PE: Drive Voltages: VIH/VIL
- VIHH Voltage
- PE Comparator Voltages: VOH/VOL
- PE Load Reference Voltage: VZ
- `rl_set(), rl_get()`
- `rl_bitmask_get()`
- `rl_ohms_get()`
- 50-ohm Termination Voltage: VTT
- `pe_driver_mode_set(), pe_driver_mode_get()`
- PE Connect/Disconnect Functions

3.15.1 Pin Electronics Levels

See Pin Electronics Voltages/Currents, Pin Electronics (PE).

Each Pin Electronics (PE) channel has the following programmable voltage and current parameters:

Table 3.15.1.0-1 Pin Electronics DC Parameters

Param.	Set Func.	Purpose
VIL	`vil()`	Logic-0 drive voltage.
VIH	`vih()`	Logic-1 drive voltage.
VIHH	`vihh()`	Third drive level. See VIHH Maps and Vihh Mode in Magnum PE Driver Modes.
VOL	`vol()`	Strobe-0 comparator reference voltage.
VOH	`voh()`	Strobe-1 comparator reference voltage.
VZ	`vz()`	Reference voltage for programmable driver termination. See Vz Mode in Magnum PE Driver Modes .
VTT	`vtt()`	Reference voltage for 50Ω driver termination. See Vtt Mode in Magnum PE Driver Modes.

3.15.2 Types, Enums, etc.

See Pin Electronics Voltages/Currents.

The following declarations are used to specify the termination resistance value when the Magnum PE Driver Modes is in Vz Mode (see Magnum PE Driver Modes). This is programmed using the `vz()` function (see RL Values):

```
#define VZRS1 0x01
#define VZRS2 0x02
#define VZRS3 0x04
#define VZRS4 0x08
#define VZRS5 0x10
```

```
#define VZRS6 0x20
#define VZRS7 0x40
#define VZRS8 0x80
```

The `PEDriverMode` enumerated type is used to access the Magnum PE Driver Modes:

```
enum PEDriverMode { t_pe_nomode, t_pe_vzmode, t_pe_vihhmode,
                    t_pe_vttmode, t_pe_dclkmode};
```

3.15.3 PE: Drive Voltages: VIH/VIL

See PE Driver, Pin Electronics Voltages/Currents.

Description

The `vil()` and `vih()` functions are used to set or get the VIL and VIH PE driver voltage.

During functional tests, the PE Driver has several possible states, as controlled from the executing test pattern and timing system:

- Drive logic-0 = VIL voltage.
- Drive logic-1 = VIH voltage.
- Drive to third voltage level = VIHH (see VIHH Voltage and VIHH Maps).
- Tri-state to selectable resistor terminated to VZ voltage (see `vz()`). VIHH not available.
- Tri-state to 50Ω terminated to VTT voltage (see `vtt()`).
- Tri-state to un-terminated line (no VTT, no VZ)

Using Magnum 1, on a per-pin basis, not all of these states are available/usable at one same time, see PE Driver and Magnum PE Driver Modes.

Note the following:

- Each pin of the Magnum 1/2/2x Site Assembly Board has independent VIL and VIH capabilities.

- When using Magnum 1/2/2x to execute a Maverick-II program which uses the `vil_offset()` and/or `vih_offset()` functions, the operation of these functions (both set and get operation) is preserved. All 16 pins on a Maverick-II PE board share a common VIL reference and a common VIH reference. Each pin also has a separate VIL offset reference and VIH offset reference, allowing VIL and/or VIH to be set to different values on each pin. In Maverick-II hardware, VIL and VIH are each implemented as a per-board *rail* value, which sets the base value for all pins on the board, plus a per-pin offset. The offset range is ±2.5V.
- During initial program load, all PE driver voltages are set to 0V.
- During execution of the Sequence & Binning Table user-code determines the value of each parameter.
- When Sequence & Binning Table execution stops, all PE driver voltages are set to 0V via the `builtin_after_testing_block`.
- These levels can also be statically driven to the DUT (i.e. without executing a test pattern) using the `setpin()`, `pin_dc_state_set()`, `set_address()`, `set_data()` and `set_chips_on()`functions.
- These levels can also be modified from an executing test pattern. See Controlling PE Levels from the Test Pattern.

Note: proper driver operation requires that `vih()` be programmed to a value greater than `vil()`. If `vih_offset()` and/or `vil_offset()` are used, `vih()` + `vih_offset()` must be greater than `vil()` + `vil_offset()`.

Usage

The following function sets the VIH/VIL level for all pins:

```
void vih( double Value );
void vil( double Value );
```

The following function sets the VIH/VIL level for one pin:

```
void vih( double Value, DutPin *pDutPin );
void vil( double Value, DutPin *pDutPin );
```

The following function sets the VIH/VIL level for all pins in the specified pin list:

```
void vih( double Value, PinList* pPinList );
void vil( double Value, PinList* pPinList );
```

The following function sets the VIH/VIL offset level for one pin:

```
void vih_offset( double Value, DutPin *pDutPin );
void vil_offset( double Value, DutPin *pDutPin );
```

The following function sets the VIH/VIL offset level for all pins in the specified pin list. Not usable on Maverick-I:

```
void vih_offset( double Value, PinList* pPinList );
void vil_offset( double Value, PinList* pPinList );
```

The following functions return the currently programmed VIH/VIL level for the specified pin:

```
double vih( DutPin *pDutPin );
double vil( DutPin *pDutPin );
```

The following function returns the currently programmed VIH/VIL offset level for the specified pin:

```
double vih_offset( DutPin *pDutPin );
double vil_offset( DutPin *pDutPin );
```

where:

`Value` specifies the desired voltage level. Units may be used (see Specifying Units). Legal values for each voltage are shown below:

Table 3.15.3.0-1 PE Driver Levels Range & Resolution

Parameter	Range	Resolution	
VIH	-1V to +7V	5mV	
VIL	-1V to +7V	5mV	
VIH Offset	±2.5V	5mV	Supported for backwards compatibility with Magnum 1 only.
VIL Offset	±2.5V	5mV	

Note: proper driver operation requires that `vih()` be programmed to a value greater than `vil()`. If using the offset values, `vih()` + `vih_offset()` must be greater than `vil()` + `vil_offset()`.

`pDutPin` is used in two contexts:

- In the setter functions, identifies one pin to be programmed. In Magnum 1, 2 & 2x Parallel Tests, the pins of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.
- In the getter functions, identifies one pin to be read.

`pPinList` specifies which pin(s) are to be programmed. In Multi-DUT Test Programs, the pins of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **`pPinList`** must only contains pins mapped to a signal pins in the Pin Assignment Table.

The `vih()` and `vil()` getter functions return the currently programmed value. The value is retrieved for the first DUT in the Active DUTs Set (ADS).

Examples

```
vih( 3.5 V );
vih( 3500 MV );
```

3.15.4 VIHH Voltage

See PE Driver, Pin Electronics Voltages/Currents, VIHH Maps, PTU, Magnum PE Driver Modes.

Description

The `vihh()` function is used to program the VIHH voltage level or get the currently programmed value.

Note the following:

- VIHH is supplied by the Per-pin Parametric Test Unit (PTU), which is set to operate on the ±32mA range when a given pin's PE driver mode is set to Vihh Mode (see Magnum PE Driver Modes). This range selects the 81 output termination, thus the actual VIHH output voltage will be affected by the VIHH current supplied to the DUT. See PTU Operating Area. Note that `rl_set()` does not affect the output termination for pins in Vihh Mode.
- During initial program load VIHH is set to 0V.

- During execution of the Sequence & Binning Table user code determines the value of VIHH.
- When Sequence & Binning Table execution stops VIHH is set to 0V via the `builtin_after_testing_block`.
- VIHH can be modified from an executing test pattern. See Controlling PE Levels from the Test Pattern.
- During test pattern execution, the VIHH selection can also affected by the over-programming logic (see Over-programming Controls and Parallel Test). In simple terms, when enabled, this logic prevents over-programming of programmable devices by temporarily inhibiting the switch to the VIHH voltage when certain conditions are met. See Over-programming Controls and Parallel Test.

In addition to programming the VIHH voltage level, actual use of the VIHH level has several additional requirements. See VIHH Maps.

Usage

The following function sets the VIHH voltage level on all pins:

```
void vihh( double Value );
```

The following function sets the VIHH level on one pin:

```
void vihh( double Value, DutPin *pDutPin );
```

The following function sets the VIHH level on all pins in the specified pin list:

```
void vihh( double Value, PinList* pPinList );
```

The following function returns the currently programmed VIHH voltage level for one pin:

```
double vihh( DutPin *pDutPin );
```

where:

`Value` specifies the desired VIHH voltage. Units may be used (see Specifying Units). Legal values for VIHH are shown below:

Table 3.15.4.0-1 VIHH Voltage Level Range & Resolution

Parameter	Range	Resolution	
VIHH	0V to +12.5V	5mV	

`pDutPin` is used in two contexts:

- In the setter functions, identifies one pin to be programmed. In Multi-DUT Test Programs, the pins of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.
- In the getter functions, identifies one pin to be read.

`pPinList` specifies which pin(s) are to be programmed. In Multi-DUT Test Programs, the pins of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **`pPinList`** must only contains pins mapped to a signal pins in the Pin Assignment Table.

The `vihh()` getter function returns the currently programmed value. In Multi-DUT Test Programs, the value is retrieved from the first DUT in the Active DUTs Set (ADS).

Example

```
vihh( 10 V );
vihh( 11.5 V, TM_pins ); // Pin list
vihh( 11.5 V, one_pin ); // DutPin
double value = vihh( one_pin ); // DutPin
```

3.15.5 PE Comparator Voltages: VOH/VOL

See PE Comparators, Pin Electronics Voltages/Currents.

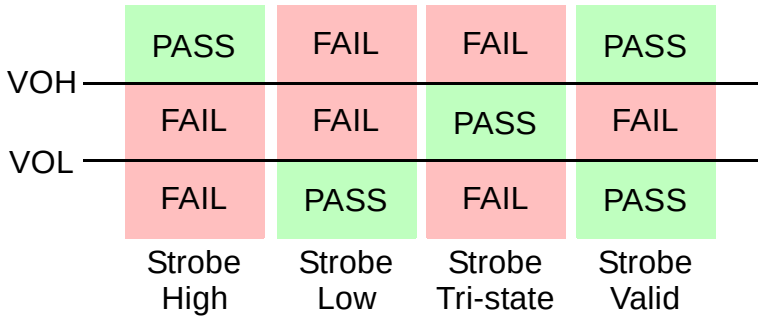
Description

The `vol()` and `voh()` functions are used to set or get the reference voltages for the PE Comparators.

During functional tests, the PE Comparators are used to sample (strobe) the output of the DUT, to determine whether that output is logically correct in both the time domain and voltage domain.

The comparator reference voltages are (conventionally) identified as VOL (voltage output low) and VOH (voltage output high).

Magnum 1/2/2x can functionally test for four logic states:



When a DUT output is strobed for a logic-1 (high), the strobe will pass if the DUT output voltage is above the VOH comparator reference for the duration of a window strobe or at the time of an edge strobe. Similarly, when a DUT output is strobed for a logic-0 (low), the strobe will pass if the DUT output voltage is below the VOL comparator reference for the duration of a window strobe or at the time of an edge strobe.

Note the following:

- VOL and VOH are per-pin parameters.
- During initial program load, VOL and VOH are set to 0V.
- During execution of the Sequence & Binning Table, user code determines the values of VOL and VOH.
- When Sequence & Binning Table execution stops, VOL and VOH are set to 0V via the `builtin_after_testing_block`.
- VOL and VOH can be programmed from an executing test pattern. See Controlling PE Levels from the Test Pattern.

Usage

The following functions set the VOH/VOL level for all pins:

```
void voh( double Value );
void vol( double Value );
```

The following function sets the VOH/VOL level for one pin:

```
void voh( double Value, DutPin *pDutPin );
void vol( double Value, DutPin *pDutPin );
```

The following function sets the VOH/VOL level for all pins in the specified pin list:

```
void voh( double Value, PinList* pPinList );
```

```
void vol( double Value, PinList* pPinList );
```

The following functions return the currently programmed VOH/VOL level for the specified pin:

```
double voh( DutPin *pDutPin );
double vol( DutPin *pDutPin );
```

where:

`Value` specifies the desired voltage level. Units may be used (see Specifying Units). Legal values for each voltage are shown below:

Table 3.15.5.0-1 PE Comparator Levels Range & Resolution

Parameter	Range	Resolution
VOH	-1V to +7V	5mV
VOL	-1V to +7V	5mV

`pDutPin` is used in two contexts:

- In the setter functions, identifies one pin to be programmed. In Multi-DUT Test Programs, the pins of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.
- In the getter functions, identifies one pin to be read.

`pPinList` specifies which pin(s) are to be programmed. In Multi-DUT Test Programs, the pins of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **`pPinList`** must only contains pins mapped to a signal pins in the Pin Assignment Table.

The `voh()` and `vol()` getter functions return the currently programmed value. In Multi-DUT Test Programs, the value is retrieved from the first DUT in the Active DUTs Set (ADS).

Examples

```
voh( 3.5 V );
voh( 3500 MV );
```

3.15.6 PE Load Reference Voltage: VZ

See PE Driver, Magnum PE Driver Modes, Pin Electronics Voltages/Currents.

Description

The `vz( )` setter function does the following:

- Programs the VZ load termination voltage for one or more pin(s)
- Optionally, selects the load termination resistance RL for one or more pin(s).

Note that the `rl_set()` function may also be used to select the RL value for one or more pin(s).

During functional test execution, for the pin(s) in Vz Mode (see Magnum PE Driver Modes), the RL selection determines the resistive load applied to the DUT output(s) when the PE Driver tri-states, as:

$$VZ = \frac{(IOL \times VOH) + (IOH \times VOL)}{IOL + IOH}$$

$$RL = \frac{VZ - VOL}{IOL}$$

$$RL = \frac{VZ - VOH}{IOH}$$

where VZ and RL are programmable, VOL and VOH are actual DUT output voltages (under load), and IOL and IOH are the load currents. As seen below, RL is selected from a discrete set of values which makes selecting an exact RL value suitable to simultaneously obtain both the desired IOL and IOH difficult.

vz( ) optionally selects the load termination resistance, RL, for the pin(s) in Vz Mode . There are 8 discrete RL values (±20%), see table below. The values in the table below include the nominal 50Ω (30Ω-85Ω) resistance of the FET switch which connects the PTU to the DUT:

Table 3.15.6.0-1 RL Values

RL Nominal	Min (-20%)	Max (+20%)	Actual RL (including 50Ω)	Min (/w 30Ω)	Max (/w 85Ω)	Select Token
500K	400K	600K	500K	400K	600K	VZRS1
125K	100K	150K	125K	100K	150K	VZRS2
31.25K	25K	35.5K	31.3K	25030	37585	VZRS3
7.81K	6248	9372	7.86K	6278	9457	VZRS4
1.95K	1560	2340	2.0K	1590	2425	VZRS5
500	400	600	550	430	685	VZRS6
125	100	150	175	130	235	VZRS7
31	25	37.5	81	55	122.5	VZRS8
The effect of the series 50Ω was not included in the 2 largest RL values.						

The 8 discrete RL values can be combined (in parallel only) to obtain additional resistance values. However, as shown in the graph below, many parallel combinations result in effectively the same value. This graph does include the 50Ω FET switch resistance:

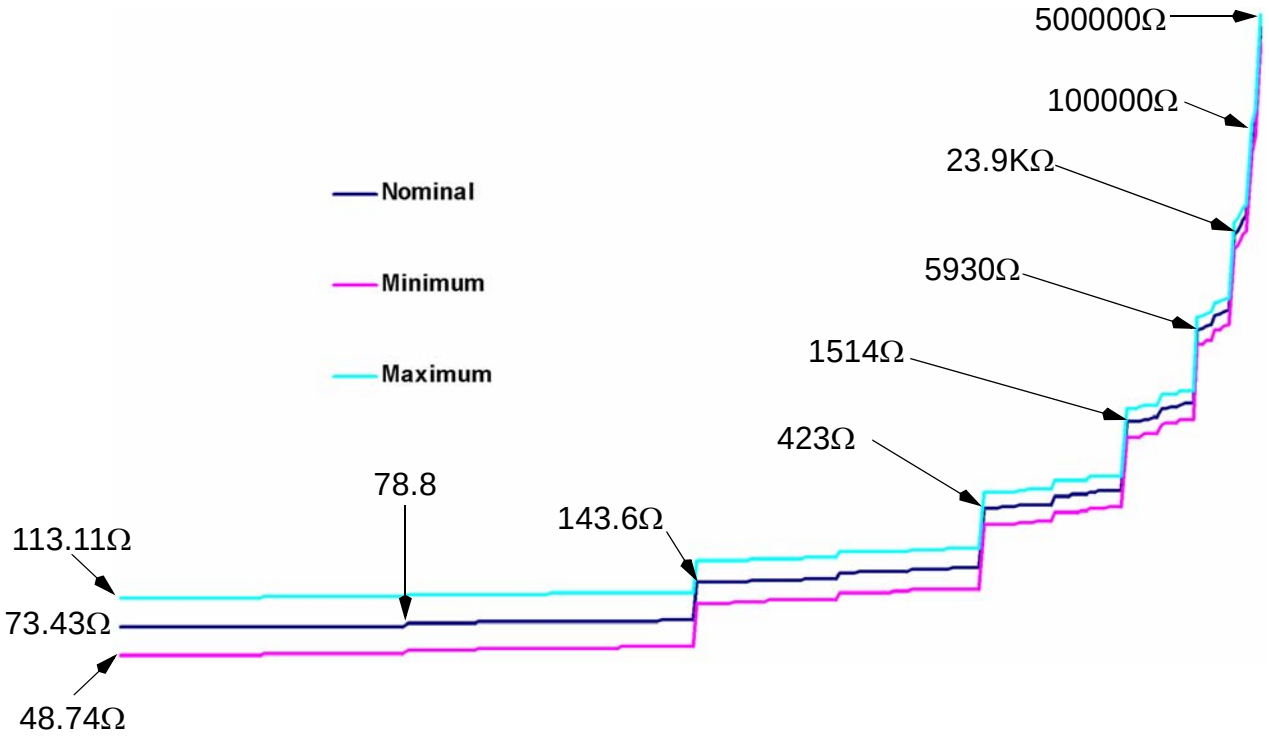


Figure-41: Parallel RL Values

When combining multiple RL resistors (in parallel only) the 50Ω (30-85 ohms) must be added to the resulting parallel resistance calculation. Also, the PTU voltage limits and maximum current the PTU can supply may constrain the use of some parallel RL combinations.

The `rl_bitmask_get()` function can be used to obtain a bit-mask suitable for use as the `RlSelect` argument to the `vz()` function. Using this function, the user specifies the desired RL value and `rl_bitmask_get()` returns both the bit-mask which would result in the closest available value and the resulting value.

Note the following:

- VZ is programmable per-pin.
- During initial program load VZ is set to 0V.
- During execution of the Sequence & Binning Table, user code determines the value VZ.

- During initial program load RL is set to 500K. The system software does not otherwise modify the RL selections.
- When a given pin's PE driver mode is set to Vz Mode (see Magnum PE Driver Modes) the last programmed RL value is selected.
- When Sequence & Binning Table execution stops, VZ is set to 0V via the `builtin_after_testing_block`.
- VZ can be programmed from an executing test pattern. See Controlling PE Levels from the Test Pattern.

Usage

The following function sets the VZ value for one pin. The value of RL is not modified:

```
void vz( double value, DutPin* pDutPin );
```

The following function sets the VZ value for one or more pin(s). The value of RL is not modified:

```
void vz( double Value, PinList*pPinList );
```

The following function sets VZ and RL value for one pin:

```
void vz( double value, int rl_val, DutPin *dutpin );
```

The following function sets VZ and RL value for one or more pin(s):

```
void vz( double value, int rl_val, PinList* pinlist );
```

The following function gets the currently programmed VZ value for one pin:

```
double vz( DutPin* pDutPin );
```

where:

`Value` specifies the desired VZ value. Units may be used (see Specifying Units). Legal values are shown below:

Table 3.15.6.0-2 VZ Level Range and Resolution

Parameter	Range	Resolution	
VZ	-1V to +7V	5mV	

`pDutPin` (**`dutpin`**) is used in two contexts:

- In the setter functions, identifies one pin to be programmed. In Multi-DUT Test Programs, the pins of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.
- In the getter functions, identifies one pin to be read.

`pPinList` (**`pinlist`**) specifies which pin(s) are to be programmed. In Multi-DUT Test Programs, only pins DUT(s) currently in the Active DUTs Set (ADS) are affected. The specified pin list must only contains pins mapped to a signal pins in the Pin Assignment Table.

`rl_val` is used to select the desired termination resistance, RL. **`rl_val`** is a bit-wise value specified using one or more *Select Token* values shown in the RL Values table or using `rl_bitmask_get()`. Combinations of resistance values are specified by OR'ing two or more values together (see Example). The values 0 and >255 are illegal.

The `vz()` getter function returns the currently programmed value for one pin. In Multi-DUT Test Programs, the value is retrieved from the first DUT in the Active DUTs Set (ADS).

Example

The following example sets VZ to 1.8V on all pins in the `data_bus` pin list. Only pins of DUT(s) in the Active DUTs Set (ADS) are affected. The value of RL is not modified:

```
vz( 1.8 V, data_bus );
```

The following example returns the currently programmed VZ value for the `D0` pin. The value is retrieved from the first DUT in the Active DUTs Set (ADS):

```
double v = vz( D0 );
```

The following example sets VZ to 1.8V on all pins in the `data_bus` pin list. The value of RL is set to 1532Ω, which is the parallel combination of 4 resistance values taken from the RL Values table. Only pins of DUT(s) in the Active DUTs Set (ADS) are affected:

```
vz( 1.8 V, (VZRS1 | VZRS3 | VZRS4 | VZRS5), data_bus );
```

Or...

```
DWORD mask;
double closest = rl_bitmask_get( 1532, &mask );
vz( 1.8 V, mask, data_bus );
```

3.15.7 rl_set(), rl_get()

See PTU, PE Driver, Magnum PE Driver Modes, PE Load Reference Voltage: VZ.

Description

The `rl_set()` function is used to select the termination resistance value, RL, for one or more pins which are in Vz Mode (see Magnum PE Driver Modes).

The `rl_get()` function is used to get the bit-mask used to set the currently programmed RL value for one pin. The current PE driver mode does not affect the value returned.

Note the following:

- Executing `rl_set()` has no immediate effect on pins which are not in Vz Mode. However, when a given pin's PE driver mode is set to Vz Mode (see Magnum PE Driver Modes) the last programmed RL value is selected.
- Details about RL values and usage options are covered in PE Load Reference Voltage: VZ.
- The `vz()` function can also be used to select the RL value for one or more pin(s).

Usage

The following function sets the RL value for one pin:

```
void rl_set( int value, DutPin * dutpin );
```

The following function sets the RL value for one or more pin(s):

```
void rl_set( int value, PinList* pinlist );
```

The following function gets the bit-mask used to set the currently programmed RL value for one pin:

```
int rl_get( DutPin * dutpin );
```

where:

`value` is a bit-wise value used to select the desired termination resistance, RL. The value is may be specified using one or more `Select Token` values from the RL Values table or using `rl_bitmask_get()`. Combinations of resistance values are specified by OR'ing two or more values together (see Example). The values 0 and >255 are illegal.

`dutpin` is used in two contexts:

- In the setter function, identifies one pin to be programmed. In Multi-DUT Test Programs, the pin(s) of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.
- In the getter functions, identifies one pin to be read.

`pinlist` specifies which pin(s) are to be programmed. In Multi-DUT Test Programs, the pin(s) of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **`pinlist`** must only contains pins mapped to signal pins in the Pin Assignment Table. Only pins in Vz Mode are immediately affected, see Description.

`rl_get()` returns the bit-mask used to set the currently programmed RL value for the specified pin. In Multi-DUT Test Programs, the value is retrieved from the first DUT in the Active DUTs Set (ADS).

Example

The following example sets the value of RL to 1532Ω for all pins in the `data_bus` pin list. 1532Ω is the parallel combination of 4 resistance values taken from the RL Values table. Only pins of DUT(s) in the Active DUTs Set (ADS) are affected:

```
rl_set( (VZRS1 | VZRS3 | VZRS4 | VZRS5), data_bus );
```

Or...

```
DWORD mask;
double closest = rl_bitmask_get( 1532, &mask );
rl_set( mask, data_bus );
```

The following example returns the bit-mask used to set the currently programmed RL value for the `D0` pin. The value is retrieved from the first DUT in the Active DUTs Set (ADS)

```
int rl = rl_get( D0 );
```

3.15.8 rl_bitmask_get()

See PTU, PE Driver, Magnum PE Driver Modes, PE Load Reference Voltage: VZ.

Description

The `rl_bitmask_get()` function may be used to obtain a bitmask suitable for use as the `RlSelect` argument to the `vz()` function and `rl_set()` function. This is the argument

which determines the load termination resistance, RL, for the pin(s) in Vz Mode (see Magnum PE Driver Modes).

Note the following:

- The 1st argument to `rl_bitmask_get()` specifies the desired RL resistance.
- `rl_bitmask_get()` then determines the closest obtainable value, considering the 256 possible RL values, see RL Values.
- The 2nd argument returns a bitmask representing this closest value. This argument is suitable for use as the `RlSelect` argument to the `vz()` function and `rl_set()` function.
- `rl_bitmask_get()` returns this (closest) value, allowing user code to compare or calculate the difference between the desired value and the value represented by the bitmask.

Usage

```
double rl_bitmask_get( double desired, DWORD * closest );
```

where:

`desired` specifies the desired RL value, in ohms.

`closest` is a pointer to an existing `DWORD` variable used to return a bitmask corresponding to the RL value closest to **`desired`**.

`rl_bitmask_get()` returns the RL value represented by bitmask, in ohms.

Example

The following example obtains a bitmask representing the obtainable RL value nearest to 1000 ohms. The difference between 1000 ohms and this RL value is calculated and if OK the bitmask is passed to `vz()` to enable this RL value on all pins in the `myPL` pin list.

```
DWORD bitmask;
double closest = rl_bitmask_get( 1000, &bitmask );
double difference = closest - 1000;
if( difference == OK ) // ... whatever OK is...
   vz( 1.85 V, bitmask, myPL );
```

3.15.9 rl_ohms_get()

See PTU, PE Driver, Magnum PE Driver Modes.

Note: first available in software release h2.2.8/h1.2.8.

Description

The `rl_ohms_get()` function returns the current RL value for a specified pin, in ohms, based on the pin's current RL hardware configuration. See RL Values, PE Load Reference Voltage: VZ, `rl_set()`.

Note the following:

- The value returned is a calculation, based on the currently programmed RL hardware configuration for the specified pin.
- As indicated elsewhere, there are 8 discrete RL values (±20%, see RL Values) which may be selected individually or may be combined (in parallel only) to obtain additional resistance values. An additional 50Ω (30Ω-85Ω) resistance is added to the combined RL value. This represents the resistance of the FET switch which connects the PTU to the DUT.

Usage

The following function returns the calculated RL value for one pin:

```
double rl_ohms_get( DutPin *pDutPin );
```

where:

`dutpin` identifies the pin to be read.

`rl_ohms_get()` returns the calculated RL value for **`dutpin`**. In Multi-DUT Test Programs, the value is retrieved from the first DUT in the Active DUTs Set (ADS).

Example:

```
double r = rl_ohms_get( myPin );
```

3.15.10 50-ohm Termination Voltage: VTT

See PTU, PE Driver, Magnum PE Driver Modes.

Description

The `vtt()` function is used to set or get the 50Ω PE Driver termination voltage, VTT.

Note the following:

- VTT is a per-pin parameter.
- During initial program load VTT is set to 0V.
- During execution of the Sequence & Binning Table, user code determines the value VTT.
- When Sequence & Binning Table execution stops, VTT is set to 0V via the `builtin_after_testing_block`.
- VTT can also be programmed from an executing test pattern. See Controlling PE Levels from the Test Pattern.

The `vtt()` getter function is used to retrieve the currently programmed VTT voltage for one specified pin.

Usage

The following function sets the VTT value all pins in the program:

```
void vtt( double value );
```

The following function sets the VTT value for one pin:

```
void vtt( double Value, DutPin *pDutPin );
```

The following function sets the VTT value for one or more pin(s):

```
void vtt( double Value, PinList* pPinList );
```

The following function returns the currently programmed VTT level for the specified pin:

```
double vtt( DutPin *pDutPin );
```

where:

Value specifies the desired VTT value. Units may be used (see Specifying Units). Legal values are shown below:

Table 3.15.10.0-1 VTT Range & Resolution

Range	Resolution
-1V to +7V	5mV

pDutPin is used in two contexts:

- In the setter function, identifies one pin to be programmed. In Multi-DUT Test Programs, the pins of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.
- In the getter function, identifies one pin to be read.

pPinList specifies which pin(s) are to be programmed. In Multi-DUT Test Programs, the pins of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **pPinList** must only contains pins mapped to a signal pins in the Pin Assignment Table.

The `vtt()` getter function returns currently programmed value. In Multi-DUT Test Programs, the value is retrieved from the first DUT in the Active DUTs Set (ADS).

Example

```
vtt( 1.5 V, pl_IOpins );
double v = vtt( D0 );
```

3.15.11 pe_driver_mode_set(), pe_driver_mode_get()

See PTU, PE Driver, Magnum PE Driver Modes.

Description

The `pe_driver_mode_set()` function is used to set the PE driver mode for one or more pin(s). See PE Driver and Magnum PE Driver Modes.

The `pe_driver_mode_get()` function may be used to get the current PE driver mode for one pin.

Note: the `pe_driver_mode_set()` function will not affect pins which are not currently connected. This is an issue when `pe_driver_mode_set()` is used in the Site Begin Block. The system software automatically connects all PE pins in the Site Begin Block, but this does not occur until after all user-code has been executed. For this reason, it is recommended that `pe_driver_mode_set()` be executed in a Before-testing Block or Test Block.

The PE Driver has four modes of operation:

- Vihh Mode
- Vz Mode
- Vtt Mode
- Dclk Mode

Details are described in Magnum PE Driver Modes.

During initial program load, all pins are set to Vz Mode. The system software does not otherwise modify any PE driver modes.

Each pin of a given pin-pair (see Functional Pin-pairs) can be in a different PE driver mode.

Executing `pe_driver_mode_get()` in simulation mode always returns `t_pe_nomode`, regardless of the mode set using `pe_driver_mode_set()`.

Usage

The following function sets the PE driver mode for one pin:

```
void pe_driver_mode_set( PEDriverMode mode, DutPin * dutpin );
```

The following function sets the PE driver mode for one or more pin(s) :

```
void pe_driver_mode_set( PEDriverMode mode, PinList* pinlist );
```

The following function returns the current PE driver mode for one pin:

```
PEDriverMode pe_driver_mode_get( DutPin * dutpin );
```

where:

`mode` specifies the desired PE driver mode. Legal values are of the `PEDriverMode` enumerated type. `t_pe_nomode` should not be used.

`dutpin` is used in two contexts:

- In the setter function, identifies one pin to be programmed. In Multi-DUT Test Programs, the pins of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.
- In the getter function, identifies one pin to be read.

`pinlist` specifies which pin(s) are to be programmed. In Multi-DUT Test Programs, the pins of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **`pPinList`** must only contains pins mapped to a signal pins in the Pin Assignment Table

`pe_driver_mode_get()` returns the currently set PE driver mode for one pin. In Multi-DUT Test Programs, the value is retrieved from the first DUT in the Active DUTs Set (ADS).

Example

```
pe_driver_mode_set( t_pe_vzmode, pl_IOpins );
PEDriverMode mode = pe_driver_mode_get( D0 );
```

3.15.12 PE Connect/Disconnect Functions

See Pin Electronics (PE), Pin Electronics Voltages/Currents.

Description

Note: these functions operate quite differently on Maverick-I/-II vs. Magnum 1/2/2x: more below.

Note: during initial program load, the system software automatically connects all used digital pins (signal pins) at the end of the Site Begin Block execution (after all user-code, if any, has executed). At the time this occurs all pin voltages have previously been set = 0V.

Note: during Sequence & Binning Table execution, the system software automatically connects all used digital pins (signal pins) during the execution of the `builtin_before_testing_block`. This occurs before any user-written code executes in a Before-testing Block (if any) or Test Block.

The `disconnect()` function is used to *effectively* disconnect Pin Electronics (PE) circuitry from the DUT pin. The word *effectively* is used because there is no physical relay or switch between the PE driver/comparator and the DUT. Instead, executing `disconnect()` does the following:

- The PE Driver is tri-stated.
- The PTU is disconnected. This disables the Vz/RL load or VIHH depending on the current driver mode (see Magnum PE Driver Modes).

Note: the previous description also applies when the system software performs the connect/disconnect operations as described below.

The `pin_connect()` function is used *effectivly* re-connect the Pin Electronics (PE) circuitry to the DUT pin. This presumes the pin was previously *disconnected* using `disconnect()`. See above.

Note: when executing Static DC Tests and Dynamic DC Tests all PE channel disconnect/connect operations are automatically controlled by the system software; i.e. it is NOT necessary for user-code to use the functions documented here except in very special situations.

PE connections are controlled as follows:

- The initial program load opens all PE connections to the DUT. See Note:.
- When the Sequence & Binning Table is executed the state of the PE Driver and PTU are not changed by the system software; i.e. they are only effectively *connected* when a test is executed which controls the underlying hardware. See Note:.
- User-code can connect and/or disconnect PE channels using the functions documented here (see Note:). Any connection changes made by user-code remain in effect until execution of the Sequence & Binning Table stops (next).
- When Sequence & Binning Table execution ends and after execution of any user-defined After-testing Block(s) the system software executes the `builtin_after_testing_block`, which disconnects all used PE channels from the DUT. See Note:.
- DPS, HV and PMU connections are NOT affected using the functions documented here.

Usage

The following function disconnects one specified PE channel:

```
void disconnect( DutPin *pDutPin );
```

The following function disconnects the specified PE channels:

```
void disconnect( PinList* pPinList );
```

The following function connects one specified PE channel:

```
void pin_connect( DutPin *pDutPin );
```

The following function connects the specified PE channels:

```
void pin_connect( PinList* pPinList );
```

The following function connects all PE channels used in the test program:

```
void pin_connect();
```

where:

`dutpin` is used in two contexts:

- Using `pin_connect()`, identifies one pin to be connected. In Multi-DUT Test Programs, the pins of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.
- Using `disconnect()`, identifies the pin to be disconnected. In Multi-DUT Test Programs, the pins of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.

`pPinList` identifies which pin(s) are to be connected or disconnected. In Multi-DUT Test Programs, the pins of each DUT currently in the Active DUTs Set (ADS) are affected. The specified **`pPinList`** must only contains pins mapped to signal pins in the Pin Assignment Table.

Example

The following example disconnects the PE channels identified by the pin list named `PL_address_high`. Other code executes as desired. Then, all used PE channels are connected.

```
disconnect( PL_address_high );
// Other test program code goes here as desired
pin_connect();
```

3.16 Pin Scramble Functions & Macros

See Pin Scramble MUX, Software.

This section contains the following:

3.16.1 Overview

See Pin Scramble Functions & Macros, Pin Scramble MUX.

In software, the Pin Scramble Macros documented in this section are used to define one or more named Pin Scramble Table(s), each of which defines up to 64 Pin Scramble Map(s) (PS1 to PS64). Each map identifies which test pattern data source is mapped to each timing channel via the Pin Scramble MUX.

The Pin Scramble MUX allows the test pattern to control/change which pattern data source is mapped to the Timing & Formatting logic of each timing channel in each pattern cycle. This capability can greatly simplify test program and test pattern creation. For example, when testing a serial memory, the APG will use one instruction to update the X/Y addresses, data, and read/write state, and a series of following instructions (typically a pattern subroutine) to sequence the individual bits of these addresses/data to one pin of the DUT. It is the Pin Scramble MUX and Pin Scramble RAM which make this possible in hardware. It is the Pin Scramble Macros which are used to define how this hardware is initialized. It is the test pattern which controls the Pin Scramble MUX, cycle-by-cycle, for each timing channel.

As the test program is initially loaded, one Pin Scramble Table is selected and loaded. This initializes the Pin Scramble RAM with 64 Pin Scramble Maps. All 64 maps are initially set to the Default Pin Scramble Map. User code modifies these as necessary using the Pin Scramble Macros.

During test pattern execution, in each pattern cycle, the APG outputs a pin scramble selection (one of PS1 through PS64), which selects one map in the Pin Scramble RAM. The output of the Pin Scramble RAM directs the Pin Scramble MUX to select the source of

pattern data, strobe control, and I/O control, independently for each of the 64 timing channels, each of which drives 2 tester pins (see Functional Pin-pairs).

In the test pattern, the default pin scramble map selection is PS1. To explicitly select a pin scramble map the PINFUNC PS# instruction is used (Memory Test Patterns) or the VEC/RPT PS# and VPINFUNC PS# instruction is used (Logic Test Patterns).

Multiple Pin Scramble Tables can be defined in the test program, however only one can be loaded (see Single Resource Types). If only one Pin Scramble Table is defined the system software will automatically select it during test program initialization. When multiple tables are defined two options are available:

- User code explicitly selects one Pin Scramble Table using the USE_PIN_SCRAMBLE() macro. This is one of the Configuration Macros, and must be used within a CONFIGURATION() or SITE_CONFIGURATION() block.
- When user code does not select a Pin Scramble Table the system software automatically presents a selection dialog. Note, however, that this dialog is presented for each Site, which is not very user friendly.

Given the name of a Pin Scramble Map, the PinScramble_find() function can be used to get a pointer to a Pin Scramble Map.

Note: the error signals to the Error Catch RAM (ECR) are *NOT* affected (descrambled) by the pin scramble MUX. There is only one, fixed, error line from each PE channel to the Error Catch RAM.

3.16.2 Pin Scramble Macros

See Pin Scramble Functions & Macros, Pin Scramble MUX.

Description

The PIN_SCRAMBLE() macro is used to create a new Pin Scramble Table.

The SCRAMBLE_MAP()macro is used, within the body code of the PIN_SCRAMBLE() macro, to select which which of the 64 Pin Scramble Maps is being defined.

The USE_PIN_SCRAMBLE() macro is used to select one Pin Scramble Table, by name, when multiple tables have been defined. This is one of the Configuration Macros, and must be used within a CONFIGURATION() block:

The INCLUDE_PIN_SCRAMBLE( ) macro is used to insert an existing [partial] Pin Scramble Table into a new table being defined. This is an optional method, allowing reuse of a [partial] Pin Scramble Table defined separately.

The EXTERN_PIN_SCRAMBLE( ) macro is used to make a forward or external declaration.

The remaining macros described in Usage below are used to actually specify the mapping of pattern data source to tester pins.

Within the body of the SCRAMBLE_MAP macro, the SCRAMBLE or SCRAMBLE2 macro is used, once for each DUT pin, to specify the source of pattern data, in that map. The SCRAMBLE2 macro is used in Double Data Rate (DDR) Mode applications (see DDR Pin Scramble).

The TesterFunc enunerated type defines the legal values used to specify the data source arguments to all versions of the SCRAMBLE macros. All of the APG data sources (t_cs1 , t_x0, t_y1, etc.) are always valid because the APG is not an option. The t_scan and t_lvm data sources are only valid if the tester has the optional Logic Vector Memory (LVM)/ Scan Vector Memory (SVM) option installed.

Note: the TesterFunc enumerated type includes values for both Maverick-I/-II and Magnum 1/2/2x applications. Using Maverick-I/-II, only t_scan1..t_scan4 are valid. Using Magnum 1/2/2x, only t_scan is valid. Do not use t_scan1..t_scan4 in Magnum 1/2/2x programs. Do not use t_scan in Maverick-I/-II programs.

Using Magnum 1/2/2x Scan Test Patterns, pin(s) which appear in a SCANDEF Compiler Directive will be scrambled to t_scan even though the user's pin scramble definition explicitly maps the pin(s) to t_lvm. Conversely, if these same pin(s) are scrambled to APG resources (t_x0, t_y1, etc.) they operate as expected (although this is not good practice). In general, pins which are to be controlled by the scan pattern should be explicitly scrambled to t_scan; to ensure the test pattern and the pin scramble map agree and don't confuse the less knowledgeable user.

Using Magnum 1/2/2x, a number of pattern independent options are available:

- t_drive_low
- t_drive_high
- t_strobe_low
- t_strobe_high
- t_strobe_valid
- t_strobe_mid

- t_tri_state

These operate much like any APG/LVM/Scan pattern source except that they are fixed i.e. independent of APG/LVM/Scan data. These options only determine what operation will be performed on the pin(s), the programmed edge timing operates normally.

Using Magnum 1/2/2x, if the test program is a non-Multi-DUT Test Program, only the SCRAMBLE or SCRAMBLE2 MACROs can be used. In Magnum 1/2/2x Multi-DUT Test Programs, only the SCRAMBLEn_xxx MACROs can be used, as described below.

Within the body of the SCRAMBLE_MAP macro, one of the SCRAMBLEn_xDUT macros is used, once for each DutPin, to specify the source of pattern data for that DutPin, for each DUT, in that map. Which macro is used (SCRAMBLE_1DUT vs. SCRAMBLE_2DUT, and SCRAMBLE2_1DUT vs. SCRAMBLE2_2DUT, etc.) is determined by two considerations:

- How many DUT(s) the test program is designed to test (xxx_1DUT, xxx_2DUT, etc.). See Pin Assignment Table. This is required because it is possible to assign a different pattern data source to a given DutPin for each DUT, in a given map, thus the number of parameters passed to the macro is dependent on how many DUT(s) are being tested. Note that the set of macros does not support testing an odd number of DUTs (except for testing one DUT).
- Whether a single data rate or Double Data Rate (DDR) Mode Pin Scramble Map is being defined (SCRAMBLE_xxx vs. SCRAMBLE2_xxx, etc.).

For example, when defining a single data rate Pin Scramble Map used to test 4 DUTs, the SCRAMBLE_**4**DUT macro is used:

```
                // DutPin  t_dut1  t_dut2  t_dut3  t_dut4
SCRAMBLE_4DUT(    D0,    t_d0,    t_d0,    t_d0,    t_d0 )
SCRAMBLE_4DUT(    A9,    t_a9,    t_a9,    t_a9,    t_a9 )
// Etc.
```

Note the following:

- This macro takes 1 DutPin argument (as do all of the SCRAMBLEn_xDUT MACROs), and 4 pattern data source arguments, listed in the order of t_dut1, t_dut2, t_dut3, and t_dut4. The name of the MACRO determines the number of DUTs being tested thus the number of arguments.

In Multi-DUT Test Programs, the TesterFunc value specified for DUTs which share Functional Pin-pairs must be identical. For example:

```
//            DutPin t_dut1  t_dut2  t_dut3  t_dut4
SCRAMBLE_4DUT( Cs,   t_cs1,  t_cs1,  t_cs1,  t_cs1 )
... or ...
SCRAMBLE_4DUT( Cs,   t_cs3,  t_cs3,  t_cs2,  t_cs2 )
```

Note: prior to software release h1.1.23 this rule was not enforced by the system software and no warnings were issued. Beginning in software release h1.1.23 it is a fatal error to violate this rule.

Again, for any DUTs which share Functional Pin-pairs, the TesterFunc values must be identical for all pins of the pair. See next item.

It may sometimes be necessary, in a Multi-DUT Test Program, that a specific device type which, for the most part, can be tested using Functional Pin-pairs, requires independent timing and/or pattern stimulus for a few pin(s).

To do this requires changes to the Pin Assignment Table because two DutPins must be defined for each of these pins, and two ASSIGN_xxxDUT statements must be used to specify connections (note the use of a_na, b_na, etc. in the examples below). Then, in the Pin Scramble Table, two statements are used to assign the pattern data source to each DutPin. For example:

```
DUT_PIN( Cs_a)
DUT_PIN( Cs_b )
//....
//            DutPin t_dut1   t_dut2   t_dut3    t_dut4
ASSIGN_4DUT( Cs_a,    a_11,   b_na,    a_23,     b_na  )
ASSIGN_4DUT( Cs_b,    a_na,    a_3,    a_na,     a_32  )
//....
SCRAMBLE_4DUT( Cs_a,  t_cs1,  t_cs2,   t_cs3,     t_cs4 )
SCRAMBLE_4DUT( Cs_b,  t_cs1,  t_cs2,   t_cs3,     t_cs4 )
```

Pin Scramble Map code

Pin Assignment Table code

In this example, the DUT actually has only one Cs pin but because all 4 DUTs need independent timing and/or test pattern control for this pin two DutPins are defined in the Pin Assignment Table, with tester channels only assigned to 1 DUT in each statement. Then, when defining the Pin Sramble Map, the 4 DutPins will be specified separately and the

values specified for the *other* DUTs are not actually used (on pins which had . a_na, b_na, etc. in the Pin Assignment Table). Above, in each SCRAMBLE_4DUT statement, only the pins circled in magenta actually use the values assigned.

Other rules:

- In each Pin Scramble Table, any Pin Scramble Map(s) which are not explicitly initialized in the user's test program are set to the Default Pin Scramble Map.
- When user-code partially defines a given Pin Scramble Map, DutPins which are not explicitly defined are implicitly set = t_y15 (Maverick-I/-II) or t_tri_state (Magnum 1/2/2x).

Usage

The following macro is used to create a new Pin Scramble Table, with the specified **name**:

```
PIN_SCRAMBLE( name )
```

The following macro is used to select one Pin Scramble Table (by **name**) to be used when multiple tables have been defined. This is one of the Configuration Macros, and must be used within a CONFIGURATION() block:

```
USE_PIN_SCRAMBLE( name )
```

The following macro is used to insert an existing [partial] Pin Scramble Table, by **name**, into the table being defined. This is an optional method, allowing reuse of a [partial] Pin Scramble Table defined separately.

```
INCLUDE_PIN_SCRAMBLE( name )
```

The following macro is used to declare as external a specified Pin Scramble Table, by **name**:

```
EXTERN_PIN_SCRAMBLE( name )
```

The following macro is used to identify which of the 64 Pin Scramble Maps is being defined. This macro must be used within the body of the PIN_SCRAMBLE() macro. Legal values for **map_num** are of the PSNumber enumerated type (PS1 through PS64). Any Pin Scramble Map(s) which are not explicitly defined are set to the Default Pin Scramble Map:

```
SCRAMBLE_MAP( map_num )
```

The following macro is used to insert an existing [partial] Pin Scramble Map into the map being defined. Legal values for **map_num** are of the PSNumber enumerated type (PS1, PS2, etc.). This is an optional method, allowing reuse of a [partial] Pin Scramble Map defined elsewhere:

```
INCLUDE_SCRAMBLE_MAP( map_num )
```

The following macro is used to assign a pattern data source to one DutPin in a single data rate Pin Scramble Map(s). Using Magnum 1/2/2x, this assigns the same pattern source to the specified DutPin all for all DUTs defined in the Pin Assignment Table:

```
SCRAMBLE( pin, TesterFunc data_source )
```

The following macro is used to assign two pattern data sources to one DutPin in a Double Data Rate (DDR) Mode Pin Scramble Map. Using Magnum 1/2/2x, this assigns the same pattern sources to the specified DutPin all for all DUTs defined in the Pin Assignment Table. See DDR Pin Scramble:

```
SCRAMBLE2(pin,
          TesterFunc data_source_A_cycle,
          TesterFunc data_source_B_cycle )
```

The following macros are used in Magnum 1/2/2x Multi-DUT Test Programs to assign a pattern data source to one DutPin for each DUT in a single data rate Pin Scramble Map. The name of the specific macro used must match the number of DUT(s) the program is testing, as defined in the Pin Assignment Table. Additional rules apply, see Description:

```
SCRAMBLE_1DUT( pin, f1 )
SCRAMBLE_2DUT( pin, f1, f2 )
SCRAMBLE_4DUT( pin, f1, f2, f3, f4 )
SCRAMBLE_6DUT( pin, f1, f2, f3, f4, f5, f6 )
... snip ...
SCRAMBLE_32DUT(   pin,
```

Do not use SCRAMBLE_32DUT, see Note: and SCRAMBLE_32DUT Work-around.

```
8,
f15, f16,
 f23, f24,
f25, f26, f27, f28, f29, f30, f31, f32 )
```

Note: after Double Data Rate (DDR) Mode support was added, which modified the SCRAMBLE_32DUT and SCRAMBLE2_32DUT macros, it was discovered that Developer Studio limits the number of arguments to a given macro. Violating this rule is fatal, making these two macros unusable. See SCRAMBLE_32DUT Work-around.

The following macros are used in Magnum 1/2/2x Multi-DUT Test Programs to assign two pattern data sources to one DutPin for each DUT in a Double Data Rate (DDR) Mode Pin Scramble Map. Additional rules apply, see Description:

```
SCRAMBLE2_1DUT( pin, f1a, f1b )
SCRAMBLE2_2DUT( pin, f1a, f1b, f2a, f2b )
SCRAMBLE2_4DUT(pin,
               f1a, f1b, f2a, f2b, f3a, f3b, f4a, f4b )
SCRAMBLE2_6DUT(pin,
               f1a, f1b, f2a, f2b, f3a, f3b, f4a, f4b,
               f5a, f5b, f6a, f6b )
... snip ...
SCRAMBLE2_32DUT(  pin,
                  f1a, f1b, f2a, f2b, f3a, f3b, f4a, f4b,
                                                 f8a, f8b,
                                              11b, f12a, f12b,
                                                 f15b, f16a, f16b,
                                                 f19b, f20a, f20b,
                                                 f23b, f24a, f24b,
                                                 f27b, f28a, f28b,
                  f29a, f29b, f30a, f30b, f31a, f31b, f32a, f32b )
```

Do not use SCRAMBLE2_32DUT, see Note: and SCRAMBLE_32DUT Work-around.

where:

pin identifies one DUT DutPin for which the Pin Scramble is being defined.

data_source specifies the desired pattern data source. Legal values are of the TesterFunc enunerated type, however some values are only valid if the tester being used has the corresponding hardware option installed. See Description and Note:..

data_source_A_cycle and **data_source_B_cycle** specify two data sources for the specified **pin**. This option applies to Double Data Rate (DDR) Mode applications. Legal values are of the TesterFunc enunerated type, however some values are only valid if the tester being used has the corresponding hardware option installed. See Description.

f1, **f2**, etc. specify a pattern data source for the specified **pin** for each DUT in a Multi-DUT Test Program. **f1** = data source for DUT1, **f7** = data source for DUT7, etc. Legal values are of the TesterFunc enunerated type, however some values are only valid if the tester being used has the corresponding hardware option installed. See Description. Additional rules apply, see Description.

f1a, **f1b**, etc. represent two data sources to be selected for the DUT corresponding to the parameter number. **f1a** = A-cycle data source for DUT1, **f7b** = B-cyle data source for DUT7, etc. This form is used in Double Data Rate (DDR) Mode applications. Legal values are of the TesterFunc enunerated type, however some values are only valid if the tester

being used has the corresponding hardware option installed. See Description. Additional rules apply, see Description.

Example

Example 1:

The following example shows a Pin Scramble Map for a small memory device: Using Magnum 1/2/2x, this is only valid in a non-Multi-DUT Test Program:

```
PIN_SCRAMBLE( soic_20_pin ) {
   SCRAMBLE_MAP( PS1 ) {
      SCRAMBLE( A0, t_x0 )
      SCRAMBLE( A1, t_x1 )
      SCRAMBLE( A2, t_x2 )
      SCRAMBLE( A3, t_y0 )
      SCRAMBLE( A4, t_y1 )
      SCRAMBLE( D0, t_d0 )
      SCRAMBLE( D1, t_d1 )
      SCRAMBLE( D2, t_d2 )
      SCRAMBLE( D3, t_d3 )
      SCRAMBLE( WE, t_cs1 )
      SCRAMBLE( OE, t_cs2 )
      SCRAMBLE( CS, t_cs3 )
   }
}
```

In this example, there is only one user-defined Pin Scramble Map (PS1) in this Pin Scramble Table (soic_20_pin).This has the following significance:

- The other 63 Pin Scramble Maps are initialized to the Default Pin Scramble Map.
- Any test pattern instructions which do not include an explicit Pin Scramble Map selections will use the default = PS1.
- This example DUT has six address pins, four data pins, and three chip selects. The user names previously assigned to these pins in the Pin Assignment Table are A0 through A5, D0 through D3, and WE, OE, and CS.
- All of the pattern data sources are from the APG, thus no optional hardware is used.

Example 2:

The following example configures two different Pin Scramble Maps (PS1, PS2) to multiplex address and data on the same set of DUT pins of a fictitious memory device. Using Magnum 1/2/2x, this is only valid in a non-Multi-DUT Test Program:

```
PIN_SCRAMBLE( sort_scramble ) {
   SCRAMBLE_MAP( PS1 ) { // Map for address input cycle
      SCRAMBLE( AD0, t_x0 )
      SCRAMBLE( AD1, t_x1 )
      SCRAMBLE( AD2, t_x2 )
      SCRAMBLE( AD3, t_x3 )
      SCRAMBLE( AD4, t_x4 )
      SCRAMBLE( AD5, t_y0 )
      SCRAMBLE( AD6, t_y1 )
      SCRAMBLE( AD7, t_y2 )
      SCRAMBLE( CLK, t_cs1 )
      SCRAMBLE( Control, t_cs2 )
      SCRAMBLE( CS, t_cs3 )
   }
   SCRAMBLE_MAP( PS2 ) {// Map for data read or write cycle
      SCRAMBLE( AD0, t_d0 )
      SCRAMBLE( AD1, t_d1 )
      SCRAMBLE( AD2, t_d2 )
      SCRAMBLE( AD3, t_d3 )
      SCRAMBLE( AD4, t_d4 )
      SCRAMBLE( AD5, t_d5 )
      SCRAMBLE( AD6, t_d6 )
      SCRAMBLE( AD7, t_d7 )
      SCRAMBLE( CLK, t_cs1 )
      SCRAMBLE( Control, t_cs2 )
      SCRAMBLE( CS, t_cs3 )
   }
}
```

Note the following:

- The user-defined names for the multiplexed address/data pins are AD0 through AD7.
- The first Pin Scramble Map (PS1) is used to send an address to the DUT. In the test pattern, when PS1 is selected, the X/Y APG Address Generator outputs indicated are the pattern data source for the AD0 through AD7 pins.

- The second Pin Scramble Map (PS2) is used to read (strobe) data from the DUT or to write data to the DUT. When PS2 is selected, X/Y APG Data Generator outputs indicated are the pattern data source for the AD0 through AD7 pins. A data read cycle versus a data write cycle is determined by the chip select, I/O, and strobe states as specified in the test pattern.
- The pattern data source for the three chip selects are the same in both Pin Scramble Maps, indicating that their DUT functions are always the same.

Example 3:

The following example demonstrates the definition of one Pin Scramble Table, named PS_table_1, which includes two local Pin Scramble Map definitions (PS1 and PS64), and includes the contents of a separately defined Pin Scramble Table. Because the SCRAMBLE_1DUT macro is used, this program is a Multi-DUT Test Program, even though only 1 DUT is being tested:

```
PIN_SCRAMBLE( PS_table_1 ) {
   SCRAMBLE_MAP( PS1 ) {
      SCRAMBLE_1DUT( D0, t_d0 )
      SCRAMBLE_1DUT( A9, t_x1 )
      ...
      SCRAMBLE_1DUT( WriteEnable, t_cs1 )
   }
   // ... other maps as desired ...
   INCLUDE_PIN_SCRAMBLE( other_PS_table_name ) // Optional
   // ... other maps as desired ...
   SCRAMBLE_MAP( PS64 ) {
      SCRAMBLE_1DUT( D0, t_d0 )
      SCRAMBLE_1DUT( A9, t_x1 )
      ...
      SCRAMBLE_1DUT( WriteEnable, t_cs1 )
   }
}
```

3.16.2.1 SCRAMBLE_32DUT Work-around

See Pin Scramble Macros, Pin Scramble Functions & Macros.

After the DDR versions of SCRAMBLE_32DUT and SCRAMBLE2_32DUT were implemented it was discovered that Developer Studio limits the number of arguments to a given macro and that violating this rule is fatal i.e. these macros cannot be used.

The work-around to this limitation expands the underlying code represented by the macro. Rather than use the most complex version for all applications several scenarios are presented below. In each example, the macro being replaced is shown commented-out:

Example 1:

The following example assigns the same data source to pinX for all 32 DUTs. This actually works correctly for any number of DUTs, to scramble the same data source to a given pin of all DUTs in the program; i.e. the work-around is not needed:

```
PIN_SCRAMBLE( PS_table_1 ) {
  SCRAMBLE_MAP( PS10 ) {
   // SCRAMBLE_32DUT( pinX, t_d0, t_d0, ... snip..., t_d0 )
   // SCRAMBLE_32DUT( pinY, t_cs1, t_cs1, ... snip..., t_cs1 )
   // Replace the previous with...
   SCRAMBLE( pinX, t_d0 );
   SCRAMBLE( pinY, t_cs1 );
  }
}
```

Example 2:

The following example maps a different APG data bit to the data_pin of each Functional Pin-pair of 32 DUTs:

```
// SCRAMBLE_32DUT( data_pin,t_d0,t_d0,t_d1,t_d1, ...snip...,t_d15 )

static void Fill( TesterFuncArray &ary, TesterFunc *func, int cnt){
   for ( int i = 0; i < cnt; ++i )
      ary.Add( func[ i ] );
}

#define NUM_DUTS 32
PIN_SCRAMBLE( PS_table_1 ) {
  SCRAMBLE_MAP( PS10 ) {
   TesterFunc funcs[ NUM_DUTS ] = {
      t_d0, t_d0, t_d1, t_d1, t_d2, t_d2, t_d3, t_d3,
      t_d4, t_d4, t_d5, t_d5, t_d6, t_d6, t_d7, t_d7,
      t_d8, t_d8, t_d9, t_d9, t_d10, t_d10, t_d11, t_d11,
      t_d12, t_d12, t_d13, t_d13, t_d14, t_d14, t_d15, t_d15
   };
```

```
    TesterFuncArray array;
    Fill( array, funcs, sizeof funcs / sizeof *funcs );
    EXTERN_DUT_PIN( data_pin );
    _scramble( obj, data_pin, array, array );
    }
   }
}
```

Example 3:

The following DDR example assigns a different data source for DDR-A cycles vs. DDR-B cycles to the `DutPin` named `data_pin`. The same 2 data sources (`t_d0` ,`t_d32`) are mapped identically to the `data_pin` of all DUTs (which guarantees that the same data source is mapped to both pins of each Functional Pin-pair in `data_pin`). Note that the `obj` argument to `_scramble()` is automatically defined and initialized by the `SCRAMBLE_MAP` macro (in this example it represents `PS10`):

```
#define NUM_DUTS 32
PIN_SCRAMBLE( PS_table_1 ) {
  SCRAMBLE_MAP( PS10 ) {
    TesterFuncArray ps_A, ps_B;
    // The following code is duplicated/modified for each DUT pin
    // using this method
    for ( int i = 0; i < NUM_DUTS; ++i ){
      ps_A.Add( t_d0 );
      ps_B.Add( t_d32 );
    }
    EXTERN_DUT_PIN( data_pin )
    _scramble( obj, data_pin, ps_A, ps_B );
    ps_A.RemoveAll(); // To re-use ps_A for next pin
    ps_B.RemoveAll(); // To re-use ps_B for next pin
   }
}
```

3.16.3 Default Pin Scramble Map

See Pin Scramble Macros.

During the initial test program load, all 64 locations (all 64 Pin Scramble Maps) in the Pin Scramble RAM are set to the default Pin Scramble Map shown below. This means that any

Pin Scramble Map which is not subsequenly re-configured by user code (using Pin Scramble Macros) will retain these default Pin Scramble Map values.

Note that when user-code partially defines a given Pin Scramble Map that pins which are not explicitly defined are implicitly set = `t_y15` (Maverick-I/-II) or `t_tri_state` (Magnum 1/2/2x).

In the default Pin Scramble Map, only APG resources are selected; e.g. to use Logic Test Patterns or Scan Test Patterns requires that one or more user-defined Pin Scramble Map(s) be defined.

Table 3.16.3.0-1 Default Pin Scramble Pin Selections

Pin#	Function	Pin#	Function	Pin#	Function	Pin#	Function
a_1 b_1	t_x0	a_17 b_17	t_y0	a_33 b_33	t_d0	a_49 b_49	t_d16
a_2 b_2	t_x1	a_18 b_18	t_y1	a_34 b_34	t_d1	a_50 b_50	t_d17
a_3 b_3	t_x2	a_19 b_19	t_y2	a_35 b_35	t_d2	a_51 b_51	t_d18
a_4 b_4	t_x3	a_20 b_20	t_y3	a_36 b_36	t_d3	a_52 b_52	t_d19
a_5 b_5	t_x4	a_21 b_21	t_y4	a_37 b_37	t_d4	a_53 b_53	t_d20
a_6 b_6	t_x5	a_22 b_22	t_y5	a_38 b_38	t_d5	a_54 b_54	t_d21
a_7 b_7	t_x6	a_23 b_23	t_y6	a_39 b_39	t_d6	a_55 b_55	t_d22
a_8 b_8	t_x7	a_24 b_24	t_y7	a_40 b_40	t_d7	a_56 b_56	t_d23
a_9 b_9	t_x8	a_25 b_25	t_y8	a_41 b_41	t_d8	a_57 b_57	t_d24
a_10 b_10	t_x9	a_26 b_26	t_y9	a_42 b_42	t_d9	a_58 b_58	t_d25
a_11 b_11	t_x10	a_27 b_27	t_y10	a_43 b_43	t_d10	a_59 b_59	t_d26

Table 3.16.3.0-1 Default Pin Scramble Pin Selections *(Continued)*

Pin#	Function	Pin#	Function	Pin#	Function	Pin#	Function
a_12 b_12	t_x11	a_28 b_28	t_cs3	a_44 b_44	t_d11	a_60 b_60	t_d27
a_13 b_13	t_x12	a_29 b_29	t_cs4	a_45 b_45	t_d12	a_61 b_61	t_d28
a_14 b_14	t_x13	a_30 b_30	t_cs5	a_46 b_46	t_d13	a_62 b_62	t_d29
a_15 b_15	t_cs1	a_31 b_31	t_cs6	a_47 b_47	t_d14	a_63 b_63	t_d30
a_16 b_16	t_cs2	a_32 b_32	t_cs7	a_48 b_48	t_d15	a_64 b_64	t_d31

As indicated, using Magnum 1/2, the corresponding pin of each pin-pair (see Functional Pin-pairs) shares the same pattern data source.

3.17 VIHH Maps

See PE Driver, Vihh ModePin Electronics Voltages/Currents, VIHH Voltage.

Overview

During functional test execution, the PE Driver circuit has the following drive states:

- Drive-0 = VIL voltage, set using the `vil()` function
- Drive-1 = VIH voltage, set using the `vih()` function
- Drive-VIHH = VIHH voltage, set using the `vihh()` function
- Drive-VZ = VZ voltage, set using the `vz()` function
- Drive-VTT = VTT voltage, set using the `vtt()` function.
- Tri-state (un-terminated)

Using Magnum 1, only one of VIHH or VZ or VTT is usable at any given time, see Magnum PE Driver Modes. The information below applies only to pin(s) which are in Vihh Mode at the time a test pattern is executed.

In this section the term *VIHH* refers to the voltage level and *Vihh* to the logic signal used to select this level during test pattern execution (more below).

VIHH is typically a high voltage level (higher than VIH) although this is not required. VIHH is typically used to force the DUT into a special test mode or as special programming stimulus for some memory devices.

To use VIHH requires the following:

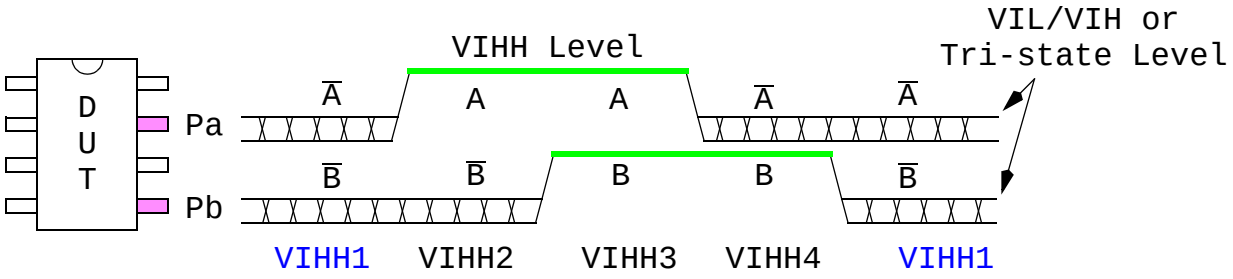
- Program the desired VIHH level(s), using the `vihh()` function. Since VIHH is supplied by a pin's Per-pin Parametric Test Unit (PTU) this also sets the operating range and output termination, see `vihh()`.
- Define one or more VIHH map(s). Each map identifies which pin(s) will react to the Vihh enable signal from the test pattern when that map is selected. VIHH maps are defined in sets, using the VIHH Map Macros. Up to 63 VIHH Maps may be defined (more below). By default, all maps are set to disable the Vihh signal on all pins.
- Select a non-default VIHH Map in the test pattern, per-cycle. This enables or disables the Vihh signal to each pin, per-cycle. The default disables Vihh to all pins. In each test pattern, every pattern instruction selects one of the 64 VIHH Maps (`VIHH1` to `VIHH64`, default = `VIHH1`, which disables VIHH on all pins and

can't be modified by user code). In the pattern, a VIHH map is selected using the PINFUNC VIHH# instruction (Memory Test Patterns) or VEC/RPT VIHH# and VPINFUNC VIHH# instructions (Logic Test Patterns).

- The user must design the test pattern to enable the desired VIHH Map(s) for enough time (cycles) to ensure the VIHH level has time to slew to the desired voltage level (more below).
- During pattern execution, the VIHH level may be dynamically disabled using Over-programming Controls and Parallel Test, a specialized hardware facility used typically when programming multiple programmable DUTs in parallel. This topic is not covered further in this section, see Over-programming Controls and Parallel Test.

Using Magnum 1/2/2x, the VIHH voltage level can be programmed independently per-pin.

The following diagram is used to explain how VIHH operates:



Memory Test Patterns

```
PATTERN( myPAT )
%  MAR          INC
%  PINFUNC      VIHH1
   MAR          INC
%  PINFUNC      VIHH2
   MAR          INC
%  PINFUNC      VIHH3
   MAR          INC
%  PINFUNC      VIHH4
   MAR          INC
%  MAR          DONE
```

```
Default VIHH1 = AB
Explicit VIHH1 = AB
Explicit VIHH2 = AB
Explicit VIHH3 = AB
Explicit VIHH4 = AB
Default VIHH1 = AB
```

Logic Test Patterns

```
PATTERN( myLpat )
% VEC  HL10X
% VEC  HL10X, VIHH1
% VEC  HL10X, VIHH2
% VEC  HL10X, VIHH3
% VEC  HL10X, VIHH4
% VEC  HL10X
  VAR  DONE
```

Note the following:

- The example needs 3 pin lists:
  - `pl_Pa_only` : contains one pin = Pa
  - `pl_Pb_only` : contains one pin = Pb
  - `pl_Pab` : contains two pins = Pa and Pb
- The example needs 3 user-defined VIHH Maps:

```
VIHH_MAP( myVIHHmaps ){ // See VIHH_MAP()
   VIHH_ACTIVE( VIHH2, pl_Pa_only ) \\ See VIHH_ACTIVE()
   VIHH_ACTIVE( VIHH3, pl_Pab )
   VIHH_ACTIVE( VIHH4, pl_Pb_only )
}
```

- Two test pattern examples are shown in the diagram. They only include those instructions needed to control VIHH Map selection and increment to the next pattern instruction.
- The test pattern examples do NOT account for any VIHH voltage slew time, typically seen as repeated (or looped) instructions with a given VIHH map selected.
- In the first pattern instruction, `VIHH1` is selected by default. `VIHH1` does not enable the Vihh signal to any pins.
- In the 2nd pattern instruction, `VIHH1` is selected explicitly (for example). `VIHH1` does not enable the Vihh signal to any pins.
- In the 3rd pattern instruction, `VIHH2` is explicitly selected. This sends the Vihh signal to pins in the pin list names `pl_Pa_only`; i.e. one pin = Pa.
- In the 4th pattern instruction, `VIHH3` is explicitly selected. This sends the Vihh signal to pins in the pin list names `pl_Pab`; i.e. two pins = Pa and Pb.
- In the 5th pattern instruction, `VIHH4` is explicitly selected. This sends the Vihh signal to pins in the pin list names `pl_Pb_only`; i.e. one pin = Pb.
- In the last pattern instruction `VIHH1` is selected again, by default. `VIHH1` does not enable the Vihh signal to any pins.
- In the waveforms, the VIHH level is represented in green. The pin level when VIHH is not enabled is determined by the normal PE Driver operation; i.e. drive-0 (VIL), drive-1 (VIH) or tri-state.
- The Vihh signal changes state at 0nS (T0) in a given cycle. This is not programmable.
- Logic Test Patterns may also use the `VPINFUNC VIHH#` instructions to control VIHH Map selection (not shown in the example).

Note: any PE Driver on which Vihh is enabled will drive to the VIHH level, *regardless of the state of normal drive or I/O control signal from the Pattern and Timing System* (provided the pin is in Vihh Mode, see Magnum PE Driver Modes). When Vihh is not activated via a VIHH map, the PE Driver drives or tri-states under normal test pattern control.

3.17.1 Types, Enums, etc.

See Software, VIHH Maps, VIHH Map Macros.

Description

The following enumerated types are used by the VIHH Map Macros to define VIHH Maps, and in test pattern VIHH Map selection:

Usage

The `VihhNumber` enumerated type is used to identify VIHH Maps:

```
enum VihhNumber { VIHH1, VIHH2, VIHH3, VIHH4,
                  ... snip ...
                  VIHH61, VIHH62, VIHH63, VIHH64, VIHH_na
};
```

3.17.2 VIHH Map Macros

See PE Driver, VIHH Maps.

Description

The Test System Macros documented in this section are used to create one or more named sets of VIHH Maps. Note the following:

- A set of VIHH Maps is created using the `VIHH_MAP()` macro.

- Within the body-code of the VIHH_MAP( ) macro, up to 63 VIHH Map(s) may be defined using the VIHH_ACTIVE( ) macro. VIHH maps are identified as VIHH2 through VIHH64 (VIHH1 cannot be modified and thus can't be used in the VIHH_ACTIVE( ) macro).
- Within the body-code of the VIHH_MAP( ) macro, the INCLUDE_VIHH_MAP( ) macro may be used to include an existing [partial] set of VIHH Maps to define the set currently being defined.
- Multiple sets of VIHH Maps can be defined in the test program, however only one can be loaded (see Single Resource Types). If only one set is defined the system software will automatically select it during test program initialization. When multiple sets are defined two options are available: User codemay explicitly select one set of VIHH maps using the USE_VIHH_MAP macro. This is one of the Configuration Macros and must be used within a CONFIGURATION( ) block. When user-code does not select a set of VIHH Maps the system software automatically presents a selection dialog. Note, however, that this dialog is presented for each Site, which is not very user friendly.
- The EXTERN_VIHH_MAP( ) macro can be used to make a forward or external declaration.

Usage

The following macro creates a named set of VIHH Map(s):

```
VIHH_MAP( name )
```

The following macro is used to add one VIHH map to the set being defined:

```
VIHH_ACTIVE( VihhNumber VihhNum, PinList* pPinList )
```

The following macro allows the inclusion of an existing [partial] set of VIHH map(s) in the definition of the set being defined. This is an optional method, allowing reuse of a [partial] set of VIHH maps defined separately:

```
INCLUDE_VIHH_MAP( name )
```

The following macro is used to explicitly select one set of VIHH maps, by name. This must be used within a CONFIGURATION( ) block (see Configuration Macros):

```
USE_VIHH_MAP( name )
```

where:

name identifies a set of VIHH Maps. Must be a valid C identifier.

VihhNum identifies the VIHH map being defined. Legal values are of the VihhNumber enumerated type, but VIHH1 cannot be used.

pinlist identifies the pin(s) which will receive the Vihh enable signal when the VIHH map being defined is selected in the test pattern. **pinlist** must only contain signal pins.

Examples

The following example defines one VIHH Map in a set named simple_vihh_set. All pins in the pin list named special_pins will be set to the VIHH level in any pattern cycles which select the VIHH2 VIHH map:

```
VIHH_MAP( simple_vihh_set ) {
   VIHH_ACTIVE( VIHH2, special_pins )
}
```

The following example supports a DUT with three special test modes, each activated by VIHH on a different set of pins. The set of VIHH maps is named vihh_modes. The three pin lists were named after the special test modes they activate and are called block_prog_mode, erase_mode, and id_prog_mode. To block-program this DUT the test pattern will select VIHH2, causing the block_prog_mode pins to receive the VIHH level. Likewise, a VIHH3 enables VIHH on the erase_mode pins, etc.:

```
VIHH_MAP( vihh_modes ) {
   VIHH_ACTIVE( VIHH2, block_prog_mode );
   VIHH_ACTIVE( VIHH3, erase_mode );
   VIHH_ACTIVE( VIHH4, id_prog_mode );
}
```

3.18 Timing and Formatting Functions

See Timing & Formatting.

This section includes the following main topics:

- Overview
- Magnum Timing Rules
- Time-sets (TSET)
- Types, Enums, etc.
- Timing Generator Modes
- Cycle Time Functions
- Timing Formats
  - Supported Timing Formats
  - Window Strobe, Edge Strobe Modes
  - Drive Format vs. Strobe Format Selection
  - APG Chip Select Drive Format Selection
  - I/O Timing and Control
  - Double Clock Mode
- Programming Timing & Formats
  - `settime()`
  - `setedge(), getedge()`
  - Per-edge Functions: Drive/Strobe
  - Per-edge Functions: I/O Edges
  - `getformat()`
- Timing Examples

Also see MUX, Super-MUX and DDR.

3.18.1 Overview

See Timing & Formatting, Timing and Formatting Functions.

The following topics cover basic information about Magnum timing operation:

- Magnum Timing Rules
- Time-sets (TSET)

The software commonly used to access the Magnum timing system consists of only a few functions:

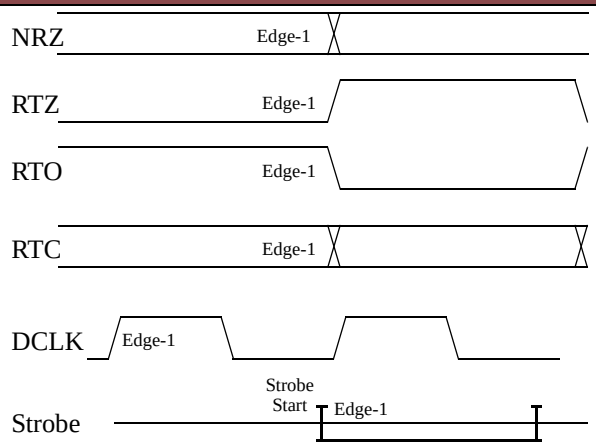
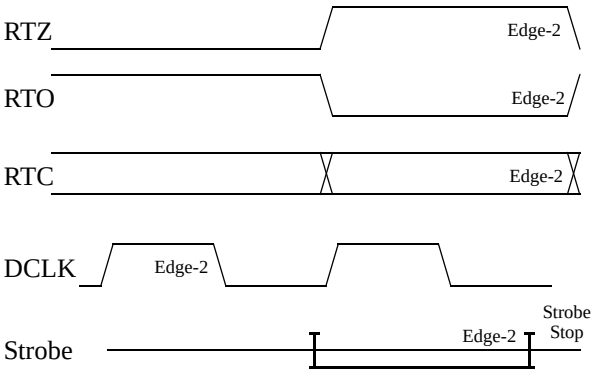
- `cycle()` - set or get the cycle period for one time-set. Executed multiple times for multiple time-sets.
- `edge_strobe()` - set the per-pin strobe mode (window vs. edge) for specified pin(s).
- `settime()` - program the drive, strobe and I/O edge times for one or more timing channels for one time-set. Each timing channel serves a pin-pair, see Functional Pin-pairs. Typically executed multiple times, for different pins, multiple time-sets, to program drive vs. strobe vs. I/O edge times.
- `tgmode()` - set or get the global timing generator mode. See Timing Generator Modes.

Additional timing related functions are available for specialized applications. These are used much less commonly:

- `setedge(), getedge()` - set or get the edge time value for one timing edge, for one time-set. A more generic solution, used instead of the Per-edge Functions: Drive/Strobe and Per-edge Functions: I/O Edges (below).
- Per-edge Functions: Drive/Strobe - set or get an edge time value for one drive or strobe edge number, for one time-set:
  - `setedge1()` `getedge1()`
  - `setedge2()` `getedge2()`
  - `setedge3()` `getedge3()`
  - `setedge4()` `getedge4()`
- Per-edge Functions: I/O Edges - set or get an I/O edge time value, for one time-set:
  - `setioedge1()` `getioedge1()`
  - `setioedge2()` `getioedge2()`
- `getformat()` - used to read-back the drive format of one specified pin in a specified time-set.

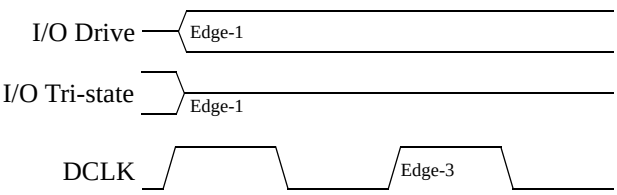
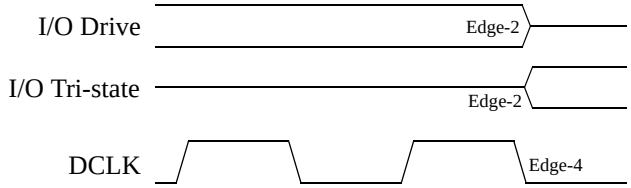
Each tester channel has 4 timing generators, each with a specific purpose:

Table 3.18.1.0-1 Magnum Timing Generators

Timing Generator	Purpose	Format Edges
TG-A	NRZ Edge-1[2] RTO/RTZ Edge-1[2] RTC Edge-1 DCLK Edge-1 (DCLKPOS shown) STROBE Start	
TG-B	RTO/RTZ Edge-2[2] RTC Edge-2 DCLK Edge-2 (DCLKPOS shown) STROBE Stop*	 * The Strobe Stop edge is not used in edge strobe mode, see Window Strobe, Edge Strobe Modes.

Notes
1) Double Clock is a special static pin mode in which TGs are used in a configuration which is quite different than all other formats. See Double Clock Mode.
2) NRZ timing applies to chip selects set to CSnT and CSnF. RTO/RTZ timing applies to CSnPT and CSnPF.

Table 3.18.1.0-1 Magnum Timing Generators

Timing Generator	Purpose	Format Edges
TG-C	I/O Drive* I/O Tri-state* DCLK Edge-3 (DCLKPOS shown)	 * Both I/O Drive and I/O Tri-state can both be generated by TG-C and TG-D. See Timing Generator Modes and I/O Timing and Control
TG-D	I/O Drive* I/O Tri-state* DCLK Edge-4 (DCLKPOS shown)	 * Both I/O Drive and I/O Tri-state can both be generated by TG-C and TG-D. See Timing Generator Modes and I/O Timing and Control

Notes
1) Double Clock is a special static pin mode in which TGs are used in a configuration which is quite different than all other formats. See Double Clock Mode.
2) NRZ timing applies to chip selects set to CSnT and CSnF. RTO/RTZ timing applies to CSnPT and CSnPF.

3.18.2 Magnum Timing Rules

See Timing and Formatting Functions, Overview.

As in all ATE systems, trade-offs exist between cost, features and performance. The Magnum timing capabilities are best described as follows:

- When using a single time-set, actual timing matches programmed timing.
- When using multiple time-sets, timing for one time-set is as-programmed, with timing in the other time-set(s) less accurate, but predictable.

- Each of the 64 timing channels on a Site Assembly Board drives two PE channels (A/B, see Functional Pin-pairs). This means that (at least) two DUTs can be tested concurrently (in parallel) receiving the same functional AC test stimulus on each pin of both DUTs.

The Magnum timing rules are defined below.

1. The System Clock has a 100MHz Oscillator reference, which is the basis for all functional timing: cycle period(s), edge times, etc. This is called the **Master Clock** . However, as noted next, and except as referenced in rule-6., the master clock is further multiplied 10x before being used.
2. In use, when calculating cycle period values and edge timing values, the Master Clock is effectively multiplied by 10, to provide a 1nS clock reference. Subsequent cycle period values and (digital) edge timing values are derived by counting this 1nS clock (more below). For documentation purposes, this effective system source is called the **X-clock** (10x clock source).
3. Cycle periods are generated by counting from 20 to 10230 X-clocks. This provides a cycle period range of 20nS to 10.23uS, in 1nS increments.
4. Any time a cycle period is programmed (see `cycle()`) all cycle period count values (all time-sets) are re-calculated. For performance reasons, this is not done until just before a functional pattern is executed (`funtest()`) or a timing getter function is executed (i.e. `getedge()`, etc.).
5. For all cycle period(s) programmed in increments of 1nS the actual cycle period value will match the programmed value. For other values the actual value will be the next lower 1nS increment from the programmed value.
6. If the cycle period which sets the Master Clock is short enough to use only 2 master clock cycles then no other cycle periods (in other time sets) may be shorter.
7. Each Magnum timing channel has 4 independent timing generators used to generate the drive, strobe, and I/O edge timing used during functional tests. See Timing Generator Block Diagram.
8. The `settime()` function is used to program edge timing and formats. Each `settime()` execution programs the edge time(s) of one format (drive, strobe, or I/O) for one or more pin(s), in one time-set. For example, the following statements program all of the possible formats for a given set of pins (`myPins`) for time-set 2 (`TSET2`):

```
settime( TSET2, myPins, RTZ, 0 NS, 10 NS );
settime( TSET2, myPins, STROBE, 13 NS, 20 NS );
settime( TSET2, myPins, IODRIVE, 0 NS, 0 NS );
settime( TSET2, myPins, IOSTROBE, 0 NS, 0 NS );
```

Note: each timing channel drives two PE channels (one pin-pair, see Functional Pin-pairs and Magnum Pattern & Timing System). When `settime()` is executed, the pin list argument is used to identify which timing channel(s) are to be programmed, as follows:

A given timing channel is identified if any pins (or both pins) of a pin-pair are included in the pin list.

This timing channel is then programmed only once, even if both pins in the pin pair are included in the pin list.

---

9. Edge placement range is from 0nS (T0) to the earlier of:
   - The (cycle period X 2) - 100pS
   - 10.2uS - 100pS = 10.999pS
10. In hardware, edge timing generation has two components:
    - Digital value: controlled by the individual timing generators, in increments of X-clock (1nS clock). Each timing generator generates a timed edge by counting 0 to 10199 X-clocks. These values are programmable per-format (drive, strobe and I/O) and per-time-set.
    - Analog value: controlled by the timing verniers, which provide a 1nS range with 100pS resolution. Each timing channel has one set of verniers each for drive vs. strobe vs. I/O formats i.e. vernier values do not change per-time-set.

    Thus, on a given pin, for a given format, the last time-set programmed will be set with 100pS resolution. Edges in the other time-sets will use the same vernier value (per-format) thus a difference may exist between the programmed value and the actual value.

11. The digital value and vernier value for the timing edge which sets the vernier is calculated as:

```
Count = (int) ( Programmed_Time / X-clock )
DigitalVal = ( Count * X-clock )
Vernier = ( Programmed_Time - DigitalVal )
```

For example, a programmed edge time of 35.5nS results in:

```
Count = (int) ( 35.5nS / 1nS ) = 35
DigitalVal = ( 35 * 1nS ) = 35nS
Vernier = ( 35.5nS - 35.0nS ) = 0.5nS (500pS)
```

12. The most recent settime() execution which sets a given format (for a given pin) also sets the vernier for that format. Since only one vernier value exists for a given format edge (per pin) it affects all edges of the same format on the pin, in all time-sets.
The digital value and actual value for the timing edge which does not set the vernier is calculated as:

```
Count = (int) ( Programmed_Time / X-clock )
DigitalVal = ( Count * X-clock )
ActualVal = ( DigitalVal + Vernier )
Error = ( Programmed_Time - ActualVal )
```

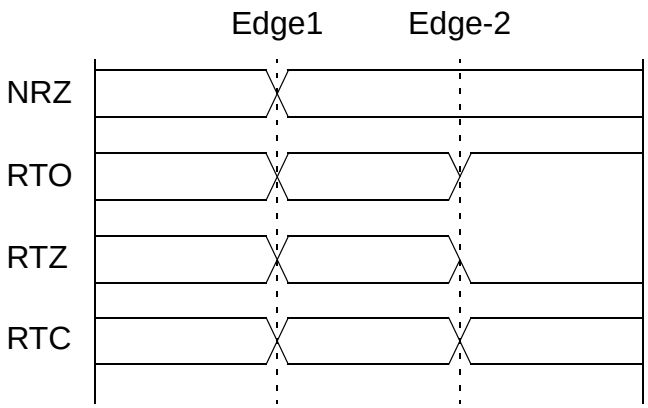
The following table shows this using several edge time values and the vernier value (500pS) calculated above. In this example, the edge which set the vernier was programmed in time-set 2 (TSET2):

Sets Vernier?	Time Set	Prog. Time	Count	Count* X-clock	Add Vernier	Actual Time	Error
	1	11.3nS	11	11.0nS	500pS	11.5nS	200pS
Yes	2	35.5nS	35	35.0nS	500pS	35.500nS	0nS
	3	22.0nS	22	22.0nS	500pS	22.5nS	500pS
	4	34.4nS	34	34.0nS	500pS	34.5nS	100pS
	5	0.0nS	0	0nS	500pS	500pS	500pS
	...	...	...	...	...	...	...

When multiple time sets are used (timing on-the-fly), the difference between programmed and actual edge times is zero only when the edge times are specified in ((multiples of the X-clock) + vernier) or ((programmed edge - vernier) % X-clock) == 0.

When this rule is violated the difference between the programmed edge time and actual edge time is dependent on the vernier value. The worse case difference is 1nS.

13. The return-to-complement (RTC) format sets the drive verniers differently than for other drive formats. The following diagram describes how the drive verniers are set for each of the different drive formats. Remember, for a given pin, the last drive format programmed sets both the drive-high and drive-low vernier for that pin:



NRZ: both the drive-high and the drive-low verniers are set by the edge-1 timing value.

RTO: the drive-low vernier is set by the edge-1 timing value, the drive-high vernier is set by the edge-2 timing value.

RTZ: the drive-high vernier is set by the edge-1 timing value, the drive-low vernier is set by the edge-2 timing value.

RTC: both the drive-high and the drive-low verniers are set by the edge-1 timing value i.e. edge-2's timing inherits the vernier value (100pS resolution) set by edge-1's timing value.

14. The minimum drive pulse-width is 4nS.
    The minimum window strobe width is 5nS.
15. The minimum time from the end of one strobe to the start of the next strobe is 5nS.
16. The minimum time from the start of one strobe to the start of the next strobe depends upon the strobe type:
    - Edge Strobe = 5nS
    - Window Strobe = 12nS (2 * 5nS)
17. The Strobe Stop edge is not used in edge strobe mode, see Window Strobe, Edge Strobe Modes.
18. Using Magnum, the timing generator mode is set to 3, ignoring any Timing Generator Modes set using `tgmode()`. In effect (and for those who have Maverick-I/-II experience)

this means that Double Data Rate (DDR) Mode and Double Clock Mode are always usable.

19. Drive and I/O timing edges can be intentionally disabled (i.e., prevent a given edge generator from firing) by programming an edge time to -1. Edge time getter functions will return -1 for any disabled edge. Note that -1 is an indicator (flag) not a usable timing value, thus any user code computations performed using values returned from timing getter functions (`getedge()`, `getedge1()`, etc.) must explicitly handle -1 uniquely from all other returned timing values. When setting edge times the system software will treat any negative edge time value as -1.

Note: it is **critical** that -1 NOT be used to disable strobe edges. Doing so causes improper strobe operation and invalid test results.

Note: early Maverick-I/-II test programs may have disabled timing edges using a time value of 10220, 10230, or 10240 (nS). This is not supported using Magnum.

3.18.3 Time-sets (TSET)

See Timing and Formatting Functions, Magnum Timing Rules.

The Magnum architecture supports 32 time-sets (TSETs). This means the user can define 32 complete sets of timing, each consisting of:

- A cycle period value. See `cycle()`.
- A drive format and associated edge times for each pin-pair. See `settime()`.
- Strobe timing for each pin-pair. See `settime()`.
- I/O timing for each pin-pair. I/O timing is independently programmable for drive cycles vs. tri-state cycles. See `settime()`.

During functional test execution, the test pattern selects, on a cycle-by-cycle basis (on-the-fly), one time-set per pattern instruction. This allows the parameters noted above to change, on a cycle-by-cycle basis, as the pattern executes.

Note: waveform timing and formats are closely linked, so they are programmed together using the `settime()` function. When the documentation discusses time-sets, format sets are also implied.

As noted above, during pattern execution, a time-set is selected in each tester cycle, based on explicit pattern statements or, if not explicitly specified, the default time-set (`TSET1`) is used. Timing sets are formally named `TSET1` through `TSET32`.

In the test pattern a time-set is selected using the `PINFUNC TSET#` instruction (Memory Test Patterns) or `VEC TS#` and `VPINFUNC TSET#` instructions (Logic Test Patterns,). In Mixed Memory/Logic Patterns, default operation uses the time-set selection from the memory instruction (MAR Engine), but the logic instruction selection (VAR Engine) can be enabled, per cycle, using the `PINFUNC VTSET` instruction.

In Scan Test Patterns the time-set is selected using the same syntax as logic patterns.

3.18.4 Types, Enums, etc.

See Timing and Formatting Functions.

Description

The following enumerated types are used in support of various timing related software functions:

Usage

The `TSETNumber` enumerated type is used to specify a time-set when programming edge times, cycle periods, and to select a time-set in the test patterns:

```
enum TSETNumber { TSET1, TSET2, TSET3, TSET4,
                  TSET5, TSET6, TSET7, TSET8,
                  TSET9, TSET10, TSET11, TSET12,
                  SET13, TSET14, TSET15, TSET16,
                  TSET17, TSET18, TSET19, TSET20,
                  TSET21, TSET22, TSET23, TSET24,
                  TSET25, TSET26, TSET27, TSET28,
                  TSET29, TSET30, TSET31, TSET32,
                  TSET_na };
```

The EdgeTypes enumerated type is used to identify one edge type when setting or getting individual timing values:

```
enum EdgeTypes {  t_drive_edges,
                  t_strobe_edges,
                  t_IO_drive_edges,
                  t_IO_strobe_edges };
```

The TimeOption enumerated type is used when getting a currently programmed time value, to specify whether a programmed value or the actual value is returned. See cycle(), setedge(), getedge():

```
enum TimeOption{ t_actual, t_programmed };
```

The TGFormat enumerated type is used to specify a timing format when programming edge times. See Timing Formats, Programming Timing & Formats, settime(), DDR Timing. Note that SBC is not usable on Magnum:

```
enum TGFormat {NRZ, RTO, RTZ, SBC, RTC, STROBE, IODRIVE, IOSTROBE,
               DCLKPOS, DCLKNEG, DDR_DRIVE, DDR_STROBE,
               DDR_IODRIVE, DDR_IOSTROBE };
```

3.18.5 Timing Generator Modes

See Timing and Formatting Functions, Magnum Timing Rules.

Description

The tgmode() function is used to set or get the global timing generator mode.

Note: using Magnum, the system software sets the timing generator mode (TG mode) to 3 and ignores other mode values set using tgmode(). This function remains documented to aid migration of Maverick-I/-II test programs to Magnum.

Usage

The following function sets the global TG mode (see Note:):

```
BOOL tgmode( int mode );
```

The following function returns the currently set TG mode:

```
int tgmode();
```

where:

mode specifies the desired TG mode. See Note:.

The getter version of `tgmode()` returns the currently programmed TG mode.

The s*etter* version of `tgmode()` returns TRUE if the TG mode sets correctly, or FALSE if an error occurs (incorrect hardware revision level, etc.). However, this is a fatal error, unloading the test program.

Examples

```
BOOL ok = tgmode( 3 );
   if( !ok ) output("ERROR setting TG mode");
output("The current TG mode => %d", tgmode() );
```

3.18.6 Cycle Time Functions

See Timing and Formatting Functions.

Description

The `cycle()` function is used to set or get the cycle period for one time-set.

> Note: this section describes the most basic timing rules related to cycle period operation. For additional details, see Magnum Timing Rules.

A unique cycle period value can be programmed for each of the 32 available Time-sets (TSET). Note the following:

- During the initial program load, the cycle period in all time-sets are set to 100nS. The system software does not otherwise change any cycle period values.
- The cycle time can be programmed between 20nS to 10.2uS, in 1nS increments (see X-clock).
- Attempting to program a cycle period less than the minimum legal value or greater than the maximum legal value will generate a warning message and leave the cycle period unchanged.

- As in all test systems, inherent hardware capabilities make it possible to program waveform edge times and cycle period values which cannot be exactly generated by the hardware. Hardware resolution and related timing rules are common reasons that *programmed* values are not exactly the same as *actual* hardware values. Detailed operation is described in Magnum Timing Rules.
- The system software stores both the programmed and actual timing values. By default, all of the timing *getter* functions return the *actual* value generated by the hardware. The `TimeValueType` argument may be used to cause the `cycle()` getter function to return the programmed cycle period value.

Note: executing the `cycle()` function while a test pattern is actively looping (see Patterns That Loop Forever and `start_pattern()`) can cause unpredictable side effects. Do NOT do it.

Usage

The following function sets the cycle time for the specified TSET:

```
void cycle( TSETNumber TSET, double Value );
```

The following function gets the current *actual* cycle time for the specified TSET:

```
double cycle( TSETNumber TSET );
```

The following function gets the current cycle time for the specified TSET. The `TimeValueType` argument determines whether the programmed or actual cycle time is returned (see Description):

```
double cycle( TSETNumber TSET, TimeOption TimeValueType );
```

where:

`TSET` identifies the time-set being programmed. Legal values are of the `TSETNumber` enumerated type.

`Value` specifies the desired cycle period. Units may be used (see Specifying Units).

`TimeValueType` specifies whether the *programmed* value (`t_programmed`) or the *actual* value (`t_actual`) is returned. Legal values must be one of the TimeOption enumerated type. The default = `t_actual`.

The `cycle()` getter functions return the currently programmed cycle period for the specified time-set.

Examples

The following example sets the cycle period for TSET1 to 500nS:

```
cycle( TSET1, 500 NS );
```

The following example sets the cycle period for TSET2 to 66.6nS:

```
cycle( TSET2, 66.66 NS );
```

The following example gets both the programmed and actual cycle period value for TSET2. Both values are output plus the difference:

```
double progval = cycle( TSET2, t_programmed );
double actval = cycle( TSET2, t_actual );
output(" Programmed TSET2 cycle period => %0.0f pS", progval );
output(" Actual TSET2 cycle period => %0.0f pS", actval );
output(" Difference => %0.0f pS", progval - actval );
```

3.18.7 Timing Formats

See Timing and Formatting Functions.

- Supported Timing Formats
- Drive Format vs. Strobe Format Selection
- APG Chip Select Drive Format Selection
- Window Strobe, Edge Strobe Modes
- I/O Timing and Control
- Double Clock Mode

3.18.7.1 Supported Timing Formats

See Timing and Formatting Functions, Timing Formats.

The following timing formats are available independently on each pin-pair (see Functional Pin-pairs), *on-the-fly*. A drive format *AND* a strobe, *AND* I/O timing are each independently programmable, on every pin-pair, in each time-set:

Table 3.18.7.1-1 Timing Formats

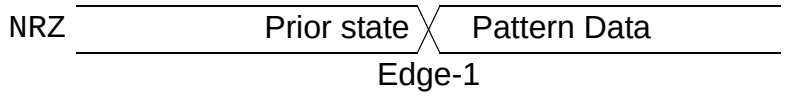
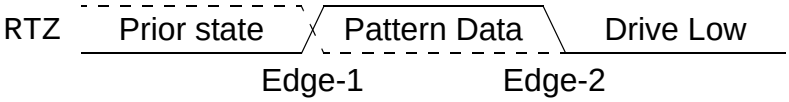
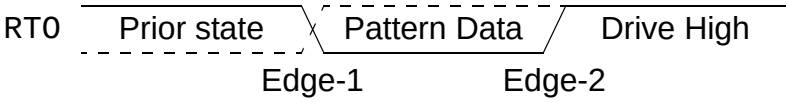
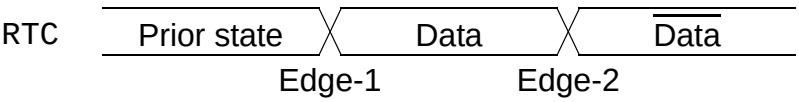
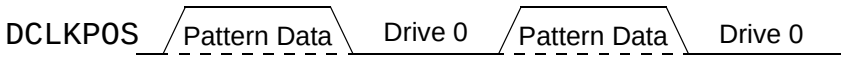
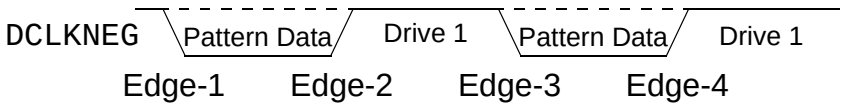
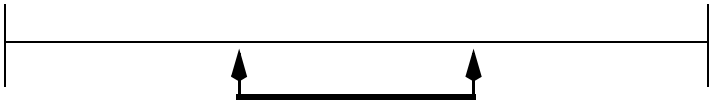
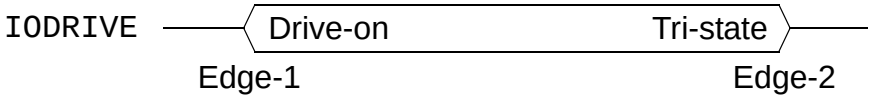
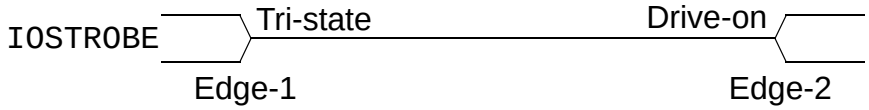
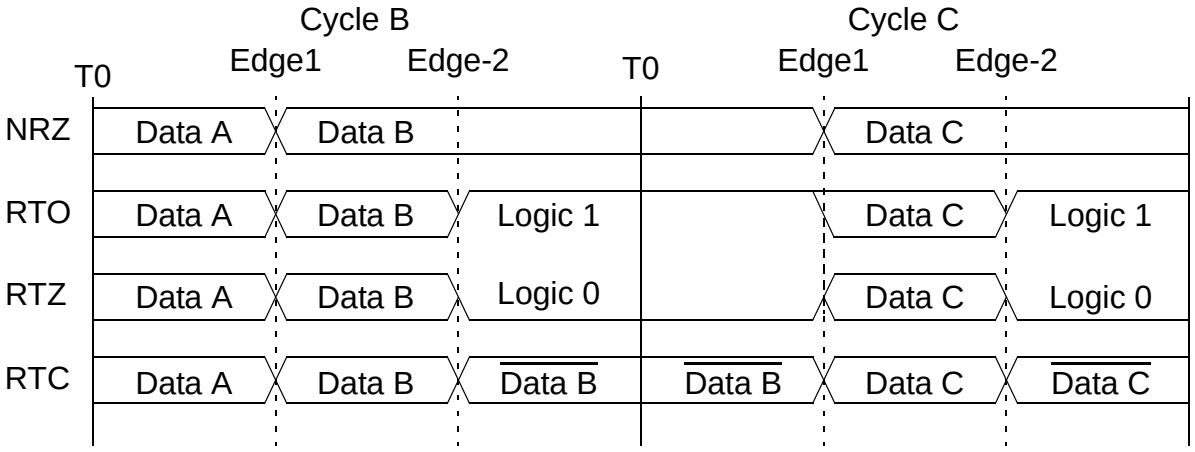
Format	Format & Timing Edges
NRZ Non-return-to-zero Drive format	 NRZ timing applies to chip selects set to CSnT and CSnF.
RTZ Return-to-zero Drive format	 RTZ timing applies to chip selects set to CSnPT and CSnPF.
RTO Return-to-one Drive format	 RTO timing applies to chip selects set to CSnPT and CSnPF.
RTC Return-to-complement Drive format	
DCLKPOS Double Clock Positive Drive format See Double Clock Mode	
DCLKNEG Double Clock Negative Drive format See Double Clock Mode	

Table 3.18.7.1-1 Timing Formats

Format	Format & Timing Edges
STROBE Strobe format	 Note: the Strobe Stop edge is not used with edge strobes, see `edge_strobe()` and Window Strobe, Edge Strobe Modes.
IODRIVE I/O format	
IOSTROBE I/O format	

The operation of the drive formats are further illustrated below. Two tester cycles, called **`Cycle B`** and **`Cycle C`**, are shown. **`Cycle A`** is intentionally not shown. For each format type, each *potential* transition is shown. Each transition is labeled with the corresponding pattern data used to control that transition. The labels are based on the pattern cycle from which the data originated (A, B, or C) or, when not controlled from the pattern, an explicit logical one or logical zero is shown.



In these diagrams, the conventional RTO and RTZ waveform shapes are not readily visible. This is because it is the pattern data which determines whether a RTO or RTZ *pulse* actually occurs and the pin's prior logic state also affects whether a pulse is actually seen. For both RTO and RTZ, the second edge is not controlled by pattern data; i.e. they are explicitly logic 1 (RTO = return to logic 1) or logic 0 (RTZ = return to logic 0).

3.18.7.2 Window Strobe, Edge Strobe Modes

See Timing and Formatting Functions, Timing Formats.

Description

The `edge_strobe()` function is used to set or get the strobe mode for one or more pin-pairs (see Functional Pin-pairs). Note the following:

- The hardware design supports both window and edge strobes, on a per-pin-pair basis.
- A window strobe has a non-zero width; i.e. the strobe is active for a window of time. An edge strobe is effectively a zero-width strobe.
- The strobe mode for all pins is set to window mode during initial program load. The system software does not otherwise change the mode.
- The mode cannot be changed during pattern execution but can be changed between pattern executions.

Usage

The following function sets the strobe mode for one or more pins:

```
void edge_strobe( PinList* pPinList, BOOL State );
void edge_strobe( DutPin *pDutPin, BOOL State );
```

The following function gets the current strobe mode for one pin:

```
BOOL edge_strobe( DutPin *pDutPin );
```

where:

`pPinList` identifies one or more pin(s) to be programmed. In Multi-DUT Test Programs only the pin(s) of DUT(s) in the Active DUTs Set (ADS) are affected.

`State` specifies whether the strobe mode is edge strobe (TRUE) or window strobe (FALSE).

pDutPin is used in two contexts:

- In the setter function, identifies one pin to be programmed. In Multi-DUT Test Programs the same pin of all DUT(s) in the Active DUTs Set (ADS) are programmed.
- In the getter function, specifies one pin to be read. In Multi-DUT Test Programs the value is read from the first DUT in the Active DUTs Set (ADS).

The getter version of `edge_strobe()` returns TRUE if the currently programmed strobe mode is edge strobe for the specified **pDutPin**, otherwise FALSE is returned.

Example

```
edge_strobe( data_bus, TRUE );
BOOL mode = edge_strobe( D1 );
```

3.18.7.3 Drive Format vs. Strobe Format Selection

See Timing and Formatting Functions, Timing Formats.

During pattern execution, on a cycle-by-cycle and per-timing channel basis, the selection of drive format vs. strobe format is controlled from the test pattern. This is independent of time-set switching, and the pattern syntax used is different for Memory Test Patterns vs. Logic Test Patterns.

Note: each timing channel serves two pins i.e. a pin-pair. See Functional Pin-pairs. The software is programmed per-pin, but both pins of a pin-pair are affected when either pin is programmed.

In Logic Test Patterns, the Logic Vector Bit Codes specified for each timing channel controls drive vs. strobe format selection. The `H`, `L`, or `X` tokens select the strobe format (and tri-state the pin), and `1` or `0` tokens select the drive format and cause the pin to drive. As indicated, these tokens also determine I/O Timing and Control.

In Memory Test Patterns, the use of the `MAR READ`, `READV`, `READZ` or `READUDATA` instructions cause a strobe format to be selected, but only on those timing channels which are scrambled to the APG Data Generator outputs (`t_d35..t_d0`) in the Pin Scramble Map selected in a given pattern instruction. On these same timing channels the `PINFUNC ADHIZ` instruction separately controls I/O switching, but has no effect on drive vs. strobe format selection.

Also in Memory Test Patterns, two of the Chip Select data sources (t_cs1 and t_cs2) have tri-state and strobe capability (t_cs3 through t_cs8 are permanently set to drive). For timing channels which are scrambled to t_cs1 and t_cs2, the drive vs. strobe format selection is controlled using the CHIPS CSmRDT or CSmRDF instructions. Any pattern instructions which use these instructions cause the strobe format to be selected (and the pin-pairs to tri-state), otherwise the drive format is selected. See below for tri-state control information.

3.18.7.4 APG Chip Select Drive Format Selection

See Timing and Formatting Functions, Timing Formats.

In Memory Test Patterns, both the drive format (NRZ vs. RTO/RTZ, etc.) and format polarity (RTO vs. RTZ) of the 8 chip selects signals (t_cs1 through t_cs2) are actually controlled from the test pattern, overriding the format using settime(). For example, the following code specifies that a return-to-one (RTO) format is to be applied to the WE_pin in pattern cycles selecting time-set TSET1:

```
settime ( TSET1, WE_pin, RTO, 5 NS, 20 NS );
```

In reality, the test pattern CHIPS instruction(s) determines whether a given chip select (WE_pin in this example) drives statically TRUE (CSnT) or FALSE (CSnF), or pulses TRUE (CSnPT) or pulses FALSE (CSnPF).

The polarity of TRUE and FALSE is specified in the test program using cs_active_high(), which must be executed *BEFORE* test patterns are loaded.

The edge times specified using settime() *do* control when the drive edge transitions occur. And, it is good programming practice to set the drive format (RTO vs. RTZ) to match the active polarity of the pin being set up; i.e. if the WE_pin is active low it is good practice to specify RTO as the timing format in the settime() statement.

3.18.7.5 I/O Timing and Control

See Timing and Formatting Functions, Timing Formats.

This section describes how functional test I/O switching is controlled.

Note: the information below applies to non-DDR I/O timing. For Double Data Rate (DDR) Mode operation, see DDR I/O Timing.

Each time-set can program drive ***and*** strobe ***and*** I/O timing for each timing channel

Note: each timing channel serves two pins i.e. a pin-pair. See Functional Pin-pairs. The software is programmed per-pin, but both pins of a pin-pair are affected when either pin is programmed.

- Two I/O timing formats are available: `IODRIVE` and `IOSTROBE`. These are programmable per- timing channel, just like drive formats and strobe formats. All other information below applies independently per-timing channel.
- `IODRIVE` timing edges occur only in drive cycles. `IOSTROBE` timing edges occur only in tri-state cycles.
- Pins which are scrambled to the APG Data Generator (see Pin Scramble Maps) will tri-state in pattern cycles which include the `PINFUNC ADHIZ` instruction (Memory Test Patterns). These pins will drive in pattern cycles which don't explicitly include this instruction. Note that the `adhiz()` function is not supported using Magnum..
- Pins which are scrambled to APG Chip Selects 1 or 2 (`t_cs1` & `t_cs2`) will tri-state in pattern cycles which include the `CHIPS CSmHIZ`, `CSmRDT` or `CSmRDF` instructions (Memory Test Patterns). These pins will drive in pattern cycles which explicitly include these instructions.
- Pins which are scrambled to the APG Chip Selects 3 through 8 (`t_cs3` .. `t_cs8`) will always drive.
- Pins which are scrambled to the APG Address Generator will always drive.
- Pins which are scrambled to Logic Vector Memory (LVM) or Scan Vector Memory (SVM) will tri-state in pattern cycles which use L/H/Z/V/X on those pins (Logic Test Patterns and Scan Test Patterns). These pins will drive in pattern cycles which use `0/1` (see Logic Vector Bit Codes).
- The following functions can over-ride test pattern tri-state/drive control: `tri_state()`, `drive_only()`, and `io_enable()`.
- An `IODRIVE` edge causes the pin to drive the ***previous driven logic state***. Any intervening strobe or tri-state cycles have no effect on the previous logic state driven.

- The *default* I/O timing is the same for all TG modes:
  - In drive cycles, drive-on at T0 (0nS), disable the tri-state edge
  - In tri-state cycles, tri-state at T0 (0nS), disable the drive-on edge
  This is important when IOSTROBE is explicitly programmed and IODRIVE is not, or vice versa.
- In hardware, I/O timing is controlled by TG-C and TG-D (see Overview). TG-C and TG-D each effectively have an independent vernier. This means that IOSTROBE timing values have no effect on IODRIVE timing values, and vice versa. And, IODRIVE vs. IOSTROBE are calibrated independently.
- IODRIVE and IOSTROBE timing edges can be disabled by programming the time value to -1. For example:

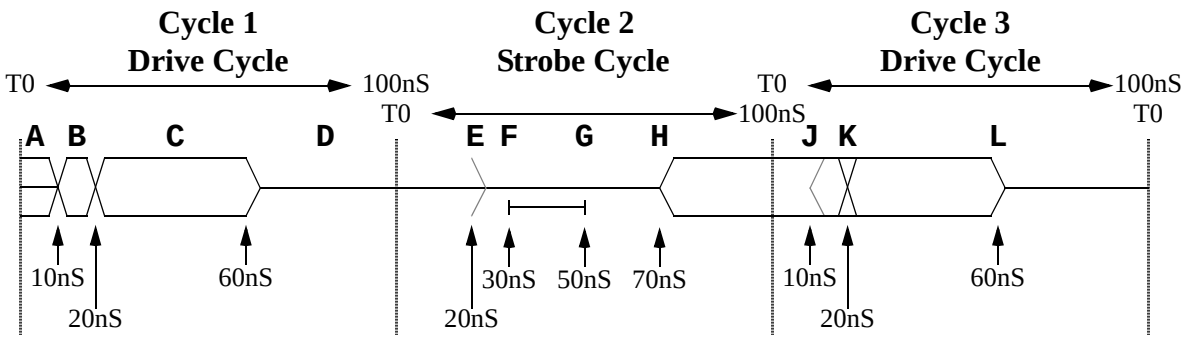
  ```
  settime ( TSET1, pins, IODRIVE, 10 NS, -1 );
  ```

  Note that disabling *both* edges of IODRIVE or IOSTROBE will disable all drive-on edges and all tri-state edges until timing is reprogrammed. This will likely not be useful (i.e. bad).

Several I/O timing examples are included below, with annotations of important features.

Example 1:

```
edge_strobe( pins, FALSE ); // See edge_strobe()
cycle( TSET1, 100 NS );
settime( TSET1, pins, NRZ , 20 NS );
settime( TSET1, pins, IODRIVE, 10 NS, 60 NS );
settime( TSET1, pins, STROBE, 30 NS, 50 NS );
settime( TSET1, pins, IOSTROBE, 20 NS, 70 NS );
```



This timing diagram shows 3 tester cycles, each containing the drive, strobe, and I/O control edges programmed in the code example above. Referring to the letter annotations, note the following:

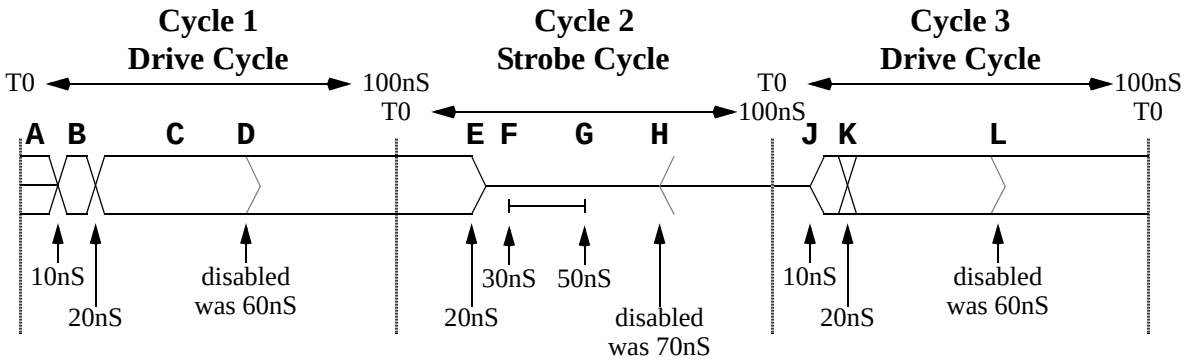
- In cycle-1, region **A** has indeterminate I/O state until the first IODRIVE edge occurs at 10nS.
- In cycle-1, during region **B** the pin is driving, but since this is the first cycle of the pattern the logic state is indeterminate until the programmed NRZ edge occurs at 20nS.
- Region **C** represents the valid drive data set by the NRZ edge which occurred at 20nS. This region ends when the pin is tri-stated by the second IODRIVE edge programmed at 60nS.
- The region described by **D-H** represents the time the pin remains tri-stated. The pin drives again when the second IOSTROBE edge programmed at 70nS in the strobe cycle occurs (edge **H**).
- In the strobe cycle the first IOSTROBE edge is programmed to tri-state the pin at 20nS. This is shown in gray (edge **E**) because the pin is already tri-stated from cycle-1.
- In the strobe cycle a window strobe is generated, starting at 30nS (edge **F**) and ending at 50nS (edge **G**).
- In the strobe cycle, the second IOSTROBE edge causes the pin to drive at 70nS (edge **H**). The logic state driven is the *previous drive logic state* from an earlier tester cycle (cycle-1 in this example). The region described by **H-K** represents the time this *prior logic state* is driven.
- In the cycle-3, the first IODRIVE edge is programmed to cause the pin to drive at 10nS. This is shown in gray (edge **J**) because the pin is already driving due to the second IOSTROBE edge programmed at 70nS in cycle-2 (edge **H**).
- In the cycle-3, edge **K** represents the new drive data set by the NRZ edge programmed at 20nS. This region ends when the pin is tri-stated by the second IODRIVE edge programmed at 60nS (edge **L**).

Example 2:

This example is identical to the previous example except that the second edge of both the IODRIVE and the IOSTROBE edges is disabled (programmed to -1). Note the effect on the resulting I/O waveforms.

```
tgmode( 1 ); // See tgmode()
edge_strobe( pins, FALSE ); // See edge_strobe()
cycle( TSET1, 100 NS );
settime( TSET1, pins, NRZ, 20 NS );
```

```
settime( TSET1, pins, IODRIVE, 10 NS, -1 );
settime( TSET1, pins, STROBE, 30 NS, 50 NS );
settime( TSET1, pins, IOSTROBE, 20 NS, -1 );
```



This timing diagram shows the same 3 tester cycles, each containing the drive, strobe, and I/O control edges as programmed in the code example above. The key differences from the previous example are noted below:

- In cycle-1, the pin does not tri-state because the second `IODRIVE` edge is disabled. The previous location of the disabled edge (60nS) is shown in gray (edge **D**) for comparison with the previous example.
- The region defined by **C-E** represents the time the pin continues to drive the logic state from the `NRZ` edge which occurred at 20nS in cycle-1. The pin tri-states at 20nS in cycle-2 as programmed by the first `IOSTROBE` edge (edge **E**).
- In cycle-2, the pin does not drive because the second `IOSTROBE` edge is disabled. The previous location of the disabled edge (70nS) is shown in gray (edge **H**) for comparison with the previous example.
- The region described by **E-J** represents the time the pin remains in tri-state.
- The pin drives at 10nS in cycle-3 as programmed by the first `IODRIVE` edge (edge **J**). The logic state driven is the *previous drive logic state* from an earlier tester cycle (cycle-1 in this example). The region described by **J-K** represents the time this *prior logic state* is driven
- As in cycle-1, the pin does not tri-state at 60nS in cycle-3 because the second `IODRIVE` edge is disabled.

3.18.7.6 Double Clock Mode

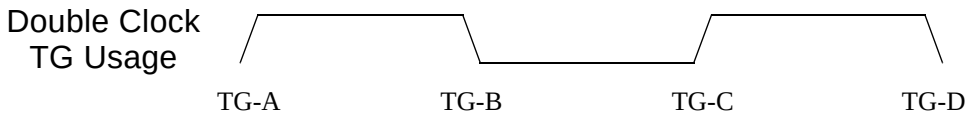
See Timing and Formatting Functions, Timing Formats.

Double clock timing formats provide a means of generating positive (DCLKPOS) or negative (DCLKNEG) double clocks, where four edge times are specified.

Double Clock is a special hardware mode. Note the following:

- Double clock mode is a static mode, which means that it is not possible to switch to other formats while using double clock. More specifically, on a given pin, double clock mode is enabled when the DCLKPOS or DCLKNEG format is programmed on that pin, and is disabled when any other format (including strobe or I/O formats) is programmed on that pin.
- To use double clock mode also requires using pe_driver_mode_set(), to set the PE driver mode to Dclk Mode. See Magnum PE Driver Modes.
- If a given pin is programmed to the DCLKPOS or DCLKNEG format the last one wins; i.e. it is not possible to generate both formats in a given pattern execution.
- Pin(s) set to Dclk Mode can only drive i.e. test pattern signals to tri-state the pin are ignored.
- Both pulses of a double clock format are controlled by the same pattern data bit; i.e. either both pulses are enabled or disabled, per-timing-channel, per-cycle.
- As noted in Magnum Timing Generators, double clock formats are generated using TG-A, TG-B, TG-C and TG-D, which are configured quite differently than when generating all other formats. This is only significant for one reason: after a pin has been configured to generate a double clock, if/when that pin is set up to generate any other format(s) the IODRIVE and IOSTROBE timing for that must also be [re]programmed. This must be done to restore I/O timing controlled by TG-C/TG-D. To restore default I/O timing use the following:

```
settime( TSETn, your_pins, IODRIVE, 0 NS, -1 );
settime( TSETn, your_pins, IOSTROBE, 0 NS, -1 );
```



3.18.8 Programming Timing & Formats

See Timing and Formatting Functions, Timing Formats, Magnum Timing Rules.

The `settime()` function is used to program per-pin edge timing and formats.

The following functions are used less commonly. They allow individual timing edges to be set or read-back (get).

`setedge1()`	`getedge1()`
`setedge2()`	`getedge2()`
`setedge3()`	`getedge3()`
`setedge4()`	`getedge4()`
`setioedge1()`	`getioedge1()`
`setioedge2()`	`getioedge2()`
`getformat()`	

3.18.8.1 settime()

See Timing and Formatting Functions, Programming Timing & Formats, Magnum Timing Rules.

Description

The `settime()` function is used to program edge timing and formats per-pin, per time-set.

Read Magnum Timing Rules.

A drive format and edge time(s) *AND* strobe edge times, *AND* I/O edge times are each programmable per-timing channel, in each time-set.

> Note: each timing channel serves two pins i.e. a pin-pair. See Functional Pin-pairs. The software is programmed per-pin, but both pins of a pin-pair are affected when either pin is programmed.

To program all three formats for one or more pin(s) requires executing `settime()` four times for each time-set used:

- Once to specify a drive format and corresponding edge times
- Once to specify strobe edge times
- Once to specify drive cycle I/O timing
- Once to specify tri-state cycle I/O timing (see below).

Programming a given edge time to -1 disables that edge.

Note: do NOT use -1 to disable strobe edges in some time-sets while enabling strobe edges in other time-sets. Operation will NOT be as desired, with symptoms which are extremely difficult to diagnose.

Usage

The various versions of `settime()` below each have a different number of edge time arguments. The comments identify which version is used to program a given timing format.

Note: Double Data Rate (DDR) Mode uses additional versions of `settime()` which are documented in DDR Timing.

The following functions program timing for a single pin:

```
void settime(  TSETNumber TSET,
               DutPin *pDutPin,
               TGFormat Format,
               double Edge1 );     // NRZ, STROBE*
void settime(  TSETNumber TSET,
               DutPin *pDutPin,
               TGFormat Format,
               double Edge1,
               double Edge2 );     // RTZ, RTO, RTC, STROBE*
                                   // IODRIVE, IOSTROBE
void settime(  TSETNumber TSET,
               DutPin *pDutPin,
               TGFormat Format,
               double Edge1,
               double Edge2,
               double Edge3,
               double Edge4 );        // DCLKPOS, DCLKNEG
```

The following functions program timing for one or more pin(s):

```
void settime(  TSETNumber TSET,
               PinList* pPinList,
               TGFormat Format,
               double Edge1 );      // NRZ, STROBE*
void settime(  TSETNumber TSET,
               PinList* pPinList,
               TGFormat Format,
               double Edge1,
               double Edge2 );      // RTZ, RTO, RTC, STROBE*
                                    // IODRIVE, IOSTROBE
void settime(  TSETNumber TSET,
               PinList* pPinList,
               TGFormat Format,
               double Edge1,
               double Edge2,
               double Edge3,
               double Edge4 );        // DCLKPOS, DCLKNEG
```

Note: ***** there is only one STROBE format, which is specified when programming both window and edge strobes. The strobe mode (window vs. edge) is programmable per-pin-pair using the edge_strobe() function. Programming a STROBE format can be done using one edge time value or two edge time values. When the second edge time value is omitted the system software automatically programs Edge2 to obtain the minimum strobe pulse width.

where:

TSET is the time-set being programmed. Legal values are of the TSETNumber enumerated type.

pDutPin identifies a single pin to program. In Multi-DUT Test Programs, the same pin of all DUT(s) in the Active DUTs Set (ADS) are affected. Only signal pins are legal.

pPinList identifies one or more pin(s) to be programmed. In Multi-DUT Test Programs the pin(s) of all DUT(s) in the Active DUTs Set (ADS) are affected. Only signal pins are legal.

Format identifies the timing format being programmed. Legal values are of the TGFormat enumerated type, as follows:

- Drive formats: NRZ, RTZ, RTO, RTC, DCLKPOS, DCLKNEG
- Strobe format: STROBE

- Drive I/O format: `IODRIVE`
- Tri-state I/O format: `IOSTROBE`
- Note: `DDR_DRIVE`, `DDR_STROBE`, `DDR_IODRIVE`, `DDR_IOSTROBE` are documented in DDR Timing.
- Note: using Magnum the SBC format is not available.

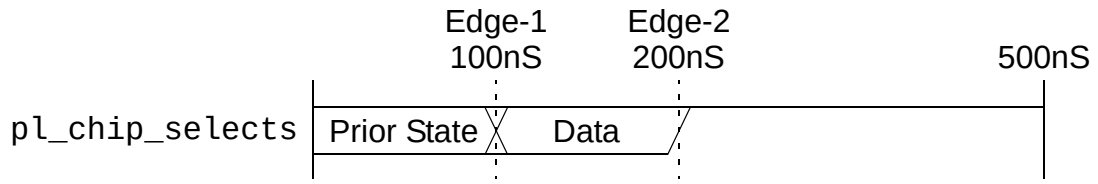
`Edge1`, **`Edge2`**, **`Edge3`**, and **`Edge4`** are used to specify the desired edge time for each edge of the specified **`Format`**. Units may be used (see Specifying Units). Edge2 is not used with edge strobes; any value programmed is ignored.

Examples

Example 1:

The following example programs an return-to-one (`RTO`) drive format on all pins in the `pl_chip_selects` pin list. The pin-pair of each pin in `pl_chip_selects` is also affected (see Functional Pin-pairs):

```
cycle ( TSET1, 500 NS );
settime( TSET1, pl_chip_selects, RTO, 100 NS, 200 NS );
```

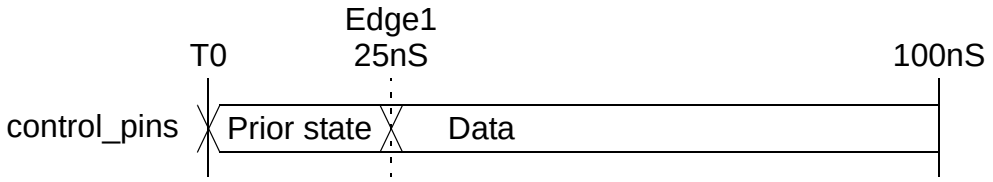


This waveform drives high or low based on the test pattern at 100nS, and drives high at 200nS. If the pins in the `pl_chip_selects` pin list also need a strobe format and/or I/O timing the `settime()` function must be called again to set up these additional formats.

Example 2:

The following example programs a `NRZ` drive format on all pins in the `control_pins` pin list, in time-set `TSET2`. Note that only one edge time parameter is used to program the `NRZ` format. The pin-pair of each pin in `control_pins` is also affected (see Functional Pin-pairs)

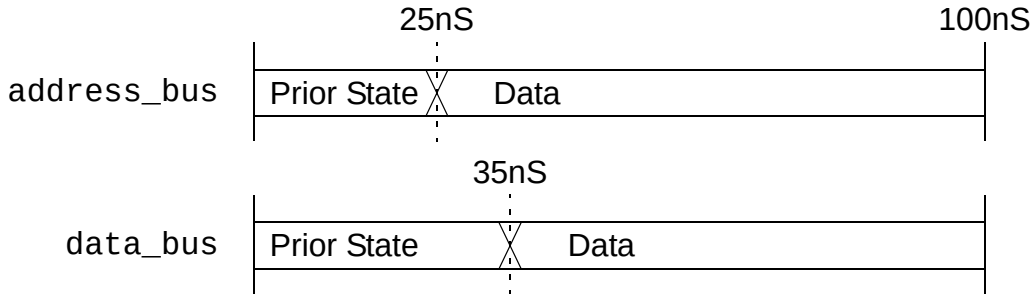
```
cycle ( TSET2, 100 NS );
settime( TSET2, control_pins, NRZ, 25 NS );.
```



Example 3:

The following example demonstrates setting two different NRZ timings on two different sets of pins. One pin list is called address_bus and the other pin list is called data_bus:

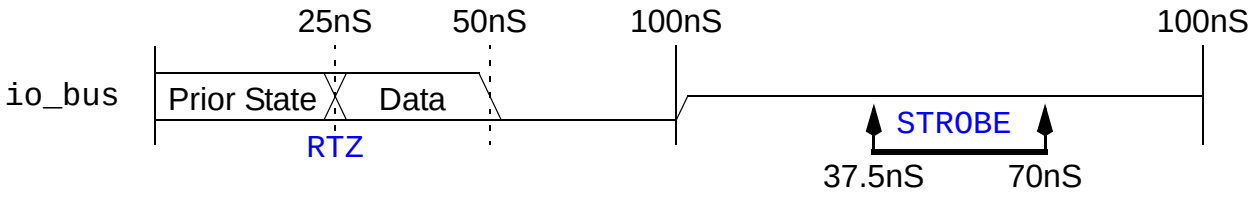
```
cycle( TSET5, 100 NS );
settime( TSET5, address_bus, NRZ, 25 NS );
settime( TSET5, data_bus, NRZ, 35 NS );
```



Example 4:

The following example programs both a drive format and strobe timing for all pins (and their pairs) in the pin list named io_bus, for TSET3. The waveforms for this example are shown below. The left waveform is the drive format, the right waveform represents the strobe timing. Since no I/O timing is programmed it will occur at 0nS:

```
cycle ( TSET3, 100 NS );
settime(TSET3, io_bus, RTZ, 25 NS, 50 NS);
settime(TSET3, io_bus, STROBE, 37.5 NS, 70 NS);
```



3.18.8.2 setedge(), getedge()

See Timing and Formatting Functions, Magnum Timing Rules.

Description

The `setedge()` function is used to set a single edge timing value.

The `getedge()` function are used to get a single edge timing value.

`getedge()` can *get* either *actual* timing values or *programmed* timing values. This supports user-written code to display the effect of system clock and vernier on programmed cycle periods and edge times. The enumerated type `TimeOption` (see Usage) is used to specify whether actual or programmed timing values are to be returned by `getedge()`.

Note: remember that drive, strobe, and I/O timing formats are always set up for every pin in every time-set. Default formats and edge times are set by system software when not explicitly programmed by user code.

Note: each timing channel serves two pins i.e. a pin-pair. See Functional Pin-pairs. The software is programmed per-pin, but both pins of a pin-pair are affected when either pin is programmed.

Usage

The following functions set the edge time for the specified edge of a specified format in a specified time-set for one pin:

```
BOOL setedge(TSETNumber TSET,
             DutPin *pDutPin,
             EdgeTypes EdgeType,
             int EdgeNumber,
             double EdgeTime );
BOOL setedge(TSETNumber TSET,
             DutPin *pDutPin,
             EdgeTypes EdgeType,
```

```
                int EdgeNumber,
                double EdgeTime,
                BOOL DualData );
```

The following functions set the edge time for the specified edge of a specified format in a specified time-set for all pins in the specified pin list:

```
BOOL setedge(TSETNumber TSET,
             PinList* pPinList,
             EdgeTypes EdgeType,
             int EdgeNumber,
             double EdgeTime );

BOOL setedge(TSETNumber TSET,
             PinList* pPinList,
             EdgeTypes EdgeType,
             int EdgeNumber,
             double EdgeTime,
             BOOL DualData );
```

The following functions get the actual edge time for the specified edge in a specified time-set for one specified pin:

```
double getedge(TSETNumber TSET,
               DutPin *pDutPin,
               EdgeTypes EdgeType,
               int EdgeNumber );

double getedge(TSETNumber TSET,
               DutPin *pDutPin,
               EdgeTypes EdgeType,
               int EdgeNumber,
               BOOL DualData );
```

The following functions get the programmed or actual edge time for the specified edge in a specified time-set for one specified pin:

```
double getedge(TSETNumber TSET,
               DutPin *pDutPin,
               EdgeTypes EdgeType,
               int EdgeNumber,
               TimeOption TimeValueType );

double getedge(TSETNumber TSET,
               DutPin *pDutPin,
               EdgeTypes EdgeType,
```

```
                int EdgeNumber,
                TimeOption TimeValueType,
                BOOL DualData );
```

where:

TSET is the time-set being accessed. Legal values are of the TSETNumber enumerated type.

pDutPin is used in 2 contexts:

- In the setter functions, **pDutPin** identifies one pin to be programmed. In Multi-DUT Test Programs, the pin of all DUT(s) in the Active DUTs Set (ADS) are affected.
- In the getter functions, **pDutPin** identifies one pin to be read. In Multi-DUT Test Programs, the value is read from the first DUT in the Active DUTs Set (ADS):

EdgeType specifies whether drive edges, strobe edges, or I/O edges are being accessed. Legal values are of the EdgeTypes enumerated type:

EdgeNumber identifies which edge is being accessed. Note that getedge() returns default timing values when user programmed values have not been set for a given edge. The table below shows valid edge numbers for each format type:

Table 3.18.8.2-1 Used Edge Numbers vs. Timing Formats

Format	EdgeTypes	DualData	Edge Number				Comments
			1	2	3	4	
NRZ	t_drive_edges	FALSE	X	n/a	n/a	n/a	
RTO	t_drive_edges	FALSE	X	X	n/a	n/a	
RTZ	t_drive_edges	FALSE	X	X	n/a	n/a	
DCLKPOS	t_drive_edges	TRUE or FALSE[2]	X	X	X	X	DDR and non-DDR[2]
DCLKNEG	t_drive_edges	TRUE or FALSE[2]	X	X	X	X	DDR and non-DDR[2]
STROBE	t_strobe_edges	FALSE	X	X	n/a	n/a	Window Strobe
STROBE	t_strobe_edges	FALSE	X	X[1]	n/a	n/a	Edge Strobe[1]
IODRIVE	t_IO_drive_edges	FALSE	X	X	n/a	n/a	

Table 3.18.8.2-1 Used Edge Numbers vs. Timing Formats

Format	EdgeTypes	DualData	Edge Number				Comments
			1	2	3	4	
IOSTROBE	t_IO_strobe_edges	FALSE	X	X	n/a	n/a	
DDR_DRIVE	t_drive_edges	TRUE	X	X	n/a	n/a	
DDR_STROBE	t_strobe_edges	TRUE	X	X[1]	X	X[1]	Edge Strobe[1]
DDR_IODRIVE	t_IO_drive_edges	TRUE	X	X	n/a	n/a	
DDR_IOSTROBE	t_IO_strobe_edges	TRUE	X	X	n/a	n/a	

Notes
1) When using `setedge()` to modify an edge strobe, Edge2 is not usedand any value programmed is ignored.
2) `DCLKPOS` and `DCLKNEG` are usable in both DDR and non-DDR mode.

`EdgeTime` specifies the desired edge timing. Units may be used (see Specifying Units). The value -1 can be used to disable an edge (but not Strobe edges).

`DualData` is optional, and must be TRUE when accessing a timing format which is configured for Double Data Rate (DDR) Mode operation. Default = FALSE = non-DDR.

`pPinList` identifies the pin(s) to be programmed. Only signal pins are valid. In Multi-DUT Test Programs, only pin(s) of DUT(s) in the Active DUTs Set (ADS) are affected.

`TimeValueType` specifies whether the programmed value or actual value is returned by `getedge()`. Legal values are of the `TimeOption` enumerated type. If not specified the actual value is returned.

`setedge()` returns FALSE if an invalid **`EdgeNumber`** is specified for the format currently set for any pins in the specified **`pPinList`**.

`getedge()` returns the requested timing value, or -1 for edges which are intentionally disabled. The value -1 is also returned when an invalid **`EdgeNumber`** is specified for the format currently set for the **`Pin`**. In Multi-DUT Test Programs, the value is read from the first DUT in the Active DUTs Set (ADS):

Example

Example 1:

The following example adds 1nS to the edge time of the 2nd edge (the drive data edge) of an expected RTO drive format on all pins in the pin list pl_databus. Several error checks are made to ensure the RTO format is already set and that legal edge_number values are used.

```
// Get the TesterPin for the first pin in pl_databus
DutPin pin;
if (pin_info( pl_databus, 0, &pin ) == FALSE)
   output("ERROR: invalid pin list) passed to pin_info()");

// Get drive format currently set for the first pin in pl_databus
TGFormat format = getformat (TSET1, pin ); // getformat()
if (! (format == RTO) )
   output("ERROR: expected getformat() to return RTO format");

// Get time programmed for the drive-true-data RTO timing edge
double val;
if ( val = getedge( TSET1, pin, t_drive_edges, 1 ) == -1)
   output("Edge is disabled, or invalid edge number specified");

// Set this same edge to the value read in previous code + 1nS
if( setedge( TSET1,
             pl_databus,
             t_drive_edges,
             1,
             (val + 1 NS) ) == -1)
   output("ERROR: invalid edge_number passed to setedge()");
```

Note the following:

- The drive-data edge of the RTO format is the first edge (Edge-1).
- Edge-1, as used above, is valid for all formats, however the error messages are included to show usage.
- The code above assumes that the drive format for all pins in pl_databus is the same as that of the first pin; i.e. the error checks are simplistic.

The code above assumes that if getedge() returns -1 an error exists. Remember that -1 can also be returned when an edge is intentionally disabled, and in that situation -1 is not an error.

3.18.8.3 Per-edge Functions: Drive/Strobe

See Timing and Formatting Functions, Programming Timing & Formats.

Description

Note: in addition to the functions documented below, the `setedge(), getedge()` functions may also used to set and get a single edge timing value. They were added instead of adding new arguments to the functions below, which would make them irregular and cumbersome. Use `setedge(), getedge()` instead.

The functions below are used to set or get the timing value for one edge of a drive format or strobe format. See Per-edge Functions: I/O Edges for similar functions which operate on I/O formats.

Several set functions and several get function are provided, supporting various argument options. Each function accesses a specific edge number. The user must understand which edges are appropriate to each drive or strobe format. See Supported Timing Formats.

Note: when programming edge strobes Edge2 is not used, any values programmed are ignored.

Note: each timing channel serves two pins i.e. a pin-pair. See Functional Pin-pairs. The software is programmed per-pin, but both pins of a pin-pair are affected when either pin is programmed.

Usage

The following functions are used to set the timing value of one edge, on one specified pin:

```
void setedge1( TSETNumber TSET,
               DutPin *pDutPin,
               BOOL Drive,
               double Edge1 );
```

```
void setedge2( TSETNumber TSET,
               DutPin *pDutPin,
               BOOL Drive,
               double Edge2 );
void setedge3( TSETNumber TSET,
               DutPin *pDutPin,
               BOOL Drive,
               double Edge3 );
void setedge4( TSETNumber TSET,
               DutPin *pDutPin,
               BOOL Drive,
               double Edge4 );
void setedge1( TSETNumber TSET,
               DutPin *pDutPin,
               BOOL Drive,
               double Edge1,
               BOOL DualData );
void setedge2( TSETNumber TSET,
               DutPin *pDutPin,
               BOOL Drive,
               double Edge2,
               BOOL DualData );
void setedge3( TSETNumber TSET,
               DutPin *pDutPin,
               BOOL Drive,
               double Edge3,
               BOOL DualData );
void setedge4( TSETNumber TSET,
               DutPin *pDutPin,
               BOOL Drive,
               double Edge4,
               BOOL DualData );
```

The following functions are used to set the timing value of one edge, on all pins in the specified pin list:

```
void setedge1( TSETNumber TSET,
               PinList* pPinList,
               BOOL Drive,
               double Edge1 );
void setedge2( TSETNumber TSET,
               PinList* pPinList,
               BOOL Drive,
               double Edge2 );
void setedge3( TSETNumber TSET,
               PinList* pPinList,
               BOOL Drive,
               double Edge3 );
void setedge4( TSETNumber TSET,
               PinList* pPinList,
               BOOL Drive,
               double Edge4 );
void setedge1( TSETNumber TSET,
               PinList* pPinList,
               BOOL Drive,
               double Edge1,
               BOOL DualData );
void setedge2( TSETNumber TSET,
               PinList* pPinList,
               BOOL Drive,
               double Edge2,
               BOOL DualData );
void setedge3( TSETNumber TSET,
               PinList* pPinList,
               BOOL Drive,
               double Edge3,
               BOOL DualData );
void setedge4( TSETNumber TSET,
               PinList* pPinList,
               BOOL Drive,
               double Edge4,
               BOOL DualData );
```

The following functions are used to get the currently programmed timing value of a specific edge, from a specific pin:

```
double getedge1( TSETNumber TSET, DutPin *pDutPin, BOOL Drive );
double getedge2( TSETNumber TSET, DutPin *pDutPin, BOOL Drive );
double getedge3( TSETNumber TSET, DutPin *pDutPin, BOOL Drive );
double getedge4( TSETNumber TSET, DutPin *pDutPin, BOOL Drive );
double getedge1(  TSETNumber TSET,
                  DutPin *pDutPin,
                  BOOL Drive,
                  BOOL DualData );
double getedge2(  TSETNumber TSET,
                  DutPin *pDutPin,
                  BOOL Drive,
                  BOOL DualData );
double getedge3(  TSETNumber TSET,
                  DutPin *pDutPin,
                  BOOL Drive,
                  BOOL DualData );
double getedge4(  TSETNumber TSET,
                  DutPin *pDutPin,
                  BOOL Drive,
                  BOOL DualData );
```

where:

TSET is the time-set being accessed. Legal values are of the `TSETNumber` enumerated type.

pDutPin is used in 2 contexts:

- In the setter functions, **pDutPin** identifies one pin to be programmed. In Multi-DUT Test Programs, the pin of all DUT(s) in the Active DUTs Set (ADS) are affected.
- In the getter functions, **pDutPin** identifies one pin to be read. In Multi-DUT Test Programs, the value is read from the first DUT in the Active DUTs Set (ADS):

Drive is used to specify whether the drive format (TRUE) or the strobe format (FALSE) is to be accessed. Remember, drive, strobe and I/O formats are always programmed in every time-set.

Edge1, **Edge2**, **Edge3**, and **Edge4** are used to specify the desired edge time for a specific edge. Units may be used (see Specifying Units). The user must understand which edges are appropriate to each drive or strobe format. See Supported Timing Formats

DualData is optional, and must be TRUE when accessing a timing format which is configured for Double Data Rate (DDR) Mode operation. Default = FALSE = non-DDR.

pPinList identifies the pin(s) to be programmed. Only signal pins are valid. In Multi-DUT Test Programs, only pin(s) of DUT(s) in the Active DUTs Set (ADS) are affected.

`getedge1()`, `getedge2()`, `getedge3()`, and `getedge4()` return the specified edge time. In Multi-DUT Test Programs, the value is read from the first DUT in the Active DUTs Set (ADS).

Example

The example below uses `settime()` to program an `RTZ` drive format, in TSET3, on all pins in the pin list named *Abus*. Then, the timing value of Edge2 of this drive format is read-back, printed, modified and used to reprogram Edge2, read-back again, and printed again to see the effect. For this example to operate as desired requires that the single pin value (A9), passed to `getedge2()`, must be one of the pins in the pin list named *Abus*.

```
settime ( TSET3, Abus, RTZ, 10 NS, 30 NS );
double e2 = getedge2( TSET3, A9, TRUE );     // Read-back one pin
output (" Original value = %5f pS", e2 );
setedge2( TSET3, Abus, TRUE, (e2 + 5 NS) );  // Modify edge2
e2 = getedge2( TSET3, A9, TRUE );            // Read-back again
output (" New value = %5f pS", e2 );
```

3.18.8.4 Per-edge Functions: I/O Edges

See Timing and Formatting Functions, I/O Timing and Control.

Description

Note: in addition to the functions documented below, the `setedge(), getedge()` functions may also used to set and get a single edge timing value. They were added instead of adding new arguments to the functions below, which would make them irregular and cumbersome. Use `setedge(), getedge()` instead.

The functions below are used to set or get a timing value for a specific I/O timing edge; i.e. `IODRIVE` and `IOSTROBE` edge timing. See Per-edge Functions: Drive/Strobe for similar functions which operate on drive and strobe formats.

Two set functions and two get function are provided. Each function accesses a specific edge number. The user must understand which edges are appropriate to each I/O. See Supported Timing Formats.

Usage

The following functions are used to set the timing value of a specific I/O timing edge, on a specified pin. See IODRIVE and IOSTROBE.

```
void setioedge1(TSETNumber TSET,
                DutPin *pDutPin,
                BOOL Drive,
                double Edge1 );
void setioedge2(TSETNumber TSET,
                DutPin *pDutPin,
                BOOL Drive,
                double Edge2 );
void setioedge1(TSETNumber TSET,
                DutPin *pDutPin,
                BOOL Drive,
                double Edge1,
                BOOL DualData );
void setioedge2(TSETNumber TSET,
                DutPin *pDutPin,
                BOOL Drive,
                double Edge2,
                BOOL DualData );
```

The following functions are used to set the timing value of a specific I/O timing edge, on all pins in the specified pin list. See IODRIVE and IOSTROBE.

```
void setioedge1(  TSETNumber TSET,
                  PinList* pPinList,
                  BOOL Drive,
                  double Edge1 );
void setioedge2(  TSETNumber TSET,
                  PinList* pPinList,
                  BOOL Drive,
                  double Edge2 );
```

```
void setioedge1(  TSETNumber TSET,
                  PinList* pPinList,
                  BOOL Drive,
                  double Edge1,
                  BOOL DualData );
void setioedge2(  TSETNumber TSET,
                  PinList* pPinList,
                  BOOL Drive,
                  double Edge2,
                  BOOL DualData );
```

The following functions are used to get the current timing value of a specific I/O timing edge, from a specific pin:

```
double getioedge1( TSETNumber TSET, DutPin *pDutPin, BOOL Drive );
double getioedge2( TSETNumber TSET, DutPin *pDutPin, BOOL Drive );
double getioedge1(TSETNumber TSET,
                  DutPin *pDutPin,
                  BOOL Drive,
                  BOOL DualData );
double getioedge2(TSETNumber TSET,
                  DutPin *pDutPin,
                  BOOL Drive,
                  BOOL DualData );
```

where:

TSET is the time-set being accessed. Legal values are of the `TSETNumber` enumerated type.

pDutPin is used in 2 contexts:

- In the setter functions, **pDutPin** identifies one pin to be programmed. In Multi-DUT Test Programs, the pin of all DUT(s) in the Active DUTs Set (ADS) are affected.
- In the getter functions, **pDutPin** identifies one pin to be read. In Multi-DUT Test Programs, the value is read from the first DUT in the Active DUTs Set (ADS):

Drive is used to specify whether the drive I/O format (TRUE = `IODRIVE`) or the tri-state I/O format (FALSE = `IOSTROBE`) is to be accessed. Remember, drive, strobe and I/O formats are always programmed in all time-sets.

Edge1 and **Edge2**are used to specify the desired edge time for a specific edge. Units may be used (see Specifying Units). The user must understand which edges are appropriate to each I/O format. See Supported Timing Formats.

DualData is optional, and must be TRUE when accessing a timing format which is configured for Double Data Rate (DDR) Mode operation. Default = FALSE = non-DDR.

pPinList identifies the pin(s) to be programmed. Only signal pins are valid. In Multi-DUT Test Programs, only pin(s) of DUT(s) in the Active DUTs Set (ADS) are affected.

`getioedge1()` and `getioedge2()` return the specified edge time. In Multi-DUT Test Programs, the value is read from the first DUT in the Active DUTs Set (ADS):

Example

The example below uses `settime()` to program an `IOSTROBE` I/O format, in `TSET8`, on all pins in the list named *IOpins*. Then, the timing value of Edge1 of this drive format is read-back, printed, modified and used to reprogram Edge1, read-back again, and printed again to see the effect. For this example to operate as desired requires that the single DUT pin (`D4`), passed to `getedge1()`, must be one of the pins in the pin list named *IOpins*.

```
settime ( TSET8, IOpins, IOSTROBE, 10 NS, 30 NS );
double e1 = getioedge1( TSET8, D4, FALSE ); // Read-back pin D4
output (" Original value = %5f pS", e1 );
setioedge1( TSET8, IOpins, FALSE, (e1 + 2 NS) ); // Modify edge1
e1 = getioedge1( TSET8, D4, FALSE );            // Read-back again
output (" New value = %5f pS", e1 );
```

3.18.8.5 getformat()

See Timing and Formatting Functions.

Description

The `getformat()` function is used to get the currently programmed drive format of one specified pin in a specified time-set.

Default drive, strobe, and I/O timing formats are automatically set up for every pin in every time-set. And, the strobe and I/O edges have fixed formats; the test program can program timing values for the strobe and I/O format edges but has no choice in format selection.

However, there are several drive formats, which can be specified on a per-pin and per-time-set basis: NRZ, RTZ, RTO, RTC, DCLKPOS, DCLKNEG. The getformat() function can be used to determine the drive format programmed for a given pin in a given time-set.

Note: using Memory Test Patterns the format applied to Chip Selects is controlled from the APG using the CHIPS pattern instruction options. See APG Chip Select Drive Format Selection.

Usage

```
TGFormat getformat( TSETNumber TSET, DutPin *pDutPin );
TGFormat getformat( TSETNumber TSET,
                    DutPin *pDutPin,
                    BOOL DualData );
```

where:

TSET is the time-set being accessed. Legal values are of the TSETNumber enumerated type.

pDutPin identifies one pin to be read. In Multi-DUT Test Programs, the value is read from the first DUT in the Active DUTs Set (ADS).

DualData is optional, and must be TRUE when accessing a timing format which is configured for Double Data Rate (DDR) Mode operation. Default = FALSE = non-DDR.

getformat() returns the drive format for the specified **Pin**. The returned value will be of the TGFormat enumerated type. In Multi-DUT Test Programs, the value is read from the first DUT in the Active DUTs Set (ADS).

Example

The following example uses settime() to program an RTO drive format, in TSET4, on all pins in the list named *IOpins*. Then, the drive format of one pin is read back, and printed. For this example to operate as desired requires that the single DUT pin value (A19), passed to getformat(), be one of the pins in the pin list named *IOpins*.

```
settime ( TSET4, IOpins, RTO, 10 NS, 30 NS );
TGFormat df = getformat( TSET4, A19 );
output (" Drive format on pin A19 =>\\");
switch (df ) {
   case NRZ     : output (" NRZ");       break;
   case RTZ     : output (" RTZ");       break;
```

```
    case RTO     : output (" RTO");       break;
    case RTC     : output (" RTZ");       break;
    case DCLKPOS : output (" DCLKPOS ");  break;
    case DCLKNEG : output (" DCLKNEG");   break;
    default : output (" UNKNOWN = error");
}
```

3.18.9 Timing Examples

See Timing and Formatting Functions, Magnum Timing Rules.

Note that the resolution of the following calculations below often exceeds that which is available using Magnum hardware; the extra digits were included here to reduce confusion caused by rounding.

Example 1:

```
cycle ( TSET1, 30 NS );
settime ( TSET1, pins, RTZ, 2 NS, 29 NS );
cycle ( TSET2, 30 NS );
settime ( TSET2, pins, RTZ, 12 NS, 19 NS );
```

Note the following:

- Both cycle periods will be as-programmed since both are specified in multiples of 1nS.
- Both drive format verniers are set by edge times programmed in TSET2. However, since all edge types are specified in increments of 1nS all of the edges noted will will be as-programmed.

3.19 MUX, Super-MUX and DDR

See Software.

This chapter documents methods which can be used to increase the maximum effective data rate and/or reduce the effective minimum cycle period of the test system. Some of this information is identical for Maverick-I/-II and Magnum 1/2/2x. However, key differences exist, pay attention.

Terminology:

Term	Description
DDR	Double Data Rate (DDR) Mode. Read Overview. Not usable on Maverick-I.
MUX	Two pins multiplexed (MUX), one can't be used at the DUT. See MUX Mode. Read Overview first.
Super-MUX	Magnum 2/2x only. Two pins multiplexed (MUX), all can be used at the DUT. See Super-MUX Mode. Read Overview first.

The following topics are covered in this section:

- MUX Mode
- Super-MUX Mode
- ECR in DDR, MUX and Super-MUX Modes
- MUX, Super-MUX & DDR Software
  - Types, Enums, etc.
  - `mux_mode_set(), mux_mode_get()`
  - `mux_mode(), mux_mode_disable()`
  - `fail_signal_mux()`

3.19.1 Overview

See MUX, Super-MUX and DDR.

The maximum data rate of a digital test system type is a key system specification. The maximum data rate for the Magnum 1I is:

System Type	Max. Data Rate	Min. Cycle Period
Magnum 1	50MHz	20nS

In this document, the term *data rate* refers to how fast the Algorithmic Pattern Generator (APG) and/or Logic Vector Memory (LVM) and/or Scan Vector Memory (SVM) can deliver new drive/expect data and I/O control signals to the Timing & Formatting system. This is the truth-table data used to functionally test the part.

It is possible to increase the effective maximum data rate and/or reduce the effective minimum cycle period using several methods, described below. Each method has corresponding capabilities and limitations. These are described as:

- Single Data Rate Mode (SDR), the default system operation, for reference.
- Double Data Rate (DDR) Mode
- MUX Mode, also commonly used to double the maximum clock frequency on selected pins.
- Super-MUX Mode, also commonly used to double the maximum clock frequency on selected pins.

The following table indicates which modes are supported on each Nextest system type. Single Data Rate Mode (SDR) is not mentioned because it is the default system operating mode:

Table 3.19.1.0-1 Supported Modes vs. System Types

System Type	Supported Options
Maverick-I	MUX Mode
Maverick-II	Double Data Rate (DDR) Mode MUX Mode
Magnum 1	Double Data Rate (DDR) Mode MUX Mode
Magnum 2 Magnum 2x	Double Data Rate (DDR) Mode MUX Mode Super-MUX Mode
Using Magnum 2/2x it is also possible to mix Double Data Rate (DDR) Mode with pins in MUX Mode or Super-MUX Mode (but not both).	

The following table provides an overview of features which are different using the various combinations of Single Data Rate Mode (SDR), Double Data Rate (DDR) Mode, MUX Mode and Super-MUX Modes. Each mode is then described in detail after the table:

Table 3.19.1.0-2 Single/DDR/MUX/Super-MUX Mode Trade-offs

Feature	Usable?	Comment
Single Data Rate Mode (SDR) 2X data rate? 4X data rate? NRZ and Edge Strobe Only? Lose use of adjacent pin? VOL must = VOH (MUX pins)? Global mode (all pins)? ECR: all pins can be logged?	 No No No No No Yes Yes	SDR is in effect only during the execution of an SDR (i.e. non-DDR) test pattern and affects all pins identically. Executing an SDR test pattern, one set of pattern data is supplied to each timing channel in each tester cycle. Select pins can also be in MUX Mode or Super-MUX Mode during SDR pattern execution.

Table 3.19.1.0-2 Single/DDR/MUX/Super-MUX Mode Trade-offs *(Continued)*

Feature	Usable?	Comment
Double Data Rate (DDR) Mode 2X data rate? 4X data rate? NRZ and Edge Strobe Only? Lose use of adjacent pin? VOL must = VOH (MUX pins)? Global mode (all pins)? ECR: all pins can be logged?	 Yes No Yes No No Yes No	DDR mode is in effect only during the execution of DDR Test Patterns and affects all pins identically. When executing DDR Test Patterns, two sets of pattern data are supplied to each timing channel in each tester cycle. Select pins can also be in MUX Mode or Super-MUX Mode during DDR pattern execution. When executing DDR Test Patterns, not all pins can be logged to the ECR, see ECR in DDR, MUX and Super-MUX Modes.
MUX Mode 2X data rate? 4X data rate? NRZ and Edge Strobe Only? Lose use of adjacent pin? VOL must = VOH (MUX pins)? Global mode (all pins)? ECR: all pins can be logged?	 Yes No No Yes Yes No[1] Yes	Pins in MUX Mode receive two sets of pattern data in each tester cycle, supplied via two timing channels. MUX Mode is independently set per-pin. For each odd pin in MUX Mode the next higher even pin is not usable at the DUT. Both pins of a given pin-pair (a_1/b_1) will always be in the same MUX mode. Pins can be in MUX Mode when executing DDR Test Patterns. All MUX pins can be logged to the ECR but DDR mode rules can change this, see ECR in DDR, MUX and Super-MUX Modes.
Super-MUX Mode 2X data rate? 4X data rate? NRZ and Edge Strobe Only? Lose use of adjacent pin? VOL must = VOH (MUX pins)? Global mode (all pins)? ECR: all pins can be logged?	 Yes No No No Yes No[1] No	Magnum 2/2x only.24

Table 3.19.1.0-2 Single/DDR/MUX/Super-MUX Mode Trade-offs *(Continued)*

Feature	Usable?	Comment
DDR + MUX Mode 2X data rate? 4X data rate? NRZ and Edge Strobe Only? Lose use of adjacent pin? VOL must = VOH (MUX pins)? Global mode (all pins)? ECR: all pins can be logged?	 No Yes Yes Yes Yes Yes[1,2] No	Pins can be in MUX Mode when executing DDR Test Patterns. These pins will receive 4 sets of pattern data in each tester cycle, supplied via two timing channels. DDR is in effect only during the execution of DDR Test Patterns and affects all pins identically. MUX Mode is independently set per-pin. For each odd pin in MUX Mode the next higher even pin is not usable at the DUT. Both pins of a given pin-pair (a_1/b_1) will always be in the same MUX mode. When executing DDR Test Patterns, not all pins can be logged to the ECR, see ECR in DDR, MUX and Super-MUX Modes.
DDR + Super-MUX Mode 2X data rate? 4X data rate? NRZ and Edge Strobe Only? Lose use of adjacent pin? VOL must = VOH (MUX pins)? Global mode (all pins) ECR: all pins can be logged?	 No Yes Yes No Yes Yes[1,2] No	Magnum 2/2x only.24

3.19.2 Single Data Rate Mode (SDR)

See MUX, Super-MUX and DDR.

Single Data Rate (SDR) mode is the default system operating mode.

The maximum SDR data rate for the Magnum 1is:

System Type	Max. Data Rate	Min. Cycle Period
Magnum 1	50MHz	20nS

Single Data Rate (SDR) mode is in effect only during the execution of an SDR (i.e. non-DDR) test pattern and affects all pins identically. When executing an SDR test pattern one set of pattern data is supplied to each timing channel in each tester cycle.

Pins can be in MUX Mode or Super-MUX Mode during SDR pattern execution but additional rules apply, see MUX Mode and Super-MUX Mode.

During SDR test pattern execution, each of the 64 timing channels on a site assembly has independent timing, format selection and test pattern data, in each tester cycle, with each timing channel driving one pin-pair. All timing formats and strobe types are usable. Using the Error Catch RAM (ECR), each pin is captured to its corresponding bit in the ECR and any pins may be captured.

The following diagram shows 2 pin-pairs in SDR configuration. Each pin-pair is independently driven by 1 timing channel:

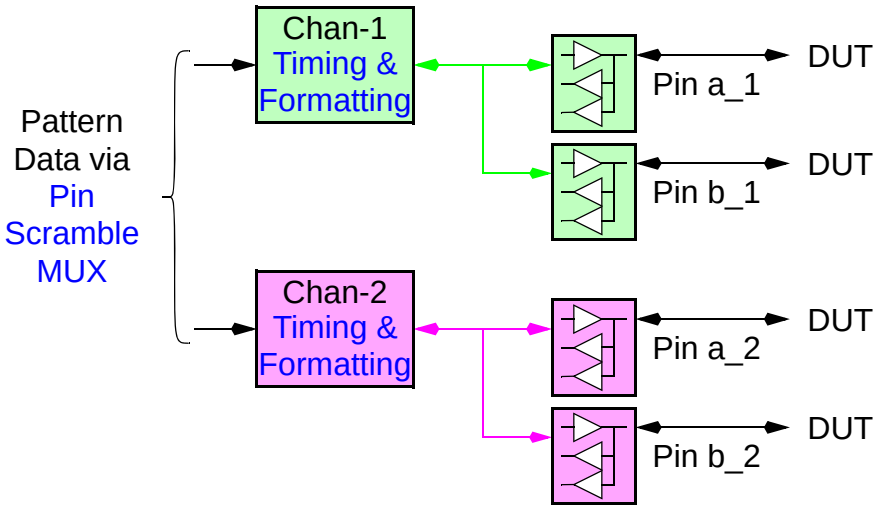


Figure-42: Single Data Rate Block Diagram

3.19.3 Double Data Rate (DDR) Mode

See MUX, Super-MUX and DDR.

Read Overview first.

Note: Double Data Rate (DDR) is not available using Maverick-I.

This section includes the following:

- DDR Overview
- DDR Hardware Details
- DDR Pin Scramble
- DDR Test Patterns
  - DDR Logic Vectors
  - DDR Scan Vectors
  - DDR Memory Patterns
- DDR Timing
  - DDR I/O Timing
- DDR Fail Signal MUX
  - DDR Fail Signal MUX: Logic Error Catch
  - DDR Fail Signal MUX: Memory Error Catch

3.19.3.1 DDR Overview

See Double Data Rate (DDR) Mode, MUX, Super-MUX and DDR.

Note: DDR-related information does not apply to Maverick-I.

Double Data Rate (DDR) Mode, as the name suggests, effectively doubles the maximum data rate of the test system:

System Type	Max SDR Data Rate	Min SDR Cycle Period	Max DDR Data Rate	Min DDR Cycle Period
Magnum 1	50MHz	20nS	100MHz	10nS

DDR mode is targeted primarily at functional testing using Logic Test Patterns and Scan Test Patterns. DDR mode can also be used, with constraints, using Memory Test Patterns.

DDR mode is in effect only during the execution of DDR Test Patterns and affects all pins identically. DDR Test Patterns are explicitly identified using Pattern Rate Attributes in the test pattern source file. See DDR Test Patterns, DDR Logic Vectors, DDR Scan Vectors and DDR Memory Patterns for additional details.

DDR mode does not increase the maximum operating frequency of the test system. Instead, during DDR test pattern execution, two sets of pattern data are supplied, via the DDR Pin Scramble, to each timing channel in each tester cycle. In effect, the DUT sees two DDR cycles in each tester cycle.

Note: for documentation purposes, in DDR mode each tester cycle is often discussed as having two halves, called the *DDR A-cycle* and the *DDR B-cycle*.

The following diagram shows 2 pin-pairs in DDR configuration. Each pin-pair is independently driven by 1 timing channel. More details are included in DDR Hardware

Details:

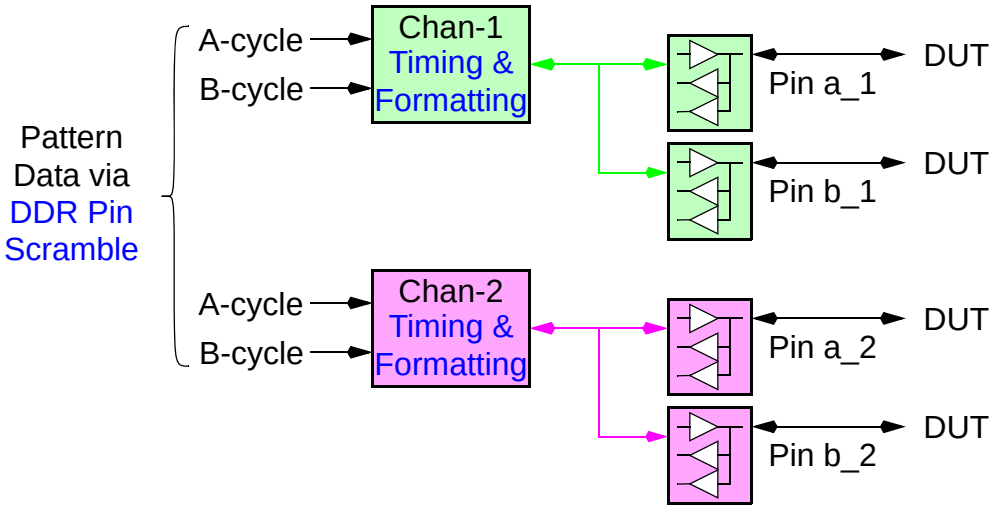


Figure-43: DDR Block Diagram

In preparation for executing DDR Test Patterns, when defining Pin Scramble Tables, two sources of pattern data are specified for each DUT pin in each Pin Scramble Map to be used. See DDR Pin Scramble. Then, during a DDR pattern execution, the Pin Scramble MUX delivers two sets of pattern data to each timing channel, one to control the DDR A-cycle, the other the DDR B-cycle. In effect, two DDR cycles execute within one tester cycle, resulting in DDR operation.

Like Single Data Rate Mode (SDR), DDR mode provides independent timing, format selection and test pattern data for each timing channel on a site assembly, with each channel driving one pin-pair. However, to obtain DDR operation, some of the hardware signals normally used to control timing format selections are instead used to supply pattern data at DDR rates. This means that, in DDR mode, only the `NRZ`, `DCLKPOS` and `DCLKNEG` drive formats can be used and only edge strobes may be used (see `edge_strobe()edge_strobe()`). Additional information and important rules are documented in DDR Timing and DDR I/O Timing.

Pins can be in MUX Mode or Super-MUX Mode (not both) during DDR pattern execution, but additional rules apply, see MUX Mode and Super-MUX Mode.

In DDR mode, the Error Catch RAM (ECR) may be used to capture errors at DDR rates, but this requires additional ECR configuration steps. See ECR in DDR, MUX and Super-MUX Modes.

3.19.3.2 DDR Hardware Details

See Double Data Rate (DDR) Mode, MUX, Super-MUX and DDR.

Note: DDR-related information does not apply to Maverick-I.

The Magnum 1 test system contains two sources of test pattern data:

- Algorithmic Pattern Generator (APG)when executing Memory Test Patterns.
- Combined Logic Vector Memory (LVM) / Scan Vector Memory (SVM) for stored Logic Test Patterns and Scan Test Patterns.

The diagram below shows key Magnum 1 architecture features when using DDR mode:

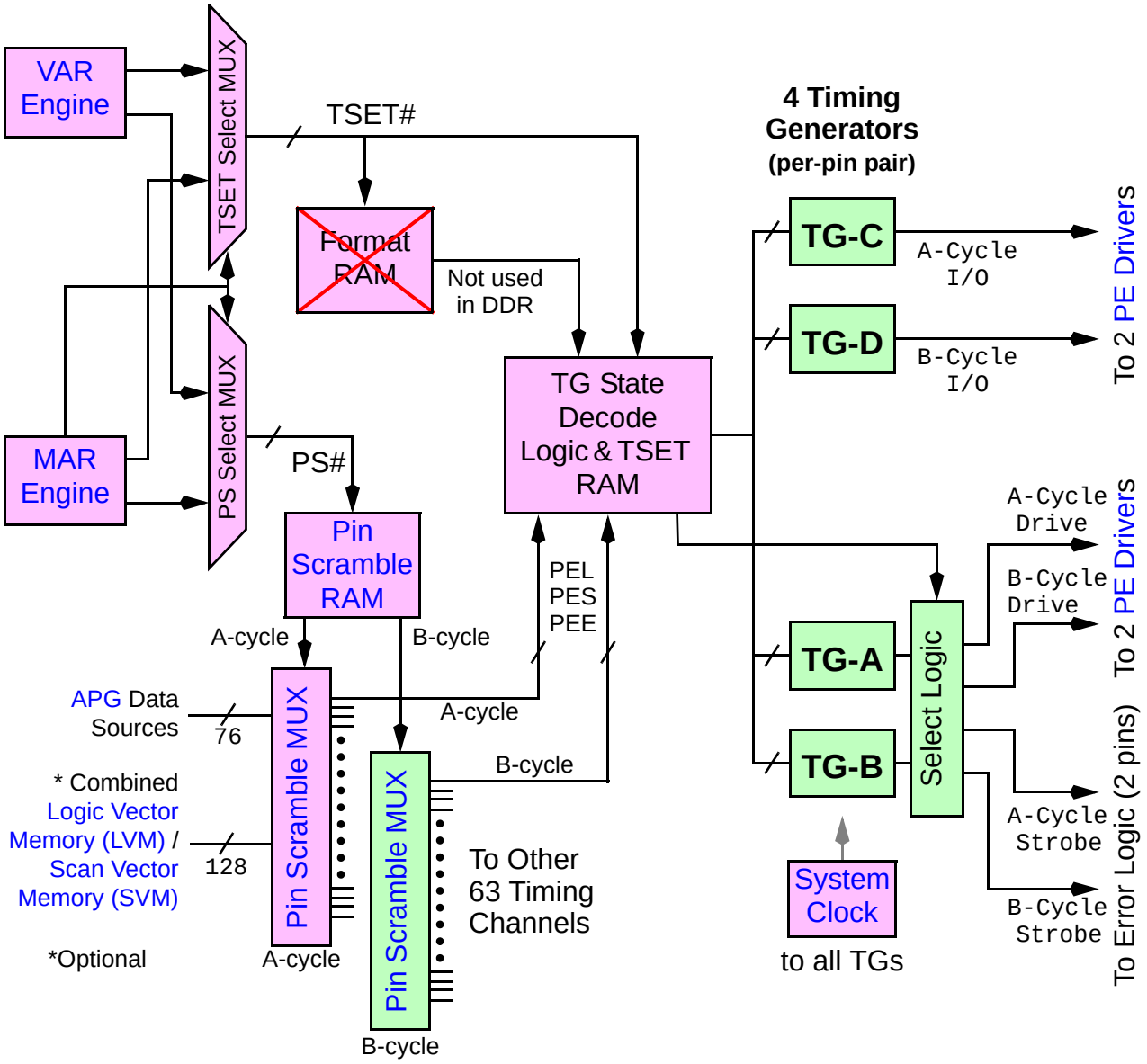


Figure-44: DDR Hardware Architecture

Note the following:

- When executing a Double Data Rate (DDR) Mode test pattern, the combined Logic Vector Memory (LVM) / Scan Vector Memory (SVM) delivers two sets of pattern data to each timing channel (each of which drives a Functional Pin-pair) in each tester cycle (see DDR Logic Vectors and DDR Scan Vectors). Using Memory Test

Patterns patterns, the Algorithmic Pattern Generator (APG) continues to operate at non-DDR rates, delivering one APG Address Generator output, one APG Data Generator output and one APG Chip Selects output in each tester cycle. However, effective DDR operation is often possible, see DDR Memory Patterns. And, the Error Catch RAM (ECR) can be used effectively, see ECR in DDR, MUX and Super-MUX Modes.

- In DDR mode, a second Pin Scramble MUX is used (shown in green in Figure-44:). Only one MUX is used in non-DDR mode, both Pin Scramble MUXes are used in DDR mode. The test pattern inputs to both Pin Scramble MUXes are identical, however, the two MUX control signals, output by the Pin Scramble RAM, may be different.
- When executing both SDR and DDR test patterns, the Pin Scramble MUX selects a source of pattern data (and strobe control and I/O control) for each timing channel, in each tester cycle. In SDR (non-DDR) mode, one source of pattern data will be selected for a given timing channel, at up to maximum data rate of the system, from 81 data sources: any APG address bit, APG data bit , or APG chip select bit, one LVM data bit, and/or Scan Vector Memory data bit. In DDR mode, two sources of pattern data will be selected for a given timing channel in each tester cycle; one selected by the first Pin Scramble MUX (DDR A-cycle data) and one by the second (DDR B-cycle data). In addition, two LVM data bits are available for each timing channel. This means that DDR mode has 82 data sources available, via the Pin Scramble MUX, per-timing channel/per-cycle.
- In DDR mode, the 4 timing generators in each Magnum 1 timing channel are used differently than in non-DDR mode. Timing formats are limited to NRZ and double clock (DCLKPOS, DCLKNEG) drive formats and edge-strobe strobe format. And, I/O edges have specific rules related to DDR operations. See DDR Timing.
- And, I/O edges have specific rules related to DDR operations.

3.19.3.3 DDR Pin Scramble

See Double Data Rate (DDR) Mode, DDR Hardware Details.

Note: DDR-related information does not apply to Maverick-I.

See Pin Scramble Functions & Macros for an overview of non-DDR pin scramble software.

When executing DDR Test Patterns (i.e. in DDR mode), two Pin Scramble MUXes are used, allowing two different test pattern data sources to be selected for each timing channel, in

each tester cycle. This corresponds to the DDR A-cycle data and DDR B-cycle data. For example, a given timing channel might select `t_lvm` in the A-cycle and `t_d9` in the B-cycle of given Pin Scramble Map.

Only the combined Logic Vector Memory (LVM) / Scan Vector Memory (SVM) actually supplies pattern data at DDR rates. The APG outputs continue to operate at non-DDR data rate, however, as noted below, it is possible to obtain useful DDR operation from memory patterns. See DDR Memory Patterns.

Usage

When programming Pin Scramble Maps for DDR use, two data sources are specified, for each pin of each DUT using one of the `SCRAMBLE2_xxxDUT` macros (where *xxx* identifies the number of DUTs being tested in a Multi-DUT Test Program). For example, the following might be used in a program testing 2 DUTs in parallel:

```
PIN_SCRAMBLE( table_name ) {
  SCRAMBLE_MAP( PS1 ) {  // DUT1-A DUT1-B  DUT2-A  DUT2-B
    SCRAMBLE2_2DUT( A1,     t_x0,   t_x18,  t_x0,   t_x18 )
    SCRAMBLE2_2DUT( D0,     t_d0,   t_d0,   t_d0,   t_d8 )
```

In Multi-DUT Test Programs, the data source selected for DUT-1A must be identical to that specified for DUT-2A. DUT-1B must match DUT-2B, etc. Violating this rule causes a fatal runtime error.

Note: Pin Scramble Maps are not inherently constrained to or identified with DDR vs. non-DDR applications. Except as noted here, no compile-time or run-time checks are made to determine whether a given Pin Scramble Map is configured for SDR vs. non-DDR use. As with many Magnum 1/2/2x software constructs, default values are used when user values are not specified.

In situations where a Pin Scramble Map is defined using `SCRAMBLE2_xxxDUT` and subsequently used in non-DDR tests, only the A-cycle data source is used.

3.19.3.4 DDR Test Patterns

See Double Data Rate (DDR) Mode, DDR Hardware Details.

Note: DDR-related information does not apply to Maverick-I.

Note: this section provides an overview of DDR-related test pattern information. Additional details are available in DDR Logic Vectors, DDR Scan Vectors, and DDR Memory Patterns.

Double Data Rate (DDR) Mode is in effect only during the execution of a DDR test pattern and affects all pins identically. A test pattern is identified as a DDR pattern using one of the following methods:

- Using Pattern Rate Attributes in the `PATTERN` instruction in the test pattern source file. This takes precedence over the following method.
- Setting Attribute Defaults as command line options when compiling the test pattern.

DDR test patterns will only execute on Maverick-II and Magnum 1/2/2x (not Maverick-I).

During each `funtest()` execution, the system runtime software detects DDR test patterns and configures the hardware appropriately.

DDR Pattern Rules

The following rules are founded in the fact that the hardware which controls execution of all test patterns always executes at the tester cycle rate, not at DDR rates:

1. Test pattern execution sequence control operations can only occur on tester cycle boundaries. This includes done, conditional jumps, repeats, subroutine calls, returns, interrupts, etc.
2. Each DDR logic vector and scan vector defines two sets of pattern data (and strobe and I/O control) per-timing channel. See DDR Logic Vectors and DDR Logic Vectors.
3. Only one Pin Scramble Map can be selected in each tester cycle. However, for useful DDR test pattern operations the DDR Pin Scramble methods will be used, which allows the Pin Scramble Map selected in each pattern instruction to actually select two pattern data sources for each timing channel. See DDR Pin Scramble.
4. Only one time-set (TSET) and one VIHH Map can be specified in each pattern instruction (each tester cycle).

3.19.3.5 DDR Logic Vectors

See Double Data Rate (DDR) Mode, DDR Hardware Details, DDR Test Patterns.

Note: DDR-related information does not apply to Maverick-I.

In logic DDR Test Patterns, the Logic Vector Memory (LVM) outputs two sets of pattern data to each timing channel, in each tester cycle; i.e. each cycle outputs a DDR A-cycle and DDR B-cycle logic vector to each timing channel. Since the LVM must store twice the amount of pattern data in DDR logic patterns, for each instruction executed the vector address (VAR) is incremented by two (this detail is mostly transparent to the user).

Note: as noted in the DDR Drive Timing & Formats, the Pin Scramble hardware delivers both A-cycle and B-cycle pattern data at the beginning (T0) of each tester cycle. It is the user's programmed edge times which determine at what time, in a given tester cycle, the two pattern data sources are actually used at the DUT.

DDR logic vectors must conform to specific DDR Pattern Rules. However, the syntax of a DDR vector ensures that these rules are difficult to violate. An example DDR Logic Vector is shown below:

```
PATTERN ( ddr_pat_name, double )
% VEC  00001   0101 HHLL ... etc ... \
       00010   1010 XXXX ... etc ...,PS#, TSET#, VIHH#
```

Note the following:

- A DDR pattern is identified to the pattern compiler (Patcom) using the `double` Pattern Rate Attributes in the `PATTERN` declaration.
- A single vector delimiter token (**%**) is used for each DDR pattern instruction.
- A single label can used in each DDR pattern instruction. See Pattern Labels.
- Two sets of logic pattern tokens are specified in each instruction. The first set (the first line above) represents the A-cycle data, the second set (line) represents the B-cycle data. The Test Pattern Line Continuation Character ('\') can be used to display A-cycle data and B-cycle data on two (or more) source lines.
- A single Pin Scramble Map selection can be specified in each DDR vector. See DDR Pin Scramble.
- A single Time-set selection can be specified in each DDR vector. See DDR Timing
- A single VIHH Map selection can be specified in each DDR vector.
- Each DDR logic instruction may only contain one instance of the following:
  - VEC Pattern Instruction or RPT Pattern Instruction

- Optional VEC/RPT Instruction Parameters
- VAR Instruction
- VCOUNT Instruction
- VPINFUNC Instruction
- VUDATA Instruction

- See Logic Error Catch (LEC).

Note: DDR I/O switching (i.e. drive-on vs. drive-off) can only be obtained using DDR Logic Vectors or DDR Scan Vectors. It is not possible to independently control A-cycle vs. B-cycle I/O using memory pattern instructions (`ADHIZ`, etc.).

3.19.3.6 DDR Scan Vectors

See Double Data Rate (DDR) Mode, DDR Hardware Details, DDR Test Patterns.

Note: DDR-related information does not apply to Maverick-I.

Using Magnum 1/2/2x, the Logic Vector Memory (LVM) and Scan Vector Memory (SVM) are the same physical memory. The memory is accessed differently for scan vectors vs. logic vectors, however this does not affect DDR test pattern operation, except as noted in DDR Pattern Rules.

3.19.3.7 DDR Memory Patterns

See Double Data Rate (DDR) Mode, DDR Hardware Details, DDR Test Patterns.

Note: DDR-related information does not apply to Maverick-I.

The Algorithmic Pattern Generator (APG), used to generate Memory Test Patterns, always operates at the tester cycle rate, even when executing DDR Test Patterns. However, as noted below, it is possible, with constraints, to obtain useful DDR memory pattern operations from DDR Test Patterns.

Fundamentally, all APG outputs and control functions change once per tester cycle, including:

- APG APG Address Generator, APG Data Generator and APG Chip Selects.
- Execution control operations, including done, conditional or unconditional branches, jumps, GOSUB, RETURN, etc.
- APG counter increment/decrement, reloading, etc.
- Any use of UDATA.
- Time-set, Pin Scramble Map, VIHH Map, selection
- READ, READUDATA, NOREAD, ADHIZ.
- APG Interrupt Timer operations.
- RESET, NOLATCH, VCOMP, OVER.
- VPULSE, LBDATA.

However, using the DDR Pin Scramble facilities it is possible to obtain useful memory pattern operations at DDR rates, as noted below:

- Use DDR Pin Scramble maps to select between two different APG data sources on a given pin in each tester cycle. For example, on a DUT data pin, the DDR A-cycle selects `t_d0` and B-cycle selects `t_d18`. The APG Data Generator Data Register, JAM Register/RAM, UDATA and DBM data sources can be creatively programmed to support DDR needs.
- Use two different chip selects per timing channel and DDR Pin Scramble maps to switch between them at DDR rates.
- To obtain an incrementing address at DDR rates the APG address generator is incremented by-2 in each instruction (using XALU/YALU ADD). Then, on the DUT's LSB address pin the DDR Pin Scramble maps a data source set to a fixed logic-0 in the A-cycle and a fixed logic-1 in the B-cycle; these fixed logic states can come from any unused pattern resource (`t_lvm`, `t_cs1`, etc.). More creative solutions may be necessary to generate more complex address sequences.

Note: DDR I/O can only be obtained using DDR Logic Vectors or DDR Scan Vectors. It is not possible to independently program A-cycle vs. B-cycle I/O using memory pattern instructions (ADHIZ, etc.).

Note: it is also possible to capture errors to the Error Catch RAM (ECR) from DDR Memory Test Patterns, however specific rules apply. See ECR in DDR, MUX and Super-MUX Modes.

3.19.3.8 DDR Timing

See Double Data Rate (DDR) Mode, DDR Hardware Details.

Note: DDR-related information does not apply to Maverick-I.

Special versions (overloads) of `settime()` are used to program DDR timing formats and edge times. In general, the DDR versions of `settime()` support programming, in a single function call, two timing values for each timing event, one for the DDR A-cycle events and one for B-cycle events.

In DDR mode, all timing events are programmed relative to the start of the tester hardware cycle (T0), not from the DDR A-cycle/B-cycle viewpoint. Both the DDR A-cycle and B-cycle pattern data is available at the output of the DDR Pin Scramble hardware at the beginning (T0) of each tester cycle, regardless of which data sources are selected and independent of how that data is subsequently used by the timing channel. Using the DDR versions of `settime()` (more below), the first time value argument(s) are rigidly bound to the DDR A-cycle pattern data, and the second value argument(s) to with the B-cycle pattern data (this association cannot be changed). However, neither the hardware nor the software check whether a timing event controlled by A-cycle pattern data actually occurs before or after an event controlled by the B-cycle data, thus, it is possible to have A-cycle timing events occur after B-cycle events if the event timing is so programmed. In some situations this may be useful, however...

Note: proper strobe operation requires that all A-cycle strobe events occur before any B-cycle strobe events.

Note: after executing DDR Test Patterns and before executing any non-DDR test patterns (i.e. when switching from DDR to non-DDR) it is always necessary to reprogram the I/O timing edges in all time-sets used in the non-DDR test patterns. Failure to do so may result in improper operation which is very difficult to diagnose.

The following format selection options are used as arguments to `settime()` to program DDR timing:

`DDR_DRIVE`

`DCLKPOS`

`DCLKNEG`

`DDR_STROBE`

`DDR_IODRIVE`

`DDR_IOSTROBE`

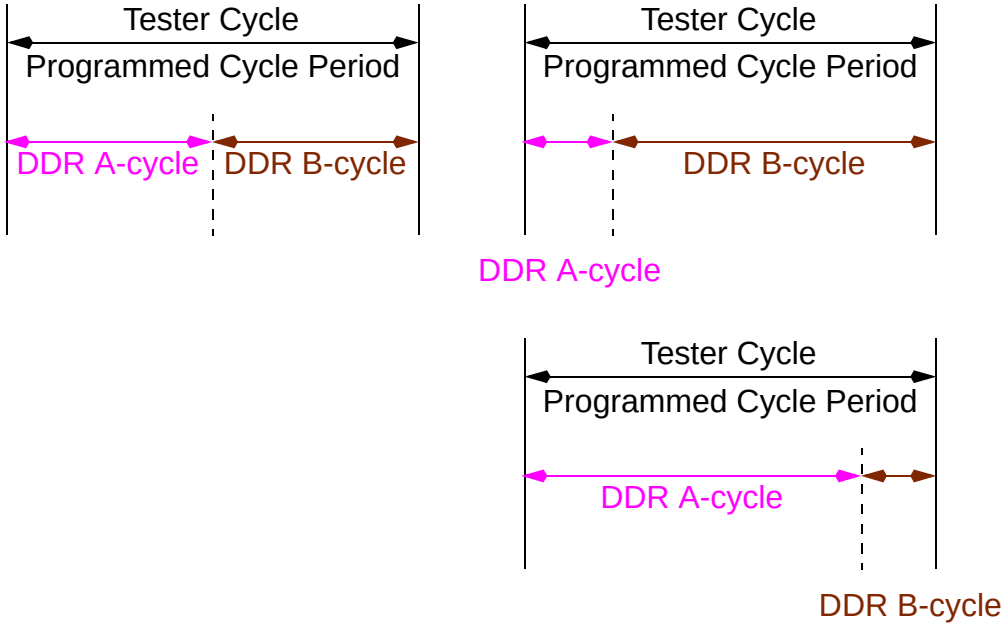
As in non-DDR timing, it is possible to program a drive format, strobe format, and I/O control for each pin in each time-set (TSET). This will require executing `funtest()` once for each format.

Note: there are no special DDR mode controls used to identify timing intended for use with DDR test patterns. By design, it is possible to program some time-sets for DDR use and others for non-DDR use. No error checks are made to validate DDR timing vs. non-DDR timing. It is the user's responsibility to properly manage these details to obtain the desired functionality.

DDR Cycle Periods

From the DUT's viewpoint, when executing DDR Test Patterns (i.e. in Double Data Rate (DDR) Mode) each tester cycle represents two cycle periods (A/B). However, internally, there is no change in how the hardware generates tester cycles. The following diagrams show this. The two right diagrams show that control of the effective cycle time of a DDR

A-cycle vs. B-cycle can be obtained by changing the location of drive, strobe and I/O edges within a given tester cycle:.



All cycle period values are programmed, using the `cycle()` function, in the context of tester cycles i.e. it is not possible to separately program A-cycle vs. B-cycle cycle periods. And, in hardware, both the A and B cycles start at the same time; it is up to the user to program the drive, strobe and I/O timing edges to obtain the effective DDR timing desired.

DDR Drive Timing & Formats

When executing DDR Test Patterns (i.e. in Double Data Rate (DDR) Mode) , drive formats are limited to `DDR_DRIVE` (NRZ) and double clock formats (`DCLKPOS`, `DCLKNEG`).

In DDR mode, NRZ is programmed using the `DDR_DRIVE` format. The `settime()` function supports programming two time value arguments for the `DDR_DRIVE` format; i.e.:

```
settime ( TSET1,
          pins,
          DDR_DRIVE,
          Acycle-NRZ-time,
          Bcycle-NRZ-time);
```

For example:

```
settime ( TSET1, ddr_pins, DDR_DRIVE, 5 NS, 55 NS );
```

In this example, the first time value (5nS) sets the DDR A-cycle `DDR_DRIVE` edge and the second time value (55nS) controls the B-cycle `DDR_DRIVE` edge. More correctly, the first time value argument is associated with the A-cycle pattern data, and the second time value argument is associated with the B-cycle pattern data.

As shown in the diagram on page 642, the A-cycle `DDR_DRIVE` events are generated by TG-A, and B-cycle `DDR_DRIVE` events are generated by TG-B.

As indicated, double clock waveforms are available in DDR mode, with the same constraints as in non-DDR mode:

- Double clock is a static pin mode.
- No other waveform formats (drive, strobe or I/O) can be used on that pin.

Double clock is enabled when a `DCLKPOS` or `DCLKNEG` format is programmed on a given pin (in any time set), and disabled when any other format is programmed on the pin (in any time set). Using Magnum 1/2/2x, the Dclk Mode must also be set to enable DDR mode (see Magnum PE Driver Modes).

In DDR mode, using the double clock format has the following additional considerations:

- The double clock format is programmed using the same `settime()` function and arguments as are used in non-DDR mode.
- In DDR mode, the number of timing edges generated (4) using double clock format is the same as in non-DDR mode; i.e. there is not a double clock in the DDR A-cycle and another double clock in the B-cycle.
- In the test pattern, the DDR A-cycle pattern data controls the first phase of the double clock and the DDR B-cycle pattern data controls the second phase of the double clock. This is different than the Maverick-I/-II.
- It is legal to use a -1 edge time value to disable selected double clock edges. Using multiple time-sets, it is possible during test pattern execution, on a per-cycle basis, to effectively convert double clock format to a single clock or no-clock format by switching to the appropriate time-set in the test pattern. For example, using the following 4 time-sets, it is possible to generate the 4 permutations of double clock pulses:

```
// Both A-cycle and B-cycle clock pulses
settime( TSET1, some_pin, DCLKPOS, 0 NS, 10 NS, 20 NS, 30 NS );

// Disable B-cycle clock pulse
settime( TSET2, some_pin, DCLKPOS, 0 NS, 10 NS, -1, -1 );

// Disable A-cycle clock pulse
settime( TSET3, some_pin, DCLKPOS, -1, -1, 20 NS, 30 NS );
```

```
// Disable both A-cycle and B-cycle clock pulses
settime( TSET4, some_pin, DCLKPOS, -1, -1, -1, -1);
```

Two of the four double clock edges are generated by the timing generators normally used to generate I/O edges. These timing generators must be configured somewhat differently when generating double clock edges. This is only significant for one reason: after a given pin has been configured to generate a double clock, if/when that pin is set up to generate any other format(s) the `IODRIVE` and `IOSTROBE` timing for that pin must also be [re]programmed in all time-sets which use I/O edges, including when default I/O timing is used. To restore default I/O timing use the following:

```
settime( TSETn, IODRIVE, your_pins, 0 NS, -1 );
settime( TSETn, IOSTROBE, your_pins, 0 NS, -1 );
```

DDR Strobe Timing & Formats

Note: proper DDR strobe operation requires that all A-cycle strobe events occur before any B-cycle strobe events.

Only edge strobes can be used in DDR mode, programmable on a per-pin basis, using the `edge_strobe()` function.

As shown in the diagram on page 642, the A-cycle edge strobe is generated by TG-A and the B-cycle edge strobe by TG-B.

A DDR version of the `settime()` function supports programming two time value arguments for the `DDR_STROBE` format; i.e.:

```
settime ( TSET1, data_pins, DDR_STROBE,
          Acycle_strobe_time,
          Bcycle_strobe_time );
```

For example:

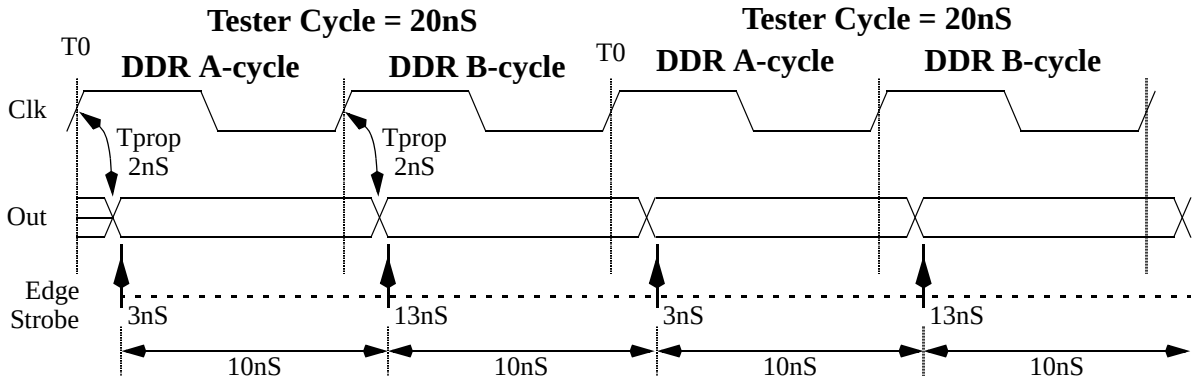
```
settime ( TSET1, data_pins, DDR_STROBE,
          15 NS,
          65 NS );
```

Note: it is critical that -1 ***NOT*** be used to disable strobe edges. Doing so causes improper strobe operation and invalid test results and is very difficult to diagnose.

Note: it is important to note that the first strobe edge is bound to the A-cycle pattern data, and the 2nd edge to B-cycle pattern data, and that this relationship cannot be changed. Proper operation requires that the A-cycle strobe be programmed (and occur) before the B-cycle strobe.

The minimum time between the end of one DDR strobe and the start of the next DDR strobe (on the same pin) is 6nS. This is true within a given tester cycle and also between tester cycles i.e. 6nS must occur between a strobe from a given B-cycle and a strobe in the next A-cycle, etc.

Using Magnum 1/2/2x, the following diagram shows 100MHz DDR strobe timing. This example uses edge strobes spaced at 10nS (the minimum spacing is 6nS):



3.19.3.9 DDR I/O Timing

See DDR Timing, Double Data Rate (DDR) Mode, DDR Hardware Details.

Note: DDR-related information does not apply to Maverick-I.

To fully understand DDR I/O timing it helps to understand non-DDR I/O timing, see Timing Generator Modes, Magnum Timing Rules.

In DDR mode the following I/O state combinations are possible within a single tester cycle:

A-cycle	B-Cycle
Drive-off	Drive-off
Drive-on	Drive-off
Drive-off	Drive-on
Drive-on	Drive-on

I/O edge timing is programmed using the `settime()` function and specifying the `DDR_IOSTROBE` or `DDR_IODRIVE` formats documented in below. It is legal to use -1 to disable DDR I/O timing edges.

Note: DDR I/O operations can only be controlled using DDR Logic Vectors or DDR Scan Vectors. The hardware does not allow independent DDR A-cycle vs. B-cycle I/O control using memory pattern instructions (`ADHIZ`, etc.).

Note: after executing DDR Test Patterns and before executing any non-DDR test patterns (i.e. when switching from DDR to non-DDR) it is always necessary to reprogram the I/O timing edges in all time-sets used in the non-DDR test patterns. Failure to do so may result in improper operation which is very difficult to diagnose.

For drive-off timing use:

```
settime ( TSET#, pinlist,
          DDR_IOSTROBE,
          A_cycle_driveoff_time,
          B_cycle_driveoff_time );
```

For example:

```
settime ( TSET1, data_pins, DDR_IOSTROBE, 10 NS, 60 NS );
```

The first time value (10nS) sets the DDR A-cycle drive-off time, and the second time value (60nS) sets the B-cycle drive-off time. Don't forget that it is the combination of test pattern and Pin Scramble data source selection that actually determines whether a given pin actually tri-states or drives in A-cycle or B-cycle.

For drive-on timing use:

```
settime ( TSET#, pinlist,
          DDR_IODRIVE,
          A_cycle_driveon_time,
          B_cycle_driveon_time );
```

For example:

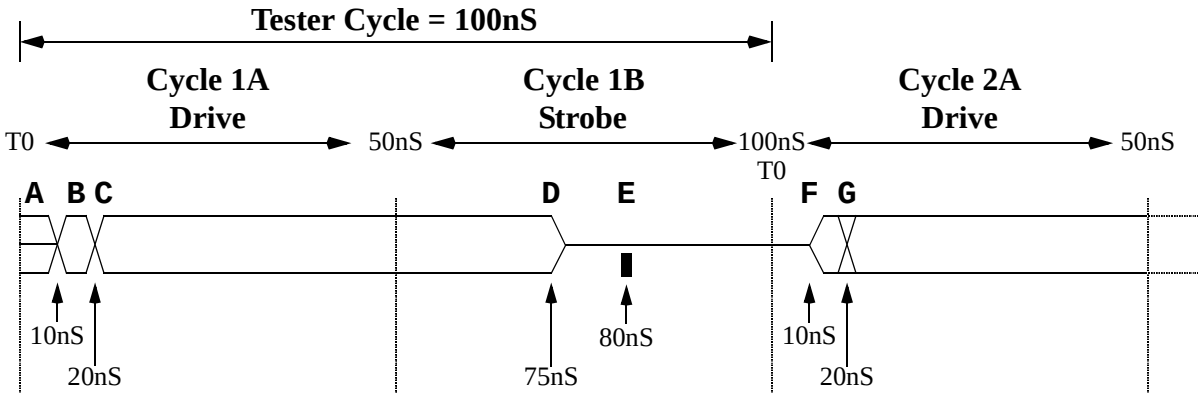
```
settime ( TSET1, data_pins, DDR_IODRIVE, 5 NS, 55 NS );
```

The first time value (5nS) sets theDDR A-cycle drive-on time, and the second time value (55nS) sets the B-cycle drive-on time. Again, it is the combination of test pattern data and Pin Scramble data source selection that actually determines whether a given pin actually drives or tri-states in either A-cycle or B-cycle.

The examples below demonstrate DDR I/O timing. The timing diagrams intentionally show 1.5 tester cycles, to show variations of drive/strobe sequences. In DDR mode, all timing events are programmed relative to the tester hardware cycle, not from the DDR A-cycle/B-cycle viewpoint.

```
edge_strobe( pins, TRUE );
cycle( TSET1, 100 NS );
settime( TSET1, pins, DDR_DRIVE, 20 NS, 70 NS);
settime( TSET1, pins, DDR_IODRIVE, 10 NS, 60 NS );
settime( TSET1, pins, DDR_STROBE, 30 NS, 80 NS );
settime( TSET1, pins, DDR_IOSTROBE, 25 NS, 75 NS );
```

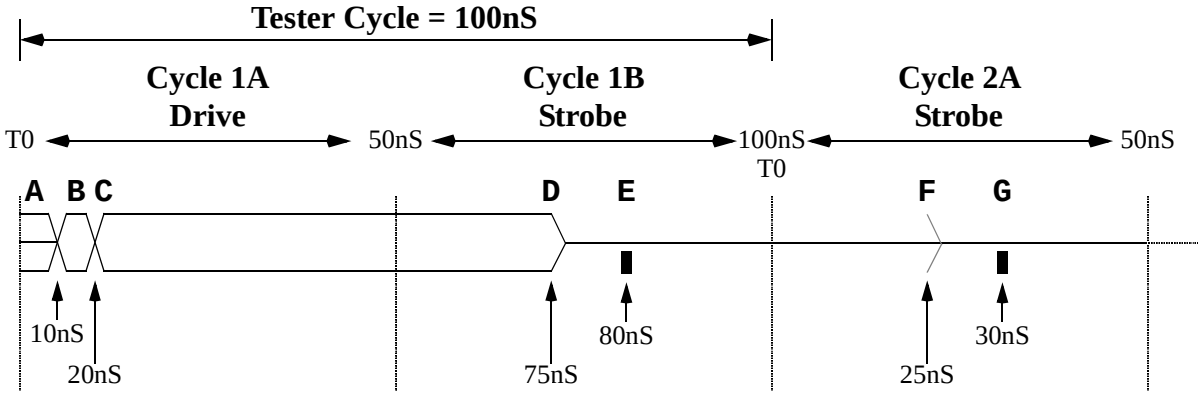
Drive, Strobe, Drive Sequence



In the previous diagram the lettered transitions represent:

- A = prior state, unknown.
- B = the DDR_IODRIVE drive-on transition at 10nS.
- C = the DDR_DRIVE transition at 20nS.
- D = the DDR_IOSTROBE tristate transition at 75nS.
- E = the DDR_STROBE at 80nS.
- F = the DDR_IODRIVE drive-on transition at 10nS.
- G = the DDR_DRIVE drive transition at 20nS.

Drive, Strobe, Strobe Sequence

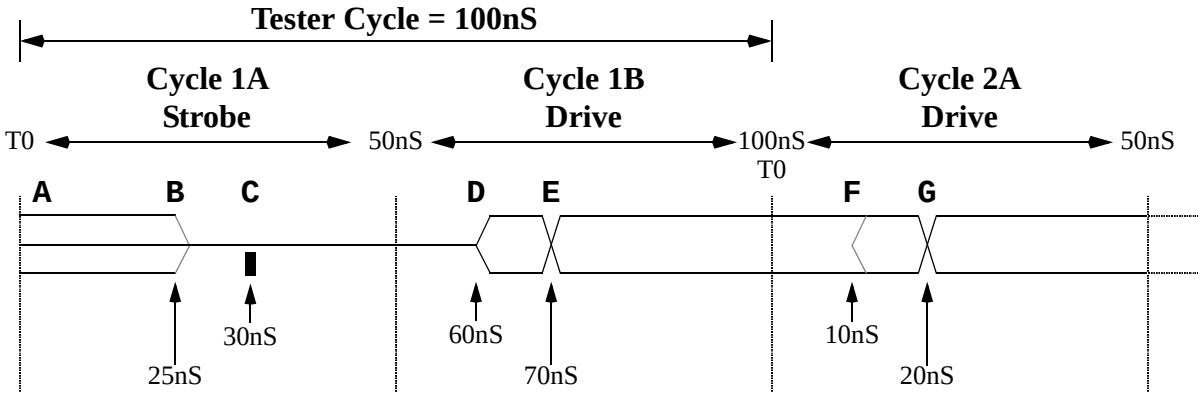


In the previous diagram the lettered transitions represent:

- A = prior state, unknown.
- B = the DDR_IODRIVE drive-on transition at 10nS.

- C = the DDR_DRIVE transition at 20nS.
- D = the DDR_IOSTROBE tristate transition at 75nS.
- E = the DDR_STROBE at 80nS.
- F = the DDR_IOSTROBE tristate transition at 25nS.
- G = the DDR_STROBE drive transition at 30nS.

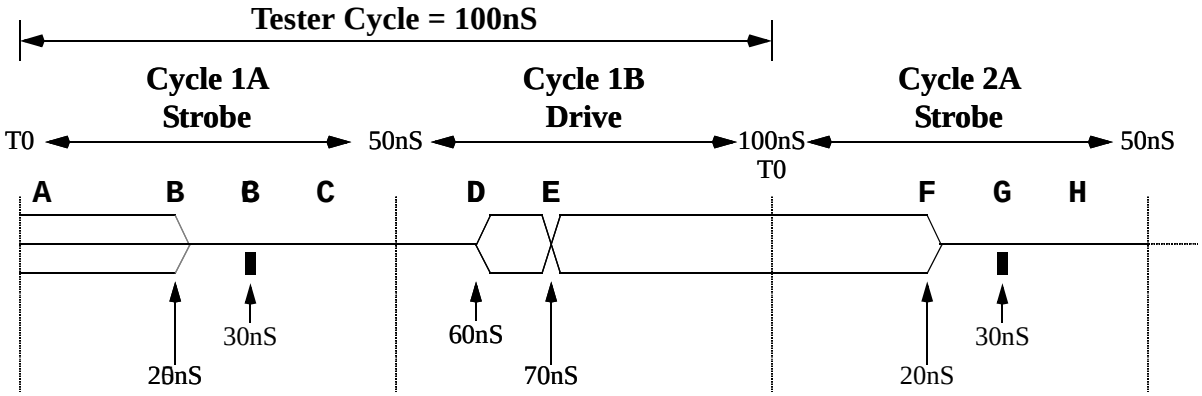
Strobe, Drive, Drive Sequence



In the previous diagram the lettered transitions represent:

- A = prior state, unknown.
- B = the DDR_IOSTROBE tristate transition at 25nS.
- C = the DDR_STROBE at 30nS.
- D = the DDR_IODRIVE transition at 60nS.
- E = the DDR_DRIVE transition at 70nS.
- F = the DDR_IODRIVE transition at 10nS.
- G = the DDR_DRIVE transition at 20nS.

Strobe, Drive, Strobe Sequence



In the previous diagram the lettered transitions represent:

- A = prior state, unknown.
- B = the `DDR_IOSTROBE` tristate transition at 25nS.
- C = the `DDR_STROBE` at 30nS.
- D = the `DDR_IODRIVE` transition at 60nS.
- E = the `DDR_DRIVE` transition at 70nS.
- F = the `DDR_IOSTROBE` tristate transition at 20nS.
- G = the `DDR_STROBE` at 30nS.

3.19.3.10 DDR Fail Signal MUX

See Double Data Rate (DDR) Mode, DDR Hardware Details, `fail_signal_mux()`.

Note: DDR-related information does not apply to Maverick-I.

The DDR Fail Signal MUX is used to support capturing failure information in the Error Catch RAM (ECR) at DDR rates.

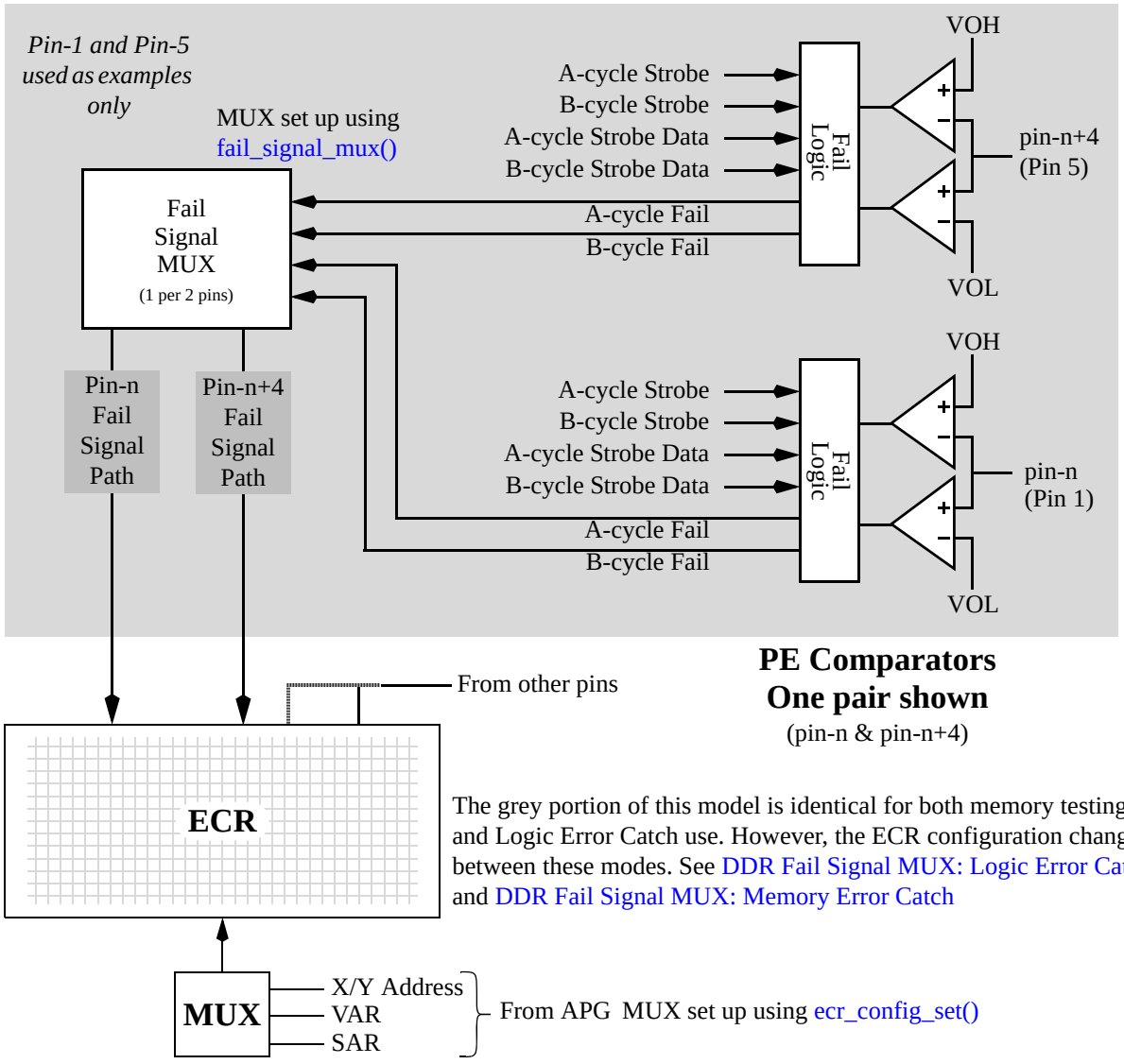
Note: this section provides an overview of the Fail Signal MUX hardware. In use, some details are different when capturing memory test failures (for redundancy and bitmapping) vs. logic test failures (for datalogging). These topics are covered separately in DDR Fail Signal MUX: Memory Error Catch and DDR Fail Signal MUX: Logic Error Catch.

The following hardware facilities exist in the Magnum 1 for capturing failure information during test pattern execution:

- Non-ECR use always captures the pins failing in the first failing cycle, all failing pins, and the first failing X/Y address, MAR, VAR, or SAR.
- Using the optional ECR with Memory Test Patterns, failing X/Y addresses and failing pins can be captured.
- Using the optional ECR with Logic Test Patterns, failing logic vector address (VAR) and failing pins can be captured.
- Using the optional ECR with scan test patterns, failing scan vector address (SAR) and failing pins can be captured.

In hardware, each pin of every pin-pair has an error signal path to the ECR, used when that pin's errors are captured to the ECR. When using a single data rate (SDR) test patterns (non-DDR test patterns) one error signal path per-pin is enough to capture each pin's results to the ECR in each tester cycle. However, when executing DDR Test Patterns, each pin may receive 2 strobes in each tester cycle. In these situations, to properly use the ECR requires logging 2 errors per-pin in each tester cycle. This, in turn, requires the use of 2 error signal paths to the ECR for the pins being logged. These extra signal paths are obtained using the Fail Signal MUX to commandeer the signal path from a near-by pin, called a *partner* pin below. Pins which have their signal path commandeered cannot be logged to the ECR. Thus, using the ECR, it is possible to capture both A-cycle & B-cycle failures i.e. DDR failures, in each tester cycle, for up to 1/2 of the tester's signal pins.

The following diagram is used to describe the Fail Signal MUX hardware and operation. Two *partner* pins are shown:



The grey portion of this model is identical for both memory testing and Logic Error Catch use. However, the ECR configuration changes between these modes. See DDR Fail Signal MUX: Logic Error Catch and DDR Fail Signal MUX: Memory Error Catch

Figure-45: Fail Signal MUX Block Diagram

As shown in the model above, for each pair of *partner* pins, the Fail Signal MUX always outputs two error signals to the ECR. By default, these represent the error signal from each pin. The inputs to the Fail Signal MUX consist of 2 error signals from each pin, above called A-cycle Fail and B-cycle Fail. Only the A-cycle fail signal is used when logging SDR errors to the ECR. When logging errors from DDR Test Patterns, the selection of which error signals are used is dependent on how the Fail Signal MUX is configured, more below.

In the context of Fail Signal MUX use, pin partners are offset by 4; i.e. pin a_1 with a_5, b_2 with b_6, etc. In this document, this is also annotated as pin-n and pin-n+4.

By default, when executing any test pattern, only the A-cycle signal from each pin is used, to capture one error, per-pin, per-cycle to the ECR. In order to capture DDR errors (i.e. 2 errors per-pin, per-cycle) requires:

- Use `ecr_ddr_mode_set()` to prepare for DDR ECR use. Must execute before `ecr_config_set()`, `lec_config_set()`.
- Use `fail_signal_mux()` to configure the Fail Signal MUX to select which pin of each pair will be logged to the ECR. Must execute before `ecr_config_set()`, `lec_config_set()`.
- Configure the ECR:
  - Use `ecr_config_set()` to configure for memory pattern ECR use.
  - Use `lec_config_set()` to configure for logic pattern ECR use.
- Execute one or more DDR Test Patterns to capture errors to the ECR. DDR test patterns can generate 2 strobe signals per-pin (DDR A-cycle strobe and B-cycle strobe) with independent A-cycle vs. B-cycle expect data from the pattern.

As shown above, for DDR acquisition, the Fail Signal MUX hardware allows the error signal paths from one pin to be used by a partner pin, during which time errors from partner pin can't be captured to the ECR. This allows one pin of each partner-pair to log both DDR A-cycle and B-cycle errors to the ECR, with up to 1/2 of all tester pins captured at DDR data rates. The *partner* pins may be used as input pins, or strobes can still be applied, which will affect the overall PASS/FAIL results and test pattern branch-on-error operations.

Each Fail Signal MUX contains internal logic, to determine how the four inputs (from 2 pins) are mapped to the 2 outputs. Three options are supported and the selection is made using the `fail_signal_mux()` function:

- The A-cycle Fail signals are output from each pin (non-DDR mode)
- The A-cycle Fail and B-cycle Fail signals from one pin are output (DDR mode).
- The logical OR of the A-cycle Fail and B-cycle Fail signals of pin-n are output on the corresponding pin and the logical OR of the A-cycle Fail and B-cycle Fail signals of pin n+4 are output on the other pin.

As indicated above, prior to executing a test pattern which logs errors to the ECR, it must also be configured. The functions used are different for memory pattern vs. logic pattern vs. scan pattern error capture:

- Memory pattern capture uses `ecr_config_set()` to specify the number of X/Y addresses, number of data bits, and which pins are to be captured (the other parameters don't matter to this discussion). These functions were not modified for DDR use. See DDR Fail Signal MUX: Memory Error Catch.
- Logic and scan fail capture uses `lec_config_set()` to specify the pins to be captured. See DDR Fail Signal MUX: Logic Error Catch.

The `lec_config_set()` function is used to control the MUX which determines whether APG X/Y address, VAR, or SAR is captured in the ECR. `lec_config_set()` must be executed before executing the test pattern which logs errors to the ECR.

Miscellaneous Comments

- The `test_pin()` and `test_pin_first_error()` functions read failing pin information on the output of each pin's fail logic, before the signals enter the Fail Signal MUX. Thus, the configuration of the MUX has no effect on these functions.
- As noted in DDR Scan Vectors, to obtain scan data at DDR data rates requires configuring one or more Pin Scramble Map(s) to map Scan outputs to A-cycle vs. B-cycle use. A failing SAR represents all Scan memory outputs in one tester cycle i.e. not DDR. However, failing pins can be resolved to A-cycle vs. B-cycle failures. User-written code must determine how failing scan data is displayed.

3.19.3.11 DDR Fail Signal MUX: Logic Error Catch

See Double Data Rate (DDR) Mode, DDR Fail Signal MUX, `fail_signal_mux()`.

Note: DDR-related information does not apply to Maverick-I.

Note: read DDR Fail Signal MUX first.

As noted in DDR Fail Signal MUX, the upper portion of the diagram below applies to using the ECR for both memory error catch and logic error catch modes. However, the ECR

configuration changes between memory and logic modes. Scan patterns are treated the same as logic patterns. The model below shows the logic testing ECR configuration:

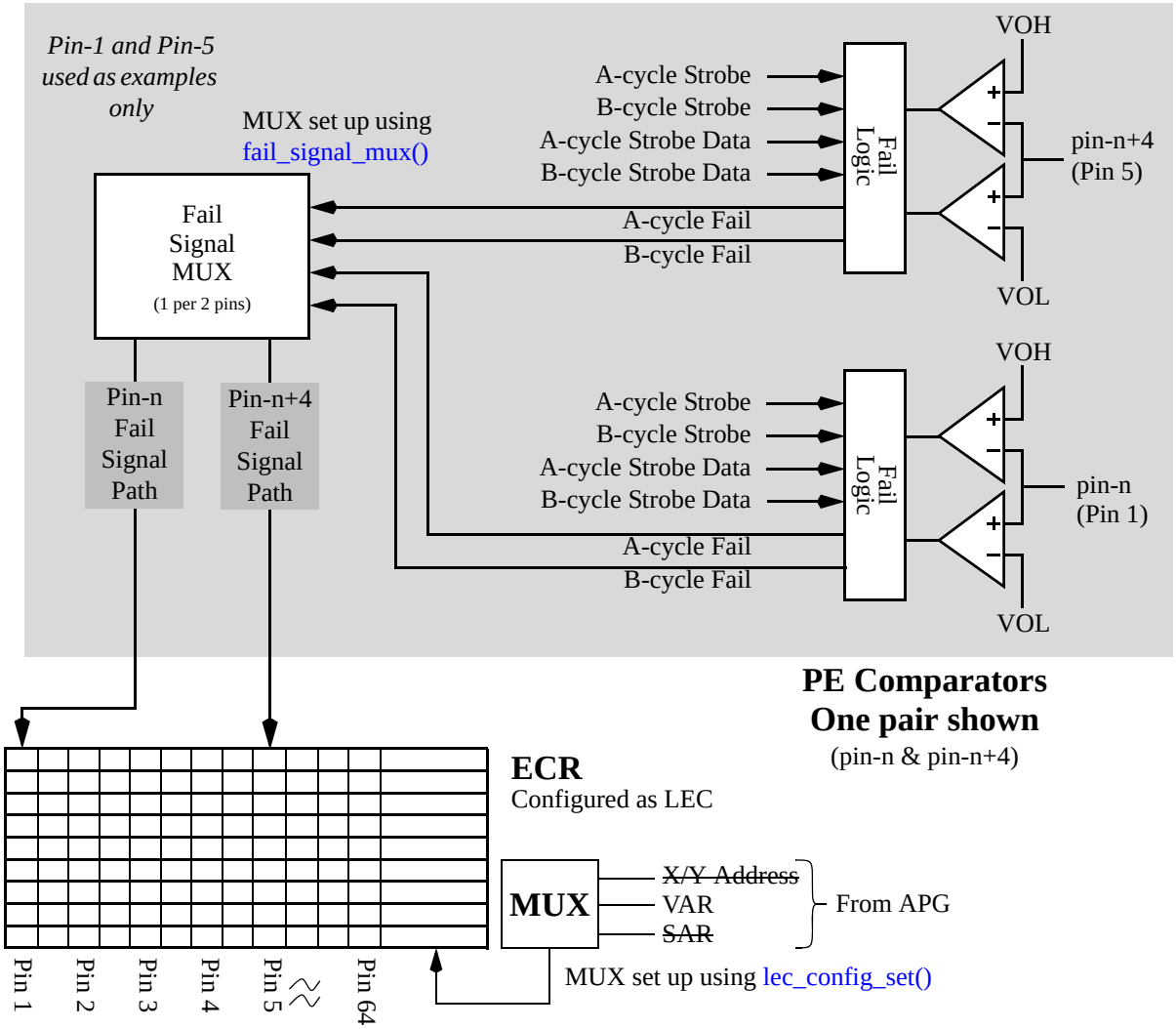


Figure-46: Fail Signal MUX Block Diagram: Logic Error Catch

To perform logic error capture the following functions are used as noted:

- `fail_signal_mux()` is used to configure the Fail Signal MUX.
- `lec_config_set()` is used to specify which [failing] pins are to be captured and to configure the ECR to capture DDR failures.

To switch from [a previously set up] memory error catch mode to logic error catch mode requires that `lec_config_set()` be used.

3.19.3.12 DDR Fail Signal MUX: Memory Error Catch

See Double Data Rate (DDR) Mode, DDR Fail Signal MUX, `fail_signal_mux()`.

Note: DDR-related information does not apply to Maverick-I.

Note: read DDR Fail Signal MUX first.

The following failure information is captured without using the ECR, in both DDR and non-DDR modes:

- The first failing MAR and failing X/Y address are captured on the APG.
- The pins failing the first instruction are captured in the PE error latches.

These values can be read using standard functions for use as datalog, etc.

When using the ECR, the main difference between logic error catch and memory error catch is the type of information logged in addition to failing pins.

As noted in DDR Fail Signal MUX, the upper board portion of the model below applies to using the ECR for both memory testing and Logic Error Catch Mode. However, the ECR configuration changes between memory and logic modes:

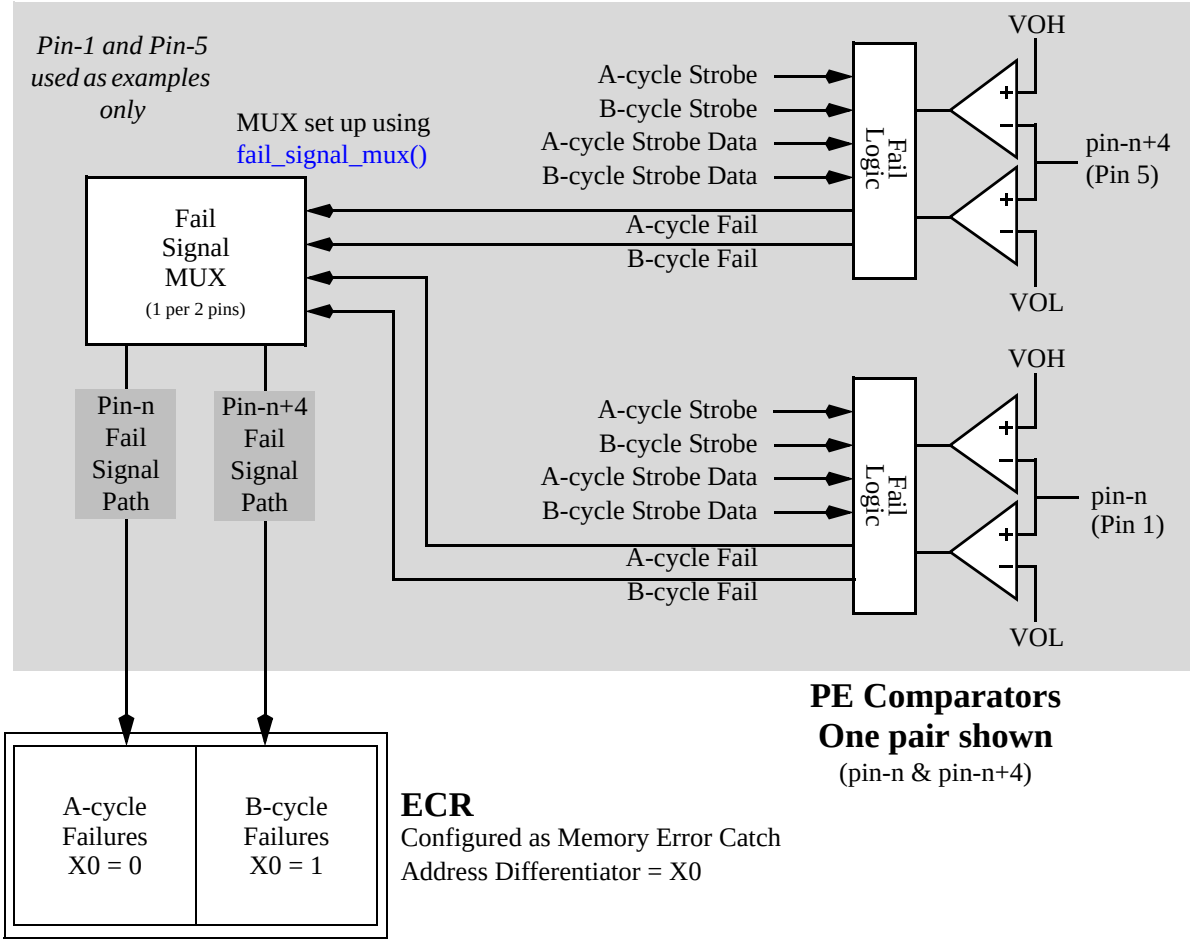


Figure-47: Fail Signal MUX Block Diagram: Memory Error Catch

In non-DDR memory testing, the ECR is typically configured to match the DUT in size, in both the X/Y address domain and in the number of data bits captured at each address. During testing, the X/Y addresses generated by the APG to test the DUT are also used to address the ECR. X/Y addresses are generated in an arbitrary sequence, as determined by the user-written test pattern. At each failing address the ECR records which pins fail.

During memory test, failures accumulate in the ECR i.e. a given DUT address may be read more than once, and thus fail more than once, with the same or different pins failing each time the address is read. Using the ECR in memory testing, the ECR does not record which address or pin(s) fail first, or how many times a given address and/or pin failed. After test execution has completed, the ECR is read to see which pins failed at each failing X/Y

address. This is done during Redundancy Analysis (RA), for failure display in BitmapTool, or for other purposes.

In DDR mode, the APG's X/Y address and data outputs can only change once in each tester cycle. To perform DDR memory testing, the DDR Pin Scramble facilities are used to map two different APG outputs to a given DUT pin, in each tester cycle. For example, on a given DUT data pin, the DDR A-cycle data can READ D0, with B-cycle data can READ D8, etc. Since two READs are performed per tester cycle two failures can occur in each tester cycle, per-pin. These are the A-cycle and B-cycle failures discussed below.

In DDR mode, the APG generates, and the ECR receives, only one X/Y address in each tester cycle. Thus, to capture two failures in each tester cycle the ECR must be configured to log two discrete failures, per pin, at each X/Y address. As noted later, the `ecr_ddr_mode_set()` function is used to set up the ECR in this configuration and the `fail_signal_mux()` function is used to refine which pins are captured. To capture 18-wide data in DDR mode the entire 36-bit width of the ECR be used. Half of the ECR captures A-cycle failures, and the other half B-cycle failures. In other words, the ECR can capture up to 36-wide data in non-DDR mode, or up to 18-wide data in DDR mode.

For DDR testing to be useful also requires that two unique addresses be applied to the DUT in each tester cycle. This is also done using the DDR Pin Scramble. But only one of these addresses (the APG X/Y address outputs) is actually used by the ECR, to log both A-cycle and B-cycle failures in each tester cycle. One address is adequate when logging failures, but not when reading the ECR to report failures. In order for the system software to report A-cycle failures at an address which is unique from B-cycle failures, the address difference between A-cycle and B-cycle must be known when the ECR is read. This is the other purpose of `ecr_ddr_mode_set()`, to specify an *address differentiator* used when reading the ECR. In most applications the address differentiator will be either X0 or Y0.

The diagram above shows the ECR configured for memory testing. In this example, both A-cycle and B-cycle failures are captured from pin-n. The *address differentiator* is specified to be **`X0`**:

Given this configuration, note the following about the example above:

- Half of the ECR will contain A-cycle failures i.e. when the address differentiator (**`X0`**) is logic-0, and the other half will contain B-cycle failures i.e. when **`X0`** = 1.
- The system software which reads the ECR depends upon this relationship. Thus, for proper results, both the test pattern *AND* the pin scrambling used to define and control the DDR address differentiator *MUST* ensure this operation. This is the user's responsibility.

To configure the Fail Signal MUX and ECR to perform DDR memory error capture the following functions are used:

- `fail_signal_mux()` is used to configure the DDR Fail Signal MUX.
- `ecr_ddr_mode_set()` is used to switch the ECR configuration between DDR and non-DDR modes. In non-DDR mode the ECR is configured as specified using `ecr_config_set()`. In DDR mode, the system software configures the ECR data width to be 2-times that specified using `ecr_config_set()`. One argument to `ecr_ddr_mode_set()` also specifies the address differentiator.
- `ecr_ddr_mode_set()` MUST be executed before `ecr_config_set()`, which is used to configure the ECR, in the form of X/Y address count, and the number of pins to be logged. In DDR mode, this configuration is modified by `ecr_ddr_mode_set()`, which MUST be executed before `ecr_config_set()`. The other arguments to `ecr_config_set()` don't affect this configuration and are ignored here. The `ecr_config_set()` function was not modified for DDR use.
- `ecr_config_set()` is used to configure the MUX which controls which failing address type is logged, which in memory test mode will be X/Y address (VAR or SAR is possible but likely not useful in memory test operations).

To switch from [a previously set up] logic error catch mode to memory error catch mode requires:

- `ecr_ddr_mode_set()` when DDR is to be used.
- `ecr_config_set()`

To switch between memory DDR and non-DDR memory configuration only requires using `ecr_ddr_mode_set()`. However, if the ECR configuration needs to change it is required that `ecr_ddr_mode_set()` be executed also, ***before*** `ecr_config_set()`.

3.19.4 MUX Mode

See MUX, Super-MUX and DDR, `mux_mode_set(), mux_mode_get()`.

MUX mode is used to double the effective maximum data rate of selected pins and/or reduce the minimum effective cycle period of those pins. This is accomplished by routing the outputs of two adjacent timing channels, normally used to drive two separate pin-pairs, to drive just one pin-pair, as shown in the following diagram. This means that in each tester cycle, pins in MUX mode will receive two sets of pattern data via two timing channels, thus

doubling the maximum data rate:

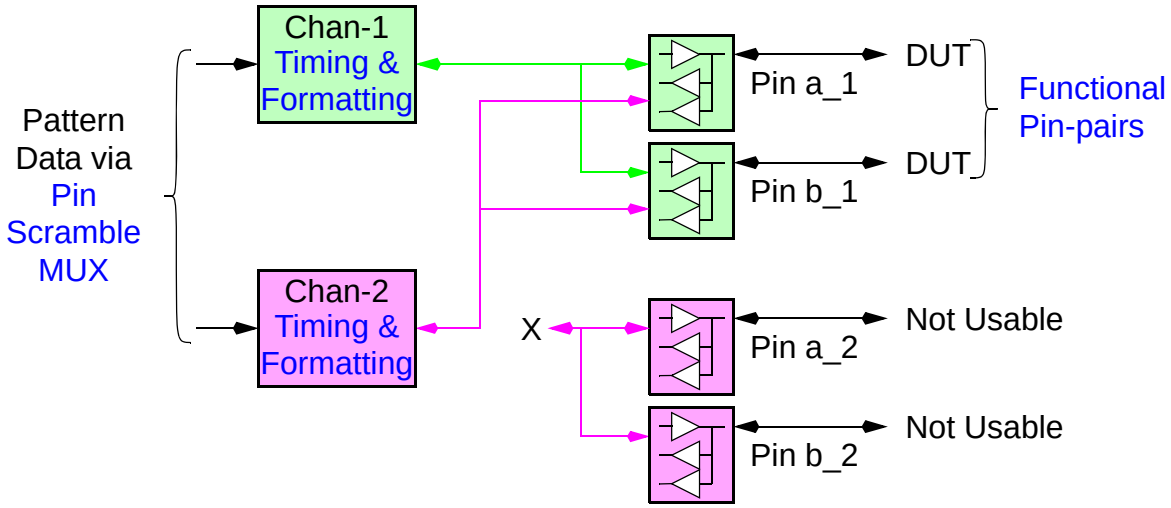


Figure-48: MUX Mode Block Diagram

MUX mode pins are configured using `mux_mode_set()`.

MUX mode pins are configured in pairs, called *MUX-pin pairs*. MUX-pin pairs consist of a *used* pin (an odd-numbered pin) and an *adjacent* pin (the next higher even-numbered pin) . The *used* pin is the pin electrically connected to the DUT and, as shown above, the *adjacent* pin is not usable at the DUT. Both pins of a given pin-pair (for example, a_1 and b_1, see Functional Pin-pairs) are always configured identically and all MUX-related rules apply to both pins.

When specifying MUX mode, even though the even-numbered pin(s) can't be used at the DUT, they must be included in the test program, to allow test pattern, pin scramble and timing parameters to be programmed on those pins, more below.

All drive formats are usable on pins in MUX mode, as are both window and edge strobes. However, all I/O control signals come from the odd pin's timing channel, which affects how I/O timing is programmed, more below.

When programming strobe timing, two strobes may occur in one tester cycle, one from each MUX timing channel.

Like all signal pins, pins used in MUX mode must be controlled from the test pattern. In general, the pattern data (and strobe and I/O control) for the odd-numbered MUX pin will control timing events in the first part of a tester cycle and the pattern data for the even-numbered MUX pin will control control timing events in the second part of a tester cycle. The actual timing of all timing events is determined by the user.

In the MUX Mode Block Diagram above, pins a_1 and b_1 (the *used* pins) are in MUX mode, and pins a_2 and b_2 (the *adjacent* pins) are not usable at the DUT. However, both the *used* and *adjacent* pins must appear in the Pin Assignment Table because, as noted earlier, from the test pattern, pin scramble and timing system viewpoint, both sets of pins are being used, and must be programmed. This means that, for MUX pins, the Pin Assignment Table, Pin Scramble Table and drive, strobe and I/O timing will typically have entries for pins which do not exist on the DUT. In the following example, note the D0_mux pin. For example:

```
DUT_PIN( D0 ){}     // The used pin
DUT_PIN( D0_mux ){} // Not used at the DUT, but required for MUX
// Etc...

PIN_ASSIGNMENTS( example ) {
   SITES_PER_CONTROLLER( 1 )
      //        DUT       DUT-1   DUT-2
      //        Pin       Tester  Tester
      //        Name      Pin #   Pin #
      //        ------    ------  ------
   ASSIGN_2DUT (D0,       a_1,    b_1 )
   ASSIGN_2DUT (D0_mux,   a_2,    b_2 )
   // Etc...

}

// MUX pins use both D0 and D8 in a PS2 tester cycle
SCRAMBLE_MAP( PS2 ){
   SCRAMBLE_2DUT( D0,        t_d0, t_d0 )
   SCRAMBLE_2DUT( D0_mux,    t_d8, t_d8 )

}

// And, elsewhere...
mux_mode_set( D0, t_mux_mode ); // See mux_mode_set()

settime( TSET1, D0,      STROBE, 0 NS, 6 NS ); // See settime()
settime( TSET1, D0_mux,  STROBE, 10 NS, 16 NS );

// Only the odd pin can control I/O timing
settime( TSET1, D0,      IODRIVE, 0 NS, -1 ); // See settime()
settime( TSET1, D0,      IOSTROBE, 10 NS, -1 );
```

The Error Catch RAM (ECR) can be used to capture errors from pins in MUX mode but, since two errors must be captured for each pin, in each tester cycle, 2 error signal paths are required for each pin being captured. Thus, the error signal paths from both the *used* pin and *adjacent* pin are used.

It is possible to use MUX pins when executing DDR test patterns (see Double Data Rate (DDR) Mode). When mixing MUX pins with DDR mode, both sets of rules apply, including those for ECR use, see ECR in DDR, MUX and Super-MUX Modes.

3.19.5 Super-MUX Mode

See MUX, Super-MUX and DDR, `mux_mode_set(), mux_mode_get()`.

Note: the information in this section applies only to Magnum 2/2x and is not displayed in the documentation for other system types.

3.19.6 ECR in DDR, MUX and Super-MUX Modes

See MUX, Super-MUX and DDR.

Note: the information in this section applies only to Magnum 2/2x and is not displayed in the documentation for other system types.

3.19.7 MUX, Super-MUX & DDR Software

See MUX, Super-MUX and DDR.

This section includes the following:

- Types, Enums, etc.
- `mux_mode_set(), mux_mode_get()`
- `mux_mode(), mux_mode_disable()`
- `fail_signal_mux()`

3.19.7.1 Types, Enums, etc.

See MUX, Super-MUX & DDR Software.

Description

The following enumerated types are used in support of the various MUX, Super-MUX and DDR functions:

Usage

The `MuxModes` enumerated type is used to set/get the MUX or Super-MUX mode state of one or more pins. See `mux_mode_set(), mux_mode_get()`:

```
enum MuxModes{ t_nomux_mode, t_mux_mode, t_super_mux_mode };
```

3.19.7.2 mux_mode_set(), mux_mode_get()

See MUX, Super-MUX & DDR Software.

Note: Maverick-I/-II and Magnum 1 do not support Super-MUX Mode. All references to Super-MUX Mode apply to Magnum 2/2x only.

Description

The `mux_mode_set()` function is used to put one or more pins into MUX Mode or Super-MUX Mode or to return one or more pins to non-MUX mode. See MUX, Super-MUX and DDR.

The `mux_mode_get()` function may be used to determine the current MUX mode, if any, for one pin.

Note the following:

- Read MUX, Super-MUX and DDR first. Then read Functional Pin-pairs.
- During the initial program load all pins are set to non-MUX mode. The system software does not otherwise change the MUX-mode of any pins.

- The test system's hardware architecture determines how pins are configured in MUX Mode and Super-MUX Mode. This imposes certain rules on which pins may be used in either mode which, in turn, affects which pins may be referenced in the arguments to `mux_mode_set()`:
  - See MUX Mode for related rules.
  - See Super-MUX Mode for related rules.

  Note that these rules all refer to low-level pin identifiers (a_1, b_4, etc.). In software, these are `HDTesterPin` data types, which are normally only used when defining the Pin Assignment Table, to map `HDTesterPin`s to `DutPin`s. Elsewhere in the Nextest software, single pins are identified using a `DutPin` and multiple pins are identified using a `PinList`, which is defined to contain one or more `DutPin`s. Thus...

Note: using `mux_mode_set()`, the pins to be programmed are identified using a `DutPin` and/or `PinList`. It is the user's responsibility to ensure that the rules noted in MUX Mode and Super-MUX Mode are considered when selecting the `DutPin` and/or `PinList` arguments to `mux_mode_set()`. Ultimately, this will affect how the `HDTesterPin` used in the Pin Assignment Table are mapped to `DutPin`s.

- If pins are to remain in the same MUX mode for the duration of the test program load session, programming the MUX configuration in the Site Begin Block will save test program execution time.

Usage

```
void mux_mode_set( DutPin *pDutPin, MuxModes mode);
void mux_mode_set( PinList* pPinList, MuxModes mode);
MuxModes mux_mode_get( DutPin *pDutPin );
```

where:

`pDutPin` is used in two contexts:

- Using `mux_mode_set()`, **`pDutPin`** identifies one pin to be programmed. Important rules are described above and in MUX Mode and Super-MUX Mode. See Note:. In Multi-DUT Test Programs the same pin of all DUT(s) in the Active DUTs Set (ADS) are programmed.

- Using `mux_mode_get()`, **`pDutPin`** identifies one pin for which the current mode is to be retrieved. In Multi-DUT Test Programs the value is read from the first DUT in the Active DUTs Set (ADS).

`mode` identified the desired MUX mode. Legal values are of the `MuxModes` enumerated type. See MUX, Super-MUX and DDR.

`pPinList` identifies one or more pin(s) to be programmed. Important rules are described above and in MUX Mode and Super-MUX Mode. See Note:. In Multi-DUT Test Programs the same pin(s) of all DUT(s) in the Active DUTs Set (ADS) are programmed.

`mux_mode_get()` returns the current MUX mode for one specified pin.

Example

```
mux_mode_set( ClockPin, t_mux_mode );
mux_mode_set( ClockPins, t_super_mux_mode );
MuxModes mode = mux_mode_get( ClockPin );
```

3.19.7.3 mux_mode(), mux_mode_disable()

See MUX, Super-MUX and DDR, Overview, MUX Mode.

Description

The `mux_mode()` and `mux_mode_disable()` functions are used to enable and disable MUX mode on specified pin.

See Overview and MUX Mode for operational details.

Note the following:

- All pins are set to non-MUX mode during the initial program load. The system software does not otherwise modify the MUX mode configuration of any pins.
- Each odd tester pin (`t_1`, `t_3`, etc.), also called the *used* pin, can only be MUX'ed to the next higher even pin, called the *adjacent* pin. This is selectable on a per-pin basis.
- When a given pin is in MUX mode, the adjacent pin is not usable at the DUT.
- From the test pattern, timing and Pin Scramble standpoint, both pins of a MUX-pair must be programmed. See MUX Mode.

- On pins in MUX mode, the drive timing generator outputs from both channels are combined such that all three drive timing edges (edges 0, 1 and 2) on the odd pin and edges 1 and 2 on the even pin are available for use.
- Since two timing channels are used, two pattern data (and strobe and I/O control bits) are available in each tester cycle; use the odd pin's pattern data to control the timing of the first part of the tester cycle and the even pin's data for the second part of the tester cycle.
- When strobing, the output of the comparator on the odd pin is routed to the compare logic on both the odd and even pins. To program two strobes in one cycle, use the odd pin's strobe timing and expect data in the first part of the tester cycle and the even pin's for the second part of the tester cycle.
- In MUX mode, all I/O timing edges must come from the odd pin; i.e. any I/O formats programmed for the even pin have no effect.
- If pins are to be placed in MUX mode for the entire test program, consider doing so in the Site Begin Block, to save test program execution time.

Usage

The following function is used to enable MUX mode on one or more pins:

```
void mux_mode( PinList* pPinList );
```

The following function is used to disable MUX mode on all pins:

```
void mux_mode_disable( );
```

The following function is used to disable MUX mode on one or more pins:

```
void mux_mode_disable( PinList* pPinList );
```

where:

`pPinList` identifies the pin(s) to be enabled or disabled. The pin list must only inlclude odd numbered tester pins (`t_1`, `t_3`, etc.) on which MUX mode is to be enabled or disabled.

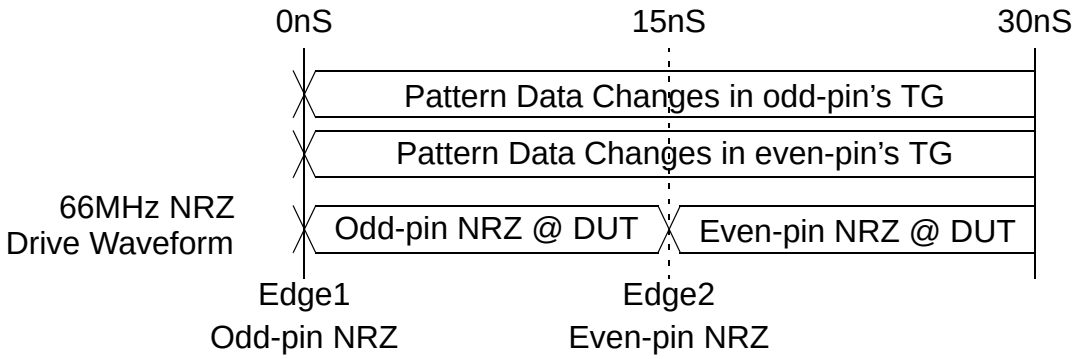
Examples

Example 1:

This following example demonstrates 66 MHz NRZ drive data:

```
mux_mode( fast_data_pins );
cycle ( 30 NS );
settime( TSET1, fast_data_pins, NRZ, 0 NS );
settime( TSET1, fast_data_pins_even, NRZ, 15 NS );
```

`mux_mode()` enables MUX mode on the pins in the pin list called `fast_data_pins`. This pin list can only include odd numbered tester pins. The first `settime()` programs an `NRZ` drive format on these same pins (the MUX pins). The second `settime()` programs an `NRZ` drive format on the even pins which are MUX'ed with the odd pins. The user is responsible for ensuring the two pin lists are correct. The resulting waveform appears at the DUT only on the odd MUX tester pins. The format associated with the odd and even pins is shown below the waveform and the edge placements are shown above the waveform:

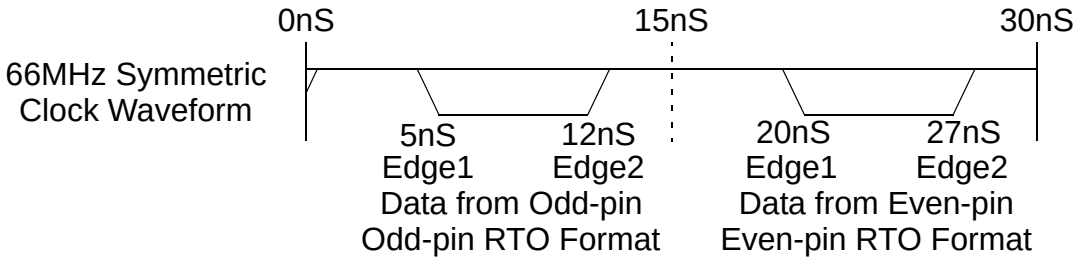


Example 2:

Th following example shows a symmetric 66 MHz clock:

```
mux_mode( fast_clock );
cycle( 30 NS );
settime( TSET1, fast_clock, RTO, 5 NS, 12 NS );
settime( TSET1, fast_clock_even, RTO, 20 NS, 27 NS );
```

`mux_mode()` enables MUX mode on the pins in the pin list called `fast_clock`. This pin list can only include odd numbered tester pins. The first `settime()` programs an `RTO` drive format on these same pins (the MUX pins). The second `settime()` programs an `RTO` drive format on the even pins which are MUX'ed with the odd pins. The user is responsible for ensuring the two pin lists are correct. The resulting MUX mode waveform appears at the DUT only on the odd pins:



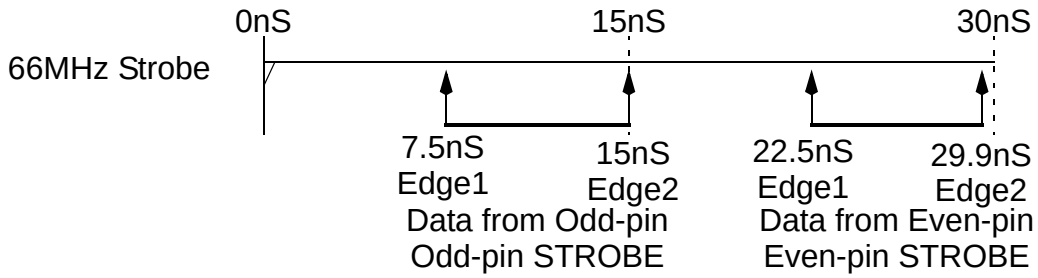
The clock is turned on (active low) when the pattern data = 0 and off when the pattern data - = 1. If the data supplied on both tester pins is always zero then a continuous 66 MHz waveform will be delivered. This type of waveform can be created using two logic vector memory data bits on a MUX mode pin pair or it can be created by using the pin scrambler to map different APG data sources to each channel of the MUX pin-pair. If logic vector memory is used, the data bits are independent and the two pulses in a tester cycle can be controlled independently. If one APG data source is used for both MUX pins, then two pulses will always occur in a tester cycle because the data supplied will always be the same for odd and even channels. Or, two different APG data sources can be mapped to the MUX mode pin pair to provide independent data bits for the two pulses in a tester cycle.

Example 3:

The following example shows 66 MHz data strobes.

```
mux_mode( fast_data_pins );
settime( TSET1, fast_data_pins, STROBE, 7.5 NS, 15 NS );
settime( TSET1, fast_data_pins_even, STROBE, 22.5 NS, 29.9 NS );
```

`mux_mode()` enables MUX mode on the pins in the pin list called `fast_data_pins`. This pin list can only include odd numbered tester pins. The first `settime()` programs a `STROBE` format on these same pins (the MUX pins). The second `settime()` programs a `STROBE` format on the even-pins which are MUX'ed with the odd pins. The user is responsible for ensuring the two pin lists are correct. The resulting MUX mode waveform is shown below for a 30nS cycle period:



3.19.7.4 fail_signal_mux()

See DDR Fail Signal MUX, See MUX, Super-MUX & DDR Software.

Description

The `fail_signal_mux()` function is used to configure the DDR Fail Signal MUX, for use in capturing errors to the ECR when executing DDR test patterns.

For proper operation, the DDR Fail Signal MUX must be configured before a test pattern executes.

The default DDR Fail Signal MUX configuration matches non-DDR operation. The default configuration is set when the test program is first loaded but not otherwise changed by the system software.

Note: there is an execution order dependency between `fail_signal_mux()` and `ecr_config_set()`. `fail_signal_mux()` sets a mode which is detected by `ecr_config_set()`, thus `fail_signal_mux()` ***MUST*** be executed before `ecr_config_set()`.

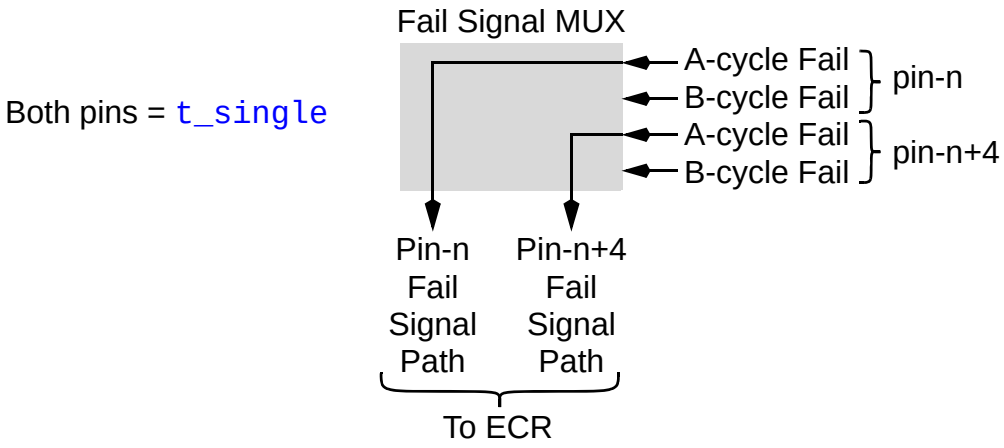
Note: proper operation requires that the Fail Signal MUX be configured identically for all pins being logged to the ECR. It is the responsibility of the user's test program code to correctly configure both the ECR and the Fail Signal MUX i.e. no error checking is performed to ensure these rules are followed. Proper operation is *unlikely* when the rules are violated.

The `FailMuxSelectOpt` enumerated type is used by `fail_signal_mux()` to specify the configuration of MUX hardware, as shown in the diagrams below:

- `t_single` is the non-DDR configuration
- `t_double` is the DDR configuration

Note that `FailMuxSelectOpt` is also used by `lec_config_set()`.

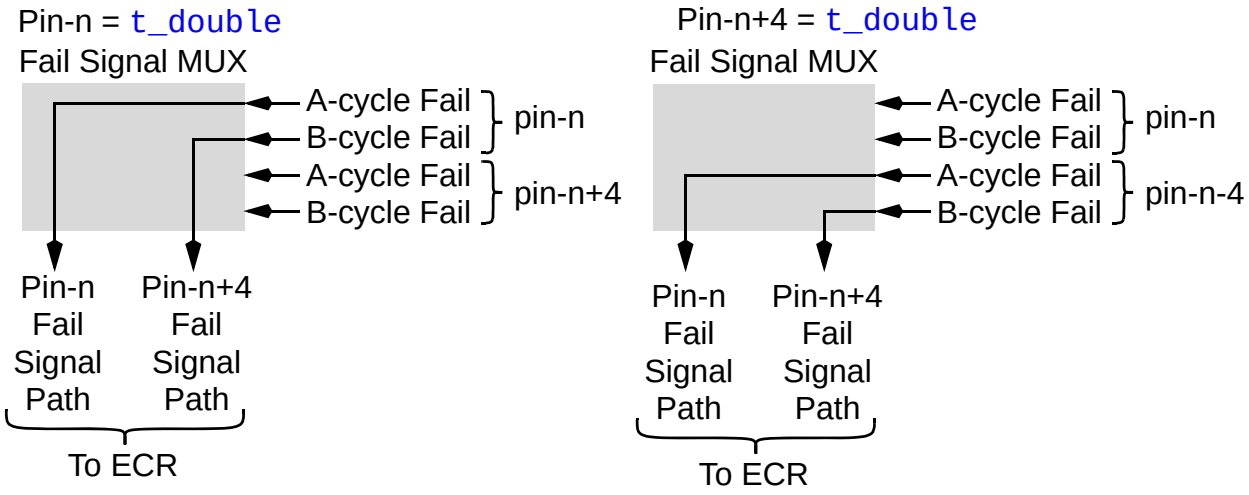
`t_single` is the non-DDR configuration. Pin-n fail information is routed to the ECR using pin-n's signal path, and pin-n+4's fail information is routed to the ECR using pin-n+4's signal path. If a DDR pattern is executed, only A-cycle failures are logged.



`t_double` represents DDR operation. In this mode, using the Fail Signal MUX, B-cycle failures from one pin are routed to the ECR using the signal path of another pin, at the expense of the other pin. And, these B-cycle fails are stored in the ECR instead of the other pin's fail data. In this mode, half of the tester pins can be logged at DDR rates. The other pins can be used as input pins, or they can be strobed to affect PASS/FAIL and branch-on-error operations.

The hardware design dictates how the Fail Signal MUX links pins: pin-n and pin-n+4 i.e. pins 1/5, pins 2/6, etc.

Using `t_double`, the Fail signal MUX can be configured in two ways, based on which pin of the pin-pair is included in the pin list argument passed to the `fail_signal_mux()` function:



In the first configuration (pin-n = `t_double`) no fail information from pin-n+4 is logged. Instead, B-cycle fails from pin-n are routed to the ECR using pin-n+4's signal path. Pin-n's B-cycle fail information is stored in the ECR in place of pin-n+4's fail information. In this configuration, the `fail_signal_mux()` *getter* function for pin-n+4 returns `t_mux_opt_na`.

In the second configuration (pin-n-4 = `t_double`) no fail information from pin-n is logged. Instead, B-cycle fails from pin-n-4 are routed to the ECR using pin-n's signal path. Pin-n-4's B-cycle fail information is stored in the ECR in place of pin-n's fail information. In this configuration, the `fail_signal_mux()` *getter* function for pin-n returns `t_mux_opt_na`.

When using `t_double` it is an error when pin-pair conflicts are programmed. This occurs when, in a single call of `fail_signal_mux()`, the pin list argument contains both pin-n and pin-n+4. When this occurs, two things happen:

- A warning message is displayed in the appropriate UI site output window.
- Desired operation will not occur.

The `fail_signal_mux()` function can be executed multiple times, and the effects are cumulative. The table below documents how the Fail Signal MUX of both pins of a pin-pair (pin-n+4) are affected when the MUX on one of the pins is reprogrammed:

Table 3.19.7.4-1 Fail Signal MUX States

Change...	Paired-pin Previous Mode	Paired-pin New Mode
Pin to `t_double`	n/a	`t_mux_opt_na`
Pin to `t_single`	`t_double` `t_mux_opt_na`	`t_single`

Miscellaneous:

- The Fail Signal MUX *always* routes 2 signals to the ECR. The user is responsible for configuring the Fail Signal MUX appropriately.
- Which pins are actually captured in the ECR depends on the pin(s) specified to `lec_config_set()`; i.e. the output of the Fail Signal MUX may or may not be captured in the ECR.

Usage

```
void fail_signal_mux ( PinList* plist,
                       FailMuxSelectOpt opt );
```

where:

`plist` identifies the pins for which the Fail Signal MUX is being configured. The following rules apply:

- All pin members must be signal pins i.e. no DPS or HV pins in the pin list.
- The runtime software checks to see if any DDR *pin-pair* conflicts exist between the pins in the specified pin list. How this is handled is noted above.

`opt` identifies the desired Fail Signal MUX configuration for the specified **`plist`**. Legal options of the `FailMuxSelectOpt` enumerated. It is not legal to use the `t_mux_opt_na` value.

Examples

The following example configures the Fail Signal MUX as indicated:

- For each pin in the `data_pins` pin list, the Fail Signal MUX is configured to route both A-cycle and B-cycle failure information to the ECR. No failures will be logged for the corresponding DDR pin-pair of each pin in `data_pins`.

```
fail_signal_mux ( data_pins, t_double );
```

3.20 Pin Frequency Measurement (PFM)

Note: first available in software release h2.2.7/h1.2.7.

This section contains the following topics:

3.20.1 Overview

See Pin Frequency Measurement (PFM).

The Pin Frequency Measurement (PFM) facility documented here was originally designed to support system timing calibration. However, the features and functionality may also be useful in a customer applications.

The pin frequency measurement facility is targeted at measuring a clock waveform generated by the DUT, within the following range:

- Low limit = 49KHz.
- High limit = 113MHz

Also note:

- The waveform being measured must have a nominal 50% duty cycle. Measurement accuracy is reduced or lost as the duty cycle deviates from 50%.
- The waveform being measured must have a minimum 4.4nS high and 4.4nS low pulse width.

Three measurement modes are available which affect the measurement resolution and accuracy (more below).

3.20.2 Pin Frequency Measurement Operation

See Pin Frequency Measurement (PFM).

The Magnum 1 hardware architecture includes one frequency measurement unit for each eight timing generators. Each frequency measurement unit can measure one pin from its associated sub-sites, as shown in the following diagram. This means that one frequency measurement unit is shared by pins a_1 through a_8 AND b_1 through b_8 AND c_1 through c_8 AND d_1 through d_8.

The following diagram shows the key PFM hardware components:

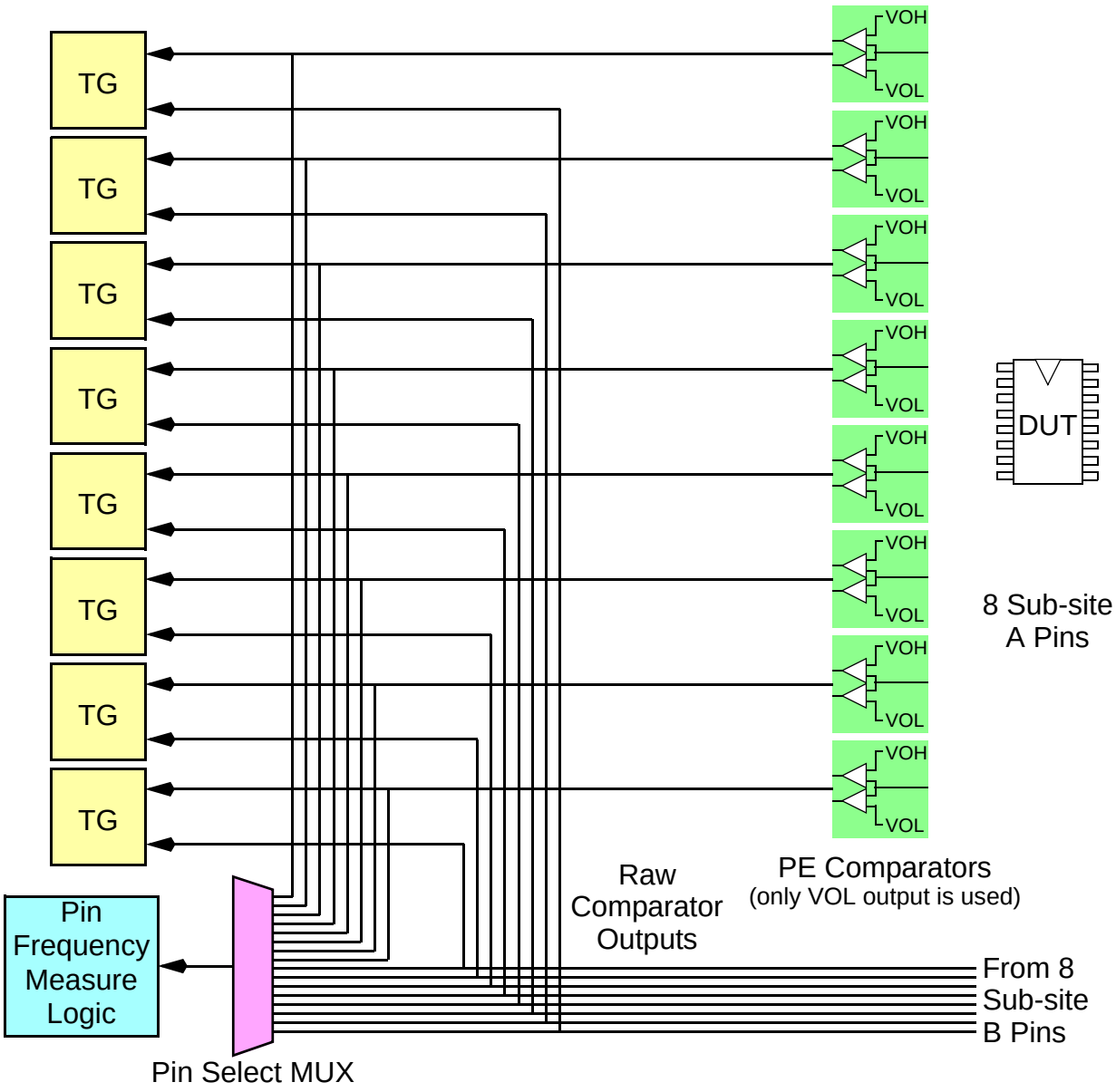


Figure-49: Frequency Measure Simplified Block Diagram

As shown below, each frequency measurement circuit consists of two counters:

- A 16-bit Reference Clock Counter which counts rising edges of the measure clock.
- A 13-bit DUT Clock Counter which enables the Reference Clock Counter and determines when a measurement is complete: more below.

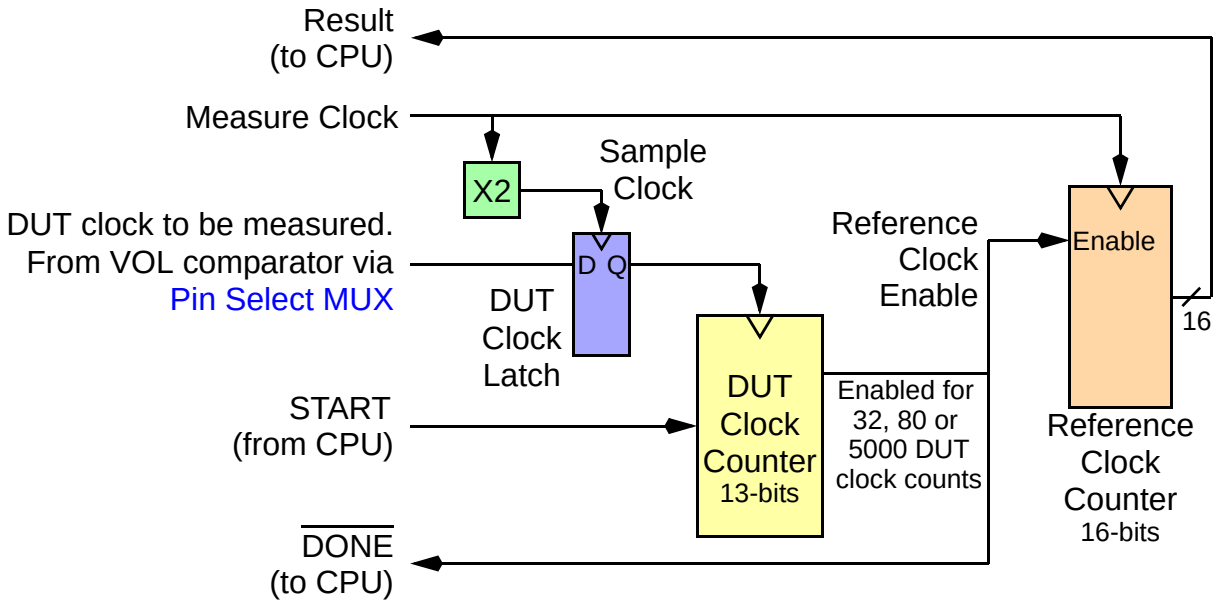


Figure-50: Frequency Measure Logic Detailed Block Diagram

Note the following:

- Executing `pin_frequency_meas()` performs the pin frequency measurement. Prior to this, the test program will typically have started the execution of a functional test pattern using `start_pattern()`. This pattern must be designed to execute continuously and generate any input waveforms required by the DUT. After the frequency measurement is complete pattern execution is terminated using `stop_pattern()`.
- Executing `pin_frequency_meas()` first clears the Reference Clock Counter and configures the DUT Clock Counter to count either 32, 80 or 5000 rising clock edges, based on the `mode` argument to `pin_frequency_meas()`.
- `pin_frequency_meas()`also routes the clock signal from the DUT to the Pin Frequency Measure Logic via the pin's VOL comparator and the Pin Select MUX (see Frequency Measure Simplified Block Diagram). The DUT waveform is then sampled at the DUT Clock Latch by the rising edge of the Sample Clock, which is

the reference clock multiplied 2X. The output of the DUT Clock Latch clocks the DUT Clock Counter, which is idle until a START signal is received from the computer.

- `pin_frequency_meas()` next sends the START signal to the DUT Clock Counter. Then, the first rising edge of the sampled DUT clock from the DUT Clock Latch increments the DUT Clock Counter and its output goes TRUE, enabling the Reference Clock Counter to begin counting rising edges of reference clocks.
- Once enabled, the Reference Clock Counter counts until either it is stopped by the DUT Clock Counter or until it reaches its maximum count (65535). The Reference Clock Counter does not roll-over after reaching the maximum count.
- During this time, each rising edge output from the DUT Clock Latch, increments the DUT Clock Counter. This repeats until 32, 80 or 5000 rising DUT clock edges have been received. At the 33/81/5001 clock edge the DUT Clock Counter output goes FALSE, which both stops the Reference Clock Counter and sends the `DONE` signal to the computer.
- If the DUT Clock Counter does not signal `DONE` before approximately 655.35uS the computer will stop both counters and the measurement result is invalid. When this occurs `pin_frequency_meas()` returns FALSE and the measured value for the related pin(s) is set = -1.
- The count value from the Reference Clock Counter is read to calculate the frequency measurement result based on the number of reference clocks counted during the time the DUT Clock Counter counted the 32, 80 or 5000 DUT clock edges. These examples presume a 100MHz reference clock (Magnum 1, Magnum 2/2x use a 110MHz reference clock):

$$\text{DUT Clock Frequency (MHz)} = 100\text{MHz} \times \frac{32}{\text{Reference Clock Count}}$$

Or...

$$\text{DUT Clock Frequency (MHz)} = 100\text{MHz} \times \frac{80}{\text{Reference Clock Count}}$$

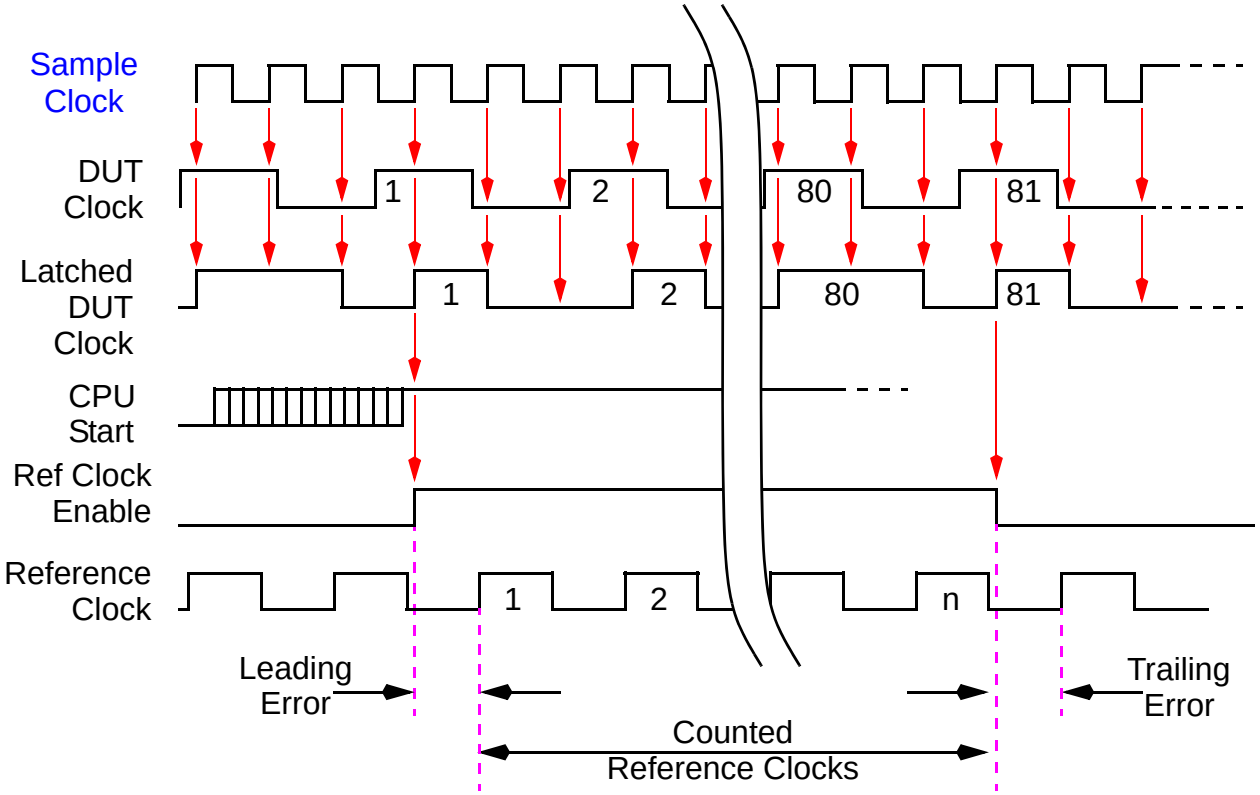
Or...

$$\text{DUT Clock Frequency (MHz)} = 100\text{MHz} \times \frac{5000}{\text{Reference Clock Count}}$$

- If the value read from the Reference Clock Counter is 65535 the measurement is considered invalid because it is not known if more than 65535 reference clocks were counted. When this occurs `pin_frequency_meas()` returns FALSE. Similarly, if the value read from the Reference Clock Counter is 0 `pin_frequency_meas()` returns FALSE. This is done separately for each pin involved in the measurement.

- For any pin which has an error, the measured value returned by `pin_frequency_meas_get()` is set = -1.
- The measured values can then be retrieved using the `pin_frequency_meas_get()` function.

The following diagram shows this operation:



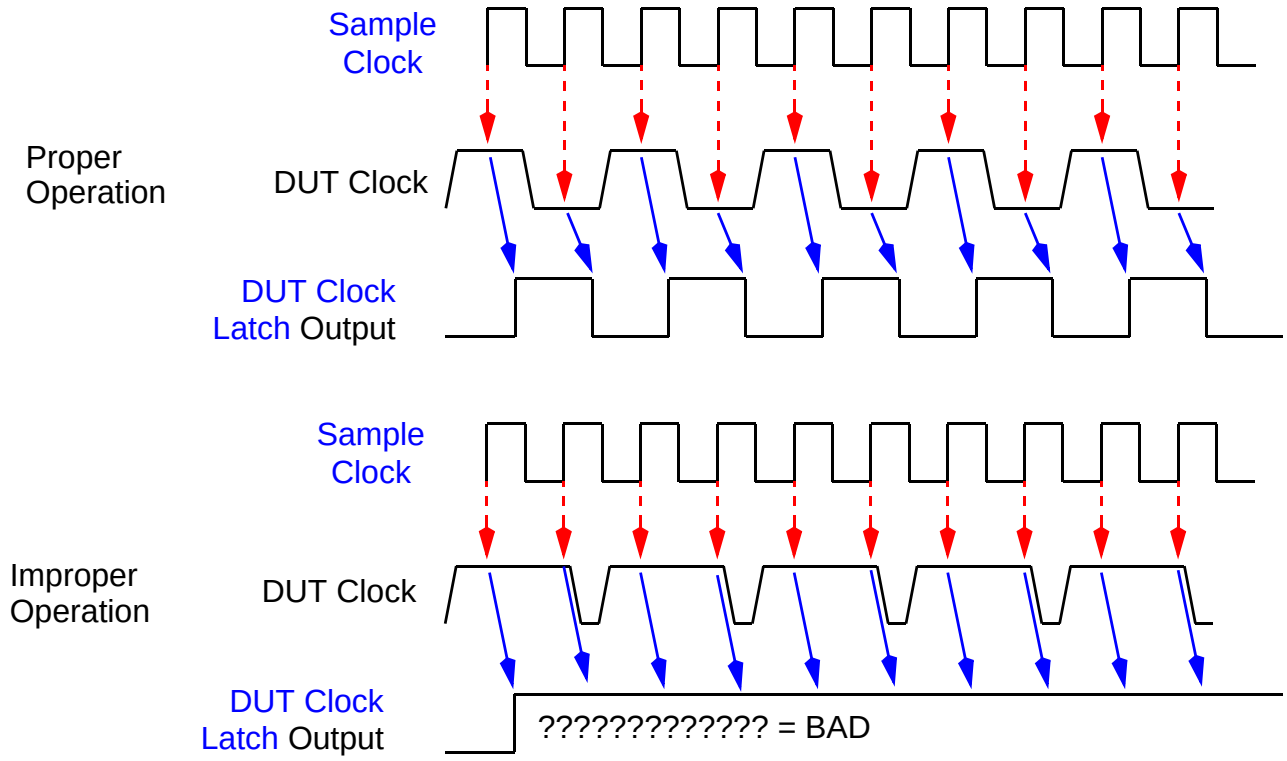
This example set the DUT Clock Counter = 80

Figure-51: Operational Timing Diagram

Because the Reference Clock Counter and DUT Clock Counter operate asynchronously, there are two errors in any measurement. In the diagram above, these are labeled *Leading Error* and *Trailing Error*: The worse-case error is ±1 reference clock. However, as shown below, this can become quite insignificant depending on the frequency of the DUT clock being measured and the measurement mode used.

A valid measurement requires that the output of the DUT Clock Latch change state for every state change of the DUT clock being measured. In effect this means that the DUT clock must be *sampled* at least once for every logic high-time and every logic low-time. If either the high or low pulse-width of the DUT clock is below the minimums noted below, the resulting measurement will not be valid. The hardware cannot detect this; i.e. it is the user's

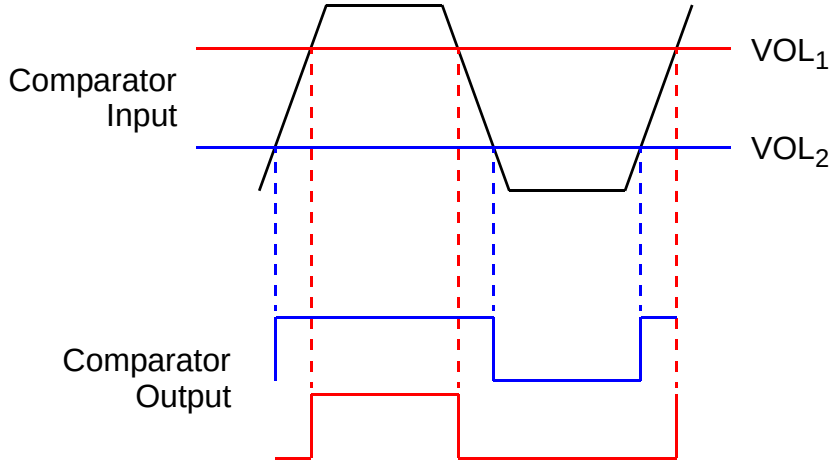
responsibility. The following two diagrams show a good example and a bad example. Note the output of the DUT Clock Latch:



The maximum frequency which can be measured is limited by one hardware feature and two issues which fall into the user's domain.

- To properly sample the DUT clock waveform requires that the minimum pulse width for both the DUT clock high-time and low-time be 4.4nS or greater. This ensures that each phase of the DUT clock being measured is sampled at least once by each reference clock.

- The user's specified VOL value also affects the pulse-widths of the DUT clock high-time and low-time at the output of the VOL comparator. The following diagram shows an exaggerated view of this effect:



  In general, optimum frequency measurements are obtained when the DUT clock waveform has 50% duty cycle and VOL is set to 50% of the actual DUT output waveform amplitude.

The minimum frequency which can be measured is limited by two factors:

- The Reference Clock Counter has a usable range of 2^{16}-2 counts (65534); i.e. up to 655.34uS count time.
- The DUT Clock Counter must count 32, 80 or 5000 DUT clocks within this 655.34uS.

Thus, using Magnum 1 (100MHz reference clock), the minimum frequency which can be measured is different for each count (below, values are rounded):

$$\left(\frac{65534}{32}\right) \times 10nS \;=\; 20479.4nS \;=\; 49KHz$$

$$\left(\frac{65534}{80}\right) \times 10nS \;=\; 8191.75nS \;=\; 123KHz$$

$$\left(\frac{65534}{5000}\right) \times 10nS \;=\; 131nS \;=\; 7.63MHz$$

As indicated above, the leading and trailing error sources can affect the overall accuracy of the measurement by up to ±1 reference clock. However, the percentage of error is dependent on both the frequency of the DUT clock being measured AND whether 32, 80 or 5000 DUT clock edges are being counted. Several examples follow. These all presume the reference clock = 100MHz (i.e. Magnum 1; Magnum 2/2x use 110MHz reference clock):

Given: 83.3MHz DUT clock, 32 clocks counted:

```
32 counts of 12nS DUT clock = 384nS count time
In 384nS the Reference Clock Counter will count 38 10nS reference clocks
The error sources make this count = 37 to 39, thus:
```

$$\text{DUT Clock Frequency (MHz)} = 100\text{MHz} \times \frac{32}{37} = 86.486\text{MHz} = 3.82\%\ \text{error}$$

$$\text{DUT Clock Frequency (MHz)} = 100\text{MHz} \times \frac{32}{39} = 82.47\text{MHz} = 1.49\%\ \text{error}$$

Given: 83.3MHz DUT clock, 80 clocks counted:

```
80 counts of 12nS DUT clock = 960nS count time
In 960nS the Reference Clock Counter will count 96 10nS reference clocks
The error sources make this count = 95 to 97, thus:
```

$$\text{DUT Clock Frequency (MHz)} = 100\text{MHz} \times \frac{80}{95} = 84.21\text{MHz} = 1.09\%\ \text{error}$$

$$\text{DUT Clock Frequency (MHz)} = 100\text{MHz} \times \frac{80}{97} = 82.47\text{MHz} = 0.96\%\ \text{error}$$

Given: 83.3MHz DUT clock, 5000 clocks counted:

```
5000 counts of 12nS DUT clock = 60000nS count time
In 60000nS the Reference Clock Counter will count 6000 10nS reference clocks
The error sources make this count = 5999 to 6001, thus:
```

$$\text{DUT Clock Frequency (MHz)} = 100\text{MHz} \times \frac{5000}{5999} = 83.347\text{MHz} = 0.05\%\ \text{error}$$

$$\text{DUT Clock Frequency (MHz)} = 100\text{MHz} \times \frac{5000}{6001} = 83.319\text{MHz} = 0.02\%\ \text{error}$$

Given: 200KHz DUT clock, 32 clocks counted:

32 counts of 5uS DUT clock = 160uS count time

In 160uS the Reference Clock Counter will count 16000 10nS reference clocks

The error sources make this count = 15999 to 16001, thus:

$$\text{DUT Clock Frequency (MHz)} = 100\text{MHz} \times \frac{32}{15999} = 200.012\text{KHz} = 0.006\% \text{ error}$$

$$\text{DUT Clock Frequency (MHz)} = 100\text{MHz} \times \frac{32}{16001} = 199.987\text{KHz} = 0.006\% \text{ error}$$

Given: 200KHz DUT clock, 80 clocks counted:

80 counts of 5uS DUT clock = 400uS count time

In 400uS the Reference Clock Counter will count 40000 10nS reference clocks

The error sources make this count = 39999 to 40001, thus:

$$\text{DUT Clock Frequency (MHz)} = 100\text{MHz} \times \frac{80}{39999} = 200.005\text{KHz} = 0.0025\% \text{ error}$$

$$\text{DUT Clock Frequency (MHz)} = 100\text{MHz} \times \frac{80}{40001} = 199.995\text{KHz} = 0.0025\% \text{ error}$$

Given: 200KHz DUT clock, 5000 clocks counted:

5000 counts of 5uS DUT clock = 25mS count time

This exceeds the maximum Reference Clock Counter count time (655.34uS).

pin_frequency_meas() will return FAIL and the measured value retrieve using pin_frequency_meas_get() will = -1.

3.20.3 Pin Frequency Measure Software

See Pin Frequency Measurement (PFM).

This section includes:

- Types, Enums, etc.
- pin_frequency_meas()

- pin_frequency_meas_get()

3.20.3.1 Types, Enums, etc.

See Pin Frequency Measurement Operation, Pin Frequency Measure Software.

The following declarations are used to support Pin Frequency Measurement (PFM)s:

The PinFreqMeasMode enumerated type is used to select the number of DUT clocks counted in subsequent frequency measurements. See pin_frequency_meas():

```
enum PinFreqMeasMode{  t_pin_freq_meas_80,
                       t_pin_freq_meas_5000,
                       t_pin_freq_meas_32 }
```

3.20.3.2 pin_frequency_meas()

See Pin Frequency Measurement Operation, Pin Frequency Measure Software.

Note: first available in software release h2.2.7/h1.2.7.

Description

The pin_frequency_meas() function is used to initiate a frequency measurement using the Pin Frequency Measure Logic.

Note the following:

- One or more pins can be measured using a single execution of pin_frequency_meas(). Pins to be measured are specified using the pPinList or pDutPin arguments.
- When the pin(s) to be measured share Pin Frequency Measure Logic the pins will be measured sequentially (one at a time).
- Only the VOL comparator output is used; i.e. the VOL setting is important to proper operation.
- It is not necessary to execute a functional test or strobe the DUT clock pin(s) being measured to make a measurement using pin_frequency_meas().

- If test pattern execution is required to stimulate the DUT during the frequency measurement:
  - The test pattern must be designed to execute continuously (endless loop).
  - The test pattern must be executing using the `start_pattern()` function.
  - After all measurements are completed the pattern execution must be terminated using the `stop_pattern()` function.
  - In Multi-DUT Test Programs, only the pins of DUTs currently in the Active DUTs Set (ADS) are actually measured.
- Measurement results can be retrieved using `pin_frequency_meas_get()`.

Usage

```
BOOL pin_frequency_meas( DutPin* pDutPin, PinFreqMeasMode mode );
BOOL pin_frequency_meas( PinList* pPinList,
                         PinFreqMeasMode mode );
```

where:

`pDutPin` identifies one pin to be measured. The specified pin must be mapped to a signal pin in the Pin Assignment Table. In Multi-DUT Test Programs, the pin(s) of all DUTs in the Active DUTs Set (ADS) are measured.

`pPinList` identifies one or more pin(s) to be measured. The specified pin(s) must be mapped to a signal pin in the Pin Assignment Table. In Multi-DUT Test Programs, the pin(s) of all DUTs in the Active DUTs Set (ADS) are measured.

`mode` specifies the number of DUT clocks counted during the frequency measurement. See Pin Frequency Measurement Operation for details. Legal values are of the `PinFreqMeasMode` enumerated type.

`pin_frequency_meas()` returns TRUE if ALL values read from ALL `Reference Clock Counter`(s) are valid (i.e. 1 to 65534) otherwise FALSE is returned. If FALSE is returned the measured value for each pin should be examined to determine which pins failed, see `pin_frequency_meas_get()`.

Example

```
vol( 1 V, meas_pins ); // Critical to measurement quality
voh( 2 V, meas_pins );
BOOL ok = pin_frequency_meas( meas_pins, t_pin_freq_meas_5000 );
CArray<double,double> meas_results;
```

```
SoftwareOnlyActiveDutIterator duts;
while ( duts.More() ) {
   pin_frequency_meas_get( meas_results); //pin_frequency_meas_get()
   DutNum dut = active_dut_get();
   int size = meas_results.GetSize(); // Get num measurements
   // Loop through measured pins and output results
   DutPin* measpin;
   CString cstr;
   for ( int pin = 0; pin < size; pin++) {
      pin_info( meas_pins, pin, &measpin);
      if( meas_results[pin] == -1 )
         output("WARNING: Invalid Measurement for DUT-%d pin %s",
                dut +1, resource_name( measpin ) );
      else{
         if ( meas_results[pin] < ( 1 MHZ ) )
            cstr.Format( "%3.3f KHz", meas_results[pin]/(1 KHZ));
         else
            cstr.Format( "%3.3f MHz", meas_results[pin]/(1 MHZ));
         output(" DUT-%d pin %s => %s",
                dut +1,
                resource_name( measpin ),
                cstr );
      }
   }
} // while
```

3.20.3.3 pin_frequency_meas_get()

See Pin Frequency Measurement Operation, Pin Frequency Measure Software.

Note: first available in software release h2.2.7/h1.2.7.

Description

The `pin_frequency_meas_get()` function is used to retrieve frequency measurements made using `pin_frequency_meas()`. Note the following:

- Since a single execution of `pin_frequency_meas()` can potentially measure multiple pins, for multiple DUTs, `pin_frequency_meas_get()` returns an array of measure results. Values in the array are ordered based on the pin list measured.
- In Multi-DUT Test Programs, values are returned for the first DUT in the Active DUTs Set (ADS).
- Measurements should be retrieved immediately after executing `pin_frequency_meas()`, to ensure subsequent measurements do not over-write and cause a loss of information.
- If `pin_frequency_meas()` returns FALSE, the measured value for each pin should be examined to determine which pin(s) failed. The value -1 is returned for pins which failed.
- Measure results are stored in base units (see Specifying Units).

Usage

```
void pin_frequency_meas_get( CArray<double, double>& data );
```

where **`data`** is a previously declared `CArray`, used to return measurement results. The array is automatically cleared and resized as necessary to store the appropriate test results. The value -1 is returned for pin(s) which failed to measure properly, see Overview and Pin Frequency Measurement Operation.

Example

See Example.

3.21 Test Patterns

See Software.

Test patterns are executed when performing the following types of tests:

Table 3.21.0.0-1 Test Pattern Applications

Test Type	Test Pattern Purpose
funtest()	Validate the logic functionality of the DUT. Validate the AC performance of the DUT under specified DC conditions. Functionally program programmable DUTs. Set up the DUT's logic state prior to executing other tests (PMU tests, DPS current, mixed signal, etc.).
ac_partest()	Perform a PMU test while concurrently executing a functional test pattern. Optionally, trigger the DC Comparators and Error Logic or DC A/D Converter from the pattern, possibly multiple times.
ac_test_supply()	Perform a DPS current test while concurrently executing a functional test pattern. Optionally, trigger the DC Comparators and Error Logic or DC A/D Converter from the pattern, possibly multiple times.
hv_ac_test_supply()	Perform a HV current test while concurrently executing a functional test pattern. Optionally, trigger the DC Comparators and Error Logic or DC A/D Converter from the pattern, possibly multiple times.
start_pattern() stop_pattern() restart()	Supports pattern looping while user code continues to execute, for example, to use Pin Frequency Measurement (PFM)..

The test pattern supplies the logic state information (the 1's and 0's) needed to functionally exercise the DUT.

The Magnum 1 test system contains two sources of test pattern data:

- APG

- Combined Logic Vector Memory (LVM) / Scan Vector Memory (SVM) for stored Logic Test Patterns and Scan Test Patterns.

These are described in the following table:

Table 3.21.0.0-2 Test Pattern Data Sources

Source	Comments
APG	For testing memory devices. Memory Test Patterns are used to algorithmically generate logic state information in the form of X/Y Addresses, read/write data, and chip selects. The APG also has a Data Buffer Memory (DBM) for storing random read/write data while generating algorithmic addresses and chip selects.
The combined Logic Vector Memory (LVM) / Scan Vector Memory (SVM)	Magnum 1/2/2x use the same memory to store both Logic Test Patterns and Scan Test Patterns.

During test pattern execution, each pin-pair's (see Functional Pin-pairs) data source can be selected cycle-by-cycle using Pin Scramble Maps.

Note that within a given test pattern these data sources can be used individually (see Memory Test Patterns, Logic Test Patterns, Scan Test Patterns), or combined to generate Mixed Memory/Logic Patterns, to test devices containing both memory and/or random logic and/or Scan logic (i.e. *SOC*: System on a Chip, etc.).

In addition to supplying logic state information, the test pattern also provides the following controls:

Table 3.21.0.0-3 Test Pattern Control Applications

Control	Comment
Test Pattern Sequence Control Instructions	Controls the test pattern execution engine. Logic patterns, scan patterns, and memory patterns each use different pattern instructions.Using Magnum 1/2/2x, the APG has two control engines (MAR Engine and VAR Engine) used to separately control logic vs. memory pattern execution.
Time-sets (TSET) Selection	Per-cycle selection of cycle period, and per-pin drive format/timing, strobe timing and I/O timing.
VIHH Maps Selection	Per-cycle selection of which pins are driven to the VIHH level.
Pin Scramble Map Selection	Per-cycle selection of the pattern data source for each pin-pair.
Strobe Control	Per-pin/per-cycle strobe enable/disable using READ or READUDATA instructions (Memory Test Patterns) or Logic Vector Bit Codes (Logic Test Patterns and Scan Test Patterns).
I/O Control	Per-pin/per-cycle drive/tri-state control using the ADHIZ instruction (Memory Test Patterns) or Logic Vector Bit Codes (Logic Test Patterns and Scan Test Patterns).
Trigger DC Comparators and Error Logic or DC A/D Converter	Per-cycle. Using the MAR VCOMP instruction (Memory Test Patterns) or VEC/RPT VCOMP, VAR VCOMP, and VPINFUNC VCOMP instructions (Logic Test Patterns). The test pattern can trigger the DC Comparators and Error Logic or DC A/D Converter to make one or more Go/NoGo comparisons or measurements.

Table 3.21.0.0-3 Test Pattern Control Applications *(Continued)*

Control	Comment
Set or tweak PE levels (VIL, VIH, VOL, VOH, VIHH, VTT, VZ, etc.)	See Controlling PE Levels from the Test Pattern.
Set or tweak DPS, PMU, HV levels	See Controlling PE Levels from the Test Pattern.
Change DPS output voltage selection	The test pattern can cause the DPS to switch between 2 previously programmed voltages, using the PINFUNC VPULSE instruction (Memory Test Patterns) or VEC/RPT VPULSE, VAR VPULSE, and VPINFUNC VPULSE instructions (Logic Test Patterns).

Test pattern source files are created using the same methods used to create C-code source files; i.e. a text editor.

Test patterns are written using a unique pattern language (source code statements), which is documented in Test Pattern Programming, which is divided into three sections:

- Memory Test Patterns
- Logic Test Patterns
- Scan Test Patterns

3.22 Functional Tests

This section includes the following topics:

- Executing Functional Tests
  - Per Pin Error Status
  - Pattern Execution Start Vector, Stop Vector
- Patterns That Loop Forever
- Checking Pattern Execution State
- Stopping Pattern Execution
- Restarting Paused Patterns
- Testing for Stopped/Paused Patterns

- Holding State Between Patterns

3.22.1 Executing Functional Tests

See Functional Tests.

Description

The `funtest()` function is used to execute a test pattern to exercise and test the DUT under AC conditions.

Test patterns are automatically loaded to hardware during test program initialization. For special cases (rarely needed) the methods documented in the Resources section can be used.

A test pattern is identified as a `Pattern*` argument to the `funtest()` function, to specify which test pattern to execute. Or, given the name of a pattern, the `Pattern_find()` function can be used to get a `Pattern*`.

Various Pattern Execution Stop Condition Options exist which are used to control pattern execution. The most common options either cause the entire pattern to execute (`finish`) or halt execution when the first failure is detected (`error`).

Using Magnum 1/2/2x, it is possible to execute a subset of logic vectors. See Pattern Execution Start Vector, Stop Vector.

The `start_pattern()` function is used to execute Patterns That Loop Forever; i.e. indefinitely. The `stop_pattern()` function is used to halt these test patterns. Note that `funtest()` also uses `stop_pattern()`, in a mode which executes the specified test pattern one time.

The following outline documents the basic operation of `funtest()`:

- Execute `start_pattern( pPattern, Condition)`. See `start_pattern()`.
- Wait for pattern execution to stop.
- Evaluate PE error latches (or Logic Error Catch (LEC), if used) to determine overall PASS/FAIL.
- Return the PASS/FAIL result.

Usage

```
PFState funtest( Pattern *pPattern, PatStopCond Condition );
Also see Pattern Execution Start Vector, Stop Vector.
```

where:

`funtest()` returns TRUE (PASS) or FALSE (FAIL). All pin(s) tested must PASS otherwise FAIL is returned. In Multi-DUT Test Programs, only DUT(s) in the Active DUTs Set (ADS) can affect test results.

`pPattern` identifies the test pattern to be executed.

`Condition` specifies pattern execution stop option. Legal values are of the `PatStopCond` enumerated type and described in the following table:

Table 3.22.1.0-1 Pattern Execution Stop Condition Options

Stop Condition	Summary Description
`finish`	Execute pattern to completion, regardless of errors. When execution finishes, the PE error latches and DC Error Flags are examined. If an error was latched, the result of `funtest()` is FAIL, otherwise the result is PASS
`error`	Stop pattern execution on first functional or DC Error Flag error and set the result of `funtest()` to FAIL. Note that the pattern generator may continue for one or more cycles past where the error occurred, depending on the cycle time and where in the cycle the error was detected.
`fullec`	Execute pattern to completion. Enable full ECR, row error catch, and column error catch to capture errors during pattern execution. This argument should be used when performing Redundancy Analysis (RA) or using BitmapTool.
`LEC_only_errors`	Enable full ECR, row error catch, and column error catch. Capture the first 2Meg (2^{21}-6) failing vectors. Intended for use when the ECR is used as an Logic Error Catch (LEC).
`LEC_first_vectors`	Enable full ECR, row error catch, and column error catch. Capture the first 2Meg (2^{21}-6) vectors executed. Ignores PASS/FAIL. Intended for use when the ECR is used as an Logic Error Catch (LEC).

Table 3.22.1.0-1 Pattern Execution Stop Condition Options *(Continued)*

Stop Condition	Summary Description
LEC_last_vectors	Enable full ECR, row error catch, and column error catch. Capture the last 2Meg (2^{21}-6) vectors executed. Ignores PASS/FAIL. Intended for use when the ECR is used as an Logic Error Catch (LEC).
LEC_before_error	Enable full ECR, row error catch, and column error catch. Capture the first failing vector plus the previous 2Meg (2^{21}-6) vectors executed. Intended for use when the ECR is used as an Logic Error Catch (LEC).
LEC_after_error	Enable full ECR, row error catch, and column error catch. Capture the first failing vector plus the next 2Meg (2^{21}-6) vectors executed. Intended for use when the ECR is used as an Logic Error Catch (LEC).
LEC_center_error	Enable full ECR, row error catch, and column error catch. Capture the first failing vector plus up to 512K vectors executed before the failure and up to 512K vectors executed after the failure. Intended for use when the ECR is used as an Logic Error Catch (LEC).
Note: in parallel test applications, the test pattern must be executed to completion, to ensure that DUT(s) which don't fail are completely tested. In other words, halting the pattern early (error) because one or more DUT(s) failed prevents DUT(s) which PASS from being completely tested. This is BAD.	

Examples

The following example executes the myPat pattern to completion:

```
PFState r = funtest( myPat, finish );
```

The following example executes the minmax pattern, stopping the pattern if any errors are detected:

```
PFState r = funtest( minmax, error );
```

3.22.1.1 Per Pin Error Status

See Functional Tests.

Description

The `test_pin()` function is used determine whether any pins in a specified pin list failed the most recent `funtest()` execution.

The `test_pin_first_error()` function is used determine whether any pins in a specified pin list failed in the first failing cycle of the most recent `funtest()` execution. Proper `test_pin_first_error()` operation requires `funtest()` be executed using the `error` option.

Both functions should be executed immediately after `funtest()`; i.e. before any other test functions are executed.

Usage

```
PFState test_pin( PinList* pPinList );
PFState test_pin_first_error( PinList* pPinList );
PFState test_pin( DutPin *pDutPin );
PFState test_pin_first_error( DutPin *pDutPin );
PFState test_pin_first_errorA( DutPin *pDutPin );
PFState test_pin_first_errorB( DutPin *pDutPin );
```

where:

`pPinList` identifies the pins to be checked for failures. In Multi-DUT Test Programs, this represents the same pins of ALL DUT(s) in the Active DUTs Set (ADS); i.e. the specified pins of all DUT(s) in the ADS are polled to determine the value returned by `test_pin()`.

`pDutPin` identifies one pin to be checked for failures. In Multi-DUT Test Programs, this represents the same pins of ALL DUT(s) in the Active DUTs Set (ADS); i.e. the specified DutPin(s) of all DUT(s) in the ADS are polled to determine the value returned by `test_pin()`.

`test_pin()` returns PASS if all pins represented by **`pPinList`** or all pins represented by `pDutPin` passed, otherwise `FAIL` is returned.

`test_pin_first_error()` returns PASS if none of the pins represented by **`pPinList`** or the none of the pins represented by `pDutPin` failed in the first failing cycle of the most recently executed `funtest()`.

Example

In the following example, after the `funtest()` execution completes, `test_pin()` is called to see if any pins in the pin list named `pl_all_data` failed. Remember, this represents the same pins for all DUTs in the Active DUTs Set (ADS). If there were failures, then the user-written function named `my_routine()` is called:

```
PFState test_result = funtest ( patname, finish );
if ( test_pin( pl_all_data ) == FAIL )
   my_routine();
```

In the following example, after the `funtest()` execution completes, `test_pin_first_error()` is called to see if any pins in the pin list named `pl_all_data` failed in the first failing cycle of `myPat`. If it did, an output message is printed:

```
PFState test_result = funtest ( myPat, error );
if ( test_pin_first_error ( pl_all_data ) == FAIL )
   output (" One or more pins in pl_all_data failed in the first
             failing cycle of myPat" );//Delete prev EOL to compile
```

3.22.1.2 Pattern Execution Start Vector, Stop Vector

See Functional Tests.

Description

Using the `funtest()` and `start_pattern()` functions, it is possible to execute a subset of a Logic Test Patterns, by specifying a start vector and stop vector.

The following rules apply:

- The start and stop vector values are zero based, and relative to the start of the specified test pattern; i.e. the first vector of a given pattern is always 0.
- In DDR Test Patterns, the start vector must be an even number and the stop vector must be an odd number.

- Any attempt to execute past the last vector of a test pattern is treated as a fatal error; i.e. the test program is unloaded. To identify the number of vectors in a pattern the `addrs()` function can be used. See example.
- When a stop vector is specified, the system software temporarily modifies the pattern instruction for the stop vector to include `VAR DONE`. At the end of pattern execution, the VAR Engine loops on the `VAR DONE` instruction to flush the hardware pipelines. This is identical to all logic pattern execution operation, where the last vector is repeated while the hardware pipelines are flushed. The system software then restores the modified pattern instruction back to the original compiled-in instruction.
- Using start/stop vector, it is possible to execute one vector:
  - For non-DDR Test Patterns, set `StopVector = StartVector`
  - For DDR Test Patterns set `StopVector = (StartVector +1 )`
  Note that this vector will be executed multiple times. See previous bullet regarding flushing the hardware pipeline.

The functions shown below are the only functions which support using the start/stop vector mechanism.

Usage

Note: `funtest()` is documented in detail in Executing Functional Tests.
`start_pattern()` is documented in Patterns That Loop Forever.
The functions below are specialized to support the start/stop vector facility.

```
PFState funtest(  Pattern *pPattern,
                  PatStopCond Condition,
                  int StartVector,
                  int StopVector );
void start_pattern( Pattern *pPattern,
                    int StartVector,
                    int StopVector );
void start_pattern( Pattern *pPattern,
                    PatStopCond Condition,
                    int StartVector,
                    int StopVector );
```

where:

`pPattern` identifies the pattern to be executed.

Condition specifies the pattern stop option. Details are documented for `funtest()`.

StartVector is the first vector to be executed. The first vector = 0. See Description for additional rules.

StopVector is the last vector to be executed. See Description.

Example

The following example executes the entire test pattern using the new mechanism. This example is used to show usage of the `addrs()` function:

```
DWORD mar, var, mLen, vLen;
BOOL ok = addrs( myLogicPat, &mar, &var, &mLen, &vLen );
if( !ok ) output("ERROR: addrs() returned FALSE, check Pattern*");
PFState r = funtest( myLogicPat, finish, 0, (vLen -1) );
```

Note that the logic pattern length returned by `addrs()` (`vLen`) is the true length of the pattern whereas a stop vector value is zero-based, thus the use of (`vLen` -1) in the example.

3.22.2 Patterns That Loop Forever

See Functional Tests.

Description

The `start_pattern()` function can be used to execute a test pattern which is designed to loop indefinitely. Once execution is started the test program C-code continues to execute immediately.

Test patterns are normally executed using `funtest()`, and execution is controlled by instructions in the pattern; i.e. the pattern `MAR DONE` instruction halts execution. Or, depending on a specified Pattern Execution Stop Condition Options, execution may halt earlier, for example if an failure is detected.

If a test pattern containing an infinite loop (by design or by accident) is executed using `funtest()` the execution will never end. The test program will *hang* because the test site controller waits for the APG to signal when execution has halted, which never happens.

To intentionally execute a pattern containing an infinite loop the `start_pattern()` function can be used. Once execution begins, user-written C-code then proceeds to execute

without waiting for the test pattern to complete. The `stop_pattern()` function is used to terminate pattern execution.

Note that `funtest()` also executes `start_pattern()` in a mode which executes the specified test pattern one time.

`start_pattern()` also supports executing a subset of a logic pattern, see Pattern Execution Start Vector, Stop Vector.

The following outline documents the basic operation of `start_pattern()`:

- Stop the APG MAR Engine and VAR Engine if running; i.e. halt any test pattern currently looping.
- Set up hardware modes based on pattern attributes.
- Update timing in the hardware (if it has changed).
- If `hold_pattern_state()` is TRUE, modify the `builtin_pipe_clear` and `builtin_pipe_clear2` patterns to use the last pattern instruction executed.
- If the combined Logic Vector Memory (LVM) / Scan Vector Memory (SVM) is installed point to the `builtin_zero_scan` pattern.
- Clear the Stop-on-Abort and Stop-on-Error flags.
- If the test pattern sets or modifies PE voltages (see Controlling PE Levels from the Test Pattern) the voltage values specified are stored in APG hardware.
- Modify the last instruction of the `pipe_clear` pattern to point to the 1st instruction of the user's test pattern.
- Execute the `pipe_clear` pattern. Stop before user's pattern executes.
- Execute pattern initial conditions C-code.
- Single-step the APG seven more times to stage the user's pattern at the DUT.
- Reset PE/APG error latches.
- Clear the APG error pipeline.
- For `ac_partest()`, and `hv_test_supply()` if `measure()` = FALSE and `vcomp` is not specified enable the DC Comparators and Error Logic.
- Set pattern execution stop condition in hardware.
- Start the pattern generator (execute the user's test pattern).

Usage

```
void start_pattern( Pattern *pPattern );
void start_pattern( Pattern *pPattern, PatStopCond Condition );
```

Also see Pattern Execution Start Vector, Stop Vector.

where:

pPattern identifies the test pattern to execute.

PatStopCond is an optional argument that specifies stop conditions. See Pattern Execution Stop Condition Options.

Example

In the following example, the pattern named `endless_pat` is set running and execution continues with the next statement following `start_pattern()`.

```
start_pattern( endless_pat );
execute_this_without_waiting_for_the_pattern_to_complete();
stop_pattern();
```

3.22.3 Checking Pattern Execution State

See Functional Tests.

Description

The `pattern_state()` function returns one of the values noted below based on the execution state of the last test pattern executed using `funtest()` or `start_pattern()`.

Pattern State	Description
PATTERN_RUNNING	A pattern is currently executing
PATTERN_PAUSED	The pattern has stopped after executing the MAR PAUSE instruction (Memory Test Patterns) or VAR PAUSE instruction (Logic Test Patterns)
PATTERN_DONE	Pattern execution has completed
PATTERN_STOPPED	Pattern execution was stopped by calling the `stop_pattern()` function.

Usage

```
PatternState pattern_state();
```

where

`pattern_state()` returns a value of the enumerated type. See Description.

Example

```
if( pattern_state() == PATTERN_PAUSED ) restart();
```

3.22.4 Stopping Pattern Execution

See Functional Tests.

Description

The `stop_pattern()` function is used to stop a test pattern which is started with `start_pattern()`.

Test patterns are sometimes intentionally designed to execute endlessly (see Patterns That Loop Forever), so that user-written C-code can execute while the pattern is looping. To properly control these patterns the `start_pattern()` function is used to begin executing the pattern and `stop_pattern()` is used to stop pattern excecution.

Usage

```
void stop_pattern();
```

Example

```
start_pattern( endless_pat ); // See start_pattern()
execute_this_without_waiting_for_the_pattern_to_complete();
stop_pattern();
```

3.22.5 Restarting Paused Patterns

See Functional Tests.

Description

Within a test pattern, execution can be paused using the MAR PAUSE instruction (Memory Test Patterns) or VAR PAUSE (Logic Test Patterns) instruction. This allows user C-code to execute before restarting the pattern from where it was paused. See Testing for Stopped/Paused Patterns and Checking Pattern Execution State.

The `restart()` function can be used to resume pattern execution from where it was previously paused. `restart()` returns control to the test program immediately, while the pattern is executing, allowing C-code to immediately continue to execute.

The `restart_and_wait()` function will restart a paused pattern but NOT return control to the test program until the pattern ends or execution reaches another MAR PAUSE instruction (Memory Test Patterns) or VAR PAUSE (Logic Test Patterns) instruction.

Both functions support an optional argument which allows the original Pattern Execution Stop Condition Options to be changed when pattern execution is restarted.

Beginning in software release h1.1.23, the `restart_and_wait()` function updates fail data as displayed via UI's FrontPanelTool and as retrieved using `results_get()`.

A special rule applies when the test pattern being restarted uses the Data Buffer Memory (DBM) as an APG data source. See page 1444.

Usage

```
void restart();
void restart_and_wait();
void restart( PatStopCond Condition );
void restart_and_wait( PatStopCond Condition );
```

where **`Condition`** specifies the desired pattern execution stop option. See Pattern Execution Stop Condition Options.

Example

In the following example, the pattern `pausing_pattern`, containing a single MAR PAUSE instruction, is executed using `funtest()`. When the pattern pauses, the test program execution continues by executing the user-written C-function called `my_special_code()`. After this function returns, the pattern is restarted using `restart()`, and the test program immediately continues to execute the user-written `more_my_special_code()` function.

```
result = funtest( pausing_pattern, error );
```

```
if( pattern_paused() ) {
   my_special_code();
   restart();
   more_my_special_code();
}
```

3.22.6 Testing for Stopped/Paused Patterns

See Functional Tests.

Description

The `pattern_paused()` function will report if a test pattern is currently executing.

> Note: `pattern_paused()` does not distinguish between a pattern which is paused using the `MAR PAUSE` instruction (Memory Test Patterns) or `VAR PAUSE` (Logic Test Patterns) instruction vs. a pattern which has stopped for other reasons (completed execution, stopped on error, etc.). Use the `pattern_state()` function to evaluate whether a pattern is running, `PAUSE`'ed, `DONE`, or stopped.

Usage

```
BOOL pattern_paused();
```

where the `pattern_paused()` returns FALSE if the pattern generator is currently executing a pattern, otherwise TRUE is returned.

Example

This example checks to see if a pattern is executing, and if so, stops the pattern.

```
if( ! pattern_paused() ) stop_pattern();
```

3.22.7 Holding State Between Patterns

See Functional Tests.

Description

The `hold_pattern_state()` function is used to ...

Note: starting with software release h2.3.xx, the `hold_pattern_state()` function has no effect on Magnum 1, Magnum 2 and Magnum 2x. Starting with this release, hardware in these systems is used to always inhibit PE state changes between the execution of the last user-instruction in one test pattern and the first user-instruction executed in the next pattern; i.e. executing `hold_pattern_state()` is silently ignored. Earlier documentation for this function described how using `hold_pattern_state()` would cause changes to some built-in test patterns and described how to restore those patterns. These patterns are no longer modified.

3.23 Manipulating Tester Hardware

This section contains the following:

3.23.1 Types, Enums, etc.

See Manipulating Tester Hardware.

Description

The following enumerated types are used in support of various functions documented in this section.

Usage

The `PEDriverState` enumerated type is used to set a static driver state. See `pin_dc_state_set()`:

```
enum PEDriverState { t_vil, t_vih, t_vihh, t_tristate }
```

The `ApgReloadRegMode` enumerated type is used to select an alternate mode of operation for the APG counter RELOAD operand. See APG Reload Register Mode Functions:

```
enum ApgReloadRegMode{ t_reload_mode1, t_reload_mode2 };
```

The ChipSelectMode enumerated is used to modify the CHIPS instruction of a pattern . See set_chip_select():

```
enum ChipSelectMode {  t_cs_false,
                       t_cs_pulse_true,
                       t_cs_pulse_false,
                       t_cs_true,
                       t_cs_na };
```

The VectorState enumerated is used as a read or write values when accessing logic test vectors in Logic Vector Memory (LVM), using vecdata():

```
enum VectorState {drive_lo = 0,
                  drive_hi = 1,
                  strobe_valid = 2,
                  strobe_mid = 3,
                  tristate = 4,
                  strobe_lo = 6,
                  strobe_hi = 7,
                  VectorState_na };
```

3.23.2 Setting DUT Pin State

Description

The setpin() function can be used to set one or more signal pins to a static logic state, without executing a test pattern. The Pin DC Static State Functions are available to perform similar but more versatile operations. Note the following:

- The static state of the specified pin(s) can be set to drive-0 (VIL), drive-1 (VIH), drive VIHH, or tri-state.
- The effect of setting VIHH or tri-state is affected by the currently set PE driver mode (see Magnum PE Driver Modes).
- Using setpin(), both pins of a Functional Pin-pair are set to the same state; the most recently programmed pin of each pin-pair sets the state for both pins. The pin_dc_state_set() function allows each pin of a pin-pair to be set to a different state.

- The pin state set using `setpin()` remains in effect until:
  - `setpin()` is executed again to change the state.
  - A functional test pattern executes and changes the state.
  - `pin_dc_state_set()` is executed to change the state.
- If a given pin is permanently configured in drive-only or receive-only mode (using the `drive_only()` or `tri_state()` functions) the `setpin()` function has no affect on the I/O state of that pin.

Usage

```
void setpin( PinList* pPinList, VRailLevel Level );
void setpin( PinList* pPinList, PinLevel Level );
```

where:

`pPinList` identifies the pin(s) to be programmed.

`Level` is one of `VIH`, `VIL`, `VIHH`, or `VZ`, as noted in the table below:.

Table 3.23.2.0-1 PE Static Drive States

State	Description
VIL	Set the pin(s) to drive VIL
VIH	Set the pin(s) to drive VIH
VIHH	Set the pin(s) to drive VIHH*
VZ	Set the pin(s) to tri-state*
* The tri-state voltage is determined by the currently set PE driver mode (see Magnum PE Driver Modes).	

Example

In the following example, the pins identified in the pin list named `input_pins` are set to drive the `VIH` level.

```
setpin( input_pins, VIH );
```

3.23.3 Pin DC Static State Functions

Description

The `pin_dc_state_set()` function is used to set one or more pins to a static DC state.

The `pin_dc_state_get()` function is used to get the current static DC state for one specified pin.

Note the following:

- `pin_dc_state_set()` controls the PE driver on the specified pin(s), setting it to either drive low (VIL), drive high (VIH), drive super-voltage (VIHH) or to tri-state.
- Using `pin_dc_state_set()`, when a given pin is set to VIL or VIH the resulting PE driver output voltage will be that programmed last using `vil()` or `vih()`. When set to drive super-voltage (VIHH) the resulting PE driver output voltage will be that programmed last using `vihh()`.
- Using `pin_dc_state_set()`, when a given pin is set to tri-state, the resulting PE driver output voltage will be determined by the current PE driver mode setting (see Magnum PE Driver Modes and `pe_driver_mode_set()`) and one of `vihh()`, `vz()` or `vtt()`. And, if the current PE driver mode is Vz Mode , the termination resistance last selected using `rl_set()` will be used.
- The `hold_state` argument determines whether the specified pin(s) will subsequently receive drive edges from the timing system.
  - TRUE means that the pin will NOT receive drive edges from the timing system during subsequent functional test executions i.e. the PE driver's DC state selected will remain in effect independent of any functional test executions.
  - FALSE allows the pin to receive drive edges from the timing system i.e. The PE driver will respond normally to subsequent functional test executions. The pin will remain in the programmed state until reprogrammed or until a test pattern executes and changes the state. Note that the when FALSE is ineffect that the `state` value returned by `pin_dc_state_get()` is invalid (the returned `state` variable is not modified).
- During initial program load, the `hold_state` state of all pins is set to FALSE. The system software does not otherwise change this state (including when Sequence & Binning Table execution ends).

- Using `pin_dc_state_set()`, once a given pin has been set to `hold_state` = TRUE, the primary method to restore normal operation is to execute `pin_dc_state_set()` again, with `hold_state` = FALSE. However, a test pattern can also use the same underlying hardware to dynamically change the static drive state of selected pins. See Setting a Static Pin-state using Level Sets.
- In Multi-DUT Test Programs, the Active DUTs Set (ADS) manipulations use the same hardware mechanism to disable drive edges to a given DUT as is used by `hold_state`.
- The currently selected output voltage may be manipulated from an executing test pattern, see Controlling PE Levels from the Test Pattern. In Multi-DUT Test Programs, this will be inhibited if the pin is associated with a DUT not currently in the Active DUTs Set (ADS).

Usage

```
void pin_dc_state_set( DutPin *dutpin,
                       PEDriverState dc_state,
                       BOOL hold_state DEFAULT_VALUE( FALSE ) );
void pin_dc_state_set( PinList* pinlist,
                       PEDriverState dc_state,
                       BOOL hold_state DEFAULT_VALUE( FALSE ) );
BOOL pin_dc_state_get( DutPin *dutpin, PEDriverState *state );
```

where:

`dutpin` is used in two contexts:

- In the setter function, identifies one pin to be programmed. In Multi-DUT Test Programs, the pin(s) of each DUT currently in the Active DUTs Set (ADS) are affected. The specified `DutPin` must be mapped to a signal pin in the Pin Assignment Table.
- In the getter function, identifies one pin to be read.

`pinlist` specifies which pin(s) are to be programmed. In Multi-DUT Test Programs, only pin(s) of DUT(s) in the Active DUTs Set (ADS) are affected. The specified **`pinlist`** must only contains pins mapped to a signal pins in the Pin Assignment Table.

`dc_state` specifies the desired PE driver state. Legal values are of the `PEDriverState` enumerated type.

`hold_state` is optional, and if specified, determines whether the pin(s) will subsequently receive waveform edges from the timing system. See Description. Default = FALSE.

state is a pointer to an existing PEDriverState variable used to return the current PE driver state, but **state** is only modified when pin_dc_state_set() was executed with **hold_state** = **TRUE**.

pin_dc_state_get() returns TRUE if the specified pin's **hold_state** is TRUE otherwise FALSE is returned. In Multi-DUT Test Programs, the value is retrieved from the first DUT in the Active DUTs Set (ADS).

Example

```
pin_dc_state_set( D0, t_vil, FALSE );
pin_dc_state_set( myPins, t_tristate, TRUE );
PEDriverState curstate;
BOOL holding = pin_dc_state_get( D0, &curstate );
```

3.23.4 Setting DUT Address Pins State

Description

The set_address() function may be used to statically set an X/Y address at the output of selected pins.

When testing memory devices, a particular address is sometimes desired at the DUT in a static state. This can be accomplished by writing and executing an appropriate test pattern or, more simply, using set_address() in the test program.

Using set_address(), the specific pins affected are determined by several things:

- The values set using numx() and numy(), to determine the size of the DUT address.
- The x_fast_axis(), to determine whether X or Y address bits are the least significant address axis.
- Pin Scramble Map PS1 is used to determine which pins are scrambled to APG address generator outputs.

The address written to the DUT pins is the *topologically scrambled* version of the address specified. See Logical vs. Physical, vs. Electrical Addresses.

This operation sets a bit in a PE driver register; i.e. it does not affect APG pipelines or APG registers.

The voltage appearing at the DUT pins is set by `vil()`, `vih()` or `vihh()`.

Usage

```
void set_address( UINT Value );
```

where:

`Value` specifies the address to be written. `value` specifies a logical address value; the TOPO scrambled address is read from the hardware Address Topological Scramble RAMs and written to the DUT.

Example

In the following example, pins scrambled to APG Address Generator outputs in Pin Scramble Map PS1 are set. Which pins are set to logic-1 vs. logic-0 depends on `numx()`, `numy()` and `x_fast_axis()`:

```
set_address( 0xffff );
```

3.23.5 Setting DUT Data Pins States

Description

The `set_data()` function may be used to statically set a data value at the output of selected pins.

When testing memory devices, a particular data value is sometimes desired at the DUT in a static state. This can be accomplished by writing and executing an appropriate test pattern or, more simply, using `set_data()`.

Pin Scramble Map PS1 is used to determine which pins will be affected; i.e. only pins which are scrambled to APG Data Generator outputs in PS1 will be affected.

This operation sets a bit in a PE driver register; i.e. it does not affect APG pipelines or APG registers.

The voltage appearing at the DUT pins is set by `vih()`, `vil()`, and `vihh()`.

Usage

```
void set_data( __int64 Value );
```

where:

`Value` specifies the data to be written. Only the low 36 bits are used (consistent with the size of the APG Data Generator.

Example

In the following example, pins scrambled to APG Data Generator outputs in Pin Scramble Map PS1 are set. The low 16 bits (`t_d0` to `t_d15`) are set to logic-1 (0xFFFF), the rest to logic-0.

```
set_data( 0xFFFF );
```

3.23.6 Setting DUT Chip Select Pin States

Description

The `set_chips_on()` function may be used to statically set the state of DUT chip-select pins at the output of selected pins.

When testing memory devices, a particular static state is sometimes desired at DUT chip select pins. This can be accomplished by writing and executing an appropriate test pattern or, more simply, using `set_chips_on()`.

Pin Scramble Map PS1 is used to determine which pins will be affected; i.e. only pins which are scrambled to APG Chip Select outputs `t_cs1` to `t_cs8` in PS1 will be affected.

This operation sets a bit in a PE driver register; i.e. it does not affect APG pipelines or APG registers.

The voltage appearing at the DUT pins is set by `vih()`, `vil()`, and `vihh()`.

Usage

```
void set_chips_on(  TesterFunc Func1 = t_tf_na,
                    TesterFunc Func2 = t_tf_na,
                    TesterFunc Func3 = t_tf_na,
                    TesterFunc Func4 = t_tf_na,
                    TesterFunc Func5 = t_tf_na,
                    TesterFunc Func6 = t_tf_na,
                    TesterFunc Func7 = t_tf_na,
                    TesterFunc Func8 = t_tf_na );
```

where:

Func1 through **Func8** identify which Chip Selects are used; only pins which are scrambled to these Chip Selects in Pin Scramble Map PS1 will be affected. Legal values are `t_cs1`, `t_cs2`, `t_cs3`, `t_cs4`, `t_cs5`, `t_cs6`, `t_cs7`, and `t_cs8`. Only the DUT pins scrambled to the chip selects that appear as arguments in this function are set to their logical TRUE state. Pins which are scrambled to other chip selects are set to their logical FALSE state.

Example

The following example sets the pins scrambled to `t_cs1`, `t_cs4` and `t_cs5` in Pin Scramble Map PS1 to logical TRUE. Pins which are scrambled to `t_cs2`, `t_cs3`, `t_cs6`, `t_cs7`and `t_cs8` are set to logical FALSE:

```
set_chips_on( t_cs1, t_cs4, t_cs5 );
```

3.23.7 Memory-pattern Related Functions

See Manipulating Tester Hardware.

See Algorithmic Pattern Generator (APG), Memory Test Patterns.

Except as noted, the functions documented in this section are used to configure the Algorithmic Pattern Generator (APG) in preparation for executing Memory Test Patterns.

- APG Counter Functions
- APG Reload Register Functions
- `dmain(), dbase()`
- APG Data Register Functions
- APG Jam Register Functions
- APG XMAIN & YMAIN Register Functions
- APG XBASE & YBASE Register Functions
- APG XFIELD & YFIELD Register Functions
- APG AMAIN, ABASE, AFIELD Set/Get Functions
- Address Cross-over Bit Functions
- APG Timer Interrupt Address Functions
- `find_label()`
- APG Y-Index Register Functions

- set_chip_select(), get_chip_select()
- set_adhiz(), get_adhiz()
- set_invsns(), get_invsns()
- get_jca(), set_jca()
- set_ps(), get_ps()
- set_tset(), get_tset()
- set_udata(), get_udata()
- Get APG Fail Information
  actualdata()
  expectdata()
  lvm_error_mode()
  errmar()
  Note that other functions are listed here but not yet documented.
- find_mar()
- find_by_mar(), find_by_var()
- addrs()
- label_offset()
- Clearing APG Pipelines
- Single-stepping APG Patterns

3.23.7.1 APG Counter Functions

See Memory-pattern Related Functions, Algorithmic Pattern Generator (APG).

Description

The count() function is used to set or get a value to/from an Algorithmic Pattern Generator (APG) counter.

Using count() to set a given counter also causes its corresponding reload register to be loaded with the same value. To load only the reload register use the reload() function.

Usage

```
void count( int Loop, UINT Value );
```

```
UINT count( int Loop );
```

where:

Loop identifies the target APG counter. When setting a counter value, legal values are 1 through 60 (counters 61, 62, 63, and 64, are reserved for system use).

Value specifies the desired count value. Legal values are 1 to 4,294,967,295 (2^{32}-1).

The getter version of `count()` returns the current value read from the specified counter.

Example

The following example loads the value 100 into APG counter 5:

```
count( 5, 100 );
```

The following example reads the current value from APG counter 10:

```
UINT val = count( 10 );
```

3.23.7.2 APG Reload Register Functions

See Memory-pattern Related Functions, Algorithmic Pattern Generator (APG).

Description

The `reload()` function is used to set or get a value to/from an Algorithmic Pattern Generator (APG) reload register. Each APG counter has a corresponding reload register.

When the `count()` function is used to initialize an APG counter, that counter's corresponding reload register is automatically loaded to the same value. The `reload()` function can be used to set or get a value to/from only the reload register, without modifying the corresponding APG counter.

Usage

```
void reload( int Loop, UINT Value );
UINT reload( int Loop );
```

where:

Loop identifies the target APG reload register. Legal values are 1 through 60 (counters 61, 62, 63, and 64, are reserved for system use).

`Value` specifies the desired count value. Legal values are 1 to 4,294,967,295 (2^{32}-1).

The getter version of `reload()` returns the current value read from the specified reload register.

Examples

The following example loads the value 100 into reload register 5:

```
reload( 5, 100 );
```

The following example gets the current value from reload register 10:

```
UNIT val = reload( 10 );
```

3.23.7.3 APG Reload Register Mode Functions

See Memory-pattern Related Functions.

Note: this feature is only available using Maverick-II and Magnum 1.

Description

The `apg_reload_register_mode_set()` function is used to modify the behavior of the Memory Test Pattern `RELOAD#` operand (see COUNT Counter Operands). Pattern operation details are documented in COUNT Counter Operands.

The `apg_reload_register_mode_get()` function is used to get the currently set mode.

Note the following:

- By default, the original `RELOAD` operation is enabled. The system does not otherwise modify the mode. Default operation is equivalent to executing `apg_reload_register_mode_set(t_reload_mode1)`.
- The mode is global; i.e. not per reload/counter register number.
- Executing `apg_reload_register_mode_set(t_reload_mode2)` changes the mode as described in COUNT Counter Operands. Executing `apg_reload_register_mode_set(t_reload_mode1)` restores operation to the default mode.

Usage

```
void apg_reload_register_mode_set( ApgReloadRegMode mode );
ApgReloadRegMode apg_reload_register_mode_get();
```

where:

mode specifies the desired mode of operation. Default = `t_reload_mode1`.

`apg_reload_register_mode_get()` returns the currently set mode.

Example

```
apg_reload_register_mode_set( t_reload_mode2 );
ApgReloadRegMode mode = apg_reload_register_mode_set();
```

3.23.7.4 dmain(), dbase()

See Memory-pattern Related Functions, Algorithmic Pattern Generator (APG).

Description

The `dmain()` function is used to set or get a value to/from the APG Data Generator's DMAIN register and Data Register Fill-bit for that register.

The `dbase()` function is used to set or get a value to/from the APG Data Generator's DBASE register and Data Register Fill-bit for that register.

Both functions set or get the value to/from the APG on the site which executes the function. When Sites-per-Controller > 1 all APGs under the control of a given master-site are executing the same pattern in sync, thus the value returned from the master-site represents all sites.

Usage

```
void dmain( __int64 value );
__int64 dmain();
void dbase( __int64 value );
__int64 dbase();
```

where:

value specifies the value to be written to the DMAIN or DBASE register. Only the low 37 bits are used, as follows:

- The low 36 bits load the data register bits D0 through D35.
- Bit 37 loads the Data Register Fill-bit.

The getter versions of `dmain()` and `dbase()` returns the current value read from the register. Only the low 37 bits are valid, as noted for **value** above (bits above bit 37 are set to 0).

Examples

The following example loads the hex value 0x20 into the APG Data Generator's DBASE register and the Data Register Fill-bit is set = 0:

```
dbase( 0x20 );
```

The following example gets the current value from the APG Data Generator's DMAIN register:

```
__int64 val = dmain();
```

3.23.7.5 APG Data Strobe Control

Note:

Description

The `data_strobe()` function is used to statically enable or disable the strobe signals which originate from the APG Data Generator. This is a somewhat complex topic, keep reading.

Overview

The Magnum 1 test system contains two sources of test pattern data:

- Algorithmic Pattern Generator (APG) when executing Memory Test Patterns.
- Combined Logic Vector Memory (LVM) / Scan Vector Memory (SVM) for stored Logic Test Patterns and Scan Test Patterns.

The diagram below shows key Magnum 1 architecture features:

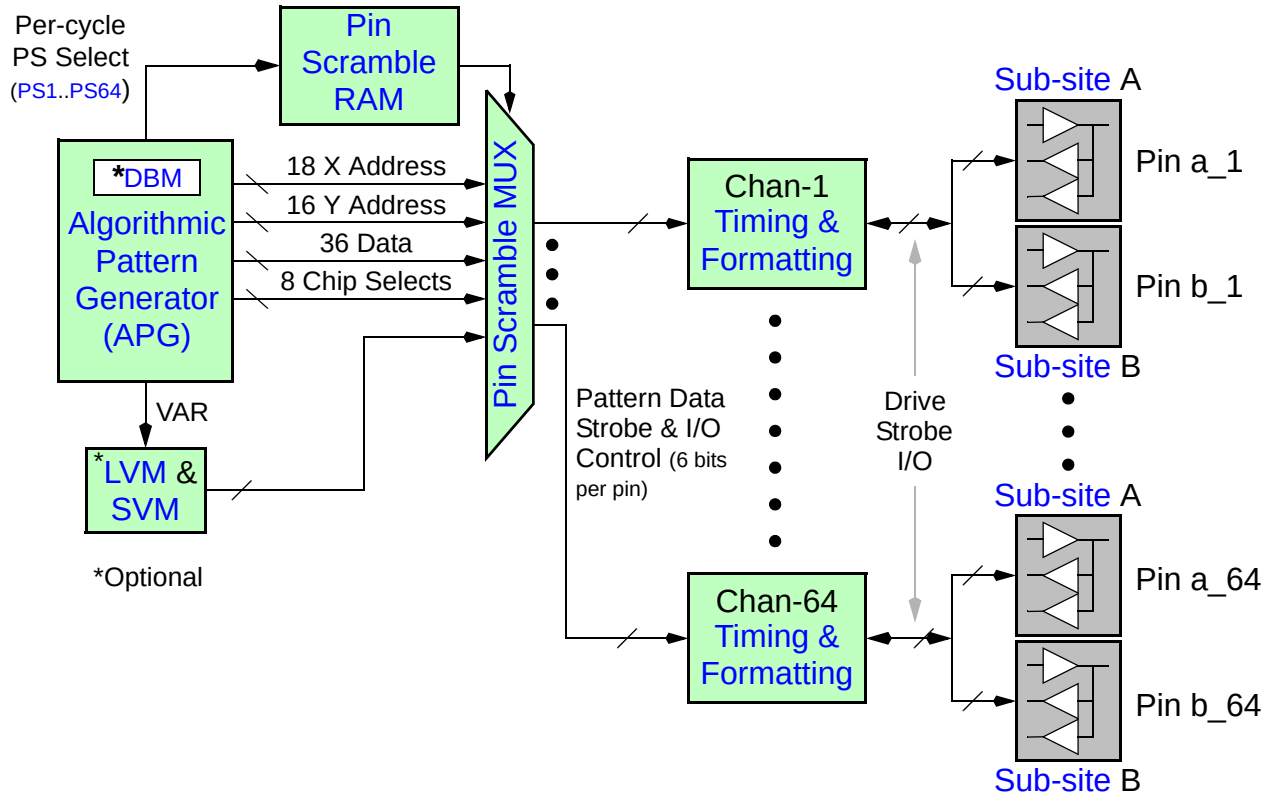


Figure-52: Test Pattern Data Source Hardware Architecture

Using the Pin Scramble MUX, the source of pattern data for each timing channel can be selected on a per-channel/per-cycle basis, at full tester speed. In other words, for any given pin-pair, the source of pattern data can be selected on a per-cycle basis from any APG address, data, or clock data bit, or any LVM data bit, or any Scan Vector Memory data bits. Using Magnum 1/2/2x (but not Maverick-I/-II) any LVM data bit to be mapped to any pin.

During pattern execution, in each tester cycle each data source is actually supplying three bits: pattern data, I/O control, and strobe control. When using LVM or SVM three distinct bits are stored in memory at each address. When using the APG, the I/O control bits and Strobe control bits are managed using special mnemonics in the test pattern. For example, `ADHIZ` controls I/O state for the APG Data Generator outputs, Address outputs are always driving and never strobe, etc.

This section documents how APG Data Generator strobes are controlled, and the related `data_strobe()` function.

Note: the `data_strobe()` function only affects APG Data Generator strobe control; i.e. it has no effect on strobes controlled from Logic Test Patterns or Scan Test Patterns.

The APG Data Generator supplies 36 data outputs which can be used for drive data and/or expect (strobe) data. Each of these 36 outputs (`t_d0..t_d35`) consists of 3 bits: one for pattern data, one for I/O Control, one for strobe enable.

APG Data Generator strobes have several *enables*:

- During initial program load, strobes are disabled for all APG Data Generator. The system software does not otherwise change these strobe enables.
- The `data_strobe()` function, called from user C-code prior to executing the test pattern. This sets up static strobe enables before a pattern is executed.
- The `MAR READ` mnemonic, provides dynamic (per pattern instruction) strobe control. Strobes will be generated on APG Data Generator outputs in any pattern instructions containing the `READ` mnemonic, but only on those outputs previously enabled by `data_strobe()`.
- The `MAR READUDATA` mnemonic, provides dynamic (per pattern instruction) control. Strobes will be generated on APG Data Generator outputs in any pattern instructions containing the `READUDATA` mnemonic, but only on those outputs enabled by `data_strobe()`, and only those outputs which have a logic-1 in the corresponding bit position of the `UDATA` value in the current pattern instruction. Using `MAR READUDATA` mnemonic the `UDATA` value is used as a bit-wise strobe mask.

It is important to *emphasize* that the `data_strobe()` function programs APG Data Generator hardware, *not* the pin electronics or timing system. Which PE channels actually use Data Generator outputs, and thus *might* be strobed, is determined by the Pin Scramble Map selected in each pattern instruction.

data_strobe(): Mask Method

Note: read the Note: and Overview in APG Data Strobe Control.

The `data_strobe()` function is used to statically enable APG Data Generator strobes.

As compared to the data_strobe(): Pin Scramble Method, the *mask* method is much simpler to understand and set up (no Pin Lists or Pin Scramble arguments are used with the mask method, which is consistent with how the related APG hardware works).

Using the mask method a simple bit-wise mask argument is specified, with each bit position corresponding to an APG Data Generator output (`t_d35..t_d0`). A given data strobe is enabled by a logic-1 mask bit value, and disabled by a logic-0 mask bit value, with the LSB controlling `t_d0`, etc. For example, a mask value of 0x3FFFF would enable data strobes for `t_d17..t_d0`.

At any given time, to determine which APG Data Generator outputs have strobes statically enabled it is only necessary to identify the most recently executed `data_strobe()` function. Then, if using the mask method documented in this section, strobes will be statically enabled on all APG Data Generator outputs which have a logic-1 in the corresponding mask bit position.

Note: the `data_strobe()` function must be called from a test block. It ***must not*** be called in the Site Begin Block, because its effect will be overwritten by the system software.

During initial program load, strobes are disabled for all APG Data Generator. The system software does not otherwise change these strobe enables.

Usage

```
void data_strobe( __int64 Mask );
void data_strobe();
```

where:

`Mask` is a 64-bit `_int64` value. Only the low 36-bits are used, to control strobes on `t_d35..t_d0`. See Description.

Calling `data_strobe()` with no argument disables strobes on all APG Data Generator outputs.

Example

The following example enables data strobes on all 36 APG Data Generator outputs:

```
data_strobe( 0xFFFFFFFFF );
```

data_strobe(): Pin Scramble Method

Note: read the Note: and Overview in APG Data Strobe Control

The `data_strobe()` function is used to statically enable APG Data Generator strobes.

The Pin Scramble method is the original (early) methodology used by a large number of test programs. However, as compared to the data_strobe(): Mask Method, this original methodology is more complex to set up, and somewhat confusing due to the arguments passed to `data_strobe()`.

Using the Pin Scramble method, two versions of `data_strobe()` are available:

```
data_strobe( pin_list );
data_strobe( pin_list, pin_scramble );
```

The first version defaults to using Pin Scramble PS1, so the rest of the description below assumes a Pin Scramble argument is used.

Remember that the purpose of `data_strobe()` is to set up static APG Data Generator strobe enables from C-code, prior to pattern execution. The arguments to the `data_strobe()` function were established based on *Megatest* Q2 compatibility, and are a source of confusion. The key to this method, is to view the combined Pin Lists and Pin Scramble Map arguments as a *map* used to identify which specific APG Data Generator outputs will have strobes statically enabled (`t_d0` through `t_d35`).

- The specific pins in the pin list used by these versions of `data_strobe()` *are* ***not*** *important* (!), they are only used as a programming tool. However, the pin list must only contain signal pins, and duplicate pins cannot be used. It is sometimes necessary for pin lists used by `data_strobe()` to contain a mix of DUT pins which might otherwise seem very strange. See Example 2:.
- The PPin Scramble Map used by these versions of `data_strobe()`, including the default `PS1` when used, identifies which APG Data Generator outputs have strobes enabled: `t_d0` through `t_d35`. It is ***not*** *necessary* that this Pin Scramble Map actually be used in a test pattern. In fact, as shown in Example 2:, it may be necessary to create a special Pin Scramble Map solely for use by `data_strobe()`. It is also legal (and common) for Pin Scramble Maps used only by `data_strobe()` to be incomplete; i.e. only contain references to APG Data Generator outputs, with other DUT pins not defined.

At any given time, to determine which APG Data Generator outputs have strobes statically enabled it is only necessary to identify the most recently executed `data_strobe()` function. Then, if using the Pin Scramble method documented in this section, strobes will be statically enabled on all APG Data Generator outputs listed in the Pin Scramble Map argument to `data_strobe()`. The default Pin Scramble Map (`PS1`) is used when the `data_strobe()` function call doesn't contain an explicit Pin Scramble argument.

Note: the `data_strobe()` function must be called from a test block. It ***must not*** be called in the Site Begin Block, because its effect will be overwritten by the system software.

During initial program load, strobes are disabled for all APG Data Generator. The system software does not otherwise change these strobe enables.

Usage

```
void data_strobe( PinList* pPinList );
void data_strobe( PinList* pPinList, PSNumber PSnum );
void data_strobe( DWORD Mask );
void data_strobe( __int64 Mask );
void data_strobe();
```

where:

`pPinList` is a previously defined pin list. See Description above for details.

`PSnum` is an optional argument explicitly specifying which Pin Scramble Map to use. If this argument is omitted, then `PS1` is assumed. See Description above for details.

Examples

Example 1:

The following example enables strobes on those APG Data Generator outputs which are mapped to the pins in `data_bus` using Pin Scramble Map `PS1`:

```
data_strobe( data_bus );
```

Example 2:

The following example is typical of a serial device, where one DUT pin is being strobed, in sequential pattern cycles, for `D7..D0`.

Strobes must be enabled for eight APG Data Generator outputs: `D7..D0`. However, since the DUT only has one serial “data pin” some creativiity is needed when defining both the Pin Lists and Pin Scramble Map to be used by `data_strobe()`.

The basic requirement is, using a single `data_strobe()` function call, to enable strobes on `t_d0.. t_d7`. To do this requires a pin list containing eight unique DUT signal pins (any 8

signal pins will do), and a Pin Scramble Map which maps those eight pins to t_d0..t_d7. Note that the example code below will typically be found in different source files.

```
PIN_SCRAMBLE( some_name ) {
   // ... other SCRAMBLE_MAP definitions ...
   // PS37 is used only for data_strobe() use.
   // Using PS37 (vs. PS2, or PS19) is arbitrary.
   SCRAMBLE_MAP( PS37 ) {
      SCRAMBLE( a_Dinout, t_d0 )  // Actual signal pins used don't
      SCRAMBLE( a_Reset, t_d1 )   // matter !!!
      SCRAMBLE( a_Clk, t_d2 )
      SCRAMBLE( a_Ale, t_d3 )
      SCRAMBLE( a_Red, t_d4 )
      SCRAMBLE( a_Blu, t_d5 )
      SCRAMBLE( a_Grn, t_d6 )
      SCRAMBLE( a_Tom, t_d7 )
   }
   // ... other SCRAMBLE_MAP definitions ...
}
// This PL used only with data_strobe()
PINLIST( special_data_strobe_pinlist) {
   PINS4( a_Dinout, a_Reset, a_Clk, a_Ale )
   PINS4( a_Red, a_Blu, a_Grn, a_Tom )
}
TEST_BLOCK( some_tb_name ) {
   // ... other C code
   data_strobe ( special_data_strobe_pinlist, PS37 );
   // ... other C code
}
```

This example statically enables strobes on APG Data Generator outputs t_d0..t_d7. Then, during pattern execution, any DUT pin(s) which are scrambled to any of these data sources via the per-instruction Pin Scramble Map selection will be strobed if/when the READ or READUDATA is used. Note the following:

- The pins in the pin list don't matter, any 8 unique signal pins will do.
- The pin list is very unlikely to be used for any other purpose. A comment to this effect is appropriate.

- In applications where there are not enough signal pins to complete the needed Pin Scramble Map it may be necessary to add pins to the Pin Assignment Table purely for this purpose. However, in applications where multiple parallel DUTs are being tested use any signal pins available - which pins are used doesn't matter!
- The order the APG Data Generator outputs are listed in the Pin Scramble Map is not important.
- It is very unlikely that this Pin Scramble Map will be used for any other purpose other than `data_strobe()` use. A comment to this effect is appropriate.

3.23.7.6 APG Data Register Functions

See Memory-pattern Related Functions, APG Data Generator.

Description

The `datreg()` function is used to set or get a value to/from the APG Data Generator, including the Data Register Fill-bit. Also see `dmain(), dbase()`.

Usage

```
void datreg( __int64 Value );
__int64 datreg();
```

where:

`Value` specifies the value to be written to the data register. Only the low 37 bits are used, as follows:

- The low 36 bits load the data register bits D0 through D35
- Bit 37 loads the Data Register Fill-bit.

The getter version of `datreg()` returns the current value from the data register. Only the low 37 bits are valid, as noted for **`Value`** above (bits above bit 37 are set to 0).

Examples

The following example loads the hex value 0x20 into the APG Data Generator and the Data Register Fill-bit is set = 0:

```
datreg( 0x20 );
```

The following example gets the current value from the data register:

```
__int64 val = datreg();
```

3.23.7.7 APG Jam Register Functions

See Memory-pattern Related Functions, APG Data Generator, JAM Logic.

Description

The `jamreg()` function is used to set or get a value in the APG Data Generator JAM Register.

Also see JAM Logic and `apg_jam_ram_set(), apg_jam_ram_get()`.

Usage

```
void jamreg( __int64 Value );
__int64 jamreg();
```

where:

`Value` specifies the value to be written to the JAM Register. Only the low 37 bits are used. The low 36 bits load the JAM register bits D0 through D35. The bit 37 loads the JAM register shift/fill bit.

The getter version of `jamreg()` returns the current value from the JAM register. Only the low 36 bits are valid; bits above bit 36 are set to 0.

Examples

The following example loads the value 0x20 into the JAM register:

```
jamreg( 0x20 );
```

The following example gets the current value from the JAM register:

```
__int64 val = jamreg();
```

3.23.7.8 APG XMAIN & YMAIN Register Functions

See Memory-pattern Related Functions, APG Address Generator.

Description

The `xmain()` function is used to set or get a value to/from the APG Address Generator's XMAIN register.

The `ymain()` function is used to set or get a value to/from the APG Address Generator's YMAIN register.

Usage

```
void xmain( int Value );
void ymain( int Value );
int xmain();
int ymain();
```

where:

`Value` specifies the value to be written to the XMAIN or YMAIN register. Only the low 16 bits are used.

The getter version of `xmain()` and `ymain()` return the current value from the XMAIN or YMAIN register. Only the low 16 bits are valid.

Examples

The following example loads the value -1 into the APG Address Generator's XMAIN register:

```
xmain( -1 );
```

The following example gets the current value from the YMAIN register:

```
int val = ymain();
```

3.23.7.9 APG XBASE & YBASE Register Functions

See Memory-pattern Related Functions, APG Address Generator.

Description

The `xbase()` function can be used to set or get a value to/from the APG Address Generator's XBASE register.

The `ybase()` function can be used to set or get a value in the APG Address Generator's YBASE register.

Usage

```
void xbase( int Value );
void ybase( int Value );
int xbase();
int ybase();
```

where:

`Value` specifies the value to be written to the XBASE or YBASE register. Only the low 16 bits are used.

The getter versions of `xbase()`and `ybase()` return the current value from the XBASE or YBASE register. Only the low 16 bits are valid.

Examples

The following example loads the value -1 into the APG Address Generator's XBASE register:

```
xbase( -1 );
```

The following example gets the current value from the YBASE register:

```
int val = ybase();
```

3.23.7.10 APG XFIELD & YFIELD Register Functions

See Memory-pattern Related Functions, APG Address Generator.

Description

The `xfield()` function can be used to set or get a value to/from the APG Address Generator's XFIELD register.

The `yfield()` function can be used to set or get a value in the APG Address Generator's YFIELD register.

Usage

```
void xfield( int Value );
void yfield( int Value );
int xfield();
int yfield();
```

where:

`Value` specifies the value to be written to the XFIELD or YFIELD register. Only the low 16 bits are used.

The getter versions of `xfield()` and `yfield()` return the current value in the XFIELD or YFIELD register. Only the low 16 bits are valid.

Examples

The following example loads the value -1 into the APG Address Generator's XFIELD register:

```
xfield( -1 );
```

The following example gets the current value from the YFIELD register:

```
int val = yfield();
```

3.23.7.11 APG AMAIN, ABASE, AFIELD Set/Get Functions

See Memory-pattern Related Functions, APG Address Generator.

Description

The `amain()` function is used to set or get a value to/from the APG Address Generator's combined XMAIN and YMAIN registers.

The `abase()` function can be used to set or get a value to/from the APG Address Generator's combined XBASE and YBASE registers.

The `afield()` function can be used to set or get a value to/from the APG Address Generator's combined XFIELD and YFIELD registers:

Note the following:

Note: amain(), abase() and afield() should NOT be used on Magnum 2 or Magnum 2x . These systems have 24 X and 24 Y address bits (40 bits total can be used) which means that a UINT value cannot be used to set/get an appropriate value. Instead, use xmain()/ymain(), xbase()/ybase() and xfield()/yfield() to set/get X vs. Y address values individually.

- When using amain(), abase() or afield() the high order address bits, i.e. whether the X address or Y address is most significant, is determined by the x_fast_axis() function.
- The values set for numx() and numy() affect how many bits of the value passed to amain(), abase() and afield() setter functions are written to XMAIN vs. YMAIN (or XBASE vs. YBASE, or XFIELD vs. YFIELD). See Example.
- Similarly, the values set for numx() and numy() affect how many bits of the value returned by the amain(), abase() and afield() getter functions are X address bits vs. Y address bits.

Usage

```
void amain( UINT Value );
void abase( UINT Value );
void afield( UINT Value );
UINT amain();
UINT abase();
UINT afield();
```

where:

Value specifies the value to be written to the combined XMAIN/YMAIN, XBASE/YBASE or XFIELD/YFIELD registers. See Description.

amain(), abase() and afield() getter functions returns the current value of the combined XMAIN/YMAIN, XBASE/YBASE or XFIELD/YFIELD registers. See Description.

Examples

Both examples below show the effect of numx() and numy() on the use of amain():

```
numx( 3 );             // See numx()
numy( 5 );             // See numy()
x_fast_axis( TRUE );   // See x_fast_axis()
```

```
xmain( 0x0 );          // See xmain()
ymain( 0x1F );         // See ymain()
output(" Amain => 0x%x", amain() );
```

This results in the following output:

```
Amain => 0xf8
//   i.e. binary 1111 1000    where:
//   xmain = 1F  ---- -
//            ymain = 0 ---
```

The following example uses `amain()` to set both XMAIN and YMAIN as follows:

```
amain( 0xAD );
// ymain    xmain
// 0x15      0x3  Hex
// 10101     101  Binary
// 1010 1101 = 0xAD Value passed to amain()
output(" Xmain => 0x%x : Ymain => 0x%x", xmain(), ymain() );
```

This results in the following output:

```
Xmain => 0x15 : Ymain => 0x3
```

3.23.7.12 Address Cross-over Bit Functions

See Memory-pattern Related Functions, APG Address Generator.

Note: first available in software release h2.2.7/h1.2.7.

Description

The `atopo_xvr()` function is used to select the Address Cross-over Bits used to replace the upper 3 X address and/or upper 3 Y bits at the input to the Address TOPO RAM.

Note the following:

- `atopo_xvr()` only affects the upper 3 address bits at the input to the Address TOPO RAMs.

- Executing `atopo_xvr()` identifies 3 X and 3 Y address bits used as Address Cross-over Bits. The most recently executed function completely defines all 6 Address Cross-over Bits.
- The argument names identify which bit is being specified. All 6 arguments are required. Legal values for each arguments are slightly different:

Argument Name	Default Value	Cross-over Values
X15	t_x15	t_y0 through t_y15
X16	t_x16	
X17	t_x17	
Y13	t_y13	t_x0 through t_x17
Y14	t_y14	
Y15	t_y15	
Note: specifying an invalid value causes the default value to be selected in hardware. To intentionally select the default value 0 may be specified for a given argument value.		

Usage

```
void atopo_xvr( TesterFunc X15, TesterFunc X16, TesterFunc X17,
                TesterFunc Y13, TesterFunc Y14, TesterFunc Y15 );
```

where:

X15, **X16** and **X17** identify the desired cross-over address bit used to replace these bits of the X APG Address Generator.

Y13, **Y14** and **Y15** identify the desired cross-over address bit used to replace these bits of the Y APG Address Generator.

Example

The following example uses `atopo_xvr()` to map address Y0 to the X15 address bit. The other high-order X and Y address bits use their default mapping:

```
atopo_xvr( t_y0, 0, 0, 0, 0, 0, 0 );
```

3.23.7.13 APG Timer Interrupt Address Functions

See Memory-pattern Related Functions, Algorithmic Pattern Generator (APG).

Description

The `intadr()` function can be used to set or get a value to/from the APG Interrupt Timer's interrupt address register.

Usage

```
void intadr( int Value );
int intadr();
```

where:

`Value` specifies the value to be written to the interrupt address register. Only the low 15 bits are used.

The getter version of `intadr()` returns the current value from the interrupt address register. Only the low 15 bits are valid; the system software sets the bits above bit 15 to 0.

Example

The following example loads the value 20 into the APG Interrupt Timer's interrupt address register.

```
intadr( 20 );
```

The following example gets the current value from the interrupt address register:

```
int val = intadr();
```

3.23.7.14 find_label()

See Memory-pattern Related Functions, Pattern Labels.

Description

The `find_label()` function can be used to return the Pattern Label, if any, of a specified pattern instruction in a specified Memory Test Pattern.

This function does report valid information in simulation mode.

Usage

```
CString find_label( Pattern *obj, DWORD mar );
```

where:

`obj` identifies the target pattern.

`mar` identifies the target pattern instruction. This is a pattern-relative value; i.e. the first **`mar`** of the specified pattern is 0.

`find_label()` returns the label of the specified instruction. If the instruction has no label the label of the closest prior instruction which does have a label is returned. If the pattern containing the specified **`mar`** has no appropriate label to return an empty string is returned ("").

Example

The example consists of three parts:

- Test Block Code
- Example Test Pattern
- Example Output

Test Block Code

```
CString lbl = find_label( myPat, 1 );
output(" lbl => %s", lbl );
```

Example Test Pattern

```
PATTERN( memPat )
%  lab1:
   mar  inc
%  lab2:
   mar  inc
%  lab3:
   mar  done
```

Example Output

```
lbl => lab2
```

3.23.7.15 APG Y-Index Register Functions

See Memory-pattern Related Functions, Algorithmic Pattern Generator (APG).

Description

The `yindex()` function can be used to set and get a value from the APG Data Generator's Y-index register (see `Yindex`).

The Y-index register is used in Memory Test Patterns to generate one or more diagonal data patterns, as a function of X/Y address.

The `yindexmask()` function can be used to set a value in the APG Y-index mask register. Only locations in the Y-index mask register which contain a logic-1 are used to satisfy the diagonal generator equation X = Y + Index. If the low 16 bits of the register are all set to logic-1 (0xFFFF), then all 16 Y-address bits are used in the comparison to generate a diagonal. Shrinking the mask below the maximum number of addresses used to test the DUT will generate multiple diagonals in the memory array.

Before `yindexmask()` is first executed, the Y-index mask is recalculated each time `xmax()` or `ymax()` is executed. In this scenario, the Y-index mask register is set to the smaller of `xmax()` or `ymax()`, which are the maximum X and Y addresses used to test the DUT.

Once `yindexmask()` is executed, the Y-index mask will only be modified by executing `yindexmask()`; i.e. it will no longer change when `xmax()` or `ymax()` are executed.

Usage

```
void yindex( int Value );
int yindex();
void yindexmask( int Value );
```

where:

`Value` specifies the value to be written to the Y-index register or Y-index mask register. Only the low 16 bits are used.

The `yindex()` getter function returns the current value in the Y-index register. All bits above bit-15 set = 0 by the system software.

Examples

The following example loads the Y-index register with value 20:

```
yindex( 20 );
```

The following example gets the current value in the Y-index register:

```
int val = yindex();
```

The following example loads the Y-index mask register with current value of `ymax()`. This enables all Y address bits for comparison in the diagonal data generator.

```
yindexmask( ymax() );
```

The following example causes diagonal comparisons to consider only Y0, Y1, and Y2. Using the diagonal generator equation X = Y + Index, the comparison (equation) will be considered TRUE when this equation is satisfied on the three low order address bits only. Higher order address bits are ignored in the comparison. In this example, a DUT with more than 3 Y address inputs (`ymax()` > 3) will have multiple diagonal data patterns automatically generated (if the function is enabled in the pattern) with each diagonal being offset by eight rows or columns from the previous diagonal:

```
yindexmask( 7 );
```

3.23.7.16 set_chip_select(), get_chip_select()

See Memory-pattern Related Functions, Algorithmic Pattern Generator (APG).

Description

The `set_chip_select` function can be used to modify the `CHIPS` instruction of a specified pattern instruction of a specified Memory Test Pattern.

The `get_chip_select()` function allows user-written C-code to read information from the `CHIPS` instruction of a specified pattern instruction of a specified Memory Test Pattern.

Usage

```
BOOL set_chip_select(  Pattern *obj,
                       LPCTSTR label,
                       TesterFunc cs,
                       ChipSelectMode mode );
```

```
BOOL set_chip_select(  Pattern *obj,
                       LPCTSTR label,
                       int delta,
                       TesterFunc cs,
                       ChipSelectMode mode );
ChipSelectMode get_chip_select(  Pattern *obj,
                                 LPCTSTR label,
                                 TesterFunc cs );
ChipSelectMode get_chip_select(  Pattern *obj,
                                 LPCTSTR label,
                                 int delta,
                                 TesterFunc cs );
```

where:

obj is the pattern of interest.

label is the label of the pattern instruction to be read or modified. See Pattern Labels. If the target instruction does not have a label, the label of a prior instruction can be used and an offset value (instruction count) from that instruction specified using **delta**.

cs is one of t_cs1 through t_cs8 and used to specify which chip select is to be read or modified.

mode is the desired modification, and must be one of the ChipSelectMode enumerated type values:

delta specifies an offset from **label**, for use when the target pattern instruction does not have a **label**.

set_chip_select() returns FALSE if the specified pattern or label is invalid, otherwise TRUE is returned.

get_chip_select() returns -1 if the specified pattern or label is invalid. Otherwise it returns the current state of the specified chip select, which will be one of the ChipSelectMode enumerated type values noted above.

Example

```
BOOL ok = set_chip_select( myPat, "L1", t_cs1, t_cs_pulse_true );
ok = set_chip_select( myPat, "L1", 1, t_cs1, t_cs_false );
ChipSelectMode m = get_chip_select( myPat, "L1", t_cs1 );
ChipSelectMode m = get_chip_select( myPat, "L1", 1, t_cs1 );
```

3.23.7.17 set_adhiz(), get_adhiz()

See Memory-pattern Related Functions, PINFUNC Instruction, Pattern Labels.

Description

The `set_adhiz()` function may be used to change the value of the `PINFUNC ADHIZ` state associated with one specified memory pattern instruction. This is the `ADHIZ` state value specified using the test pattern `PINFUNC` instruction.

The `get_adhiz()` function may be used to get the current `ADHIZ` state of one specified memory pattern instruction.

Note: using Magnum 1/2/2x, these instructions do not apply to Logic Test Patterns; i.e. patterns with `logic` Pattern System Attributes.

Usage

The following 2 functions are used to change the `ADHIZ` state of one pattern instruction:

```
BOOL set_adhiz( Pattern *obj, LPCTSTR label, BOOL value );
BOOL set_adhiz(Pattern *obj,
               LPCTSTR label,
               int delta,
               BOOL value );
```

The following 2 functions are used to *get* the `ADHIZ` state of one pattern instruction:

```
int get_adhiz( Pattern *obj, LPCTSTR label );
int get_adhiz( Pattern *obj, LPCTSTR label, int delta );
```

where:

`obj` specifies the pattern of interest.

`label` specifies the Pattern Label of the instruction of interest. See Pattern Labels. If the target instruction does not have a label, the **`label`** for an earlier instruction must be specified and **`delta`** used to specify an offset from that earlier instruction. See Example.

`value` specifies the desired `ADHIZ` state to be set in the specified pattern instruction, replacing the previous value. Legal values are `TRUE` to set the `ADHIZ` state and `FALSE` to clear the `ADHIZ` state:

`set_adhiz()` returns TRUE if the operation succeeded, otherwise FALSE is returned.

The `get_adhiz()` get functions return:

- 1 = ADHIZ TRUE
- 0 = ADHIZ FALSE
- -1 = ERROR

Example

The following example has two parts:

- Example Test Pattern
- Program Code

Example Test Pattern

In this simple memory pattern, the second instruction does not have a Pattern Label. This was done intentionally to show how the `delta` argument is used.

```
PATTERN ( myPat )
% Label_1:
   MAR       INC     // No ADHIZ this instruction
%  PINFUNC   ADHIZ   // This is the target instruction = Label_1 + 1
   MAR       INC
% Label_2:
   MAR       INC
%  MAR DONE
```

Program Code

See comments in code below.

```
// Get current ADHIZ state for instruction at Label_1 + 1
int az = get_adhiz( myPat, "Label_1", 1 );
// Check it
if ( az == -1 )
   output(" ERROR: specified pattern or label is invalid");
// Output info
output(" ADHIZ state in 2nd pattern instruction =>\\");
output(" %s", az ? "TRUE" : "FALSE");
```

```
// Clear ADHIZ state in same instruction
BOOL ok = set_adhiz( myPat, "Label_1", 1, FALSE );
// Check it
if (! ok ) output(" ERROR: specified pattern or label is invalid");
// Get it again and output info.
az = get_adhiz( myPat, "Label_1", 1 );
if ( az == FALSE )
   output(" ADHIZ state change OK");
else
   output(" ERROR: ADHIZ state change Failed");
```

3.23.7.18 set_invsns(), get_invsns()

See Memory-pattern Related Functions, DATGEN Invert Sense Operand, Pattern Labels.

Description

The `set_invsns()` function may be used to change the value of the APG Data Generator's inversion state associated with one specified pattern instruction. This is the value set using the DATGEN Invert Sense Operand of the test pattern `DATGEN` instruction.

The `get_invsns()` function may be used to get the current DATGEN Invert Sense Operand value of one specified pattern instruction.

Note: using Magnum 1/2/2x, these instructions do not apply to Logic Test Patterns; i.e. patterns with `logic` Pattern System Attributes.

Usage

The following 2 functions are used to set the `invsns` value of one pattern instruction:

```
BOOL set_invsns( Pattern *obj, LPCTSTR label, int value );
BOOL set_invsns(  Pattern *obj,
                  LPCTSTR label,
                  int delta,
                  int value );
```

The following 2 functions are used to *get* the `invsns` value of one pattern instruction:

```
int get_invsns( Pattern *obj, LPCTSTR label );
int get_invsns( Pattern *obj, LPCTSTR label, int delta );
```

where:

obj specifies the pattern of interest.

label specifies the Pattern Label of the instruction of interest. See Pattern Labels. If the target instruction does not have a label, the **label** for an earlier instruction must be specified and **delta** used to specify an offset from that earlier instruction. See Example.

value specifies the desired `invsns` value to be set in the specified pattern instruction, replacing the previous value. Legal values are:

- 0 to set the invsns state to `NOTINV`
- 1 to set the invsns state to `INVSNS`
- 2 to set the invsns state to `XORINV`

`set_invsns()` returns TRUE if the operation succeeded, otherwise FALSE is returned.

The `get_invsns()` get functions return:

- 2 = `XORINV`
- 1 = `INVSNS`
- 0 = `NOTINV`
- -1 = ERROR (bad Pattern*, bad label, etc.)

Example

The following example has two parts:

- Example Test Pattern
- Program Code

Example Test Pattern

In this simple Memory Test Pattern, the second instruction does not have a Pattern Label. This was done intentionally to show how the `delta` argument is used.

```
PATTERN ( myPat )
% Label_1:
   DATGEN    DATDAT          // No inversion @ this instruction
%  DATGEN    DATDAT, XORINV // Target instruction = Label_1 + 1
```

```
% Label_2:
   DATGEN    DATDAT  // No inversion @ this instruction
%  MAR       DONE
```

Program Code

See comments in code below.

```
// Get invert sense value for instruction at Label_1 + 1
int iv = get_invsns( myPat, "Label_1", 1 );

// Check it
if ( iv == -1 )
   output(" ERROR: specified pattern or label is invalid");

// Output info
output(" Invert value for 2nd pattern instruction => \\");
switch (iv) {
   case 2 : output("XORINV"); break;
   case 1 : output("INVSNS"); break;
   case 0 : output("NOTINV");
}

// Change value in same instruction
BOOL ok = set_invsns( myPat, "Label_1", 1, 0 );

// Check it
if (! ok ) output(" ERROR: specified pattern or label is invalid");

// Get it again and output info.
iv = get_invsns( myPat, "Label_1", 1 );

if ( iv == 0 )
   output(" Inversion value change OK");
else
   output(" ERROR: Inversion value change Failed");
```

3.23.7.19 get_jca(), set_jca()

See Memory-pattern Related Functions.

Description

The get_jca() function may be used to retrieve the jump-call address (JCA) from a specified pattern instruction.

The set_jca() function may be used to modify the JCA in a specified pattern instruction.

JCA stands for jump/call address. This is the absolute (not pattern relative) address of the target subroutine (instruction) referenced by one of the jump/call pattern instructions (JUMP, GOSUB, CJMPZ, CJMPNZ, etc.).

In the test pattern source file, a jump/call address is identified using a Pattern Label. Then, as the test program loads, the system software resolves the actual APG MAR address of each label to determine each JCA value. In the following example, *jump_label_1* sets the JCA for the pattern instruction below to the address of the instruction containing the label jump_label_1:

```
% COUNT   COUNT1, DECR, AON
  MAR     CJMPNZ, jump_label_1
```

The set_jca() function can be used, from user C code, to modify the *JCA* of a specified pattern instruction in a specified pattern. The target subroutine address must be specified as a DWORD value; i.e. not as a label. The *JCA* of a target pattern instruction can be determined using addrs() and label_offset(). The addrs() function is used to look-up the absolute address of the first instruction of the pattern containing the target subroutine (instruction). The label_offset() function is used to locate a specific label in the pattern containing the target subroutine, as an offset from the first instruction of the pattern. Using label_offset() is only required when the target subroutine (instruction) is not the first instruction in the subroutine pattern. See test block TB3 in the example below.

The get_jca() function can be used to read the *JCA* of a pattern instruction which uses one of the jump/call instructions. This can be useful to save an original (compiled) pattern subroutine address in anticipation of restoring it later.

Usage

```
BOOL set_jca( Pattern *obj, LPCTSTR label, WORD value );
BOOL set_jca( Pattern *obj, LPCTSTR label, int delta, WORD value );
WORD get_jca( Pattern *obj, LPCTSTR label );
WORD get_jca( Pattern *obj, LPCTSTR label, int delta );
set_mar() // Deprecated, identical to set_jca()
get_mar() // Deprecated, identical to get_jca()
```

where:

obj is the pattern to be accessed.

label is the Pattern Label of the instruction to be accessed. When the target instruction does not have a label the **delta** argument can be used.

delta is an offset from the pattern instruction containing the specified **label**. **delta** = 0 for the instruction containing the label.

value is the absolute pattern address (MAR) of the target subroutine instruction.

`set_jca()` returns FALSE if the specified label is not found in the specified pattern, otherwise TRUE is returned.

`get_jca()` returns -1 if the specified label is not found in the specified pattern, otherwise the returned value is the absolute address (MAR) of the instruction in the specified pattern at the specified **label** + **delta**.

Example

The following example is rather large and includes the following parts. Each part has separate comments:

- Test Pattern Code
- Test Block Code
- Sequence/Binning Table Code
- Runtime Output Messages

The Test Block Code uses `label_offset()`, `addrs()`, and `set_jca()` to modify the *JCA* in a simple test pattern to call a different pattern subroutine each time.

Test Pattern Code

The three test blocks in Test Block Code each execute `my_pattern`. As compiled, `my_pattern` calls the pattern subroutine `my_subr1`.

The pattern compiler uses `my_subr1` to determine the jump/call address (*JCA*) of the target subroutine. Test Block Code modifies this jump/call address two times, to call different pattern subroutines in different test blocks.

```
PATTERN( my_pattern )
% label_1:
   COUNT   COUNT1, INCR
   MAR     GOSUB, my_subr1 // Original subroutine target
%  MAR     DONE

PATTERN ( my_subr1 )     // TB1 calls here
% subr1_label_1:
   COUNT   COUNT2, INCR
   MAR     RETURN

// Other potential subroutines
PATTERN ( my_subr2 )     // Test block TB2 calls here

% subr2_label_1:
   COUNT   COUNT3, INCR

% subr2_label_2:         // Test block TB3 calls here
   COUNT   COUNT4, INCR

% subr2_label_3:
   COUNT   COUNT5, INCR
   MAR     RETURN
```

Test Block Code

Each test block sets 5 APG counters = 0. Then, depending on which pattern instructions actually execute, some APG counters are incremented, while others are not. In each test block, after pattern execution completes, the counter values are printed, and can be reviewed to see which pattern instructions actually executed.

The first test block, `TB1`, executes `my_pattern` as compiled. This calls one pattern subroutine, `my_subr1`. Only `COUNT1` and `COUNT2` will be incremented when `TB1` executes. This can be seen in the Runtime Output Messages.

Test block `TB2` modifies the pattern subroutine address, the *JCA* (jump/call address) in `my_pattern`. Instead of calling `my_subr1` it is modified to call `my_subr2`. When this occurs, `COUNT1`, `COUNT3`, `COUNT4`, and `COUNT5` are incremented, which can be seen in the Runtime Output Messages.

Test block `TB3` also modifies the *JCA* in `my_pattern`. In this case, the first instruction executed in the subroutine is in `my_subr2` at label `subr2_label_1`. When this occurs, `COUNT1`, `COUNT4`, and `COUNT5` are incremented, which can be seen in the Runtime Output Messages.

```
int test_result;
```

```
TEST_BLOCK( TB1 ) {
   output("\n=====================================");
   output(" Executing %s", current_test_block());
   pipe_clear();
   count( 1, 0 ); count( 2, 0 ); count( 3, 0 );
   count( 4, 0 ); count( 5, 0 );
   test_result = funtest (my_pattern, error );
   output("  count 1 => %d (SB=1)", count(1) ); // Should be = 1
   output("  count 2 => %d (SB=1)", count(2) ); // Should be = 1
   output("  count 3 => %d (SB=0)", count(3) ); // Should be = 0
   output("  count 4 => %d (SB=0)", count(4) ); // Should be = 0
   output("  count 5 => %d (SB=0)", count(5) ); // Should be = 0
   return TRUE;
}

TEST_BLOCK( TB2 ) {
   output("\n=====================================");
   output(" Executing %s", current_test_block());
   pipe_clear();
   count( 1, 0 ); count( 2, 0 ); count( 3, 0 );
   count( 4, 0 ); count( 5, 0 );
   // Get MAR of first instruction in target pattern
   DWORD mar, var;
   addrs( my_subr2, &mar, &var ); // addrs(), var not used
   output("  Target pattern MAR => %d", mar );
   // Modify the calling pattern to change the original target
   // subroutine address from "my_subr1" to the MAR of
   // my_subr2: subr2_label_2.
    if ( set_jca ( my_pattern, "label_1", mar ) == FALSE ) {
      output(" ERROR: bad label passed to set_jca()");
      return FALSE;
   }
   test_result = funtest (my_pattern, error );
   output("  count 1 => %d (SB=1)", count(1) ); // Should be = 1
   output("  count 2 => %d (SB=0)", count(2) ); // Should be = 0
   output("  count 3 => %d (SB=1)", count(3) ); // Should be = 1
```

```
   output("  count 4 => %d (SB=1)", count(4) ); // Should be = 1
   output("  count 5 => %d (SB=1)", count(5) ); // Should be = 1
   return TRUE;
}


TEST_BLOCK( TB3 ) {
   output("\n====================================");
   output(" Executing %s", current_test_block());
   pipe_clear();
   count( 1, 0 ); count( 2, 0 ); count( 3, 0 );
   count( 4, 0 ); count( 5, 0 );
   // Get MAR of first instruction in target pattern
   DWORD mar, var;
   addrs( my_subr2, &mar, &var ); // addrs(), var not used
   output("  Target pattern MAR => %d", mar );
   // If target instruction is not the first instruction of
   // the target pattern, locate the label instruction relative
   // to the start of the target pattern, as an offset
   int offset = label_offset ( my_subr2,  "subr2_label_2" );
   output("  Target label offset => %d", offset );
   // Add label offset to pattern start MAR
   mar += offset;
   output("  Target instructon MAR => %d", mar);
   // Modify the calling pattern to change the original target
   // subroutine address from "my_subr1" to the MAR of
   // my_subr2: subr2_label_2.
    if ( set_jca ( my_pattern, "label_1", mar ) == FALSE ) {
      output(" ERROR: bad label passed to set_jca()");
      return FALSE;
   }
   test_result = funtest (my_pattern, error );
   output("  count 1 => %d (SB=1)", count(1) ); // Should be = 1
   output("  count 2 => %d (SB=0)", count(2) ); // Should be = 0
   output("  count 3 => %d (SB=0)", count(3) ); // Should be = 0
   output("  count 4 => %d (SB=1)", count(4) ); // Should be = 1
```

```
   output("  count 5 => %d (SB=1)", count(5) ); // Should be = 1
   return TRUE;
}
```

Sequence/Binning Table Code

This is included to help comprehend the order of Runtime Output Messages.

```
SEQUENCE_TABLE( SBB ) {
   SEQUENCE_TABLE_INIT
   TEST( TB1, NEXT, STOP )
   TEST( TB2, NEXT, STOP )
   TEST( TB3, STOP, STOP )
}
```

Runtime Output Messages

```
TestStarted(1)...
=====================================
Executing TB1
count 1 => 1 (SB=1)
count 2 => 1 (SB=1)
count 3 => 0 (SB=0)
count 4 => 0 (SB=0)
count 5 => 0 (SB=0)
=====================================
Executing TB2
Target pattern MAR => 77
count 1 => 1 (SB=1)
count 2 => 0 (SB=0)
count 3 => 1 (SB=1)
count 4 => 1 (SB=1)
count 5 => 1 (SB=1)
=====================================
Executing TB3
Target pattern MAR => 77
Target label offset => 1
Target instructon MAR => 78
count 1 => 1 (SB=1)
count 2 => 0 (SB=0)
count 3 => 0 (SB=0)
```

```
count 4 => 1 (SB=1)
count 5 => 1 (SB=1)
TestDone...bin = builtin_Pass
```

3.23.7.20 set_ps(), get_ps()

See Memory-pattern Related Functions.

Description

The `set_ps()` function may be used to change the Pin Scramble Map selection in one specified pattern instruction. This is the pin scramble value specified using the test pattern `PINFUNC` instruction.

The `get_ps()` function may be used to get the Pin Scramble Map selection from one specified pattern instruction.

Note: using Magnum 1/2/2x, these instructions do not apply to Logic Test Patterns; i.e. patterns with `logic` Pattern System Attributes.

Usage

The following 2 functions are used to change the Pin Scramble Map selection in one pattern instruction:

```
BOOL set_ps( Pattern *obj, LPCTSTR label, PSNumber ps );
BOOL set_ps( Pattern *obj, LPCTSTR label, int delta, PSNumber ps );
```

The following 2 functions are used to *get* the Pin Scramble Map selection from one pattern instruction:

```
PSNumber get_ps( Pattern *obj, LPCTSTR label );
PSNumber get_ps( Pattern *obj, LPCTSTR label, int delta );
```

where:

`obj` specifies the pattern of interest.

`label` specifies the Pattern Label of the instruction of interest. If the target instruction does not have a label, the **`label`** for an earlier instruction must be specified and **`delta`** used to specify an offset from that earlier instruction. See Example.

ps specifies the desired pin scramble to be inserted into the specified pattern instruction, replacing the previous value. Legal values are from the PSNumber enumerated type (PS_na is not legal in this context):

set_ps() returns TRUE if the operation succeeds, otherwise FALSE is returned.

The get_ps() get functions return the pin scramble currently associated with the specified pattern instruction. If the specified pattern or label is invalid get_ps() returns PS_na.

Example

The following example has two parts:

- Example Test Pattern
- Program Code

Example Test Pattern

In this simple Memory Test Pattern, the second instruction does not have a Pattern Label. This was done intentionally to show how the delta argument is used.

```
PATTERN( myPat )
% Label_1:
   PINFUNC   PS1
%  PINFUNC   PS3     // This is the target instruction = Label_1 + 1
% Label_2:
   PINFUNC   PS2
%  MAR       DONE
```

Program Code

See comments in code below.

```
// Get current pin scramble for instruction at Label_1 + 1
PSNumber ps = get_ps( myPat, "Label_1", 1 );
// Check it
if ( ps == PS_na )
   output(" ERROR: specified pattern or label is invalid");
// Output info. Note that by default enumerated types begin at 0,
// thus it is necessary to add +1 to the value
output(" PS for 2nd pattern instruction => PS%d", ps + 1);
```

```
// Change pin scramble from PS3 to PS4
BOOL ok = set_ps( myPat, "Label_1", 1, PS4);
// Check it
if (! ok ) output(" ERROR: specified pattern or label is invalid");
// Get it again and output info.
ps = get_ps( myPat, "Label_1", 1 );
if ( ps == PS4 )
   output(" PS change OK, now => PS%d", ps + 1);
else
   output(" ERROR: PS change Failed");
```

3.23.7.21 set_tset(), get_tset()

See Memory-pattern Related Functions, Time-sets (TSET).

Description

The `set_tset()` function may be used to change the Time-set(TSET) selection of one specified pattern instruction.

The `get_tset()` function may be used to get the current TSET from one specified pattern instruction.

Note: using Magnum 1/2/2x, these instructions do not apply to Logic Test Patterns; i.e. patterns with `logic` Pattern System Attributes.

Usage

The following 2 functions are used to change the TSET of one pattern instruction:

```
BOOL set_tset( Pattern *obj, LPCTSTR label, TSETNumber tset );
BOOL set_tset( Pattern *obj,
               LPCTSTR label,
               int delta,
               TSETNumber tset );
```

The following 2 functions are used to *get* the TSET of one pattern instruction:

```
TSETNumber get_tset( Pattern *obj, LPCTSTR label );
TSETNumber get_tset( Pattern *obj, LPCTSTR label, int delta );
```

where:

obj specifies the pattern of interest.

label specifies the Pattern Label of the instruction of interest. If the target instruction does not have a label, the **label** for an earlier instruction must be specified and **delta** is used to specify an offset from that earlier instruction. See Example.

tset specifies the desired Time-set(TSET) to be inserted into the specified pattern instruction, replacing the previous value. Legal values are from the `TSETNumber` enumerated type (`TSET_na` is not legal in this context).

`set_tset()` returns FALSE if an error occurs (invalid Pattern Label, etc.).

The `get_tset()` get functions return the current `TSETNumber` from the specified pattern instruction. If the specified pattern or label is invalid `get_tset()` returns `TSET_na`.

Example

The following example has two parts:

- Example Test Pattern
- Program Code

Example Test Pattern

In this simple Memory Test Pattern, the second instruction does not have a Pattern Label. This was done intentionally to show how the `delta` argument is used.

```
PATTERN( myPat )
% Label_1:
   PINFUNC   TSET1
%  PINFUNC   TSET3   // This is the target instruction = Label_1 + 1
% Label_2:
   PINFUNC   TSET1
%  MAR DONE
```

Program Code

See comments in code below.

```
// Get current TSET for instruction at Label_1 + 1
TSETNumber ts = get_tset( myPat, "Label_1", 1 );

// Check for error
if ( ts == TSET_na )
   output(" ERROR: specified pattern or label is invalid");

// Output info. Note that by default enumerated types begin at 0,
// thus it is necessary to add +1 to the value
output(" TSET => TSET_%d", ts + 1);

// Change TSET from TSET3 to TSET4
BOOL ok = set_tset( myPat, "Label_1", 1, TSET4);

// Check it
if (! ok ) output(" ERROR: specified pattern or label is invalid");

// Get it again and output info.
ts = get_tset( myPat, "Label_1", 1 );

if ( ts == TSET4 )
   output(" TSET change OK, now => TSET_%d", ts + 1);
else
   output(" ERROR: TSET change Failed, TSET = TSET_%d", ts + 1);
```

3.23.7.22 set_udata(), get_udata()

See Memory-pattern Related Functions.

Description

The `set_udata()` function is used to modify the UDATA value of one specified pattern instruction. This is the value set in a given pattern instruction using the UDATA Instruction.

The `get_udata()` function may be used to retrieve the UDATA value from a specified pattern instruction.

Note: caution must be exercised when using `set_udata()`. In some pattern instructions the UDATA value is used implicitly; i.e. without an explicit user-specified UDATA pattern instruction. Thus, carelessly modifying a UDATA value may cause incorrect, and difficult to diagnose, pattern operation. ***Do*** read the introduction to the UDATA Instruction section which discusses these issues.

Note: the set_udata() and get_udata() functions must ***NOT*** be used to access UDATA values in pattern instruction(s) which are Controlling PE Levels from the Test Pattern.

Usage

```
BOOL set_udata( Pattern *obj, LPCTSTR label, __int64 value );
BOOL set_udata(Pattern *obj,
               LPCTSTR label,
               int delta,
               __int64 value );
__int64 get_udata( Pattern *obj, LPCTSTR label );
__int64 get_udata( Pattern *obj, LPCTSTR label, int delta );
```

where:

obj is a pointer to a test pattern. This equates to the pattern name specified in the PATTERN statement. See example.

label specifies the Pattern Label of the instruction of interest. If the target instruction does not have a label, the **label** for an earlier instruction must be specified and **delta** is used to specify an offset from that earlier instruction.

value specifies the value to be written to the UDATA field.

The get_udata() functions return the current UDATA value from the specified pattern instruction.

The set_udata() functions return FALSE if the specified pattern or label cannot be resolved, otherwise TRUE is returned.

Example

The example below assumes the following target pattern. In this pattern, using UDATA to load COUNT1 is arbitrary.

```
PATTERN( myPat )
%  label_1:
   COUNT   COUNT1, COUNTUDATA  // Load COUNT1 from UDATA
   UDATA   0xFF                // UDATA value
   MAR     DONE
```

The code below reads and prints the value of UDATA from the pattern above, then modifies it:

```
__int64 val = get_udata( myPat, "label_1" );
output(" Initial UDATA value => 0x%I64x", val ); // Outputs "0XFF"
if ( ! set_udata( myPat, "label_1", 0xA5 ) ) // Set UDATA to 0xA5
   output (" ERROR: set_udata() reported an error");
```

3.23.7.23 set_vihh(), get_vihh()

See Memory-pattern Related Functions.

Description

The `set_vihh()` function may be used to change the VIHH Map selection in one specified pattern instruction. This is the VIHH Map value specified using the test pattern `PINFUNC` instruction (or the default value).

The `get_vihh()` function may be used to get the current VIHH Map of one specified pattern instruction.

Note: using Magnum 1/2/2x, these instructions do not apply to Logic Test Patterns; i.e. patterns with `logic` Pattern System Attributes.

Usage

The following 2 functions are used to change the VIHH Map of one pattern instruction:

```
BOOL set_vihh( Pattern *obj, LPCTSTR label, VihhNumbervihh );
BOOL set_vihh( Pattern *obj,
               LPCTSTR label,
               int delta,
               VihhNumber vihh );
```

The following 2 functions are used to *get* the VIHH Map of one pattern instruction:

```
VihhNumber get_vihh( Pattern *obj, LPCTSTR label );
VihhNumber get_vihh( Pattern *obj, LPCTSTR label, int delta );
```

where:

obj specifies the pattern of interest.

label specifies the Pattern Label of the instruction of interest. If the target instruction does not have a label, the **label** for an earlier instruction must be specified and **delta** used to specify an offset from that earlier instruction. See Example.

vihh specifies the desired VIHH map to be inserted into the specified pattern instruction, replacing the previous value. Legal values are from the `VihhNumber` enumerated type (`VIHH_na` is not legal in this context).

`set_vihh()` returns `TRUE` if the operation succeeded, otherwise `FALSE` is returned.

The `get_vihh()` get functions return the VIHH map currently associated with the specified pattern instruction. If the specified pattern or label is invalid `get_vihh()` returns `VIHH_na`.

Example

The following example has two parts:

- Example Test Pattern
- Program Code

Example Test Pattern

In this simple memory pattern, the second instruction does not have a Pattern Label. This was done intentionally to show how the `delta` argument is used.

```
PATTERN( myPat )
% Label_1:
   PINFUNC   VIHH1
%  PINFUNC   VIHH3   // This is the target instruction = Label_1 + 1
% Label_2:
   PINFUNC   VIHH2
%  MAR DONE
```

Program Code

See comments in code below.

```
// Get current VIHH map for instruction at Label_1 + 1
VihhNumber vm = get_vihh( myPat, "Label_1", 1 );
```

```
// Check it
if ( vm == VIHH_na )
   output(" ERROR: specified pattern or label is invalid");

// Output info. Note that by default enumerated types begin at 0,
// thus it is necessary to add +1 to the value
output(" VIHH for 2nd pattern instruction => VIHH%d", vm + 1);

// Change VIHH map from VIHH3 to VIHH4
BOOL ok = set_vihh( myPat, "Label_1", 1, VIHH4);

// Check it
if (! ok ) output(" ERROR: specified pattern or label is invalid");

// Get it again and output info.
vm = get_vihh( myPat, "Label_1", 1 );

if ( vm == VIHH4 )
   output(" VIHH change OK, now => VIHH%d", vm + 1);
else
   output(" ERROR: VIHH map change Failed");
```

3.23.7.24 Get APG Fail Information

See Memory-pattern Related Functions.

Description

After Memory Test Pattern execution ends the Algorithmic Pattern Generator (APG) retains a short pipeline history of generated X/Y address, data, and MAR values. When pattern execution stops using stop-on-error (more below), various values from this pipeline can be read to determine where pattern execution actually stopped and what data bits failed. Proper operation requires that the test pattern be executed using the stop-on-error mode; i.e.

```
funtest( patname, error );
```

See Executing Functional Tests, and Pattern Execution Stop Condition Options.

These registers are read-only and can not be modified from user software.

Usage

These functions get information from the cycle prior to the first failing cycle:

```
prevmar();   // Read mar previous to error cycle
prevadr();   // Read unscrambled address previous to error cycle
sprevadr();  // Read scrambled address previous to error cycle
prevxadr();  // Read unscrambled X address previous to error cycle
prevyadr();  // Read unscrambled Y address previous to error cycle
sprevxadr(); // Read scrambled X address previous to error cycle
sprevyadr(); // Read scrambled Y address previous to error cycle
prevdata();  // Read drive/expected data previous to error cycle
```

These functions get information from the failing cycle:

```
actualdata();// Read actual data from DUT at error cycle.
expectdata();// Read APG expected data at error cycle.
errmar();    // Read MAR at error cycle
erradr();    // Read unscrambled address at error cycle
serradr();   // Read scrambled address at error cycle
errxadr();   // Read unscrambled X address at error cycle
erryadr();   // Read unscrambled Y address at error cycle
serrxadr();  // Read scrambled X address at error cycle
serryadr();  // Read scrambled Y address at error cycle
```

These functions get information from the cycle at the DUT when pattern execution actually stops. Several cycles occur after the failing cycle before the pattern generator actually stops:

```
dutmar();    // Read mar currently at the DUT
dutadr();    // Read unscrambled address currently at the DUT
sdutadr();   // Read scrambled address currently at the DUT
dutxadr();   // Read unscrambled X address currently at the DUT
dutyadr();   // Read unscrambled Y address currently at the DUT
sdutxadr();  // Read scrambled X address currently at the DUT
sdutyadr();  // Read scrambled Y address currently at the DUT
dutdata();   // Read data currently driven to the DUT
```

The values retrieved are of the following types:

field	integer size in bits	bits used	type
MAR	32	15	int
full address	32	32	DWORD
X or Y address	16	16	int
data	64	36	__int64

The system software sets the high order bits above those specified in the bits-used column to zero.

Example

In this example, the APG pipeline is interrogated to find the actual data from the DUT in the first failing cycle and outputs that value.

```
int result = funtest( mypat, error );
if( result == FAIL) {
   _int64 val = actualdata();
   output("Actual data => %I64d", val );
}
```

3.23.7.25 actualdata()

See Memory-pattern Related Functions.

Description

After Memory Test Pattern execution ends the Algorithmic Pattern Generator (APG) retains a short pipeline history of generated X/Y addresses, data, and MAR values. When pattern execution stops using stop-on-error (more below), key values can be read from this pipeline to determine where the pattern halted and what data bits failed. Proper operation requires that the test pattern be executed using the stop-on-fail mode; i.e.

```
funtest( patname, error );
```

See Executing Functional Tests, and Pattern Execution Stop Condition Options.

The `actualdata()` function returns a value derived from expect data (see `expectdata()`). In simple terms, failing data bits returned by `actualdata()` will be the inverse of the corresponding bits in expect data. Detailed operation is more complex, as noted below.

- If the pattern execution passes, the value returned by `actualdata()` is invalid.
- When pattern execution stops (on error) the expect data value from the failing pattern instruction is copied into the actual data value, and modified as noted below.
- If the failing pattern instruction did not contain a MAR READ, READV or READZ, or READUDATA instruction, the value returned by `actualdata()` matches the original expect data value.
- Assuming the failing pattern instruction did contain a MAR READ, READV or READZ, or READUDATA, the system software references the Pin Scramble Map of the failing instruction, to identify which APG Data Generator outputs (`t_d0..t_d35`) were mapped to DUT pin(s). Then, for each of these APG data outputs, if the first pin mapped to that output failed the corresponding bit in actual data is inverted.
- When complete, the (modified) actual data value is returned by the `actualdata()` function.

Note the following:

- The actual data bit for APG Data Generator output(s) which are not pin scrambled to a (failing) pin are not changed; i.e. they remain the same as expect data.
- `actualdata()` does not consider whether the APG is configured as 18-wide data or 36-wide data. The user must know how the 36 APG Data Generator outputs are actually used in their test program.
- `actualdata()` does not know the reason a given pin fails. It is up to the user to understand that the value reported by `actualdata()` may not actually reflect a failure caused by a defective DUT.

Usage

```
_int64 actualdata();
```

where:

`actualdata()` returns the information noted in Description above.

Example

```
int result = funtest( mypat, error );
if( result == FAIL) {
   _int64 val = actualdata();
   output("Actual data => %I64d", val );
}
```

3.23.7.26 expectdata()

See Memory-pattern Related Functions.

Description

After Memory Test Pattern execution ends the Algorithmic Pattern Generator (APG) retains a short pipeline history of generated X/Y addresses, data, and MAR values. When pattern execution stops using stop-on-error (more below), key values can be read from this pipeline to determine where the pattern halted and what data bits failed. Proper operation requires that the test pattern be executed using the stop-on-fail mode; i.e.

```
funtest( patname, error );
```

See Executing Functional Tests, and Pattern Execution Stop Condition Options.

The `expectdata()` function returns the output of the APG Data Generator from the first failing pattern cycle. Note the following:

- If the pattern execution passes, the value returned by `expectdata()` is invalid.
- The value returned by `expectdata()` does not depend on whether any of the APG Data Generator outputs were actually used (i.e. pin scrambled to pin(s)) in the failing cycle. The return value is also not affected by whether the failing cycle contained a MAR READ, READV or READZ, or READUDATA instruction. For example, if the first failure is controlled by expect data from a logic pattern instruction (see Mixed Memory/Logic Patterns) the value returned by `expectdata()` will not be affected. Similarly, the value returned is not affected by how many pins were scrambled to a given data generator output.

Usage

```
_int64 expectdata();
```

where:

`expectdata()` returns the information noted in Description above.

Example

```
int result = funtest( mypat, error );
```

```
if( result == FAIL) {
   _int64 val = expectdata();
   output("Expect data => %I64d", val );
}
```

3.23.7.27 lvm_error_mode()

See Memory-pattern Related Functions, Logic Test Patterns.

Description

The `lvm_error_mode()` function is not required nor supported on Magnum 1/2/2x.

3.23.7.28 errmar()

See Memory-pattern Related Functions, Memory Test Patterns.

Description

The `errmar()` function returns an integer value representing the microRAM address (MAR) of the first failing Algorithmic Pattern Generator (APG) pattern instruction.

After Memory Test Pattern execution ends the Algorithmic Pattern Generator (APG) retains a short pipeline history of generated X/Y addresses, data, and MAR values. When pattern execution stops using stop-on-error (more below), key values can be read from this pipeline to determine where the pattern halted and what data bits failed. Proper operation requires that the test pattern be executed using the stop-on-fail mode; i.e.

```
funtest( patname, error );
```

See Executing Functional Tests, and Pattern Execution Stop Condition Options.

The returned MAR value is an *absolute* address that can be converted to a pattern relative address using `find_mar()` as described below.

Usage

```
int errmar();
```

where `errmar()` returns the absolute MAR of the first failing memory pattern instruction.

Example

```
int result = funtest( myPat, error );
if( result == FAIL ) {
   int first_fail = errmar();
   output (" First Fail MAR (absolute) => %n", first_fail );
}
```

3.23.7.29 find_mar()

See Memory-pattern Related Functions, Memory Test Patterns.

Description

Given an absolute memory pattern instruction address (absolute MAR) the `find_mar()` function can be used to return the following:

- A pointer to the memory pattern containing the specified MAR
- The Pattern Label at or prior to the specified instruction. See Usage.

The term absolute means relative to the start of Algorithmic Pattern Generator (APG) uRAM.

Usage

```
BOOL find_mar( DWORD mar,
               Pattern **obj,
               CString *label,
               DWORD *offset );
BOOL find_mar( DWORD mar, Pattern **obj, DWORD *offset );
```

where:

`mar` specifies the absolute APG uRAM address (MAR) of the target memory pattern instruction.

`obj` is a pointer to an existing `Pattern*` variable used to return a pointer to the pattern containing the specified **`mar`** value. `find_mar()` will return FALSE if the specified **`mar`** does not correspond to a valid memory pattern instruction.

`label` is a pointer to an existing `CString` variable used to return the Pattern Label associated with the specified **`mar`**. The following rules apply:

- If `find_mar()` returns FALSE the value in **`label`** is invalid.
- If the specified **`mar`** doesn't have a label, the label of the closest prior instruction which does have a label is returned. When this occurs, **`offset`** indicates how many instructions earlier than **`mar`** the label was found.
- If the memory pattern containing the specified **`mar`** has no appropriate label to return an empty string is returned (`""`).

`offset` is a pointer to an existing `DWORD` variable used to return a value. **`offset`** is used in two contexts:

- When the **`label`** parameter is used, **`offset`** returns information about **`label`** when the target **`mar`** has no label. See **`label`**.
- When label is not used, **`offset`** returns the offset of the specified **`mar`** from the start of the test pattern.

In either application, if `find_mar()` returns FALSE the value in **`offset`** is invalid.

`find_mar()` returns TRUE if the specified **`mar`** is within a user-defined Memory Test Pattern, otherwise FALSE is returned.

Example

The example consists of three parts:

- Test Block Code
- Example Test Pattern
- Example Output

Test Block Code

The following example uses the `addrs()` function to obtain the first MAR of the `myMemPat` test pattern. That value is subsequently used by `find_mar()` to get information for the 3rd instruction in the pattern (+2). Note that this instruction has no label, but the prior instruction does; thus the first offset value returned = 1. See Example Output. The second offset value reflects the MAR offset from the start of the pattern; i.e. 2.

```
DWORD mar;
BOOL ok = addrs( myMemPat, &mar );
if( ! ok ) output("  ERROR: addrs() returned an error");
else output(" First MAR for myMemPat => %d", mar );

Pattern* pat;
CString label;
DWORD offset;
```

```
ok = find_mar( (mar +2), &pat, &label, &offset );
if( ! ok ) output("  ERROR: find_mar() returned an error");
else
   output(" pat => %s, label => %s, offset => %d",
        resource_name(pat),
        label,
        offset );
ok = find_mar( (mar +2), &pat, &offset );
if( ! ok ) output("  ERROR: find_mar() returned an error");
else output(" pat => %s, offset => %d",
     resource_name(pat),
     offset );
```

Example Test Pattern

```
PATTERN( myMemPat )
%  lab1:
   MAR  INC
%  lab2:
   MAR  INC
%  MAR  INC // Target instruction
%  MAR  DONE
```

Example Output

```
pat => myMemPat, label => lab2, offset => 1
pat => M1memPat, offset => 2
```

3.23.7.30 find_by_mar(), find_by_var()

See Memory-pattern Related Functions, Memory Test Patterns.

Description

The `find_by_mar()` function can be used to identify the Memory Test Pattern which contains a specified uRAM instruction address (MAR).

The `find_by_var()` function can be used to identify the Logic Test Pattern which contains a specified vector address (VAR).

These functions do report useful information in simulation mode.

Usage

```
Pattern *find_by_mar( DWORD mar );
Pattern *find_by_var( DWORD var );
```

where:

`mar` specifies the target Memory Test Pattern instruction absolute address (MAR). Absolute means relative to the start of APG uRAM memory.

`var` specifies the target Logic Test Pattern instruction absolute address (VAR). Absolute means relative to the start of logic vector memory (LVM).

Both `find_by_mar()` and `find_by_var()` return a pointer to the pattern which contains the specified **`mar`** or **`var`** value.

Example

The following examples use the `addrs()` function to obtain the first MAR or VAR of a test pattern. That value is subsequently used by `find_by_mar()` and `find_by_var()`.

```
DWORD mar;
BOOL ok = addrs( myMemoryPat, &mar ); // addrs()
if( ! ok ) output("  ERROR: addrs() returned an error");
else {
   output(" First MAR for myMemoryPat => %d", mar );
   Pattern* mpat = find_by_mar( mar );
   if( mpat )
      output("  mpat name => %s", resource_name( mpat ) );
   else
      output("  WARNING: no memory pattern at MAR(%d)", mar);
}
DWORD var;
ok = addrs( myLogicPat, &mar, &var ); // addrs()
if( ! ok ) output("  ERROR: addrs() returned an error");
else {
   output(" First VAR for myLogicPat => %d", var );
   Pattern* vpat = find_by_var( var );
```

```
    if( vpat )
       output("  vpat name => %s", resource_name( vpat ) );
    else
       output("  WARNING: no logic pattern at VAR(%d)", var);
}
```

3.23.7.31 addrs()

See Memory-pattern Related Functions.

Description

The `addrs()` function is used to determine the following information about individual Memory Test Patterns, individual Logic Test Patterns, or individual Scan Test Patterns:

- The first Algorithmic Pattern Generator (APG) microRAM address (MAR) used.
- The first Logic Vector Memory (LVM) address (VAR) used.
- The first Scan Vector Memory (SVM) address (SAR) used
- The number of APG microRAM addresses used
- The number of LVM addresses used
- The number of SVM addresses used

See `addrs()` for details.

3.23.7.32 label_offset()

See Memory-pattern Related Functions, Memory Test Patterns, Logic Test Patterns.

Description

The `label_offset()` function is used to locate the pattern instruction containing a specified Pattern Label, as an offset relative to the first instruction of a specified pattern. The offset of the first instruction in a pattern is 0. See Pattern Labels.

The second version of `label_offset()` was added in software release h1.1.23, to support Logic Test Patterns. By default, it operates the same as the original (first) version, returning the offset for the specified label from a pattern stored in the APG's uRAM (MAR engine). Optionally, if the `uram` argument is set = FALSE, it returns the offset for the

specified label from a specified Logic Test Pattern stored in the APG's vRAM (VAR engine). This version is not usable on Maverick-I.

Usage

The following function returns the offset for the specified label from a pattern stored in the APG's uRAM (MAR engine):

```
int label_offset( Pattern *obj, LPCTSTR label );
```

The following function is first available in software release h1.1.23 (see above):

```
int label_offset( Pattern *obj,
                  LPCTSTR label,
                  BOOL uram DEFAULT_VALUE( TRUE ) );
```

where:

`obj` is the pattern of interest.

`label` is the Pattern Label of the pattern instruction of interest.

`uram` is optional and, if specified, selects whether the target **`label`** is to be located in the APG's uRAM (MAR engine) or vRAM (VAR engine). Default = TRUE = MAR engine (uRAM).

`label_offset()` returns -1 if the specified **`label`** is not found in the specified pattern, otherwise it returns the offset from the first instruction in the pattern.

Example

Given the following pattern:

```
PATTERN ( my_pat )
% label_1:      // Offset = 0
   MAR INC
% label_2:      // Offset = 1
   MAR INC
% label_3:      // Offset = 2
   MAR DONE
```

The value of `offset` = 1 below.

```
int offset = label_offset ( my_pat, "label_2" );
```

Given the following pattern:

```
PATTERN ( my_pat, mav2, logic )
% label_1:      // Offset = 0
   VEC     HL10XVZ
% label_2:      // Offset = 1
   VEC     HL10XVZ
% label_3:      // Offset = 2
   VEC     HL10XVZ
```

The label offset value retrieved from the VAR engine (vRAM) = 1 below.

```
int offset = label_offset ( my_pat, "label_2", FALSE );
```

3.23.7.33 Clearing APG Pipelines

See Memory-pattern Related Functions.

Description

The `pipe_clear()` function clears the APG pipelines of prior pattern instructions by filling them with the Default Memory Pattern Instruction.

> Note: the `pipe_clear()` function is **`rarely`** needed because the system software clears the pipelines before a pattern is executed.

This function may be useful if the `step()` function is being used to single-step the pattern generator or when user-code is initializing selected APG registers (see `lbdata()`).

Usage

```
void pipe_clear();
```

Example

```
pipe_clear();
```

3.23.7.34 Single-stepping APG Patterns

See Memory-pattern Related Functions.

Description

The `step()` function is used to single-step the Algorithmic Pattern Generator (APG) a specified number of cycles.

This feature is typically used during pattern debug, but it can also be used in a test program for special test applications. When the pattern generator is single stepped, a single pattern instruction is executed with all timing generators firing at their programmed times for that cycle.

Usage

```
void step( int Value );
```

where:

`value` is the number of pattern instructions to execute. Legal values are 1 to 0xFFFFFFFF (2^{32}).

Note: there is computer overhead associated with *every* tester cycle generated by the `step()` function, thus entering large numbers for `value` is not practical because of the processing time required.

Example

```
step( 5 ); // Step the pattern generator 5 cycles
```

3.23.8 Logic Pattern Related Functions

See Logic Test Patterns.

Except as noted, the functions documented in this section apply to logic pattern applications only.

- VAR Counter Functions
- `errvar()`
- `find_var()`
- `vecdata()`
- `addrs()`

- var_pinfunc()

3.23.8.1 VAR Counter Functions

See Logic Pattern Related Functions, Logic Test Patterns.

Description

The vcount() function is used to set or get a value to/from a Magnum 1/2/2x VAR engine counter. These are the counters controlled in Logic Test Patterns using the VCOUNT Instruction.

Note: when VAR engine counters are explicitly used to control pattern loops the count value assigned is the number of desired loop iterations (n), which is different than when using MAR engine counters, which use n-1.

Note: on Maverick-II and Magnum 1 the 4 VAR engine counters are implicitly used for STARTLOOP/ENDLOOP (counter 1 and 2/3 when nesting STARTLOOP/ENDLOOPs) and RPT (counter 4) operations. In these applications the counter used is loaded with value(s) specified in the test pattern source file. Magnum 2/2x have seperate counters for these applications.

Usage

```
void vcount( int Loop, UINT Value );
UINT vcount( int Loop );
```

where:

Loop identifies which counter is being accessed. Legal values are 1 to 4.

Value specifies the desired counter value. Legal values are 0 to 2^{32}.

The vcount() getter function returns the current counter value.

Example

```
vcount( 1, 15 );
```

```
UINT c = vcount( 1 );
output(" VAR Counter 1 => %d", c );
```

3.23.8.2 errvar()

See Logic Pattern Related Functions.

Description

The `errvar()` function returns an integer value representing the vector address (VAR) of the first failing logic instructin (vector).

Note the following:

- The returned VAR value is an *absolute* address that can be converted to a pattern relative address using `find_var()`.
- Proper operation requires that the test pattern be executed using the stop-on-fail mode; i.e. `funtest( patname, error )`; See Executing Functional Tests.

Usage

```
int errvar();
```

`errvar()` returns the absolute vector address (VAR) of the first failing vector.

Example

```
int test_result = funtest( some_pat, error );
int first_fail = errvar();
output (" First Fail VAR (absolute) => %n", first_fail );
```

3.23.8.3 find_var()

See Logic Pattern Related Functions.

Description

Given an absolute logic pattern vector address (VAR) the `find_var()` function can be used to return the following:

- A pointer to the logic pattern containing the specified VAR.
- The offset of the specified VAR from the first vector of the test pattern; i.e. the zero-based pattern-relative address of the instruction.

The term absolute means relative to the start of logic vector memory (LVM).

Usage

```
BOOL find_var( DWORD var, Pattern **obj, DWORD *vOffset );
```

where:

`var` specifies the absolute logic vector memory address (VAR) of the target vector.

`obj` is a pointer to an existing `Pattern*` variable used to return a pointer to the pattern containing the specified **`var`**. `find_var()` will return FALSE if the specified **`var`** does not correspond to a valid logic pattern instruction.

`vOffset` is a pointer to an existing `DWORD` variable used to return the zero-based offset of the specified **`var`** from the start of the test pattern. If `find_var()` returns `FALSE` the value in **`vOffset`** is invalid.

`find_var()` returns `TRUE` if the specified **`var`** is within a user-defined Logic Test Pattern, otherwise FALSE is returned.

Example

The example consists of three parts:

- Test Block Code
- Example Test Pattern
- Example Output

Test Block Code

The following example uses the `addrs()` function to obtain the first VAR of the `myLogicPat` test pattern. That value is subsequently used by `find_var()` to get information for the 3rd vector in the pattern (+2), thus the offset value returned = 2. See Example Output.

```
DWORD var;
BOOL ok = addrs( myLogicPat, 0, &var ); // addrs()
if( ! ok ) output("  ERROR: addrs() returned an error");
else output(" First VAR for myLogicPat => %d", var );
```

```
Pattern* pat;
DWORD vOffset;
ok = find_var( var, &pat, &vOffset );
if( ! ok ) output("  ERROR: find_var() returned an error");
else
   output(" pat => %s, vOffset => %d",
        resource_name( pat ),
        vOffset );
}
```

Example Test Pattern

```
PATTERN( myLogicPat, logic )
% vec 0000000000000000
% vec 0000000000000000
% vec 0000000000000000 // Target vector
% vec 0000000000000000
   var done
```

Example Output

```
pat => myLogicPat, offset => 2
```

3.23.8.4 vecdata()

See Logic Pattern Related Functions.

Description

The `vecdata()` function is used to read or write a single logic state value to/from Logic Vector Memory (LVM).

When reading LVM a `VectorState` is returned. To write, a `VectorState` is speciifed as the third parameter. `VectorState` is an enumerated type used to specify vector data. Valid values for `VectorState` are:

- **`drive_lo`** represents the pattern source character “0”
- **`drive_hi`** represents the pattern source character “1”
- **`tristate`** represents the pattern source character “X”
- **`strobe_lo`** represents the pattern source character “L”

- **strobe_hi** represents the pattern source character “H”
- **strobe_valid** represents the pattern source character “V”
- **strobe_mid** represents the pattern source character “Z”

See Logic Vector Bit Codes for more details of how these pattern source characters are used.

Usage

The following function sets the value of the bit in Logic Vector Memory (LVM) for the specified pin:

```
void vecdata( int var, DutPin *pin, VectorState state );
```

The following function returns the value of the bit in LVM for the specified pin:

```
VectorState vecdata(int var,
                    DutPin *pin,
                    DutNum dut DEFAULT_VALUE( t_dut1 ) );
```

The following functions are used in DDR mode to access either the A-cycle or B-cycle data for the bit in LVM for the specified pin. These are first available in software release h1.1.23:

```
void vecdata( int var, DutPin *pin, VectorState state, int bank );
VectorState vecdata(int var,
                    DutPin *pin,
                    DutNum dut,
                    int bank DEFAULT_VALUE( 0 ) );
```

where:

var is the *absolute* vector address to be read or written. Note that the system software does not know whether this address is a valid location in a test pattern; i.e. the user is responsible.

pin identifies a single DutPin, which is used to identify which bit in Logic Vector Memory (LVM) is to be accessed. In Multi-DUT Test Program, the bit is modified for each DUT in the Active DUTs Set (ADS). The specified DutPin must be mapped to a signal pin in the Pin Assignment Table.

state is the desired VectorState value being written to the specified VAR, or the VectorState value being returned when reading LVM.

dut is used in Multi-DUT Test Programs to identify the DUT for which the **pin** is read from LVM. The specified DutPin must be mapped to a signal pin in the Pin Assignment Table.

bank in used with DDR mode test patterns to identify whether the A-cycle data (**bank** = 0) or B-cycle data (**bank** = 1) is being accessed.

The `vecdata()` getter function returns the value read from LVM as a `VectorState`.

Example

The following example writes absolute VAR address 100 for the ClockPin = '0'. In a Multi-DUT Test Program, the bit is modified for each DUT in the Active DUTs Set (ADS):

```
vecdata( 100, ClockPin, drive_low );
```

3.23.8.5 addrs()

See Logic Pattern Related Functions.

Description

The `addrs()` function can be used to determine the following information about a Logic Test Patterns, Memory Test Patterns, or Scan Test Patterns:

- The first APG microRAM address used
- The first logic vector memory address used. Applies to logic patterns only.
- The first scan vector memory address used. Applies to scan patterns only.
- The number of APG microRAM addresses used
- The number of logic vector addresses used
- The number of scan vector addresses used
- The number of scan bits used

Usage

```
BOOL addrs(  Pattern *obj,
             DWORD *mar,
             DWORD *var DEFAULT_VALUE( 0 ),
             DWORD *mLen DEFAULT_VALUE( 0 ),
             DWORD *vLen DEFAULT_VALUE( 0 ) );
BOOL addrs(  LogicVector *obj,
             DWORD *var,
             DWORD *len DEFAULT_VALUE( 0 ) );
```

```
BOOL addrs(  ScanPattern *obj,
             DWORD *sar,
             DWORD *sLen DEFAULT_VALUE( 0 ),
             DWORD *nPins DEFAULT_VALUE( 0 ) );
```

where:

obj is a pointer to the pattern of interest.

mar is a pointer to an existing DWORD variable used to return the absolute APG microcode address of the first instruction of the specified pattern.

var is a pointer to an existing DWORD variable used to return the absolute address of the first logic vector of the specified pattern.

mLen is optional, and is a pointer to an existing DWORD variable used to return the number of APG microRAM addresses used by the specified pattern.

vLen and **len** are optional, and are a pointer to an existing DWORD variable used to return the number of logic vector memory addresses used by the specified pattern.

sar is a pointer to an existing DWORD variable used to return the absolute address of the first vector of the specified scan pattern. This represents the location at which the scan pattern was loaded in scan memory .

sLen is optional, and is a pointer to an existing DWORD variable used to return the length of the specified scan pattern.

nPins is optional, and is a pointer to an existing DWORD variable used to return the number of scan bits used in the specified scan pattern.

addrs() returns TRUE if the specified pattern is valid, otherwise FALSE is returned.

Example

This example assumes a test pattern named “my_pat” exists .

```
DWORD first_mar, first_var, mar_count, var_count;
CString patname = "";
addrs ( my_pat, &first_mar, &first_var, &mar_count, &var_count);
output("%s First MAR => %d Count = %d : First VAR => %d Count = %d",
          patname = resource_name( my_pat ), // Lookup name
          first_mar, mar_count,
          first_var, var_count );
```

3.23.8.6 var_pinfunc()

See Logic Pattern Related Functions.

Description

The `var_pinfunc()` function can be used to set or get the three basic `PINFUNC` parameters from a logic vector.

These functions do not report useful information in simulation mode.

Usage

The following function sets the values in the specified vector:

```
BOOL var_pinfunc( DWORD var,
                  TSETNumber tset,
                  VihhNumber vihh,
                  PSNumber ps );
```

The following function gets the values from the specified vector:

```
BOOL var_pinfunc( DWORD var,
                  TSETNumber *tset,
                  VihhNumber *vihh,
                  PSNumber *ps );
```

where:

`var` specifies the vector address (VAR) of the target vector. This is an absolute address; i.e. relative to the start of vector memory. The `addrs()` function can be used to identify the first VAR of a specified target test pattern.

`tset` is used in two ways: in the *set* function it identifies the Time-set value to be set; in the *get* function it is used to return the time-set value from the specified vector. In the latter case, **`tset`** must be a pointer to an existing `TSETNumber` variable.

`vihh` is used in two ways: in the *set* function it identifies the VIHH Map value to be set; in the *get* function it is used to return the VIHH Map value from the specified vector. In the latter case, **`vihh`** must be a pointer to an existing `VihhNumber` variable.

`ps` is used in two ways: in the *set* function it identifies the Pin Scramble Map value to be set; in the *get* function it is used to return the Pin Scramble Map value from the specified vector. In the latter case, **`ps`** must be a pointer to an existing `PSNumber` variable.

The BOOL value returned by var_pinfunc() is not meaningful.

Example

The example consists of three parts:

- Test Block Code
- Example Test Pattern
- Example Output

Test Block Code

The following code gets the address of the first vector (absolute VAR) for the test pattern myPat, then gets the pinfunc values for the 3rd instruction in this pattern (VAR +2):

```
DWORD mar, var;
BOOL ok = addrs( myPat, &mar, &var ); // addrs()
if( ! ok ) output("  ERROR: invalid pattern passed to addrs()");

TSETNumber ts;
VihhNumber vihh;
PSNumber ps;

ok = var_pinfunc( (var +2), &ts, &vihh, &ps );

if( ! ok ) output("  ERROR: var_pinfunc() returned an error");
else
   output("ts => TSET%d, vihh => VIHH%d, ps => PS%d", ts, vihh, ps);
```

Example Test Pattern

```
PATTERN( myPat, logic)

% VEC     0000000000000000, TSET1, PS3
% VEC     LL00000000000000, TSET2, PS1
% VEC     0L00000000000000, TSET3, VIHH2, PS2
% VEC     0000000000000000, TSET4, PS1
% MAR  DONE
```

Example Output

```
ts => TSET3, vihh => VIHH2, ps => PS2
```

3.23.9 Scan Pattern Related Functions

See Scan Test Patterns.

- `errsar(), prevsar(), dutsar()`
- `find_sar()`
- `scandata()`
- `get_scanpatterns()`
- `load_scan_from_file()`

3.23.9.1 errsar(), prevsar(), dutsar()

See Scan Pattern Related Functions.

Description

The Maverick-II APG contains hardware which can capture the failing scan pattern address (SAR).

The failing SAR can be accessed from C-code using the `errsar()`, `prevsar()` and `dutsar()` functions. These operate much the same as `errmar()`, `errvar()`, etc.

To be useful, these functions require the test pattern be executed using the stop-on-fail option; i.e. `funtest(patname, error)`. In this mode, it takes 7 pattern cycles for the first failure to be pipelined from the DUT to the APG, causing the pattern to stop. These three functions allow access to 3 different SAR values, at different locations in the error pipline:

- `errsar()` returns the first failing SAR.
- `prevsar()` returns the SAR prior to the first failing SAR.
- `dutsar()` returns the SAR at the DUT when the pattern actually stops.

Note: as of 6/12/2008, the information above which specifies “...7 pattern cycles...” has not been updated for Magnum 1/2/2x and the number of cycles is likely different using these systems. This note will be removed when the documetation is corrected.

Usage

```
int prevsar();
int errsar();
int dutsar();
```

where the returned `integer` value is the SAR of the instruction as noted above.

Example

```
int psar = prevsar();
int esar = errsar();
int dsar = dutsar();
```

3.23.9.2 find_sar()

See Scan Pattern Related Functions.

Description

The `find_sar()` function is used to identify which scan pattern contains a specified absolute Scan Address (SAR).

Used in conjunction with `errsar()`, which records the absolute SAR at which a failing pattern stopped, the `find_sar()` function can be called to identify the scan pattern which contains that SAR.

Given a scan pattern, the `addrs()` will identify the absolute scan address at which the pattern is loaded.

Usage

```
BOOL find_sar(  DWORD sar,
                ScanPattern **scanPattern,
                DWORD *offset );
```

where:

`sar` is the absolute scan address (SAR) of interest.

`scanPattern` is a pointer to an existing `ScanPattern` pointer variable used to return a pointer to the scan pattern containing the specified **`sar`**.

offset is a pointer to an existing DWORD variable used to return the offset from scan pattern vector-0 when the specified **sar** is not the first vector in the scan pattern.

find_sar() returns TRUE if the specified sar is valid. If FALSE is returned the values in **scanPattern** and **offset** are invalid.

Example

```
BOOL ok = find_sar( 102, myScanPat, 13 );
```

3.23.9.3 scandata()

See Scan Pattern Related Functions.

Note: this function is not supported on Magnum 1/2/2x.

3.23.9.4 get_scanpatterns()

See Scan Pattern Related Functions.

Note: this function is not supported on Magnum 1/2/2x.

3.23.9.5 load_scan_from_file()

See Scan Pattern Related Functions.

Note: this function is not supported on Magnum 1/2/2x.

3.23.10 Board Functions

The following functions are used to access board-level information about the various PC board types in the Magnum 1/2/2x:

- `BoardPresent()`
- `board_type()`
- `SerialNumber()`
- PWA/PWB Number and Revision Get Functions

3.23.10.1 BoardPresent()

Description

The `BoardPresent()` function allows user C-code to interrogate the tester hardware to determine if a specific Site Assembly Board(HSB) is installed.

`BoardPresent()` only operates correctly when executed in a Site process; i.e. Site Begin Block, `PIN_ASSIGNMENTS()`, `PIN_SCRAMBLE()`, `INITIALIZATION_HOOK()`, Test Blocks or functions called from test blocks, etc.

Usage

```
BOOL BoardPresent( HSBBoard board );
```

where:

`HSBBoard` identifies a specific Site Assembly Board (HSB board). Legal values are of the `HSBBoard` enumerated type.

Example

The following example used `BoardPresent()` tp determine if the 2nd Site Assembly Board (HSB board) is installed:

```
if( OnSite() )
   BOOL b = BoardPresent ( t_hab2 );
```

3.23.10.2 board_type()

Documention not completed for Magnum 1/2/2x.

3.23.10.3 SerialNumber()

Documention not completed for Magnum 1/2/2x.

3.23.11 DUT Board I/O Port Functions

See DUT Board I/O Ports.

This section covers the following functions used to control the DUT Board I2C Bus, GPIO Port & SPI Port:

- Types, Enums, etc.
- I2C Bus Functions
- `gpio_mode_set()`
- `gpio_direction_set()`
- `gpio_value_set(), gpio_value_get()`
- `spi_cmd()`

3.23.11.1 Types, Enums, etc.

See DUT Board I/O Port Functions, I2C Bus Functions.

Description

The following enumerated types are used in support of the various I2C, DUT Board I/O Port Functions:

Usage

The `GPIOMode` enumerated type is used to set the mode for the upper 4-bits of the GPIO Port (see DUT Board I/O Ports):

```
enum GPIOMode { t_spi_mode, t_parallel_io_mode };
```

3.23.11.2 I2C Bus Functions

See DUT Board I/O Port Functions, DUT Board I/O Ports, I2C Bus.

Description

The `I2C_operation()` function is used to perform an I2C transaction with one device connected to the I2C Bus on the DUT board.

Each execution of `I2C_operation()` can write data to an I2C device and/or read data from an I2C device.

The `I2C_control` struct defines the information `I2C_operation()` uses to perform a transaction:

```
struct I2C_control {
   int target;
   bool ten_bit_target;
   int address;
   int address_length;
   unsigned __int8 *write_data;
   int write_data_length;
   int target_buffer_size;
   unsigned __int8 *read_data;
   int read_data_length;
   int actual_read_length;
};
```

`target` specifies the address of the target I2C device, which must be unique for each device on the I2C Bus. Device address 0x0 (Device 0) is reserved for Nextest use.

`ten_bit_target` specifies whether the value in **`target`** is a 7-bit value (FALSE) or a 10-bit value (TRUE).

address identifies the device register to be accessed. **address** is sent to the I2C Bus in 8-bit bytes, with the MSB sent first. User code must reverse the bytes in **address** if the target requires the LSB first.

address_length specifies the number of 8-bit chunks of **address** to send. Two options are available:

- **address_length** = 0 is used when the target device has a single register and an address is not required
- **address_length** = 1 to 4 specifies the number of 8-bit chunks of **address** required to access the device register. This is determined from the target device data sheet.

write_data is a pointer to the data to be written.

write_data_length specifies the number of bytes of **write_data** to be sent.

target_buffer_size specifies the size of the device's input buffer when the target device has one (typically an I2C memory device, EEPROM, etc.). In these devices, the input buffer temporarily stores data before it is written to the device's main memory array. The device's buffer size defines the maximum amount of data which can be written to the device in one I2C transaction. In effect, **target_buffer_size** divides **write_data** into **target_buffer_size** chunks which are written to the device in a single transaction, which is repeated until **write_data_length** bytes have been written. **target_buffer_size** must be set to 0 when the target device does not have an input buffer.

read_data is a pointer to an existing `unsigned __int8` array used to store data read from the target device. User code is responsible for allocating the necessary memory and, if appropriate, freeing it when done.

read_data_length specifies the number of bytes of data to be read from the target device.

actual_read_length will return the number of bytes actually read from the target device. This will either match **read_data_length** or be 0 because once the target device has acknowledged the read command the `I2C_operation()` code will continue to read **read_data_length** bytes, assuming valid data is being read.

In general, `I2C_operation()` is not targeted at performing both a device write operation and a read operation in the same transaction. For this reason, for a given execution of `I2C_operation()`, it is normal that only the `write_data` array or the `read_data` array be used i.e. one will be (should be) NULL. However, some I2C devices, in order to perform a read, need to first be sent some data, typically to configure the device in preparation for the read operation. And, this data must normally be sent in the same transaction as that which

performs the read. Thus, it may be necessary for both `write_data` and `read_data` to be used to perform a read transaction, but it should not be necessary for the `read_data` to be used when performing a write operation.

Usage

```
BOOL I2C_operation ( HSBBoard Board, I2C_control& my_I2C_op );
BOOL I2C_operation ( I2C_control& my_I2C_op );
```

where:

`Board` identifies one Site Assembly Board (HSB) to be accessed. This is only usable when Sites-per-Controller is > 1, allowing the master site controller to direct the I2C transaction to a site slaved to the master.

`my_I2C_op` is an instance of the `I2C_control` struct (see description), initialized with the various values used during the I2C transaction.

`I2C_operation()` returns FALSE if an error is detected, otherwise TRUE is returned.

Example

The following example uses 2 methods for initializing the data to be used by `I2C_operation()`. Both methods perform an indentical operation:

```
#define RSIZE 256
unsigned __int8 read_data[RSIZE];

struct I2C_control Ex1 = {
   0x4A,
   FALSE,
   0x1700,
   0x2,
   0x0,
   0x4,
   0x0,
   0x0,
   RSIZE,
   0 };

void myFunc(){
   I2C_control Ex2;
   Ex2.target = 0x4A;
   Ex2.ten_bit_target = FALSE;
   Ex2.address = 0x1700;
```

```
    Ex2.address_length = 2;
    Ex2.write_data = 0;
    Ex2.write_data_length = 4;
    Ex2.target_buffer_size = 0;
    Ex2.read_data = read_data;
    Ex2.read_data_length = RSIZE;
    Ex2.actual_read_length = 0;

    BOOL ok = I2C_operation( Ex1 );
    if( ! ok ) output("  ERROR: I2C_operation(Ex1) returned FALSE");

    ok = I2C_operation( Ex2 );
    if( ! ok ) output("  ERROR: I2C_operation(C1) returned FALSE");
}
```

3.23.11.3 gpio_mode_set()

See DUT Board I/O Port Functions, DUT Board I/O Ports.

Description

The `gpio_mode_set()` function is used to configure the upper 4-bits of the 7-bit GPIO Port:

- `t_spi_mode` configures these 4-bits to SPI mode, enabling the `spi_cmd()` to write/read correctly.
- `t_parallel_io_mode` configures these 4-bits to be used as part of the 7-bit GPIO Port. In this mode, `spi_cmd()` will return FALSE indicating an incorrect configuration for SPI Port use.

Usage

```
void gpio_mode_set( HSBBoard Board, GPIOMode mode );
```

where:

`Board` specifies which Site Assembly Board (HSB) is being accessed. Legal values are of the `HSBBoard` enumerated type.

`mode` specifies the desired GPIO Port operation mode. Legal values are of the `GPIOMode` enumerated type. See Description.

Example

```
gpio_mode_set( t_hsb1, t_parallel_io_mode );
gpio_mode_set( t_hsb1, t_spi_mode );
```

3.23.11.4 gpio_direction_set()

See DUT Board I/O Port Functions, DUT Board I/O Ports.

Description

The `gpio_direction_set()` function is used to set the per-bit I/O direction for the GPIO Port signals. Note the following:

- `gpio_direction_set()` uses a bit-wise mask to determine the I/O direction for each bit of the GPIO Port.
- Depending on the GPIO mode, set using `gpio_mode_set()`, the GPIO Port will consist of 3 or 7 bits. Thus, 3 or 7 bits must be set in the mask.
- A logic-1 mask bit sets the corresponding GPIO bit to write mode (drive).
- A logic-0 mask bit sets the corresponding GPIO bit to read mode (tri-state).
- The system software does not set a default mask i.e. using the GPIO Port without setting the direction mask is invalid. This is not checked by the system software.

Usage

```
void gpio_direction_set( HSBBoard Board, int mask );
```

where:

`Board` specifies which Site Assembly Board (HSB) is being accessed. Legal values are of the `HSBBoard` enumerated type.

`mask` is a bit-wise value where either 3 or 7 bits determines the I/O direction of the corresponding GPIO Port bit. See Description

Example

```
gpio_mode_set( t_hsb1, t_parallel_io_mode );
gpio_direction_set( t_hsb1, 0x3F );
gpio_value_set( t_hsb1, 0x25 );
```

```
gpio_direction_set( t_hsb1, 0x00)
int val = gpio_value_get( t_hsb1 );
```

3.23.11.5 gpio_value_set(), gpio_value_get()

See DUT Board I/O Port Functions, DUT Board I/O Ports.

Description

The `gpio_value_set()` function is used to write to the GPIO Port.

The `gpio_value_get()` function is used to read to the GPIO Port.

Note the following:

- Depending on the mode set using `gpio_mode_set()`, the GPIO Port consists of 3 or 7 bits.
- Each GPIO bit is configured to write/drive or read/tri-state using `gpio_direction_set()`.
- Using `gpio_value_set()`, only those bits set to the write/drive direction using `gpio_mode_set()` will actually drive. Using `gpio_value_get()`, only those bits set to the read/tri-state direction using `gpio_mode_set()` will actually be read. The system software does not check the per-bit direction i.e. user code is responsible for ensuring proper configuration.

Usage

```
void gpio_value_set( HSBBoard Board, int value );
int gpio_value_get( HSBBoard Board );
```

where:

`Board` specifies which Site Assembly Board (HSB) is being accessed. Legal values are of the `HSBBoard` enumerated type.

`value` specifies the value to write to the GPIO Port. Only the low 3 or 7 bits are used, depending on the mode set using `gpio_mode_set()`.

`gpio_value_get()` returns the value read from the specified **`Board`**. Only the low 3 or 7 bits are valid, depending on the mode set using `gpio_mode_set()`.

Example

See Example

3.23.11.6 spi_cmd()

See DUT Board I/O Port Functions, DUT Board I/O Ports.

Description

The `spi_cmd()` function is used to write and/or read data to/from the SPI Port on a specified Site Assembly Board. Note the following:

- As indicated, the `spi_cmd()` function is used to write to the SPI Port, read from the SPI Port, or both.
- A combined write/read transaction can be completed in a single execution of `spi_cmd()`. The write data is first sent then SPI Port is read.
- The four SPI Port (signals) are not usable unless the GPIO mode is set to `t_spi_mode` using `gpio_mode_set()`. `spi_cmd()` returns FALSE if this configuration is not correct.

Usage

```
BOOL spi_cmd(HSBBoard Board,
             int wrlen,
             unsigned __int8* wrdata,
             int rdlen,
             unsigned __int8* rddata,
             int* count );
```

where:

`Board` specifies which Site Assembly Board (HSB) is being accessed. Legal values are of the `HSBBoard` enumerated type.

`wrlen` specifies the number of values to write to the SPI Port from the **`wrdata`** array.

`wrdata` is an array of at least **`wrlen`** values to be written to the SPI Port.

`rdlen` specifies the number of values to read from the SPI Port into the **`rrdata`** array.

`rddata` is a pointer to an existing `unsigned __int8` array of at least **`rrlen`** length used to return the values read from the SPI Port.

count is a pointer to an existing `int` variable used to return the number of values actually returned in **rddata**.

`spi_cmd()` returns FALSE if the current GPIO mode is not `t_spi_mode`, as set using `gpio_mode_set()`, otherwise TRUE is returned.

Example

```
unsigned __int8 wr_data[] = { 0xA5,  0x5A,  0x69,  0x96 };
unsigned __int8 rd_data[ 1024 ];
int count;
gpio_mode_set( t_hsb1, t_spi_mode ); // gpio_mode_set()
BOOL ok = spi_cmd( t_hsb1, 4, wr_data, 32, rd_data, &count );
if( !ok ) output(" ERROR: spi_cmd() returned FALSE");
```

3.23.12 Loadboard Board Data Bits

Description

The `lbdata()` function is used to read and/or write data to/from the four loadboard data bits.

There are four loadboard data bits that can be manipulated on-the-fly from the pattern generator using the `LBDATA` operand in the `CHIPS` instruction of an APG instruction. Using the `lbdata()` function, these four loadboard data bits can also be set from user C-code, but only when the test pattern is not executing. When loadboard data bits are set using the `lbdata()` function they remain latched until `lbdata()` is called again or until an `LBDATA` instruction is executed in a test pattern.

The loadboard data bits are TTL level with IOH and IOL capability of 4.0mA and -8.0mA respectively.

Note: When changing the loadboard data bits using the `lbdata()` function, only the last APG pipeline stage is written with the new information. All earlier pipeline stages remain unchanged. If test pattern execution stops leaving one or more LBDATA instructions in the APG pipeline the `pipe_clear()` command which automatically executes at the start of the next pattern execution will cause the pipelined `LBDATA` bits to be sent to the DUT board, over-riding the bit states set up using the `lbdata()` function. This can be prevented by executing a `pipe_clear()` instruction before executing the `lbdata()` function, as shown in the example below.

Usage

The first two versions of `lbdata()` below write a new value to the last stage of the APG pipeline. The second two versions read the current state of the LBDATA bits.

```
void lbdata( int Value );
void lbdata( HSBBoard Board, int value );
int lbdata();
int lbdata( HSBBoard Board );
```

where:

`Value` is an integer (int) from 0 to 15 (0x0 to 0xF).

`Board` is used when when Sites-per-Controller > 1 to identify a specific Magnum 1/2/2x Site Assembly Board (HSB board).

The versions of `lbdata()` which return an `int` value are returning the value read from the hardware.

Example

The example n below writes the two low order loadboard data bits to zero and the two high order loadboard data bits to one.

```
pipe_clear();  // Flush tester pipeline to clear any undesirable
               // LBDATA states
lbdata( 0xc );// Write a C-hex value to loadboard data bits
```

3.23.13 DUT Board ID and DUT Board User Data Area

Description

This set of functions allow test programs to read information that is stored in the EEPROM on each DUT Board.

This information includes hardware rev numbers if they have been coded into the EEPROM. Note that DUT Board is another name for loadboard; these terms are often used interchangeably.

Two more functions allow you to read and write 256 WORDs in a user area of the EEPROM.

Usage

The following functions return the PWA/PWB number or revision information about the currently mounted DUT Board.

```
DWORD db_pwa()
DWORD db_pwa_rev()
DWORD db_pwb()
DWORD db_pwb_rev()
```

The following function returns the DUT board serial number for the currently mounted DUT Board:

```
DWORD db_id()
```

The following function reads and returns one word of **`Data`** from the DUT Board EEPROM, at the specified **`Address`**.

```
WORD db_data( BYTE Address )
```

The following function writes one word of **`Data`** to the DUT Board EEPROM, at the specified **`Address`**.

```
void db_data( BYTE Address, WORD Data )
```

3.23.13.1 PWA/PWB Number and Revision Get Functions

Documention not completed for Magnum 1/2/2x.

3.24 Data Buffer Memory Software (DBM)

See Data Buffer Memory (DBM) for an overview of DBM Architecture.

This section contains the following topics:

- Overview
- DBM DRAM Interleaving
- DBM Sequential Mode
- DBM Usage Rules
- DBM Data Widths
- Masked vs. Un-masked DBM Operations
- DBM & Multiple Sites-per-controller
- DBM Configuration Tables
- Types, Enums, etc.
- `dbm_config_set()`
- `dbm_config_get()`
- DBM Segment Selection
- `dbm_num_segments_get()`
- `dbm_fill()`
- `dbm_write()`
- `dbm_read()`
- `dbm_file_image_write()`
- `dbm_file_image_read()`
- DBM Data File Format
- DBM Address Masks
- `dbm_pattern_use()`
- `datbuf()`

Also see DBMTool.

3.24.1 Overview

See Data Buffer Memory Software (DBM), DBM Usage Rules.

The Data Buffer Memory (DBM) hardware is an optional component of the Algorithmic Pattern Generator (APG). When installed, the DBM is a source of stored test pattern data, which is accessed, cycle-by-cycle using the same X/Y address used to address the DUT. The DBM can be viewed as a large memory array with software configurable addressing, data width, address compression and data compression.

The DBM must be configured before use, using `dbm_config_set()`. The current DBM configuration can be retrieved using `dbm_config_get()`.

After being configured and prior to executing a test pattern which uses DBM contents the data stored in the DBM is loaded, normally from disk using `dbm_fill()`and/or `dbm_write()` and/or `dbm_file_image_read()`.

The number of APG Address Generator bits used to test the DUT is set using the `numx()` and `numy()` functions. These same X and Y address bits are *available* at the DBM to select the DBM address which is read using test pattern instructions which select the DBM as the APG data source (`DATGEN BUFBUF`, etc.) in each pattern cycle. However, arguments to `dbm_config_set()` determine the number of APG Address Generator X and Y address bits which will *actually* be used to access the DBM. Normally, this will be set up to match the DUT's X/Y address size (the values set using `numx()` and `numy()`) but not always (more below regarding DBM address compression).

During pattern execution, in cycles which select the DBM as the data source, data is read from DBM at the address generated by the X/Y APG Address Generators and output by the APG Data Generator. And, in cycles which write to the DBM (`DATGEN DBMWR`) the output of the APG Data Generator is written into the DBM at the address generated by the X/Y APG Address Generators.

The DBM DRAM Interleaving configuration determines the effective size of the DBM which can be increased by a factor of 4 when used at reduced access rates or by using sequential addressing. DBM Sequential Mode describes the DBM mode which increases the effective DBM size but requires the user's test pattern be designed to ensure a sequence of sequential DBM addresses are being read. Special rules apply, see DBM DRAM Interleaving and DBM Sequential Mode.

To use DBM address compression, DBM Address Masks are specified (see `dbm_config_set()`). When address compression is used, one or more X and/or Y address bits at the input to the DBM are ignored. In effect, this means that multiple addresses generated by the APG Address Generator will access the same DBM address.

Configuring the DBM also defines the DBM data width, which can be configured as x1, x2, x4, x8/9, x16/18, x32/36, also using `dbm_config_set()`. As the data width is reduced the effective DBM depth increases.

As indicated above, on a cycle-by-cycle basis, the DBM can be the source of pattern drive/strobe data (i.e. a DBM read operation) or can capture the output of the APG data generator (i.e. a DBM write operation). However, since the DBM architecture doesn't support bit-wise (data-width) *write* operations, a DBM write operation (`DATGEN DBMWR`) requires that the hardware actually perform a read/modify/write, to ensure that only the appropriate bits are modified based on the current DBM configuration. This is one basis for the DBM Usage Rules.

Note: beginning with software release h1.1.23, the following paragraph no longer applies. It was not completely deleted to support a transition period. See Note: for more information.

~~Proper operation of test patterns which use the DBM as a data source REQUIRES setting `dbm_pattern_use()` = TRUE. This advises the system software that all subsequent test patterns will use the DBM as a data source. This is required because the APG pipeline configuration must be modified when using the DBM. See Error Pipeline Requirements.~~

Once configured, if the DBM size (i.e. the number of used X/Y address bits) is smaller than the amount of installed DBM memory the system software automatically partitions the DBM into segments. Each segment is a separate section of DBM memory equal in size to the current DBM configuration. Multiple segments makes it possible for the DBM to store multiple data patterns, each in its own segment. The `dbm_segment_set()` function is used to select one DBM segment. The `dbm_segment_get()` function can be used to identify which segment is currently selected. The `dbm_num_segments_get()` function can be used to determine how many segments are available with the current DBM configuration.

The value specified using `x_fast_axis()` determines whether the APG Address Generator X-addresses or Y-addresses are used as the low-order addresses into the DBM.

The DBM can be initialized (loaded) from a disk file using `dbm_file_image_read()`. The file must conform to the DBM Data File Format. The current DBM contents can be saved (written) to a disk file using `dbm_file_image_write()`, which will generate a file in the DBM Data File Format.

Note: it is possible to load a DBM image file generated (`dbm_file_image_write()`) on Magnum 1 into a Magnum 2/2x DBM (but not vice-versa). See DBM Data File Format.

User code can load or modify DBM contents using `dbm_fill()` and `dbm_write()`. User code can read DBM contents using `dbm_read()`.

Using Magnum 1/2/2x, the DBM can be used to capture APG *generated* pattern data, as sourced from the APG Data Generator. This is done using the test pattern `DATGEN DBMWR` instruction. This capability can be used to:

- Accumulate or convert complex *generated* data pattern(s) into a stored data pattern.
- Analyze pattern operation by evaluating DBM contents after pattern execution completes.

Like the APG in general, the DBM has no direct parallel testing support. The DBM is one of the three APG data sources, selectable cycle-by-cycle in Memory Test Patterns. How the APG Data Generator outputs are mapped to individual DUT pins, including in parallel test applications, is controlled by the Pin Scramble Map and the PINFUNC Instruction in the test pattern.

Do read DBM Usage Rules.

3.24.2 DBM DRAM Interleaving

See Overview, DBM Usage Rules, `dbm_config_set()`.

The Data Buffer Memory (DBM) is implemented using DRAM. To operate at full tester speed requires that the DBM DRAMs be configured for 4-way interleaving, which means that the DBM data for each DUT address is actually stored in 4 different DBM DRAM addresses. Using this interleaved configuration, the DBM may be accessed completely randomly at full speed (50MHz), but the usable size of the DBM is 1/4 the size *potentially* available if interleaving was not used. Note that the Magnum 1/2/2x product specifications describe DBM size options for maximum spped (interleaved) use.

In some applications the effective DBM size can be increased by a factor of 4 by configuring the DBM for non-interleave operation. This is done using the `rate` argument to `dbm_config_set()`, which has the following options:

- `t_dbm_full_speed` (default): the DBM is configured for 4-way interleaving providing random access at full speed.
- `t_dbm_slow_speed`: DBM interleaving is disabled. This mode provides random access but at a reduced DBM access rate. See DBM Usage Rules.

- `t_dbm_sequential`: DBM interleaving is disabled but DBM read access speed is improved as compared to `t_dbm_slow_speed`. This is possible when a series of sequential addresses is read from the DBM. See DBM Sequential Mode and DBM Usage Rules. This option is first available in software release h2.2.7/h1.2.7.

During pattern execution, when interleaving is enabled, a pattern instruction which reads (outputs) a DBM value (`DATGEN BUFBUF`, `MAINBUF`, etc.) can read any 1 of the 4 addesses which store a given value (it doesn't matter which copy is read). However, when a pattern instruction writes a data value to the DBM 4 different DBM addresses must be written. This is one reason the maximum DBM write speeds are slower than the read speeds (see DBM Usage Rules) when interleaving is used.

3.24.3 DBM Sequential Mode

See DBM DRAM Interleaving.

Note: first available in software release h2.2.7/h1.2.7.

Description

Read DBM DRAM Interleaving first.

DBM DRAM Interleaving is used to provide maximum-speed DBM access at random addresses but has the effect of reducing the effective size of the DBM. The DBM has an optional configuration, called DBM Sequential Mode, which allows high-speed DBM read access without using DBM DRAM Interleaving. This maximizes the effective DBM size but requires that the user's test pattern generate X/Y addresses in a sequence of a specific length, as described below.

Normally, (i.e. in non-DBM Sequential Mode) the address being read from the DBM in each tester cycle is the address generated by X/Y APG Address Generators and sent to the DUT. The APG Address Generators are controlled by the user's test pattern, using the XALU and YALU pattern instructions. In DBM Sequential Mode, the DBM uses the address from the APG for the first DBM access in a given sequence but the remaining addresses in the sequence are generated internal to the DBM, more below.

The DBM Sequential Mode is enabled by setting the `rate` arguement to `dbm_config_set()` = `t_dbm_sequential`. Note the following:

- DBM DRAM Interleaving is disabled when DBM Sequential Mode is enabled..

- During pattern execution, the first DBM read instruction (DATGEN BUFBUF, etc.) in each sequence 4 DBM reads causes the DBM to latch the address from the APG Address Generator. Subsequently, the low-address bits in the next 3 DBM read instructions are generated by a 2-bit counter internal to the DBM hardware. This causes the fast axis address (see x_fast_axis()) to be incremented once for each DBM read instruction executed, until the sequence is complete. During these cycles the upper address bits are those latched in the first DBM read instruction in a sequence.
- Proper DBM sequential mode operation requires that the user's test pattern ensure that each address sequence begins with a modulo-4 address value.
- In DBM Sequential Mode test patterns, DBM read instructions are not required to occur in adjacent pattern instructions; i.e., non-DBM-accessing instructions may occur between the DBM-accessing instructions.
- Normally, it is critical that the user's test pattern also generate the exact same X/Y address sequence as that generated internal to the DBM. This means the address sequence must increment the LSB of the fast axis (see x_fast_axis()). For example, given the first address in each sequence = n, the test pattern must generate address n+1, n+2, n+3 to complete the sequence. If not, the addresses used to read data from the DBM will not match the addresses presented to read or write data to/from the DUT, which is normally BAD! Correct test pattern design can be tricky near maximum X and/or Y address boundaries. Note that the X/Y addresses can be doing other things in cycles which don't access the DBM.
- Magnum 1 DBM write operations (DATGEN DBMWR) are affected by DBM Sequential Mode and DBM DRAM Interleaving, but the rules documented in this section account for this. However...

Note: do not use a DBM write instruction (DATGEN DBMWR) within a sequence of 4 DBM reads (DATGEN BUFBUF, etc.). Do not use any DBM read instructions within a sequence of 4 DBM writes.

When DBM sequential mode is used, 4 sequential DBM addresses will always be accessed in each sequence, before a different DBM address sequence is started. The following table shows some examples:

APG Address at the start of the Sequence	Actual DBM Address Sequence	Comments
0x0	0x0 0x1 0x2 0x3	DBM operation is as expected.
0x3 (error)	0x0 0x1 0x2 0x3	DBM operation is not as expected: the 2 LSB bits from the APG Address Generator output are replaced by the DBM's internal 2-bit counter. This represents a user error: the test pattern must start each address sequence with a modulo-4 value.
0xFE4F (error)	0xFE4C 0xFE4D 0xFE4E 0xFE4F	

As indicated, the DBM address inputs come from both the X and the Y APG Address Generators. However, the low address bits can be from the X address generator (X-fast) or from the Y address generator (Y-fast). The default = X-fast but the selection (which also affects other APG operations) can be modified using the `x_fast_axis()` function. The 2-bit counter used in DBM Sequential Mode always determines the LSBs of the DBM address, independent of which address generator is used for the low addresses.

3.24.4 DBM Usage Rules

See Data Buffer Memory (DBM), Overview, DBM DRAM Interleaving.

Proper DBM operation depends on adherence to the following rules. Note that neither the hardware nor the system software checks for most rule violations; i.e. the user is responsible. Proper DBM operation will not occur if any rules are violated (and this will likely be quite difficult to diagnose:

1. When the DBM is used in a given test pattern, additional pattern cycles are required to pipeline errors to the branch-on-error logic. See Error Pipeline Requirements. It is the user's responsibility to design the test pattern accordingly.
2. As indicated in DBM Memory Size Options , the DBM has two effective sizes, which is determined by whether DBM DRAM Interleaving used, which is determined by the `rate` argument to `dbm_config_set()`. When `rate` = `t_dbm_slow_speed`, the test pattern must not use cycle periods any faster than 80nS. When rate = `t_dbm_full_speed` the DBM will operate correctly with a 20nS (50MHz) minimum cycle. When rate = `t_dbm_sequential`, the DBM will operate correctly with a 30nS (33MHz) minimum cycle.
3. The following table specifies additional DBM rules:

Table 3.24.4.0-1 DBM Operation Rules

1st Operation	to	2nd Operation	Rule
DBM Read (DATGEN BUFBUF, etc.)	→	DBM Read (DATGEN BUFBUF, etc.)	The other rules apply.
DBM Read (DATGEN BUFBUF, etc.)	→	DBM Write (DATGEN DBMWR)	16 tester cycles.
DBM Write (DATGEN DBMWR)	→	DBM Write (DATGEN DBMWR)	Based on DBM DRAM Interleaving: • t_dbm_slow_speed = 155nS • t_dbm_sequential = 155nS • t_dbm_full_speed = 171nS
DBM Write (DATGEN DBMWR)	→	DBM Read (DATGEN BUFBUF, etc.)	Minimum of 25 system clocks. Given that the Magnum 1 minimum cycle period (20nS) consists of 2 system clocks it will always be legal to perform DBM read in the 13th tester cycle after a DBM write. This can be reduced by increasing the cycle time(s) of the cycles (APG instructions) executed between the DBM write and the DBM read.
These rules represent the minimum time between the start of a pattern cycle doing the one DBM operation to the start of the next cycle doing a DBM operation.			

Note: beginning with software release h1.1.23, the following paragraph no longer applies. It was not completely deleted to support a transition period. See Note: for more information.

~~4. Prior to executing a test pattern which uses the DBM, the dbm_pattern_use() function must be executed (dbm_pattern_use(TRUE)). This is required because the APG pipeline configuration must be modified when using the DBM. This changes the number of pipeline cycles which must be executed when performing test pattern branch on error, branch on abort, etc. See Error Pipeline Requirements.~~

3.24.5 DBM Data Widths

See Data Buffer Memory Software (DBM).

Using Maverick-I/-II, separate DBM API functions were provided to read and write specific DBM data widths (e.g., `dbm_write_18()`, `dbm_write_36()`, etc.). The Magnum 1/2/2x DBM supports many more data width options, thus, adding new function names for each width option was not practical.

Instead, the `dbm_write()` and `dbm_read()` access functions were created. These write or read the correct number of bits based on the DBM data width configuration set using `dbm_config_set()`. Note that the legacy Maverick/-I/-II functions are not documented in the Magnum 1/2/2x manuals for this reason.

Note: while the various versions of the Maverick `dbm_write_xxx()` and `dbm_read_xxx()` functions do operate correctly on Magnum 1, the newer `dbm_write()` and `dbm_read()`functions are recommended for performance reasons. These functions use a newer DBM device driver, which is optimized to more efficiently access the DBM in the direction specified by `x_fast_axis()`.

Some DBM access functions have multiple versions (overloads) supporting different size data arguments. For example, a given function may support both `DWORD` (32-bits) and `__int64` (64-bits) for the data argument. The following rules describe operation when the current DBM data width configuration isn't the same size as the data width argument type.

Given the current DBM data width configuration = **`w`** (see `dbm_config_set()`):

- If a function's data argument size is greater than **`w`**:

  - For DBM writes, the low order **w** bits of the function's data argument will be written to the DBM; high order bits are silently discarded.
  - For DBM reads, the function will return the low order **w** bits; all unused high order bits are set = 0.
- If a function's data argument size is less than **w**:
  - For DBM writes, the function's argument effective size is extended, with the high order bits set = 0.
  - For DBM reads, the function will return the low order bits read from the DBM; high order bits that don't fit in the argument are silently discarded.

3.24.6 Masked vs. Un-masked DBM Operations

See Data Buffer Memory Software (DBM).

Note: do not confuse the topic discussed in this section, which discusses whether the APG Address Generator X/Y size configuration affects how the addresses into the DBM are treated, with the topic of DBM address masks (see DBM Address Masks), which allows the user to enable DBM address compression.

The functions which write values to or read values from the Data Buffer Memory (DBM) from the test program require one or more X/Y addresses to be specified, to identify which DBM address(es) is/are to be accessed.

Rather than check each address for validity vs. DBM size, vs. DBM X/Y configuration, and report errors, etc., the following methods are used. The term *masked* and *unmasked* distinguishes between the two methods:

Masked	Default. DBM address argument values are masked using the APG address masks, set using `numx()` or `numy()`. For example, if the APG address mask configuration is 11X, 9Y the X-address mask is 0x7FF (11 bits) and only those bits of the X-address argument are used. Using masked addresses has the effect of wrapping (rolling over, etc.) an address value which exceeds the size of the DUT. And, if the DBM address configuration matches the APG address mask configuration it also wraps the corresponding DBM address. All DBM functions which write/ read the DBM default to masked operation.
Unmasked	DBM address argument values are used as-is, and careless usage mostly results in undesirable results. To use *unmasked* operations requires explicitly including the `t_unmasked_access` argument to the functions which support it.
Note: if `dbm_write()`, `dbm_fill()` or `dbm_read()` attempts to access the DBM outside the current DBM X/Y configuration (see `dbm_config_set()`) a warning is generated and the function terminates.	

3.24.7 DBM & Multiple Sites-per-controller

See Data Buffer Memory (DBM), Data Buffer Memory Software (DBM).

When testing memory devices which have non-algorithmic data requirements (typically read-only data from ROMs, etc.) the DBM is commonly used to store non-algorithmic expect data (strobe data). When more than 64 pins are needed to test a DUT, Sites-per-Controller will be set to 2 or more.

When Sites-per-Controller > 1, multiple DBMs are be utilized when testing a given DUT. Each site (HSB board) has an APG with a corresponding DBM (if installed), thus when Sites-per-Controller = 2, for example, there are 2 APGs and 2 DBMs used when testing the DUTs connected to those sites. In general, much of the details are transparent to the user. However, additional considerations exist when using the DBM.

For example, given Sites-per-Controller = 2, pins a_1/b_1 through a_64/b_64 have a corresponding APG + DBM and pins a_65/b_65 through a_128/b_128 have a

corresponding, and different, APG + DBM. By design, the 2 APGs operate in lock-step, executing the same test pattern, generating the same X/Y address, etc.

When Sites-per-Controller > 1, if some of the DUT's data pins are connected to the first 64 pins and other DUT data pins are connected to the second 64 pins it is typical that the data pattern in the two DBMs will be different; i.e. the data in the DBM for the first 64 pins will not be identical to the data for the 2nd 64 pins. This means that each DBM must be loaded separately. The functions which write or load data into the DBM have a `HSBBoard` argument, used to identify one DBM to be accessed.

The following rules apply when Sites-per-Controller > 1:

- Using `dbm_config_set()`, all DBMs will be configured identically.
- Using `dbm_fill()`and `dbm_write()` if the `HSBBoard` argument is not specified all DBMs will be loaded identically. When the DBMs on each site require different data, each DBM must be loaded discretely, using the `HSBBoard` argument with `dbm_fill()`and/or `dbm_write()`.
- When using `dbm_file_image_read()` and `dbm_file_image_write()`, the `HSBBoard` argument must be used to explicitly save and load a discrete DBM data file for each DBM in use. More below.
- Using `dbm_segment_set()` selects the same segment for all DBMs.
- Using `dbm_config_get()`, `dbm_segment_get()`, `dbm_num_segments_get()`, `dbm_read()` and the getter version of `datbuf()` if the `HSBBoard` argument is not specified the information is retrieved from the DBM on the master controller board, otherwise it is retrieved from the DBM on the specified `HSBBoard`.

Other DBM-related functions also accept an `HSBBoard` `board` argument, useful only when Sites-per-Controller > 1. The general rules are:

- When Sites-per-Controller = 1, the `HSBBoard` argument is not useful, but if used only `t_hsb1` is valid.
- When Sites-per-Controller > 1, any *setter* functions executed without specifying an `HSBBoard` argument will write to the DBM on all sites identically. This does not apply when using `dbm_file_image_read()` and `dbm_file_image_write()`, see above.
- When Sites-per-Controller > 1, any *getter* functions executed without specifying an `HSBBoard` argument will return information from `t_hsb1`.
- The `HSBBoard` argument may only specify boards which are accessible to the master controller. A warning is generated if this rule is violated and the function returns immediately.

3.24.8 DBM Configuration Tables

See Data Buffer Memory Software (DBM).

This section describes why the Magnum 1/2/2x DBM documentation does *not* include a set of Configuration Tables similar to those provided in the Maverick-I/-II documentation.

As noted above, the Data Buffer Memory (DBM) hardware can be viewed as a large memory array with software configurable addressing. Regarding the DBM's X/ Y address configuration, note the following:

1. Unlike the Maverick-I/-II, the Magnum 1/2/2x have no set minimum number of APG Address Generator X and/or Y addresses which are used to access the DBM.
    - The Magnum 1 DBM supports 1-to-18 X-address bits and 1-to-16 Y-address bits, limited to a total of 32 combined address bits and by the amount of installed of DBM memory.
    - The Magnum 2/2x DBM supports 1-to-20 X-address bits and 1-to-20 Y-address bits, limited to a total of 40 combined address bits and by the amount of installed of DBM memory.
2. For reference, using Maverick-I/-II, the DBM hardware addressing scheme has both fixed addresses and software configurable addresses and the first 10 X-addresses are fixed and the first 8 Y-addresses are fixed. This sets the minimum X/Y configuration possible. For more information, see the *Maverick-I/-II Programmers Manual*.
3. Any unused X and/or Y addresses are available for DBM segment selection, using `dbm_segment_set()`. See DBM Segment Selection.

3.24.9 Types, Enums, etc.

See Data Buffer Memory Software (DBM).

Description

The following enumerated types are used in support of various Data Buffer Memory Software (DBM) functions:

Usage

The DbmPatternRate enumerated type determines the DBM DRAM Interleaving configuration, which affects the effective DBM size and how fast the DBM can be accessed. See DBM Architecture, DBM DRAM Interleaving, DBM Sequential Mode, DBM Usage Rules, dbm_config_set(), dbm_config_get().

```
enum DbmPatternRate {  t_dbm_full_speed,
                       t_dbm_slow_speed,
                       t_dbm_sequential,
                       t_dbm_speed_na };
```

The DbmAccessType enumerated type is used to select whether the DBM is accessed using masked or unmasked addresses. See Masked vs. Un-masked DBM Operations.

```
enum DbmAccessType { t_masked_access, t_unmasked_access };
```

The DbmFastDirection enumerated type is used to specify which DBM address axis will be the fast axis (LSB). See DBM Architecture, dbm_write(), dbm_read(). Note that t_auto_fast means use the result returned by x_fast_axis().

```
enum DbmFastDirection {  t_dbm_x_fast,
                         t_dbm_y_fast,
                         t_dbm_auto_fast };
```

3.24.10 dbm_config_set()

See Data Buffer Memory (DBM).

Description

The dbm_config_set() function is used to configure the Data Buffer Memory (DBM). The DBM must be configured before it can be used in a test pattern and before data can be loaded (more below).

dbm_config_set() sets the DBM size in the terms of used X/Y addresses and data width, and configures DBM DRAM Interleaving and DBM Sequential Mode (more below).

Note: the DBM Usage Rules must be followed for proper DBM operation.

The numx and numy arguments to dbm_config_set() configure the size of the DBM, in the terms of the number of APG Address Generator X addresses (numx) and Y addresses (numy) which are used to access the DBM during test pattern execution. See Overview.

Note: APG address masks, set using numx() and numy(), are ***NOT*** considered by the system software when configuring the DBM.

The width argument to dbm_config_set() configures the DBM data width. As the data width is reduced the effective DBM size (depth) increases. width is typically set to match the DUT's data width, which typically matches the APG Data Generator width, set using data_reg_width(). The Magnum 1/2/2x DBM supports the data widths of 1, 2, 4, 8/9, 16/18, 32, 36. The width configuration affects all DBM access functions which read or write to the DBM. See DBM Data Widths.

The rate argument is used by the system software to configure DBM DRAM Interleaving and to enable DBM Sequential Mode. This also affects the effective DBM size. The following rate options are available:

- t_dbm_full_speed = DBM is configured for 4-way interleaving, providing random access DBM operation up to 50MHz. See DBM Usage Rules.
- t_dbm_slow_speed = DBM is configured in non-interleaved mode, effectively increasing the DBM depth by a factor of 4 and providing random access but at a reduced DBM read rate. See DBM Usage Rules.
- t_dbm_sequential = DBM is configured in non-interleaved mode, effectively increasing the DBM depth by a factor of 4. The DBM may be accessed at up to 33MHz (not 50MHz) but DBM addressing is constrained and partially controlled by a counter internal to the DBM, see DBM Sequential Mode, DBM DRAM Interleaving and DBM Usage Rules.

Also note the following:

- During test program initialization, the system software detects the installed DBM size. This is used to validate any DBM configurations specified using dbm_config_set(). If a DBM configuration is attempted that is not possible, a warning is displayed and proper DBM operation should not be expected.
- When the DBM configuration is modified, the current DBM contents should be considered invalid.
- Executing dbm_config_set() resets the current DBM segment selection to segment 1. See DBM Segment Selection.
- Executing dbm_config_set() sets both DBM Address Masks to enable all address bits for the configuration being set. See dbm_masks_set().

- Changing the APG's X or Y address masks has no effect on DBM segmentation or configuration. When either of the APG's X or Y address masks is changed, whether interactively (PatternDebugTool) or via test program C-code (using `numx()` and/or `numy()`), the number of DBM segments is NOT automatically recomputed by the system software. Thus, in most applications, if either of `numx()` and/or `numy()` are modified the DBM will should be re-configured (`dbm_config_set()`) and reloaded.
- When the APG Address Generator fast axis state is changed (using `x_fast_axis()`), the DBM must be re-configured (`dbm_config_set()`) and reloaded. This is required because `dbm_config_set()` records the state returned by `x_fast_axis()`, and this state affects the subsequent operation of many DBM access functions.
- The `dbm_file_image_write()` and `dbm_file_image_read()` functions are available to write and read a DBM data to/from a disk file. The file is in a binary format, as described in DBM Data File Format. This format includes a DBM configuration, in the form of a header. The `dbm_read()` function will only load the file into the DBM if the current DBM configuration exactly matches the configuration saved in the file header.

Usage

```
void dbm_config_set(
          int numx,
          int numy,
          int width,
          DbmPatternRate rate DEFAULT_VALUE( t_dbm_full_speed ) );
```

The following function is the same as the previous but may be used when Sites-per-Controller > 1 to identify the target DBM by identifying its `HSBBoard`. See DBM & Multiple Sites-per-controller:

```
void dbm_config_set(
          HSBBoard board,
          int numx ,
          int numy,
          int width,
          DbmPatternRate rate DEFAULT_VALUE( t_dbm_full_speed ) );
```

where:

numx configures the DBM X-address size, and must be an integer value from 0 to 18.

numy configures the DBM Y-address size, and must be an integer value from 0 to 16.

Note: the total combined X + Y address bits is limited to 32.

width configures the DBM data width. Legal values are 1, 2, 4, 8, 9, 16, 18, 32, 36.

rate is optional and, if used, determines the DBM DRAM Interleaving configuration. Legal values are of the DbmPatternRate enumerated type. Default = t_dbm_full_speed. DO review DBM DRAM Interleaving, DBM Sequential Mode and DBM Usage Rules.

board may be used when Sites-per-Controller > 1 to identify the target DBM by identifying its HSBBoard. See DBM & Multiple Sites-per-controller.

Example

```
dbm_config_set( 8, 10, 8, t_dbm_full_speed );
```

3.24.11 dbm_config_get()

See Data Buffer Memory (DBM), Data Buffer Memory Software (DBM).

Description

The dbm_config_get() function is used to retrieve the current Data Buffer Memory (DBM) configuration, as set using dbm_config_set().

All of the arguments to dbm_config_get() are optional. Passing 0 for a given argument causes that value to not be returned.

Usage

```
void dbm_config_get(
            int* numx DEFAULT_VALUE( 0 ),
            int* numy DEFAULT_VALUE( 0 ),
            int* width DEFAULT_VALUE( 0 ),
            DbmPatternRate* rate DEFAULT_VALUE( 0 ) );
```

The following function is the same as the previous but may be used when Sites-per-Controller > 1 to identify a target DBM by identifying its HSBBoard. See DBM & Multiple Sites-per-controller:

```
void dbm_config_get(
            HSBBoard board,
            int* numx DEFAULT_VALUE( 0 ),
```

```
                int* numy DEFAULT_VALUE( 0 ),
                int* width DEFAULT_VALUE( 0 ),
                DbmPatternRate* rate DEFAULT_VALUE( 0 ) );
```

where:

numx is optional and, if used, is a pointer to an existing `int` variable, used to return the current DBM X-address size. Specify 0 if this value is not desired.

numy is optional and, if used, is a pointer to an existing `int` variable, used to return the current DBM Y-address size. Specify 0 if this value is not desired.

width is optional and, if used, is a pointer to an existing `int` variable, used to return the current DBM data width. Specify 0 if this value is not desired.

rate is optional and, if used, is a pointer to an existing `DbmPatternRate` variable, used to return the current setting. See `dbm_config_set()` and DBM DRAM Interleaving. Specify 0 if this value is not desired.

board may be used when Sites-per-Controller > 1 to identify the target DBM by identifying its `HSBBoard`. See DBM & Multiple Sites-per-controller.

Example

In the following example the 2nd argument (numy) is not returned (argument = 0):

```
int numx, width;
DbmPatternRate rate;
dbm_config_get( &numx, 0, &width, &rate );
```

3.24.12 DBM Segment Selection

See Data Buffer Memory (DBM), Data Buffer Memory Software (DBM).

Description

The `dbm_segment_set()` function is used to choose which DBM segment is currently selected. The `dbm_segment_get()` function is used to determine which DBM segment is currently selected. Note the following:

- When the Data Buffer Memory (DBM) is configured (see `dbm_config_set()`), if its size (i.e. the number of used X/Y address bits) is substantially smaller than the amount installed DBM memory the system software automatically partitions the

DBM into segments. Each segment is a separate section of DBM memory equal in size to the current DBM configuration. Multiple segments makes it possible for the DBM to store multiple data patterns, each in its own segment.

For example, a 256MB DBM may use up to 27 address bits to access the DBM. In a DBM configuration of 9X and 8Y (17 bits), 10 address bits are available for DBM segment selection. These 10 bits allow up to 1024 ($2^{\wedge 10}$) DBM segments.

- The number of available DBM segments is also affected by the DBM DRAM Interleaving configuration. When maximum interleaving is used the usable DBM size and thus the number of available segments is based solely on the number of used X/Y DBM addresses. When no interleaving is used the usable DBM size is 4 times larger, potentially increasing the number of DBM segments. See DBM Architecture, DBM DRAM Interleaving and DBM Sequential Mode.
- After the DBM is configured, the number of available DBM segments can be read using the `dbm_num_segments_get()` function.
- During the initial program load, DBM segment 1 is selected. The system software does not otherwise modify the DBM selection.
- Executing `dbm_config_set()` resets the current DBM segment selection = segment 1.
- Using `dbm_segment_set()`, attempting to select an invalid segment results in an error message and the currently selected DBM segment remains selected.
- Once a segment is selected, any *masked* reads from or writes to the DBM will occur only in that segment; however, *unmasked* read/writes may access other segments. See Masked vs. Un-masked DBM Operations.
- During test pattern execution, only the currently selected segment is accessed.
- When either of the APG's X or Y address masks is changed, whether interactively (PatternDebugTool) or via test program C-code (using `numx()` and/or (`numy()`), the number of DBM segments is NOT automatically recomputed by the system software. Thus, when either of `numx()` and/or (`numy()` change the APG configuration the DBM must also be re-configured (`dbm_config_set()`) and reloaded.

Usage

```
void dbm_segment_set( int segment );
int dbm_segment_get();
```

The following functions are the same as the previous but may be used when Sites-per-Controller > 1 to identify a target DBM by identifying its `HSBBoard`. See DBM & Multiple Sites-per-controller:

```
void dbm_segment_set( HSBBoard board, int segment );
int dbm_segment_get( HSBBoard board );
```

where:

segment identifies the DBM segment to be selected.

board may be used when Sites-per-Controller > 1 to identify the target DBM by identifying its HSBBoard. See DBM & Multiple Sites-per-controller.

dbm_segment_get() returns the currently selected DBM segment.

Example

The following example selects DBM segment 5 as the active segment. This implies that the currently installed DBM is at least eight times larger than the DUT being tested:

```
dbm_segment_set( 5 );
```

The following example returns the current DBM segment selection:

```
int segment = dbm_segment_get();
```

3.24.13 dbm_num_segments_get()

See Data Buffer Memory (DBM), Data Buffer Memory Software (DBM).

Description

The dbm_num_segments_get() function is used to determine how many DBM segments are currently available, which is based on the current the DBM configuration, as set using dbm_config_set(). See DBM Segment Selection.

Usage

```
int dbm_num_segments_get();
```

The following function is the same as the previous but may be used when Sites-per-Controller > 1 to identify a target DBM by identifying its HSBBoard. See DBM & Multiple Sites-per-controller:

```
int dbm_num_segments_get( HSBBoard board );
```

where:

board may be used when Sites-per-Controller > 1 to identify the target DBM by identifying its HSBBoard. See DBM & Multiple Sites-per-controller.

dbm_num_segments_get() returns the number of available DBM segments.

Example

```
output(" Number of DBM segments => %d", dbm_num_segments_get() );
```

3.24.14 dbm_fill()

See Data Buffer Memory (DBM), Data Buffer Memory Software (DBM).

Description

The dbm_fill() function is used to write a specified value to a rectangular area of DBM addresses. Note the following:

- The rectangular area of the DBM to be written is identified by locating the upper left corner (xstart/ystart) and lower right corner (xstop/ystop).
- The value written to each DBM address is specified using the data argument.
- The number of bits actually written is affected by the current DBM data width configuration, set using dbm_config_set(). See DBM Data Widths.
- By default, dbm_fill() uses *masked* DBM addresses. The type argument can be used to change operation to use *unmasked* DBM addresses. See Masked vs. Un-masked DBM Operations.
- If dbm_fill() attempts to write to addresses outside the current DBM X/Y configuration (see dbm_config_set()) a warning is generated and the function terminates without modifying the DBM.
- dbm_fill() operations ignore any non-default DBM Address Masks i.e. all configured DBM address bits are used.
- Also see dbm_write().

Usage

The following function writes to the currently selected DBM segment:

```
void dbm_fill(
          DWORD xstart,
          DWORD xstop,
```

```
            DWORD ystart,
            DWORD ystop,
            __int64 data,
            DbmAccessType type DEFAULT_VALUE( t_masked_access ) );
```

The following function is the same as the previous but may be used when Sites-per-Controller > 1 to identify a target DBM by identifying its HSBBoard. See DBM & Multiple Sites-per-controller:

```
void dbm_fill(
            HSBBoard board,
            DWORD xstart,
            DWORD xstop,
            DWORD ystart,
            DWORD ystop,
            __int64 data,
            DbmAccessType type DEFAULT_VALUE( t_masked_access ) );
```

where:

xstart, **xstop**, **ystart**, and **ystop** specify a rectangle of DBM X and Y addresses to be written. The following rules apply:

- 0 <= **xstart** <= **xstop** <= Max DBM X-address (set by dbm_config_set())
- 0 <= **ystart** <= **ystop** <= Max DBM Y-address (set by dbm_config_set())

data specifies a single value to be written to each address in the specified area of the DBM.

type is optional and, if used, specifies whether masked or unmasked addresses are used to access the DBM. See Description. Legal values are of the DbmAccessType enumerated type. Default = t_masked_access (recommended).

board may be used when Sites-per-Controller > 1 to identify the target DBM by identifying its HSBBoard. See DBM & Multiple Sites-per-controller.

Example

The following example will fill a rectangular area of the DBM with the value 0xA5. The area written is bounded by X0..X127 and Y0..Y63, inclusive:

```
dbm_fill( 0, 127, 0, 63, 0xA5);
```

3.24.15 dbm_write()

See Data Buffer Memory (DBM), Data Buffer Memory Software (DBM).

Description

The `dbm_write()` function writes specified value(s) to one or more DBM addresses.

See Note: regarding the use of the Maverick `dbm_write_xxx()` and `dbm_read_xxx()` functions on Magnum 1/2/2x.

All versions of `dbm_write()` are affected by the current DBM data width configuration (set using `dbm_config_set()`). See DBM Data Widths.

`dbm_write()` operations ignore any non-default DBM Address Masks i.e. all configured DBM address bits are used.

Seven basic versions (overloads) of `dbm_write()` are available to provide the capabilities noted below:

- The first 3 versions (below) sequentially write the number of values specified by the `count` argument from the `pData` array to the DBM, beginning at the *electrical* address specified by the `addr` argument (see Logical vs. Physical, vs. Electrical Addresses). The value of `x_fast_axis()` at the time `dbm_config_set()` was last executed determines how the electrical address is interpreted. Three functions provide for 3 different data sizes. These 3 versions are equivalent to the Maverick-I/-II `load_dbm()` function.
- The 4th and 5th versions of `dbm_write()` (below) write a single `data` value to a single *logical* DBM address specified by the `x` and `y` arguments. By default, `dbm_write()` uses *masked* DBM addresses. The `type` argument can be used to change operation to use *unmasked* DBM addresses. See Masked vs. Un-masked DBM Operations. Two functions provide for 2 different data types.
- The 6th and 7th versions of `dbm_write()` (below) sequentially write `count` values from the `pData` array to a rectangle of DBM addresses. The upper-left corner of the rectangle is located using the `xstart`/`ystart` arguments; the lower-right corner by the `xstop`/`ystop` arguments. The *direction* that data is written to the DBM, i.e. whether the X or Y DBM address is incremented fast, is controlled by the optional `fast` argument, which defaults to the value returned by `x_fast_axis()` at the time `dbm_config_set()` was last executed. Note that `dbm_write()` performance will be slower if `fast` is different than this default value. By default, `dbm_write()` uses *masked* DBM addresses. The `type`

argument can be used to change operation to use *unmasked* DBM addresses. See Masked vs. Un-masked DBM Operations. Two functions provide for 2 different data types.

The following rules apply and are checked:

- 0 <= `xstart` <= `xstop` <= Max DBM X-address (set by `dbm_config_set()`)
- 0 <= `ystart` <= `ystop` <= Max DBM Y-address (set by `dbm_config_set()`)
- `pData` must not be NULL
- In the versions of `dbm_write()` which use the `xstart/ystart/xstop/ystop` arguments the `count` must be >= than the number of addresses in the described rectangle.

If any rules are violated a warning is issued and the DBM is not modified.

Also see `dbm_fill()`.

Usage

The following functions write one or more DBM addresses with values from an array of values. See Description:

```
void dbm_write( __int64 addr, BYTE* pData, int count );
void dbm_write( __int64 addr, DWORD* pData, int count );
void dbm_write( __int64 addr, __int64* pData, int count );
```

The following functions write the specified data value to a single DBM address. See Description:

```
void dbm_write(
          DWORD x,
          DWORD y,
          DWORD data,
          DbmAccessType type DEFAULT_VALUE( t_masked_access ) );
void dbm_write(
          DWORD x,
          DWORD y,
          __int64 data,
          DbmAccessType type DEFAULT_VALUE( t_masked_access ) );
```

The following functions write one or more DBM addresses with values from an array of values. See Description:

```
void dbm_write(
          DWORD xstart,
          DWORD xstop,
          DWORD ystart,
          DWORD ystop,
          DWORD* pData,
          int count,
          DbmFastDirection fast DEFAULT_VALUE( t_dbm_auto_fast ),
          DbmAccessType type DEFAULT_VALUE( t_masked_access ) );
void dbm_write(
          DWORD xstart,
          DWORD xstop,
          DWORD ystart,
          DWORD ystop,
          __int64* pData,
          int count,
          DbmFastDirection fast DEFAULT_VALUE( t_dbm_auto_fast ),
          DbmAccessType type DEFAULT_VALUE( t_masked_access ) );
```

The following functions are the same as the previous set but may be used when Sites-per-Controller > 1 to identify the target DBM by identifying its HSBBoard. See DBM & Multiple Sites-per-controller:

```
void dbm_write(
          HSBBoard board,
          __int64 addr,
          BYTE* pData,
          int count );
void dbm_write(
          HSBBoard board,
          __int64 addr,
          DWORD* pData,
          int count );
void dbm_write(
          HSBBoard board,
          __int64 addr,
          __int64* pData,
          int count );
```

```
void dbm_write(
          HSBBoard board,
          DWORD x,
          DWORD y,
          DWORD data,
          DbmAccessType type DEFAULT_VALUE( t_masked_access ) );
void dbm_write(
          HSBBoard board,
          DWORD x,
          DWORD y,
          __int64 data,
          DbmAccessType type DEFAULT_VALUE( t_masked_access ) );
void dbm_write(
          HSBBoard board,
          DWORD xstart,
          DWORD xstop,
          DWORD ystart,
          DWORD ystop,
          DWORD* pData,
          int count,
          DbmFastDirection fast DEFAULT_VALUE( t_dbm_auto_fast ),
          DbmAccessType type DEFAULT_VALUE( t_masked_access ) );
void dbm_write(
          HSBBoard board,
          DWORD xstart,
          DWORD xstop,
          DWORD ystart,
          DWORD ystop,
          __int64* pData,
          int count,
          DbmFastDirection fast DEFAULT_VALUE( t_dbm_auto_fast ),
          DbmAccessType type DEFAULT_VALUE( t_masked_access ) );
```

where:

addr identifies the starting *electrical* address to be written. See Description.

pData is an array of (at least) **count** values to be written to the DBM.

count specifies the number of values written from the **pData** array to the DBM.

x and **y** identify a single DBM address to be written.

data specifies the value to be written to the DBM address specified by **x** and **y**.

type is optional and, if used, specifies whether masked or unmasked addresses are used to access the DBM. See Description. Legal values are of the `DbmAccessType` enumerated type. Default = `t_masked_access` (recommended).

xstart, **xstop**, **ystart**, and **ystop** define a rectangle of DBM X and Y addresses to be written.

fast is optional and, if used, determines how DBM addresses are sequentially incremented when writing multiple values. Legal values are of the `DbmFastDirection` enumerated type. Default = `t_dbm_auto_fast` i.e. the value returned by `x_fast_axis()` at the time `dbm_config_set()` was last executed. See Description.

board may be used when Sites-per-Controller > 1 to identify the target DBM by identifying its `HSBBoard`. See DBM & Multiple Sites-per-controller.

Example

The following example writes the 4 sequential values from the `vals` array to the DBM, beginning at electrical address 0x100:

```
BYTE vals[] = { 0x0, 0x1, 0x2, 0x3 };
dbm_write( 0x100, vals, 4 );
```

The following example writes the value 0x5A to the DBM at X-address 13, Y-address 9:

```
dbm_write( 13, 9, 0x5A );
```

The following example writes 8 sequential values from the `vals` array to the DBM, beginning at logical address 0. Note that the last 2 values in the `vals` array are not used. Since the `fast` argument is `t_dbm_x_fast` the X-address is incremented fast i.e. the 2nd DBM address written is X = 1, Y = 0, the third address written is X = 2, Y = 0, etc.

```
BYTE vals[] = {0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0xFF, 0xFF};
dbm_write( 0, 3, 0, 1, vals, 8, t_dbm_x_fast );
```

3.24.16 dbm_read()

See Data Buffer Memory (DBM), Data Buffer Memory Software (DBM).

Description

The `dbm_read()` function is used to read one or more values from the DBM.

See Note: regarding the use of the Maverick dbm_write_xxx() and dbm_read_xxx() functions on Magnum 1/2/2x.

All versions of dbm_read() are affected by the current DBM data width configuration (set using dbm_config_set()). See DBM Data Widths.

dbm_read() operations ignore any non-default DBM Address Masks i.e. all configured DBM address bits are used.

Five basic versions (overloads) of dbm_read() are available to provide the capabilities noted below. An additional five versions are identical except for the initial HSBBoard argument: these may be used when Sites-per-Controller > 1 to identify the target DBM by identifying its HSBBoard:

- The first 2 versions (below) sequentially read the number of values specified by the count argument from the DBM into the pData array, beginning at the *electrical* address specified by the addr argument (see Logical vs. Physical, vs. Electrical Addresses). The value returned by x_fast_axis() at the time dbm_config_set() was last executed determines how the electrical address is interpreted. Multiple functions provide for different data sizes. These versions are equivalent to the Maverick-I/-II read_dbm() function.
- The 3rd version of dbm_read() (below) reads the value from a single *logical* DBM address specified by the x and y arguments and returns the value. By default, dbm_read() uses *masked* DBM addresses. The type argument can be used to change operation to use *unmasked* DBM addresses. See Masked vs. Un-masked DBM Operations.
- The 4th and 5th versions of dbm_read() (below) sequentially read count values from the DBM into the pData array from a rectangle of DBM addresses. The upper-left corner of the rectangle is located using the xstart/ystart arguments; the lower-right corner by the xstop/ystop arguments. The *direction* that data is read from the DBM (i.e. whether the X or Y DBM address is incremented fast) is controlled by the optional fast argument, which defaults to the value returned by x_fast_axis() the last time dbm_config_set() was executed. Note that dbm_read() performance will be slower if fast is different than this default value. By default, dbm_read() uses *masked* DBM addresses. The type argument can be used to change operation to use *unmasked* DBM addresses. See Masked vs. Un-masked DBM Operations. Two functions provide for 2 different data sizes.

The following rules apply and are checked:

- 0 <= xstart <= xstop <= Max DBM X-address (set by dbm_config_set())
- 0 <= ystart <= ystop <= Max DBM Y-address (set by dbm_config_set())

- pData must not be NULL
- In the versions of dbm_read() which use the xstart/ystart/xstop/ystop arguments the count must be >= than the number of addresses in the described rectangle.

If any rules are violated a warning is issued and proper operation is not likely.

Usage

The following functions read one or more DBM addresses into an array. See Description:

```
void dbm_read( __int64 addr, DWORD* pData, int count );
void dbm_read( __int64 addr, __int64* pData, int count );
```

The following function reads and returns the value from one DBM address. See Description:

```
__int64 dbm_read(
          DWORD x,
          DWORD y,
          DbmAccessType type DEFAULT_VALUE( t_masked_access ) );
```

The following functions read one or more DBM addresses into an array. See Description:

```
void dbm_read(
          DWORD xstart,
          DWORD xstop,
          DWORD ystart,
          DWORD ystop,
          DWORD* pData,
          int count,
          DbmFastDirection fast DEFAULT_VALUE( t_dbm_auto_fast ),
          DbmAccessType type DEFAULT_VALUE( t_masked_access ) );
void dbm_read(
          DWORD xstart,
          DWORD xstop,
          DWORD ystart,
          DWORD ystop,
          __int64* pData,
          int count,
          DbmFastDirection fast DEFAULT_VALUE( t_dbm_auto_fast ),
          DbmAccessType type DEFAULT_VALUE( t_masked_access ) );
```

The following functions are the same as the previous set but may be used when Sites-per-Controller > 1 to identify the target DBM by identifying its HSBBoard. See DBM & Multiple Sites-per-controller:

```
__int64 dbm_read(
          HSBBoard board,
          DWORD x,
          DWORD y,
          DbmAccessType type DEFAULT_VALUE( t_masked_access ) );

void dbm_read( HSBBoard board,
               __int64 addr,
               DWORD* pData,
               int count );

void dbm_read( HSBBoard board,
               __int64 addr,
               __int64* pData,
               int count );

void dbm_read(
          HSBBoard board,
          DWORD xstart,
          DWORD xstop,
          DWORD ystart,
          DWORD ystop,
          DWORD* pData,
          int count,
          DbmFastDirection fast DEFAULT_VALUE( t_dbm_auto_fast ),
          DbmAccessType type DEFAULT_VALUE( t_masked_access ) );

void dbm_read(
          HSBBoard board,
          DWORD xstart,
          DWORD xstop,
          DWORD ystart,
          DWORD ystop,
          __int64* pData,
          int count,
          DbmFastDirection fast DEFAULT_VALUE( t_dbm_auto_fast ),
          DbmAccessType type DEFAULT_VALUE( t_masked_access ) );
```

where:

addr identifies the starting *electrical* address to be read. See Description.

pData is a user-defined array of (at least) **count** values used to return values read from the DBM.

count specifies the number of values to read from the DBM and return in the **pData** array.

x and **y** identify a single DBM address to be read.

type is optional and, if used, specifies whether masked or unmasked addresses are used to access the DBM. See Description. Legal values are of the DbmAccessType enumerated type. Default = t_masked_access (recommended).

xstart, **xstop**, **ystart**, and **ystop** define a rectangle of DBM X and Y addresses to be read.

fast is optional and, if used, determines how DBM addresses are sequentially incremented when reading multiple values. Legal values are of the DbmFastDirection enumerated type. Default = t_dbm_auto_fast i.e. the current x_fast_axis() selection determines which DBM address is incremented fast.

board may be used when Sites-per-Controller > 1 to identify the target DBM by identifying its HSBBoard. See DBM & Multiple Sites-per-controller.

The version of dbm_read() which returns __int64 returns the value read from the DBM.

Example

The following example reads 4 sequential values from the DBM into the vals array, beginning at electrical address 0x100:

```
BYTE vals[ 4 ];
dbm_read( 0x100, vals, 4 );
```

The following example reads and returns the value from the DBM at X-address 13, Y-address 9:

```
__int64 value = dbm_read( 13, 9 );
```

The following example reads 8 sequential values from the DBM into the vals array, beginning at logical address 0. Since the fast argument is t_dbm_x_fast the X-address is incremented fast i.e. the 2nd DBM address read is X = 1, Y = 0, the third address read is X = 2, Y = 0, etc.

```
BYTE vals[ 8 ];
dbm_read( 0, 3, 0, 1, vals, 8, t_dbm_x_fast );
```

3.24.17 dbm_file_image_write()

See Data Buffer Memory (DBM), Data Buffer Memory Software (DBM).

Description

The `dbm_file_image_write()` function is used to save (write) all or part of the data in the currently selected DBM segment to a file on disk. Note the following:

- The `filename` argument identifies the destination file on disk.
- **NO FILE CLOBBER** checks are made i.e. if the specified `filename` exists (and is writable) it will be silently over-written, regardless of content.
- If `filename` doesn't include an absolute path the file will be located relative to the test program executable file, typically in the test program *Debug*\ folder.
- Only data from the currently selected DBM segment is written, not the entire DBM. See DBM Segment Selection.
- `dbm_file_image_write()` operation ignores any non-default DBM Address Masks i.e. all configured DBM address bits are used.
- When multiple DBM locations are written, the DBM address is incremented based on the value returned by `x_fast_axis()` at the time `dbm_config_set()` was last executed. This sets the fast parameter in the DBM Data File Format header (more below).
- The `xstart`, `xstop`, `ystart` and `ystop` arguments are optional. If none of these arguments are specified the entire DBM segment is written to the file. Using these values describes a rectangular area of the DBM to be written to the file.
- When any of `xstart`, `xstop`, `ystart` and `ystop` arguments are specified the following rules apply:
  - 0 <= `xstart` <= `xstop` <= Max DBM X-address (set by `dbm_config_set()`)
  - 0 <= `ystart` <= `ystop` <= Max DBM Y-address (set by `dbm_config_set()`)

  If any of these rules are violated `dbm_file_image_write()` exits immediately, without writing to the file.
- The following errors are also checked. Any of these errors cause `dbm_file_image_write()` to exit without writing to the file:
  - `filename` is NULL
  - `filename` cannot be opened or written.

- The file written is in a binary format, as described in DBM Data File Format. This format includes, in the form of a header, the current DBM configuration (set using `dbm_config_set()`) at the time the file is written. The `dbm_file_image_read()` function is available to read a previously saved DBM data file into the DBM. `dbm_file_image_read()` will only load the file into the DBM if the DBM configuration at that time matches the configuration saved in the file header.

Usage

```
BOOL dbm_file_image_write(
                        LPCTSTR filename,
                        DWORD xstart DEFAULT_VALUE( 0x0 ),
                        DWORD xstop DEFAULT_VALUE( 0xffffffff ),
                        DWORD ystart DEFAULT_VALUE( 0x0 ),
                        DWORD ystop DEFAULT_VALUE( 0xffffffff ) );
```

The following function is the same as the previous but may be used when Sites-per-Controller > 1 to identify the target DBM by identifying its `HSBBoard`. See DBM & Multiple Sites-per-controller:

```
BOOL dbm_file_image_write(
                        HSBBoard board,
                        LPCTSTR filename,
                        DWORD xstart DEFAULT_VALUE( 0x0 ),
                        DWORD xstop DEFAULT_VALUE( 0xffffffff ),
                        DWORD ystart DEFAULT_VALUE( 0x0 ),
                        DWORD ystop DEFAULT_VALUE( 0xffffffff ) );
```

where:

`filename` specifies the disk file to be written.

`xstart`, **`xstop`**, **`ystart`**, and **`ystop`** are optional and, if used, define a rectangle of DBM X and Y addresses to be written.

`board` may be used when Sites-per-Controller > 1 to identify the target DBM by identifying its `HSBBoard`, see DBM & Multiple Sites-per-controller.

`dbm_file_image_write()` returns TRUE if the file write operation had no errors otherwise FALSE is returned.

Example

```
fname[] = "D:\myPath\thatPath\thisFile";
BOOL ok = dbm_file_image_write( fname );
if( !ok ) output("ERROR: dbm_file_image_write() returned FALSE");
```

3.24.18 dbm_file_image_read()

See Data Buffer Memory (DBM), Data Buffer Memory Software (DBM).

Description

The `dbm_file_image_read()` function is used to load data from a disk file into the currently selected DBM segment. Note the following:

- A DBM file is a binary file that contains a DBM File Header followed by one or more DBM File Data Records. `dbm_file_image_read()` will only load the file into the DBM if the configuration defined in this header matches the current DBM configuration, as set using `dbm_config_set()`.
- The `filename` argument identifies the disk file to load.
- If `filename` doesn't include an absolute path the file will be located relative to the test program executable file, typically in the test program *Debug*\ folder.
- Only the currently selected DBM segment is modified, not the entire DBM. See DBM Segment Selection.
- `dbm_file_image_read()` will only load a file which conforms to the DBM Data File Format. This format includes, in the form of a header, a DBM configuration. `dbm_file_image_read()` will only load the file to the DBM if this configuration matches the current DBM configuration, as set using `dbm_config_set()`.
- `dbm_file_image_read()` operation ignores any non-default DBM Address Masks i.e. all configured DBM address bits are used.
- The file read into the DBM may or may not completely define the current DBM segment. It is possible, when creating the file, to limit the values to a defined a rectangle of DBM addresses (see DBM Data File Format). When such a file is read into the DBM, only those addresses are modified.
- `dbm_file_image_write()` is available to create a DBM data file by writing data from the currently selected DBM segment to a specified file. See `dbm_file_image_write()`.

- The following errors are checked. Any of these errors cause `dbm_file_image_read()` to exit without modifying the DBM:
  - `filename` is NULL
  - `filename` cannot be opened or read.
  - The disk file does not contain a valid header (see DBM Data File Format).
  - The current DBM configuration does not match the configuration read from the disk file.
- Other errors can occur which result in a partially configured DBM (file read error, for example). The return value from `dbm_file_image_read()` should always be tested.

Usage

```
BOOL dbm_file_image_read( LPCTSTR filename );
```

The following function is the same as the previous but may be used when Sites-per-Controller > 1 to identify the target DBM by identifying its `HSBBoard`. See DBM & Multiple Sites-per-controller:

```
BOOL dbm_file_image_read( HSBBoard board, LPCTSTR filename );
```

where:

`filename` specifies the disk file to be read.

`board` may be used when Sites-per-Controller > 1 to identify the target DBM by identifying its `HSBBoard`, see DBM & Multiple Sites-per-controller.

`dbm_file_image_read()` returns TRUE if the file read operation had no errors otherwise FALSE is returned.

Example

```
fname[] = "D:\myPath\thatPath\thisFile";
BOOL ok = dbm_file_image_read( fname );
if( !ok ) output("ERROR: dbm_file_image_read() returned FALSE");
```

3.24.19 DBM Data File Format

See Data Buffer Memory Software (DBM), `dbm_file_image_write()`, `dbm_file_image_read()`.

Description

The dbm_file_image_write()and dbm_file_image_read() functions and (beginning is software release h2.2.7/h1.2.7), DBMTool support accessing a disk-resident file, called a DBM data file, with a specific format documented here.

The DBM file is a binary file that contains a DBM File Header followed by one or more DBM File Data Records. dbm_file_image_read() will only load the file to the DBM if the configuration defined in the header matches the current DBM configuration, as set using dbm_config_set().

DBM File Header

The header structure of the DBM data file is:

```
DWORD      version_number;
DWORD      numx;
DWORD      numy;
DWORD      width;
DbmPatternRate rate; // Required, but used on Magnum 1 only
BOOL       fast_axis;
DWORD      xstart;
DWORD      xstop;
DWORD      ystart;
DWORD      ystop;
__int64    num_records;
```

where:

version_number is a special value allowing dbm_file_image_read() to verify that the file is a DBM file. The initial value for this field = 0xDBF1.

numx and **numy** represent the numx and numy values of the DBM configuration, as set using dbm_config_set().

width represents the width value of the DBM configuration, as set using dbm_config_set(). The value of **width** also determines whether the DBM File Data Records which follow the header are 32-bit or 64-bit values. **width** values of 1, 2, 4, 8, 16, and 32 indicate 32-bit data records; **width** values of 9, 18, and 36 indicate 64-bit records.

rate represents the DBM's DBM DRAM Interleaving configuration and DBM Sequential Mode, as set using dbm_config_set().

fast_axis indicates how the DBM File Data Records are ordered i.e. which DBM address axis (X or Y) is incremented fast as the data is being read from the file and written to the

DBM. TRUE means the X-axis is fast, FALSE means the Y-axis is fast. When the DBM file is created by `dbm_file_image_write()`, the value of **`fast_axis`** will be the value returned by `x_fast_axis()` at the time is `dbm_config_set()` was last executed.

`xstart`, **`xstop`**, **`ystart`** and **`ystop`** describe a rectangular area of the DBM represented by the DBM File Data Records stored in this file. If the DBM file was created using `dbm_file_image_write()` these values will correspond to the `xstart`, `xstop`, `ystart` and `ystop` arguments to `dbm_file_image_write()`. When `dbm_file_image_read()` is used to load the file into the DBM only the addresses in this rectangle are modified.

`num_records` indicates the number of 32-bit or 64-bit DBM File Data Records following the header. Note that, except for **`width`**s of 32 and 36, **`num_records`** will not represent the number of DBM addresses represented by the stored data since each record may store values for more than one DBM address.

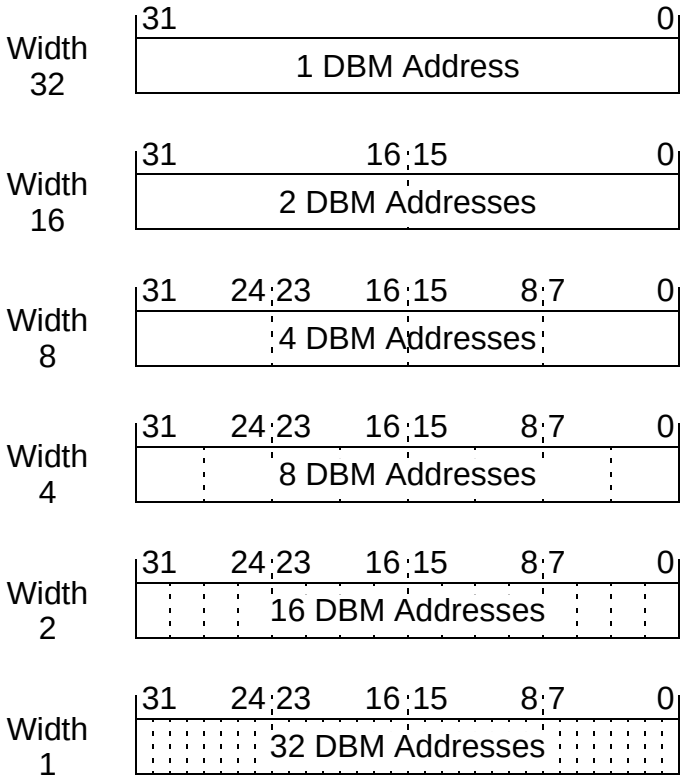
DBM File Data Records

As indicated above, the DBM file is a binary file that contains a DBM File Header followed by one or more DBM File Data Records.

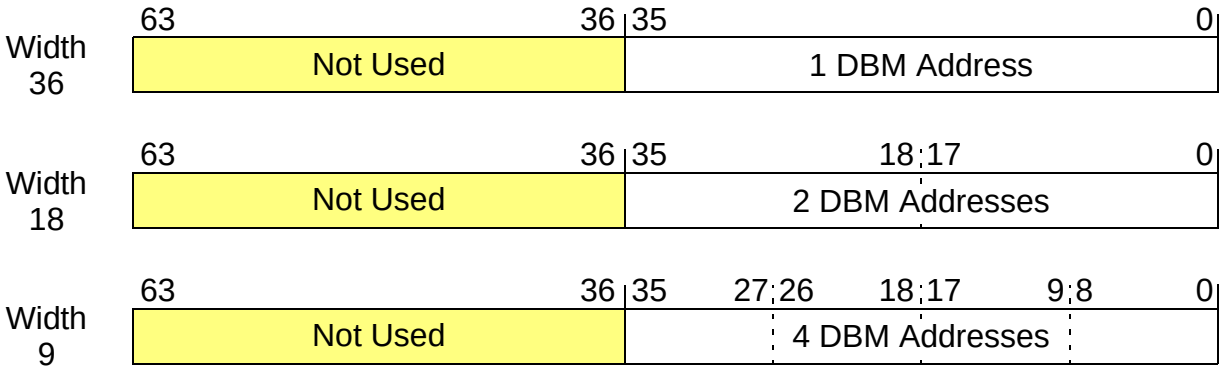
Each data record is either a 32-bit or 64-bit value, as determined by the DBM File Header **`width`** value. Each record consists of data for one or more DBM address(es).

When a data record stores data for more than one DBM address the earlier DBM address will be at the LSB of the record. The order the data records are stored in the file is from lowest DBM address to highest address, with the equivalent DBM address incrementing in the **`fast_axis`** direction.

Widths of 1, 2, 4, 8, 16 and 32 are packed into 32-bit records as shown below:



Widths of 9, 18, and 36 are packed into 64-bit records as shown below:



Note that a partial record may exist at the end of each DBM row or column, depending on the **`fast_axis`** parameter. For example, if **`fast_axis`** = TRUE (i.e. X addresses are incrementing fast) the last record for each column may contain data for less than the number of addresses noted above (each new column address will start a new data record).

3.24.20 DBM Address Masks

See Data Buffer Memory (DBM), Data Buffer Memory Software (DBM).

Description

The `dbm_masks_set()` function is used to configure the X and Y Data Buffer Memory (DBM) address masks. The `dbm_masks_get()` function is used to get the current X/Y DBM address mask values.

Note: do not confuse DBM address masks, which allow the user to enable DBM address compression, with APG address masks (set using `xmax()` or `ymax()`) which configure the number of APG X and Y addresses to generate.

By default, all used APG Address Generator X and Y address bits are applied to the DBM to determine which value is output by the DBM in each pattern instruction. The values set using `numx()` and `numy()` determine which APG X/Y addresses are *used*. The value specified using `x_fast_axis()` determines whether X-address or Y-address bits are the low-order addresses into the DBM.

It is possible to mask selected X and/or Y address bit(s), at the input to the DBM, such that they do not affect which DBM address is accessed in a given pattern instruction. In this context, masked DBM X/Y address bit(s) are set to logic-0 at the input to the DBM. Then, regardless of the actual logic state on these address bit(s), as generated by the APG, the DBM will output the same value.

For example:

- The APG X address generator is configured to 9 bits i.e. `numx()` = 9, X-addresses X0..X8 are used and the X-address range is 0x0 to 0x1FF.
- The APG Y address generator is configured to 5 bits i.e. `numy()` = 5, Y-addresses Y0..Y4 are used and the Y-address range is 0x0 to 0x1F.
- The Y-address generator is fast; i.e. `x_fast_axis()` is FALSE.
- Using `dbm_masks_set()` (more below) the MSB X-address (X8) is masked i.e. the `xmask` argument = 0xFF.
- As the pattern executes, the APG X-address bit X8 may change between logic-0 and logic-1. However, at the input to the DBM, the masked X8 address bit is always logic-0.

- Thus, when the current APG X-address = 0x1FF the DBM will output the value stored at X-address 0x0FF. And, the same value is output by the DBM when the APG X-address is 0x0FF.

Note the following:

- A 0 bit value in the mask represents an address bit to be ignored. For example, to mask X-address X0, the X-address mask would be 0x1FE.
- Only the bit(s) in `xmask` and `ymask` which overlap the currently configured DBM X-address size and Y-address size (see `dbm_config_set()`) have any effect. A warning is issued if `xmask` is longer than the currently configured DBM X-address size (`numx`), or if `ymask` is longer than the currently configured DBM Y-address size (`numy`).
- Executing `dbm_config_set()` always sets the DBM address masks to enable all address bits for the configuration being set.
- The system software does not modify the DBM address masks.
- DBM address masks are not modified when the value of `numx()` and/or `numy()` is modified. If, for example, a used upper Y-address bit is masked and later `numy()` is reduced, that upper Y-address bit remains masked, which will affect DBM segmentation operation. Therefore, **always** re-configure the DBM, using `dbm_config_set()`, when `numx()` and/or `numy()` are changed.
- DBM address masking only affects test pattern DBM access, to determine which APG X/Y addresses are used to address the DBM. Conversely, the API functions which read or write values to/from the DBM (`dbm_fill()`, `dbm_write()`, `dbm_read()` and `dbm_file_image_read()`) operate as though no DBM addresses are masked.

Usage

```
void dbm_masks_set( DWORD xmask, DWORD ymask );
void dbm_masks_get( DWORD* xmask, DWORD* ymask );
```

The following functions are the same as the previous set but may be used when Sites-per-Controller > 1 to identify the target DBM by identifying its `HSBBoard`. See DBM & Multiple Sites-per-controller:

```
void dbm_masks_set( HSBBoard board, DWORD xmask, DWORD ymask );
void dbm_masks_get( HSBBoard board, DWORD* xmask, DWORD* ymask );
```

where:

`xmask` and **`ymask`** are used in two contexts:

- Using `dbm_masks_set()`, the **`xmask`** argument sets the mask for the X-address inputs and the **`ymask`** argument sets the mask for the Y-address inputs.
- Using `dbm_masks_get()`, the **`xmask`** argument returns the current X-address mask and the **`ymask`** argument returns the current Y-address mask. **`xmask`** and **`ymask`** must be a pointer to an existing `DWORD` variables.

`board` may be used when Sites-per-Controller > 1 to identify the target DBM by identifying its `HSBBoard`, see DBM & Multiple Sites-per-controller.

Example

The following example assumes the current DBM configuration (see `dbm_config_set()`) sets `numx` = 10 and `numy` = 10. This example causes the LSB of the X-address to be ignored and the MSB of the Y-address to be ignored:

```
dbm_masks_set(0x3fe, 0x1ff);
```

The following example returns the currently configured DBM address masks:

```
DWORD xmask, ymask;
dbm_masks_get( &xmask, &ymask );
```

3.24.21 dbm_pattern_use()

See Data Buffer Memory (DBM), Data Buffer Memory Software (DBM).

Description

Note: beginning in software release h1.1.23, the need for the `dbm_pattern_use()` function was eliminated; the pattern compiler and system software were enhanced to automatically detect DBM use and configure the APG pipelines appropriately. However, if `dbm_pattern_use()` is used in the test program it takes precedence over the automated method. And, the automated method requires that any test pattern which uses the DBM be compiled using software release h1.1.23 or later.

~~The `dbm_pattern_use()` function is used to advise the system software that all subsequent test patterns will use the DBM as a data source. Note the following:~~

Note: proper operation of test patterns which use the DBM as a data source REQUIRES setting `dbm_pattern_use() = TRUE`.

- This is required because the APG pipeline configuration must be modified when using the DBM. This changes the number of pipeline cycles which must be executed when performing test pattern branch on error, branch on abort, etc. See Error Pipeline Requirements.
- Test patterns which do not use the DBM as a data source will operate correctly regardless of the state of `dbm_pattern_use()`, however the additional pipeline cycles enabled when `dbm_pattern_use() = TRUE` will continue to occur.
- During initial program load the `dbm_pattern_use()` state is set to FALSE. The system software does not otherwise modify this state.

Usage

```
void dbm_pattern_use( BOOL use );
```

where:

use specifies whether to configure the APG to support the DBM use in subsequent test pattern execution(s) (TRUE) or not (FALSE).

Example

The following example prepares the APG to execute test pattern(s) which use the DBM as a data source:

```
dbm_pattern_use( TRUE );
```

The following example restores the APG to execute test pattern(s) which do not use the DBM as a data source:

```
dbm_pattern_use( FALSE );
```

3.24.22 datbuf()

See Data Buffer Memory (DBM), Data Buffer Memory Software (DBM).

Description

The `datbuf()` function is used to access the DBM, to set or get the data value at the address currently selected by the APG X/Y address generator outputs.

The versions of `datbuf()` which take the `HSBBoard` argument are targeted at applications with Sites-per-Controller > 1. See DBM & Multiple Sites-per-controller.

Usage

The following functions read the DBM and return the value at the current X/Y address being output by the APG:

```
__int64 datbuf();
__int64 datbuf( HSBBoard Board );
```

The following functions write the DBM, changing the value at the current X/Y address being output by the APG:

```
void datbuf( __int64 Value );
void datbuf( HSBBoard Board, __int64 Value );
```

where:

`Board` identifies a specific Site Assembly Board (`HSBBoard`). See DBM & Multiple Sites-per-controller.

`Value` specifies the value to be written to the DBM.

The versions of `datbuf()` which return `__int64` return the value read from the DBM.

Example

```
datbuf( 0xA569 );
__int64 val = datbuf();
```

3.25 Error Catch RAM Software

See Error Catch RAM (ECR).

This section contains the following main topics:

- Overview
- ECR Functions
  - Types, Enums, etc.
  - ecr Data Type
  - PointFailure Structure
  - PointFailure Memory Management
  - Too many other functions to list here, see ECR Functions.
- ECR DDR Functions
  - `ecr_ddr_mode_set(), ecr_ddr_mode_get()`
- ECR Simulation
- Magnum 1 vs. Maverick ECR Functions

3.25.1 Overview

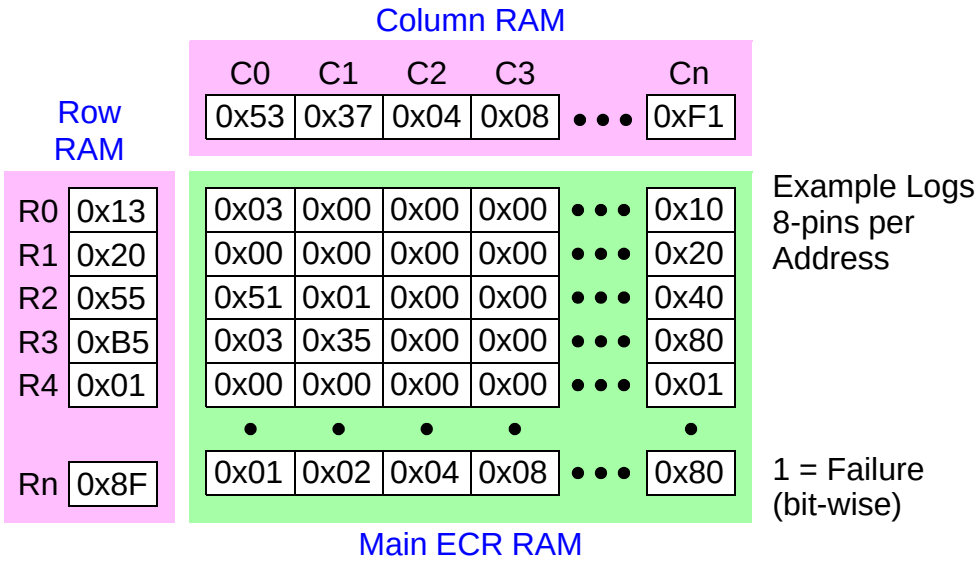
See Error Catch RAM (ECR), Error Catch RAM Software.

The ECR is a hardware option consisting of memory DIMM modules, counters/comparators and associated software. It is used, during test pattern execution, to capture per-pin failure information in real-time and supports the following features.

- Redundancy Analysis (RA) and repair for memory devices containing redundant circuitry. In the typical memory test application the ECR is configured (sized) to match the DUT size, thus any failing memory address and data bits in the DUT will be captured at the corresponding ECR locations. Then, Redundancy Analysis (RA) software uses the ECR contents as a map of good/bad DUT addresses and data bits. Applies to Memory Test Patterns applications only.
- BitmapTool display of ECR failure information, allowing the user to visualize failure patterns. Applies to Memory Test Patterns applications only.
- Capture logic vector failures, including vector address, for datalogging applications and display in LEC Tool. Applies to Logic Test Patterns applications only. This is documented separately in Logic Error Catch (LEC).

Note: except as noted, the remainder of this section applies to Memory Test Patterns applications only.

A detailed hardware description is covered in Error Catch RAM (ECR). A simplified model is shown below, after some errors have been captured:



Total Error Counters (not shown) count the total number of address or bit errors.
IOC Error Counters (not shown) count the number errors for each pin being logged.

Figure-53: ECR Simplified Model

The ECR configuration must be set up before use, using `ecr_config_set()`, typically in the Site Begin Block. Executing `ecr_config_set()` also executes `ecr_all_clear()`, which deletes any previously captured errors from the ECR.

The ECR captures errors when `funtest()` (or `start_pattern()`) is executed with the `fullec` argument. Note that there are other argument options to `funtest()` which impact the use of the ECR. See Pattern Execution Stop Condition Options for details.

BitmapTool accesses the ECR automatically; i.e. no additional user code is required.

User code accesses errors in the ECR by reading (scanning) the Main ECR RAM using `ecr_main_ram_scan()`. To improve scan efficiency/performance the ECR Mini-RAM and/or Row RAM and/or Column RAM can be scanned first (using `ecr_miniram_scan()`, `ecr_row_ram_scan()` and/or `ecr_column_ram_scan()`), to determine which areas of the Main ECR RAM contain errors. The Example uses the Row RAM in this fashion.

The Row RAM and Column RAM contain a bit-wise map of the errors in the corresponding row or column of the Main ECR RAM, with one bit for each pin being logged.

The diagram above has the ECR configured to capture errors from 8 pins). The Row RAM and Column RAM are used, for example, during Redundancy Analysis (RA) to quickly identify which rows and columns contain errors, and thus must be scanned and analyzed.

3.25.2 ECR Functions

See Error Catch RAM (ECR), Error Catch RAM Software.

The various ECR functions are listed below and subsequently organized in this section in two groups: routinely used and rarely used.

Common to all ECR functions are:

- Types, Enums, etc.
  - ecr Data Type
  - PointFailure Structure
  - PointFailure Memory Management

The following ECR functions are routinely used, listed alphabetically:

- `ecr_all_clear()`
- `ecr_any_overflow_get()`
- `ecr_column_ram_scan()`
- `ecr_compare_reg_set(), ecr_compare_reg_get()`
- `ecr_config_set()`
- `ecr_config_get()`
- `ecr_configured_get()`
- `ecr_interleave_get()`
- ECR Hardware Size Functions
  - `ecr_size_get()`
  - `ecr_ram_module_count_get()`
  - `ecr_ram_module_size_get()`
- `ecr_counters_config_set(), ecr_counters_config_get()`
- `ecr_dut_number_set(), ecr_dut_number_get()`
- `ecr_fast_image_write(), ecr_fast_image_read()`

- ecr_file_image_write(), ecr_file_image_read()
- ecr_main_ram_scan()
- ecr_miniram_config_set(), ecr_miniram_config_get()
- ecr_miniram_scan()
- ecr_overflow_get()
- ecr_row_ram_scan()

The following ECR functions are rarely used. Listed alphabetically:

- ecr_area_clear()
- ecr_col_ram_read()
- ecr_col_ram_write()
- ecr_counters_clear()
- ecr_ddr_mode_set(), ecr_ddr_mode_get()
- ecr_error_add()
- ecr_error_counter_set(), ecr_error_counter_get()
- ecr_error_delete()
- ecr_error_get()
- ecr_error_set()
- ecr_miniram_read()
- ecr_miniram_write()
- ecr_rams_clear()
- ecr_rams_update()
- ecr_row_ram_read()
- ecr_row_ram_write()
- ecr_scramble_bank_set(), ecr_scramble_bank_get()
- ecr_scramble_ram_write(), ecr_scramble_ram_read()
- ecr_x_y_data_set()

3.25.2.1 Types, Enums, etc.

See Error Catch RAM Software.

Description

The following enumerated types are used in support of various software functions:

The `EcrWriteMode` enumerated type is used to specify how the ECR captures errors. See `ecr_config_set()`, `ecr_config_get()`. Note that only `t_accum_1` is supported:

```
enum EcrWriteMode { t_accum_1 = 0,
                    t_abs_write = 2,
                    t_write_mode_na };
```

The `EcrFastDirection` enumerated type is used when configuring the ECR to optimize subsequent ECR scan performance. See `ecr_config_set()`, `ecr_config_get()`, `ecr_main_ram_scan()`:

```
enum EcrFastDirection { t_x_fast, t_y_fast, t_auto_fast };
```

The `EcrRamTypes` enumerated type is used to identify a specific ECR RAM type. See `ecr_rams_clear()`:

```
enum EcrRamTypes {t_main_array = 0x1,
                  t_row_catch = 0x2,
                  t_col_catch = 0x4,t_mini = 0x8,
                  t_rcm_ram = 0x10, // Not used for Magnum 1
   t_all_ecr_rams =
     (t_main_array|t_row_catch|t_col_catch|t_mini|t_rcm_ram),
                  t_ram_type_na };
```

The `EcrScrambleRamTypes` enumerated type is used to identify a specific ECR scramble RAM. See `ecr_scramble_ram_write()`, `ecr_scramble_ram_read()`:

```
enum EcrScrambleRamTypes {  t_x_scramble,
                            t_y_scramble,
                            t_scramble_ram_type_na };
```

The `EcrErrorCounters` enumerated type is used to identify specific ECR Error Counters or group of counters. See `ecr_counters_clear()`, `ecr_overflow_get()`, `ecr_compare_reg_set()`, `ecr_compare_reg_get()`, `ecr_error_counter_set()`, `ecr_error_counter_get()` Note that `t_rec` and `t_cec` are not supported on
Magnum 2/2x.:

```
enum EcrErrorCounters {  t_tec, t_rec,n t_cec,
                         t_ioc1, t_ioc2, t_ioc3, t_ioc4,
                         t_ioc5, t_ioc6, t_ioc7, t_ioc8,
```

```
                                t_ioc9, t_ioc10, t_ioc11, t_ioc12,
                                t_ioc13, t_ioc14, t_ioc15, t_ioc16,
                                t_ioc17, t_ioc18, t_ioc19, t_ioc20,
                                t_ioc21, t_ioc22, t_ioc23, t_ioc24,
                                t_ioc25, t_ioc26, t_ioc27, t_ioc28,
                                t_ioc29, t_ioc30, t_ioc31, t_ioc32,
                                t_ioc33, t_ioc34, t_ioc35, t_ioc36,
                                t_all_ioc, t_all_ecr_counters };
```

The `EcrCountingModes` enumerated type is used to configure the mode of the ECR Error Counters. See `ecr_counters_config_set(), ecr_counters_config_get()`:

```
enum EcrCountingModes {t_address_duplicates = 0,
                       t_bit_duplicates = 1,
                       t_address_no_dups = 2,
                       t_bit_no_dups = 3,
                       t_count_mode_na };
```

3.25.2.2 ecr Data Type

See Error Catch RAM Software.

In Maverick-I and Maverick-II test programs, many of the ECR and Redundancy Analysis (RA) related functions referenced the `ecr` data type, either by returning a pointer to an `ecr` or requiring an `ecr` argument.

Using Magnum 1, Magnum 2 and Magnum 2x, these legacy functions have all been replaced with functions which are more consistent with and appropriate for using the Magnum 1/2/2x ECR hardware architecture. These new functions neither need nor support the `ecr` data type. As noted in Magnum 1 vs. Maverick ECR Functions the legacy functions are supported, but the `ecr` parameter is ignored.

3.25.2.3 PointFailure Structure

See Error Catch RAM Software, Types, Enums, etc.

Description

The `PointFailure` structure supports the following functions which scan selected ECR hardware looking for failure information.

- `ecr_column_ram_scan()`
- `ecr_row_ram_scan()`
- `ecr_main_ram_scan()`
- `ecr_miniram_scan()`
- `ra_execute()` (and `ra_scan_area_callback()`)

The ECR functions can return a variable number of failures, with each failure representing a specific row address, column address, and failing data bit pattern. An array of `PointFailure` is used to return multiple failures.

However, since the number of failures logged to the ECR for a large memory can be substantial (huge), rather than declare a large array of `PointFailures`, the `ALLOCA_POINT_FAILURE()` and `ALLOC_POINT_FAILURE()` macros are available to help with memory management.

Note: proper memory management methods are critical to program reliability.

In most cases, software performing Redundancy Analysis (RA) does not need to process `PointFailure` information directly.

Usage

```
struct PointFailure {
   DWORD row, col;
   __int64 data;
};
```

Example

See Example.

3.25.2.4 PointFailure Memory Management

See Error Catch RAM Software, Types, Enums, etc.

Description

The ALLOCA_POINT_FAILURE() macro is used to allocate stack memory to store an array of PointFailure elements.

The ALLOC_POINT_FAILURE() macro is used to allocate heap memory to store an array of PointFailure elements. It also creates a PointFailure* variable which points to the start of the array.

The stack should be used when performance is critical; i.e. in production operations performing redundancy repair.

Note: ***fatal*** *stack overflow* errors can occur if the array size is made too large. The maximum size allowed is dynamic, depending on available RAM and the amount of stack memory already used by the test program. Use caution, test thoroughly.

Usage

```
PointFailure* ALLOCA_POINT_FAILURE( size );
ALLOC_POINT_FAILURE( name, size );
```

where:

ALLOCA_POINT_FAILURE is a Test System Macro used to allocate *stack* memory for storing PointFailure elements. **ALLOCA_POINT_FAILURE** returns a pointer to the allocated memory.

ALLOC_POINT_FAILURE is a Test System Macro used to allocate *heap* memory for storing PointFailure elements. The **name** argument becomes a pointer to the allocated memory. **name** is automatically destroyed and associated memory free'ed when execution exits the scope where the macro is defined.

size defines the number of PointFailure elements in the array being created.

Example

ALLOC_POINT_FAILURE is used in Example.

3.25.2.5 ecr_all_clear()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_all_clear()` function is used to clear the various RAMs in the Error Catch RAM (ECR). Note the following:

- The term *clear* means to delete any errors logged in the ECR.
- `ecr_all_clear()` clears all ECR RAMs (Main ECR RAM, ECR Mini-RAM, Row RAM and Column RAM).
- `ecr_all_clear()` clears all ECR counters (Total Error Counters, Row Error Counters, Col Error Counters, IOC Error Counters).

- By default, the `all_duts` argument is TRUE which causes `ecr_all_clear()` to clear all ECRs identically e.g. in Multi-DUT Test Programs, the Active DUTs Set (ADS) and Ignored DUTs Set (IDS) have no effect. Setting `all_duts` = FALSE causes `ecr_all_clear()` to only clear those ECRs for DUT(s) currently in the Active DUTs Set (ADS).
- Executing `ecr_config_set()()` executes `ecr_all_clear(TRUE)`.
- The Maverick-I/-II `clear()` function maps to `ecr_all_clear(TRUE)`.

Also see `ecr_area_clear()`, `ecr_rams_clear()`, `ecr_counters_clear()`.

Usage

```
void ecr_all_clear( BOOL all_duts DEFAULT_VALUE( TRUE ));
```

where:

`all_duts` is optional and, is used, modifies the behavior of `ecr_all_clear()`, see Description. Default = TRUE.

Note: executing `ecr_all_clear(TRUE)` is typically more efficient (faster) than clearing the ECR per-DUT. In many configurations, `ecr_all_clear(FALSE)` requires a series of read-modify-write operations, to maintain the ECR contents for DUT(s) which are not being cleared. This can consume a noticeable amount of time.

Example

```
ecr_all_clear();
```

3.25.2.6 ecr_any_overflow_get()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_any_overflow_get()` function is used to determine whether any ECR Error Counters overflowed (i.e. counted past the maximum, rolled-over, etc.). Note the following:

- In Multi-DUT Test Programs, only the ECR for the first DUT in the Active DUTs Set (ADS) is checked.

- When `ecr_any_overflow_get()` returns TRUE the specific counter which overflowed can be determined using `ecr_overflow_get()`.

Usage

```
BOOL ecr_any_overflow_get();
```

where:

`ecr_any_overflow_get()` returns TRUE if any ECR Error Counters from the ECR read have overflowed, otherwise FALSE is returned.

Example

```
if( ecr_any_overflow_get() == TRUE )
   output("At least one ECR counter overflowed");
else
   output("No ECR counters overflowed");
```

3.25.2.7 ecr_column_ram_scan()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_column_ram_scan()` function is used to scan a range of Column RAM addresses to determine the number of column errors which have been logged. Note the following:

- Two versions of `ecr_column_ram_scan()` are provided:
  - The 1st only returns an error count, which can be a count of error bits or error addresses.
  - The 2nd counts error addresses and also returns details about each error counted.
- In Multi-DUT Test Programs, only the ECR logging errors from the first DUT in the Active DUTs Set (ADS) is scanned.
- The `ecr_row_ram_scan()` function operates similarly on the Row RAMs and the `ecr_miniram_scan()` function operates similarly on the ECR Mini-RAMs.

Usage

The following function scans the specified range of Column RAM address(es) counting either failing addresses or failing bits:

```
int ecr_column_ram_scan( int cmin,
                         int cmax,
                         __int64 mask,
                         int max,
                         BOOL bit_cnt );
```

The following function scans the specified range of Column RAM address(es) counting failing addresses. Error details are returned via the `failures` argument:

```
int ecr_column_ram_scan( int cmin,
                         int cmax,
                         __int64 mask,
                         int max,
                         struct PointFailure *failures );
```

where:

`cmin` and **`cmax`** identify a range of Column RAM address to be read where 0 <= **`cmin`** <= **`cmax`** <= `ymax()`.

`mask` is a bit-mask which can be used to ignore errors on a per-pin basis. A logic-1 enables the error for that bit position to be counted/returned, a logic-0 inhibits an error from being counted/returned. The order of bits in **`mask`** must correspond to the order of pins in the `datapins` argument specified for `ecr_config_set()`.

`max` specifies a maximum number of errors to be read. If this value is exceeded `ecr_column_ram_scan()` returns, potentially without scanning the entire Column RAM.

`bit_cnt` specifies whether error bits are counted (TRUE) or error addresses are counted (FALSE). Using the latter, any number of error bits at a give Column RAM address will count as one error.

`failures` is a pointer to an existing `PointFailure`s array, used to return error details: row address (always 0 when scanning the Column RAM), column address, and data. User code is responsible for allocating memory for this array. See `ALLOCA_POINT_FAILURE()` for one method. Since the number of failures returned can vary, the number of `PointFailure` elements allocated must anticipate the worse case number of failures which *might* be returned. When using **`failures`**, proper memory management methods are critical to program reliability.

`ecr_column_ram_scan()` returns the number of errors counted. This also equates to the number of valid elements in the **`failures`** array.

Example

The following example scans the Column RAM from address 10 to address 20 inclusive and returns a count of failing bits (up to 100). Only the low 8 bits are counted (`0xFF` mask):

```
int c = ecr_column_ram_scan( 10, 20, 0xFF, 100, TRUE );
```

Also see Example for similar application of `ecr_row_ram_scan()`.

3.25.2.8 ecr_compare_reg_set(), ecr_compare_reg_get()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_compare_reg_set()` function is used to write a value to specified ECR Counter Comparators.

The `ecr_compare_reg_get()` function can be used to get the currently programmed value from a specified ECR Counter Comparators.

Note the following:

- The ECR's Row Error Counters, Col Error Counters, and Total Error Counterseach have a corresponding comparator, which can signal the Branch-on-error Logic (Branch Error Choice Logic) when the value in the corresponding counter equals or exceeds the count specified using `ecr_compare_reg_set()`. This can be used to control test pattern branch operations in Memory Test Patterns using the MAR Error-choice Operands.
- Each counter type has a single compare value, set using `ecr_compare_reg_set()`, stored in the compare register for that counter type. In Multi-DUT Test Program, all counters of a given type share the same compare register.

- The `type` argument identifies the counter-type, which determines which compare register will be accessed (Magnum 2/2x have only one option):

Table 3.25.2.8-1 ECR Counter Comparator Selection

Type Value	Counter Comparator	Count Range
`t_tec`	TEC Comparator Register	1 to 17,179,869,183 (2^{34}-1)
`t_rec`	REC Comparator Register	1 to 262,143 (2^{18}-1)
`t_cec`	CEC Comparator Register	1 to 262,143 (2^{18}-1)
Note: if the value is set to maximum the only comparator match will occur when the counter exactly equals the value because the next counter increment will overflow = 0.		

Note: using Total Error Counters and/or IOC Error Counters with either the t_bit_no_dups or t_address_no_dups options to ecr_counters_config_set(), the hardware performs a read/modify/write operation, writing only new errors to the ECR with no duplicates counted in the TEC or IOC. This cannot be done when ECR interleaving is used, see Magnum ECR Memory Size Options. The controls for this are the fastest_cycle and seq_length arguments to ecr_config_set(). Proper TEC and IOC counter operation will NOT occur if this rule is violated.

Usage

```
void ecr_compare_reg_set( EcrErrorCounters type, __int64 value );
__int64 ecr_compare_reg_get( EcrErrorCounters type );
```

where:

`type` identifies the counter-type, which determines which compare register will be accessed. Legal values are of the `EcrErrorCounters` enumerated type, see table above.

`value` specifies the comparator value. Legal values are listed in the table above.

`ecr_compare_reg_get()` returns the currently programmed value. In Multi-DUT Test Programs, the value is returned for the first DUT in the Active DUTs Set (ADS).

Example

```
ecr_compare_reg_set( t_tec, 0x1234 );
__int64 value = ecr_compare_reg_get( t_tec );
```

3.25.2.9 ecr_config_set()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_config_set()` function is used to configure the Error Catch RAM (ECR) prior to use. The `ecr_config_get()` function can be used to retrieve (get) the current ECR configuration.

Note the following:

- Executing `ecr_config_set()` also executes `ecr_all_clear()`, clearing errors from all ECRs.
- The `numx` and `numy` arguments configure the ECR's X and Y address size. In hardware, this configures the Address CrossPoint. The values specified are before any address compression is applied (more below). Any X/Y address aspect ratio is allowed, up to the maximum supported by the APG Address Generator. If the ECR is too small to support the specified configuration a fatal error is issued.
- The `datapins` argument determines which pin(s) are logged to the ECR. In hardware, this configures the Data CrossPoint. Up to 36 pins of each 64-pin Sub-site (see PE Sub-site Architecture) can be captured.
- For proper Redundancy Analysis (RA) operation, the ECR data configuration must match the DUT's data configuration.

Note: in Multi-DUT Test Programs, the `datapins` pin list will identify pins for a single DUT. However, the user must account for the number of hardware tester channels this represents and limit the pin list members such that no more than 36 hardware pins are specified for each 64-pin Sub-site. This requires that the user carefully consider which tester pins are connected, via the DUT board, to the DUT pins which are to be captured in the ECR, see DUT-pin to Tester-pin Connection Requirements.

- The Redundancy Analysis (RA) software does not provide any address compression facilities; i.e. redundancy software depends upon the ECR logging failures in a compressed manner, when appropriate. The `x_compress_mask` and `y_compress_mask` arguments provide for address compression by configuring the Address CrossPoint. Address compression is used when a single redundancy spare element replaces more than one row or one column. For example, if the least significant bit in the `x_compress_mask` is set to zero then the LSB X-address bit will be ignored by the ECR and all failures from X-addresses 0 and 1 will be logged in ECR X-address 0. Similarly, all failures from device X-addresses 2 and 3 will be logged to ECR X-address 1, failures from device X-addresses 4 and 5 will be logged to ECR X-address 2, etc. The net result is a 50% compression (reduction) of the amount of ECR used to log failures. Thus, if testing 64M device, this compression would log all the failures in 32M of ECR memory.
- The Redundancy Analysis (RA) software does not provide any data compression facilities; i.e. redundancy software depends upon the ECR logging failures in a compressed manner, when appropriate. The `data_compress` argument indirectly configures the Data CrossPoint to provide for ECR data compression. Legal values for `data_compress` are 1 (no compression), 2, 4, 8, 16, 32. When a data compression value >1 is specified, failures on 2 (or 4, 8, 16, etc.) adjacent pins are logged as a single failure in the ECR. For example. setting `data_compress` to 2 causes failures from the first 2 `datapins` to be logically OR'ed together into a single ECR bit. Similarly, the 2nd two `datapins` are OR'ed together into a single bit, etc.

Note: on 10/31/06 the documentation for the `wr_mode` argument was modified to remove reference to `t_abs_write` mode. This mode was never implemented due to hardware limitations. The following paragraphs referring to the ECR capture mode now reflect what is supported. References to `ecr_write_mode_set()` were deleted since it is no longer useful.

- The `wr_mode` argument sets the ECR capture mode. Only the `t_accum_1` mode is supported, and operates as follows:
  - `t_accum_1` = accumulate errors. Multiple errors can be logged to a given ECR address; i.e. errors are accumulated. This allows multiple reads of a given DUT address to be logged without later reads overwriting errors logged earlier. This mode is consistent with Maverick-I and Maverick-II ECR operation.
  - `t_accum_1` sets the write mode for the Main ECR RAM, Row RAM, Column RAM.

  - The `wr_mode` argument has no effect on the ECR Error Counters, see `ecr_counters_config_set()`.
- The `fast` argument is used to configure the ECR to optimize read (scan) performance, based on which address axis (X or Y) will be scanned *fast* (see `ecr_main_ram_scan()`). This has no effect on ECR capture operation but will improve read performance. The following options are available:
  - `t_auto_fast` - the system software chooses the fast direction, based on which axis (X or Y) uses the most address bits. If equal, `t_x_fast` is selected.
  - `t_x_fast` - configure the ECR to optimize scanning the X-address fast.
  - `t_y_fast` - configure the ECR to optimize scanning the Y-address fast.
- The Main ECR RAM is implemented using burst DRAM, using an interleaving technique to provide random access at full speed. To capture errors at the Magnum's maximum data rate (i.e. 50MHz/20nS strobe rate) requires an interleave ratio of 8; i.e. the entire ECR main memory size is divided by 8. Conversely, as the capture data rate is reduced the interleave ratio is reduced, increasing the effective size of the ECR. The `fastest_cycle` argument is used

to specify the maximum strobe rate on pins which are to capture errors in the ECR. The system software uses this value to set up the ECR interleaving, in one of 4 configurations:

Table 3.25.2.9-1 Magnum 1 ECR Interleave Configurations

Strobe/Error Capture Rate			
>=20nS and <40nS ~25MHz to 50MHz	>=40nS and <80nS ~12.5MHz to 25MHz	>=80nS and <160nS ~6.25MHz to 12.5MHz	>=160nS <=6.25MHz
8-way Interleave Minimum ECR Size	4-way Interleave	2-way Interleave	No Interleave Maximum ECR Size
The Total Error Counters and/or IOC Error Counters should not be used when interleaving is enabled. See Note: below and `ecr_counters_config_set()`.			Total Error Counters and/or IOC Error Counters are usable.
Note that the interleave ratio also affects the effective ECR size, see Magnum ECR Memory Size Options. The `ecr_interleave_get()` function may be used to determine the interleave ratio after the ECR has been configured using `ecr_config_set()`.			

Note: proper ECR operation will **NOT** be correct if any 2 (or more) strobes on any pin(s) being captured occur at a rate faster than specified by the `fastest_cycle` value.

Note: using Total Error Counters and/or IOC Error Counters with either the `t_bit_no_dups` or `t_address_no_dups` options to `ecr_counters_config_set()`, the hardware performs a read/modify/write operation, writing only new errors to the ECR with no duplicates counted in the TEC or IOC. This cannot be done when ECR interleaving is used, see Magnum ECR Memory Size Options. The controls for this are the `fastest_cycle` and `seq_length` arguments to `ecr_config_set()`. Proper TEC and IOC counter operation will NOT occur if this rule is violated.

- Executing `ecr_config_set()` configures all ECRs identically. In Multi-DUT Test Programs, the Active DUTs Set (ADS) and Ignored DUTs Set (IDS) have no effect on `ecr_config_set()`.
- Often, the ECR configuration only needs be set up one time, and is commonly done in the Site Begin Block. However, the ECR configuration can be changed at any time as needed. Remember, `ecr_config_set()` always clears the ECR, which can impact performance (test time).
- Executing `ecr_config_set()` does not modify the X/Y Scramble RAM, see `ecr_scramble_ram_write()`.

Note: to capture errors when executing a Double Data Rate (DDR) Mode test pattern requires additional configuration, using `fail_signal_mux()` and `ecr_ddr_mode_set()`, which must be executed before `ecr_config_set()`.

- A special and rarely used application may require that `ecr_dut_number_set()` be executed before `ecr_config_set()` executed. See `ecr_dut_number_set()` and Shared Tester Pins.

Usage

```
void ecr_config_set(
                int numx ,
                int numy,
                PinList* datapins,
                int x_compress_mask DEFAULT_VALUE( 0xffffffff ),
                int y_compress_mask DEFAULT_VALUE( 0xffffffff ),
                int data_compress DEFAULT_VALUE( 1 ),
                EcrWriteMode wr_mode DEFAULT_VALUE( t_accum_1 ),
                EcrFastDirection fast DEFAULT_VALUE( t_auto_fast ),
                double fastest_cycle DEFAULT_VALUE( -1.0 ),
                int seq_length DEFAULT_VALUE( 1 ) );
```

where:

numx configures the ECR size in the X dimension, and corresponds to the X-address size of the DUT. The maximum value is 18. The **numx** value is specified before any X address compression.

numy configures the ECR size in the Y dimension, and corresponds to the Y-address size of the DUT. The maximum value is 16. The **numy** value is specified before any Y address compression.

datapins is a pin list identifying which pins are being captured to the ECR.

x_compress_mask is optional and, if used, specifies X address compression. Legal values are 0x0 to `0x3ffff` (default). In both cases, the default value = no compression. See Description.

y_compress_mask is optional and, if used, specifies Y address compression. Legal values are 0x0 to `0xffff` (default). In both cases, the default value = no compression. See Description.

data_compress is optional, and provides for data compression. Legal values for **data_compress** are 1 (default = no compression), 2, 4, 8, 16, 32, however the value specified must evenly divide the number of pins in **datapins**. See Description.

wr_mode is optional, and if used specifies the desired ECR capture mode. Only the `t_accum_1` mode is supported. Legal values are of the `EcrWriteMode` enumerated type. Default = `t_accum_1`. See Description.

fast is optional, and if used specifies how the ECR should be configured to optimize read (scan) performance (see `ecr_main_ram_scan()`). Legal values are of the `EcrFastDirection` enumerated type. Default = `t_auto_fast`. See Description.

fastest_cycle is optional, and if used specifies the maximum strobe rate for all pins being logged. This determines how the ECR interleaving is configured, which affects the maximum usable ECR size. Default = -1 = maximum strobe rate = 50Mhz/, which results in the minimum ECR size configuration. See Description. When not using -1, units should be used (see Specifying Units).

Note: proper ECR operation will NOT be correct if any 2 (or more) strobes on any pin(s) being captured occur at a rate faster than specified by the fastest_cycle value. Also see Note: if using Total Error Counters and/or IOC Error Counters.

---

seq_length is not used on Magnum 1. Any value specified is silently ignored.

simul_cap_scan is not used on Magnum 1. Any value specified is silently ignored.

Example

```
ecr_config_set(numx(), numy(), pl_datapins, 0xFFFFF, 0xFFFF, 1,
               t_accum_1, t_auto_fast, -1 );
```

3.25.2.10 ecr_config_get()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_config_get()` function can be used to retrieve (get) the current Error Catch RAM (ECR) configuration, as set using `ecr_config_set()`.

Note: beginning in software release h3.3.xx, the values returned by `ecr_config_get()` are valid only if the ECR is currently configured as an ECR (i.e. not as a Logic Error Catch (LEC)). See `ecr_configured_get()` and `lec_configured_get()`.

Detailed descriptions of each return value is described in `ecr_config_set()`.

In Multi-DUT Test Programs, the Active DUTs Set (ADS) and Ignored DUTs Set (IDS) have no effect on `ecr_config_get()`.

The `ecr_config_set()` function always configures all sites identically. However, when Sites-per-Controller > 1 not all pins of the specified `datapins` pin list may reside on the same site. The version of `ecr_config_get()` with the `HSBBoard` argument may be used to identify which pins are associated with a given site (`HSBBoard`). When Sites-per-Controller > 1 if the version of `ecr_config_get()` without the `HSBBoard` argument is used, it returns all pins previously specified using the `datapins` argument to `ecr_config_set()`.

`ecr_configured_get()` may be used with `lec_mode_get()` to determine whether the Error Catch RAM (ECR) is configured for ECR use vs. Logic Error Catch (LEC) use.

Usage

The following function returns multiple values using variable parameter arguments. Each argument defaults to a NULL pointer, which must be replaced with a pointer to an existing variable of the appropriate type to obtain the desired value. Any don’t-care values can remain NULL (0):

```
void ecr_config_get(int* numx DEFAULT_VALUE( 0 ),
                    int* numy DEFAULT_VALUE( 0 ),
                    int* datawidth DEFAULT_VALUE( 0 ),
                    PinList** datapins DEFAULT_VALUE( 0 ),
                    int* x_compress_mask DEFAULT_VALUE( 0 ),
                    int* y_compress_mask DEFAULT_VALUE( 0 ),
                    int* data_compress DEFAULT_VALUE( 0 ),
                    EcrWriteMode* wr_mode DEFAULT_VALUE( 0 ),
                    EcrFastDirection* fast DEFAULT_VALUE( 0 ),
                    double* fastest_cycle DEFAULT_VALUE( 0 ),
                    int *seq_length DEFAULT_VALUE( 0 ),
                    BOOL *simul_cap_scan DEFAULT_VALUE( 0 ));
```

The following function is used when Sites-per-Controller > 1, see Description. Only the pins returned via the `datapins` argument will change as the `board` argument is changed.

```
void ecr_config_get(HSBBoard board,
                    int* numx DEFAULT_VALUE( 0 ),
                    int* numy DEFAULT_VALUE( 0 ),
                    int* datawidth DEFAULT_VALUE( 0 ),
                    PinList** datapins DEFAULT_VALUE( 0 ),
                    int* x_compress_mask DEFAULT_VALUE( 0 ),
                    int* y_compress_mask DEFAULT_VALUE( 0 ),
                    int* data_compress DEFAULT_VALUE( 0 ),
                    EcrWriteMode* wr_mode DEFAULT_VALUE( 0 ),
                    EcrFastDirection* fast DEFAULT_VALUE( 0 ),
                    double* fastest_cycle DEFAULT_VALUE( 0 ),
                    int *seq_length DEFAULT_VALUE( 0 ),
                    BOOL *simul_cap_scan DEFAULT_VALUE( 0 ));
```

where:

numx and **numy** are pointers to two existing `int` variables, used to return the number of X and Y addresses enabled in the current ECR configuration last set using `ecr_config_set()`.

datawidth is a pointer to an existing `int` variable, used to return the number of pins per DUT in the **datapins** pin list.

datapins is a pointer to an existing `PinList*` variable, used to return a pin list containing the pins being captured.

x_compress_mask and **y_compress_mask** are pointers to two existing `int` variables, used to return the X and Y address compression mask values last set using `ecr_config_set()`.

data_compress is a pointer to an existing `int` variable, used to return the data compression value, last set using `ecr_config_set()`.

wr_mode is a pointer to an existing `EcrWriteMode` variable, used to return the ECR capture (write) mode, last set using `ecr_config_set()`.

fast is a pointer to an existing `EcrFastDirection` variable, used to return the current X/Y fast configuration, last set using `ecr_config_set()`.

fastest_cycle is a pointer to an existing `double` variable, used to return the corresponding value, last set using `ecr_config_set()`.

seq_length is a pointer to an existing `int` variable, used to return the corresponding value, last set using `ecr_config_set()`.

simul_cap_scan is a pointer to an existing `BOOL` variable, used to return the corresponding value, last set using `ecr_config_set()`.

board may be used when Sites-per-Controller > 1 to specify which board will be read.

Example

The following example retrieves all of the currently programmed ECR configuration parameters except `x_compress_mask` and `y_compress_mask`:

```
int numx, numy, datawidth, dmask;
PinList*pins;
EcrWriteMode wr_mode;
EcrFastDirection fast;
double strobe_rate;
ecr_config_get(&numx, &numy, &datawidth, &pins, 0, 0,
               &dmask, &wr_mode, &fast, &strobe_rate );
```

3.25.2.11 ecr_configured_get()

Note: first available in software release h3.3.xx.

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_configured_get()` function may be used to determine if the Error Catch RAM (ECR) is currently configured.

In this context, `ecr_configured_get()` returns TRUE if the ECR is currently configured, but returns FALSE if the ECR is not configured or if the ECR is currently configured for LEC use (using `lec_config_set()`).

Similarly, the `lec_configured_get()` function returns TRUE if the ECR is currently configured for LEC use, but returns FALSE if the ECR not configured, or if ECR is currently configured for ECR use.

In Multi-DUT Test Programs, the Active DUTs Set (ADS) and Ignored DUTs Set (IDS) have no effect on `ecr_configured_get()`.

Usage

```
BOOL ecr_configured_get();
BOOL ecr_configured_get( HSBBoard board );
```

where:

`board` is used when Sites-per-Controller > 1 to identify a target site assembly (`HSBBoard`).

`ecr_configured_get()` returns TRUE if the ECR is currently configured, but returns FALSE if the ECR is not configured or if the ECR is currently configured for LEC use (using `lec_config_set()`).

Example

```
BOOL configured = ecr_configured_get();
BOOL ecr_hsb1_configured = ecr_configured_get( t_hsb1 );
```

3.25.2.12 ecr_interleave_get()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_interleave_get()` function may be used to determine the current ECR interleave ratio. See Overview. This function is not useful until after the ECR has been configured using `ecr_config_set()`.

Usage

```
int ecr_interleave_get();
```

`ecr_interleave_get()` returns the current ECR interleave ratio.

Example

```
int Iratio = ecr_interleave_get();
```

3.25.2.13 ECR Hardware Size Functions

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_size_get()` function may be used to determine the overall size of the installed ECR on a specified site assembly (`HSBBoard`).

The `ecr_ram_module_count_get()` function may be used to determine the number of ECR modules installed on a specified site assembly (`HSBBoard`).

The `ecr_ram_module_size_get()` function may be used to determine the size of a specified ECR module installed on a specified site assembly (`HSBBoard`).

Usage

```
int ecr_size_get( HSBBoard board );
int ecr_ram_module_count_get( HSBBoard board );
int ecr_ram_module_size_get( HSBBoard board, int module );
```

where:

`board` identifies the target site assembly board (`HSBBoard`).

`module` identifies the target ECR module on **`board`**.

`ecr_size_get()` returns the overall ECR size for **`board`**, in Giga-bits. 0 is returned if an invalid **`board`** is specified.

`ecr_ram_module_count_get()` returns the number of ECR modules installed on **`board`**. 0 is returned if an invalid **`board`** is specified.

`ecr_ram_module_size_get()` returns the module size of **`module`** on **`board`**, in Gigabits. 0 is returned if an invalid **`board`** or **`module`** is specified.

Example

```
int Esize = ecr_size_get( t_hsb1 );
if( Esize == -1 ) output("ERROR");

int count = ecr_ram_module_count_get( t_hsb1 );
if( count == -1 ) output("ERROR");

int Msize = ecr_ram_module_size_get( t_hsb1, 1 );
if( Msize == -1 ) output("ERROR");
```

3.25.2.14 ecr_counters_config_set(), ecr_counters_config_get()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_counters_config_set()` function is used to configure the count mode for the ECR Error Counters.

The `ecr_counters_config_get()` function may be used to retrieve the currently programmed count mode for the ECR Error Counters.

Note the following:

- All ECR Error Counters, except Row Error Counters, can be configured to count errors in one of two modes:
  - Count individual bit errors
  - Count errors per-address
- Row Error Counters can only count per-address errors.
- ECR Error Counters can also be configured to ignore or count duplicate errors; i.e. the 2nd through nth error at the same address or same bit (pin).

- Thus, in combination, 4 options are available, one of which is specified for each type of ECR Error Counters:

Option	Operation
t_address_duplicates	Default. The counter is incremented once for each failing address, regardless of how many pin(s) are failing. Duplicate failing addresses are counted.
t_address_no_dups	The counter is incremented once for each failing address but duplicate failing addresses are not counted. See note below.
t_bit_duplicates	The counter is incremented once for each failing bit (pin). Duplicate failures are counted. This mode cannot be used for Row Error Counters.
t_bit_no_dups	The counter is incremented once for each failing bit (pin) but duplicate failures are not counted. This mode cannot be used for Row Error Counters. See note below.

- All counters of the same type are configured identically.
- In Multi-DUT Test Programs, the Active DUTs Set (ADS) and Ignored DUTs Set (IDS) have no effect on `ecr_counters_config_set()` or `ecr_counters_config_get()`.

Note: using Total Error Counters and IOC Error Counters with either `_no_dups` option, the hardware performs a read/modify/write operation, writing only new errors to the ECR with no duplicates counted in the TEC or IOC. This cannot be done when ECR interleaving is used, see Magnum ECR Memory Size Options. The controls for this are the `fastest_cycle` and `seq_length` arguments to `ecr_config_set()`. Proper TEC and IOC counter operation will NOT occur if this rule is violated.

Usage

```
void ecr_counters_config_set( EcrCountingModes tec_mode,
                              EcrCountingModes rec_mode,
                              EcrCountingModes cec_mode,
                              EcrCountingModes ioc_mode );
```

```
void ecr_counters_config_get( EcrCountingModes* tec_mode,
                              EcrCountingModes* rec_mode,
                              EcrCountingModes* cec_mode,
                              EcrCountingModes* ioc_mode );
```

where:

tec_mode, **rec_mode**, **cec_mode** and **ioc_mode** are used in two contexts:

- In the set function, these specify the desired counter mode for each counter type.
- In the get function, these are the addresses of existing EcrCountingModes variables used to return the currently programmed mode for each counter type.

Example

```
ecr_counters_config_set( t_bit_duplicates, t_bit_duplicates,
                         t_address_duplicates, t_address_duplicates );
EcrCountingModes tec_mode, rec_mode, cec_mode, ioc_mode;
ecr_counters_config_get( &tec_mode, &rec_mode,
                         &cec_mode, &ioc_mode );
```

3.25.2.15 ecr_dut_number_set(), ecr_dut_number_get()

See Error Catch RAM (ECR), Error Catch RAM Software, Shared Tester Pins.

Description

The ecr_dut_number_set() function is used to specify how many DUTs in the pin list passed to ecr_config_set() should actually be logged to the ECR. This is used only in situations where multiple DUTs share tester pins, more below.

The ecr_dut_number_get() function may be used to get the current value.

This (uncommon) application shares pins between two DUTs, allowing more DUTs to be tested in parallel than could be done without sharing pins (however, this application is also more complex, requiring the user to manage details normally controlled automatically by the system software). For example, assume the Pin Assignment Table for a given test program is configured to test 16 DUTs, each with 8 DutPins (signal pins, not DPS, HV, etc.). Then, for each DutPin, the tester pin mapped to DUT-1 is also mapped to DUT-9, etc. (see Shared Tester Pins). This example allows twice the number of DUTs to be tested than could be if pins were not shared between DUTs.

In any Multi-DUT Test Program, each `DutPin` element in each pin list will, by design, represent that pin for all DUTs defined in the program. Thus, in this example, each `DutPin` element of each pin list will actually represent 16 pins (`HDTesterPin`), one for each DUT in the test program (see Pin Lists). Normally this is transparent to both the user and to the desired program operation. However, when DUTs share pins some of the `HDTesterPin`s represented by a given `DutPin` will be duplicates. In this example, half of the `HDTesterPin`s representing a given `DutPin` will be duplicates.

By default, when one of these pin lists is used with `ecr_config_set()`, the ECR Data CrossPoint will be configured to capture the same pin twice. This is not useful and may prevent capturing some desired pins.

Thus, when pins are shared, the ECR configuration must specify that it will capture one set of DUTs at a time. To do this, the `ecr_dut_number_set()` is used to advise the ECR software how many DUTs are to be captured. Given the example above, `ecr_dut_number_set(8)` is used, which ensures that the duplicate pins for DUT-9 through DUT-16 are not logged to the ECR.

Note the following:

- `ecr_dut_number_set()` *must* be executed *before* executing `ecr_config_set()`.
- The `num` argument specifies the number of DUTs to be logged to the ECR. By design, this will always be the first `num` DUTs (DUT-1 through DUT-`num`).
- Proper operation depends upon the rules documented in Shared Tester Pins being followed. Specifically, the first DUT of the second half must exactly mirror DUT-1, etc. In the example above, DUT-1 and DUT-9 must use the same pins, DUT-2 and DUT-10 must use the same pins, etc.
- The `num` argument to `ecr_dut_number_set()` may be less than 1/2 of the number of DUTs defined in the test program, but the pins actually logged to the ECR will always be begin with DUT-1, followed by DUT-2, etc. See Pin Assignment Table.
- The system software does not modify or reset any settings made using `ecr_dut_number_set()`; i.e. they remain in effect until user code executes `ecr_dut_number_set()` AND `ecr_config_set()` again.
- Normal operation can be restored by executing `ecr_dut_number_set(-1)` followed by `ecr_config_set()`.
- The Active DUTs Set (ADS) and Ignored DUTs Set (IDS) have no effect on these functions.

Usage

```
void ecr_dut_number_set( int num );
int ecr_dut_number_get();
```

where:

num specifies how many DUTs should be logged to the ECR.

`ecr_dut_number_get()` returns the value last set using `ecr_dut_number_set()`. -1 is returned if `ecr_dut_number_set()` has not been used to change the default operation.

Usage

```
ecr_dut_number_set( 8 );
int num = ecr_dut_number_get();
```

3.25.2.16 ecr_fast_image_write(), ecr_fast_image_read()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_fast_image_write()` function is used to save an image of one ECR's contents to a disk file.

The `ecr_fast_image_read()` function is used to read an ECR image file into one ECR

Note the following:

- These functions are used instead of `ecr_file_image_write()`, `ecr_file_image_read()` when write/read performance is important (only the Main ECR RAM and ECR configuration is written or read).
- The `filename` argument specifies the input or output file name.
- If `filename` doesn't specify an absolute path the file will be located relative to the test program executable file, typically in the test program *Debug*\ folder.
- Using `ecr_fast_image_write()`, no file clobber check is performed; i.e. any existing file of the same name WILL be over-written.

- Using `ecr_fast_image_write()`, the current user must have write permission on the folder and file being written.
- In Multi-DUT Test Programs, only the ECR of the first DUT in the Active DUTs Set (ADS) is saved to disk or loaded from disk.
- As indicated, performance is improved because less information is written or read as compared to `ecr_file_image_write(), ecr_file_image_read()`. The following information is NOT written or read:
  - The Column RAM, Row RAM, ECR Mini-RAM.
  - ECR Error Counters (Total Error Counters, IOC Error Counters, Row Error Counters, Col Error Counters).
  - ECR Counter Comparators.
  - None of the Address CrossPoint, Data CrossPoint, X/Y Scramble RAM, etc.
- Using `ecr_fast_image_read()`, the current ECR configuration MUST exactly match the configuration in effect when the file being read was written. If this rule is violated a warning is issued, the specified file is not loaded into the ECR, and `ecr_fast_image_read()` returns FALSE.
- Using `ecr_fast_image_read()`, since most of the ECR configuration was not included when the file being read was saved, the contents of the various ECR memories/counters must NOT be used unless first updated using `ecr_rams_update()`. This is quite important since most ECR scan routines use one or more of the ECR RAMs to optimize the scan performance. Depending on the size of the ECR, `ecr_rams_update()` can consume a noticeable amount of time.
- Using `ecr_fast_image_write()`, the exact number of ECR addresses containing errors must be determined before any ECR data is written.
  - In Multi-DUT Test Programs, this equates to the errors for the first DUT in the Active DUTs Set (ADS).
  - User code can specify this using the optional `count` argument. The hardware Total Error Counters can be used to obtain the `count` value, but note the following two items:
  - The TEC count mode must be set to `t_address_duplicates` or `t_address_no_dups` prior to executing the pattern.
  - If the test pattern reads a given address more than once while logging errors to the ECR the TEC counter mode must be set to `t_address_no_dups` prior to executing the pattern (see Note:).
  - If `count` is not specified, `ecr_fast_image_write()` will determine the number of errors by scanning the Main ECR RAM, which takes additional time.

Usage

```
BOOL ecr_fast_image_write(  LPCTSTR filename,
                            __int64 count DEFAULT_VALUE( -1 ) );
BOOL ecr_fast_image_read( LPCTSTR filename );
```

where:

`filename` specifies the desired input or output file name. See Description for rules.

`count` is optional, and if used must specified the exact number of ECR addresses containing errors. See Description.

Both functions return FALSE if any error(s) occur, otherwise TRUE is returned.

Example

```
if( ecr_fast_image_write( "d:/myEcrOutputFileName") == FALSE)
   output(" ERROR: ecr_fast_image_write() returned an error");
if( ecr_fast_image_read( "d:/myEcrInputFileName") == FALSE)
   output(" ERROR: ecr_fast_image_read() returned an error");
```

3.25.2.17 ecr_file_image_write(), ecr_file_image_read()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_file_image_write()` function is used to save an image of one ECR's contents to a disk file.

The `ecr_file_image_read()` function is used to read an ECR image file into one ECR

Note the following:

- The `filename` argument specifies the input or output file name.
- If an absolute path is not specified the file will be located relative to the test program executable file, typically in the test program *Debug*\ folder.
- Using `ecr_file_image_write()`, no file clobber check is performed; i.e. any existing file of the same name WILL be over-written.

- Using `ecr_file_image_write()`, the current user must have write permission on the folder and file being written.
- In Multi-DUT Test Programs, only the ECR logging errors from the first DUT in the Active DUTs Set (ADS) is saved to disk or loaded from disk.

The following ECR information is written or read:

- Main ECR RAM
- Column RAM, Row RAM
- ECR Mini-RAM
- ECR Error Counters (Total Error Counters, IOC Error Counters, Row Error Counters, Col Error Counters.)
- ECR Counter Comparators values
- Address CrossPoint, X/Y Scramble RAMand Data CrossPoint configuration.

Usage

```
BOOL ecr_file_image_write( LPCTSTR filename );
BOOL ecr_file_image_read( LPCTSTR filename );
```

where:

`filename` specifies the desired input or output file name. See Description for rules.

Both functions return FALSE if any error(s) occur, otherwise TRUE is returned.

Example

```
if( ecr_file_image_write( "d:/myEcrOutputFileName") == FALSE)
   output(" ERROR: ecr_file_image_write() returned an error");
if( ecr_file_image_read( "d:/myEcrInputFileName") == FALSE)
   output(" ERROR: ecr_file_image_read() returned an error");
```

3.25.2.18 ecr_main_ram_scan()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_main_ram_scan()` function is used to read (scan) the Main ECR RAM and return a count of errors and, optionally, information about those errors.

In Multi-DUT Test Programs, only the ECR logging errors from the first DUT in the Active DUTs Set (ADS) is scanned.

Errors scanned can optionally be cached to local CPU memory. See `ecr_cache_enable()`. This is targeted at improving ECR scan (read) performance in situations where the ECR contents have not changed but the errors from a defined ECR address range need to be read multiple times.

Usage

The following function scans a specified area of the Main ECR RAM and returns a count of the errors up to a specified maximum:

```
int ecr_main_ram_scan( int rmin, int rmax,
                       int cmin, int cmax,
                       __int64 mask,
                       int max,
                       BOOL bit_cnt );
```

The following function scans a specified area of the Main ECR RAM and returns both an error count and details about the errors up to a specified maximum. The ECR X/Y scan direction is optionally specified (scan performance will be slower if the `fastdir` is different than that set using `ecr_config_set()`):

```
int ecr_main_ram_scan( int rmin, int rmax,
                       int cmin, int cmax,
                       __int64 mask,
                       int max,
                       struct PointFailure *failures,
           EcrFastDirection fastdir DEFAULT_VALUE( t_auto_fast ) );
```

The following function scans a specified area of the Main ECR RAM and returns both an error count and details about the errors up to a specified maximum per unmasked pin. The ECR X/Y scan direction is optionally specified (scan performance will be slower if the `fastdir` is different than that set using `ecr_config_set()`):

```
int ecr_main_ram_scan( int rmin, int rmax,
                       int cmin, int cmax,
                       __int64 mask,
                       int max,
```

```
                        struct PointFailure *failures,
                        WORD *counts,
                EcrFastDirection fastdir DEFAULT_VALUE( t_auto_fast ) );
```

where:

rmin, **rmax**, **cmin** and **cmax** are used to define a rectangular area of ECR addresses (inclusive) to be scanned.

mask is a bit-mask which can be used to ignore failures on a per-pin basis. A logic-1 enables failures for that bit position to be counted and returned, a logic-0 inhibits a failure from being counted/returned.

max is used in two contexts:

- Using the versions of `ecr_main_ram_scan()` which have a **bit_cnt** argument **max** will specify either a maximum number of failing *addresses* to count or a maximum number of failing *bits* to count. The **bit_cnt** argument determines the mode.
- Using the other versions of `ecr_main_ram_scan()`, **max** specifies a maximum number of failing *addresses* to count and return.

bit_cnt controls two things:

- Determines whether **max** specifies a maximum number of failing addresses (FALSE) or failing bits (TRUE). See **max** above.
- Determines whether `ecr_main_ram_scan()` returns the number of failing addresses counted (FALSE) or failing bits counted (TRUE).

failures is a pointer to an array of `PointFailure` structs, each element of which represents detailed information about one failure: row address, column address, and data. The **fastdir** argument controls which ECR axis is scanned fast (scan performance will be slower if the `fastdir` is different than that set using `ecr_config_set()`). User code is responsible for allocating memory for the **failures** array. See `ALLOCA_POINT_FAILURE()` for one method. Since the number of failures returned can vary, the number of `PointFailure` elements allocated must anticipate the worse case number of failures which *might* be returned. When using **failures**, proper memory management methods are critical to program reliability.

fastdir see **failures** above.

counts is a pointer to an existing `WORD` array, which contains a maximum count value for each pin which was logged to the ECR. The software assumes that each non-**mask**ed pin has a non-zero value in **counts**. During the scan operation, for each failing ECR address read, the appropriate value in **counts** is decremented for each bit failing at that address,

and the total failed bit count is incremented for each decrement performed. Then, for each pin which is not **mask**'ed, if the post-decrement value in **counts** = 0, a local mask bit is set, causing that pin/count value to be ignored for the rest of the scan. The function returns when one of the following occurs:

- The total failed address count = **max**
- When all bits are set in the local mask
- When the entire ECR region has been processed

pins identifies a PinList containing the pins of interest. Note that these can be any signal pins, including pins which span sites when Sites-per-Controller > 1.

The version of ecr_main_ram_scan() which has the **bit_cnt** argument returns either the number of failing addresses or number of failing bits, as controlled by **bit_cnt**.

The version of ecr_main_ram_scan() which has the **counts** argument returns an integer count of the number of failing addresses counted. See **counts** for details.

Example

The example below shows usage for several ECR functions and macros documented in this section. The example assumes that ecr_config_set() has previously configured the ECR, and that funtest() has executed a test pattern using one of the execution options which logs to the ECR (see Pattern Execution Stop Condition Options). This example uses each overload of ecr_main_ram_scan() to scan the ECR and report (datalog) up to 100 (MAX_FAILS) failing addresses for the low 8 data pins. This is not typical in that only one overload is typically used. The Program Output is shown below:

```
// ECR scan routine: get a list of rows containing errors into
// row_fails PointFailure array. Then for each of these rows scan
// main array, getting errors into main_fails1 PointFailure array.

// Allocate heap memory for the PointFailure array which stores
// failed row info. Sized as though every row failed.
ALLOC_POINT_FAILURE( row_fails, xmax() ); // ALLOC_POINT_FAILURE()

// Allocate heap memory for the PointFailure array which stores
// fails read from main array, one row at a time. Sized as though
// every column failed.
ALLOC_POINT_FAILURE( main_fails1, ymax() );
ALLOC_POINT_FAILURE( main_fails2, ymax() );
```

```
// Set mask to only scan 8 pins
int dmask = 0xff;
// Set max fail limit
#define MAX_FAILS 100

// This example assumes a multi-DUT program, Scan per active DUT
{
   ActiveDutIterator duts;
   while ( duts.More() ) {

      // Count and ID which rows have failures
      int row_fail_count =
         ecr_row_ram_scan(0, xmax(), dmask, xmax(), row_fails);

   output("\nFailed Addresses for DUT => %d ",active_dut_get()+1 );

      // Separate counters for each overload
      int fail_address_count1 = 0, fail_bit_count1 = 0;
      int fail_address_count2 = 0, fail_address_count3 = 0;

      // For each row with failures, scan main array and get fails
      for ( int i = 0; i < row_fail_count; ++i ) {
         int row = row_fails[ i ].row; // Get the next failed row

         //////////////  Overload #1A //////////////
         BOOL bit_cnt = FALSE;
         int main_fail_count1a =
            ecr_main_ram_scan(row, row, 0, ymax(),
                              dmask, MAX_FAILS, bit_cnt );

         if( main_fail_count1a > 0 ) {
            output("  #1A  Failed Address Count => %d",
                     main_fail_count1a );
            fail_address_count1 += main_fail_count1a;
         }
         else output("  #1A reports 0 fails");

         //////////////  Overload #1B //////////////
         bit_cnt = TRUE;
         int main_fail_count1b =
            ecr_main_ram_scan(row, row, 0, ymax(),
                              dmask, MAX_FAILS, bit_cnt );

         if( main_fail_count1b > 0 ) {
            output("  #1B  Failed Bit Count => %d",
                     main_fail_count1b );
```

```
      fail_bit_count1 += main_fail_count1b;
    }
    else output("  #1B reports 0 fails");
    //////////////  Overload #2 //////////////
    // Count and ID failures in this row
    int main_fail_count2 =
      ecr_main_ram_scan(row, row, 0, ymax(),
                        dmask, MAX_FAILS, main_fails1 );
    if( main_fail_count2 > 0 ) {
      for ( int i = 0; i < main_fail_count2; ++i )
        output("  #2 Row = %d, Col = %d, Data => 0x%I64x",
                  main_fails1[i].row,
                  main_fails1[i].col,
                  main_fails1[i].data);
      output("");
      fail_address_count2 += main_fail_count2;
    }
    else output("  #2 reports 0 fails");
    //////////////  Overload #3 //////////////
    WORD counts[] = { 1,2,3,4,5,6,7,8 }; // Size = 8 pins
    int main_fail_count3 =
      ecr_main_ram_scan(row, row, 0, ymax(), dmask,
                        MAX_FAILS, main_fails2, counts,
                        t_auto_fast );
    if( main_fail_count3 > 0 ) {
      for ( int j = 0; j < main_fail_count3; ++j )
        output("  #3 Row = %d, Col = %d, Data => 0x%I64x",
                  main_fails2[j].row,
                  main_fails2[j].col,
                  main_fails2[j].data);
      fail_address_count3 += main_fail_count3;
    }
    else output("  #3 reports 0 fails");
  output("Total failed addresses = %d\n", fail_address_count2);
 } // End: for each active DUT...
}
```

Program Output

TBD

3.25.2.19 ecr_cache_enable()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_cache_enable()` function is used to enable Error Catch RAM (ECR) error caching. This only affects subsequent executions of `ecr_main_ram_scan()` as described below.

This is targeted at improving ECR scan (read) performance in situations where the ECR contents have not changed but the errors from a defined ECR address range need to be read multiple times. The performance benefit of ECR error caching is obtained when testing 4 or more DUTs per Site Assembly Board.

Note the following:

- `ecr_cache_enable()` is used to enable or disable the ECR cache mode. The Active DUTs Set (ADS) and Ignored DUTs Set (IDS) have no effect on this function.
- `ecr_cache_enable(FALSE)` is set during initial program load. The cache enable state is not otherwise changed by the system software.
- When the cache mode is FALSE, all versions (overloads) of `ecr_main_ram_scan()` operate without caching errors in local CPU memory i.e. each execution of `ecr_main_ram_scan()` always reads the ECR hardware to return error data.
- When the cache mode is TRUE, the operation of `ecr_main_ram_scan()` changes, as follows:
  - Only the versions of `ecr_main_ram_scan()` which have the `PointFailure` argument are affected.
  - Memory is allocated to store errors read from the ECR. The memory allocation is sized as though every address within the range specified to `ecr_main_ram_scan()` contained an error. If insufficient memory is available `ecr_main_ram_scan()` will not cache any errors, and no warning is issued.

- The ECR is scanned (read) and any errors are stored in the cache. In Multi-DUT Test Programs, errors from all DUTs are cached by a given execution of `ecr_main_ram_scan()`.
- The errors from the first DUT in the Active DUTs Set (ADS) are returned via the `PointFailure` parameter.
- Any subsequent execution of `ecr_main_ram_scan()` which specifies an address range which completely resides within the cache will retrieve the errors from the cache. `ecr_main_ram_scan()` will be faster than when reading errors from the ECR hardware.
- Any subsequent execution of `ecr_main_ram_scan()` which specifies an address which is not *completely* resident in the cache will cause the ECR hardware to be scanned, for the entire address range specified. This does NOT reset the cache range and does NOT cause the new ECR scan errors to be cached.
- To reset the ECR cache address range requires user code to execute `ecr_cache_enable(FALSE)`, then `ecr_cache_enable(TRUE)`, before executing `ecr_main_ram_scan()` again.

- `ecr_cache_enable(FALSE)` invalidates any current ECR error cache content.

Usage

```
void ecr_cache_enable( BOOL on_off );
```

where:

`on_off` specifies whether ECR error caching is enabled (TRUE) or disabled (FALSE).

Example

```
ecr_cache_enable( TRUE );
```

3.25.2.20 ecr_miniram_config_set(), ecr_miniram_config_get()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_miniram_config_set()` function is used to configure the ECR Mini-RAM prior to use. See ECR Mini-RAM for the hardware description and target application description.

The `ecr_miniram_config_get()` function is used to get the current ECR Mini-RAM configuration.

Note the following:

- The ECR Mini-RAM is not configured by the system software; i.e. user code is required, using `ecr_miniram_config_set()`.
- Proper operation requires that `ecr_miniram_config_set()` be executed after configuring the ECR using `ecr_config_set()`, and prior to logging errors to the ECR.
- In hardware, the Mini-RAM is a 16K memory, storing one bit per DUT at each address. This limits the total number of address inputs to 14 (combined X + Y). The Mini RAM CrossPoint allows any combination of X vs. Y addresses to be used, as configured using `ecr_miniram_config_set()`.
- Each address in the Mini-RAM potentially represents many addresses in the Main ECR RAM. To configure the Mini-RAM, user code specifies how many X (row) address(es) and how many Y (column) address(es) are routed to the Mini RAM and which addresses are enabled (not masked). See Usage and Examples.
- The address inputs to the Mini RAM CrossPoint are not compressed; i.e. any address compression applied to the Main ECR RAM occurs after the addresses are routed to the Mini RAM CrossPoint.
- In Multi-DUT Test Programs, the Active DUTs Set (ADS) and Ignored DUTs Set (IDS) have no effect on `ecr_miniram_config_set()` or `ecr_miniram_config_get()`.

Usage

```
void ecr_miniram_config_set int numx,
                            int numy,
                            int x_compress_mask DEFAULT_VALUE(0),
                            int y_compress_mask DEFAULT_VALUE(0));
void ecr_miniram_config_get(int* numx,
                            int* numy,
                            int* x_compress_mask,
                            int* y_compress_mask);
```

where:

numx and **numy** are used in two contexts:

- In the set function, **numx** and **numy** specify the number of row and column address bits used to access the ECR Mini-RAM. These values configure the Mini-RAM with 2^{numx} rows and 2^{numy} columns. The maximum combined **numx** plus **numy** values must result in 14 or less total address bits.
- In the get function, **numx** and **numy** are the addresses of existing `int` variables used to return the currently programmed **numx** and **numy** values.

x_compress_mask is used in two contexts:

- In the set function, **x_compress_mask** is optional, and if used specifies a bit-wise value which determines which row address bits, of the 18 available, are enabled to the ECR Mini-RAM. A logic-1 enables a given address bit, a logic-0 disables the address bit. The default value (0) causes the system software to set a mask which enables **numx** MSB X address bits, where the specific bits are determined by the **numx** argument passed to `ecr_config_set()`. For example, if the **numx** argument passed to `ecr_config_set()` = 10, and the **numx** value passed to `ecr_miniram_config_set()` = 3, the default **x_compress_mask** will be 0x380.
- In the get function, **x_compress_mask** is a pointer to an existing `int` variable used to return the currently programmed **x_compress_mask** value.

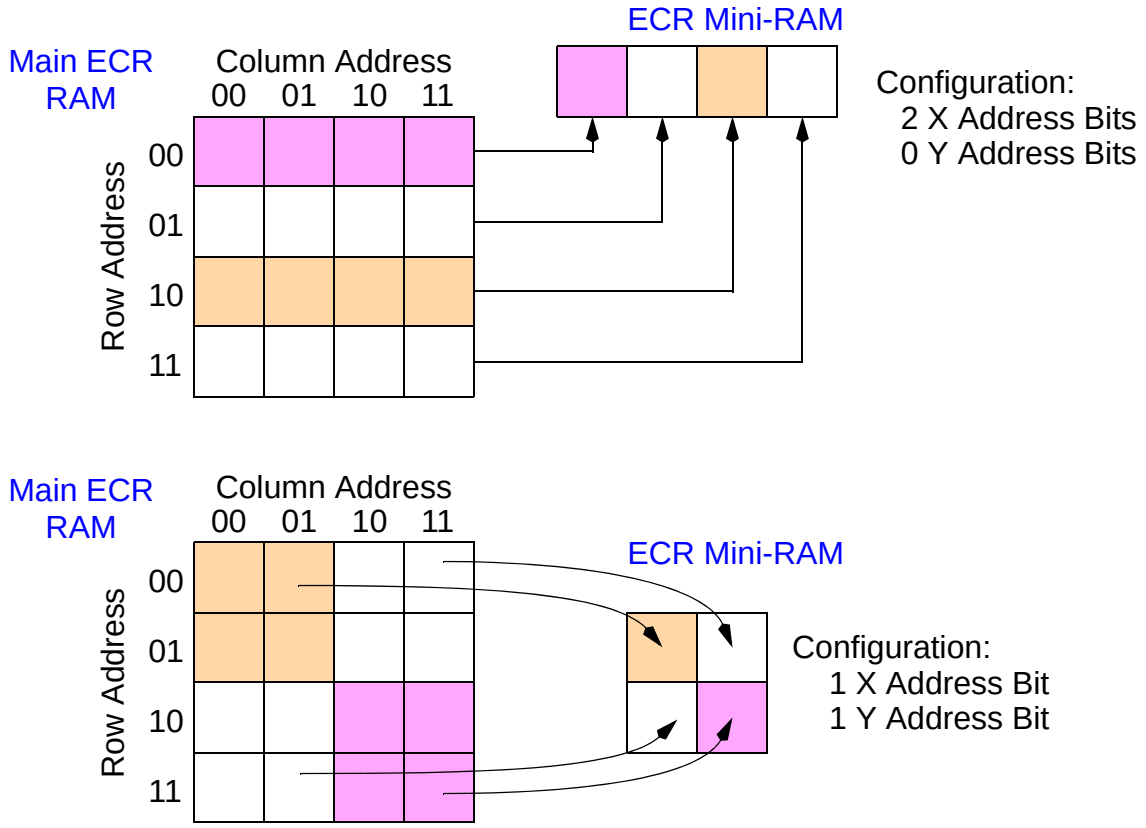
y_compress_mask is used in two contexts:

- In the set function, **y_compress_mask** is optional, and if used specifies a bit-wise value which determines which column address bits, of the 16 available, are enabled to the ECR Mini-RAM. A logic-1 enables a given address bit, a logic-0 disables the address bit. The default value (0) causes the system software to set a mask which enables **numy** MSB Y address bits where the specific bits are determined by the **numy** argument passed to `ecr_config_set()`. For example, if the **numy** argument passed to `ecr_config_set()` = 8, and the **numy** value passed to `ecr_miniram_config_set()` = 3, the default **y_compress_mask** will be 0xE0.
- In the get function, **y_compress_mask** is a pointer to an existing `int` variable used to return the currently programmed **y_compress_mask** value.

Note: if **x_compress_mask** and/or **y_compress_mask** are specified, there must be exactly **numx** logic-1 bits in the **x_compress_mask** and exactly **numy** logic-1 bits in the **y_compress_mask**.

Example

The following diagrams illustrate two different Mini RAM configurations (a more complex example follows). To keep the diagram manageable the ECR is configured to only use 2 X address bits and 2 Y address bits, for a total 16 Main ECR RAM addresses:



The top example shows the Mini RAM configured with 4 addresses, using 2 X address bits and 0 Y address bits. In this configuration, each ECR Mini-RAM address corresponds to an entire row in the Main ECR RAM. This configuration is obtained using one of the following:

```
ecr_miniram_config_set( 2, 0 );          // Default X/Y masks
ecr_miniram_config_set( 2, 0, 0, 0 );    // Default X/Y masks
ecr_miniram_config_set( 2, 0, 0x3 );     // Default Y mask
ecr_miniram_config_set( 2, 0, 0x3, 0 ); // Default Y mask
ecr_miniram_config_set( 2, 0, 0x3, 0x0 );
```

The bottom example shows the Mini RAM configured to use 1 X address bit and 1 Y address bit. Again, the Mini RAM will have 4 addresses (2 address bits), with each address

corresponding to a 2 x 2 square main ECR RAM addresses, as shown. This configuration is obtained using one of the following:

```
ecr_miniram_config_set( 1, 1 );          // Default X/Y masks
ecr_miniram_config_set( 1, 1, 0, 0 );    // Default X/Y masks
ecr_miniram_config_set( 1, 1, 0x1 );     // Default Y mask
ecr_miniram_config_set( 1, 1, 0x1, 0 ); // Default Y mask
ecr_miniram_config_set( 1, 1, 0x1, 0x1 );
```

The following example demonstrates the use of the X/Y address mask arguments to `ecr_miniram_config_set()`. This example assumes the ECR is configured to use 4 row address bits and 4 column address bits, for a total ECR size of 256:

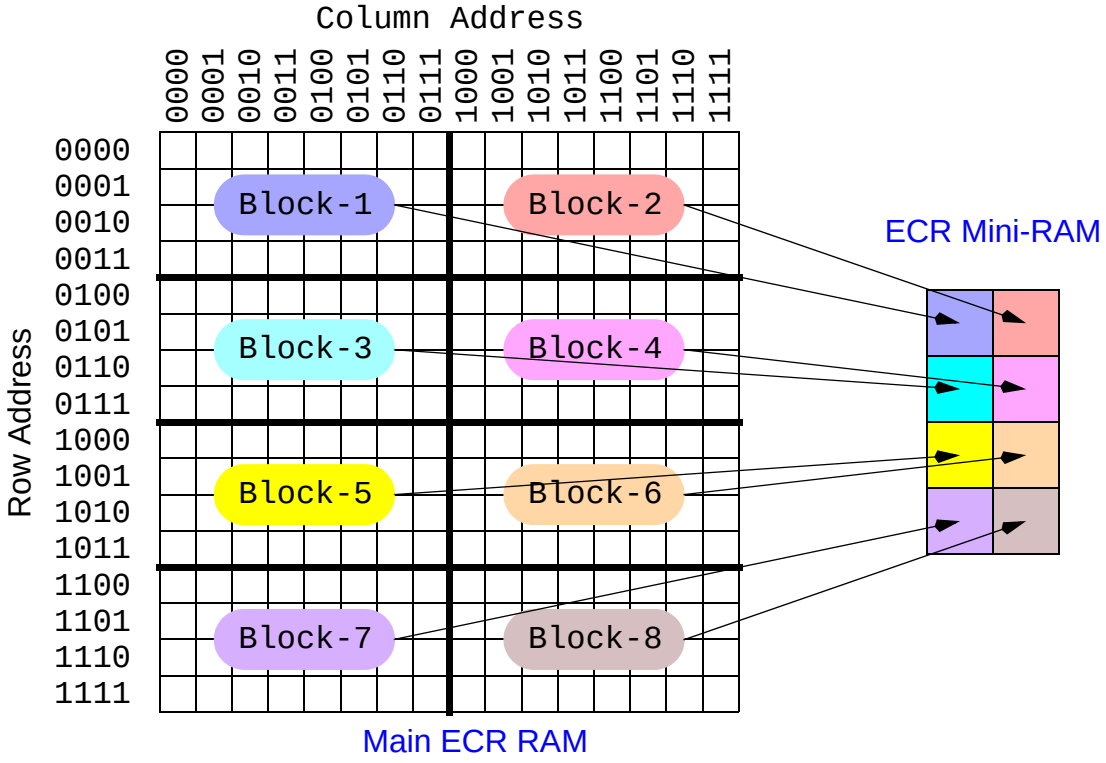


Figure-54: Mini-RAM Example Configuration

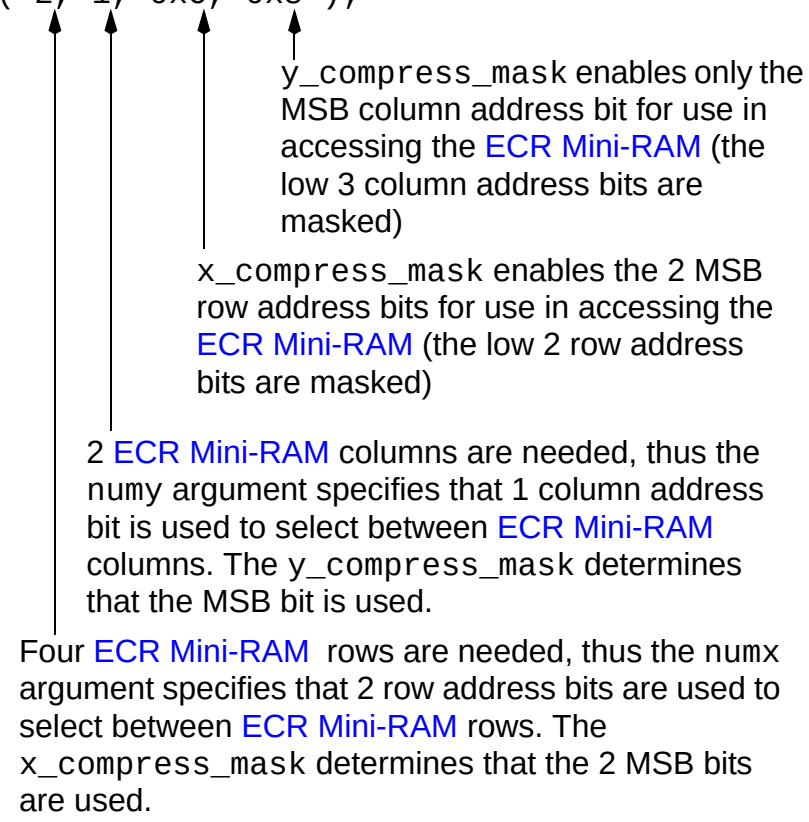
This configuration can be obtained using:

```
ecr_miniram_config_set( 2, 1 );          // Default X/Y masks
ecr_miniram_config_set( 2, 1, 0, 0 );    // Default X/Y masks
ecr_miniram_config_set( 2, 1, 0xC, 0 ); // Default Y mask
```

```
ecr_miniram_config_set( 2, 1, 0xC, 0x8 );
```

This was determined as follows:

```
ecr_miniram_config_set( 2, 1, 0xC, 0x8 );
```



3.25.2.21 ecr_miniram_scan()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_miniram_scan()` function is used to scan a range of ECR Mini-RAM addresses to obtain failure count and optionally failure details. Note the following:

- Each ECR Mini-RAM address actually represents an arbitrary number of Main ECR RAM addresses, as configured using `ecr_miniram_config_set()`. Each Mini-RAM address stores one bit indicating whether any of the associated Main ECR RAM addresses contain error(s).

- When a given Mini-RAM address/bit contains an error it indicates that one or more errors were logged at any of the DUT addresses mapped to the Mini-RAM.
- Two versions of `ecr_miniram_scan()` are provided:
  - The 1st only returns an error count.
  - The 2nd counts error addresses and also returns details about each error counted.
- In Multi-DUT Test Programs, only the ECR logging errors from the first DUT in the Active DUTs Set (ADS) is scanned.
- The `ecr_column_ram_scan()` function operates similarly on the Column RAMs and the `ecr_row_ram_scan()` function operates similarly on the Row RAMs.

Usage

The following function scans the specified range of ECR Mini-RAM address(es) counting either failing addresses or failing bits:

```
int ecr_miniram_scan(  int rmin,
                       int rmax,
                       int cmin,
                       int cmax,
                       int max,
                       BOOL bit_cnt );
```

The following function scans the specified range of ECR Mini-RAM address(es) counting errors. Error details are returned via the `failures` argument:

```
int ecr_miniram_scan(  int rmin,
                       int rmax,
                       int cmin,
                       int cmax,
                       int max,
                       struct PointFailure *failures );
```

where:

`rmin`, **`rmax`**, **`cmin`** and **`cmax`** identify a range of ECR Mini-RAM address(es) to be read where
0 <= **`rmin`** <= **`rmax`** <= `(( 1 << miniram_numx) -1)` and
0 <= **`cmin`** <= **`cmax`** <= `(( 1 << miniram_numy) -1)`. Note that these values should be specified in the context of the Mini-RAM; i.e. were there any errors logged to the Mini-RAM within the range of Mini-RAM **`rmin/cmin`** to **`rmax/cmax`**, as configured using `ecr_miniram_config_set()`).

max specifies a maximum number of errors to be read.

bit_cnt specifies whether error bits are counted (TRUE) or error addresses are counted (FALSE). Since the Mini-RAM only stores 1 bit per address, bit count and address count scans of the Mini-RAM will return identical results. This argument is included for consistency with the other scan functions.

failures is a pointer to an existing PointFailures array, used to return error details: Mini-RAM row address, column address, and data (0 or 1). User code is responsible for allocating memory for this array. See ALLOCA_POINT_FAILURE() for one method. Since the number of failures returned can vary, the number of PointFailure elements allocated must anticipate the worse case number of failures which *might* be returned. When using **failures**, proper memory management methods are critical to program reliability.

ecr_miniram_scan() returns the number of errors counted. This also equates to the number of valid elements in the **failures** array.

Example

The following example could be used to scan the Mini-RAM Example Configuration shown above:

```
PointFailure *fails = ALLOCA_POINT_FAILURE( 8 );
int c = ecr_miniram_scan( 0, 3, 0, 1, 8, fails );
```

3.25.2.22 ecr_overflow_get()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The ecr_overflow_get() function is used to determine whether a specific ECR Error Counters overflowed (i.e. counted past the maximum, rolled-over, etc.). Note the following:

- The ecr_any_overflow_get() function can be used to determine whether any counter overflowed. Then, the specific counter which overflowed can be determined using ecr_overflow_get().
- In Multi-DUT Test Programs, only the ECR from the first DUT in the Active DUTs Set (ADS) is checked.

Usage

```
BOOL ecr_overflow_get( EcrErrorCounters type );
```

where:

type specifies which of the ECR Error Counters is to be checked. Legal values are of the `EcrErrorCounters` enumerated type.

`ecr_overflow_get()` returns TRUE if the specified counter had overflowed and FALSE if not.

Example

The following example checks the Total Error Counter for each DUT in the Active DUTs Set (ADS):

```
ActiveDutIterator duts;
while ( duts.More() ) {
   if( ecr_overflow_get( t_tec ) == TRUE)
      output(" DUT-%d: the TEC DID overflow",
         active_dut_get()+1);
   else
      output(" DUT-%d: the TEC did NOT overflow",
         active_dut_get()+1);
}
```

3.25.2.23 ecr_row_ram_scan()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_row_ram_scan()` function is used to scan a range of Row RAM addresses to determine the number of row errors which have been logged. Note the following:

- Two versions of `ecr_row_ram_scan()` are provided:
  - The 1st only returns an error count, which can be a count of error bits or error addresses.
  - The 2nd counts error addresses and also returns details about each error counted.

- In Multi-DUT Test Programs, only the ECR logging errors from the first DUT in the Active DUTs Set (ADS) is scanned.
- The `ecr_column_ram_scan()` function operates similarly on the Column RAMs and the `ecr_miniram_scan()` function operates similarly on the ECR Mini-RAMs.

Usage

The following function scans the specified range of Row RAM address(es) counting either failing addresses or failing bits:

```
int ecr_row_ram_scan(  int rmin,
                       int rmax,
                       __int64 mask,
                       int max,
                       BOOL bit_cnt );
```

The following function scans the specified range of Row RAM address(es) counting failing addresses. Error details are returned via the `failures` argument:

```
int ecr_row_ram_scan(  int rmin,
                       int rmax,
                       __int64 mask,
                       int max,
                       struct PointFailure *failures );
```

where:

`rmin` and **`rmax`** identify a range of Row RAM address to be read where 0 <= **`rmin`** <= **`rmax`** <= `xmax()`.

`mask` is a bit-mask which can be used to ignore errors on a per-pin basis. A logic-1 enables the error for that bit position to be counted/returned, a logic-0 inhibits an error from being counted/returned. The order of bits in **`mask`** must correspond to the order of pins in the `datapins` argument specified for `ecr_config_set()`.

`max` specifies a maximum number of errors to be read. If this value is exceeded `ecr_row_ram_scan()` returns, potentially without scanning the entire Row RAM.

`bit_cnt` specifies whether error bits are counted (TRUE) or error addresses are counted (FALSE). Using the latter, any number of error bits at a given Row RAM address will count as one error.

`failures` is a pointer to an existing `PointFailure`s array, used to return error details: row address, column address (always 0 when scanning the Row RAM), and data. User code

is responsible for allocating memory for this array. See `ALLOCA_POINT_FAILURE()` for one method. Since the number of failures returned can vary, the number of `PointFailure` elements allocated must anticipate the worse case number of failures which *might* be returned. When using **`failures`**, proper memory management methods are critical to program reliability.

`ecr_row_ram_scan()` returns the number of errors counted. This also equates to the number of valid elements in the **`failures`** array.

Example

```
int c = ecr_row_ram_scan( 10, 20, 0xFF, 100, TRUE );
```

Also see Example.

3.25.2.24 ecr_write_mode_set(), ecr_write_mode_get()

See Error Catch RAM (ECR), Error Catch RAM Software.

Note: on 10/31/06 the Magnum 1 documentation for the ECR write mode was modified to remove references to the `t_abs_write` mode. The mode was never implemented due to hardware limitations. This effectively makes these two functions obsolete using Magnum 1 since the ECR write mode must be set to the only supported mode using `ecr_config_set()`. Thus, the text for these functions was removed.

3.25.2.25 ecr_area_clear()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_area_clear()` function is used to clear a specific area of an Error Catch RAM (ECR). Note the following:

- The term *clear* means to delete any errors logged within the specified region of ECR address(es).

- By default, `ecr_area_clear()` clears only the Main ECR RAM; i.e. the Row RAM, Column RAM and ECR Mini-RAM are not cleared. The `clear_rcmini` argument can be used to change this operation (not recommended, see Note:).
- `ecr_area_clear()` does not modify the ECR Error Counters.
- By default, the `all_duts` argument is FALSE which, in Multi-DUT Test Programs, causes `ecr_area_clear()` to only clear ECRs for DUT(s) currently in the Active DUTs Set (ADS). Setting `all_duts` = TRUE causes `ecr_area_clear()` to clear the ECR for all DUTs; i.e. ignore the Active DUTs Set (ADS).
- The ECR area to be cleared is specified using 4 values:
  - `rmin` and `cmin` identify the upper left corner of the area to be cleared.
  - `rmax` and `cmax` identify the lower right corner of the area to be cleared.
  - The ECR address region bounded by these corners will be cleared.
- A single ECR address can be cleared: set `rmin=rmax` and `cmin=cmax`. However, `ecr_error_set()` is a better method.
- A single ECR column can be cleared: set `cmin=cmax` to identify the column. Set `rmin` and `rmax` to identify which addresses within the column to be cleared.
- A single ECR row can be cleared: set `rmin=rmax` to identify the row. Set `cmin` and `cmax` to identify which addresses within the row are to be cleared.
- The default Maverick-I/-II `clear_area()` function maps to:
  `ecr_area_clear(r,r,c,c,TRUE,TRUE)`.

Note: proper Redundancy Analysis (RA) and BitmapTool operation depends upon synchronization between the contents of the main ECR RAM and values in all ECR RAMs and counters. These values are guaranteed to be valid after `funtest()` is executed. Conversely, these values are suspect ANY time user-code or ECRTool modifies some values in the ECR hardware. Beware! See `ecr_rams_update()`.

Also see `ecr_all_clear()`, `ecr_rams_clear()`, `ecr_counters_clear()`.

Usage

```
void ecr_area_clear(DWORD rmin,
                    DWORD rmax,
                    DWORD cmin,
                    DWORD cmax,
                    BOOL clear_rcmini DEFAULT_VALUE( FALSE ),
                    BOOL all_duts DEFAULT_VALUE( FALSE ) );
```

where:

rmin and **cmin** identify the upper left corner of the region to be cleared. These are zero-based values; i.e. 0,0 is the first address of the ECR (upper left corner).

rmax and **cmax** identify the lower right corner of the region to be cleared.

clear_rcmini is optional, and if used specifies whether the corresponding Row RAM, Column RAM and ECR Mini-RAM addresses should be cleared. Default = FALSE. Important: see Note:.

all_duts is optional, and only used in Multi-DUT Test Programs. See Description.

Example

The following function sets up the ECR for the subsequent clear examples:

```
ecr_config_set( numx(), numy(), datapins ); // ecr_config_set()
```

The following function clears only the first ECR address of the Main ECR RAM, for the DUT(s) in the Active DUTs Set (ADS):

```
ecr_area_clear( 0, 0, 0, 0 );
```

The following function clears the first row of the Main ECR RAM for all DUTs:

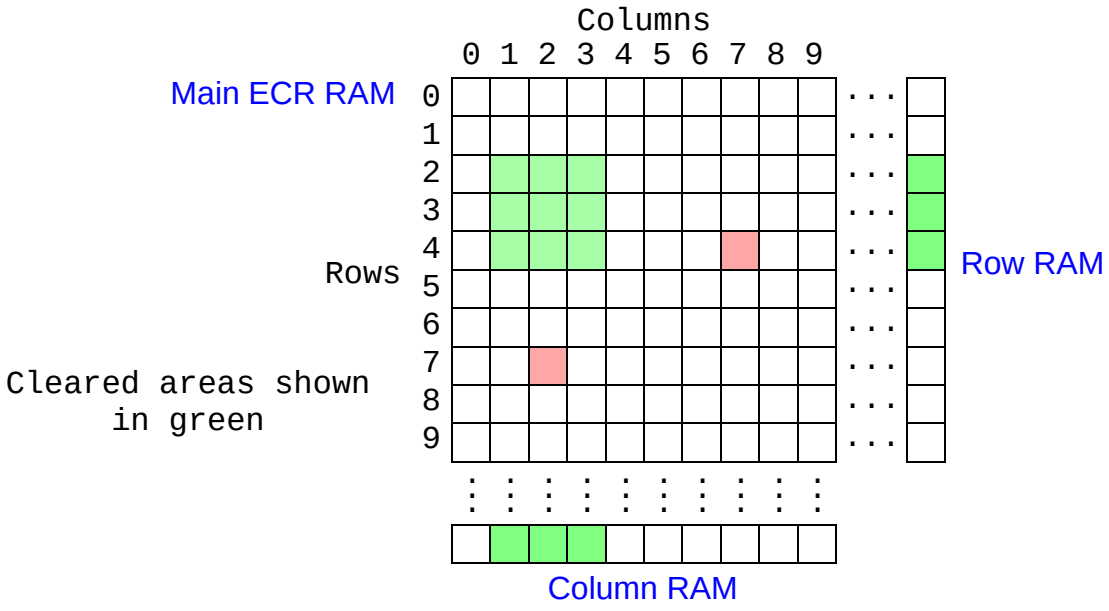
```
ecr_area_clear( 0, 0, 0, ymax(), FALSE, TRUE );
```

The following function clears the first column for DUT(s) in the Active DUTs Set (ADS):

```
ecr_area_clear( 0, xmax(), 2, 2);     // Clear entire third column
```

The following example clears a 3x3 area of the Main ECR RAM and the corresponding Row RAM, Column RAM, and ECR Mini-RAM (not shown below) addresses. Note that 2 errors (red) remain in the Main ECR RAM which are not correctly flagged in the Row RAM and Column RAM; i.e. when if the ECR Row RAM and/or Column RAM are scanned (read) these 2 errors will not be reported. In most applications, this is BAD. See Note:.

```
ecr_area_clear( 2, 4, 1, 3, TRUE );
```



3.25.2.26 ecr_col_ram_read()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_col_ram_read()` function is used to read error data from one or more ECR Column RAM addresses. Note the following:

- Two versions of `ecr_col_ram_read()` are provided:
  - The first returns the value of one location in the Column RAM.
  - The second returns an array of values from a range of Column RAM addresses, identified using the `cmin` and `cmax` arguments.
- Using the latter, each bit of each returned value represents a corresponding pin being logged to the ECR. The number of valid bits and the bit order matches the order of pins specified by the `datapins` argument to `ecr_config_set()`.
- In Multi-DUT Test Programs, the Column RAM of the ECR for the first DUT in the Active DUTs Set (ADS) is read.
- Also see `ecr_row_ram_read()`.

Usage

```
__int64 ecr_col_ram_read( int adr );
void ecr_col_ram_read( int cmin,
                       int cmax,
                       unsigned __int64* values,
                       int numValues );
```

where:

adr identifies one Column RAM address to be read.

cmin and **cmax** identify a range of Column RAM address(es) to be read where 0 <= **cmin** <= **cmax** <= `ymax()`.

values is a pointer to an existing `unsigned __int64` array used to return one or more values. The **values** array must be allocated by user code, and be [at least] ((**cmax** - **cmin**) + 1) elements long.

numValues specifies the size of the **values** array. If **numValues** < ((**cmax** - **cmin**) + 1), a warning is issued, and the contents of **values** is unchanged.

The first version of `ecr_col_ram_read()` returns the value read from the specified **adr**. See Description.

Example

The following example will read the Column RAM from address 5 through 10 and place the values read into the `vals` array at indexes 0 through 5. This requires an array with 6 elements:

```
unsigned __int64 vals[6];
ecr_col_ram_read( 5, 10, vals, 6);
```

3.25.2.27 ecr_col_ram_write()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_col_ram_write()` function is used to write value(s) into one or more ECR Column RAM addresses. Note the following:

- Two versions of `ecr_col_ram_write()` are provided:
  - The first writes to one specified Column RAM address.
  - The second writes an array of values to a range of Column RAM addresses, identified using the `cmin` and `cmax` arguments. Each array value represents the value written to one Column RAM address.
- Each bit of each value written represents a corresponding pin being logged to the ECR. The number of valid bits and the bit order must match the order of pins specified by the `datapins` argument to `ecr_config_set()`.
- In Multi-DUT Test Programs, values are written to the Column RAMs of all ECRs of DUT(s) in the Active DUTs Set (ADS).
- Also see `ecr_row_ram_write()`.

Note: proper Redundancy Analysis (RA) and BitmapTool operation depends upon synchronization between the contents of the main ECR RAM and values in all ECR RAMs and counters. These values are guaranteed to be valid after funtest() is executed. Conversely, these values are suspect ANY time user-code or ECRTool modifies some values in the ECR hardware. Beware! See ecr_rams_update().

Usage

```
void ecr_col_ram_write(int adr,
                       unsigned __int64 value );
void ecr_col_ram_write(int cmin,
                       int cmax,
                       unsigned __int64* values,
                       int numValues );
```

where:

`adr` identifies one Column RAM address to be written.

`value` specifies the value to be written to **`adr`**.

`cmin` and **`cmax`** identify a range of Column RAM address to be written where 0 <= **`cmin`** <= **`cmax`** <= `ymax()`.

`values` is an array containing one or more values to be written to the range of Column RAM address specified by **`cmin`** and **`cmax`**. The array must contain at least as many values as ((**`cmax`**-**`cmin`**)+1). If this rule is violated a warning is issued and the Column RAM is not modified.

numValues specifies the size of the **values** array.

Example

```
ecr_col_ram_write( 10, 0xA5 );
unsigned __int64 errs[] = {0x1, 0x2, 0x3,0x4 };
ecr_col_ram_write( 10, 13, errs, 4 );
```

3.25.2.28 ecr_counters_clear()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_counters_clear()` function is used to clear a specific Error Catch RAM (ECR) counter. Note the following:

- The term *clear* means to set the specified counter to 0.
- By default, all ECR counters are cleared. The `counter` argument can be used to identify specific ECR counter(s) to be cleared (see Note:):

EcrRamTypes	Hardware RAM
t_tec	Total Error Counters
t_rec	Row Error Counters
t_cec	Col Error Counters
t_ioc1 thru t_ioc36	Specified IOC Error Counters
t_all_ioc	All IOC Error Counters
t_all_ecr_counters	All the above

- The Main ECR RAM, Row RAM, Column RAM and ECR Mini-RAMare not modified.

- By default, the `all_duts` argument is FALSE which, in Multi-DUT Test Programs, causes `ecr_counters_clear()` to clear the counter(s) only for DUT(s) currently in the Active DUTs Set (ADS). Setting `all_duts` = TRUE causes `ecr_counters_clear()` to clear the counter(s) for all DUTs; i.e. ignore the Active DUTs Set (ADS).

Note: proper Redundancy Analysis (RA) and BitmapTool operation depends upon synchronization between the contents of the main ECR RAM and values in all ECR RAMs and counters. These values are guaranteed to be valid after funtest() is executed. Conversely, these values are suspect ANY time user-code or ECRTool modifies some values in the ECR hardware. Beware! See ecr_rams_update().

Also see `ecr_all_clear()`, `ecr_area_clear()`, `ecr_rams_clear()`.

Usage

```
void ecr_counters_clear(
     EcrErrorCounters counter DEFAULT_VALUE( t_all_ecr_counters ),
                        BOOL all_duts DEFAULT_VALUE ( FALSE ) );
```

where:

`counter` is optional, and if specified, identifies which counter(s) are to be cleared. Default = all ECR counters (`t_all_ecr_counters`). Legal values are of the `EcrErrorCounters` enumerated type (see table above).

`all_duts` is optional, and only useful in Multi-DUT Test Programs. **`all_duts`** determines whether the counter(s) of all ECRs are cleared (TRUE) or whether only the counter(s) of ECR(s) connected to DUT(s) in the Active DUTs Set (ADS) are cleared (FALSE, default).

Example

```
ecr_counters_clear();
ecr_counters_clear( t_all_ioc, TRUE );
```

3.25.2.29 ecr_error_add()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_error_add()` function is used to add one or more error(s) to the ECR RAMs (Main ECR RAM, Row RAM, Column RAM and ECR Mini-RAM). Note the following:

- `ecr_error_add()` can only add errors, it cannot remove errors (see `ecr_error_delete()` and `ecr_error_set()`).
- Two versions of `ecr_error_add()` are provided:
  - The 1st modifies a single ECR address.
  - The 2nd modifies one or more addresses, using an array of values. Array values are always written to the ECR in X-fast order.
- Errors are added using a bit-wise value, with each logic-1 bit adding an error on a corresponding pin being logged to the ECR. The number of valid bits and the bit order must match the order of pins specified by the `datapins` argument to `ecr_config_set()`. Logic-0 bit values have no effect on the ECR contents.
- Errors are added as appropriate to all ECR RAMs; i.e. Main ECR RAM, Row RAM, Column RAM and ECR Mini-RAM.
- The ECR Error Counters are not modified, see note below.
- In Multi-DUT Test Programs, the ECRs for all DUTs in the Active DUTs Set (ADS) are modified.

Note: proper Redundancy Analysis (RA) and BitmapTool operation depends upon synchronization between the contents of the main ECR RAM and values in all ECR RAMs and counters. These values are guaranteed to be valid after funtest() is executed. Conversely, these values are suspect ANY time user-code or ECRTool modifies some values in the ECR hardware. Beware! See ecr_rams_update().

Usage

```
void ecr_error_add(int row, int col, __int64 data);
void ecr_error_add( int rmin, int rmax,
                    int cmin, int cmax,
                    __int64* values,
                    int numValues);
```

where:

`row` and **`col`** identify a single ECR address to be modified, where 0 <= **`row`** <= `xmax()` and 0 <= **`col`** <= `ymax()`.

data specifies the value to write to the ECR. See Description.

rmin, **rmax**, **cmin** and **cmax** identify a range of ECR addresses to be modified, where 0 <= **rmin** <= **rmax** <= `xmax()` and 0 <= **cmin** <= **cmax** <= `ymax()`.

values is an array of values to be written to the ECR. See Description. The **values** array must include [at least] ((**rmax** - **rmin)+1**)*((**cmax** - **cmin**)+ 1) elements. If this rule is violated a warning is issued and the ECR is not modified.

numValues specifies the size of the **values** array.

Example

```
ecr_error_add( 10, 20, 0xA5 );
__int64 values[] = { 0x1, 0x2, 0x3, 0x4, 0x5, 0x6 };
ecr_error_add( 2, 3, 5, 7, values, 6 )
```

3.25.2.30 ecr_all_tecs_get(), ecr_all_ioc_get()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_all_tecs_get()` function is used to read the values from all Total Error Counters into a user-defined array.

The `ecr_all_ioc_get()` function is used to read the values from all IOC Error Counters into a user-defined array.

Note the following:

Values are read and stored for every DUT in the program, as defined in the Pin Assignment Table; i.e. the Active DUTs Set (ADS)is ignored.ECR Error Counters**Usage**

```
void ecr_all_tecs_get( __int64 *tec_array );
void ecr_all_ioc_get(  EcrErrorCounters counter,
                       __int64 *ioc_array );
```

where:

tec_array is a user-defined and allocated __int64 array, used to return the values read from the Total Error Counters. Values are stored in the array in DUT-number order; i.e. t_dut1, t_dut2, etc.

counter identifies one I/O counter to be read. Legal values are of the EcrErrorCounters enumerated type but only t_ioc1 to t_ioc36 are valid here.

ioc_array is a user-defined and allocated __int64 array, used to return the values read from **counter**. Values are stored in the array in DUT-number order; i.e. t_dut1, t_dut2, etc.

Example

```
__int64 vals[ 8 ]; // 8 DUTs in Pin Assignment Table
ecr_all_tecs_get( vals );
```

3.25.2.31 ecr_error_counter_set(), ecr_error_counter_get()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The ecr_error_counter_set() function is used to write a value to a specified ECR Error Counters.

The ecr_error_counter_get() function is used to read a value from a specified ECR Error Counters.

Note the following:

- The maximum legal count value depends upon the counter type:

Counter	Type	Bits	Max Count
Total Error Counters	t_tec	34	2^{34} -1 = 17,179,869,183

Counter	Type	Bits	Max Count
Row Error Counters	t_rec	18	2^{18} -1 = 262,143
Col Error Counters	t_cec	18	2^{18} -1 = 262,143
IOC Error Counters	t_ioc1 thru t_ioc36	32	2^{32} -1 = 4,294,967,295

- In Multi-DUT Test Programs, `ecr_error_counter_set()` sets the ECR for all DUT(s) in the Active DUTs Set (ADS) and `ecr_error_counter_get()` reads the ECR for the first DUT in the Active DUTs Set (ADS).

Usage

```
void ecr_error_counter_set( EcrErrorCounters counter,
                            __int64 value);
__int64 ecr_error_counter_get( EcrErrorCounters counter );
```

where:

`counter` identifies the counter to be accessed. Legal values are of the `EcrErrorCounters` enumerated type. See table above.

`value` specifies the value to write to **`counter`**. Legal values are 0 to maximum, where maximum is based on the specific counter being written. See table above.

`ecr_error_counter_get()` returns the current value from the **`counter`** read.

Example

```
ecr_error_counter_set( t_tec, 12345 );
__int64 val = ecr_error_counter_get( t_tec );
```

3.25.2.32 ecr_error_delete()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_error_delete()` function is used to delete an error from one Main ECR RAM, and optionally to update the ECR RAMs (Row RAM, Column RAM and ECR Mini-RAM), but not ECR Error Counters

Note the following:

- The `data` value specified is a bit-wise value. The number of valid bits and the bit order must match the order of pins specified by the `datapins` argument to `ecr_config_set()`. For a given bit (pin), a logic-1 deletes an error from the ECR and a logic-0 has no effect.
- By default, the other various ECR RAMs (see above) but not ECR Error Counters are updated after the specified ECR address is written. To do this may require scanning much or all of the Main ECR RAM, which can consume a noticeable amount of time. The optional `recalc` argument can be used to inhibit this update, but subsequent ECR scan (read) operations may not report errors correctly (see Note below). In general, the only time the update should be inhibited is when multiple ECR modifications are being made and `ecr_rams_update()` will be executed (last) to update the ECR Mini-RAMs and ECR Error Counters.
- In Multi-DUT Test Programs, the ECRs for all DUTs in the Active DUTs Set (ADS) are modified.
- Also see `ecr_error_set()` and `ecr_error_add()`.

Note: proper Redundancy Analysis (RA) and BitmapTool operation depends upon synchronization between the contents of the main ECR RAM and values in all ECR RAMs and counters. These values are guaranteed to be valid after funtest() is executed. Conversely, these values are suspect ANY time user-code or ECRTool modifies some values in the ECR hardware. Beware! See ecr_rams_update().

Usage

```
void ecr_error_delete(int row,
                      int col,
                      __int64 data,
                      BOOL recalc DEFAULT_VALUE(TRUE) );
```

where:

`row` and **`col`** identify a single ECR address to be modified, where 0 <= **`row`** <= `xmax()` and 0 <= **`col`** <= `ymax()`.

data specifies a bit-wise mask, indicating which errors are to be deleted. See Description.

recalc is optional, and if specified determines whether the various ECR RAMs are updated after the ECR modifications are complete. Default = TRUE; i.e. do update the ECR RAMs. Before changing this to FALSE, read the Description and the Note above. This argument has no effect on the ECR Error Counters, which are not updated.

Example

```
ecr_error_delete( 10, 20, 0xA5 );
```

3.25.2.33 ecr_error_get()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_error_get()` function is used to read one Main ECR RAM address.

In Multi-DUT Test Programs, only the ECR for the first DUT in the Active DUTs Set (ADS) is read.

Usage

```
__int64 ecr_error_get(int row, int col);
```

where:

row and **col** identify a single Main ECR RAM address to be read, where 0 <= **row** <= `xmax()` and 0 <= **col** <= `ymax()`.

`ecr_error_get()` returns the value read. The number of valid bits and the bit order will match the order of pins specified by the `datapins` argument to `ecr_config_set()`. For a given bit (pin), a logic-1 represents an error and a logic-0 represents no error.

Example

```
__int64 errs = ecr_error_get( 10, 20 );
```

3.25.2.34 ecr_error_set()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_error_set()` function is used to completely initialize one or more address(es) in the Main ECR RAM, and optionally to update the ECR RAMs (Row RAM, Column RAM and ECR Mini-RAM), but not ECR Error Counters.

Note the following:

- Two versions of `ecr_error_set()` are provided:
  - The 1st writes to a single Main ECR RAM address.
  - The 2nd writes to one or more addresses, using an array of values. Array values are always written to the Main ECR RAM in X-fast order.
- The value written to each address is bit-wise value. The number of valid bits and the bit order must match the order of pins specified by the `datapins` argument to `ecr_config_set()`. For a given bit (pin), a logic-1 inserts an error to the ECR and a logic-0 clears an error from the ECR.
- By default, the other various ECR RAMs (see above) but not ECR Error Counters are updated after the specified ECR addresses are written. This may require scanning much or all of the Main ECR RAM, which can consume a noticeable amount of time. The optional `recalc` argument can be used to inhibit this update, but subsequent ECR scan (read) operations may not report errors correctly (see Note below). In general, the only time the update should be inhibited is when multiple ECR modifications are being made and `ecr_rams_update()` will be executed (last) to update the ECR RAMs and ECR Error Counters.
- In Multi-DUT Test Programs, the ECRs for all DUTs in the Active DUTs Set (ADS) are modified.
- Also see `ecr_error_delete()` and `ecr_error_add()`.

Note: proper Redundancy Analysis (RA) and BitmapTool operation depends upon synchronization between the contents of the main ECR RAM and values in all ECR RAMs and counters. These values are guaranteed to be valid after funtest() is executed. Conversely, these values are suspect ANY time user-code or ECRTool modifies some values in the ECR hardware. Beware! See ecr_rams_update().

Usage

```
void ecr_error_set( int row,
                    int col,
                    __int64 data,
                    BOOL recalc DEFAULT_VALUE( TRUE ));
void ecr_error_set( int rmin, int rmax,
                    int cmin, int cmax,
                    __int64* values,
                    int numValues,
                    BOOL recalc DEFAULT_VALUE( TRUE ));
```

where:

row and **col** identify a single ECR address to be modified, where 0 <= **row** <= xmax() and 0 <= **col** <= ymax().

data specifies the value to write to the ECR. See Description.

recalc is optional, and if specified determines whether the various ECR RAMs are updated after the ECR modifications are complete. Default = TRUE; i.e. do update the ECR RAMs. Before changing this to FALSE, read the Description and the Note above. This argument has no effect on the ECR Error Counters, which are not updated.

rmin, **rmax**, **cmin** and **cmax** identify a range of ECR address to be modified, where 0 <= **rmin** <= **rmax** <= xmax() and 0 <= **cmin** <= **cmax** <= ymax().

values is an array of values to be written to the ECR. See Description. The **values** array must include [at least] ((**(rmax** - **rmin**) +1) * (**cmax** - **cmin**) + 1)) elements. If this rule is violated a warning is issued and the ECR is not modified.

numValues specifies the size of the **values** array.

Example

```
ecr_error_set( 10, 20, 0xA5 );
__int64 values[] = { 0x1, 0x2, 0x3, 0x4, 0x5, 0x6 };
ecr_error_set( 2, 3, 5, 7, values, 6 );
```

3.25.2.35 ecr_miniram_read()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_miniram_read()` function is used to read one or more ECR Mini-RAM addresses. Note the following:

- The addresses to be read are specified in the context of the ECR Mini-RAM, not DUT addresses or ECR addresses. See ECR Mini-RAM for a description of the hardware and `ecr_miniram_config_set()` for a description of ECR Mini-RAM addressing and how it maps to the main ECR.
- Values are returned in a user-defined array.
- Only the LSB bit of each value read is valid. A logic-1 indicates an error was logged to the ECR Mini-RAM address and a logic-0 indicates no error was logged to the ECR Mini-RAM address. As noted in `ecr_miniram_config_set()`, a given Mini-RAM error can represent a large number of errors in the Main ECR RAM.
- The ECR Mini-RAM is always read in the X-fast direction.
- A single address can be read by setting `xmax` = `xmin` and `ymax` = `ymin`.
- In Multi-DUT Test Programs, the ECR Mini-RAM of the ECR for the first DUT in the Active DUTs Set (ADS) is read.
- Also see `ecr_miniram_scan()`.

Usage

The following function reads one ECR Mini-RAM address:

```
int ecr_miniram_read( int x, int y );
```

The following function reads one or more ECR Mini-RAM address(es):

```
void ecr_miniram_read( int xmin,
                       int xmax,
                       int ymin,
                       int ymax,
                       int* values,
                       int numValues);
```

where:

x and **y** specify one address to be read.

xmin, **xmax**, **ymin** and **ymax** identify a range of ECR Mini-RAM address to be read, where:

- 0 <= **xmin** <= **xmax** <= `MiniRam_xmax`, where `MiniRam_xmax` is determined by the `numx` argument to `ecr_miniram_config_set()`; i.e. `MiniRam_xmax` = 2^{numx}.

- 0 <= **ymin** <= **ymax** <= MiniRam_ymax, where MiniRam_ymax is determined by the numy argument to ecr_miniram_config_set(); i.e. MiniRam_xmax = 2^{numy}.

values is a pointer to an existing user-defined array used to return a value for each address read. User code is responsible for allocating array memory, which must include [at least] ((**xmax** - **xmin**) +1) * ((**ymax** - **ymin**) + 1)) elements. If this rule is violated a warning is issued and no values are returned.

numValues specifies the size of the **values** array.

Example

```
int vals[12];
ecr_miniram_read( 0, 2, 0, 3 , vals, 12);
```

3.25.2.36 ecr_miniram_write()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The ecr_miniram_write() function is used to write errors to one or more ECR Mini-RAM addresses. Note the following:

- ecr_miniram_write() does NOT update any other ECR RAMs (Main ECR RAM, Row RAM, Column RAM and ECR Mini-RAM) or ECR Error Counters. See Note below.
- The addresses to be written are specified in the context of the ECR Mini-RAM, not DUT addresses or ECR addresses. See Mini-RAM for a description of the hardware and ecr_miniram_config_set() for a description of Mini-RAM addressing and how it maps to the main ECR. As noted in ecr_miniram_config_set(), a given Mini-RAM error can represent a large number of errors in the Mini-RAM.
- Value(s) are written to the ECR Mini-RAM using an array, with each element written to one Mini-RAM address. Values are always written in the X-fast order.
- Only the LSB bit of each value is valid. A logic-1 represents an error and a logic-0 represents no error.
- In Multi-DUT Test Programs, the ECRs for all DUTs in the Active DUTs Set (ADS) are modified.

Note: proper Redundancy Analysis (RA) and BitmapTool operation depends upon synchronization between the contents of the main ECR RAM and values in all ECR RAMs and counters. These values are guaranteed to be valid after funtest() is executed. Conversely, these values are suspect ANY time user-code or ECRTool modifies some values in the ECR hardware. Beware! See ecr_rams_update().

Usage

The following function modifies one ECR Mini-RAM address:

```
void ecr_miniram_write(int x, int y, int value);
```

The following function modifies one or more Mini-RAM address(es):

```
void ecr_miniram_write(int xmin,
                       int xmax,
                       int ymin,
                       int ymax,
                       int* values,
                       int numValues);
```

where:

x and **y** specify one address to be written.

xmin, **xmax**, **ymin** and **ymax** identify a range of Mini-RAM address to be written, where:

- 0 <= **xmin** <= **xmax** <= `MiniRam_xmax`, where `MiniRam_xmax` is determined by the `numx` argument to `ecr_miniram_config_set()`; i.e. `MiniRam_xmax` = (2^{numx} -1).
- 0 <= **ymin** <= **ymax** <= `MiniRam_ymax`, where `MiniRam_ymax` is determined by the `numy` argument to `ecr_miniram_config_set()`; i.e. `MiniRam_xmax` = (2^{numy} -1).

values is a user-defined array of values. See Description. The array must contain at least as many values as ((**xmax** - **xmin**) +1) * (**ymax** - **ymin**) +1). If this rule is violated a warning is issued and the ECR Mini-RAM is not modified.

numValues specifies the size of the **values** array.

Example

```
int errs[] = { 1, 0, 1, 1, 1, 0 };
ecr_miniram_write( 0, 1, 0, 2, errs, 6 );
```

3.25.2.37 ecr_rams_clear()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_rams_clear()` function is used to clear a specific Error Catch RAM (ECR) RAMs. Note the following:

- The term *clear* means to delete any errors logged in the specified RAM.
- The `ram_type` argument identifies the ECR RAM to be cleared:

EcrRamTypes	Hardware RAM
`t_main_array`	Main ECR RAM
`t_row_catch`	Row RAM
`t_col_catch`	Column RAM
`t_mini`	ECR Mini-RAM
`t_all_ecr_rams`	All the above

- The ECR Error Counters are not modified, see `ecr_counters_clear()` and `ecr_all_clear()`.
- By default, the `all_duts` argument is FALSE which, in Multi-DUT Test Programs, causes `ecr_rams_clear()` to only clear the RAM(s) of ECRs for DUT(s) currently in the Active DUTs Set (ADS). Setting `all_duts` = TRUE causes `ecr_rams_clear()` to clear the RAM(s) of all ECRs; i.e. ignore the Active DUTs Set (ADS).

Note: proper Redundancy Analysis (RA) and BitmapTool operation depends upon synchronization between the contents of the main ECR RAM and values in all ECR RAMs and counters. These values are guaranteed to be valid after funtest() is executed. Conversely, these values are suspect ANY time user-code or ECRTool modifies some values in the ECR hardware. Beware! See ecr_rams_update().

Also see `ecr_all_clear()`, `ecr_area_clear()`, `ecr_counters_clear()`.

Usage

```
void ecr_rams_clear(EcrRamTypes ram_type,
                    BOOL all_duts DEFAULT_VALUE( FALSE ) );
```

where:

`ram_type` identifies the ECR RAM to be cleared. Legal values are of the `EcrRamTypes` enumerated type. See table above.

`all_duts` is optional, and only used in Multi-DUT Test Programs. **`all_duts`** determines whether the RAM(s) of all ECRs are cleared (default = FALSE) or whether only the RAM(s) of ECR(s) connected to DUT(s) in the Active DUTs Set (ADS) are cleared (TRUE). See Description.

Example

The following example clears only the Main ECR RAM. In Multi-DUT Test Programs, only ECRs for DUT(s) currently in the Active DUTs Set (ADS) are affected:

```
ecr_rams_clear( t_main_array );
```

3.25.2.38 ecr_rams_update()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_rams_update()` function is used to re-synchronize the Row RAM, Column RAM and ECR Mini-RAM and optionally the ECR Error Counters. This may be necessary after modifying the contents of the Main ECR RAM using any of `ecr_error_delete()`,

ecr_error_set(), ecr_area_clear() (but not ecr_error_add()) or using ECRTool. See Note below.

By default, the Row RAM, Column RAM and ECR Mini-RAM and ECR Error Counters are all updated when ECR contents are modified using ecr_error_delete(), ecr_error_set() or ecr_area_clear(), but this operation can be over-ridden. And, if the Main ECR RAM contents are modified using ECRTool, these RAMs/counters are guaranteed to be out-of-sync with the main ECR RAM contents.

Note: proper Redundancy Analysis (RA) and BitmapTool operation depends upon synchronization between the contents of the main ECR RAM and values in all ECR RAMs and counters. These values are guaranteed to be valid after funtest() is executed. Conversely, these values are suspect ANY time user-code or ECRTool modifies some values in the ECR hardware. Use ecr_rams_update() to reestablish synchronization.

Executing ecr_rams_update() causes the entire Main ECR RAM to be scanned, which can consume a noticeable amount of time. To improve this performance, each time the ECR is modified using the functions noted above, the system software records the changes made to the ECR. Then, to re-synchronize the RAMs/counters with the main ECR RAM this change list is used by ecr_rams_update() to selectively update the RAMs/counters.

The main ECR RAM and the associated RAMs/counters are also re-synchronized any time the ECR is cleared, which can be done using ecr_all_clear() or ecr_config_set(), which also clears the ECR.

Beginning in software release h2.3.xx/h1.3.xx, after 16k modifications are made to the main ECR RAM without performing either of the above and without using ecr_rams_update(), the system software stops tracking any new changes to the main RAM. Then, if ecr_rams_update() is executed the system software assumes that the entire ECR has been modified and re-scans the entire ECR (prior to 16K changes ecr_rams_update() uses the change list to scan only the area of the ECR that has been changed, which may be faster). This can be prevented by clearing the ECR (ecr_all_clear()), reconfiguring the ECR (ecr_config_set()) or executing ecr_rams_update() before 16K ECR modifications are made as described above.

In Multi-DUT Test Programs, ecr_rams_update() only affects the ECRs for DUT(s) currently in the Active DUTs Set (ADS).

In Multi-DUT Test Programs, the ECRs for all DUTs in the Active DUTs Set (ADS) are modified.

Usage

```
void ecr_rams_update(
        BOOL update_counters DEFAULT_VALUE ( FALSE) );
```

where:

`update_counters` is optional, and if specified determines whether the various ECR Error Counters are also updated (TRUE) or not updated (FALSE). Default = FALSE.

Example

```
ecr_rams_update( TRUE );
```

3.25.2.39 ecr_row_ram_read()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_row_ram_read()` function is used to read error data from one or more ECR Row RAM addresses. Note the following:

- Two versions of `ecr_row_ram_read()` are provided:
  - The first returns the value in one specified Row RAM address.
  - The second returns an array of values from a range of Row RAM addresses, identified using the `rmin` and `rmax` arguments.
- Using the latter, each bit of each returned value represents a corresponding pin being logged to the ECR. The number of valid bits and the bit order matches the order of pins specified by the `datapins` argument to `ecr_config_set()`.
- In Multi-DUT Test Programs, only the Row RAM of the ECR for the first DUT in the Active DUTs Set (ADS) is read.
- Also see `ecr_col_ram_read()`.

Usage

```
__int64 ecr_row_ram_read( int adr );
```

```
void ecr_row_ram_read( int rmin,
                       int rmax,
                       unsigned __int64* value,
                       int numValues );
```

where:

adr identifies one Row RAM address to be read.

rmin and **rmax** identify a range of Row RAM address to be read where 0 <= **rmin** <= **rmax** <= `xmax()`.

values is a pointer to an existing `unsigned __int64` array used to return one or more error values. The **values** array must be allocated by user code, and be [at least] ((**rmax** - **rmin**) + 1) elements long.

numValues specifies the size of the **values** array. If **numValues** < ((**rmax** - **rmin**) + 1), a warning is issued, and the contents of **values** is unchanged.

The first version of `ecr_row_ram_read()` returns the number of errors read from the specified **adr**. See Description.

Example

The following example will read the Row RAM from address 3 through 8 and place the values read into the `vals` array at indexes 0 through 5. This requires an array with 6 elements:

```
unsigned __int64 vals[6];
ecr_row_ram_read( 3, 8, vals, 6);
```

3.25.2.40 ecr_row_ram_write()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_row_ram_write()` function is used to write value(s) into one or more ECR Row RAM addresses. Note the following:

- Two versions of `ecr_row_ram_write()` are provided:
  - The first writes to one specified Row RAM address.

  - The second writes an array of values to a range of Row RAM addresses, identified using the `rmin` and `rmax` arguments. Each array value represents the value written to one Row RAM address.
- Each bit of each value written represents a corresponding pin being logged to the ECR. The number of valid bits and the bit order must match the order of pins specified by the `datapins` argument to `ecr_config_set()`.
- In Multi-DUT Test Programs, values are written to the Row RAMs of all ECRs of DUT(s) in the Active DUTs Set (ADS).
- Also see `ecr_col_ram_write()`.

Note: proper Redundancy Analysis (RA) and BitmapTool operation depends upon synchronization between the contents of the main ECR RAM and values in all ECR RAMs and counters. These values are guaranteed to be valid after funtest() is executed. Conversely, these values are suspect ANY time user-code or ECRTool modifies some values in the ECR hardware. Beware! See ecr_rams_update().

Usage

```
void ecr_row_ram_write( int adr, unsigned __int64 value );
void ecr_row_ram_write(int rmin,
                       int rmax,
                       unsigned __int64* value,
                       int numValues );
```

where:

`adr` identifies one Row RAM address to be written.

`value` specifies the value to be written to **`adr`**.

`rmin` and **`rmax`** identify a range of Row RAM address to be written where 0 <= **`rmin`** <= **`rmax`** <= `xmax()`.

`values` is an array containing one or more values to be written to the range of Row RAM address specified by **`rmin`** and **`rmax`**. The array must contain at least as many values as ((**`rmax`**-**`rmin`**)+1). If this rule is violated a warning is issued and the Row RAM is not modified.

`numValues` specifies the size of the **`values`** array.

Example

```
ecr_row_ram_write( 13, 0xA5 );

unsigned __int64 errs[] = {0x1, 0x2, 0x3,0x4 };
ecr_col_ram_write( 22, 25, errs, 4 );
```

3.25.2.41 ecr_scramble_bank_set(), ecr_scramble_bank_get()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The ECR has 4 banks each of X/Y Scramble RAM. The X/Y Scramble RAMs allow the X and/or Y address bit(s) mapping to the ECR to be modified.

The `ecr_scramble_bank_set()` function is used to select which of 4 X and/or Y Scramble RAM banks is enabled.

The `ecr_scramble_bank_get()` function is used to get the current bank selections.

Note the following:

- During initial program load, bank-0 is selected for both X and Y addresses. The system software does not otherwise change this selection.
- The currently selected banks are used during ECR capture operations.
- Only the currently selected banks are accessed by `ecr_scramble_ram_write(), ecr_scramble_ram_read()`.

Usage

```
void ecr_scramble_bank_set(int x_bank, int y_bank);

void ecr_scramble_bank_get(int* x_bank, int* y_bank);
```

where:

x_bank and **y_bank** are used in two contexts:

- In the set function, they are used to select the bank. Legal values are 0 to 3, representing bank-0 to bank-3.
- In the get function, these are pointers to 2 existing `int` variables used to return the current bank selection.

Example

The following example sets bank-2 for both the X and Y Scramble RAM, then gets the current selection:

```
ecr_scramble_bank_set( 2, 2 );
int x_bank, y_bank;
ecr_scramble_bank_set( &x_bank, &y_bank );
```

3.25.2.42 ecr_scramble_ram_write(), ecr_scramble_ram_read()

See Error Catch RAM (ECR), Error Catch RAM Software.

Description

The `ecr_scramble_ram_write()` function is used to write values to the X/Y Scramble RAMs.

The `ecr_scramble_ram_read()` function can be used to read values from the X/Y Scramble RAMs.

Note the following:

- The X/Y Scramble RAM has 4 banks, selectable using the `ecr_scramble_bank_set()` function. The currently selected bank is accessed when using `ecr_scramble_ram_write()` and `ecr_scramble_ram_read()`.
- During the initial program load, bank-0 is linearized, the other 3 banks are not configured i.e. they contain random contents. The system software does not otherwise modify the X/Y Scramble RAM.

Usage

```
void ecr_scramble_ram_write(EcrScrambleRamTypes type,
                            int start,
                            int stop,
                            int* values,
                            int numValues);
```

```
void ecr_scramble_ram_read( EcrScrambleRamTypes type,
                            int start,
                            int stop,
                            int* values,
                            int numValues);
```

where:

type selects whether the X (t_x_scramble) or Y (t_y_scramble) Scramble RAM is being accessed. Legal values are of the EcrScrambleRamTypes enumerated type.

start and **stop** specify the first and last address being accessed.

values is an array of **numValues** size. In the set function, **values** contains the values to be written to the selected Scramble RAM. In the get function, **values** returns the values read from the selected Scramble RAM.

If **numValues** < (**stop**-**start**)+1 a warning is issued and the function returns immediately.

Example

```
int write_values[] = { 0, 2, 4, 6, 8, 1, 3, 5, 7, 9 };
ecr_scramble_ram_write( t_x_scramble, 0, 9, write_values, 10 );
int read_values[ 10 ];
ecr_scramble_ram_read( t_x_scramble, 0, 9, read_values, 10 );
```

3.25.2.43 ecr_x_y_data_set()

See Error Catch RAM (ECR), Error Catch RAM Software.

Note: on 10/14/2008, this function was un-documented as a way of deprecating its use. Use ecr_config_set(), to ensure all dependent hardware and software is correctly configured.

3.25.3 ECR DDR Functions

See Error Catch RAM (ECR), Error Catch RAM Software, Double Data Rate (DDR) Mode.

This section contains the following:

- `ecr_ddr_mode_set(), ecr_ddr_mode_get()`

3.25.3.1 ecr_ddr_mode_set(), ecr_ddr_mode_get()

See Error Catch RAM (ECR), Error Catch RAM Software, Double Data Rate (DDR) Mode.

Description

The `ecr_ddr_mode_set()` function is used to enable DDR ECR use. The `ecr_ddr_mode_get()` function is used to look-up the current DDR state of the ECR.

Note: there is an execution order dependency between `ecr_ddr_mode_set()` and `ecr_config_set()`. `ecr_ddr_mode_set()` is used to set a mode which is detected by `ecr_config_set()`. Thus `ecr_ddr_mode_set()` ***MUST*** be executed before `ecr_config_set()`.

When `ecr_config_set()` is executed, a check is made to see if `ecr_ddr_mode_set()` has set DDR mode and the ECR is configured accordingly. See DDR Fail Signal MUX: Memory Error Catch.

To toggle between DDR and non-DDR mode is simpler, provided the basic ECR configuration is not changed:

- To toggle to non-DDR, execute `ecr_ddr_mode_set(FALSE)`.
- To re-enable DDR, execute `ecr_ddr_mode_set(TRUE,t_xx)`, where `xx` identifies the *address differentiator* (see DDR Fail Signal MUX: Memory Error Catch).
- When `ecr_ddr_mode_set()` is executed to change the address differentiator (`t_xx` above) `ecr_config_set()` MUST be executed again.
- If `ecr_config_set()` is to be executed again, as might be done to change the ECR configuration, the `ecr_ddr_mode_set()` function must also be executed again, *before* `ecr_config_set()`.
- The Active DUTs Set (ADS) and Ignored DUTs Set (IDS) have no effect on these functions.

Usage

The following function is used to set the ECR DDR mode configuration:

```
void ecr_ddr_mode_set( BOOL allow_ddr, TesterFunc ddr_adr );
```

The following function is used to get the current ECR DDR configuration:

```
BOOL ecr_ddr_mode_get( TesterFunc *ddr_adr );
```

where:

allow_ddr specifies whether the ECR is to be configured for DDR capture (TRUE) or non-DDR capture (FALSE).

ddr_adr is used in two contexts:

- In the setter function, **ddr_adr** specifies the address differentiator (see DDR Fail Signal MUX: Memory Error Catch). Legal values are one of the TesterFunc enumerated types.
- In the getter function, **ddr_adr** is a pointer to an existing TesterFunc variable used to return the current address differentiator value.

ecr_ddr_mode_get returns the ECR's current DDR configuration state.

Example

```
ecr_ddr_mode_set( TRUE, t_y0 );
TesterFunc f;
BOOL state = ecr_ddr_mode_get( &f );
```

3.25.4 ECR Simulation

See Error Catch RAM Software.

To simulate Magnum 1/2/2x ECR operation only requires performing the Magnum 1/2/2x Simulation Setup.

Note the following:

- User code must insert failures into the simulated ECR, using ecr_error_set() and/or ecr_error_add(). Executing a pattern (i.e. funtest()) does not add errors to the simulated ECR.
- The simulated ECR uses computer RAM to store errors. Simulating a large device may require more memory than is available in the computer.

3.25.4.1 Magnum 1 vs. Maverick ECR Functions

See Error Catch RAM Software.

This section provides a very limited correlation of Maverick ECR functions to Magnum 1 ECR functions.

As stated at the start of this chapter:

- DO NOT mix the Maverick ECR functions with the Magnum 1 ECR functions: they DO NOT inter-operate.
- For all new Magnum 1 test programs, it is highly recommended that the new ECR functions be used.

The following table lists the Maverick-I/-II ECR functions with the Magnum 1 function which performs an equivalent or related operation:

Maverick Function	Magnum Function	Notes
make_ecr()	ecr_config_set()	
clear()	ecr_all_clear() ecr_rams_clear() ecr_counters_clear()	
clear_area()	ecr_area_clear()	
set_error() get_error()	ecr_error_set() ecr_error_get()	
add_error()	ecr_error_add()	
del_error()	ecr_error_delete()	
scan_x()	ecr_column_ram_scan() ecr_row_ram_scan()	
scan_rc()	ecr_main_ram_scan()	
configure()	ecr_config_set()	Not documented for Maverick
set_numx() set_numy()	TBD ecr_x_y_data_set() ???	Not documented for Maverick

Maverick Function	Magnum Function	Notes
set_datawidth()	TBD ecr_x_y_data_set() ???	Not documented for Maverick
set_x_compress_mask() set_y_compress_mask()	TBD ecr_config_set() ???	Not documented for Maverick
set_data_compress()	TBD ecr_config_set() ???	Not documented for Maverick
get_numx() get_numy()	TBD ecr_config_get() ???	Not documented for Maverick
get_datawidth()	TBD ecr_config_get() ???	Not documented for Maverick
get_x_compress_mask() get_y_compress_mask()	TBD ecr_config_get() ???	Not documented for Maverick
get_data_compress()	TBD ecr_config_get() ???	Not documented for Maverick
linearize_ecr_scram()	TBD ecr_scramble_ram_write() ???	Not documented for Maverick
set_ecr_xscram() set_ecr_yscram()	ecr_xscram_set() ecr_yscram_set() Not documented yet	
error_count_area()	ra_error_count_area_get() Not documented yet Change from ECR to DUT	

The following table lists the ECR functions which are DONE. The table which follows lists the ECR functions which still need work:

ecr_all_clear()	ecr_area_clear()
ecr_any_overflow_get()	ecr_col_ram_read() ecr_col_ram_write()
ecr_column_ram_scan()	ecr_counters_clear()
ecr_compare_reg_set(), ecr_compare_reg_get()	ecr_error_add()
ecr_config_set() ecr_config_get()	ecr_error_counter_set(), ecr_error_counter_get()
ecr_counters_config_set(), ecr_counters_config_get()	ecr_error_delete()
ecr_fast_image_write(), ecr_fast_image_read()	ecr_error_set() ecr_error_get()
ecr_file_image_write(), ecr_file_image_read()	ecr_miniram_write() ecr_miniram_read()
ecr_main_ram_scan()	ecr_rams_clear()
ecr_miniram_config_set(), ecr_miniram_config_get()	ecr_rams_update()
ecr_miniram_scan()	ecr_row_ram_write() ecr_row_ram_read()
ecr_overflow_get()	ecr_scramble_bank_set(), ecr_scramble_bank_get()
ecr_row_ram_scan()	ecr_scramble_ram_write(), ecr_scramble_ram_read()
ecr_write_mode_set(), ecr_write_mode_get()	ecr_x_y_data_set()

3.26 Logic Error Catch (LEC)

See Error Catch RAM (ECR), Error Catch RAM Software.

This section includes the following:

3.26.1 Overview

See Logic Error Catch (LEC).

The Logic Error Catch (LEC) facility uses the optional Error Catch RAM (ECR) hardware to capture pin PASS/FAIL information (and other information, see LEC Counters, LEC Mode and LEC Capture Options) when executing Logic Test Patterns.

To use the LEC requires the steps outlined below. The related code will typically be found in Test Blocks:

1. Using `lec_config_set()`, configure the ECR for use as an LEC, specify which DUT pins to capture, and set the default LEC Mode.

2. Optionally, use `lec_mode_set()` to select a non-default LEC Mode.

Note: some WaveTool options can modify the Logic Error Catch (LEC) configuration set using `lec_config_set()` and/or `lec_mode_set()`. WaveTool does not restore any changes made.

---

3. Execute a functional test (`funtest()`) to log PASS/FAIL pin information to the LEC. Arguments to `funtest()` affect which vectors will log information to the LEC, see LEC Capture Options. Note that passing vectors can be logged in addition to failing vectors.

Note: unlike memory error catch mode, logic capture is not cumulative i.e. each `funtest()` execution clears the LEC.

---

4. View LEC information using LEC Tool. Or...
5. Retrieve information from the LEC using `lec_scan()`. Information must be retrieved before executing another `funtest()` or changing the LEC mode.
6. Use the returned LEC Capture Data.

Details of each step and the associated data structures are documented below. A complete Example is included in the LEC Capture Data section.

3.26.2 LEC Counters

See Logic Error Catch (LEC), Error Catch RAM (ECR).

Several LEC capture options allow LEC counter values to be captured/logged (see `lec_mode_set()`). These counters are targeted at resolving which iteration(s) of a pattern loop were logged.

When the test pattern contains single-instruction loops (i.e. `RPT` or a loop controlled using a VAR counter and an explicit branch instruction) or multi-instruction loops (i.e. `STARTLOOP`/`ENDLOOP` or a loop explicitly controlled using a VAR counter with an explicit branch instruction) the LEC may capture the same VAR value more than once. To determine which execution(s) of a loop were actually logged requires logging the LEC counter associated with the VAR counter used to control the loop.

For example, if a `RPT` 10 instruction is logged using `LEC_only_errors` it is not possible to determine which loop iteration(s) contained errors (and thus were logged) without also

logging LEC counter 4, the LEC counter associated with the VAR counter which controls RPT loops.

As indicated, the four LEC counters operate in conjunction with the four logic pattern VAR counters, but do not contain the same values and do not count the same way. Instead, a given LEC counter is set and modified based on how its corresponding VAR counter is used, as follows:

Table 3.26.2.0-1 LEC Counter Operation

VAR Counter Operation	First Pattern Cycle	nth Pattern Cycle	VAR Counter Used
RPT	Clear	Increment	COUNT4
STARTLOOP[1]	Clear		COUNT1, COUNT2[2], COUNT3[2]
ENDLOOP[3]	Increment		COUNT1, COUNT2[2], COUNT3[2]
COUNTVUDATA	Clear		See Pattern VCOUNT Instruction
DECR	Increment		See Pattern VCOUNT Instruction
DEC2	Increment		See Pattern VCOUNT Instruction
INCR	Increment		See Pattern VCOUNT Instruction
NOCOUNT	Hold		None. Default VCOUNT Instruction

All LEC counters are set = 0 at the start of pattern execution.
Notes:
1) The associated LEC counter is cleared once when a STARTLOOP begins
2) COUNT2 and COUNT3 are used when the pattern contains nested STARTLOOPs. COUNT1 controls the outermost loop, etc.
3) The associated LEC counter is incremented each time pattern execution decrements the VAR counter used to control the loop; i.e. each time the loop repeats.

The following pattern and setup conditions are used to describe the LEC counter operation which follows:

- The following pattern is executed, logging all cycles (LEC_first_vectors, see LEC Capture Options):

```
PATTERN( myPat, logic )
%  VEC        HL10XZV
%  RPT 4      HL10XZV // Controlled using VAR COUNTER 4
%  VEC        HL10XZV
STARTLOOP 3          // Controlled using VAR COUNTER 1
%  VEC        HL10XZV
%  VEC        HL10XZV
ENDLOOP
%  VEC        HL10XZV
   VAR        DONE
```

- Prior to pattern execution, the LEC Mode is set to `t_lec_mode_2` (see `lec_mode_set()`) and LEC counters 4 (`t_lec_vcount4`) and 1 (`t_lec_vcount1`) are selected to be logged:

As indicated, by design, `RPT` instructions are controlled by VAR counter-4. Given the LEC setup noted above the value in LEC counter-4 is captured for each pattern cycle captured. When the LEC captures the VAR for the `RPT` instruction, the captured LEC counter value can be used to determine which loop iteration(s) were logged:

Pattern Relative VAR	LEC Counter 4	LEC Counter 1	Comments
0	0	0	All LEC counters reset at pattern start
1	0	0	LEC Counter 4 reset in 1st `RPT` cycle
1	1	0	LEC Counter 4 incremented in 2nd `RPT` cycle
1	2	0	LEC Counter 4 incremented in 3rd `RPT` cycle
1	3	0	LEC Counter 4 incremented in 4th `RPT` cycle
2	3	0	All LEC counters hold, no VAR counters used
3	3	0	LEC Counter 1 reset in 1st `STARTLOOP` cycle
4	3	0	All LEC counters hold, no VAR counters used
3	3	1	LEC Counter 1 incremented in 2nd `STARTLOOP` cycle
4	3	1	All LEC counters hold, no VAR counters used

Pattern Relative VAR	LEC Counter 4	LEC Counter 1	Comments
3	3	2	LEC Counter 1 incremented in 3rd STARTLOOP cycle
4	3	2	All LEC counters hold, no VAR counters used
5	3	2	All LEC counters hold, no VAR counters used

Note that VAR counters are used implicitly for RPT and STARTLOOP control and thus must not be used explicitly in vectors with those instructions. VAR counter COUNT4 is always used for RPT control. VAR counters COUNT1 through COUNT3 are used for STARTLOOP control, allowing up to three levels of nesting. Since nested STARTLOOPs are rare, it is likely that VAR counters COUNT2 and COUNT3 are available for explicit LEC applications.

3.26.3 VAR/SAR Description

See Logic Error Catch (LEC), Error Catch RAM (ECR).

The Logic Error Catch (LEC) stores (captures, logs, etc.) the VAR and/or SAR for each entry in the LEC.

The term VAR originally referred to Vector Address Register, which stores the address of the logic instruction (vector) being executed (or about to be executed). In this documentation, the term VAR is also used:

- When referring to the hardware engine which controls logic pattern instruction execution; i.e. VAR Engine.
- To represent the pattern instruction which controls logic pattern execution sequence; i.e. VAR instruction. Note that most pure Logic Test Patterns will not contain an explicit VAR instruction because the VEC/RPT instructions implicitly control the VAR Engine.
- The current value in the VAR register. This is the address of the logic pattern instruction being executed.

When using the Logic Error Catch (LEC), the VAR of a logic instruction (vector) may be logged, allowing the user to identify which pattern instructions had information captured in the LEC.

The term SAR refers to Scan Address Register, the address of a scan instruction. Like the VAR, when using the LEC, the SAR of a scan instruction may be logged, allowing the user to identify which scan instructions had information captured in the LEC.

Since both logic instructions and scan instructions are stored in the same physical memory; i.e. the combined Logic Vector Memory (LVM)/Scan Vector Memory (SVM), it is necessary to describe the VAR/SAR numbering system. These will be the values logged to the LEC and retrieved by the user. Note the following:

- Any VAR value captured in the LEC is the absolute address of the associated logic instruction in the LVM. The `addrs()` function may be used to convert this to a pattern relative value, where the first instruction in the pattern = 0.
- When the test pattern contains loops the LEC may capture the same VAR value more than once. To determine which execution(s) of a loop were actually logged requires logging the LEC counter associated with the VAR counter which controls the loop. See LEC Counters and LEC Mode.
- Scan instructions are packed into the LVM/SVM, thus a given VAR will typically store multiple scan instructions. The packing ratio (i.e. the maximum number of SARs stored in each VAR) is set by the number of scan pins defined for the test pattern. See Scan Test Patterns and SCANDEF Compiler Directive. However, the actual packing is also constrained to specific increments of 1, 2, 4, 8, or 16 (and other things, more below). Thus, for example, if the test pattern defines 6 scan pins the pattern compiler generates output for 8 pins (the two undefined pins have no effect except to consume LVM/SVM).
- The reason the previous bullet is important is because the SAR values logged to the LEC are derived from the VAR which stores a given scan instruction and the data width used to store each scan instruction:

  $$SAR = VAR \times \frac{64}{DataWidth} + P$$

  where **P** is the scan vector position in the packed LVM/SVM location (more below). Note that the LEC hardware logs absolute VAR and SAR values, not pattern-relative values. Absolute values can be converted to the more useful pattern-relative values using the `addrs()` function. In the example below the VAR and SAR values are shown as pattern-relative values.
- In Scan instructions, a new LVM instruction is started (and the VAR incremented):
  - Any time a Scan instruction selects a new time-set and/or pin-scramble map.
  - In any instruction containing `VAR DONE`, `RETURN` or `PAUSE`.

  This means that the SAR values captured for a series of sequential scan instructions may not be sequential.

For example:

```
%%SCANDEF p1, p2, p3, p4, p5, p6 // Sets data width = 8
PATTERN( myPat, logic )
% VEC   HL10XZVHL10XZVHL10XZV  // VEC-1:    VAR = 0
% VEC   HL10XZVHL10XZVHL10XZV  // VEC-2:    VAR = 1
% SVEC  HL10XH                 // SVEC-1:   VAR = 2, SAR = 16
% SVEC  HL10XH                 // SVEC-2:   VAR = 2, SAR = 17
% SVEC  HL10XH                 // SVEC-3:   VAR = 2, SAR = 18
% SVEC  HL10XH, TSET2          // SVEC-4:   VAR = 3, SAR = 24
% SVEC  HL10XH, TSET2          // SVEC-5:   VAR = 3, SAR = 25
% SVEC  HL10XH, TSET2          // SVEC-6:   VAR = 3, SAR = 26
% SVEC  HL10XH, TSET2          // SVEC-7:   VAR = 3, SAR = 27
% SVEC  HL10XH, TSET2          // SVEC-8:   VAR = 3, SAR = 28
% SVEC  HL10XH, TSET2          // SVEC-9:   VAR = 3, SAR = 29
% SVEC  HL10XH, TSET2          // SVEC-10:  VAR = 3, SAR = 30
% SVEC  HL10XH, TSET2          // SVEC-11:  VAR = 3, SAR = 31
% SVEC  HL10XH, TSET2          // SVEC-12:  VAR = 4, SAR = 32
% SVEC  HL10XH, TSET2          // SVEC-13:  VAR = 4, SAR = 33
% VEC   HL10XZVHL10XZVHL10XZV  // VEC-3:    VAR = 5
% SVEC  HL10XH                 // SVEC-14:  VAR = 6, SAR = 48
% VEC   HL10XZVHL10XZVHL10XZV  // VEC-4:    VAR = 7
```

As noted above, the VAR and SAR values shown are pattern-relative values whereas the values captured in the LEC will be the absolute VAR/SAR addresses of each instruction in LVM. Use `addrs()` to convert to a pattern-relative value.The following table shows how the LVM/SVM stores the example above and the (pattern relative) VAR/SAR values which are valid and thus will be captured in the LEC given that each SVEC instruction is logged. Scan width = 8:

VAR[1]	Inst.	SAR								Notes
0	VEC									VEC-1
1	VEC									VEC-2
2	SVEC	16	17	18[2]	19	20	21	22	23	SVECs 1-3
3	SVEC	24	25	26	27	28	29	30	31	SVECs 4-11
4	SVEC	32	33[3]	34	35	36	37	38	39	SVECs 12,13
5	VEC									VEC-3

VAR[1]	Inst.	SAR								Notes
6	SVEC	48[4]	48	50	51	52	53	54	55	SVEC 14
7	VEC									VEC-4

Notes:
1) See earlier comment about absolute VAR/SAR values.
2) VAR increments because the time-set changes in next SVEC
3) VAR increments because the next instruction is not SVEC
4) VAR increments because the next instruction is not SVEC
5) Grey cells do not store scan instructions thus SAR will not be logged to the LEC.

As indicated, the improvement in LVM/SVM memory consumption gained by packing scan instructions is best when each sequence of scan instructions completely consumes an entire VAR.

3.26.4 LEC Mode

See Logic Error Catch (LEC), Error Catch RAM (ECR).

The Logic Error Catch (LEC) in Magnum 1/2/2x has four capture modes, which determine the type of information captured, as described in the following table. Executing `lec_config_set()` always sets the default LEC mode. Subsequently, the `lec_mode_set()` function may be used to change the mode. The `lec_mode_get()` function may be used to retrieve the currently set LEC Mode:

LEC Mode	Capture
`t_lec_mode_1`	64 pin P/F states (per ECR), 32 bit VAR, 32 bit SAR. Default.
`t_lec_mode_2`	64 pin P/F states (per ECR), 28 bit VAR or SAR, 1 bit VAR/SAR flag, one 32 bit counter value, one 18 bit counter value.
`t_lec_mode_3`	64 pin P/F states (per ECR), 28 bit VAR or SAR; 1 bit VAR/SAR flag, four 12 bit counter values.
`t_lec_mode_4`	64 pin P/F states (per ECR), 28 bit VAR, 28 bit SAR, one 20 bit counter value.
VAR = Vector Address. SAR = Scan Vector Address. See VAR/SAR Description.	

Note the following:

- See LEC Capture Data for a more detailed explanation of each mode.
- All LECs are configured the same, including in multi-site program configurations.
- The LEC Mode must be set prior to executing a functional test which is to log data to the LEC using that mode.
- The contents of the LEC should be considered as undefined after changing the LEC configuration (`lec_config_set()`) or LEC Mode (`lec_mode_set()`).
- Arguments to `funtest()` determine which vectors will log information to the LEC (see LEC Capture Options).
- Unlike in Maverick, the LEC Mode does not affect the information read by `errmar()`, `errvar()`, and `errsar()`.

Note: some WaveTool options can modify the Logic Error Catch (LEC) configuration, including the LEC mode. WaveTool does not restore any changes made.

3.26.5 LEC Capture Options

See Logic Error Catch (LEC), Error Catch RAM (ECR).

When using the Logic Error Catch (LEC), two controls are provided which affect the information logged:

- The LEC Mode selects which parameters are logged, which can be various combinations of VAR, SAR, and LEC Counters.
- The `PatStopCond` argument passed to `funtest()` when executing the test pattern. This determines which test vectors are logged. In some applications, only failing vectors are desired; however, the LEC design supports logging passing vectors also.

The rest of this section refers to `PatStopCond`.

Note: the `PatStopCond` values documented here are limited to those used for LEC applications. See `funtest()` for other `PatStopCond` options.

Usage

```
PFState funtest( Pattern *pPattern, PatStopCond Condition );
```

where:

pPattern is a pointer to the pattern to be executed. See funtest().

Condition is one of the PatStopCond values from the following table:

Table 3.26.5.0-1 funtest() Logic Error Capture Options

Option	Purpose
LEC_first_vectors	Capture the first 2Meg (2^{21}-6) vectors executed. Ignores Pass/Fail.
LEC_last_vectors	Capture the last 2Meg (2^{21}-6) vectors executed. Ignores Pass/Fail.
LEC_before_error	Capture the first failing vector plus the previous 2Meg (2^{21}-6) vectors executed.
LEC_after_error	Capture the first failing vector plus the next 2Meg (2^{21}-6) vectors executed.
LEC_only_errors	Capture the first 2Meg (2^{21}-6) failing vectors.
LEC_center_error	Capture the first failing vector plus up to 512K vectors executed before the failure and up to 512K vectors executed after the failure.

3.26.6 DDR LEC Operation

See Logic Error Catch (LEC), Error Catch RAM (ECR).

The Logic Error Catch (LEC) can be used with Double Data Rate (DDR) Mode patterns, with the following constraints:

- The LEC must be explicitly configured for DDR operation using the t_double option. For example:

  ```
  lec_config_set( myPins, t_double );
  ```

- Up to 32 pins per sub-site can be captured. This is described in the DDR Fail Signal MUX: Logic Error Catch section.

- `fail_signal_mux()` must be executed to complete the hardware configuration before executing the test being logged to the LEC.
- In DDR mode, one LEC entry (address) stores the captured pin data for both the A and B cycles. See LEC Capture Data.This effectively doubles the size of the LEC for DDR patterns.
- When logging errors (`LEC_only_errors`, `LEC_center_error`, etc.) both the A and B DDR cycles will be logged for each error, even of one of the DDR cycles has no failing strobes.

3.26.7 Types, Enums, etc.

See Logic Error Catch (LEC), Error Catch RAM (ECR).

Description

The following enumerated types are used in support of the various Logic Error Catch (LEC), functions:

Usage

The `FailMuxSelectOpt` enumerated type is used to configure the DDR Fail Signal MUX. See `fail_signal_mux()`:

```
enum FailMuxSelectOpt { t_single, t_double, t_mux_opt_na };
```

The `LecMode` enumerated type is used to set and get the LEC Mode:

```
enum LecMode { t_lec_mode_1,
               t_lec_mode_2,
               t_lec_mode_3,
               t_lec_mode_4,
               t_lec_mode_na };
```

The `LecCounter` enumerated type is used to specify which VAR counter(s) are to be logged to the LEC. See LEC Mode and `lec_mode_set()`:

```
enum LecCounter { t_lec_vcount1,
                  t_lec_vcount2,
                  t_lec_vcount3,
                  t_lec_vcount4 };
```

The LecEntry and LecEntryArray data structures are used to return information read from the LEC (see lec_scan()). Each of the LEC Modes uses different subsets of data in this structure; i.e. not all fields will contain valid data in a given mode. Fields that are not used in a given mode will be set = -1 (0xFFFFFFFF):

```
typedef struct LecEntryStruct {
 unsigned __int64 data;
 DWORD var;              // VAR value for this entry
 DWORD sar;              // SAR value for this entry
 DWORD counter1;         // Value in VCOUNT1 for this entry
 DWORD counter2;         // Value in VCOUNT2 for this entry
 DWORD counter3;         // Value in VCOUNT3 for this entry
 DWORD counter4;         // Value in VCOUNT4 for this entry
 unsigned char var_sar_id; // 0 = VAR is valid. 1 = SAR is valid
} LecEntry;

typedef CArray< LecEntry, LecEntry & > LecEntryArray;
```

3.26.8 lec_config_set()

See Logic Error Catch (LEC), Error Catch RAM Software, Error Catch RAM (ECR).

Description

The lec_config_set() function is used configure the LEC prior to use. Note the following:

- Executing lec_config_set() puts the Error Catch RAM (ECR) into LEC mode. To restore the ECR for use with memory patterns use ecr_config_set().
- The datapins argument selects which pin(s) are to be captured in the LEC. If the pins in datapins span sites (i.e. Sites-per-Controller > 1) the system software will create a separate pin list per-site, and configure the LEC for each site appropriately.
- The option argument allows the user to specify whether the LEC should be configured for DDR LEC Operation. By default (i.e. non-DDR mode) up to 128 pins (64 per sub-site) can be captured to the LEC. In DDR mode 64 pins (32 per sub-site) can be captured. Read the DDR LEC Operation; important details are documented there.

- Executing `lec_config_set()` always sets the LEC Mode to `t_lec_mode_1`. The `lec_mode_set()` function may subsequently be used to change the mode, which alters the type of information captured in the LEC.
- `lec_config_set()` does not clear the LEC but any previously captured information should be considered invalid.

Note: some WaveTool options can modify the Logic Error Catch (LEC) configuration. WaveTool does not restore any changes made.

Usage

```
void lec_config_set(
            PinList* datapins,
            FailMuxSelectOpt option DEFAULT_VALUE( t_single ) );
```

where:

`datapins` identifies the pin(s) to be captured in the LEC. Only signal pins are legal. The order pins appear in the pin list determines the order of data returned in the `entries` parameter passed to `lec_scan()`. The first pin in **`datapins`** will be the LSB of the returned data, etc. See LEC Capture Data. Pins from multiple sites are allowed if Sites-per-Controller > 1 (see Description).

`option` is optional and, if specified, determines whether the LEC is configured for DDR LEC Operation. Default = `t_single` = non-DDR. The user is responsible for correctly setting the DDR mode (`t_double`) if/when a DDR test pattern is to log errors to the LEC. Additional steps are normally required to log errors from DDR patterns, see DDR LEC Operation.

Example

See Example.

3.26.9 lec_config_get()

See Logic Error Catch (LEC), Error Catch RAM Software, Error Catch RAM (ECR).

Description

The `lec_config_get()` function is used to retrieve the values last set using `lec_config_set()`. Two versions are supplied:

- If Sites-per-Controller = 1 both versions will return identical values.
- If Sites-per-Controller > 1 the returned `option` value will be identical for all `boards` but the `datapins` returned may be different based on which version of `lec_config_get()` is executed. See Usage.

Each `lec_config_get()` argument defaults to a NULL pointer, which must be replaced with a pointer to an existing variable of the appropriate type to obtain the desired value. Any don't-care values can remain NULL (0).

Note: beginning in software release h3.3.xx, the values returned by `lec_config_get()` are valid only if the ECR is currently configured as an Logic Error Catch (LEC) (i.e. not as an ECR). See `lec_configured_get()` and `ecr_configured_get()`.

Usage

The following version should be used to retrieve the original pin list passed to `lec_config_set()`:

```
void lec_config_get(PinList** datapins DEFAULT_VALUE(0),
                    FailMuxSelectOpt *option DEFAULT_VALUE(0) );
```

The following version should be used to retrieve pins for a specific board. When Sites-per-Controller = 1 both functions return the same pin list:

```
void lec_config_get(HSBBoard board,
                    PinList** datapins DEFAULT_VALUE(0),
                    FailMuxSelectOpt *option DEFAULT_VALUE(0) );
```

where:

`datapins` is optional. If not 0 (NULL) **`datapins`** must be a pointer to an existing `PinList*` used to return the pins being logged to the LEC.

`option` is optional. If not 0 (NULL) **`option`** must be a pointer to an existing `FailMuxSelectOpt` variable used to return the `option` argument passed to `lec_config_set()`.

Example

See Example.

3.26.10 lec_configured_get()

See Logic Error Catch (LEC), Error Catch RAM Software, Error Catch RAM (ECR).

Note: first available in software release h3.3.xx.

Description

The `lec_configured_get()` function may be used to determine if the Error Catch RAM (ECR) is currently configured for Logic Error Catch (LEC) use.

In this context, `lec_configured_get()` returns TRUE if the ECR is currently configured for LEC use, but returns FALSE if the ECR is not configured or if the ECR is currently configured for ECR use (using `ecr_config_set()`).

Similarly, the `ecr_configured_get()` function returns TRUE if the ECR is currently configured for ECR use, but returns FALSE if the ECR not configured, or if ECR is currently configured for LEC use.

In Multi-DUT Test Programs, the Active DUTs Set (ADS) and Ignored DUTs Set (IDS) have no effect on `lec_configured_get()`.

Usage

```
BOOL lec_configured_get();
BOOL lec_configured_get( HSBBoard board );
```

where:

`board` is used when Sites-per-Controller > 1 to identify a target site assembly (`HSBBoard`).

`lec_configured_get()` returns TRUE if the ECR is currently configured for LEC use, but returns FALSE if the ECR is not configured or if the ECR is currently configured for ECR use (using `ecr_config_set()`).

Example

```
BOOL lec_configured = lec_configured_get();
BOOL lec_hsb1_configured = lec_configured_get( t_hsb1 );
```

3.26.11 lec_mode_set(), lec_mode_get()

See Logic Error Catch (LEC), Error Catch RAM Software, Error Catch RAM (ECR).

Description

The `lec_mode_set()` function is used to change the LEC Mode from the default mode (`t_lec_mode_1`) set each time `lec_config_set()` is executed. In some modes this determines which LEC counter(s) are captured.

The `lec_mode_get()` function is used to get the current LEC Mode and, in some modes, which LEC counter(s) are currently being captured.

Note the following:

- `lec_config_set()` must be executed at least once, to put the ECR into LEC mode. Then, `lec_mode_set()` can be executed zero or more times to change the LEC mode, as desired. Executing `lec_config_set()` always sets the LEC mode to `t_lec_mode_1`.
- `lec_config_set()` must be executed to set the LEC Mode before executing a test pattern which is to log values to the LEC using that mode.
- `lec_mode_set()` sets the LEC Mode for all sites; i.e. in Multi-DUT Test Programs the Active DUTs Set (ADS) and Ignored DUTs Set (IDS) have no effect on `lec_mode_set()`.
- Using `lec_mode_set()`, the `counter1` argument is only used in LEC Modes `t_lec_mode_2` and `t_lec_mode_4`, to specify the first LEC counter to be logged to the LEC. The `counter2` argument is only used in LEC Mode `t_lec_mode_2` to specify the second LEC counter to be logged to the LEC. See LEC Counters.
- Using `lec_mode_get()`, the `counter1` argument only returns a valid value in LEC Modes `t_lec_mode_2` and `t_lec_mode_4`. The `counter2` argument only returns a valid value in LEC Mode `t_lec_mode_2`.
- Using `lec_mode_set()`, in LEC Mode `t_lec_mode_2` the two LEC counters specified using `counter1` and `counter2` must be different counters. A warning is issued if this rule is violated.

- Executing `lec_mode_set()` does not clear the LEC, but the contents are undefined (not valid if scanned).
- The `t_lec_mode_na` may not be specified using `lec_mode_set()`. To change the LEC back to ECR operation use `ecr_config_set()`.

A brief example which logs a LEC counter is shown in VAR/SAR Description.

Note: some WaveTool options can modify the Logic Error Catch (LEC) configuration, including the LEC mode. WaveTool does not restore any changes made.

`lec_mode_set()` may be used with `ecr_configured_get()` to determine whether the Error Catch RAM (ECR) is configured for ECR use vs. LEC use.

Usage

```
void lec_mode_set(
            LecMode mode,
            LecCounter counter1 DEFAULT_VALUE( t_lec_vcount1 ),
            LecCounter counter2 DEFAULT_VALUE( t_lec_vcount4 ) );
LecMode lec_mode_get(  LecCounter *counter1 DEFAULT_VALUE( 0 ),
                       LecCounter *counter2 DEFAULT_VALUE( 0 ) );
```

where:

mode specifies the desired LEC Mode.

counter1 and **counter2** are used in two contexts:

- Using `lec_mode_set`, **counter1** and **counter2** are both optional and, if used, select which LEC counter is to be logged. See Description, LEC Counters and LEC Mode. Default for **counter1** = `t_lec_vcount1` (STARTLOOP). Default for **counter2** = `t_lec_vcount4` (RPT).
- Using `lec_mode_get`, **counter1** and **counter2** are optional and, if used, are pointers to existing `LecCounter` variables used to return which LEC counters are currently being logged. See Description, LEC Counters and LEC Mode.

`lec_mode_get()` returns the currently set LEC Mode. `t_lec_mode_na` is returned if the ECR is not currently configured for LEC use (see Overview, `lec_config_set()`).

Example

```
lec_config_set( myPins ); // lec_config_set()
lec_mode_set( t_lec_mode_4, t_lec_vcount2);
```

```
LecCounter c1, c2;
LecMode mode = lec_mode_get( &c1, &c2 );
if( mode == t_lec_mode_2)
   // Use c1 and c2
else if( mode == t_lec_mode_4)
   // Use c1 only
```

Also see Example.

3.26.12 lec_scan()

See Logic Error Catch (LEC), Error Catch RAM Software, Error Catch RAM (ECR).

Description

The `lec_scan()` function is used to retrieve information from the Logic Error Catch (LEC).

Note the following:

- Two versions of `lec_scan()` are provided. The version to be used is determined by the value set for Sites-per-Controller. See Usage.
- When the ECR is configured for LEC use, the LEC content is over-written each time a test pattern is executed; i.e. it is not accumulated (unlike memory ECR use). The user must retrieve any desired LEC information before executing another test pattern.
- `lec_scan()` returns information retrieved from the LEC via the `entries` argument. See Usage and LEC Capture Data.

Usage

The following function should be used when Sites-per-Controller = 1. In Multi-DUT Test Programs values are returned for the first DUT in the Active DUTs Set (ADS). In non-Multi-DUT Test Programs values are returned for the pins logged:

```
int lec_scan(  int start,
               int stop,
               __int64 mask,
               int max,
               LecEntryArray &entries );
```

The following function should be used when Sites-per-Controller > 1. In Multi-DUT Test Programs values are returned for the first DUT in the Active DUTs Set (ADS) from the LEC

on the specified `board`. In non-Multi-DUT Test Programs values are returned for the pins logged on the specified `board`:

```
int lec_scan(  HSBBoard board,
               int start,
               int stop,
               __int64 mask,
               int max,
               LecEntryArray & entries );
```

where:

`board` is used only when Sites-per-Controller > 1, to identify which Site Assembly Board (HSB) is to be accessed.

`start` and **`stop`** identify the first and last LEC location to be read. Legal values are 0 to 2^{21}-6. Only the number of entries actually captured will be returned and the system software will set **`stop`** to the maximum value if the user specifies a value too large. The **`max`** parameter (below) will also limit the number of values returned. However...

Note: `lec_scan()` performance is proportional to the number of addresses read, set using **`start`**/**`stop`** and **`max`** (below). The time needed to retrieve ^{21}-6 entries can be substantial (minutes).

`mask` determines which pin(s) are to have values read from the LEC. This is a bit-wise mask where a logic-1 enables values to be read for the corresponding pin and logic-0 disables values to be read for the corresponding pin. Mask bits must be ordered to match the DUT pin order in the pin list passed to `lec_config_set()` with the LSB **`mask`** bit = first pin in the pin list, etc. `lec_config_get()` can be used to retrieve the pins being logged.

`max` specifies the maximum number of **`entries`** to return from the LEC. This will set the upper size limit of the returned **`entries`** array. See Note above.

`entries` is a pointer to an existing `LecEntryArray`, used to return the information retrieved from the LEC. The array is automatically resized by the system software and any prior contents are lost. The standard `CArray` member functions may be used to access the array. See LEC Capture Data.

`lec_scan()` returns the number of elements in the **`entries`** array.

Example

```
LecEntryArray lec_data;
int num_entries = lec_scan( 0, 0x7fff, 0xffff, 0x8000, lec_data );
```

Also see Example.

3.26.13 LEC Capture Data

See Logic Error Catch (LEC), Error Catch RAM Software, Error Catch RAM (ECR).

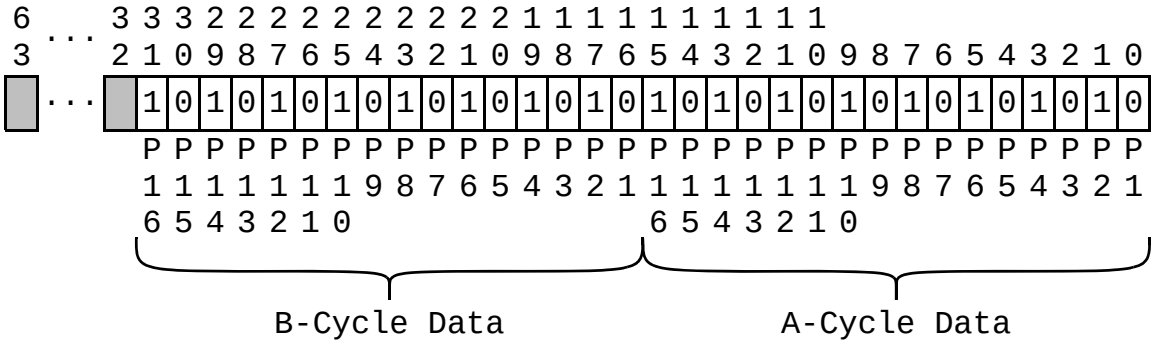
The `lec_scan()` function is used to retrieve information from the LEC. The information is returned in an `LecEntryArray`, which is an array of `LecEntry` structs. These structures are shown below:

```
typedef struct LecEntryStruct {
 unsigned __int64 data;
 DWORD var;              // VAR value for this entry
 DWORD sar;              // SAR value for this entry
 DWORD counter1;         // Value in VCOUNT1 for this entry
 DWORD counter2;         // Value in VCOUNT2 for this entry
 DWORD counter3;         // Value in VCOUNT3 for this entry
 DWORD counter4;         // Value in VCOUNT4 for this entry
 unsigned char var_sar_id; // 0 = VAR is valid. 1 = SAR is valid
} LecEntry;

typedef CArray< LecEntry, LecEntry & > LecEntryArray;
```

Note the following:

- `lec_scan()` automatically resizes the passed `LecEntryArray` array to contain the number of elements (addresses) read from the LEC. Any prior array contents are lost. The standard `CArray` member functions can be used to access the array.
- Each `LecEntry` element in the array contains the information read from one LEC address. In Multi-DUT Test Programs values are returned for the first DUT in the Active DUTs Set (ADS).
- The 64-bit `data` field captures the Pass/Fail results for up to 64 pins (32 pins in DDR mode). The order of bits in this field matches the order of pins in the `datapins` pin list argument passed to `lec_config_set()`, with the LSB = the first pin in `datapins`, etc.
- In Double Data Rate (DDR) Mode (see DDR LEC Operation), each LEC entry (address) stores the captured pin data for both the A and B cycles. Thus, the `data` field returns the pin status information for up to 32 pins, for both the A and B

cycles stored in a given physical LEC address. The pin information is packed such that the A cycle data is stored in the LSB bits, with the B cycle data immediately above it. For example, given a 16-pin pin list is passed to `lec_config_set()`:



- The LEC Mode in effect at the time the LEC captures data determines the type of information logged to the LEC. The `LecEntry` structure is designed to accommodate all possible LEC data, even though not all of the parameters will contain valid information.
- Several LEC Modes allow capture of one or more VAR counter values. Arguments to `lec_mode_set()` specify which one or two of the 4 possible VAR counter(s) are to be logged to the LEC using t_lec_mode_2 or t_lec_mode_4. The `LecEntry` structure has separate fields used to store values for each counter. Thus, for example, if VAR counter-3 is logged to the LEC, the `LecEntry counter3` field will contain the value read from the LEC for that counter.

- The following table indicates which fields are valid for each LEC Mode:

Mode	Valid Fields	Comments
t_lec_mode_1	data var sar	Pass/Fail for 64 pins 32-bit VAR 32-bit SAR
t_lec_mode_2	data var or sar var_sar_id counter1 counter2 counter3 counter4	Pass/Fail for 64 pins 28-bit VAR or SAR 0 = VAR, 1 = SAR Select any two of four LEC counters First counter, 32 bits logged Second counter, 18 bits logged
t_lec_mode_3	data var or sar var_sar_id counter1 counter2 counter3 counter4	Pass/Fail for 64 pins 28-bit VAR or SAR 0 = VAR, 1 = SAR All four LEC counters are logged 12 bits logged
t_lec_mode_4	data var sar counter1 counter2 counter3 counter4	Pass/Fail for 64 pins 28-bit VAR 28-bit SAR Select any one of four LEC counters 20 bits logged

Note the following:

- The VAR value captured in the LEC is the absolute address of the instruction in the LVM. The addrs() function should be used to convert this to a pattern relative value, where the first instruction in the pattern = 0. In DDR mode, a given VAR stores both the A and B cycle information (see DDR LEC Operation).
- In t_lec_mode_1 mode the VAR is a 32-bit value (4,294,967,296 max.). In the other three modes the VAR is a 28-bit value (268,435,456 max.). Both exceed the maximum size of the LVM by a substantial margin.

- The SAR value captured in the LEC is derived from the VAR which stores a given scan instruction and the data width used to store each scan instruction:

$$SAR = VAR \times \frac{64}{DataWidth} + P$$

  where **P** is the scan vector position in the packed LVM/SVM location (see VAR/SAR Description). Note that the LEC hardware logs absolute VAR and SAR values, not pattern-relative values. Absolute values can be converted to the more useful pattern-relative values using the `addrs()` function.
- In `t_lec_mode_1` mode the SAR is a 32-bit value (4,294,967,296 max.). In the other three modes the SAR is a 28-bit value (268,435,456 max.). It is possible, in programs which mostly fill the LVM/SVM with scan patterns, for the maximum SAR value to exceed the number of bits available to record it. When this occurs, the SAR value will roll-over (and there is no way to know this has occurred).
- In `t_lec_mode_1` mode and `t_lec_mode_4` mode both the VAR and SAR are always captured.
- In `t_lec_mode_2` mode and `t_lec_mode_3` mode, if the captured instruction is a scan vector the SAR will be captured and the value of `var_sar_id` will = 1, otherwise the VAR will be captured and the value of `var_sar_id` will = 0.

Note: a given scan instruction controls the pins defined to receive scan data. See Scan Test Patterns and SCANDEF Compiler Directive. However, non-scan pins can still be logged if they are included in the pin list passed to `lec_config_set()`.

- In `t_lec_mode_1` mode no VAR counters are captured.
- In `t_lec_mode_2` mode one or two VAR counters are captured, as specified using `lec_mode_set()` prior to pattern execution. The `counter1` argument to `lec_mode_set()` determines which counter is logged as a 32-bit value. The `counter2` argument to `lec_mode_set()` determines which counter is logged as a 18-bit value. In either case, user code must retrieve values from the `LecEntry` field based on the specific counter being logged. For example, if `t_lec_vcount4` is specified as either counter argument to `lec_mode_set()` the `counter4` field of the `LecEntry` struct will be used to return the value logged in the LEC.
- In `t_lec_mode_3` mode all four VAR counters are captured, as 12-bit values. User code will retrieve the value of VAR counter-3 from the `counter3` field of the `LecEntry` struct, etc.

- In t_lec_mode_4 mode one VAR counter is captured, as specified using the counter1 argument to lec_mode_set() prior to pattern execution. User code must retrieve values from the LecEntry field based on the specific counter being logged.
- When the value in a VAR counter exceeds the number of bits captured by the LEC the upper bits are not captured. Thus, for example, in t_lec_mode_3, when the VAR counters are being captured as 12-bit values (i.e. 4096 max.) it will not be possible to distinguish the value 12 from 4108 (4096+12).

Example

The following example uses many of the LEC functions documented here. A supporting function (myPrintLECcount) follows:

```
lec_config_set( datapins ); // lec_config_set()
lec_mode_set( t_lec_mode_2, t_lec_vcount4, t_lec_vcount2 );
PFState result = funtest( myPat, LEC_only_errors ); // funtest()
if( result == FAIL) {
   LecEntryArray errs;
   LecCounter ctr1, ctr2;
   LecModemode = lec_mode_get( &ctr1, &ctr2 );
   { // ActiveDutIterator scope
   ActiveDutIterator duts;
   while ( duts.More() ) { // For each DUT
      output("DUT-%d", active_dut_get() );
      int count = lec_scan( 0, 0x1FFF, 0x3F, 0x2000, errs );
      for( int i = 0; i < count; ++i ) { // For each LEC entry
         if( errs[i].var_sar_id == 0 ) // t_lec_mode_2
            output(" VAR => %d", errs[i].var );
         else
            output(" SAR => %d", errs[i].sar );
         // Print LEC counter values, if appropriate
         switch( mode ) {
         case t_lec_mode_2 :
            myPrintLECcount( ctr1, &errs[i] ); // See below
            myPrintLECcount( ctr2, &errs[i] );
         break;
         case t_lec_mode_3 :
            myPrintLECcount( t_lec_vcount1, &errs[i] );
            myPrintLECcount( t_lec_vcount2, &errs[i] );
            myPrintLECcount( t_lec_vcount3, &errs[i] );
```

```
            myPrintLECcount( t_lec_vcount4, &errs[i] );
          break;
          case t_lec_mode_4 :
            myPrintLECcount( ctr1, &errs[i] );
          }
          PinList* pins;
          FailMuxSelectOpt opt;
          lec_config_get( &pins, &opt );
          unsigned __int64 data = errs[i].data;

          DutPin* dp;
          if( opt == t_single) { // Not DDR
            for( int p = 0; pin_info( pins, p, &dp ); ++p ) {
              output( "%s => %d", resource_name( dp ), data & 1);
              data >>= 1;
            }
          }
          else { // t_double: DDR, need A/B cycle outer loop
            for( int cycle = 0; cycle <= 1; ++cycle ) {
              if( cycle == 0 )
                output(" A Cycle");
              else
                output(" B Cycle");
              for( int p = 0; pin_info( pins, p, &dp ); ++p ) {
          output( "    %s => %d", resource_name( dp ), data & 1 );
                data >>= 1;
              }
            }
          }
        } // for each LEC entry
      } // while each DUT
      } // ActiveDutIterator scope
    } // if result
    void myPrintLECcount( LecCounter c, const LecEntry *e ) {
      switch( c ){
      case t_lec_vcount1 :
        output(" LEC Counter-1 => %d", e->counter1 );
        break;
      case t_lec_vcount2 :
        output(" LEC Counter-2 => %d", e->counter2 );
```

```
      break;
   case t_lec_vcount3 :
      output(" LEC Counter-3 => %d", e->counter3 );
      break;
   case t_lec_vcount4 :
      output(" LEC Counter-4 => %d", e->counter4 );
      break;
   }
}
```

3.26.14 Magnum 1/2/2x vs. Maverick-I/-II LEC Software Compatibility

See Logic Error Catch (LEC), Error Catch RAM Software, Error Catch RAM (ECR).

The Magnum 1/2/2x Error Catch RAM (ECR) architecture is a super-set of the ECR available using Maverick-I and Maverick-II. Rather than attempt to adapt the legacy ECR related functions to the Magnum 1/2/2x ECR architecture, a new set of functions was implemented. This includes the functions which support the Logic Error Catch (LEC).

There are only three related LEC-specific functions, which operate as noted below when executed on a Magnum 1/2/2x:

- `lvm_error_mode()` - does nothing. Replaced by `lec_config_set()`.
- `ConfigureLEC()` - calls `lec_config_set()` with the same arguments. However, it is recommended that the user migrate to the Magnum 1/2/2x methods; i.e. use `lec_config_set()`.
- `LEC_scan()` - the function name is the same for Maverick-I/-II and Magnum 1/2/2x however the arguments are different. Operation of the Maverick-I/-II version on Magnum 1/2/2x is not defined, and vice-versa. The user must modify the arguments before desired operation will occur.

3.27 Waveform Functions

Note: the Waveform functions and related information documented here were added to the Magnum 1 software beginning in software release h2.2.7/h1.2.7. These functions were developed for the Lightning Test System, which contains waveform generation and capture hardware not available using Magnum 1/2/2x, thus, Magnum 1/2/2x applications must synthesize waveform contents using other methods. References to Lightning hardware were removed from this section, to reduce confusion. Refer to the *Lightning Programmer's Manual* for more information.

Note: many of the Waveform functions use parameter values specified in MKS Units. In most cases, this will just work. However, the user should read and understand the requirements for migrating test programs which don't currently use MKS Units to use MKS Units.

The `Waveform*` container is used to store information, such as waveforms, measured DC values, histograms, FFT data, etc. The software documented in this section addresses how waveforms are created, defined, stored and loaded to/from disk, etc.

- Waveform Overview, including Waveform* Attributes and Waveform Terminology
- Waveform Mathematical View
- Decibel (dB)
- Waveform File Formats
- Types, Enums, etc.
- Waveform Sample Value Notations
- Waveform Units
- Waveform Macros
- `waveform_create(), waveform_destroy()`
- `waveform_invalidate()`
- Waveform Generate Functions
  - `waveform_generate_triangle_wave()`
  - `waveform_generate_sine_wave()`
  - `waveform_generate_ramp()`

- waveform_generate_square_wave()
- waveform_generate_gaussian_noise()
- waveform_generate_white_noise()
- waveform_generate_periodic_white_noise()
- waveform_generate_periodic_pink_noise()
- waveform_generate_DC()
- waveform_constant_fill()
- waveform_randomize(), waveform_reset_random_seed()

- Waveform File Write/Read Functions
- waveform_fetch(), waveform_send()
- Waveform Name, Date, Type and Version Information
  - waveform_dump()
  - waveform_get_date(), waveform_set_date()
  - waveform_get_name()
  - waveform_get_typename()
  - waveform_get_version()
  - Waveform Set/Get X/Y Units Functions
  - reciprocal()
- Waveform Sample Programming
  - waveform_set_rrect(), waveform_get_rrect()
  - waveform_set_rlong(), waveform_get_rlong()
  - waveform_set_crect(), waveform_get_crect()
  - waveform_set_polar(), waveform_get_polar()
  - waveform_get_x_start()
  - waveform_get_x_increment()
  - waveform_set_x_scale()
  - waveform_get_size()
  - waveform_get_element(), waveform_set_element()
  - waveform_set_signal_spread(), waveform_get_signal_spread()
  - waveform_zero_pad()
- Waveform Manipulation Functions
- Waveform Equality Functions
- Waveform Conversion Functions
- Waveform Boolean Functions
- Waveform Bitwise Functions
- Waveform Logrithmic Functions
- Waveform Window Functions

- Waveform Convolution/Corrleation Functions
- Waveform Wierd Functions
- Waveform Compression Functions
- Waveform FFT Functions
- Waveform Analysis Functions including INL & DNL Functions

3.27.1 Waveform Overview

See Waveform Functions, Waveform Mathematical View.

Note that a formal mathematical description of waveforms is described in Waveform Mathematical View.

User code must create waveforms. Various functions documented in this section are used to define a waveforms's properties. Other functions save a waveform to disk for subsequent reuse. The attributes of waveforms can be defined within the the executing test program, or, waveforms can be created off-line, each stored in a separate ASCII file, and loaded by the test program. Any defined waveform can be stored on disk and reused (loaded) as needed.

3.27.2 Waveform* Attributes

See Waveform Functions, Waveform Mathematical View.

In software, each waveform object is represented as a Waveform*, which can be created using the Waveform Macros or waveform_create(). The waveform_destroy() function can be used to destroy an existing Waveform*.

The following attributes are defined for Waveform*. This list does not include the waveform's sample values:

Table 3.27.2.0-1 Waveform* Attributes

Attribute	Default Value	Comments
Type	BAD_WAVE	Identifies the notation used to store the waveform's sample values. See Waveform Sample Value Notations. The terms waveform *type* and *sample value* notation are used interchangably in this document.
Size	0	Number of sample values. For two-part notations, size will be indentical for both parts. See waveform_get_size().
X_start	0.0	Identifies the first value in the X axis, in X_units. See waveform_get_x_start() and waveform_set_x_scale(). Note that a waveform's Y axis values are the discrete sample values, described in Y_units.
X_increment	1.0	Identifies the resolution of X axis values, in X_units. This is often called sample frequency or sample period. See waveform_get_x_increment().
X_units	SCALE_NOXUNITS	The base units of X axis values. See Waveform Units.
Y_units	SCALE_NOYUNITS	The base units of Y axis values. See Waveform Units.
The functions noted in this table are included for reference only. A waveform's attributes are accessed and/or modified in many ways, not limited to just the functions noted here.		

Table 3.27.2.0-1 Waveform* Attributes *(Continued)*

Attribute	Default Value	Comments
Signal spread	0	Very specialized. Refers to how windowing spreads a waveform's sample values. See Waveform Window Functions.
Odd flag	FALSE	Very specialized. Identifies whether the number of sample values in the input waveform to real inverse FFT is even or odd. See waveform_real_ifft_even(), waveform_real_ifft_odd().
Date	n/a	Identifies the waveform's creation date. See waveform_get_date(), waveform_set_date().
The functions noted in this table are included for reference only. A waveform's attributes are accessed and/or modified in many ways, not limited to just the functions noted here.		

3.27.3 Waveform Terminology

The diagram below shows how various terminology is used in the context of an AC waveform:

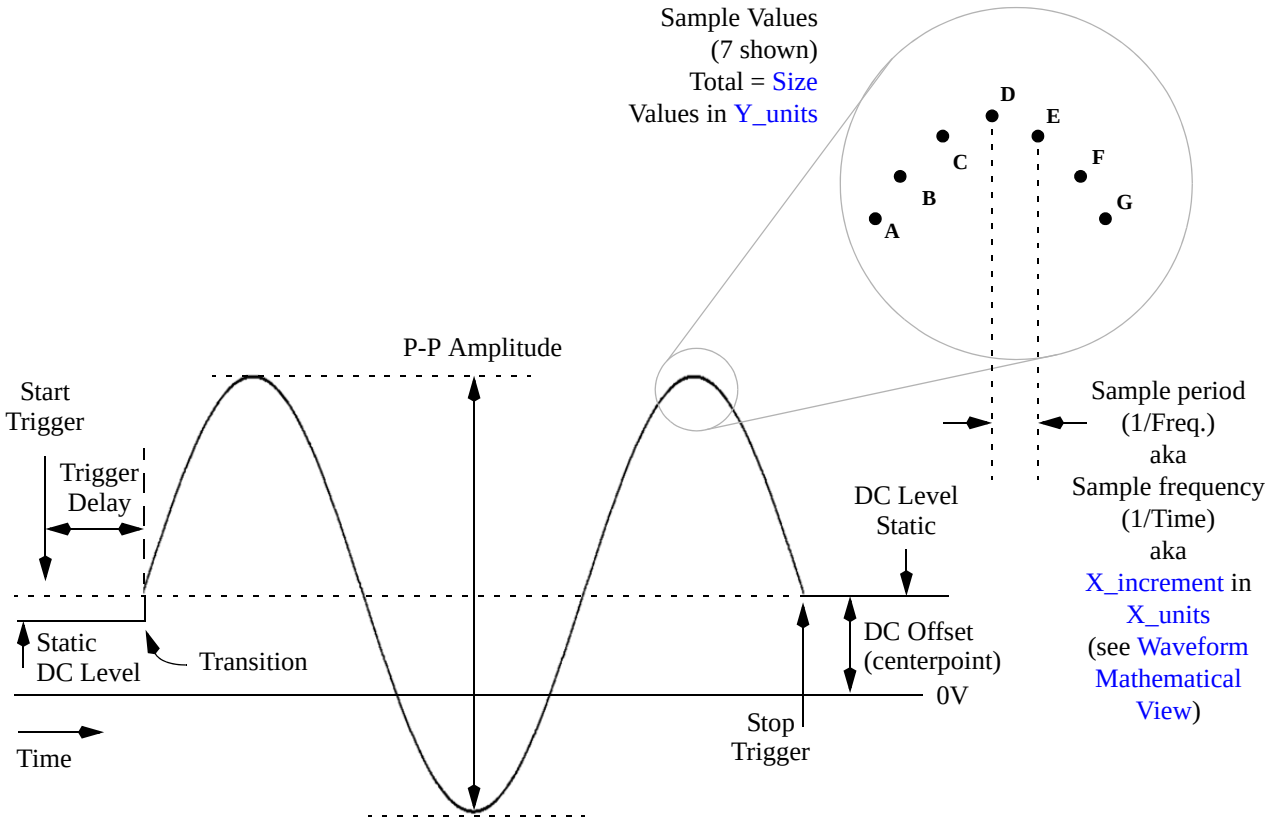


Figure-55: AC Waveform Terminology

Note the following:

- Each waveform consists of one or more digital sample values.
- Waveform sample values can be represented in several ways (see Waveform Sample Value Notations):
  - RRECT_WAVE: real rectangular notation
  - RLONG_WAVE: long real rectangular notation
  - CRECT_WAVE: complex rectangular notation
  - POLAR_WAVE: complex polar notation

The waveform in Figure-55: is a sine wave. In the magnified portion of the waveform note the 7 sample values shown, labeled A through G. Given a 1000 sample waveform (Size = 1000), the corresponding sample values (in RRECT_WAVE notation) for these might be:

A	0.98228725072868872
B	0.99211470131447788
C	0.99802672842827156
D	1.00000000000000000
E	0.99802672842827156
F	0.99211470131447776
G	0.98228725072868850

- AC waveforms have predefined X and Y unit types. X_units, represents time (SCALE_SECONDS), Y_units represent voltage (SCALE_VOLTS). Other waveform applications will utilize other units types. See Waveform Units.

3.27.4 Waveform Mathematical View

See Waveform Functions, Waveform Overview.

Earlier paragraphs describe waveforms in the context of test applications. However, there is also a mathematical view, which becomes significant when analyzing or manipulating waveforms.

- A waveform contains discrete values of a function, where the term function refers to the formal mathematical definition.
- Waveforms are always two dimensional, storing Y data as a function of X. For each X value, there can only be one Y value. The Y values must be specified for a finite and evenly spaced span of X values. Generic X and Y data pairs, sometimes referred to as XY data, are not supported.
- Waveforms restrict X values to be *real*, but Y values can be *real*, *complex*, or *polar*.
- Instead of storing all possible X values, a waveform stores a starting X value (X_start), the number of values (Size), and the distance between each values (X_increment).
- The *units* associated with the X axis values and Y axis values are stored (as LPCTSTR). In a given waveform, X axis values are defined in the terms of X_units, Y axis values are defined in the terms of Y_units.

Example

Imagine an AC signal (waveform) digitized and captured using a hardware capture instrument:

- The waveform Y axis represents sampled voltage values
- The waveform X axis represents even increments of time
- The time scale on the X axis starts at the time the instrument receives the first sample clock. If no trigger delay is specified the first waveform sample is considered to have occured at time zero (X_start = 0).
- If the instrument samples at 1MHz, then the time between each sample is 1uS (X_increment = 1uS).
- If 1000 samples are captured the last sample would have an X value of:

  `X_start + 999 * X_increment = 999uS`

- Using `waveform_subset()` to remove the first one hundred samples and return the remainder, the new X_start value is 100uS. The X_increment remains 1uS and the last data point remains at 999uS.
- Using `waveform_deinterleave()` to split the waveform into two waveforms, the X_increment value changes to 2uS in both waveforms. The first waveform's X_start value remains at 100uS. The second waveform's X_start value is 101uS. These waveforms still associate the Y data values with the original relative time at which they were captured.

3.27.5 Decibel (dB)

See Waveform Functions, Waveform Overview.

The decibel (dB) is a unit based on a ratio of two values:

$$Voltage = dB = 20 \times \log_{10}\left(\frac{V1}{V2}\right)$$

$$Power = dB = 10 \times \log_{10}\left(\frac{P1}{P2}\right)$$

A positive dB value indicates a ratio of >1, negative dB < 1:

$$6.02dB = 20 \times \log_{10}\left(\frac{2V}{1V}\right) \qquad -6.02dB = 20 \times \log_{10}\left(\frac{1V}{2V}\right)$$

$$20dB = 20 \times \log_{10}\left(\frac{10V}{1V}\right) \qquad -20dB = 20 \times \log_{10}\left(\frac{1V}{10V}\right)$$

$$40dB = 20 \times \log_{10}\left(\frac{100V}{1V}\right) \qquad -40dB = 20 \times \log_{10}\left(\frac{1V}{100V}\right)$$

$$60dB = 20 \times \log_{10}\left(\frac{1000V}{1V}\right) \qquad -60dB = 20 \times \log_{10}\left(\frac{1V}{1000V}\right)$$

$$80dB = 20 \times \log_{10}\left(\frac{10000V}{1V}\right) \qquad -80dB = 20 \times \log_{10}\left(\frac{1V}{10000V}\right)$$

$$-100dB = 20 \times \log_{10}\left(\frac{1V}{100000V}\right)$$

Digital devices are based on powers of two, where each bit is worth 6dB:

$$\texttt{1 Bit} = 6.02dB = 20 \times \log_{10}(2^{1})$$

$$\texttt{8 Bits} = 48dB = 20 \times \log_{10}(2^{8})$$

$$\texttt{10 Bits} = 60dB = 20 \times \log_{10}(2^{10})$$

$$\texttt{12 Bits} = 72dB = 20 \times \log_{10}(2^{12})$$

$$\texttt{14 Bits} = 84dB = 20 \times \log_{10}(2^{14})$$

$$\texttt{16 Bits} = 96dB = 20 \times \log_{10}(2^{16})$$

$$\texttt{24 Bits} = 144dB = 20 \times \log_{10}(2^{24})$$

In digital devices the LSB is worth 1 part in 2^N possible states:

8 Bit DUT, LSB = $-48dB = 20 \times \log_{10}\left(\frac{1}{2^8}\right)$

10 Bit DUT, LSB = $-60dB = 20 \times \log_{10}\left(\frac{1}{2^{10}}\right)$

12 Bit DUT, LSB = $-72dB = 20 \times \log_{10}\left(\frac{1}{2^{12}}\right)$

14 Bit DUT, LSB = $-84dB = 20 \times \log_{10}\left(\frac{1}{2^{14}}\right)$

16 Bit DUT, LSB = $-96dB = 20 \times \log_{10}\left(\frac{1}{2^{16}}\right)$

24 Bit DUT, LSB = $-144dB = 20 \times \log_{10}\left(\frac{1}{2^{24}}\right)$

To convert from dB to a ratio, use the following:

$$10^{\left(\frac{dB}{20}\right)} = ratio$$

$$10^{\left(\frac{6.02dB}{20}\right)} = 2$$

$$10^{\left(\frac{20dB}{20}\right)} = 10$$

$$10^{\left(\frac{40dB}{20}\right)} = 100$$

$$10^{\left(\frac{60dB}{20}\right)} = 1000$$

To convert from dB to a N bits, use the following:

$\frac{dB}{6}$ = Bits

$\frac{48dB}{6}$ = 8 Bits

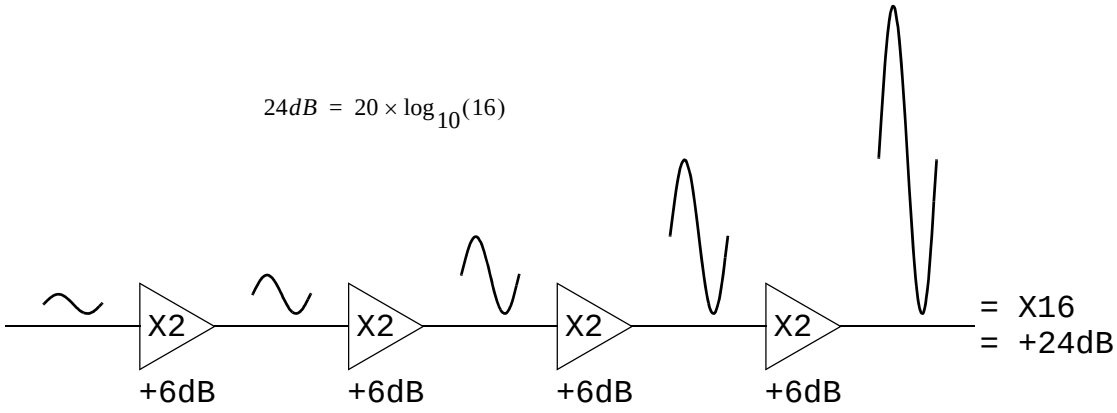
$\frac{60dB}{6}$ = 10 Bits

$\frac{72dB}{6}$ = 12 Bits

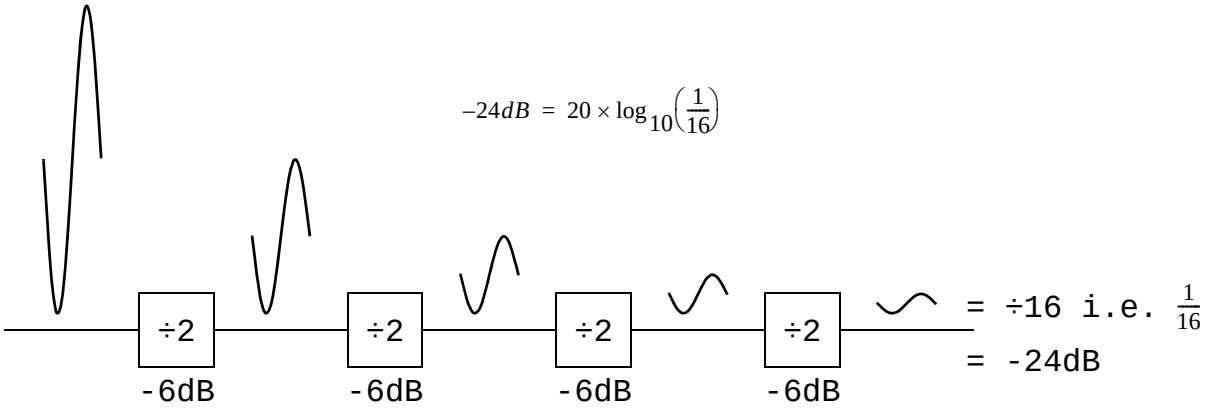
$\frac{84dB}{6}$ = 14 Bits

$\frac{96dB}{6}$ = 16 Bits

Using values in dB, products can be added, ratios can be subtracted.



Attenuators reduce amplitude, resulting in negative dB values:



Beware! Some competitor's software outputs *Volts peak squared* when converting CRECT_WAVE to POLAR_WAVE:

$$db = 10 \times \log_{10}\left(\frac{V1^2}{V2^2}\right)$$

This must be considered when converting programs from other companies equipment. Notice that multiplying by 10 instead of 20 is the same thing as taking the square root of the ratio before converting to dB.

3.27.6 Waveform File Formats

See Waveform Functions, Waveform Overview.

A waveform may be stored or read to/from a disk file using Waveform File Write/Read Functions and WaveformTool (MSWT)'s File->Open, File->Save and File->Save As controls.

The following waveform file formats are supported:

Format	Origin	Comments
.nwav	Nextest proprietary file format	This is the *native* Nextest mixed signal (Lightning) file format. The only file format which supports all sample value notations (see Waveform Sample Value Notations).
.wav	Microsoft WAV file format	.wav files are used by various Microsoft Windows (and other) applications. Only RRECT_WAVE sample value notation is supported (see Waveform Sample Value Notations). See Note: below.
.au	Suns audio format	.au files are used by various Sun Microsystems applications. Only RRECT_WAVE sample value notation is supported (see Waveform Sample Value Notations).
.aif	Apple Macintosh audio format	.aif files are used by various Apple Computer applications. Only RRECT_WAVE sample value notation is supported (see Waveform Sample Value Notations).

Note: there are 23 different `.wav` waveform encodings, most of which can NOT be used directly (MPEG 3 layer 1, for example).

File read/write support for *foreign* file formats is documented in Waveform File Write/Read Functions.

3.27.6.1 Nextest Waveform File Format (.nwav)

See Waveform Functions, Waveform Overview.

The `.nwav` file format is the only format directly supported by the system software. Other formats (`.wav`, `.au` etc) will be supported selectively.

Guiding principles

- The `.nwav` file format is intended for machine generated and machine read files.
- It is not a programming language. It is text based to make it possible for the user to create an input file with a text editor, C program, Perl script, etc.
- The format grammar should be simple.
- An `.nwav` waveform definition should be editable with a standard text editors (emacs, notepad, etc.).
- An `.nwav` waveform should support all attributes of the Nextest `Waveform*` object.
- Keywords are case insensitive.
- Unknown keywords generate a warning, but, are otherwise ignored. This assumes that the statement conforms to basic grammar: <keyword> <value string>
- Warnings are written to the appropriate UI output window.

3.27.6.2 .nwav Grammar Description

See Waveform Functions, Waveform Overview.

```
// ... <eol>
comment specifier.
```

```
version  <string>
Some string used to determine the structure of the rest of the file.

name  <string>
Name of the waveform.
Optional, default is the filename minus it extension.

sample_interval  <double>
The time interval between samples.
Optional, default is 1.0e-3

delay  <double>
The delay of the waveform.
Optional, default is 0.0

x_units <string>
X axis description, can be any string.
Optional, default is "s"

y_units <string>
X axis description, can be any string.
Optional, default is "v"

domain  [ Frequency | Time ]
Does the data represent a time domain or frequency domain waveform?
Optional, default is Time.

type  [ rrect | crect | polar | rlong ]
What type of waveform is this. Used to interpret
waveform data that follows.

size  <integer>
How many waveform samples follow. This is optional. The size of the
resulting waveform is defined by the number samples in the file.
If the sizes don't match, reader issues a warning and ignores the
value.

[*<double> | *(<double>,<double>) | *<hex>]
Data for waveform. Must be last data in the file. The "hex" format
is legal with rlong only. Once the reader starts reading the
waveform data, it reads to the end of the file.
```

3.27.7 Types, Enums, etc.

See Waveform Overview.

Description

The following type definitions support the waveform functions documented in this section.

Usage

The two Pi definitions below are available for convenience:

```
#define NEXTEST_PI 3.14159265358979323846264338327950288419716939937751
#define NEXTEST_TWO_PI (2.0 * NEXTEST_PI)
```

The `WType` enumerated type values are used as both arguments to and values returned from waveform related functions. These values represent the notation used to define a waveform's sample values. Details are documented in Waveform Sample Value Notations and Waveform Sample Programming. The `BadWave` value is returned in situations where a given waveform is partially or completely undefined i.e. the `Waveform*` is valid but the underlying waveform is incompletly defined:

```
enum WType { BadWave, RRectWave, CRectWave, PolarWave, RLongWave };
```

The `LineFitMethod` enumerated type is used to specify the method used to analyize histograms when performing INL/DNL analysis. See INL & DNL Functions.

```
enum LineFitMethod {t_endpoint_fit,
                    t_adjusted_fit,
                    t_least_squares_fit };
```

The `RoundingMethod` enumerated type is used to specify a rounding method when rounding is to be performed:

```
enum RoundingMethod {  t_round_to_nearest,
                       t_round_down,
                       t_round_up,
```

```
t_round_to_even,
t_round_to_odd,
t_truncate };
```

Enum Value	Description
t_round_to_nearest	The value is set to the closest integer. If equidistant values exist, the next largest integer is selected.
t_round_down	The greatest integer not greater than the value is selected. Sometimes referred to as the *floor* function.
t_round_up	The smallest integer not less than the value is selected. Sometimes referred to as the *ceiling* function.
t_round_to_even	The value is set to the closest even integer. If equidistant values exist, the next largest even integer is selected.
t_round_to_odd	The value is set to the closest odd integer. If equidistant values exist, the next largest odd integer is selected.
t_truncate	Truncation is the default operation used by C and C++ to convert from a floating point number to an integer. It involves removing the fractional part of the value and leaving the integer part. For positive numbers, it has the same effect as t_round_down, but for negative numbers it has the same effect as t_round_up.

The following table helps illustrate the effect of each of the rounding methods:

Original Data	-2.3	-1.7	-1.3	-0.5	0.3	0.5	0.7	1.0	1.3	1.7	2.0	2.3	2.7
t_round_to_nearest	-2	-2	-1	0	0	1	1	1	1	2	2	2	3
t_round_down	-3	-2	-2	-1	0	0	0	1	1	1	2	2	2
t_round_up	-2	-1	-1	0	1	1	1	1	2	2	2	3	3
t_round_to_even	-2	-2	-2	0	0	0	0	2	2	2	2	2	2
t_round_to_odd	-3	-1	-1	-1	1	1	1	1	1	1	3	3	3
t_truncate	-2	-1	-1	0	0	0	0	1	1	1	2	2	2

The following declarations are used by waveform_write_file() to specify the type of encoding which is to be used when writing the output file:

```
extern const unsigned long WavefileEncoding_PCM;
extern const unsigned long WavefileEncoding_FLOAT;
```

3.27.8 Waveform Sample Value Notations

See Waveform Functions, Waveform Overview, Waveform Mathematical View

Waveform sample values can be represented in several ways, using either rectangular or polar notation.

Note: in this document, the term *waveform type* refers to the notation used to describe its sample values.

The notation used to define the sample values of a given waveform can be determined using `waveform_get_typename()`.

In general, the notation that most naturally fits a given task is used. Sample values in one notation can be transformed into the other representations using Waveform Manipulation Functions. Note that many Waveform Functions only support specific notation(s) or the effect of the function is defined separately for some notations vs. other notations.

There are three rectangular notations and one polar notation:

Notation	Description	Comments
RRECT_WAVE	Real rectangular notation A one-part notation WType = RRectWave	Each waveform sample consists of a signed floating point value (double) representing its magnitude relative to 0V.
RLONG_WAVE	Long real rectangular notation A one-part notation WType = RLongWave	Identical to a RRECT_WAVE except integer values are used instead of floating point values. Only 53 bits + sign are used. Typically used to represent DAC/ADC codes and histogram data. More below.
CRECT_WAVE	Complex rectangular notation A two-part notation WType = CRectWave	Each waveform sample consists of two floating point (double) values, a real value (cosine) and an imaginary value (sine), stored in separate arrays. Notation typically returned by FFT.
POLAR_WAVE	Complex polar notation A two-part notation WType = PolarWave	Polar notation. Also called a vector notation (not to be confused with logic test vectors). Each sample consists of two values: magnitude@angle, where angle is specified in radians (2π radians = 360°). The angle is also called *theta*, and shown using the greek symbol θ. In signal analysis, the angle is used to represent phase, and magnitude frequency. Typically used to store frequency domain (spectral) data.
BAD_WAVE	Used to indicate an undefined waveform or an error when a Waveform* is returned by a function. WType = BadWave.	

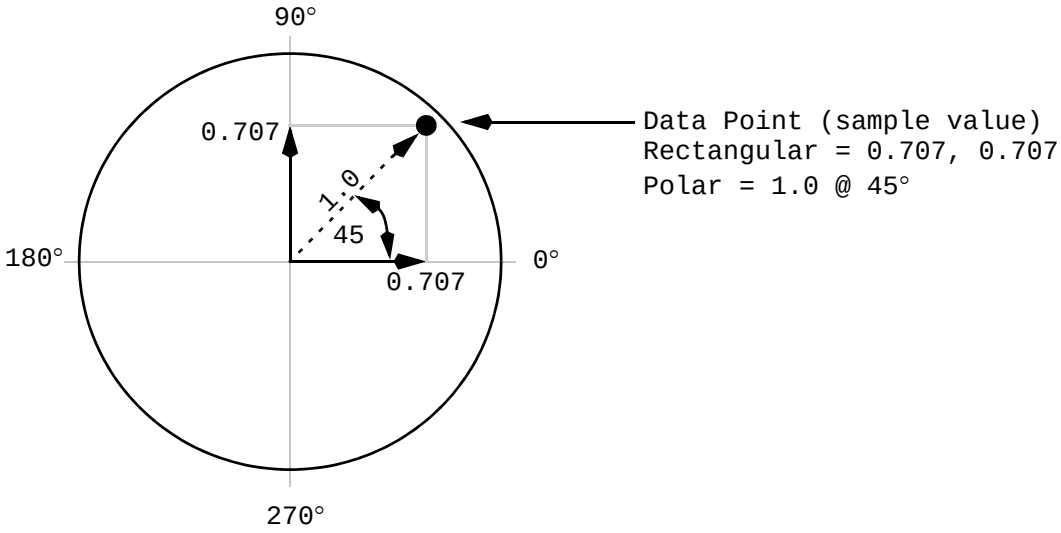
Figure-56: Waveform Sample value Notation

The notation used by a given waveform can be determined using:

```
waveform_get_typename( some_waveform ); //waveform_get_typename()
```

Note: it is not legal to mix notations defining the sample values of a given waveform. For example, an existing waveform defined using the RRECT_WAVE notation can only be sucessfully accessed using the *_rrect* form functions (waveform_get_rrect(), etc.). In most cases, if this rule is violated a warning message will be displayed in the appropriate site output window and/or the function will return an error value. BUT, program execution will not be interrupted and incorrect test results will be likely.

Rectangular forms describe a data point in terms of length and direction of two sides of a rectangle. Polar describe a data point in terms of the length of a [diagonal] line across a rectangle and the phase angle of that line:



Using polar notation, magnitude values are specified in peak amplitude:

$RMS = peak \times 0.7071068$

Peak-to-peak = peak × 2

$dBv = 20 \times Log_{10}\left(\frac{Peak}{1.0}\right)$

$dBm = 20 \times Log_{10}\left(\frac{Peak}{0.2236}\right)$ Assuming 50Ω

Using polar notation, phase is in Radians:

Phase is relative to cosine i.e. a sinewave will have a -90° $\left(\frac{\pi}{2}\right)$ offset.

$$degrees = radians\left(\frac{180}{\pi}\right)$$

$$radians = degrees\left(\frac{\pi}{180}\right)$$

In general, all functions which accept RLONG_WAVE notation will try to determine the resulting sample value type. If it can be predicted that no non-integer sample values can result from the operation, the result will use RLONG_WAVE notation. If it's possible that a non-integer sample value may occur, the result will use the RRECT_WAVE notation, even if all the samples end up being integer values.

The notation terms used above, and throughout this document, are based on the following definitions:

```
#define RRECT_WAVE "RRectWave"
#define RLONG_WAVE "RLongWave"
#define CRECT_WAVE "CRectWave"
#define POLAR_WAVE "PolarWave"
#define BAD_WAVE   "BadWave"
```

3.27.9 Waveform Units

See Waveform Functions, Waveform Overview, Waveform Mathematical View

Description

To most, a waveform is a sine wave, triangle wave, etc. However, using the Waveform Functions, a waveform is just a container for 2-D data.

To use a waveform requires that the *units* of the data be specified, for both the X and Y axis of the waveform. All waveforms have a single X_units type and a single Y_units type.

Waveform units are specified as a *"string"* (LPCTSTR). Although any string may be used, the following strings are defined in and recognized, to make conversions, check for errors, etc.:

```
#define SCALE_AMPERES     "amperes"
#define SCALE_BOOLEAN     "boolean"
#define SCALE_CODES       "codes"
#define SCALE_COUNTS      "counts"
#define SCALE_DECIBELS    "decibels"
#define SCALE_HERTZ       "Hertz"
#define SCALE_MICROVOLTS  "microvolts"
#define SCALE_NANOAMPERES "nanoamperes"
#define SCALE_NOXUNITS    "noXUnits"
#define SCALE_NOYUNITS    "noYUnits"
#define SCALE_OFFSET      "offset"
#define SCALE_PICOSECONDS "picoseconds"
#define SCALE_RADIANS     "radians"
#define SCALE_SAMPLES     "samples"
#define SCALE_SECONDS     "seconds"
#define SCALE_VOLTS       "volts"
```

`seconds`
`minutes`
`hours`

Allows time values to be converted between any of the three base units, in concert with optional metric prefixes (see Unit Prefixes) i.e. it is possible to convert milliseconds to microhours.

Unit Prefixes

The following standard prefixes are recognized when applied to base units (volts, amperes, hertz, seconds). The following prefixes are recognized:

`atto`	`centi`	`deci`	`femto`
`giga`	`kilo`	`nano`	`mega`
`micro`	`milli`	`pico`	`tera`

Thus, in the context of Waveform Units, the following (useful) examples are recognized:

- “millivolts” “nanovolts”
- “milliseconds” “microseconds” “nanoseconds”
- “milliamperes” “microamperes” “picoamperes”

Some invalid examples include:

- “millicodes” “nanolsbs” “picoradians”

Units Applications

The X_units type applies equally to all sample values in the waveform.

BAD_WAVE type waveforms have X_units = SCALE_NOXUNITS and Y_units = SCALE_NOYUNITS.

RRECT_WAVE and RLONG_WAVE waveform types only have one data part, therefore the Y_units type applies to it.

CRECT_WAVE waveform types have two data parts, real and imaginary. The Y_units type applies to both parts.

POLAR_WAVE waveform types have two data parts, magnitude and phase. The Y_units type applies only to the magnitude part; the Y_units type of the phase data part is always assumed to be SCALE_RADIANS.

The following Waveform Generate Functions each create a time domain waveform, thus they each set the output waveform's X_units to SCALE_SECONDS and the Y_units to SCALE_VOLTS.

- `waveform_generate_triangle_wave()`
- `waveform_generate_sine_wave()`
- `waveform_generate_ramp()`
- `waveform_generate_square_wave()`
- `waveform_generate_gaussian_noise()`
- `waveform_generate_white_noise()`
- `waveform_generate_periodic_white_noise()`
- `waveform_generate_periodic_pink_noise()`
- `waveform_generate_DC()`

The following functions require explicit X_units and Y_units be specified for the output waveform because they can be used to create waveform containing various data types:

- waveform_set_rrect()
- waveform_set_rlong()
- waveform_set_crect()
- waveform_set_polar()
- waveform_constant_fill()

User code can change X_units and/or Y_units for a specified waveform using waveform_set_x_scale() and waveform_set_y_units().

The Waveform FFT Functions recognize time domain waveforms as well as frequency domain waveforms. Using these functions, if the X_units type of the input waveform is a recognized time unit the output waveform's X_units will be changed to SCALE_HERTZ, after the corresponding conversions are made. Similarly, if the X_units type of the input waveform is a recognized frequency unit the output waveform's X_units will be converted to SCALE_SECONDS, after the corresponding conversion from frequency to time is made. An FFT does not modify the waveform's Y_units. Any unrecognized X_units will be converted as follows:

```
"unknown"          --> "fourier(unknown)"
"fourier(unknown) --> "unknown"
```

Other functions that set the X_units or Y_units to known values include:

- waveform_settling_time()
- waveform_histogram() sets the output waveform's X_units to match the input waveform's Y_units. It sets the output waveform's Y_units to SCALE_COUNTS.
- The INL and DNL waveforms returned by the INL & DNL Functions have X_units and X_units set based on the input waveform. See INL & DNL Functions.
- If waveform_split() is called with an input waveform of type POLAR_WAVE, the first output waveform's Y_units will be set to match the input waveform's Y_units. The second output waveform's Y_units are set to SCALE_RADIANS.
- The waveform Waveform Equality Functions and Waveform Boolean Functions may output a waveform consisting of boolean sample values. These waveforms have Y_units set to SCALE_BOOLEAN.

Only the functions noted above set or check for specific waveform X_units/Y_units types.

3.27.9.1 Units Applications

See Waveform Units.

The following table shows the types of units to be used in the situations noted:

Type	X Axis	Y Axis	Comments
AC Waveform	Time SCALE_SECONDS	Voltage SCALE_VOLTS	waveform_set_rrect(), waveform_set_rlong(), waveform_set_crect(), waveform_set_polar()
Histogram		SCALE_COUNTS	waveform_histogram()
Voltage Measurement	Voltage SCALE_VOLTS	SCALE_SAMPLES	
Current Measurement	Current SCALE_AMPERES	SCALE_SAMPLES	
These values need example applications =>	SCALE_CODES	SCALE_BOOLEAN	
	SCALE_HERTZ	SCALE_RADIANS	
	SCALE_AMPERES	SCALE_NOXUNITS	
	SCALE_DECIBELS	SCALE_NOYUNITS	

The functions noted above directly require that units be specified as an argument to the function. This results in a waveform which has correctly defined units for the target application. Other functions change the units on the X or Y axis. It is possible to explicitly set units for a waveform using the Waveform Set/Get X/Y Units Functions functions.

Note: some functions check the units of a waveform, and an attempt to use a waveform with units inappropriate for the target application will cause an error message to be displayed in the appropriate controller window. However, testing otherwise continues and proper operation is unlikely.

3.27.10 Waveform Macros

See Waveform Overview.

Description

All waveforms are fundamentally a C++ object. However since the Nextest software is C-based, waveforms are represented by the `Waveform*` data type.

Static waveforms (`Waveform*`) are created using the `WAVEFORM()` macro. These are created as the test program loads, are expected to be persistent for the duration of the program session, and do not require any user code to manage memory, etc.

The `WAVEFORM()` macro is designed to allow waveform definition functions to be executed within the macro body code, to define waveform attributes, however this is not required. See Examples below. These waveforms will always have a *name* which can be retrieved using `waveform_get_name()`.

Within the body of the `WAVEFORM()` macro, the waveform object being defined is accesible using either the waveform name, or using the generic token `obj`. For example:

```
WAVEFORM( myWF ) {
   LPCTSTR n = waveform_get_name( obj ); // Or...
   n = waveform_get_name( myWF );
}
```

Dynamic waveforms are created using the `waveform_create()` function. This method is discouraged due to the requirement for robust memory management and the associated risk of memory leaks. The `waveform_destroy()` function is used to destroy dynamic waveforms and free the associated memory. Waveforms created using `waveform_create()` optionally have a “name” which can be retrieved using `waveform_get_name()`.

See Waveform* Attributes for a description attributes common to all waveforms.

Usage

The following macro is used to create a new `Waveform` object:

```
WAVEFORM( name ){
   // Optionally, call functions here to define waveform attributes
}
```

The following macro is used to make an external or forward declaration:

```
EXTERN_WAVEFORM( name )
```

The following macro is used to add an existing `Waveform` object as a component of a new `Waveform` being created:

```
INCLUDE_WAVEFORM( name )
```

where:

name is the waveform (`Waveform*`) being create, defined, or include.

Example

The examples below demonstrate the four basic styles of defining static waveforms using the `WAVEFORM` macro.

Example 1:

These examples both create a new waveform named `WF1`. Neither define any waveform attributes:

```
WAVEFORM( WF1 ){} // Static waveform
Waveform* WF1 = waveform_create(); // Dynamic waveform
```

Example 2:

This example creates a new waveform named `WF1` and defines its attributes by reading the specified waveform file using `waveform_read_file()`. Note that the `Waveform*` being created is passed as an argument to `waveform_read_file()`:

```
WAVEFORM( WF1 ) {
   waveform_read_file( obj, "../wf1File.nwav");
}
```

Example 3:

This example creates the waveform named `simpleWave` and defines its attributes in place:

```
WAVEFORM ( simpleWave ) {
   double samples[] = {
      0.0,
      0.0627905,
      0.125333,
      0.187381,
      // ... snip: typically lots more values are needed ...
      -0.125333,
      -0.0627905,
   };
   int size = sizeof(samples)/sizeof(double);
```

```
   // See waveform_set_rrect()
   void waveform_set_rrect( obj,
                            size,
                            samples,
                            SCALE_VOLTS,
                            0,
                            reciprocal(10 MHZ),
                            SCALE_SECONDS );
}
```

Example 4:

This example creates the waveform named `tri_1K`. The set of waveform samples is created within the macro body by calling `waveform_generate_triangle_wave()`:

```
WAVEFORM( tri_1K ) {
   int num_samples = 1000;
   // See waveform_generate_triangle_wave()
   void waveform_generate_triangle_wave( obj,
                                         int num_samples,
                                         10 MHZ,
                                         0 V,
                                         3 V,
                                         10 MHZ);
}
```

3.27.11 waveform_create(), waveform_destroy()

See Waveform Overview.

Description

The `waveform_create()` function can be used to dynamically create a new `Waveform*`. The `WAVEFORM()` macro is used to create static waveforms during the program load process.

The `waveform_destroy()` function can be used to destroy an existing `Waveform*` and free associated memory.

Note: when using dynamic waveforms it is possible to create memory leaks, which may cause the test program to crash, with very confusing symptoms. It is the user's responsibility to correctly manage memory when using dynamic waveforms.

Usage

```
Waveform* waveform_create( LPCTSTR name WDEFAULT_VALUE( 0 ) );
void waveform_destroy( Waveform* &obj );
```

where:

name is optional, and if used is the name of the waveform to be created (as a `LPCTSTR`). See `waveform_get_name()`.

obj is a pointer to an existing `Waveform*` to be destroyed.

`waveform_create()` returns a valid `Waveform*` if the waveform is created successfully, otherwise a `NULL` pointer is returned.

Example

The following example dynamically creates a `Waveform*` (with no name), uses it to store a waveform, and destroys it when done.

```
Waveform* tmpWF = waveform_create();
// Define and process tmpWF samples here
waveform_destroy( tmpWF ); // Free waveform memory
```

3.27.12 waveform_invalidate()

See Waveform Overview.

Description

The `waveform_invalidate()` function deallocates the memory used by the specified waveform and marks the waveform Type = `BAD_WAVE` . This is the same state as an uninitialized waveform.

Usage

```
void waveform_invalidate( Waveform* obj );
```

where:

Example

???

3.27.13 Waveform Generate Functions

See Waveform Overview, Waveform Mathematical View, Waveform Units

The following functions are used to create waveforms with properties related to the name of the function:

- `waveform_generate_triangle_wave()`
- `waveform_generate_sine_wave()`
- `waveform_generate_ramp()`
- `waveform_generate_square_wave()`
- `waveform_generate_gaussian_noise()`
- `waveform_generate_white_noise()`
- `waveform_generate_periodic_white_noise()`
- `waveform_generate_periodic_pink_noise()`
- `waveform_constant_fill()`
- `waveform_randomize(), waveform_reset_random_seed()`

3.27.13.1 waveform_generate_triangle_wave()

See Waveform Overview, Waveform Generate Functions.

Description

The `waveform_generate_triangle_wave()` function is used to define a waveform with sample values which represent a triangle AC waveform.

Arguments to `waveform_generate_triangle_wave()` are used to specify:

- Number of sample values (`num_samples`)
- Sample frequency (`sampling_freq`). (Also see X_increment in Waveform Mathematical View)
- The triangle waveform frequency (`frequency`)
- The initial waveform phase
- The minimum (`low_level`) and maximum level (`high_level`) of the waveform, which also sets the initial ramp direction (ascending or descending, see Usage)

The number of triangle waveform cycles contained in the waveform can be determined using:

$$M = \frac{Ft \times N}{Fs}$$

where:

```
M = number of triangle wave cycles
N = number of samples
Ft = frequency of triangle wave
Fs = sample frequency
```

Also note the following:

- The waveform samples are defined in `RRECT_WAVE` notation
- Y_units is set to `SCALE_VOLTS`
- X_start is set to 0
- X_units is set to `SCALE_SECONDS`

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_generate_triangle_wave(
                    Waveform* out_wave,
                    int num_samples,
                    MKSFrequency sampling_freq,
                    MKSVolts low_level,
```

```
                    MKSVolts high_level,
                    MKSFrequency frequency DEFAULT_VALUE( 1.0 ),
                    double phase DEFAULT_VALUE( 0.0 ) );
```

where:

out_wave identifies the output waveform.

num_samples specifies the number of samples in the `Waveform*` being defined.

sampling_freq specifies the sample frequency, in `MKSFrequency`.

low_level and **high_level** specify the minimum and maximum voltage level of the waveform, in `MKSVolts`.

frequency is optional, and if used, specifies actual frequency of the triangle waveform, in `MKSFrequency`. Default = 1Hz.

phase is optional, and if used, specifies initial waveform phase, in radians. Default = 0.0. Commonly used values include:

- **phase** = 0.0 the the triangle wave will start at midlevel $\left(\frac{\text{low\_level} + \text{high\_level}}{2}\right)$
- To start the triangle wave at the **high_level** specify **phase** of $\frac{\Pi}{2}$
- To start at the **low_level** specify $\frac{-\Pi}{2}$.

Example

???

3.27.13.2 waveform_generate_sine_wave()

See Waveform Overview, Waveform Generate Functions.

Description

The `waveform_generate_sine_wave()` function is used to define a waveform with sample values which represent a sine wave AC waveform.

Arguments to `waveform_generate_sine_wave()` are used to specify:

- Number of sample values (`num_samples`)

- Sample frequency (`sampling_freq`). (Also see X_increment in Waveform Mathematical View)
- The sine wave waveform frequency (`frequency`)
- The initial waveform phase
- The minimum (`low_level`) and maximum level (`high_level`) of the waveform, which also sets the initial ramp direction (ascending or descending, see Usage)
- The number of sine wave cycles contained in the waveform can be determined using:

$$M = \frac{Ft \times N}{Fs}$$

where:

```
N = number of samples
Ft = frequency of the sine wave
Fs = sample frequency
M = number of sine wave cycles
```

Also note the following:

- Y_units is set to SCALE_VOLTS
- X_start is set to 0
- X_units is set to SCALE_SECONDS

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_generate_sine_wave(
                            Waveform* out_wave,
                            int num_samples,
                            MKSFrequency sampling_freq,
                            MKSVolts pkpkAmplitude,
                            MKSVolts DCoffset,
                            MKSFrequency frequency,
                            double phase DEFAULT_VALUE( 0.0 ) );
```

where:

`out_wave` identifies the output waveform.

num_samples specifies the number of samples in the `Waveform*` being defined.

sampling_freq specifies the sample frequency, in `MKSFrequency`.

pkpkAmplitude specifies the peak-to-peak voltage level of the waveform, in `MKSVolts`.

DCoffset specifies the DC offset voltage of the waveform, in `MKSVolts`.

frequency is optional, and if used, specifies actual frequency of the sine waveform, in `MKSFrequency`. Default = 1Hz.

phase is optional, and if used, specifies initial waveform phase, in radians. Default = 0.0. Commonly used values include:

- **phase** = 0.0 the the sine wave will start at midlevel $\left(\frac{\text{low\_level} + \text{high\_level}}{2}\right)$
- To start the sine wave at the **high_level** specify **phase** of $\frac{\Pi}{2}$
- To start at the **low_level** specify $\frac{-\Pi}{2}$.

Example

???

3.27.13.3 waveform_generate_ramp()

See Waveform Overview, Waveform Generate Functions.

Description

The `waveform_generate_ramp()` function is used to define a waveform with sample values which represent a linear ramp AC waveform.

Arguments to `waveform_generate_ramp()` are used to specify:

- Number of sample values (`num_samples`)
- Sample frequency (`sampling_freq`). (Also see X_increment in Waveform Mathematical View)
- The linear ramp waveform frequency (`frequency`)
- The initial waveform phase
- The `start_level` and `end_level` of the waveform, which also sets the ramp direction (ascending or descending, see Usage)

- The number of ramp cycles contained in the waveform can be determined using:

$$M = \frac{Ft \times N}{Fs}$$

where:

```
N = number of samples
Ft = frequency of ramp wave
Fs = sample frequency
M = number of ramp wave cycles
```

Also note the following:

- Y_units is set to SCALE_VOLTS
- X_start is set to 0
- X_units is set to SCALE_SECONDS

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_generate_ramp(
                    Waveform* out_wave,
                    int num_samples,
                    MKSFrequency sampling_freq,
                    MKSVolts start_level,
                    MKSVolts end_level,
                    MKSFrequency frequency DEFAULT_VALUE( 1.0 ),
                    double phase DEFAULT_VALUE( 0.0 ) );
```

where:

out_wave identifies the output waveform.

num_samples specifies the number of samples in the Waveform* being defined.

sampling_freq specifies the sample frequency, in MKSFrequency.

start_level and **end_level** specify the lower and upper voltage levels of the waveform, in MKSVolts. If **end_level** < **start_level** an ascending ramp will be defined. If **end_level** > **start_level** a descending ramp will be defined.

frequency is optional, and if used, specifies actual frequency of the ramp waveform, in MKSFrequency. Default = 1Hz.

phase is optional, and if used, specifies initial waveform phase, in radians. Default = 0.0. Commonly used values include:

- **phase** = 0.0 the the ramp will start at midlevel $\left(\frac{\text{low\_level} + \text{high\_level}}{2}\right)$
- To start the ramp at the **high_level** specify **phase** of $\frac{\Pi}{2}$
- To start at the **low_level** specify $\frac{-\Pi}{2}$.

Example

???

3.27.13.4 waveform_generate_square_wave()

See Waveform Overview, Waveform Generate Functions.

Description

The `waveform_generate_square_wave()` function is used to define a waveform with sample values which represent a square wave AC waveform.

Arguments to `waveform_generate_square_wave()` are used to specify:

- Number of sample values (`num_samples`)
- Sample frequency (`sampling_freq`). (Also see X_increment in Waveform Mathematical View)
- The square wave waveform frequency (`frequency`)
- The `low_level` and `high_level` of the waveform. The generated sample values will only contain the values `low_level` and `high_level` i.e. no intermediate (transition) values are introduced. `low_level` and `high_level` also set the initial square wave level (high or low, see Usage)
- The number of square wave cycles contained in the waveform can be determined using:

$$M = \frac{Ft \times N}{Fs}$$

where:

```
N = number of samples
Ft = frequency of square wave
Fs = sample frequency
M = number of square wave cycles
```

Also note the following:

- Y_units is set to SCALE_VOLTS
- X_start is set to 0
- X_units is set to SCALE_SECONDS

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_generate_square_wave(
                    Waveform* out_wave,
                    int num_samples,
                    MKSFrequency sampling_freq,
                    MKSVolts low_level,
                    MKSVolts high_level,
                    MKSFrequency frequency DEFAULT_VALUE( 1.0 ),
                    double duty_cycle DEFAULT_VALUE( 50.0 ) );
```

where:

out_wave identifies the output waveform.

num_samples specifies the number of samples in the Waveform* being defined.

sampling_freq specifies the waveform sample frequency, in MKSFrequency.

low_level and **high_level** specify the minimum and maximum voltage level of the waveform, in MKSVolts. If **low_level** < **high_level** the initial square wave sample value will be set to **low_level**. If **low_level** > **high_level** the initial square wave sample value will be set to **high_level**.

frequency is optional, and if used, specifies actual frequency of the square wave waveform, in MKSFrequency. Default = 1Hz.

duty_cycle is optional, and if used specifies desired high time vs low time duty cycle, in percent. Default = 50%.

Example

???

3.27.13.5 waveform_generate_gaussian_noise()

See Waveform Overview, Waveform Generate Functions.

Description

The `waveform_generate_gaussian_noise()` function is used to define a waveform with sample values which represent white noise with a Guassian distribution. Note the following:

- Gaussian (normal) white noise is noise containing all frequencies, with a Gaussian distribution of voltages within the specified standard deviation. This means there will be more sample values clustered near the `DCoffset` voltage than there will be positioned further away.
- A histogram (see `waveform_histogram()`) of a Gaussian white noise waveform will exhibit the so-called *bell-curve*.
- The standard deviation of the waveform is specified using `std_dev`. This means that the actual voltage range will only be within the specified std_dev about 68% of the time; the full voltage swing will usually be 3 or 4 times as large.
- The random number generator is used. See `waveform_randomize()`, `waveform_reset_random_seed()`.
- When a waveform is required to have a Gaussian distribution within fixed voltage limits the `waveform_rescale()` function can be used to linearly rescale the waveform without affecting the Gaussian distribution of the sample values.

See `waveform_generate_white_noise()` and `waveform_generate_periodic_white_noise()`.

Also note the following:

- Y_units is set to `SCALE_VOLTS`
- X_start is set to 0
- X_increment is set to the reciprocal of the `sampling_frequency` argument.
- X_units is set to `SCALE_SECONDS`

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_generate_gaussian_noise(
                          Waveform* out_wave,
                          int num_samples,
                          MKSFrequency sampling_freq,
                          MKSVolts std_dev,
                          MKSVolts DCoffset DEFAULT_VALUE( 0.0 ) );
```

where:

out_wave identifies the output waveform.

num_samples specifies the number of samples in the `Waveform*` being defined.

sampling_freq specifies the sample frequency, in `MKSFrequency`.

std_dev specifies the peak-to-peak amplitude of the waveform, in `MKSVolts`.

DCoffset is optional, and if used, specifies the DC offset (centerpoint) of the waveform, in `MKSVolts`. Default = 0V.

Example

???

3.27.13.6 waveform_generate_white_noise()

See Waveform Overview, Waveform Generate Functions.

Description

The `waveform_generate_white_noise()` function is used to define a waveform with sample values which represent white noise.

White noise is noise containing all frequencies, with uniform probability. This is a random burst of noise and will not be periodic. See `waveform_generate_periodic_white_noise()`.

Each sample of the output waveform will vary between:

$$\frac{-pkpkAmplitude}{2} + DCoffset \quad \text{and} \quad \frac{+\text{pkpkAmplitude}}{2} + DCoffset$$

See `waveform_generate_periodic_white_noise()` and `waveform_generate_gaussian_noise()`.

The random number generator is used. See `waveform_randomize()`, `waveform_reset_random_seed()`.

Also note the following:

- Y_units is set to `SCALE_VOLTS`
- X_start is set to 0
- X_increment is set to the reciprocal of the `sampling_frequency` argument.
- X_units is set to `SCALE_SECONDS`

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_generate_white_noise(
                          Waveform* out_wave,
                          int num_samples,
                          MKSFrequency sampling_freq,
                          MKSVolts pkpkAmplitude,
                          MKSVolts DCoffset DEFAULT_VALUE( 0.0 ) );
```

where:

`out_wave` identifies the output waveform.

`num_samples` specifies the number of samples in the `Waveform*` being defined.

`sampling_freq` specifies the sample frequency, in `MKSFrequency`.

`pkpkAmplitude` specifies the peak-to-peak amplitude of the waveform, in `MKSVolts`.

`DCoffset` is optional, and if used, specifies the DC offset (centerpoint) of the waveform, in `MKSVolts`. Default = 0V.

Example

???

3.27.13.7 waveform_generate_periodic_white_noise()

See Waveform Overview, Waveform Generate Functions.

Description

The `waveform_generate_periodic_white_noise()` function is used to define a waveform with sample values which are an approximation of periodic white noise.

Using a periodic white noise waveform, the frequency response of a device may be analyzed with an FFT (see Waveform FFT Functions) without requiring averaging or windowing techniques. See `waveform_average()` and Waveform Window Functions.

The waveform created by `waveform_generate_periodic_white_noise()` results in an approximation to white noise since not all frequencies are represented, just all of the periodic frequencies up to the Nyquist limit. Given the number of waveform sample values (*N*) and sample frequency (*Fs)*, the output waveform will be:

$$\frac{1Fs}{N} + \frac{2Fs}{N} + \frac{3Fs}{N} + \ldots + \frac{\left(\frac{N}{2} - 1\right)Fs}{N}$$

The random number generator is used. See `waveform_randomize()`, `waveform_reset_random_seed()`.

The phases of each of the sine waves will be random.

The overall peak-to-peak amplitude is set to the value specified by `pkpkAmplitude`.

Also note the following:

- Y_units is set to `SCALE_VOLTS`
- X_start is set to 0
- X_increment is set to the reciprocal of the `sampling_frequency` argument.
- X_units is set to `SCALE_SECONDS`

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_generate_periodic_white_noise(
                            Waveform* out_wave,
                            int num_samples,
                            MKSFrequency sampling_freq,
                            MKSVolts pkpkAmplitude,
                            MKSVolts DCoffset DEFAULT_VALUE( 0.0 )
);
```

where:

out_wave identifies the output waveform.

num_samples specifies the number of samples in the Waveform* being defined.

sampling_freq specifies the sample frequency, in MKSFrequency.

pkpkAmplitude specifies the peak-to-peak amplitude of each sine wave component of the noise waveform, in MKSVolts. As noted in Description, this does NOT necessarily represent the peak-to-peak amplitude of the resulting waveform.

DCoffset is optional, and if used, specifies the DC offset (centerpoint) of the waveform, in MKSVolts. Default = 0V.

Example

???

3.27.13.8 waveform_generate_periodic_pink_noise()

See Waveform Overview, Waveform Generate Functions.

Description

The waveform_generate_periodic_pink_noise() function is used to define a waveform with sample values which are an approximation of periodic pink noise.

The waveform created by waveform_generate_periodic_pink_noise() results in an approximation to pink noise because not all frequencies are represented, just all of the periodic frequencies up to the Nyquist limit. Given the number of waveform sample values (*N*) and sample frequency (*Fs)*, the output waveform will be:

$$\frac{1Fs}{N} + \frac{2Fs}{N} + \frac{3Fs}{N} + ... + \frac{\left(\frac{N}{2} - 1\right)Fs}{N}$$

The random number generator is used. See `waveform_randomize()`, `waveform_reset_random_seed()`.

The phases of each of the sine waves will be random.

The overall peak-to-peak amplitude is set to the value specified by `pkpkAmplitude`.

Also note the following:

- Y_units is set to `SCALE_VOLTS`
- X_start is set to 0
- X_increment is set to the reciprocal of the `sampling_frequency` argument.
- X_units is set to `SCALE_SECONDS`

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_generate_periodic_pink_noise(
                          Waveform* out_wave,
                          int num_samples,
                          MKSFrequency sampling_freq,
                          MKSVolts pkpkAmplitude,
                          MKSVolts DCoffset DEFAULT_VALUE( 0.0 ) );
```

where:

`out_wave` identifies the output waveform.

`num_samples` specifies the number of samples in the `Waveform*` being defined.

`sampling_freq` specifies the sample frequency, in `MKSFrequency`.

`pkpkAmplitude` specifies the peak-to-peak amplitude of the first sine wave component of the noise waveform, in `MKSVolts`. See Description.

DCoffset is optional, and if used, specifies the DC offset (centerpoint) of the waveform, in MKSVolts. Default = 0V.

Example

???

3.27.13.9 waveform_generate_DC()

See Waveform Overview, Waveform Generate Functions.

Description

The waveform_generate_DC() function is used to generate a waveform containing a specified number of sample values, each of a specified constant value . Note the following:

- The output waveform Size is set to the num_samples argument. Any prior waveform contents are overwritten or deleted.
- Waveform sample values are defined using RRECT_WAVE notation.
- The output waveform Y_units type is set = SCALE_VOLTS.
- The output waveform X_units type is set = SCALE_SECONDS.
- The output waveform X_increment value is set to the reciprocal of the sampling_freq.
- The output waveform sample rate = 1, and X_units = SCALE_SECONDS.

Usage

```
void waveform_generate_DC( Waveform* out_wave,
                           int num_samples,
                           MKSFrequency sampling_freq,
                           MKSVolts DCoffset DEFAULT_VALUE( 0.0 ) );
```

where:

out_wave identifies the output waveform.

num_samples specifies the number of samples in the Waveform* being defined.

sampling_freq specifies the sample frequency, in MKSFrequency.

DCoffset is optional, and if used, specifies the value for each sample value, in MKSVolts. Default = 0V.

Example

???

3.27.13.10 waveform_constant_fill()

See Waveform Overview, Waveform Generate Functions.

Description

The waveform_constant_fill() function can be used to define a waveform containing a specified number of sample values, each of a specified constant value . Note the following:

- The output waveform Size is set to the num_samples argument. Any prior waveform contents are overwritten or deleted.
- If the fill value is an integer, the resulting waveform will be defined using RLONG_WAVE notation otherwise it will be RRECT_WAVE.
- The output waveform Y_units types are set = SCALE_NOYUNITS. See waveform_set_y_units().
- The output waveform sample rate = 1, and X_units = SCALE_NOXUNITS. See waveform_set_x_scale().

Usage

```
void waveform_constant_fill(  Waveform* out_wave,
                              int num_samples,
                              double value DEFAULT_VALUE( 0.0 ) );
```

where:

out_wave identifies the output waveform.

num_samples specifies the number of sample values in the output waveform. See Description.

value is optional, and if specified sets the fill value. Default = 0.0.

Example

???

3.27.13.11 waveform_randomize(), waveform_reset_random_seed()

See Waveform Overview, Waveform Generate Functions.

Description

The `waveform_randomize()` and `waveform_reset_random_seed()` functions do not create or define a Waveform. However, they will indirectly affect the creation of all noise Waveforms (`waveform_generate_gaussian_noise()`, `waveform_generate_white_noise()`, etc.), all of which use the random number generator to determine each sample value during the waveform creation process.

When generating a random number, the generator uses a *seed*. Each time a test program is loaded, the initial *seed* is set to the same value. This means that, by default, the first random number generated after the program is loaded will always be the same. And, the second random number will be the same each time, etc. Thus, by default, `waveform_generate_white_noise()`, for example, will always generate the same noise waveform the first time it is executed. This is often useful during program development or to reduce variations during correlation efforts.

But, it is also useful, at times, to have truly random data. To accomplish this, the seed value may be specified using `waveform_reset_random_seed()`, or the `waveform_randomize()` function can be used to generate a random seed.

Usage

```
void waveform_randomize();
void waveform_reset_random_seed(
                              unsigned int seed DEFAULT_VALUE(1));
```

where:

seed is optional, and if used specifies the value used to initialize the computer's random number generator seed value. Default = 1.

Example

???

3.27.14 Waveform File Write/Read Functions

See Waveform Overview, Waveform File Formats.

Description

The `waveform_write_file()` function is used to save one or more existing `Waveform*`(s) to a disk file. The `waveform_read_file()` function is used to read a waveform from a file to define one or more `Waveform*`(s).

Note the following:

- Several waveform file formats are supported, see Waveform File Formats.
- The `.nwav` format only supports one channel whereas the other formats can have up to 256 channels (stereo, for example).
- Two encoding schemes are supported:

Encoding Scheme	Formats Supported
`WavefileEncoding_PCM`	`.wav, .aif, .au`
`WavefileEncoding_FLOAT`	`.nwav, .wav`

Figure-57: Waveform Encoding Schemes

- The number of bits to use to represent the data can be specified. The value can be between 8 .. 32.
- User code is responsible for checking for read/write file permissions, file exists, etc.

Usage

The following functions read one or more waveform(s) from the specified file:

```
BOOL waveform_read_file( Waveform* obj, LPCTSTR fname );
BOOL waveform_read_file( WaveformArray** pWaveforms,
                         LPCTSTR fname );
BOOL waveform_read_file( LPCTSTR fname, Waveform* first,...);
```

The following functions write one or more waveform(s) to the specified file:

```
BOOL waveform_write_file( Waveform* obj, LPCTSTR fname );
```

```
BOOL waveform_write_file(const WaveformArray& waveforms,
                         LPCTSTR fname,
                         unsigned int encoding,
                         unsigned int bitwidth);
BOOL waveform_write_file(const WaveformArray* waveforms,
                         LPCTSTR fname,
                         unsigned int encoding,
                         unsigned int bitwidth);
BOOL waveform_write_file(LPCTSTR fname,
                         unsigned int encoding,
                         unsigned int bitwidth,
                         Waveform* first,...);
```

where:

obj identifies the `Waveform*` being written to the file on disk or being defined by reading the file from the disk.

fname specifies the path/file name to the fiole to be read or written. If **fname** specifies a relative path to the the target file its location is relative to the test program executable file i.e. the *testprogram\Debug* folder.

pWaveforms is a pointer to an existing `WaveformArray` used to store one or more waveforms read from the specified input file. **pWaveforms** will be resized automatically, with each member of the array consisting of one `Waveform*`. This method is typically used when the input file contains an unknown number of sound channels.

first identifies the first of a variable number of `Waveform*` arguments. Each argument represents one `Waveform*` being read from the input file. The last argument must be 0. This method is typically used when the file contains a known number of sound channels i.e. stereo. See Example 2:.

waveforms is a `WaveformArray` containing one or more `Waveform*` to be written to the output file. Two forms are available:

- `WaveformArray&` specifies an existing `WaveformArray` variable.
- `WaveformArray*` must be a pointer to an existing `WaveformArray` variable.

encoding specifies the type of encoding to use when writing the output file. See Description. Legal values are `WavefileEncoding_PCM` and `WavefileEncoding_FLOAT`.

bitwidth specifies the number of bits used to represent each sample value. Legal values are 8..32.

These functions return TRUE if the specified operation was successful, otherwise FALSE is returned.

Example

Example 1:

In the following example, tmpWF is created and then saved to the specified file..

```
Waveform* outWF = waveform_create("outWF"); // waveform_create()
// Setup waveform outWF as desired
LPCTSTR fname = "c:/WF_file_name.nwav";
BOOL OK = waveform_write_file( outWF , fname );
if( !OK ) output(" ERROR: writing waveform file %s", fname );
Waveform* inWF = waveform_create("inWF");
OK = waveform_read_file( inWF , fname );
if( !OK ) output(" ERROR: reading waveform file %s", fname );
```

Example 2:

In the following example, a stereo waveform is read from a file into two Waveform* variables. Then those two are interleveaved into a single Waveform*.

```
WAVEFORM( ding ) {
   Waveform* ding0 = waveform_create("ding0");// waveform_create()
   Waveform* ding1 = waveform_create( "ding1" );
   waveform_read_file( "ding.wav", ding0, ding1, 0);
   waveform_interleave(ding, ding0, ding1);//waveform_interleave()
   waveform_destroy( ding0 );// waveform_destroy()
   waveform_destroy( ding1 );
}
```

3.27.15 waveform_fetch(), waveform_send()

Description

The waveform_send() function is used to send a waveform to a specified site.

The waveform_fetch() function is used to get a waveform from a specified site.

Note the following:

- These functions are used to transfer a waveform between sites.
- A `Waveform` of the same name must exist on both the source and destination sites.
- All existing attributes of the waveform on the destination site will be replaced (over-written, lost, etc.).
- Site numbering is the same as used with `remote_set()`, etc. i.e. Host = 0, site-1 = 1, etc. See User Tools for information about tool site numbering.
- It is not legal to specify site -1 (UI) as the source or destination site.

Usage

```
void waveform_fetch( Waveform* wave, int site );
void waveform_send( Waveform* wave, int site );
```

where:

`wave` identifies the waveform to be sent or fetched.

`site` specifies the source or destination site.

Example

The following examples assume that `waveform_create()` was executed as shown in both the Host process and on site-2:

```
Waveform* wf1 = waveform_create();
if( SiteNum() == 2) waveform_fetch( wf1, 0 );// Get from Host
if( OnHost() ) waveform_send( wf1, 2 );      // Send to Site-2
```

3.27.16 Waveform Name, Date, Type and Version Information

See Waveform Overview.

Description

All waveforms (`Waveform*`) have name, date, version, and typename attributes automatically defined as the waveform is created. These attributes are used by the system software for various needs but can also be accessed by user code using:

- `waveform_dump()`
- `waveform_get_date(), waveform_set_date()`

- waveform_get_name()
- waveform_get_typename()
- waveform_get_version()
- Waveform Set/Get X/Y Units Functions
- reciprocal()

3.27.16.1 waveform_dump()

See Waveform Overview.

Description

The waveform_dump() function is used to dump the contents of a specified waveform into the controller output window.

Usage

```
void waveform_dump( Waveform* obj );
```

where:

obj identifies the target waveform.

Example

???

3.27.16.2 waveform_get_date(), waveform_set_date()

See Waveform Overview.

Description

Each waveform has a date attribute which can be retrieved using waveform_get_date() or modified using waveform_set_date().

A waveform's date is set when the waveform is:

- Created, using the WAVEFORM() macro or waveform_create()

- Loaded from disk using `waveform_read_file()`. The `Waveform*` date is set to the modification date of the file.
- `waveform_copy()` transfers the date of the source `Waveform*` to the copy.

Usage

```
CTime waveform_get_date( Waveform* obj );
void waveform_set_date( Waveform* obj, const CTime& date );
```

where:

`obj` is a pointer to the target waveform.

`date` is a a pointer to an existing `CTime` variable previously initialized with the desired date/time information.

`waveform_get_date()` returns date/time information in the form of a `CTime` structure. `CTime` can be researched using the Developer Studio on-line documentation.

Example

```
Waveform* inWF = waveform_create("inWF"); // waveform_create()
LPCTSTR fname = "c:/WF_file_name.nwav";
BOOL OK = waveform_read_file( inWF , fname );
if( !OK ) output(" ERROR: reading waveform file %s", fname );
CTime time = waveform_get_date( inWF );
output("  date => %d/%d/%d",  time.GetDay(),
                              time.GetMonth(),
                              time.GetYear() );
waveform_destroy( inWF );
```

3.27.16.3 waveform_get_name()

Description

Each waveform has an optional name attribute which can be retrieved using `waveform_get_name()`. Note the following:

A waveform's name is set when:

- A waveform's name is set when the waveform is created using the `WAVEFORM()` macro

- A waveform's name is optionally set when the waveform is created using the `waveform_create()` function.

Usage

```
CString waveform_get_name( Waveform* obj );
```

where:

`obj` is a pointer to the target waveform.

`waveform_get_name()` returns a specified waveform's name as a `CString`. An empty string is returned if a name was not specified when creating the waveform.

Example

In the following example, since the optional name argument was not specified when `waveform_create()` is called `waveform_get_name()` returns an empty string.

```
Waveform* wf = waveform_create();
CString name = waveform_get_name( wf );
output(" name => %s", name );
waveform_destroy( wf ); // waveform_destroy()
```

3.27.16.4 waveform_get_typename()

See Waveform Overview.

Description

A waveform's sample values are defined using one of the Waveform Sample Value Notations. The `waveform_get_typename()` function can be used to get the notation used to define a given waveform's sample values.

The value returned by `waveform_get_typename()` is a `CString`. The following definitions can be used to evaluate the returned value:

```
#define BAD_WAVE   "BadWave"
#define RRECT_WAVE "RRectWave"
```

```
#define CRECT_WAVE "CRectWave"
#define POLAR_WAVE "PolarWave"
#define RLONG_WAVE "RLongWave"
```

BAD_WAVE is returned for any Waveform* which exists but has not been defined, or which has invalid sample values based on an error detected by the system software.

Usage

```
CString waveform_get_typename( Waveform* obj );
```

where:

obj is a pointer to the target waveform.

waveform_get_typename() returns values in the form of a CString. See Description for returned value options. Also see Waveform Sample Value Notations.

Example

```
CString tname = waveform_get_typename( wf );
output("  Waveform Sample Notation => %s", tname );
```

3.27.16.5 waveform_get_version()

See Waveform Overview, Waveform Name, Date, Type and Version Information

Description

The waveform library has a version attribute, targeted at allowing the system software to adapt to possible future changes to the waveform software. The version attribute can be retrieved by user code using the waveform_get_version().

Usage

```
CString waveform_get_version();
```

where:

waveform_get_version() returns values in the form of a CString. The first version was: "000.000".

Example

```
CString vers  = waveform_get_version();
output("  vers => %s", vers );
```

3.27.16.6 Waveform Set/Get X/Y Units Functions

See Waveform Overview.

Description

The `waveform_set_y_units()` function is used to set the Y_units value for a specified waveform.

The `waveform_get_y_units()` function is used to get the Y_units value of a specified waveform.

The `waveform_get_x_units()` function is used to get the X_units value of a specified waveform.

See Waveform Units for a description of why *units* are required, how they are used, etc. The functions documented here are used to set and get the units information for a specified waveform.

The Waveform Sample Programming functions and `waveform_set_x_scale()` explicitly set a waveform's X_units/Y_units values. The Waveform Generate Functions functions implicitly set a waveform's X_units/Y_units value. Many other waveform functions set or modify X_units and/or Y_units based on the operation being performed by the function. See Waveform Units and Units Applications.

Note: there is no `waveform_set_x_units()` function. To set X_units requires also setting the X_start value and X_increment value (sample period) using the Waveform Sample Programming functions or `waveform_set_x_scale()` function.

Usage

```
void waveform_set_y_units( Waveform* obj, LPCSTR yunits );
CString waveform_get_y_units( Waveform* obj );
```

```
CString waveform_get_x_units( Waveform* obj );
```

where:

obj identifies the target waveform.

yunits specifies the Y_units type of the waveform's sample values. See Waveform* Attributes, Waveform Units.

waveform_get_x_units() and waveform_get_y_units() return the currently programmed X_units or Y_units value for the specified Waveform*.

Example

```
waveform_set_y_units( wf, SCALE_VOLTS );
CString val = waveform_get_y_units( wf );
if( val == SCALE_VOLTS) { // ... do something ... }
```

3.27.16.7 reciprocal()

Description

The reciprocal() function is used to convert [waveform] frequency values, specified in MKSFrequency, to time values, specified in MKSPeriod, and vice versa.

The reciprocal() function is declared inline, which can mostly be ignored. For those interested, the inline declaration can be researched using the Developer Studio on-line documentation.

Usage

```
inline MKSPeriod reciprocal( MKSFrequency freq ) {
   return ( 1.0 HZ / freq ) S;
}
inline MKSFrequency reciprocal( MKSPeriod time ) {
   return ( 1.0 S / time ) HZ;
}
```

Example

???

3.27.17 Waveform Sample Programming

See Waveform Overview, Waveform Mathematical View, Waveform Units

The functions documented in this section are used to set (define) or get the following attributes for a specified `Waveform*` (see Waveform* Attributes).:

- Number of samples (Size)
- Sample values
- Y_units value
- X_start value
- X_increment value
- X_units value

See Waveform Overview, Waveform Mathematical View, Waveform Sample Value Notations, Waveform Units.

One or more arrays are used to store a waveform's sample values. Waveform samples defined using Waveform Sample Value Notations, thus different functions are available to support the different notations:

- The `waveform_set_rrect()` function is used to define waveform sample values using the *real rectangular* notation (`RRECT_WAVE`). The `waveform_get_rrect()` function can be used to read sample values for a waveform defined using this notation.
- The `waveform_set_rlong()` function is used to define waveform sample values using the *long real rectangular* notation (`RLONG_WAVE`). The `waveform_get_rlong()` function can be used to read sample values for a waveform defined using this notation.
- The `waveform_set_crect()` function is used to define waveform sample values using the *complex rectangular* notation (`CRECT_WAVE`). The `waveform_get_crect()` function can be used to read sample values for a waveform defined using this notation.
- The `waveform_set_polar()` function is used to define waveform sample values using the *polar* notation (`POLAR_WAVE`). The `waveform_get_polar()` function can be used to read sample values for a waveform defined using this notation.

When a given waveform is described using any of the rectangular notations each waveform sample will have a corresponding magnitude. The magnitudes of all samples of that waveform can be retrieved using the `waveform_magnitudes()` function.

Note: it is not legal to mix notations defining the sample values of a given waveform. For example, an existing waveform defined using the RRECT_WAVE notation can only be sucessfully accessed using the *_rrect* form functions (waveform_get_rrect(), etc.). In most cases, if this rule is violated a warning message will be displayed in the appropriate site output window and/or the function will return an error value. BUT, program execution will not be interrupted.

A waveform's initial X axis value (X_start) and increment value (X_increment) are specified as arguments to the various *setter* functions. These attributes can subsequently be modified or retrieved using waveform_set_x_scale().

A waveform's Size represents the number of samples specified for that waveform, and can be retrieved using the waveform_get_size() function. For generated waveforms, the waveform's Size is explicitly set or modified using various waveform functions or implicitly set based on the size of a data structure storing sample values. For captured waveforms, the waveform Size is determined by the number of samples actually captured.

3.27.17.1 waveform_set_rrect()

See Waveform Overview, Waveform Sample Programming.

Description

The waveform_set_rrect() function is used to specify attributes for a waveform defined using RRECT_WAVE notation (see Waveform* Attributes, Waveform Sample Value Notations):

Usage

```
void waveform_set_rrect( Waveform* obj,
                         int size,
                         const double *waveform,
                         LPCTSTR yunits,
                         double xstart,
                         double xincr,
                         LPCTSTR xunits );
```

```
void waveform_set_rrect( Waveform* obj,
                         const DoubleArray& waveform,
                         LPCTSTR yunits,
                         double xstart,
                         double xincr,
                         LPCTSTR xunits );
void waveform_set_rrect( Waveform* obj,
                         int size,
                         const float *waveform,
                         LPCTSTR yunits,
                         double xstart,
                         double xincr,
                         LPCTSTR xunits );
void waveform_set_rrect( Waveform* obj,
                         const FloatArray& waveform,
                         LPCTSTR yunits,
                         double xstart,
                         double xincr,
                         LPCTSTR xunits );
```

where:

obj identifies the target waveform.

size specifies the number of sample values (Size) in the waveform being defined. Does not apply when using the DoubleArray form (see **waveform**).

waveform represents an array of waveform sample values, specified in **yunits**. Two usages are available:

- **waveform** is a pointer to an existing array of double or float containing at least **size** sample values.
- **waveform** is a pointer to an existing DoubleArray or FloatArray containing sample values. All values in the DoubleArray or FloatArray are used, which sets **size** implicitly.

yunits specifies the Y_units value. See Waveform* Attributes, Waveform Units.

xstart specifies the first X axis value (X_start), in **xunits**. See Waveform* Attributes, Waveform Mathematical View.

xincr specifies the waveform's X_increment value, in **xunits**. Must be > 0.

xunits specifies the X_units value. See Waveform Units.

Example

Example 1:

This example shows the preferred method for creating a `Waveform*` i.e. using the `WAVEFORM()` macro to create a static waveform. Sample values are stored in a local array, then used to define the `simpleWave` waveform. The number of samples of `simpleWave` waveform is based on the number of values in the `samples[]` array. `simpleWave` has a sample frequency of 10MHz:

```
WAVEFORM ( simpleWave ) {
   double samples[] = {
      0.0,
      0.0627905,
      0.125333,
      0.187381,
      // ... snip: typically lots more values are needed ...
      -0.125333,
      -0.0627905,
   };
   int size = sizeof(samples)/sizeof(double);

// See waveform_set_rrect()
void waveform_set_rrect( simpleWave,
                         size,
                         samples,
                         SCALE_VOLTS,
                         0,
                         reciprocal(10 MHZ),
                         SCALE_SECONDS );
}
```

Example 2:

This example uses `waveform_create()` to dynamically create a waveform identical to that in Example 1:. This method is discouraged due to the potential for creating memory leaks:

```
double samples[] = {
   0.0,
   0.0627905,
   0.125333,
   0.187381,
   // ... snip: typically lots more values are needed ...
```

```
   -0.125333,
   -0.0627905,
};
int size = sizeof(samples)/sizeof(double);
// See waveform_create()
Waveform* simpleWave = waveform_create("simpleWave");
// See waveform_set_rrect()
void waveform_set_rrect( simpleWave,
                         size,
                         samples,
                         SCALE_VOLTS,
                         0,
                         reciprocal(10 MHZ),
                         SCALE_SECONDS );
}
```

3.27.17.2 waveform_get_rrect()

See Waveform Overview, Waveform Sample Programming.

Description

The `waveform_get_rrect()` function is used to *get* sample values from a waveform defined using the *real rectangular* notation (`RRECT_WAVE`).

Usage

```
int waveform_get_rrect( Waveform* obj, const double *waveform[] );
int waveform_get_rrect(  Waveform* obj,
                         DoubleArray *waveform,
                         int start DEFAULT_VALUE( 0 ),
                         int nElements DEFAULT_VALUE( -1 ) );
int waveform_get_rrect(  Waveform* obj,
                         FloatArray *waveform,
                         int start DEFAULT_VALUE( 0 ),
                         int nElements DEFAULT_VALUE( -1 ) );
```

where:

`obj` identifies the target waveform.

waveform represents the array used to return sample values:

- **waveform** is a pointer to an existing double pointer. This pointer will be returned pointing to the actual data array for the specified waveform i.e. the sample data is not copied.
- **waveform** is a pointer to an existing DoubleArray or FloatArray. This array will be resized as necessary, and returned containing a copy of the waveform sample values specified via **nElements** and **start**.

start is optional, and if used specifies the first sample value to return. Default is 0 = first sample value.

nElements is optional, and if used specifies the number of samples to return. Default is -1 = all samples after **start** are returned.

waveform_get_rrect() returns the number of samples actually returned. If **obj** is NULL or if the notation of the Waveform* is not RRECT_WAVE the return value is -1. If **start** + **nElements** is greater than the number of available sample values the available values are returned, an error message is output in the appropriate controller output window, and testing continues.

Example

Example 1:

Using the DoubleArray form, get 5 sample values starting with sample 7:

```
DoubleArray vals;
int count = waveform_get_rrect(rrect_WF, &vals, 5, 7 );
if( count == -1 )
   output("  ERROR: waveform_get_rrect() returned -1");
else
   for( int i = 0; i < count; i++)
      output("  RRect: vals[%d] => %0.20g", i, vals.GetAt(i) );
output("");
```

Example 2:

Using the const *double[] form, get all sample values:

```
const double* dvals;
int count = waveform_get_rrect( rrect_WF, &dvals );
if( count == -1 )
   output("  ERROR: waveform_get_rrect() returned -1");
else
```

```
   for( int i = 0; i < count; i++)
     output("  RRect: dvals[%d] => %0.20g", i, dvals[i] );
output("");
```

3.27.17.3 waveform_set_rlong()

See Waveform Overview, Waveform Sample Programming

Description

The `waveform_set_rlong()` function is used to specify attributes for a waveform defined using `RLONG_WAVE` notation (see Waveform* Attributes, Waveform Sample Value Notations).

Several versions of these functions are available, the only difference is the data type of the sample values. The diffrerent data types are intended to facilitate the transfer of waveform data to/from alternate data representations. However, normally waveform data is stored internally using double precision values, with only 54 bits used (53 bits plus sign, equivalent to two's complement 54 bit). This supports a range of values of -9,007,199,254,740,992 to +9,007,199,254,740,991. If `waveform_set_rlong()` attempts to set a value outside of this range, an underflow/overflow message is displayed in the appropriate controller window and the value is clamped. The underlying data type of `ShortArray`, `IntArray`, `LongArray`, and `Int64Array` are signed and can handle positive and negative values. The underlying data type of `ByteArray`, `WordArray`, and `DWordArray` are unsigned and cannot store a negative sample value.

Usage

```
void waveform_set_rlong( Waveform* obj,
                         int size,
                         const BYTE *waveform,
                         LPCTSTR yunits,
                         double xstart,
                         double xincr,
                         LPCTSTR xunits );
void waveform_set_rlong( Waveform* obj,
                         const ByteArray& waveform,
                         LPCTSTR yunits,
```

```
                            double xstart,
                            double xincr,
                            LPCTSTR xunits );
void waveform_set_rlong( Waveform* obj,
                            int size,
                            const short *waveform,
                            LPCTSTR yunits,
                            double xstart,
                            double xincr,
                            LPCTSTR xunits );
void waveform_set_rlong( Waveform* obj,
                            const ShortArray& waveform,
                            LPCTSTR yunits,
                            double xstart,
                            double xincr,
                            LPCTSTR xunits );
void waveform_set_rlong( Waveform* obj,
                            int size,
                            const int *waveform,
                            LPCTSTR yunits,
                            double xstart,
                            double xincr,
                            LPCTSTR xunits );
void waveform_set_rlong( Waveform* obj,
                            const IntArray& waveform,
                            LPCTSTR yunits,
                            double xstart,
                            double xincr,
                            LPCTSTR xunits );
void waveform_set_rlong( Waveform* obj,
                            int size,
                            const WORD *waveform,
                            LPCTSTR yunits,
                            double xstart,
                            double xincr,
                            LPCTSTR xunits );
void waveform_set_rlong( Waveform* obj,
                            const WordArray& waveform,
                            LPCTSTR yunits,
```

```
                            double xstart,
                            double xincr,
                            LPCTSTR xunits );
void waveform_set_rlong( Waveform* obj,
                            int size,
                            const long *waveform,
                            LPCTSTR yunits,
                            double xstart,
                            double xincr,
                            LPCTSTR xunits );
void waveform_set_rlong( Waveform* obj,
                            const LongArray& waveform,
                            LPCTSTR yunits,
                            double xstart,
                            double xincr,
                            LPCTSTR xunits );
void waveform_set_rlong( Waveform* obj,
                            int size,
                            const DWORD *waveform,
                            LPCTSTR yunits,
                            double xstart,
                            double xincr,
                            LPCTSTR xunits );
void waveform_set_rlong( Waveform* obj,
                            const DWordArray& waveform,
                            LPCTSTR yunits,
                            double xstart,
                            double xincr,
                            LPCTSTR xunits );
void waveform_set_rlong( Waveform* obj,
                            int size,
                            const __int64 *waveform,
                            LPCTSTR yunits,
                            double xstart,
                            double xincr,
                            LPCTSTR xunits );
void waveform_set_rlong( Waveform* obj,
                            const Int64Array& waveform,
                            LPCTSTR yunits,
```

```
                                double xstart,
                                double xincr,
                                LPCTSTR xunits );
```

where:

obj identifies the target waveform.

size specifies the number of sample values (Size) in the waveform being defined. Does not apply when using the LongArray or Int64Array forms (see **waveform**).

waveform represents an array of waveform sample values, specified in **yunits**. Two basic forms are available:

- In the first usage, **waveform** is a pointer to an existing array of BYTE, short, int, WORD, long, DWORD or __int64 containing at least **size** sample values.
- In the second usage, **waveform** is a pointer to one of the array types noted above. All values in the array are used, which sets **size** implicitly.

yunits specifies the Y_units value. See Waveform* Attributes, Waveform Units.

xstart specifies the first X axis value (X_start) , in **xunits**. See Waveform* Attributes, Waveform Mathematical View.

xincr specifies the waveform's X_increment value, in **xunits**. Must be > 0.

xunits specifies the X_units value. See Waveform Units.

Example

???

3.27.17.4 waveform_get_rlong()

See Waveform Overview, Waveform Sample Programming.

Description

The waveform_get_rlong() function is used to *get* sample values from a waveform defined using the *real long rectangular* notation (RLONG_WAVE).

Several versions of these functions are available, the only difference is the data type of the sample values. The diffrerent data types are intended to facilitate the transfer of waveform data to/from alternate data representations. However, normally waveform data is stored

internally using double precision values, with only 54 bits used (53 bits plus sign, equivalent to two's complement 54 bit). This supports a range of values of -9,007,199,254,740,992 to +9,007,199,254,740,991. If `waveform_get_rlong()` attempts to retrieve a value outside of this range, an underflow/overflow message is displayed in the appropriate controller window and the value is clamped.

`waveform_get_rlong()` will also complain when a value is too large to be placed into the requested data type. The underlying data type of `ShortArray`, `IntArray`, `LongArray`, and `Int64Array` are signed and can handle positive and negative values. The underlying data type of `ByteArray`, `WordArray`, and `DWordArray` are unsigned and will produce an underflow warning if the retrieval of a negative value is attempted. All forms of underflow/ overflow will result in the value being clamped to the closest value supported by the destination data type.

Usage

```
int waveform_get_rlong(Waveform* obj,
                       ByteArray *waveform,
                       int start DEFAULT_VALUE( 0 ),
                       int nElements DEFAULT_VALUE( -1 ) );
int waveform_get_rlong(Waveform* obj,
                       ShortArray *waveform,
                       int start DEFAULT_VALUE( 0 ),
                       int nElements DEFAULT_VALUE( -1 ) );
int waveform_get_rlong(Waveform* obj,
                       IntArray *waveform,
                       int start DEFAULT_VALUE( 0 ),
                       int nElements DEFAULT_VALUE( -1 ) );
int waveform_get_rlong(Waveform* obj,
                       WordArray *waveform,
                       int start DEFAULT_VALUE( 0 ),
                       int nElements DEFAULT_VALUE( -1 ) );
int waveform_get_rlong(Waveform* obj,
                       LongArray *waveform,
                       int start DEFAULT_VALUE( 0 ),
                       int nElements DEFAULT_VALUE( -1 ) );
int waveform_get_rlong(Waveform* obj,
                       DWordArray *waveform,
                       int start DEFAULT_VALUE( 0 ),
                       int nElements DEFAULT_VALUE( -1 ) );
```

```
int waveform_get_rlong(Waveform* obj,
                       Int64Array *waveform,
                       int start DEFAULT_VALUE( 0 ),
                       int nElements DEFAULT_VALUE( -1 ) );
```

where:

obj identifies the target waveform.

waveform is a pointer to an existing array, of one of the types noted above, used to return sample values. The array will be resized as necessary, and return containing a copy of the waveform sample values specified via **nElements** and **start**.

start is optional, and if used specifies the first sample value to return. Default is 0 = first sample value.

nElements is optional, and if used specifies the number of samples to return. Default is -1 = all samples after **start** are returned.

`waveform_get_rlong()` returns the number of samples actually returned. If **obj** is `NULL` or if the notation used in defining the `Waveform*` is not `RLONG_WAVE` the return value is -1. If **start** + **nElements** is greater than the number of available sample values the available values are returned, an error message is output in the appropriate controller output window, and testing continues.

Example

???

3.27.17.5 waveform_set_crect()

See Waveform Overview, Waveform Sample Programming

Description

The `waveform_set_crect()` function is used to specify attributes for a waveform defined using `CRECT_WAVE` notation (see Waveform* Attributes, Waveform Sample Value Notations):

Two sample value arrays are used to store the two-part sample values. One array stores the real components, the other array stores the imaginary components. These arrays must be the same Size. If not, the shorter array determines the size of the waveform. When size is explicitly set, if the shorter array is smaller than size the undefined values are set to 0.

In use, each waveform sample value is comprised of the two parts:

$a + bi$ where $i = \sqrt{-1}$

Or, in the context of the function arguments

$\text{real}Part[n] + \text{imag}inaryPart[n] \times i$ where $i = \sqrt{-1}$ and [*n*] is the position in the array

Usage

```
void waveform_set_crect( Waveform* obj,
                         int size,
                         const double *realPart,
                         const double *imaginaryPart,
                         LPCTSTR yunits,
                         double xstart,
                         double xincr,
                         LPCTSTR xunits );
void waveform_set_crect( Waveform* obj,
                         const DoubleArray& realPart,
                         const DoubleArray& imaginaryPart,
                         LPCTSTR yunits,
                         double xstart,
                         double xincr,
                         LPCTSTR xunits );
void waveform_set_crect( Waveform* obj,
                         int size,
                         const float *realPart,
                         const float *imaginaryPart,
                         LPCTSTR yunits,
                         double xstart,
                         double xincr,
                         LPCTSTR xunits );
void waveform_set_crect( Waveform* obj,
                         const FloatArray& realPart,
                         const FloatArray& imaginaryPart,
                         LPCTSTR yunits,
                         double xstart,
                         double xincr,
                         LPCTSTR xunits );
```

where:

`obj` identifies the target waveform.

size specifies the number of sample values (Size) in the waveform being defined. Does not apply when using the DoubleArray or FloatArray form (see **waveform**).

realPart represents an array of waveform sample values. This array stores the real component of the two-part notation (see Waveform Sample Value Notations). Two usages are available:

- In the first usage, **waveform** is a pointer to an existing array of double or float containing at least **size** sample values.
- In the second usage, **waveform** is a pointer to an existing DoubleArray or FloatArray containing sample values. All values in the DoubleArray or FloatArray are used, which sets **size** implicitly.

imaginaryPart represents an array of waveform sample values. This array stores the imaginary component of the two-part notation (see Waveform Sample Value Notations). Two usages are available:

- In the first usage, **waveform** is a pointer to an existing array of double or float containing at least **size** sample values.
- In the second usage, **waveform** is a pointer to an existing DoubleArray or FloatArray containing sample values. All values in the DoubleArray or FloatArray are used, which sets **size** implicitly.

yunits specifies the Y_units value. See Waveform* Attributes, Waveform Units.

xstart specifies the first X axis value (X_start), in **xunits**. See Waveform* Attributes, Waveform Mathematical View.

xincr specifies the waveform's X_increment value, in **xunits**. Must be > 0.

xunits specifies the X_units value. See Waveform Units.

Example

???

3.27.17.6 waveform_get_crect()

See Waveform Overview, Waveform Sample Programming.

Description

The `waveform_get_crect()` function is used to *get* sample values from a waveform defined using the *complex rectangular* notation (`CRECT_WAVE`).

Usage

```
int waveform_get_crect(  Waveform* obj,
                         const double *realPart[],
                         const double *imaginaryPart[] );
int waveform_get_crect(  Waveform* obj,
                         DoubleArray *realPart,
                         DoubleArray *imaginaryPart,
                         int start DEFAULT_VALUE( 0 ),
                         int nElements DEFAULT_VALUE( -1 ) );
int waveform_get_crect(  Waveform* obj,
                         FloatArray *realPart,
                         FloatArray *imaginaryPart,
                         int start DEFAULT_VALUE( 0 ),
                         int nElements DEFAULT_VALUE( -1 ) );
```

where:

`obj` identifies the target waveform.

`realPart` and **`imaginaryPart`** represent two arrays used to return sample values:

- **`realPart`** and **`imaginaryPart`** are the addresses of two existing `double` pointers. These pointers will be returned pointing to the actual data arrays for the specified waveform i.e. the sample data is not copied.
- **`realPart`** and **`imaginaryPart`** are the addresses of two existing `DoubleArray` or `FloatArray`. These arrays will be resized as necessary, and return containing a copy of the waveform sample values specified via **`nElements`** and **`start`**.

`start` is optional, and if used specifies the first sample value to return. Default is 0 = first sample value.

`nElements` is optional, and if used specifies the number of samples to return. Default is -1 = all samples after **`start`** are returned.

`waveform_get_crect()` returns the number of samples actually returned. If **`obj`** is `NULL` or if the notation of the `Waveform*` is not `DoubleArray` the return value is -1.
If **`start`** + **`nElements`** is greater than the number of available sample values the available

values are returned, an error message is output in the appropriate controller output window, and testing continues.

Example

???

3.27.17.7 waveform_set_polar()

See Waveform Overview, Waveform Sample Programming, Waveform Units

Description

The `waveform_set_polar()` function is used to specify attributes for a waveform defined using `POLAR_WAVE` notation (see Waveform* Attributes, Waveform Sample Value Notations).

Two sample value arrays are used to store the two-part polar values. One array stores the magnitude components, the other array stores the phase components. These arrays must be the same Size. If not, the shorter array determines the size of the waveform. When size is explicitly set if the shorter array is smaller than size the undefined values are set to 0.

Usage

```
void waveform_set_polar( Waveform* obj,
                         int size,
                         const double *magnitude,
                         const double *phase,
                         LPCTSTR yunits,
                         double xstart,
                         double xincr,
                         LPCTSTR xunits );
void waveform_set_polar( Waveform* obj,
                         const DoubleArray& magnitude,
                         const DoubleArray& phase,
                         LPCTSTR yunits,
                         double xstart,
                         double xincr,
                         LPCTSTR xunits );
```

```
void waveform_set_polar( Waveform* obj,
                         int size,
                         const float *magnitude,
                         const float *phase,
                         LPCTSTR yunits,
                         double xstart,
                         double xincr,
                         LPCTSTR xunits );
void waveform_set_polar( Waveform* obj,
                         const FloatArray& magnitude,
                         const FloatArray& phase,
                         LPCTSTR yunits,
                         double xstart,
                         double xincr,
                         LPCTSTR xunits );
```

where:

obj identifies the target waveform.

size specifies the number of sample values (Size) in the waveform being defined. Does not apply when using the DoubleArray form (see **waveform**).

magnitude represents an array of waveform sample values. This array stores the magnitude component of the two-part notation (see Waveform Sample Value Notations). Two basic forms are available:

- In the first usage, **waveform** is a pointer to an existing array of double or float containing at least **size** sample values.
- In the second usage, **waveform** is a pointer to an existing DoubleArray or FloatArray containing sample values. All values in the DoubleArray or FloatArray are used, which sets **size** implicitly.

phase represents an array of waveform sample values. This array stores the phase component of the two-part notation (see Waveform Sample Value Notations). Two basic forms are available:

- In the first usage, **waveform** is a pointer to an existing array of double or float containing at least **size** sample values.
- In the second usage, **waveform** is a pointer to an existing DoubleArray or FloatArray containing sample values. All values in the DoubleArray or FloatArray are used, which sets **size** implicitly.

yunits specifies the Y_units value. See Waveform* Attributes, Waveform Units.

xstart specifies the first X axis value (X_start) , in **xunits**. See Waveform* Attributes, Waveform Mathematical View.

xincr specifies the waveform's X_increment value, in **xunits**. Must be > 0.

xunits specifies the X_units value. See Waveform Units.

Example

???

3.27.17.8 waveform_get_polar()

See Waveform Overview, Waveform Sample Programming.

Description

The `waveform_get_polar()` function is used to *get* sample values from a waveform defined using the *polar* notation (`POLAR_WAVE`).

Usage

```
int waveform_get_polar(  Waveform* obj,
                         const double *magnitude[],
                         const double *phase[] );
int waveform_get_polar(  Waveform* obj,
                         DoubleArray *magnitude,
                         DoubleArray *phase,
                         int start DEFAULT_VALUE( 0 ),
                         int nElements DEFAULT_VALUE( -1 ) );
int waveform_get_polar(  Waveform* obj,
                         FloatArray *magnitude,
                         FloatArray *phase,
                         int start DEFAULT_VALUE( 0 ),
                         int nElements DEFAULT_VALUE( -1 ) );
```

where:

obj identifies the target waveform.

magnitude and **phase** represent two arrays used to return sample values:

- **magnitude** and **phase** are the addresses of two existing double pointers. These pointers will be returned pointing to the actual data arrays for the specified waveform i.e. the sample data is not copied.
- **magnitude** and **phase** are the addresses of two existing DoubleArray or FloatArray. These arrays will be resized as necessary, and return containing a copy of the waveform sample values specified via **nElements** and **start**.

start is optional, and if used specifies the first sample value to return. Default is 0 = first sample value.

nElements is optional, and if used specifies the number of samples to return. Default is -1 = all samples after **start** are returned.

waveform_get_polar() returns the number of samples actually returned. If **obj** is NULL or if the notation of the Waveform* is not DoubleArray the return value is -1.
If **start** + **nElements** is greater than the number of available sample values the available values are returned, an error message is output in the appropriate controller output window, and testing continues.

Example

???

3.27.17.9 waveform_get_x_start()

See Waveform Overview

Description

The waveform_get_x_start() function is used to get the initial X axis value for a specified waveform. See Waveform Mathematical View for details.

The Waveform Sample Programming functions and waveform_set_x_scale() explicitly set a waveform's X_start value. The Waveform Generate Functions functions implicitly set a waveform's X_start value.

Usage

```
double waveform_get_x_start( Waveform* obj );
```

where:

obj identifies the target waveform.

`waveform_get_x_start()` returns the current value of X_start, in X_units (see `waveform_get_x_units()`).

Example

???

3.27.17.10 waveform_get_x_increment()

See Waveform Overview

Description

The `waveform_get_x_increment()` function is used to get the X_increment value for a specified waveform. See Waveform* Attributes, Waveform Mathematical View for details.

The Waveform Sample Programming functions and `waveform_set_x_scale()` set the X_increment value.

The Waveform Sample Programming functions and `waveform_set_x_scale()` explicitly set the X_increment value. The Waveform Generate Functions functions implicitly set the X_increment value.

Usage

```
double waveform_get_x_increment( Waveform* obj );
```

where:

`obj` identifies the target waveform.

`waveform_get_x_increment()` returns the X_increment value for the specified waveform, in X_units (see `waveform_get_x_units()`).

Example

???

3.27.17.11 waveform_set_x_scale()

See Waveform Overview

Description

The `waveform_set_x_scale()` function is used to set the following X axis parameters for a specified waveform:

- X_start: the initial X axis value, in X_units
- X_increment: the sample increment for X axis values, in X_units
- X_units: the units in which the X axis values are represented

See Waveform* Attributes, Waveform Mathematical View and Waveform Units for details of these parameters, how they are used, etc.

The Waveform Sample Programming functions and `waveform_set_x_scale()` explicitly set these waveform attributes. The Waveform Generate Functions functions implicitly set these waveform attributes.

The `waveform_get_x_start()` function is used to get the currently programmed X_start value.

The `waveform_get_x_increment()` function is used to get the currently programmed X_increment value.

The `waveform_get_x_units()` function is used to get the currently programmed X X_units value.

Usage

```
void waveform_set_x_scale(  Waveform* obj,
                            double xstart,
                            double xincr,
                            LPCSTR xunits );
```

where:

`obj` identifies the target waveform.

`xstart` specifies the first X axis value (X_start), in **`xunits`**. See Waveform* Attributes, Waveform Mathematical View.

`xincr` specifies the waveform's X_increment value, in **`xunits`**. Must be > 0.

`xunits` specifies the X_units value. See Waveform Units.

Example

???

3.27.17.12 waveform_get_size()

See Waveform Overview, Waveform Sample Programming.

Description

The `waveform_get_size()` function is used to return the number of sample values (Size) of a specified waveform.

For waveforms defined using two-part values (`CRECT_WAVE`, `POLAR_WAVE`) the number of samples is the same as if the waveform was defined using a one-part value (`RLONG_WAVE`, `RRECT_WAVE`).

Usage

```
int waveform_get_size( Waveform* obj );
```

where:

`obj` identifies the `Waveform*` of interest.

`waveform_get_size()` returns the number of samples (Size) currently defined for **`obj`**.

Example

???

3.27.17.13 waveform_get_element(), waveform_set_element()

See Waveform Overview, Waveform Sample Programming.

Description

The `waveform_get_element()` function is used to retrieve one sample value from a specified waveform.

The `waveform_set_element()` function is used to modify one sample value for a specified waveform.

Different prototypes are defined for accessing waveforms defined using one-part vs. two-part notations (see below).

Usage

The following functions are used with waveforms defined using a one-part notation (RRECT_WAVE, RLONG_WAVE):

```
double waveform_get_element( Waveform* obj, int element_index );
void waveform_set_element(  Waveform* obj,
                            int element_index,
                            double scalar );
```

The following functions are used with waveforms defined using a two -part notation (CRECT_WAVE, POLAR_WAVE):

```
void waveform_get_element(  Waveform* obj,
                            int element_index,
                            double *scalar1,
                            double *scalar2 );
void waveform_set_element(  Waveform* obj,
                            int element_index,
                            double scalar1,
                            double scalar2 );
```

where:

obj identifies the target waveform.

element_index identifies the zero based sample value to be accessed. For the *getter* functions, if index exceeds the sample Size of the waveform (see waveform_get_size()) an error message is displayed and the return value(s) = 0.0. For the *setter* functions, if index exceeds the sample Size of the waveform the waveform is enlarged to accommodate the new value and intervening samples are set = 0.

scalar1 and **scalar2** are used to identify a two-part sample value. **scalar1** corresponds to the *magnitude* value of a POLAR_WAVE sample, or the *real* value of a CRECT_WAVE sample. **scalar2** corresponds to the *phase* value of a POLAR_WAVE sample, or the *imaginary* value of a CRECT_WAVE sample. In the *set* function, **scalar1** and **scalar2** are simple double values or variables. In the *get* function, **scalar1** and **scalar2** are pointers to existing double variables used to return the two-part value.

scalar specifies the sample value to be set in **obj**.

In the first form above, waveform_get_element() returns a one-part sample value.

Example

???

3.27.17.14 waveform_set_signal_spread(), waveform_get_signal_spread()

See Waveform Overview, Waveform Sample Programming.

Description

The `waveform_set_signal_spread()` function is used to specify the signal spread value for a user defined window coefficient waveform. See Waveform Window Functions for details of how a coefficient waveform is used to apply windowing to an existing waveform.

The `waveform_get_signal_spread()` function may be used to get the signal spread value for a specified window coefficient waveform.

When *windowing* is applied to a time domain waveform before an FFT is performed, the signals and harmonics get *spread* into more than one FFT bin. The amount of spread is determined by the windowing coefficients applied. See Waveform Window Functions.

Windowing is normally employed when a signal is captured under noncoherent conditions. This prevents the signal from spreading completely throughout all of the FFT bins.

The *signal spread* is defined to be the number of bins on both sides of a signal bin to include when calculating signal strengths (i.e. SNR, SINAD, THD, etc.). This value is normally zero.

When a set of window coefficients is generated, the corresponding signal spread associated with that window is recorded within the coefficient waveform. When the window is applied to a waveform, the signal spread information is also applied.

Functions such as `waveform_sinad()`, `waveform_snr()`, `waveform_thd()`, etc. comprehend windowing and will utilize the signal spread value when determining signal strengths. Signal spread handling is therefore automatic, and only needs explicit attention if the user needs to generate and apply a custom window type.

The built-in window types have the following spread values:

Window Type	Signal Spread
Triangle	1 See Waveform Windowing Coefficient Functions.
Hamming	1 See Waveform Windowing Coefficient Functions.
Hanning	1 See Waveform Windowing Coefficient Functions.
Blackman	2 See Waveform Windowing Coefficient Functions.
Blackman-Harris	3 See Waveform Windowing Coefficient Functions.
Dolph-Chebyshev	Variable, computed from alpha value. See waveform_dolph_chebyshev_window_coefficients()

Usage

```
void waveform_set_signal_spread( Waveform* obj, int spread );
int waveform_get_signal_spread( Waveform* obj );
```

where:

obj identifies the target waveform.

spread ???

`waveform_get_signal_spread()` the current signal spread value for a specified waveform.

Example

???

3.27.17.15 waveform_zero_pad()

See Waveform Overview, Waveform Sample Programming.

Description

The `waveform_zero_pad()` function is used to append zeroes to the end of the given waveform i.e. append sample values, with the value 0, to the end of a specifed waveform. Note the following:

- The waveform to be padded can be defined using the following notations: `RRECT_WAVE`, `RLONG_WAVE`, `CRECT_WAVE`, or `POLAR_WAVE`. The output waveform will use the same notation.
- The `length` parameter represents the number of sample values to be appended. `length` must be a non-negative value.

Usage

```
void waveform_zero_pad(Waveform* out_wave,
                       Waveform* in_wave,
                       int length );
```

where:

`out_wave` specifies the output waveform.

`in_wave` identifies the input waveform.

`length` specifies how many sample values to append.

Example

???

3.27.18 Waveform Manipulation Functions

See Waveform Overview.

The following functions are used to modify or otherwise manipulate an existing waveform. Also see Waveform Analysis Functions:

- `waveform_absolute_value()`
- `waveform_add()`
- `waveform_clamp()`
- `waveform_concat()`
- `waveform_copy()`

- waveform_decimate()
- waveform_differencing()
- waveform_divide()
- waveform_double_strided_copy()
- waveform_integerize()
- waveform_join_complex()
- waveform_join_polar()
- waveform_lookup()
- waveform_make_complex()
- waveform_multiply()
- waveform_negate()
- waveform_polar_to_rectangular()
- waveform_reciprocal()
- waveform_rectangular_to_polar()
- waveform_replace_subset()
- waveform_resample()
- waveform_rescale()
- waveform_reverse()
- waveform_rotate_left(), waveform_rotate_right()
- waveform_sort()
- waveform_split()
- waveform_strided_copy()
- waveform_subset()
- waveform_subtract()
- waveform_sum()
- waveform_summing()

3.27.18.1 waveform_absolute_value()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_absolute_value()` function is used to convert each sample of an input waveform into its equivalent absolute value.

The mathematical operation used for each waveform Type is shown below (see Waveform* Attributes, Waveform Sample Value Notations and Waveform Mathematical View):

RRECT_WAVE	f(x) = |x|
CRECT_WAVE	f(x + yi) = sqrt(x^2 + y^2) + 0i
POLAR_WAVE	f(r, theta) = |r|, 0
RLONG_WAVE	f(x) = |x| (same as RRECT_WAVE)

Note the following about the output waveform:

- X_units, Y_units, X_start, and X_increment are set to match the input waveform

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_absolute_value( Waveform* out_wave,
                              Waveform* in_wave );
```

where:

`out_wave` is the output waveform:

`in_wave` is the input waveform to be processed.

Example

???

3.27.18.2 waveform_add()

See Waveform Overview, Waveform Manipulation Functions.

Description

The waveform_add() function is used to modify an existing waveform by adding a value to each sample of a specified waveform.

Two basic forms are provided:

- Each sample of the output waveform consists of the addition of the two corresponding samples of two input waveforms.
- Each sample of the output waveform consists of the addition of a corresponding sample of an input waveform plus a scalar value (RRECT_WAVE, RLONG_WAVE) or two scaler values (CRECT_WAVE, POLAR_WAVE).

Also note:

- If both inputs are RLONG_WAVE, the output will be RLONG_WAVE.
- If both inputs are RRECT_WAVE, the output will be RRECT_WAVE.
- If one input is RRECT_WAVE and the other is RLONG_WAVE, the output will be RRECT_WAVE.
- If both inputs are CRECT_WAVE, the output will be CRECT_WAVE:
  real = real1 + real2
  imag = imag1 + imag2;
- If both inputs are POLAR_WAVE, vector addition is performed. Both input waveforms are converted to rectangular notation, summed, then converted back to polar notation.
- If one input is CRECT_WAVE, the other input waveform is converted to complex using waveform_make_complex(). The results are summed and left in complex form (CRECT_WAVE). waveform_make_complex() appends an imaginary value of zero to RRECT_WAVE and RLONG_WAVE waveforms. For POLAR_WAVE waveforms, waveform_polar_to_rectangular() is called.
- When adding a waveform and a single scalar:
  - If the input waveform is RRECT_WAVE, the output is RRECT_WAVE.
  - If the input waveform is RLONG_WAVE, the output remains RLONG_WAVE if the scalar is an integer value otherwise the output is RRECT_WAVE.
  - If the input waveform is CRECT_WAVE, the results are formed by:
    real = real1 + scalar
    imag = imag1
  - If the input is POLAR_WAVE, the scalar is promoted to the vector (scalar, 0.0) and vector addition is performed.

- When adding a waveform and two scalars, the input waveform type must be either CRECT_WAVE or POLAR_WAVE. The scalars are intrepeted to represent a tuple of the same type as the input wave type and the addition is performed.

In all cases, addition is performed on a per-sample basis (see Waveform Sample Programming). The following rules apply:

- When two waveforms are being added they must have the sample sample Size. If this rule is violated the output waveform will be the size of the smaller input waveform, a warning message is output in the appropriate controller output window, and testing otherwise continues.
- The output waveform will be [re-]sized to contain only the samples resulting from the addition. This makes sense when the output Waveform* initially contains no sample data, but also applies when an existing Waveform*, which does contain sample values, is redefined (clobbered).

Also note the following about the output waveform:

- X_units, Y_units, X_start, and X_increment are set to match the first input waveform. See Waveform* Attributes, Waveform Mathematical View and Waveform Units for descriptions of these parameters.
- BAD_WAVE propegates i.e. if any input waveform is BAD_WAVE, the output waveform will be BAD_WAVE.

Usage

The following function adds two waveforms, sample-by-sample:

```
void waveform_add(  Waveform* out_wave,
                    Waveform* in_wave1,
                    Waveform* in_wave2 );
```

The following function is only usable with waveforms defined using a one-part notation (RRECT_WAVE, RLONG_WAVE):

```
void waveform_add(  Waveform* out_wave,
                    Waveform* in_wave,
                    double scalar );
```

The following function is only usable with waveforms defined using a two-part notation (CRECT_WAVE, POLAR_WAVE):

```
                    Waveform* in_wave,
                    double scalar1,
                    double scalar2 );
```

where:

out_wave is the output waveform which results from the addition:

in_wave1 and **in_wave2** identify two waveforms which are to be added.
out_wave = **in_wave1** + **in_wave2**.

in_wave identifies one waveform which will have scalar value(s) addition to each sample.
out_wave = **in_wave1** + **scalar** (RRECT_WAVE, RLONG_WAVE) or
out_wave = **in_wave1** + **scalar1** and **scalar2** (CRECT_WAVE, POLAR_WAVE). See Description.

scalar, **scalar1**, and **scalar2** are scalar values to be added to each sample of the input waveform. See Description.

Example

???

3.27.18.3 waveform_clamp()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_clamp()` function is used to modify waveform sample values. Note the following:

- Only valid for RRECT_WAVE and RRECT_WAVE waveforms. See Waveform Sample Value Notations.
- `waveform_clamp()` sets (clamps) values less than `min_value` to `min_value`. Values greater than `max_value` will be set to `max_value`.
- If the input waveform is RLONG_WAVE, the output waveform remains RLONG_WAVE only if `min_value` and `min_value` are both integers, otherwise the output waveform is converted to RRECT_WAVE.

Note the following about the output waveform:

- X_units, Y_units, X_start, and X_increment are set to match the input waveform

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Note that the sequence:

```
waveform_clip_lower( out, in, min_value ); //waveform_clip_lower()
wavefrom_clip_upper( out, out, max_value );//waveform_clip_upper()
```

results in the same output as:

```
waveform_clamp( out, in, min_value, max_value );
```

Usage

```
void waveform_clamp(Waveform* out_wave,
                    Waveform* in_wave,
                    double min_value,
                    double max_value );
```

where:

`out_wave` identifies the output waveform.

`in_wave` identifies the input waveform.

`min_value` identifies the minimum clip value. See Description.

`max_value` identifies the maximum clip value. See Description.

Example

???

3.27.18.4 waveform_concat()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_concat()` function is used to concatenate two waveforms into a single waveform. This has the effect of appending the sample values for one waveform to the end of another waveform, resulting in a waveform consisting of the combined samples.

Note the following about the output waveform:

- Except as noted next, both input waveforms must be of the same type (`RRECT_WAVE`, `CRECT_WAVE`, etc.). The output waveform will be the same type.

- RRECT_WAVE and RLONG_WAVE waveforms can be concatenated. The output waveform will be RRECT_WAVE.
- X_units, Y_units, X_start, and X_increment are set to match the first input waveform

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_concat(  Waveform* out_wave,
                       Waveform* in_wave1,
                       Waveform* in_wave2 );
```

where:

in_wave1 and **in_wave2** are the two waveforms to be concatenated. The result is placed in **out_wave** with the sample values of **in_wave1** occuring before those from **in_wave2**.

Example

???

3.27.18.5 waveform_copy()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_copy()` function is used to make an identical copy of an existing waveform, often as the basis for creating a new waveform by modifying the copy.

Note the following about the output waveform:

- X_units, Y_units, X_start, and X_increment are set to match the input waveform

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_copy( Waveform* out_wave, Waveform* in_wave );
```

where:

out_wave is an existing Waveform* which becomes the new copy.

in_wave is the Waveform* to be copied.

Note: any prior attributes of the Waveform* **out_wave** argument are lost when this function is executed.

Example

In the following example, existingWF is an existing waveform being copied to tmpWF.

```
// See waveform_create()
Waveform* tmpWF = waveform_create("tmpWF");
waveform_copy( existingWF, tmpWF );
// Use tmpWF as desired
waveform_destroy( tmpWF ); // See waveform_destroy()
```

3.27.18.6 waveform_decimate()

See Waveform Overview, Waveform Manipulation Functions.

Description

The waveform_decimate() function is used to copy every n^{th} sample value from an input waveform to an output waveform, starting with a specified sample. Note the following:

- Copying begins with the sample value specified by start.
- The stride parameter specifies which samples to copy. For example, if stride = 5, every 5^{th} sample value from the input waveform will be copied to the output waveform.
- The copying stops when the end of the input waveform is reached, thus the Size of the output waveform will be:

$$floor\left(\frac{size + start}{stride}\right)$$

where `floor()` rounds-down to the nearest integer value and `size` is the number of sample values (Size) in the input waveform.

- The input waveform sample values may be defined using any notation (see Waveform Sample Value Notations). The ouput waveform sample values will be defined using the same notation.

The `waveform_deinterleave()` function can be used to *deinterleave* one waveform into two waveforms. `waveform_decimate()` is more versatile, since it can be used to deinterleave one waveform into to any number of waveforms. For example, to deinterleave one waveform to three waveforms:

```
waveform_decimate( part1WF, inWF, 0, 3 );
waveform_decimate( part2WF, inWF, 1, 3 );
waveform_decimate( part3WF, inWF, 2, 3 );
```

Also note the following about the output waveform:

- X_units and Y_units are set to match the input waveform
- X_start is set to the time associated with the first sample value.
- X_increment is set to the X_increment of the input waveform times the stride value. The rationale for this is logical. For example. If only every fourth sample is copied the effect is the same as if the waveform was sampled four times slower.

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_decimate(Waveform* out_wave,
                       Waveform* in_wave,
                       int start DEFAULT_VALUE( 0 ),
                       int stride DEFAULT_VALUE( 10 ) );
```

where:

`out_wave` identifies the output waveform.

`in_wave` identifies the input waveform.

`start` specifies the first sample value to be copied to the output waveform.

`stride` specifies which sample values to copy, beginning with **`start`**. See Description.

Example

???

3.27.18.7 waveform_differencing()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_differencing()` function returns a waveform containing sample values each of which is the difference of the corresponding input waveform sample value and the sample after it. In other words, the n^{th} element of the output waveform is equal to the $(n+1)^{th}$ element of the input waveform minus the n^{th} element of the input waveform.

Subtraction is performed appropriate to the notation used to define the sample values of the input waveform (see Waveform Sample Value Notations).

Note the following about the output waveform:

- The ouput waveform sample value notation is the same as the input waveform (see Waveform Sample Value Notations), but the waveform Size (number of sample values) is one less.
- X_units, Y_units, X_start, and X_increment are set to match the input waveform

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_differencing( Waveform* out_wave,
                            Waveform* in_wave );
```

where:

`out_wave` specifies the output waveform.

`in_wave` specifies the input waveform.

Example

???

3.27.18.8 waveform_divide()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_divide()` function is used to perform sample-by-sample division on a waveform, with the resulting values stored in an output waveform.

Two basic forms are provided:

- Each sample of the output waveform consists of the division of the two corresponding samples of two input waveforms.
- Each sample of the output waveform consists of the division of a corresponding sample of an input waveform by a scalar value (`RRECT_WAVE`, `RLONG_WAVE`) or two scaler values (`CRECT_WAVE`, `POLAR_WAVE`).

Also note:

- If both inputs are `RLONG_WAVE`, the output will be `RLONG_WAVE`.
- If both inputs are `RRECT_WAVE`, the output will be `RRECT_WAVE`.
- If one input is `RRECT_WAVE` and the other is `RLONG_WAVE`, the output will be `RRECT_WAVE`.
- If both inputs are `CRECT_WAVE`, the output will be `CRECT_WAVE` and calculated as:

$$real = \frac{(\text{real}1 \times \text{real}2 + \text{imag}1 \times \text{imag}2)}{(\text{real}2^2 + \text{imag}2^2)}$$

$$\text{imag} = \frac{(\text{real}2 \times \text{imag}1 - \text{real}1 \times \text{imag}2)}{(\text{real}2^2 + \text{imag}2^2)}$$

- If the first input is `CRECT_WAVE` and the other is an `RRECT_WAVE` or `RLONG_WAVE`, the output is `CRECT_WAVE` and calculated as:

$$real = \frac{\text{real}1}{\text{real}2}$$

$$imag = \frac{\text{imag}1}{\text{real}2}$$

- If the first input is RRECT_WAVE or RLONG_WAVE and the other is an CRECT_WAVE, the output is CRECT_WAVE and calculated as:

$$real = \frac{(\text{real}1 \times \text{real}2)}{(\text{real}2^2 + \text{imag}2^2)}$$

$$imag = -\frac{(\text{real}1 \times \text{real}2)}{(\text{real}2^2 + \text{imag}2^2)}$$

- The `waveform_divide()` function does not support division by a POLAR_WAVE value.

When performing division using two scalar value(s):

- The input waveform must be CRECT_WAVE waveform.
- The scalars are intrepeted to represent a tuple of the same type as the input wave type and the division is performed.

When performing division using one scalar value

- If the input is RRECT_WAVE, the output is RRECT_WAVE.
- If the input is RLONG_WAVE, the output remains RLONG_WAVE if the scalar is an integer value otherwise the output is RRECT_WAVE.
- If the input is CRECT_WAVE, the results are formed by:

$$real = \frac{real}{scalar}$$

$$imag = \frac{imag}{scalar}$$

- If the input is POLAR_WAVE, the results are formed by:

$$magnitude = \frac{magnitude}{scalar}$$

$$angle = angle$$

In all cases, division is performed on a per-sample basis (see Waveform Sample Programming). The following rules apply:

- When two waveforms are being divided they must have the same sample Size. If this rule is violated the output waveform will be the size of the smaller input waveform, a warning message is output in the appropriate controller output window, and testing otherwise continues.

- The output waveform will be [re-]sized to contain only the samples resulting from the division. This makes sense when the output `Waveform*` initially contains no sample data, but also applies when an existing `Waveform*`, which does contain sample values, is redefined (clobbered).
- X_units, Y_units, X_start, and X_increment of the output waveform are set to match the first input waveform. See Waveform* Attributes, Waveform Mathematical View and Waveform Units for descriptions of these parameters.
- `BAD_WAVE` propegates i.e. if any input waveform is `BAD_WAVE`, the output waveform will be `BAD_WAVE`.

Usage

The following function performs a sample-by-sample division; in_wave1 divided by in_wave2:

```
void waveform_divide(  Waveform* out_wave,
                       Waveform* in_wave1,
                       Waveform* in_wave2 );
```

The following function is only usable with waveforms defined using a one-part notation (`RRECT_WAVE`, `RLONG_WAVE`):

```
void waveform_divide(  Waveform* out_wave,
                       Waveform* in_wave,
                       double scalar );
```

The following function is only usable with `CRECT_WAVE` waveforms:

```
void waveform_divide(  Waveform* out_wave,
                       Waveform* in_wave,
                       double scalar1,
                       double scalar2 );
```

where:

`out_wave` is the output waveform which results from the division:

`in_wave1` and **`in_wave2`** identify two waveforms which are to be divided. See description.

`in_wave` identifies one waveform which will have each sample divided by scalar value(s). See Description.

`scalar`, **`scalar1`**, and **`scalar2`** are the scalar values to be divided into each sample of the input waveform. See Description.

Example

???

3.27.18.9 waveform_double_strided_copy()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_double_strided_copy()` function is used to copy sample values from an input waveform to an output waveform. Note the following:

- The output waveform is set to be the same Size as the input waveform.
- Copying begins by taking the input waveform sample value at the location (index) specified by the `input_start` argument and copying it to the output waveform at the index specified by the `output_start` argument.
- The `input_stride` parameter then identifies how values are subsequently taken from the input waveform. For example, if `input_stride` = 5, every 5th sample value from the input waveform will be copied to the output waveform.
- The `output_stride` parameter specifies how subsequent destinations are identified. For example, if `output_stride` = 7, after the first value is copied to `output_start`, subsequent values will copied into the 7th index location after the previous location. See Figure-58: .
- The copying process wraps back to the beginning of both the input and output waveforms as needed, and stops when the number of samples copied matches the Size of the input waveform.
- As with `waveform_strided_copy()`, if the `input_stride` value is not *mutually prime* with the input waveform Size (see Mutually Prime), some of the input waveform sample values will be repeated in the output waveform, and other sample values will not be copied. This is not an error.
- However, if the `output_stride` value is not mutually prime with the *input* waveform Size not all of the output waveform samples will be valid i.e. the output waveform would contain undefined values (holes). When this is detected, an error message is output in the appropriate controller window, the copy process is terminated, the output waveform will be invalid, and the waveform Type is set = `BAD_WAVE`.

- Also, the `input_stride` and `output_stride` values must be mutually prime with each other, or a similar error is issued.
- The input waveform sample values may be defined using any notation (see Waveform Sample Value Notations). The ouput waveform sample values will be defined using the same notation.

Also note the following about the output waveform:

- X_units and Y_units are set to match the input waveform
- $X\_increment = \frac{input\_stride}{output\_stride} \times X\_increment_{in}$
- $X\_start = X\_start_{in} + input\_start \times X\_increment_{in} - out\_start \times X\_increment$

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

The diagram below shows how an input waveform containing 12 sample values is copied when mutually prime rules are satisified:

```
waveform_strided_copy( outWF, inWF, 2, 5, 9, 7 );
```

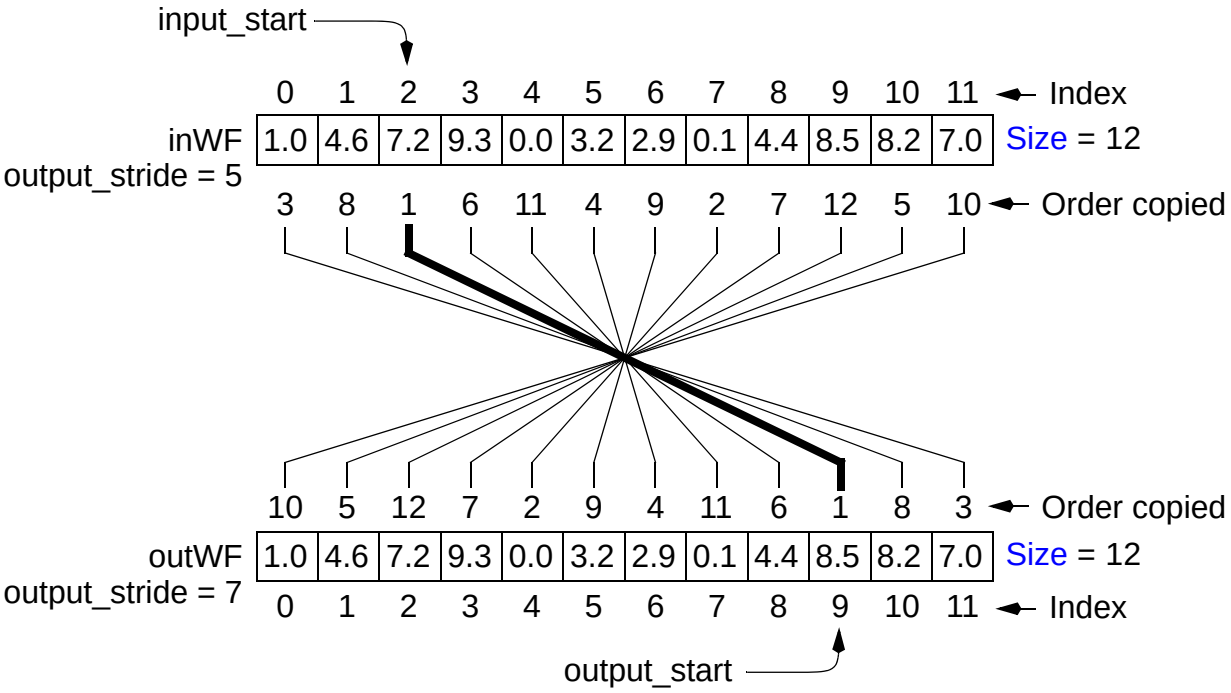


Figure-58: `waveform_double_strided_copy()` **User Model**

The `waveform_double_strided_copy()` function may be used to *undo* the effects of `waveform_strided_copy()`. For example:

```
waveform_strided_copy( outWF, inWF, start, stride );
```

may be undone with:

```
waveform_double_strided_copy( outWF, inWF, start, stride );
```

Note that this only operates correctly if `waveform_strided_copy()` performed a full copy (`num_elements` = input waveform Size) ***AND*** the `input_stride` value represented a *well-formed* value i.e. was mutually prime with the Size of the input waveform (see Mutually Prime).

This capability can be used to *unravel* a sampled waveform. For instance, given a waveform consisting of 1024 sample values which represent five cycles of a periodic signal, assuming that the waveform was coherently captured, it can be converted to a representation of a single cycle of the signal using:

```
waveform_double_strided_copy( outWF, inWF, 0, 5 );
```

This *cycle scrambling* application may extend in both directions. For example, the following can be used to to expand the same five signal cycles into seven:

```
waveform_double_strided_copy( outWF, inWF, 0, 5, 0, 7 );
```

Usage

```
void waveform_double_strided_copy(
                              Waveform* out_wave,
                              Waveform* in_wave,
                              int output_start,
                              int output_stride DEFAULT_VALUE( 1 ),
                              int input_start DEFAULT_VALUE( 0 ),
                              int input_stride DEFAULT_VALUE( 1 ) );
```

where:

`out_wave` identifies the output waveform.

`in_wave` identifies the input waveform.

`output_start` where the first value copied to the output waveform will be located. This is the zero based location (index) in the output waveform where the input waveform sample located by **`input_start`** will be copied.

output_stride is optional, and if used specifies how destination locations in the output waveform are identified after **output_start**. See Description. Default = 1 i.e. no output locations are skipped after **output_start**.

input_start is optional, and if used identifies the zero based location (index) of the first value to be copied from the input waveform. Default = 0 = copy the first value in the input waveform.

input_stride is optional, and if used specifies how input waveform samples after **input_start** are located. See Description. Default = 1 = no values are skipped.

Example

???

3.27.18.10 waveform_integerize()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_integerize()` function is used to convert the sample values of waveform defined using `RRECT_WAVE` notation to `RLONG_WAVE` notation. Conversion is done by rounding each sample value individually, using a specified rounding method. See RoundingMethod in Types, Enums, etc.

Usage

```
void waveform_integerize(
   Waveform* out_wave,
   Waveform*in_wave,
   RoundingMethod rounding_method
      DEFAULT_VALUE( t_round_to_nearest )
);
```

where:

out_wave specifies the output waveform.

in_wave specifies the waveform to be converted.

rounding_method is optional, and if used specifies the desired rounding method. See RoundingMethod in Types, Enums, etc. Default = `t_round_to_nearest`.

Example

???

3.27.18.11 waveform_join_complex()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_join_complex()` function is used to create an output waveform containing sample values defined in the `CRECT_WAVE` notation, from two input waveforms defined using a `RRECT_WAVE` or `RLONG_WAVE` notation.

It is expected that the two input waveforms are the same Size (see `waveform_get_size()`). If not, an error message is output in the appropriate controller window, sample values from the longer waveform are ignored, and testing otherwise continues.

Note the following about the output waveform:

- X_units, Y_units, X_start, and X_increment are set to match the first input waveform

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_join_complex( Waveform* out_wave,
                            Waveform* in_wave1,
                            Waveform* in_wave2 );
```

where:

`out_wave` specifies the destination for the joined waveform.

`in_wave1` and **`in_wave2`** specify the two waveforms to be joined into **`out_wave`**.

Example

???

3.27.18.12 waveform_join_polar()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_join_polar()` function is used to create an output waveform with sample values defined in the `POLAR_WAVE` notation from two inputs waveforms defined using `RRECT_WAVE` or `RLONG_WAVE` notation.

It is expected that the two input waveforms are the same Size (see `waveform_get_size()`). If not, an error message is output in the appropriate controller window, sample values from the longer waveform are ignored, and testing otherwise continues.

Note the following about the output waveform:

- X_units, Y_units, X_start, and X_increment are set to match the first input waveform

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_join_polar(Waveform* out_wave,
                         Waveform* in_wave1,
                         Waveform* in_wave2 );
```

where:

`out_wave` specifies the destination for the joined waveform.

`in_wave1` and **`in_wave2`** specify the two waveforms to be joined into **`out_wave`**.

Example

???

3.27.18.13 waveform_lookup()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_lookup()` function provides a table lookup facility using a waveform's sample values as table index values.

Note the following:

- The `lookup_table` argument may be a waveform containing sample values in any notation. Values from `in_wave` will be transferred to `out_wave` as noted below,
- The `in_wave` waveform Type must be `RLONG_WAVE`. Each sample value of `in_wave` is interpreted as an index into the `lookup_table`. The `in_wave` sample values must be from zero up to the Size of the `lookup_table` minus one.
- As each element of the input waveform (`in_wave`) is processed, the corresponding sample value from the `lookup_table` waveform is placed into the `out_wave` waveform. Output waveform sample values will be the same notation as those in the `lookup_table`, and `out_wave` will be the same Size as the `in_wave` waveform.

Description

```
void waveform_lookup(  Waveform* out_wave,
                       Waveform* in_wave,
                       Waveform* lookup_table );
```

where:

`out_wave` specifies the output waveform.

`in_wave` specifies a set of index values, in the form of a waveform sample values, which determine which sample values are taken from **`lookup_table`** to define **`out_wave`**.

`lookup_table` specifies the waveform from which sample values will be taken to define **`out_wave`**.

Example

???

3.27.18.14 waveform_make_complex()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_make_complex()` function is used to convert a waveform defined using other notations to complex values. See Waveform Sample Value Notations.

Note the following:

- The output waveform Type is always `CRECT_WAVE`.
- For `RRECT_WAVE` and `RLONG_WAVE` waveforms, imaginary zeros are added to create complex values.
- For `CRECT_WAVE`, the operation copies the input `Waveform*` to the output `Waveform*`.
- For `POLAR_WAVE`, a polar-to-rectangular conversion is performed.

Note the following about the output waveform:

- X_units, Y_units, X_start, and X_increment are set to match the input waveform

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_make_complex( Waveform* out_wave,
                            Waveform* in_wave );
```

where:

`out_wave` specifies the output waveform.

`in_wave` specifies the waveform to be converted.

Example

???

3.27.18.15 waveform_multiply()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_multiply()` function is used to perform sample-by-sample multiplication on a waveform, with the resulting values stored in an output waveform.

Two basic forms are provided:

- Each sample of the output waveform consists of the multiplication of the two corresponding samples of two input waveforms.
- Each sample of the output waveform consists of the multiplication of a corresponding sample of an input waveform times a scalar value (RRECT_WAVE, RLONG_WAVE) or two scaler values (CRECT_WAVE, POLAR_WAVE).

Also note:

- If both inputs are RLONG_WAVE, the output will be RLONG_WAVE.
- If both inputs are RRECT_WAVE, the output will be RRECT_WAVE.
- If one input is RRECT_WAVE and the other is RLONG_WAVE, the output will be RRECT_WAVE.
- If both inputs are CRECT_WAVE, the output will be CRECT_WAVE and calculated as:

$$real = \text{real}1 \times \text{real}2 - \text{imag}1 \times \text{imag}2$$
$$imag = \text{real}1 \times \text{imag}2 + \text{real}2 \times \text{imag}1$$

- If both inputs are POLAR_WAVE, the vector dot product is performed as:

$$mag = |mag1 \times mav2| \times \cos(|phase1 - phase2|)$$
$$phase = 0$$

- If one input is CRECT_WAVE and the other is an RRECT_WAVE or RLONG_WAVE, the output is CRECT_WAVE and calculated as:

$$real = \text{real}1 \times \text{real}2$$
$$imag = \text{imag}1 \times \text{real}2$$

- If one input is CRECT_WAVE and the other is POLAR_WAVE, the polar waveform is converted to rectangular, the product is computed using complex multiplication and the result is left in complex form (CRECT_WAVE).

When performing multiplication using two scalar value(s):

- The input waveform must be CRECT_WAVE or POLAR_WAVE waveform.
- The scalars are intrepeted to represent a tuple of the same type as the input wave type and the multiplication is performed.

When performing multiplication using one scalar value

- If the input is RRECT_WAVE, the output is RRECT_WAVE.
- If the input is RLONG_WAVE, the output remains RLONG_WAVE if the scalar is an integer value otherwise the output is RRECT_WAVE.

- If the input is CRECT_WAVE, the results are formed by:

$$real = \text{real}1 \times scalar$$
$$imag = \text{imag}1 \times scalar$$

- If the input is POLAR_WAVE, the results are formed by:

$$real = \text{real}1 \times scalar$$
$$imag = \text{imag}1$$

In all cases, multiplication is performed on a per-sample basis (see Waveform Sample Programming). The following rules apply:

- When two waveforms are being multiplied they must have the same sample Size. If this rule is violated the output waveform will be the size of the smaller input waveform, a warning message is output in the appropriate controller output window, and testing otherwise continues.
- The output waveform will be [re-]sized to contain only the samples resulting from the multiplication. This makes sense when the output Waveform* initially contains no sample data, but also applies when an existing Waveform*, which does contain sample values, is redefined (clobbered).
- X_units, Y_units, X_start, and X_increment of the output waveform are set to match the first input waveform. See Waveform* Attributes, Waveform Mathematical View and Waveform Units for descriptions of these parameters.
- BAD_WAVE propegates i.e. if any input waveform is BAD_WAVE, the output waveform will be BAD_WAVE.

Usage

The following function performs a sample-by-sample multiplication. Both waveforms must be defined using the same notation (see Waveform Sample Value Notations).

```
void waveform_multiply(  Waveform* out_wave,
                         Waveform* in_wave1,
                         Waveform* in_wave2 );
```

The following function is only usable with waveforms defined using a one-part notation (RRECT_WAVE, RLONG_WAVE):

```
void waveform_multiply(  Waveform* out_wave,
                         Waveform* in_wave,
                         double scalar );
```

The following function is only usable with waveforms defined using a two-part notation (CRECT_WAVE, POLAR_WAVE):

```
void waveform_multiply(  Waveform* out_wave,
                         Waveform* in_wave,
                         double scalar1,
                         double scalar2 );
```

where:

out_wave is the output waveform which results from the multiplication:

in_wave1 and **in_wave2** identify two waveforms which are to be multiplied. **out_wave** = **in_wave1** * **in_wave2**.

in_wave identifies one waveform which will have each sample multiplied by scalar value(s). **out_wave** = **in_wave1** * **scalar** (RRECT_WAVE, RLONG_WAVE) or **out_wave** = **in_wave1** * **scalar1** and **scalar2** (CRECT_WAVE, POLAR_WAVE). See Description.

scalar, **scalar1**, and **scalar2** are scalar values to be multiplied with each sample of the input waveform. See Description.

Example

???

3.27.18.16 waveform_negate()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_negate()` function is used to negate each sample value of a specified input waveform placing the results in an output waveform.

The mathematical operation used for each waveform Type is shown below (see Waveform* Attributes, Waveform Sample Value Notations and Waveform Mathematical View):

- RRECT_WAVE $f(x) = -x$
- CRECT_WAVE $f(x + yi) = -x \ -yi$
- POLAR_WAVE $f(r, \ theta) = -r, \ theta$
- RLONG_WAVE $f(x) = -x$ (same as RRECT_WAVE)

Note the following about the output waveform:

- X_units, Y_units, X_start, and X_increment are set to match the input waveform

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_negate( Waveform* out_wave, Waveform* in_wave );
```

where:

out_wave specifies the output waveform.

in_wave specifies the waveform to be negated.

Example

???

3.27.18.17 waveform_polar_to_rectangular()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_polar_to_rectangular()` function is used to convert an input waveform, with sample values defined using polar notation (`POLAR_WAVE`), to rectangular notation (`CRECT_WAVE`) placing the result in the output waveform. See Waveform Sample Value Notations. Note the following:

- The input `Waveform*` Type must be `POLAR_WAVE`.
- The output `Waveform*` Type will be `CRECT_WAVE`.
- The input waveform samples will be interpreted as magntude and phase (angle) and will be converted to real and imaginary amplitude values.

Note the following about the output waveform:

- X_units, Y_units, X_start, and X_increment are set to match the input waveform

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_polar_to_rectangular(Waveform* out_wave,
                                   Waveform* in_wave );
```

where:

out_wave specifies the output waveform.

in_wave specifies the waveform to be converted.

Example

???

3.27.18.18 waveform_reciprocal()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_reciprocal()` function is used to convert each input waveform sample value to the reciprocal of that value with the results placed in the output waveform.

The mathematical operation used for each waveform Type is shown below (see Waveform Sample Value Notations and Waveform Mathematical View):

- RRECT_WAVE $f(x) = \frac{1}{x}$
- CRECT_WAVE $f(x-yi) = \frac{1}{(x-yi)} = \frac{x}{(x^2+y^2)} - \frac{yi}{(x^2+y^2)}$
- POLAR_WAVE Error
- RLONG_WAVE Transformed to RRECT_WAVE

Note the following about the output waveform:

- X_units, Y_units, X_start, and X_increment are set to match the input waveform

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_reciprocal( Waveform* out_wave, Waveform* in_wave );
```

where:

out_wave specifies the output waveform.

in_wave specifies the input waveform to be processed.

Example

???

3.27.18.19 waveform_rectangular_to_polar()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_rectangular_to_polar()` function is used to convert a waveform from rectangular notation to polar notation. See Waveform Sample Value Notations. This is commonly done to convert the complex results of an FFT to polar representation.

Note the following:

- The input `Waveform* Type` must be `CRECT_WAVE`.
- The output `Waveform* Type` will be `POLAR_WAVE`.
- The input waveform sample values will be interpreted as real and imaginary amplitude values and will be converted to magnitude and phase (angle) values.
- Phase values are returned in the range of -Pi to +Pi.

Note the following about the output waveform:

- X_units, Y_units, X_start, and X_increment are set to match the input waveform

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_rectangular_to_polar(Waveform* out_wave,
                                   Waveform* in_wave );
```

where:

out_wave specifies the output waveform.

in_wave specifies the input waveform to be converted.

Example

???

3.27.18.20 waveform_replace_subset()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_replace_subset()` function is used to replace all or part of one waveform (`in_wave1`) with the entire contents of a second waveform (`in_wave2`), placing the result in a third waveform (`out_wave`). Note the following:

- Neither `in_wave1` nor `in_wave2` are modified.
- `in_wave1` and `in_wave2` must use the same sample notation (see Waveform Sample Value Notations). `RRECT_WAVE` and `RRECT_WAVE` are considered compatible.
- `out_wave` will use the same sample notation, and will inherit all of the scaling information from `in_wave1`. See Waveform* Attributes.
- The starting point (`starting_offset`) for the data replacement must be a valid offset within `in_wave1`. i.e. 0 to size(`in_wave1`) -1.
- The output waveform will, if necessary, be resized to include all samples from `in_wave2`.

Usage

```
void waveform_replace_subset(
                            Waveform* out_wave,
                            Waveform* in_wave1,
                            Waveform* in_wave2,
                            int starting_offset DEFAULT_VALUE( 0 ) );
```

where:

out_wave specifies the output waveform.

in_wave1 specifies the first input waveform.

in_wave2 specifies the second input waveform.

starting_offset is optional, and if used specifies the first value from **in_wave1** to be repaced. Default = 0 i.e. start with the first sample value.

Example

The following example rotates a portion of a waveform

```
// Using waveform_subset(), get some samples from the original
// waveform
waveform_subset( subsetWF, originalWF, 10, 25 );

// Rotate these samples: waveform_rotate_left()
waveform_rotate_left( subsetWF, subsetWF );

// Replace the original WF with the rotated WF
waveform_replace_subset( originalWF, originalWF, subsetWF, 10 );
```

3.27.18.21 waveform_resample()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_resample()` function utilizes Shannon's resampling theorem to convert a waveform to an alternate representation. The new representation reflects the same waveform sampled at a different frequency. Note the following:

- The input Waveform must be `RRECT_WAVE` or `RLONG_WAVE`.
- The output waveform will be `RRECT_WAVE`.
- The desired number of samples in the output waveform is specified using the `num_samples` parameter.
- Since the function relies on FFT operations, both the input waveform and the output waveform must be at least eight samples long.
- The output waveform will have the same X_start, X_units, and Y_units as the input, but the X_increment value will be appropriately adjusted.
- `num_samples` may represent a value which is less than, equal to, or greater than the input waveform's length. If it is less than, the resulting waveform may change in appearance. This would occur only if the input waveform has high frequency components that fall below the newly lowered Nyquist frequency. In this situation,

the output waveform is an accurate representation of what the input waveform would look like if it was undersampled. The high frequency components would alias to lower frequencies.

Usage

```
void waveform_resample(Waveform* out_wave,
                       Waveform* in_wave,
                       int num_samples );
```

where:

out_wave specifies the output waveform.

in_wave specifies the input waveform to be converted.

num_samples specifies the desired number of samples in the output waveform. See Description.

Example

???

3.27.18.22 waveform_rescale()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_rescale()` function is used to linearly map the current range of waveform sample values from an input waveform to a new range with the results placed in an output waveform.

`waveform_rescale()` is only usable on waveforms defined a one-part notation (`RRECT_WAVE`, `RLONG_WAVE`). The resulting output waveform is in the `RRECT_WAVE` notation. See Waveform Sample Value Notations.

Two versions of `waveform_rescale()` are available:

- The first version takes a single scaling parameter: `max_value`. The input waveform's sample values are analyzed to identify the maximum positive sample value (`sample_max`) and all samples matching this value are reset to the specified `max_value`. All other waveform sample values are then scaled using the formula:

$$out = \frac{in \times \text{max\_value}}{\text{sample\_max}}$$

- The second version takes two scaling parameters: `max_value` and `min_value`. The input waveform's sample values are analyzed to identify the maximum sample value (`sample_max`) and the minimum sample value (`sample_min`). Then, all waveform sample values are scaled using the formula:

$$out = \frac{(in - \text{sample\_min}) \times (\text{max\_value} - \text{min\_value})}{(\text{sample\_max} - \text{sample\_min})} + \text{min\_value}$$

Note the following about the output waveform:

- X_units, Y_units, X_start, and X_increment are set to match the input waveform

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_rescale( Waveform* out_wave,
                       Waveform* in_wave,
                       double max_value DEFAULT_VALUE( 1.0 ) );
void waveform_rescale( Waveform* out_wave,
                       Waveform* in_wave,
                       double min_value,
                       double max_value );
```

where:

`out_wave` specified the destination for the rescaled waveform.

`in_wave` specifies the waveform to be rescaled

`max_value` and **`min_value`** are used as noted in Description.

Example

???

3.27.18.23 waveform_reverse()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_reverse()` function can be used to reverse the sample value order of an input waveform, putting the result into an output waveform.

Note the following about the output waveform:

- X_units and Y_units are set to match the input waveform
- X_start is set to: input waveform X_start + (size-1) * input waveform X_increment
- X_increment is set to: -1 * input waveform X_increment

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_reverse( Waveform* out_wave, Waveform* in_wave );
```

where:

`out_wave` identifies the output waveform.

`in_wave` identifies the input waveform.

Example

???

3.27.18.24 waveform_rotate_left(), waveform_rotate_right()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_rotate_left()` function may be used to left rotate the sample values of a specified waveform a specified number of `locations`. This could be used to phase shift a periodic time domain waveform.

The `waveform_rotate_right()` function may be used to right rotate the sample values of a specified waveform a specified number of locations.

Note the following:

- The input waveform sample values may be defined using any notation (see Waveform Sample Value Notations). The ouput waveform sample values will be defined using the same notation.
- The output waveform will be the same Size as the input waveform (see `waveform_get_size()`).
- The specified `locations` value is reduced modulo the Size of the input waveform.
- Negative numbers effectively shift in the opposite direction, therefore:

  ```
  waveform_rotate_left( outWF, inWF, -3 );
  ```

  is equivalent to:

  ```
  waveform_rotate_right( outWF, inWF, 3 );
  ```

Also note the following about the output waveform:

- X_units and Y_units are set to match the input waveform
- X_start is set to to match the time associated with the first sample in the output waveform
- X_increment is set to match the input waveform

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_rotate_left(  Waveform* out_wave,
                            Waveform* in_wave,
                            int locations DEFAULT_VALUE( 1 ) );
void waveform_rotate_right( Waveform* out_wave,
                            Waveform* in_wave,
                            int locations DEFAULT_VALUE( 1 ) );
```

where:

`out_wave` identifies the output waveform.

`in_wave` identifies the input waveform.

`locations` specifies the number of locations the sample values are rotated.

Example

???

3.27.18.25 waveform_sort()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_sort()` function is used to sort the sample values of an input waveform and copy the result to an output waveform. The sort can order the results in ascending or descending order.

Note the following about the output waveform:

- X_units and Y_units units are set to match the input waveform
- X_start is set to to match the time associated with the first sample in the output waveform
- X_increment is set to match the input waveform

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_sort( Waveform* out_wave,
                    Waveform* in_wave,
                    BOOL ascending DEFAULT_VALUE( TRUE ) );
```

where:

`out_wave` identifies the output waveform.

`out_wave` identifies the input waveform to be sorted.

`ascending` is optional, and if used specifes whether the sort should put the sample values in ascending order (TRUE) or descending order (FALSE). Default = TRUE = ascending.

Example

???

3.27.18.26 waveform_split()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_split()` function is used to split an input waveform with sample values defined using `CRECT_WAVE` or `POLAR_WAVE` notation into two output waveforms defined using the `RRECT_WAVE` notation.

The sample data of the first output `RRECT_WAVE` waveform corresponds to the *magnitude* values of a `POLAR_WAVE`, or the *real* value of a `CRECT_WAVE`.

The sample data of the second output `RRECT_WAVE` waveform corresponds to the *phase* value of a `POLAR_WAVE`, or the *imaginary* value of a `CRECT_WAVE`.

Note the following about the output waveform:

- X_units, Y_units, X_start, and X_increment are set to match the input waveform

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_split(Waveform* out_wave1,
                    Waveform* out_wave2,
                    Waveform* in_wave );
```

where:

`out_wave1` and **`out_wave2`** specify the destinations for the split waveform. See Description.

`in_wave` specifies the waveform to be split. This must be a waveform defined using a two-part notation (`CRECT_WAVE` or `POLAR_WAVE`); see Description.

Example

???

3.27.18.27 waveform_strided_copy()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_strided_copy()` function is used to copy sample values from an input waveform to an output waveform. Note the following:

- Copying begins with the input waveform sample value identified by the `input_start` argument.
- The `input_stride` parameter specifies which subsequent samples to copy. For example, if `input_stride` = 5, every 5th sample value from the input waveform will be copied to the output waveform.
- The `num_elements` argument specifies the number of elements to copy. The output waveform Size will be `num_elements`. If `num_elements` is small enough, copying will stop before reaching the end of the input waveform. If `num_elements` is large enough, copying will wrap back to the start of the input waveform. If `num_elements` is not supplied, or specified as -1, its value will be the Size of the input waveform.
- The input waveform sample values may be defined using any notation (see Waveform Sample Value Notations). The ouput waveform sample values will be defined using the same notation.

Also note the following about the output waveform:

- X_units and Y_units units are set to match the input waveform
- X_start is set to the time associated with the first sample value.
- X_increment is set to ???.

X_incrementSee Waveform Mathematical View and Waveform Units for descriptions of these parameters.

For example, given an input waveform containing the following 10 sample values:

```
{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
```

Executing:

```
waveform_strided_copy( outWF, inWF, 2, 3 ),
```

operates as follows:

1. Since `num_elements` is not specified, it defaults = 10 = Size of the input waveform.

2. `input_start` = 2, so the sample value at index 2 (which = 2) is copied to the output waveform.
3. `input_stride` = 3, so the next sample to be copied is at index 2 + 3 = 5, followed by the sample value at 5 + 3 = 8, etc.
4. The next index should be 8 + 3 = 11, but this is beyond the end of the input waveform, so it is reduced by the length of the input waveform i.e. 11 - 10 = 1. Thus the sample value at 1 is copied next.
5. Copying continues with the samples at index 4 and 7, then another wrap occurs at index 10, which becomes 0. The process continues for all 10 sample values.

The output waveform will contain the following:

```
{ 2, 5, 8, 1, 4, 7, 0, 3, 6, 9 }
```

Note that if the `input_stride` value is set = 1 (default), `waveform_strided_copy()` operates the same as `waveform_rotate_left()`, i.e.:

```
waveform_strided_copy( outWF, inWF, 10 );
```

is equivalent to:

```
waveform_rotate_left( outWF, inWF, 10 );
```

Another use for `waveform_strided_copy()` is to replicate a waveform a specified number of times and to concatenate the copies into the output waveform. For example, if `inWF` contains 1000 sample values (Size = 1000), the following will replicate/concatenate the waveform five times in `outWF`:

```
waveform_strided_copy( outWF, inWF, 0, 1, 5000 );
```

The `waveform_double_strided_copy()` function may be used to undo the effects of `waveform_strided_copy()`. For example:

```
waveform_strided_copy( outWF, inWF, start, stride );
```

may be undone with:

```
waveform_double_strided_copy( outWF, inWF, start, stride );
```

Note that this only functions correctly if `waveform_strided_copy()` had previously performed a full copy (`num_elements` = input waveform Size) ***AND*** the `input_stride` value represented a *well-formed* value (see Mutually Prime, next).

Mutually Prime

Two numbers are said to be *mutually prime* if they share no common factors (other than one). The concept of mutually prime is important when using `waveform_strided_copy()` and `waveform_double_strided_copy()`.

If `waveform_strided_copy()` is executed with `num_elements` = input waveform Size, it might seem that all samples from the input waveform would be copied to the output waveform, with only the order changing. However, this is not necessarily true. If the `input_stride` parameter is not *mutually prime* with the Size of the input waveform, some sample values of the input waveform will be duplicated in the output waveform, while others will be skipped. When this occurs, the input parameters (`input_stride` vs. input waveform Size) are referred to as not *well formed*. This situation is not detected and thus no warning messages are issued.

For example, given an input waveform of 100 sample values, executing the following will not copy all input waveform sample values to the output waveform, even though `num_elements` = input waveform Size:

```
// Not "well formed"
waveform_strided_copy( outEF, inWF, 11, 18, 100 );
```

The `input_stride` value (18) shares a factor (2) with the waveform Size (100). Since copying starts on an odd sample value index (11) , and an even number (18) is always added to obtain the next index, and wrapping always subtracts an even number (Size = 100), the index will always remain an odd number. The result is that all of the sample values at odd index locations will be copied to the output waveform, twice, and none of the even indexed values will be copied.

Usage

```
void waveform_strided_copy(
                          Waveform* out_wave,
                          Waveform* in_wave,
                          int input_start DEFAULT_VALUE( 0 ),
                          int input_stride DEFAULT_VALUE( 1 ),
                          int num_elements DEFAULT_VALUE( -1 ) );
```

where:

`out_wave` identifies the output waveform.

`in_wave` identifies the input waveform.

`input_start` is optional, and if used identifies location (zero based) of the first value from the input waveform to be copied to the output waveform. Default = 0 = first sample value.

input_stride is optional, and if used specifies how sample values to be copied after **input_start** to are identified. See Description. Default = 1 = no values are skipped.

num_elements is optional, and if used specifies how many sample values to copy. See Description. Default = -1 = copy all values.

Example

???

3.27.18.28 waveform_subset()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_subset()` function can be used to create an output waveform containing a subset of sample values from an input waveform.

Note the following about the output waveform:

- X_units and Y_units units are set to match the input waveform
- X_start of the output waveforms is set to to match the time associated with the first sample in the waveform
- X_increment is set to match the input waveform

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_subset(  Waveform* out_wave,
                       Waveform* in_wave,
                       int offset DEFAULT_VALUE( 0 ),
                       int length DEFAULT_VALUE( -1 ) );
```

where:

out_wave specified the output waveform.

in_wave specifies the input waveform.

offset is optional, and if used specifies the first value from **in_wave** to be copied. Default = 0 i.e. start with the first sample value.

length is optional, and if used specifies the number of samples to take from **in_wave**. Default = -1 i.e. take all samples from **offset** to the end of the waveform.

Example

???

3.27.18.29 waveform_subtract()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_subtract()` function is used to perform sample-by-sample subtraction on waveform(s), with the resulting values stored in an output waveform.

Two basic forms are provided:

- Each sample of the output waveform consists of the subtraction of two corresponding samples of two input waveforms.
- Each sample of the output waveform consists of the subtraction of a corresponding sample of an input waveform minus a scalar value (`RRECT_WAVE`, `RLONG_WAVE`) or two scaler values (`CRECT_WAVE`, `POLAR_WAVE`).

Also note:

- If both inputs are `RLONG_WAVE`, the output will be `RLONG_WAVE`.
- If both inputs are `RRECT_WAVE`, the output will be `RRECT_WAVE`.
- If one input is `RRECT_WAVE` and the other is `RLONG_WAVE`, the output will be `RRECT_WAVE`.
- If both inputs are `CRECT_WAVE`, the output will be `CRECT_WAVE`:
  real = real1 - real2
  imag = imag1 - imag2;
- If both inputs are `POLAR_WAVE`, vector subtraction is performed. Both input waveforms are converted to rectangular notation, subtracted, then converted back to polar notation.

- If one input is CRECT_WAVE, the other input waveform is converted to complex using waveform_make_complex(). The results are subtracted and left in complex form (CRECT_WAVE). waveform_make_complex() appends an imaginary value of zero to RRECT_WAVE and RLONG_WAVE waveforms. For POLAR_WAVE waveforms, waveform_polar_to_rectangular() is called.
- When subtracting a waveform and a single scalar:
  - If the input waveform is RRECT_WAVE, the output is RRECT_WAVE.
  - If the input waveform is RLONG_WAVE, the output remains RLONG_WAVE if the scalar is an integer value otherwise the output is RRECT_WAVE.
  - If the input waveform is CRECT_WAVE, the results are formed by:
    real = real1 - scalar
    imag = imag1
  - If the input is POLAR_WAVE, the scalar is promoted to the vector (scalar, 0.0) and vector subtraction is performed.
- When subtracting a waveform and two scalars, the input waveform type must be either CRECT_WAVE or POLAR_WAVE. The scalars are intrepeted to represent a tuple of the same type as the input wave type and the subtraction is performed.

In all cases, subtraction is performed on a per-sample basis (see Waveform Sample Programming). The following rules apply:

- When two waveforms are being subtracted they must have the sample sample Size. If this rule is violated the output waveform will be the size of the smaller input waveform, a warning message is output in the appropriate controller output window, and testing otherwise continues.
- The output waveform will be [re-]sized to contain only the samples resulting from the subtraction. This makes sense when the output Waveform* initially contains no sample data, but also applies when an existing Waveform*, which does contain sample values, is redefined (clobbered).

Also note the following about the output waveform:

- X_units, Y_units, X_start, and X_increment are set to match the first input waveform. See Waveform* Attributes, Waveform Mathematical View and Waveform Units for descriptions of these parameters.
- BAD_WAVE propegates i.e. if any input waveform is BAD_WAVE, the output waveform will be BAD_WAVE.

Usage

The following function performs a sample-by-sample subtraction; in_wave1 minus in_wave2. Both waveforms must be defined using the same notation (see Waveform Sample Value Notations).

```
void waveform_subtract(Waveform* out_wave,
                       Waveform* in_wave1,
                       Waveform* in_wave2 );
```

The following function is only usable with waveforms defined using a one-part notation (RRECT_WAVE, RLONG_WAVE):

```
void waveform_subtract(Waveform* out_wave,
                       Waveform* in_wave,
                       double scalar );
```

The following function is only usable with waveforms defined using a two-part notation (CRECT_WAVE, POLAR_WAVE):

```
void waveform_subtract(Waveform* out_wave,
                       Waveform* in_wave,
                       double scalar1,
                       double scalar2 );
```

where:

out_wave is the output waveform which results from the subtraction:

in_wave1 and **in_wave2** identify two waveforms which are to be subtracted. **out_wave** = **in_wave1** - **in_wave2**.

in_wave identifies one waveform which will have scalar value(s) subtracted from each sample. **out_wave** = **in_wave1** - **scalar** (RRECT_WAVE, RLONG_WAVE) or **out_wave** = **in_wave1** - **scalar1** and **scalar2** (CRECT_WAVE, POLAR_WAVE). See Description.

scalar, **scalar1**, and **scalar2** are scalar values to be subtracted from each sample of the input waveform. See Description.

Example

???

3.27.18.30 waveform_sum()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_sum()` function can be used to obtain a sum of all sample values of a specified waveform. Only waveforms defined using the `RRECT_WAVE` or `RLONG_WAVE` notation are supported.

Usage

```
double waveform_sum( Waveform* in_wave );
```

where:

`in_wave` specifies the waveform to be summed.

`waveform_sum()` returns the sum of all sample values of **`in_wave`**.

Example

???

3.27.18.31 waveform_summing()

See Waveform Overview, Waveform Manipulation Functions.

Description

The `waveform_summing()` function returns an output waveform containing sample values which are the running sum of sample values from an input waveform. The n^{th} sample value of the output waveform is equal to the sum of n^{th} sample value of the input waveform and all sample values preceding it. The sum is computed using the method of addition appropriate for the notation used to define the input waveform sample values (see Waveform Sample Value Notations).

The Size (see `waveform_get_size()`) and sample value notation (see Waveform Sample Value Notations) of the output waveform is the same as the input waveform.

Note the following about the output waveform:

- X_units and Y_units units are set to match the input waveform
- X_start is set to ???
- X_increment is set to ???

X_startX_incrementSee Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_summing( Waveform* out_wave, Waveform* in_wave );
```

where:

`out_wave` identifies the output waveform.

`in_wave` identifies the input waveform.

Example

???

3.27.19 Waveform Equality Functions

See Waveform Overview, Waveform Units

- `waveform_gt()`
- `waveform_lt()`
- `waveform_ge()`
- `waveform_le()`
- `waveform_eq()`
- `waveform_within_bounds()`

3.27.19.1 waveform_gt()

See Waveform Overview, Waveform Equality Functions.

Description

The `waveform_gt()` function is used to determine if each sample value of a specified input waveform is greater than (gt) either a specified scalar value or a corresponding sample value of a second specified waveform.

Two versions of `waveform_gt()` are available:

- The versions which return a `BOOL` value only return an overall result of the comparison.
- The versions with the `out_wave` argument return the `out_wave` `Waveform*` with each sample value containing the result of comparing the corresponding sample values of the two input waveforms.

Also note the following:

- The waveform(s) must be defined using `RRECT_WAVE` or `RLONG_WAVE` notation.
- When comparing two waveforms they are not required to match in Type i.e. it is valid to compare `RRECT_WAVE` vs. `RLONG_WAVE`.
- When two waveforms are compared they should have the same number of sample values i.e. be the same Size (see `waveform_get_size()`). If not, an error is output in the appropriate controller window, the extra sample values of the larger waveform are ignored, and testing otherwise continues.
- When two waveforms are compared, the X_units type and sample rate (X_increment) of the output waveform are inherited from the first input waveform. The Y_units type = `SCALE_BOOLEAN`.

Usage

```
BOOL waveform_gt( Waveform* in_wave, double scalar );
BOOL waveform_gt( Waveform* in_wave1, Waveform* in_wave2 );
void waveform_gt( Waveform* out_wave,
                  Waveform* in_wave1,
                  Waveform* in_wave2 );
void waveform_gt( Waveform* out_wave,
                  Waveform* in_wave,
                  double scalar );
```

where:

`in_wave` identifies the waveform to be evaluated.

`scalar` specifies the value to be compared with each **`in_wave`** sample value.

out_wave specifies the output waveform in which each sample value is the result of the comparison of the corresponding sample values of **in_wave1** vs. **in_wave2**. The **out_wave** sample values (RLONG_WAVE notation) will consist of 1 = TRUE or 0 = FALSE.

The versions of waveform_gt() which return BOOL will return TRUE when the test of every sample value returns TRUE, otherwise FALSE is returned. See Description.

Example

???

3.27.19.2 waveform_lt()

See Waveform Overview, Waveform Equality Functions.

Description

The waveform_lt() function is used to determine if each sample value of a specified input waveform is less than (lt) either a specified scalar value or a corresponding sample value of a second specified waveform.

Two versions of waveform_lt() are available:

- The versions which return a BOOL value only return an overall result of the comparison.
- The versions with the out_wave argument return the out_wave Waveform* with each sample value containing the result of comparing the corresponding sample values of the two input waveforms.

Also note the following:

- The waveform(s) must be defined using RRECT_WAVE or RLONG_WAVE notation.
- When comparing two waveforms they are not required to match in Type i.e. it is valid to compare RRECT_WAVE vs. RLONG_WAVE.
- When two waveforms are compared they should have the same number of sample values i.e. be the same Size (see waveform_get_size()). If not, an error is output in the appropriate controller window, the extra sample values of the larger waveform are ignored, and testing otherwise continues.
- When two waveforms are compared, the X_units type and sample rate (X_increment) of the output waveform are inherited from the first input waveform. The Y_units type = SCALE_BOOLEAN.

Usage

```
BOOL waveform_lt( Waveform* in_wave, double scalar );
BOOL waveform_lt( Waveform* in_wave1, Waveform* in_wave2 );
void waveform_lt( Waveform* out_wave,
                  Waveform* in_wave1,
                  Waveform* in_wave2 );
void waveform_lt( Waveform* out_wave,
                  Waveform* in_wave,
                  double scalar );
```

where:

in_wave identifies the waveform to be evaluated.

scalar specifies the value to be compared with each **in_wave** sample value.

out_wave specifies the output waveform in which each sample value is the result of the comparison of the corresponding sample values of **in_wave1** vs. **in_wave2**. The **out_wave** sample values (RLONG_WAVE notation) will consist of 1 = TRUE or 0 = FALSE.

The versions of waveform_lt() which return BOOL will return TRUE when the test of every sample value returns TRUE, otherwise FALSE is returned. See Description.

Example

???

3.27.19.3 waveform_ge()

See Waveform Overview, Waveform Equality Functions.

Description

The waveform_ge() function is used to determine if each sample value of a specified input waveform is greater than (g) or equal to (e) either a specified scalar value or a corresponding sample value of a second specified waveform.

Two versions of waveform_ge() are available:

- The versions which return a BOOL value only return an overall result of the comparison.

- The versions with the out_wave argument return the out_wave Waveform* with each sample value containing the result of comparing the corresponding sample values of the two input waveforms.

Also note the following:

- The waveform(s) must be defined using RRECT_WAVE or RLONG_WAVE notation.
- When comparing two waveforms they are not required to match in Type i.e. it is valid to compare RRECT_WAVE vs. RLONG_WAVE.
- When two waveforms are compared they should have the same number of sample values i.e. be the same Size (see waveform_get_size()). If not, an error is output in the appropriate controller window, the extra sample values of the larger waveform are ignored, and testing otherwise continues.
- When two waveforms are compared, the X_units type and sample rate (X_increment) of the output waveform are inherited from the first input waveform. The Y_units type = SCALE_BOOLEAN.

Usage

```
BOOL waveform_ge( Waveform* in_wave, double scalar );
BOOL waveform_ge( Waveform* in_wave1, Waveform* in_wave2 );
void waveform_ge( Waveform* out_wave,
                  Waveform* in_wave1,
                  Waveform* in_wave2 );
void waveform_ge( Waveform* out_wave,
                  Waveform* in_wave,
                  double scalar );
```

where:

in_wave identifies the waveform to be evaluated.

scalar specifies the value to be compared with each **in_wave** sample value.

out_wave specifies the output waveform in which each sample value is the result of the comparison of the corresponding sample values of **in_wave1** vs. **in_wave2**. The **out_wave** sample values (RLONG_WAVE notation) will consist of 1 = TRUE or 0 = FALSE.

The versions of waveform_ge() which return BOOL will return TRUE when the test of every sample value returns TRUE, otherwise FALSE is returned. See Description.

Example

???

3.27.19.4 waveform_le()

See Waveform Overview, Waveform Equality Functions.

Description

The `waveform_le()` function is used to determine if each sample value of a specified input waveform is less than (l) or equal to (e) either a specified scalar value or a corresponding sample value of a second specified waveform.

Two versions of `waveform_le()` are available:

- The versions which return a `BOOL` value only return an overall result of the comparison.
- The versions with the `out_wave` argument return the `out_wave` `Waveform*` with each sample value containing the result of comparing the corresponding sample values of the two input waveforms.

Also note the following:

- The waveform(s) must be defined using `RRECT_WAVE` or `RLONG_WAVE` notation.
- When comparing two waveforms they are not required to match in Type i.e. it is valid to compare `RRECT_WAVE` vs. `RLONG_WAVE`.
- When two waveforms are compared they should have the same number of sample values i.e. be the same Size (see `waveform_get_size()`). If not, an error is output in the appropriate controller window, the extra sample values of the larger waveform are ignored, and testing otherwise continues.
- When two waveforms are compared, the X_units type and sample rate (X_increment) of the output waveform are inherited from the first input waveform. The Y_units type = `SCALE_BOOLEAN`.

Usage

```
BOOL waveform_le( Waveform* in_wave, double scalar );
BOOL waveform_le( Waveform* in_wave1, Waveform* in_wave2 );
```

```
void waveform_le( Waveform* out_wave,
                  Waveform* in_wave1,
                  Waveform* in_wave2 );
void waveform_le( Waveform* out_wave,
                  Waveform* in_wave,
                  double scalar );
```

where:

in_wave identifies the waveform to be evaluated.

scalar specifies the value to be compared with each **in_wave** sample value.

out_wave specifies the output waveform in which each sample value is the result of the comparison of the corresponding sample values of **in_wave1** vs. **in_wave2**. The **out_wave** sample values (RLONG_WAVE notation) will consist of 1 = TRUE or 0 = FALSE.

The versions of `waveform_le()` which return BOOL will return TRUE when the test of every sample value returns TRUE, otherwise FALSE is returned. See Description.

Example

???

3.27.19.5 waveform_eq()

See Waveform Overview, Waveform Equality Functions.

Description

The `waveform_eq()` function is used to determine if each sample value of a specified waveform is equal to (eq) either a specified scalar value or a corresponding sample value of another specified waveform.

Equality comparisons of floating point values are not always useful when an exact match is performed. For instance, the expressions

$$\frac{1.0}{3.0} \times 3.0 \quad \text{... and ...} \quad 1.0 \times \frac{3.0}{3.0}$$

both evaluate to 1.0000..., but if the operations are performed in a different order, the results may not exactly match, depending on how floating point round-off is handled.

Therefore, the waveform_eq() function supports an optional tolerance parameter, which allows the user to control the *equality tolerance*. The tolerance parameter may be interpreted as an absolute tolerance or a relative tolerance, depending on the values being compared:

- When the values being compared are < 1, tolerance is treated as an absolute tolerance. For example, if tolerance = 0.01, then 0.815 is equal to 0.822, because the absolute difference is <= 0.01.
- When the values being compared are >= 1, tolerance is treated as an relative tolerance. Using the same tolerance value (0.01) 123,456.0 is equal to 122,789.0 because the difference is >= 1234.56 (0.01 * 123,456). Note that this large tolerance value is examined for illustration purposes only; normally tolerance will be a much smaller value.
- If no tolerance is specified, it defaults to 1.0e-7.
- If tolerance is set = 0, an exact match is performed.

Two versions of waveform_eq() are available:

- The versions which return a BOOL value only return an overall result of the comparison.
- The versions of waveform_eq() which use the out_wave argument return a Waveform* in which each sample value is the result of an equality test of the corresponding sample values from the two specified input waveforms.

Also note the following:

- When comparing two waveforms defined using RRECT_WAVE or RLONG_WAVE notation they are not required to match in Type i.e. it is valid to compare RRECT_WAVE vs. RLONG_WAVE.
- Unlike the other Waveform Equality Functions, some forms of waveform_eq() may be used with waveforms defined using CRECT_WAVE or POLAR_WAVE notation. See Usage. When comparing CRECT_WAVE or POLAR_WAVE waveforms, the two waveforms must be defined using the same notation.
- The sample values of waveform defined using CRECT_WAVE or POLAR_WAVE notation consist of two parts. For CRECT_WAVE these are the *real* and *imaginary* parts. For POLAR_WAVE these are the *magnitude* and *phase*. waveform_eq() performs two equality tests for each sample value, one for each part, and the tolerance value is applied separately to each part. The two equality tests for each sample must both be equal otherwise the comparison returns FALSE.

- When two Waveform* are compared they should have the same number of sample values i.e. be the same Size (see waveform_get_size()). If not, an error is output in the appropriate controller window, the extra sample values of the longer waveform are ignored, and testing otherwise continues.
- When two Waveform* are compared, the X_units type and sample rate (X_increment) of the output waveform are inherited from the first input waveform. The Y_units type = SCALE_BOOLEAN.

Usage

```
BOOL waveform_eq( Waveform* in_wave,
                  double scalar,
                  double tolerance DEFAULT_VALUE( 1.0e-7 ) );
void waveform_eq( Waveform* out_wave,
                  Waveform* in_wave,
                  double scalar,
                  double tolerance );
```

The following versions of waveform_eq() (only) may be used with waveforms defined using CRECT_WAVE or POLAR_WAVE notation:

```
BOOL waveform_eq( Waveform* in_wave1,
                  Waveform* in_wave2,
                  double tolerance DEFAULT_VALUE( 1.0e-7 ) );
void waveform_eq( Waveform* out_wave,
                  Waveform* in_wave1,
                  Waveform* in_wave2,
                  double tolerance DEFAULT_VALUE( 1.0e-7 ) );
```

where:

in_wave identifies the waveform to be evaluated.

scalar specifies the value to be used during the equality test of each **in_wave** sample value.

tolerance see Description.

out_wave specifies the output waveform in which each sample value is the result of an equality test of the corresponding sample values of **in_wave1** and **in_wave2**. The **out_wave** sample values (RLONG_WAVE notation) will consist of 1 = TRUE or 0 = FALSE.

The versions of waveform_eq() which return BOOL will return TRUE when every equality test returned TRUE, otherwise FALSE is returned. See Description.

Example

???

3.27.19.6 waveform_within_bounds()

See Waveform Overview, Waveform Equality Functions.

Description

The `waveform_within_bounds()` function is used to determine if every sample value of a specified waveform is numerically within specified bounds.

Two basic variations of `waveform_within_bounds()` are available (which partially explains the number of `waveform_within_bounds()` function overloads):

- The versions which return a BOOL value only return an overall result of the evaluation.
- The versions which use the `out_wave` argument return a `Waveform*` in which each sample value is the result of the evaluation of each input sample value.

Note the following:

- The input waveform must be defined using `RRECT_WAVE` and `RLONG_WAVE` notation.
- Both minimum bound and maximum bound values must be specified. The within-bounds test is inclusive. e.g. a value is within bounds when:

  ```
  minimum bound value <= sample value <= maximum bound value
  ```

- The minimum and/or maximum bound values may be specified using a scalar value or a waveform. When a scalar value is used for either bound the within-bounds test is made by comparing every sample value of the input waveform against the same scalar value. When a bound is specified using a waveform, the within-bounds test is made by comparing every sample value of the input waveform against the corresponding sample value of the specified waveform. Four combinarions of scalar vs. waveform are possible, which partially explains the number of `waveform_within_bounds()` function overloads.

- When a bound is specified using a waveform, that waveform is expected to have the same number of sample values as the input waveform i.e. be the same Size (see `waveform_get_size()`). If not, an error is output in the appropriate controller window, the extra sample values of the longer waveform are ignored, and testing otherwise continues.
- When an output waveform is specified, the sample values of that waveform are the boolean result of the within-bounds test of each corresponding sample value of the input waveform.

Also note the following about the output waveform:

- The sample values are specified in `RLONG_WAVE` notation, and will consist of 1 = TRUE or 0 = FALSE
- Y_units is set to `SCALE_BOOLEAN`
- X_start is set to 0
- X_increment is set = 1
- X_units is set to `SCALE_NOXUNITS`

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

*X_units/Y_unitsX_increment*Usage

```
BOOL waveform_within_bounds(Waveform* in_wave,
                            double min_scalar,
                            double max_scalar );
BOOL waveform_within_bounds(Waveform* in_wave,
                            Waveform* min_wave,
                            double max_scalar );
BOOL waveform_within_bounds(Waveform* in_wave,
                            double min_scalar,
                            Waveform* max_wave );
BOOL waveform_within_bounds(Waveform* in_wave,
                            Waveform* min_wave,
                            Waveform* max_wave );
void waveform_within_bounds(Waveform* out_wave,
                            Waveform* in_wave,
                            double min_scalar,
                            double max_scalar );
```

```
void waveform_within_bounds(Waveform* out_wave,
                            Waveform* in_wave,
                            Waveform* min_wave,
                            double max_scalar );
void waveform_within_bounds(Waveform* out_wave,
                            Waveform* in_wave,
                            double min_scalar,
                            Waveform* max_wave );
void waveform_within_bounds(Waveform* out_wave,
                            Waveform* in_wave,
                            Waveform* min_wave,
                            Waveform* max_wave );
```

where:

in_wave identifies the waveform to be evaluated.

Bound minimum and maximum values can be set using any combination of:

- **min_scalar** and **max_scalar**, which specify a single value to compare with each **in_wave** sample value.
- **min_wave** and **max_wave**. When used, each sample value of **min_wave** and/or **max_wave** are compared with the corresponding sample value of **in_wave**.

out_wave specifies the output waveform. When used, each **out_wave** sample value is the result of the within-bounds check of the corresponding sample value of **in_wave**. See Description.

The versions of `waveform_within_bounds()` which return `BOOL` will return `TRUE` if all input waveform sample values are within bounds, otherwise `FALSE` is returned. See Description.

Example

???

3.27.20 Waveform Conversion Functions

See Waveform Overview.

The waveform conversion functions are used for translating waveform sample values from one integer representation to another.

Only input waveforms with sample values defined using the RLONG_WAVE notation are supported.

The output waveform Type = RLONG_WAVE.

Those representations whose interpretation is based on word size also take a bitwidth argument, which represents the width of each sample value in bits. In this mode, when converting from binary, the output waveform will only contain non-zero bits (ones) in the least significant bitwidth bits of each sample value. The most significant bits will be all zeroes. Legal bitwidth values range from 2 to 54. Sample values which cannot be legally represented in the bitwidth specified are clamped (with no warning) or converted to valid values, the details of which may be found in the description for the specific translation function.

The term *binary* refers to the native representation of RLONG_WAVEs. If a waveform containing converted sample values is passed to a math function, the contained values will not be interpreted as their encoded values. The bit sequences would instead be reinterpreted as a different binary value. All alternate representations should be converted to binary before attempting math operations, comparison operations, etc.

3.27.20.1 waveform_binary_to_gray_code()

See Waveform Overview, Waveform Conversion Functions.

Description

The waveform_binary_to_gray_code() function will convert the binary representation of each sample value of the input waveform to its Grey code equivalent binary value. Note the following:

- Only positive input waveform sample values will be converted. Negative values are set to binary 0.
- The table below illustrates the first 16 gray code value:

Binary Value	Gray Code Representation
0	0000
1	0001
2	0011
3	0010

Binary Value	Gray Code Representation
4	0110
5	0111
6	0101
7	0100
8	1100
9	1101
10	1111
11	1110
12	1010
13	1011
14	1001
15	1000

Usage

```
void waveform_binary_to_gray_code( Waveform* out_wave,
                                   Waveform* in_wave );
```

where:

out_wave specifies the output waveform. See Description.

in_wave specifies the input waveform. See Description.

Example

???

3.27.20.2 waveform_gray_code_to_binary()

See Waveform Overview, Waveform Conversion Functions.

Description

This is the inverse function of `waveform_binary_to_gray_code()`.

Only positive input waveform sample values will be converted. Negative values are set to binary 0.

Usage

```
void waveform_gray_code_to_binary( Waveform* out_wave,
                                   Waveform* in_wave );
```

where:

`out_wave` specifies the output waveform. See Description.

`in_wave` specifies the input waveform. See Description.

Example

???

3.27.20.3 waveform_binary_to_bcd()

See Waveform Overview, Waveform Conversion Functions.

Description

The `waveform_binary_to_bcd()` function will convert the binary representation of each sample value of the input waveform to its equivalent BCD binary value. Note the following:

- BCD stands for Binary Coded Decimal.
- In BCD representation, every four binary bits represents a decimal value (0 to 9). BCD is less compact than binary.
- The largest BCD value supported is 19,999,999,999,999 (53 bits). Larger values will be clamped to this limit.
- Only positive sample values are converted. Negative numbers are set to binary zero.

- The table below illustrates the first 16 BCD values:

Binary Value	BCD Representation
0	0 0000
1	0 0001
2	0 0010
3	0 0011
4	0 0100
5	0 0101
6	0 0110
7	0 0111
8	0 1000
9	0 1001
10	1 0000
11	1 0001
12	1 0010
13	1 0011
14	1 0100
15	1 0101

Usage

```
void waveform_binary_to_bcd(  Waveform* out_wave,
                              Waveform* in_wave );
```

where:

out_wave specifies the output waveform. See Description.

in_wave specifies the input waveform. See Description.

Example

???

3.27.20.4 waveform_bcd_to_binary()

See Waveform Overview, Waveform Conversion Functions.

Description

This is the inverse function of `waveform_binary_to_bcd()`.

Only positive input waveform sample values will be converted. Negative values are set to binary 0.

Usage

```
void waveform_bcd_to_binary(  Waveform* out_wave,
                              Waveform* in_wave );
```

where:

`out_wave` specifies the output waveform. See Description.

`in_wave` specifies the input waveform. See Description.

Example

???

3.27.20.5 waveform_binary_to_ones_complement()

See Waveform Overview, Waveform Conversion Functions.

Description

The `waveform_binary_to_ones_complement()` function will convert the binary representation of each sample value of the input waveform to its equivalent one's complement binary value. Note the following:

- One's complement is a representation where negative integers are inverted (all of the bits are flipped).

- There are two potential representations for zero. For an eight bit word, 00000000 is the usual form of zero and is sometimes called +0, however, 11111111 is also equal to zero and is referred to as -0. `waveform_binary_to_ones_complement()` always uses the +0 representation for zero.
- A `bitwidth` of *N* can represent values in the range:

  $$-2^{N-1}+1 \quad \text{to} \quad 2^{N-1}-1$$

  Values will be clamped to fit within the range.
- The table below illustrates the values available for a `bitwidth` of four:

Binary Value	One's Complement Representation
-7	1000
-6	1001
-5	1010
-4	1011
-3	1100
-2	1101
-1	1110
-0*	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Usage

```
void waveform_binary_to_ones_complement(Waveform* out_wave,
                                        Waveform* in_wave,
                                        int bitwidth );
```

where:

out_wave specifies the output waveform. See Description.

in_wave specifies the input waveform. See Description.

bitwidth specifies the number of valid bits in each the output waveform sample value. Bits above **bitwidth** are set to zero.

Example

???

3.27.20.6 waveform_ones_complement_to_binary()

See Waveform Overview, Waveform Conversion Functions.

Description

This is the inverse function of `waveform_ones_complement_to_binary()`.

Illegal values are converted to binary zero. Both +0 and -0 encodings are converted to zero. See `waveform_ones_complement_to_binary()`.

Usage

```
void waveform_ones_complement_to_binary(Waveform* out_wave,
                                        Waveform* in_wave,
                                        int bitwidth );
```

where:

out_wave specifies the output waveform. See Description.

in_wave specifies the input waveform. See Description.

bitwidth specifies the number of valid bits in each the output waveform sample value. Bits above **bitwidth** are set to zero.

Example

???

3.27.20.7 waveform_binary_to_twos_complement()

See Waveform Overview, Waveform Conversion Functions.

Description

The `waveform_binary_to_twos_complement()` function will convert the binary representation of each sample value of the input waveform to its equivalent two's complement binary value. Note the following:

- `RLONG` values are stored in 32 or 64 bit two's complement integer arrays.
- `waveform_binary_to_twos_complement()` allows integer sample values to be translated to different word widths.
- A `bitwidth` of *N* can represent values in the range:

  $$-2^{N-1}+1 \quad \text{to} \quad 2^{N-1}-1$$

  Values will be clamped to fit within the range.
- The table below illustrates the values available for a `bitwidth` of four:

Binary Value	Two's complement Representation
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000

Binary Value	Two's complement Representation
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Usage

```
void waveform_binary_to_twos_complement(Waveform* out_wave,
                                        Waveform* in_wave,
                                        int bitwidth );
```

where:

out_wave specifies the output waveform. See Description.

in_wave specifies the input waveform. See Description.

bitwidth specifies the number of valid bits in each the output waveform sample value. Bits above **bitwidth** are set to zero.

Example

???

3.27.20.8 waveform_twos_complement_to_binary()

See Waveform Overview, Waveform Conversion Functions.

Description

This is the inverse function of `waveform_twos_complement_to_binary()`.

Illegal values are converted to binary zero.

Usage

```
void waveform_twos_complement_to_binary(Waveform* out_wave,
                                        Waveform* in_wave,
                                        int bitwidth );
```

where:

`out_wave` specifies the output waveform. See Description.

`in_wave` specifies the input waveform. See Description.

`bitwidth` specifies the number of valid bits in each the output waveform sample value. Bits above **`bitwidth`** are set to zero.

Example

???

3.27.20.9 waveform_binary_to_offset_binary()

See Waveform Overview, Waveform Conversion Functions.

Description

The `waveform_binary_to_offset_binary()` function will convert the binary representation of each sample value of the input waveform to its equivalent offset binary value. Note the following:

- Offset binary is the representation that results from adding an offset value to each sample value. This has the effect of shifting the the representable range so that the entire range is encoded as positive values.
- A `bitwidth` of *N* can represent values in the range:

  $$-2^{N-1}+1 \quad \text{to} \quad 2^{N-1}-1$$

  These values are offset by:

  $$2^{N}$$

to yield values in the range of:

$$0 \quad \text{to} \quad 2^N - 1$$

- Values will be clamped to fit within the range.
- The table below illustrates the values available for a bitwidth of four:

Binary Value	Offset Binary Representation
-8	0000
-7	0001
-6	0010
-5	0011
-4	0100
-3	0101
-2	0110
-1	0111
0	1000
1	1001
2	1010
3	1011
4	1100
5	1101
6	1110
7	1111

Usage

```
void waveform_binary_to_offset_binary(Waveform* out_wave,
                                      Waveform* in_wave,
                                      int bitwidth );
```

where:

out_wave specifies the output waveform. See Description.

in_wave specifies the input waveform. See Description.

bitwidth specifies the number of valid bits in each the output waveform sample value. Bits above **bitwidth** are set to zero.

Example

???

3.27.20.10 waveform_offset_binary_to_binary()

See Waveform Overview, Waveform Conversion Functions.

Description

This is the inverse function of `waveform_binary_to_offset_binary()`.

Illegal values are converted to 0.

Usage

```
void waveform_offset_binary_to_binary(Waveform* out_wave,
                                      Waveform* in_wave,
                                      int bitwidth );
```

where:

out_wave specifies the output waveform. See Description.

in_wave specifies the input waveform. See Description.

bitwidth specifies the number of valid bits in each the output waveform sample value. Bits above **bitwidth** are set to zero.

Example

???

3.27.20.11 waveform_binary_to_sign_and_magnitude()

See Waveform Overview, Waveform Conversion Functions.

Description

The `waveform_binary_to_sign_and_magnitude()` function will convert the binary representation of each sample value of the input waveform to its equivalent sign and magnitude binary value. Note the following:

- The sign and magnitude representation uses a separate bit (the MSB) of each sample value to represent the sign of the integer value. A zero indicates positive and a one indicates negative.
- The remaing bits of the sample value represent the magnitude value (absolute value) of the integer. As with one's complement (see `waveform_ones_complement_to_binary()`), this yields two representations for zero (+0 and -0). `waveform_binary_to_sign_and_magnitude()` always uses the +0 representation for zero.
- A `bitwidth` of *N* can represent values in the range:

  $$-2^{N-1}+1 \qquad \text{to} \qquad 2^{N-1}-1$$

  Values will be clamped to fit within the range.
- The table below illustrates the values available for a `bitwidth` of four:

Binary Value	Sign & Magnitude Representation
-7	1111
-6	1110
-5	1101
-4	1100
-3	1011
-2	1010
-1	1001
-0*	1000
0	0000
1	0001
2	0010
3	0011

Binary Value	Sign & Magnitude Representation
4	0100
5	0101
6	0110
7	0111
* Negative zero (-0) is converted to 0.	

Usage

```
void waveform_binary_to_sign_and_magnitude(Waveform* out_wave,
                                           Waveform* in_wave,
                                           int bitwidth );
```

where:

out_wave specifies the output waveform. See Description.

in_wave specifies the input waveform. See Description.

bitwidth specifies the number of valid bits in each the output waveform sample value. Includes 1 bit for the sign with remaining bits used for magnitude. Bits above **bitwidth** are set to zero.

Example

???

3.27.20.12 waveform_sign_and_magnitude_to_binary()

See Waveform Overview, Waveform Conversion Functions.

Description

This is the inverse function of `waveform_binary_to_sign_and_magnitude()`.

Illegal values are converted to 0. Both +0 and -0 encodings are converted to 0 (see `waveform_ones_complement_to_binary()`).

Usage

```
void waveform_sign_and_magnitude_to_binary(Waveform* out_wave,
                                           Waveform* in_wave,
                                           int bitwidth );
```

where:

`out_wave` specifies the output waveform. See Description.

`in_wave` specifies the input waveform. See Description.

`bitwidth` specifies the number of valid bits in each the output waveform sample value. Bits above **`bitwidth`** are set to zero.

Example

???

3.27.21 Waveform Boolean Functions

See Waveform Overview.

- Waveform Logical Functions
- `waveform_select_indices()`
- `waveform_select_elements()`
- `waveform_selective_merge()`
- `waveform_reorder()`

3.27.21.1 Waveform Logical Functions

See Waveform Overview, Waveform Boolean Functions.

Description

waveform_logical_and()

The `waveform_logical_and()` function creates an output waveform containing a sample-by-sample logical AND of two specified input waveforms.

The `waveform_logical_or()` function creates an output waveform containing a sample-by-sample logical OR of two specified input waveforms.

The `waveform_logical_xor()` function creates an output waveform containing a sample-by-sample logical XOR (exclusive OR) of two specified input waveforms.

The `waveform_logical_not()` function creates an output waveform containing a sample-by-sample complement of one specified input waveform.

Note the following:

- The name of the function determines the logical operation i.e. *AND*, *OR*, *XOR* (exclusive OR), *NOT* (invert). The *NOT* operation evaluates values from a single input waveform, whereas the other operations require two values, one from each of two input waveforms.
- The two input waveforms must be the same Size (see `waveform_get_size()`). If not, an error message is output in the appropriate controller window, values from the larger waveform are ignored, and testing otherwise continues.
- The sample values of the two input waveform are treated as boolean values, where 0 = FALSE and any non-zero value = TRUE. The two input waveforms must be defined using the `RRECT_WAVE` or `RLONG_WAVE` notation.
- The sample values of the output waveform will be `RLONG_WAVE`, and will contain only the values '1' and '0', representing TRUE and FALSE. Note that any non-zero value in the input waveform is considered TRUE.

Also note the following about the output waveform:

- Y_units is set to `SCALE_BOOLEAN`
- X_start is set to 0
- X_increment is set = 1
- X_units is set to `SCALE_NOXUNITS`

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

*X_unitsY_unitsX_increment*Usage

```
void waveform_logical_and(  Waveform* out_wave,
                            Waveform* in_wave1,
                            Waveform* in_wave2 );

void waveform_logical_or(Waveform* out_wave,
                         Waveform* in_wave1,
                         Waveform* in_wave2 );
```

```
void waveform_logical_xor(  Waveform* out_wave,
                            Waveform* in_wave1,
                            Waveform* in_wave2 );
void waveform_logical_not(  Waveform* out_wave,
                            Waveform* in_wave );
```

where:

out_wave specifies the output waveform. See Description.

in_wave1 and **in_wave2** specify the two input waveforms. See Description.

Example

???

3.27.21.2 waveform_select_indices()

See Waveform Overview, Waveform Boolean Functions.

Description

The `waveform_select_indices()` function is used to create a list of input waveform sample value *locations* (indices) which contain *TRUE* values

. Note the following:

- The input waveform sample values are treated as boolean, where 0 = FALSE and non-zero = TRUE. The input waveform must be defined using the RRECT_WAVE or RLONG_WAVE notation.
- The `select` waveform must be defined using the RRECT_WAVE or RLONG_WAVE notation.

- `waveform_select_indices()` evaluates each sample value of the `select` waveform. For each non-zero value (= `TRUE`), the index of that sample value is placed in the output waveform (in `RLONG_WAVE` notation). The number of sample values in the output waveform (see `waveform_get_size()`) will be equal to the number of non-zero values in the `select` waveform.

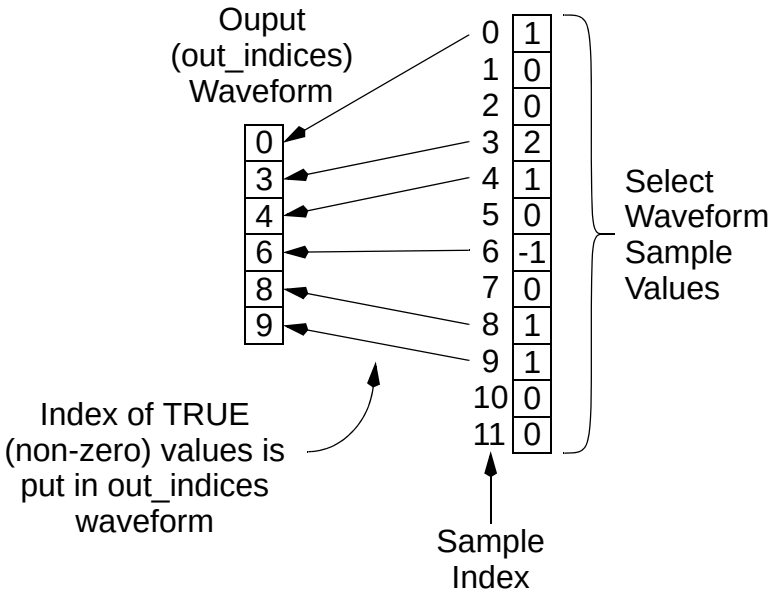


Figure-59: `waveform_select_indices()` **User Model**

Usage

```
void waveform_select_indices( Waveform* out_indices,
                              Waveform* select );
```

where:

`out_indices` specifies the output waveform.

`select` specifies the waveform to be evaluated. See Description.

Example

???

3.27.21.3 waveform_select_elements()

See Waveform Overview, Waveform Boolean Functions.

Description

The `waveform_select_elements()` function copies selected sample values from an input waveform to an output waveform based on the boolean sample values of a `select` waveform. Note the following:

- The input waveform may be any Type (see Waveform* Attributes, Waveform Sample Value Notations). The output waveform will be of the same type, the X_units and Y_units of the output waveform are set to match the input waveform.
- The input waveform and the select waveform must be the same Size (see `waveform_get_size()`). If not, an error message is output in the appropriate controller window, values from the larger waveform are ignored, and testing otherwise continues.
- The sample values of the `select` waveform are treated as boolean values, where 0 = FALSE and any non-zero value = TRUE. The `select` waveform must be defined using the `RRECT_WAVE` or `RLONG_WAVE` notation.
- `waveform_select_elements()` uses the `select` waveform as a mask, to determine which values from the input waveform to put into the output waveform. For each TRUE value in the `select` waveform, the corresponding value in the

input waveform is put into the output waveform. The number of sample values in the output waveform (Size) will be equal to the number of non-zero values in the `select` waveform.

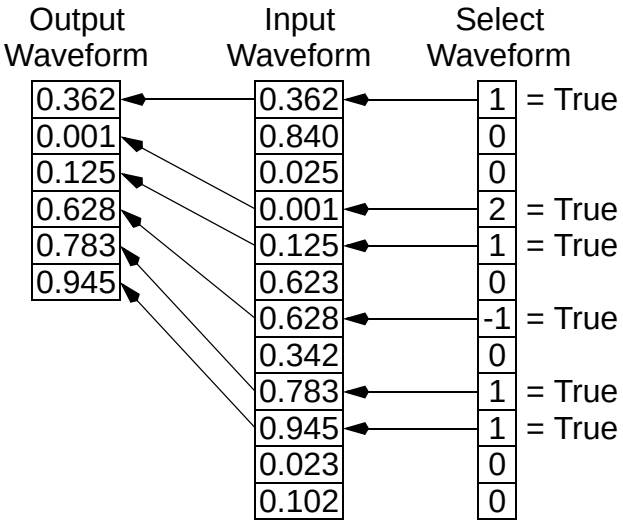


Figure-60: `waveform_select_elements()` **User Model**

- The following code...

```
waveform_select_elements( out, in, select );
```

... is equivalent to

```
waveform_select_indices( indices, select );
waveform_reorder( out, in, indices );
```

See `waveform_select_indices()`, `waveform_bitwise_reorder()`.

Usage

```
void waveform_select_elements(Waveform* out_wave,
                              Waveform* in_wave,
                              Waveform* select );
```

where:

`out_wave` specifies the output waveform. See Description.

`in_wave` specifies the input waveform. See Description.

`select` specifies the waveform to be used as the mask. See Description.

Example

???

3.27.21.4 waveform_selective_merge()

See Waveform Overview, Waveform Boolean Functions.

Description

The `waveform_selective_merge()` function selectively copies sample values from two input waveforms into an output waveform based on the boolean sample values of a select waveform.

Note the following:

- The input waveforms may be defined using any notation (see Waveform Sample Value Notations) but must be defined using the same notation. If not, `waveform_selective_merge()` returns immediately, an error message is output in the appropriate controller window, the output waveform is corrupt, and testing otherwise continues.
- The two input waveforms and the `select` waveform must be the same Size (see `waveform_get_size()`). If not, an error message is output in the appropriate controller window, values from the larger waveform(s) are ignored, and testing otherwise continues.
- The sample values of the `select` waveform are treated as boolean values, where 0 = FALSE and any non-zero value = TRUE. The `select` waveform must be defined using the `RRECT_WAVE` or `RLONG_WAVE` notation.
- `waveform_selective_merge()` copies values from the first input waveform when the corresponding sample value of the `select` waveform is TRUE, otherwise the value from the second input waveform is copied to the output waveform. The

number of sample values in the output waveform (Size) will be equal to the number of sample values of the smallest of the two input or the `select` waveform (which as noted above should be of equal size).

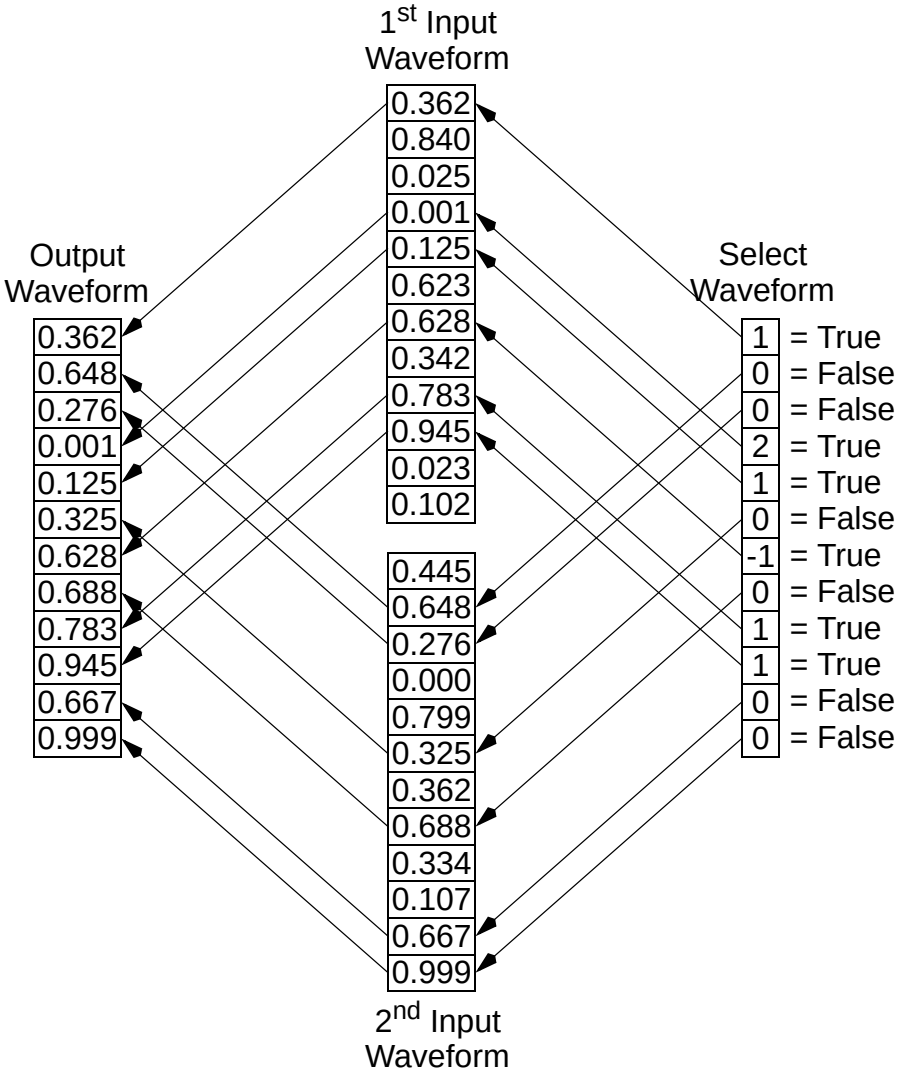


Figure-61: `waveform_selective_merge()` **User Model**

- The output waveform will be of the same Type as the first input waveform, the X_units and Y_units of the output waveform are set to match the first input waveform.

Usage

```
void waveform_selective_merge(Waveform* out_wave,
                              Waveform* in_wave1,
                              Waveform* in_wave2,
                              Waveform* select );
```

where:

`out_wave` specifies the output waveform. See Description.

`in_wave1` and **`in_wave2`** specifies the two input waveforms. See Description.

`select` specifies the waveform to be used as the merge map. See Description.

Example

???

3.27.21.5 waveform_reorder()

See Waveform Overview, Waveform Boolean Functions.

Description

The `waveform_reorder()` function is used to reorder the sample values of a specified waveform based on an indices waveform (see `waveform_select_indices()`). Note the following:

- The input waveform may defined using any notation. See Waveform Sample Value Notations.
- Each sample value of the `indices` waveform selects an element of the input waveform to be copied into the corresponding sample value of the output waveform.
- The `indices` waveform must be defined using RLONG_WAVE notation.
- The `indices` waveform sample values must consist of valid indices into the input waveform (i.e. 0 through Size-1, inclusive).
- The output waveform will be defined using the same notation as the input waveform and will be the same Size as the `indices` waveform. Both X_units and Y_units of the output waveform will match the input waveform.

- Input sample values may be duplicated in the output by including duplicate values in the `indices` waveform.
- Input sample values may be excluded from the output by omitting value(s) from the `indices` waveform.
- Any errors cause execution to be aborted, an error message will be displayed in the appropriate controller window, and `BAD_WAVE` will be returned in the `out_wave` argument.

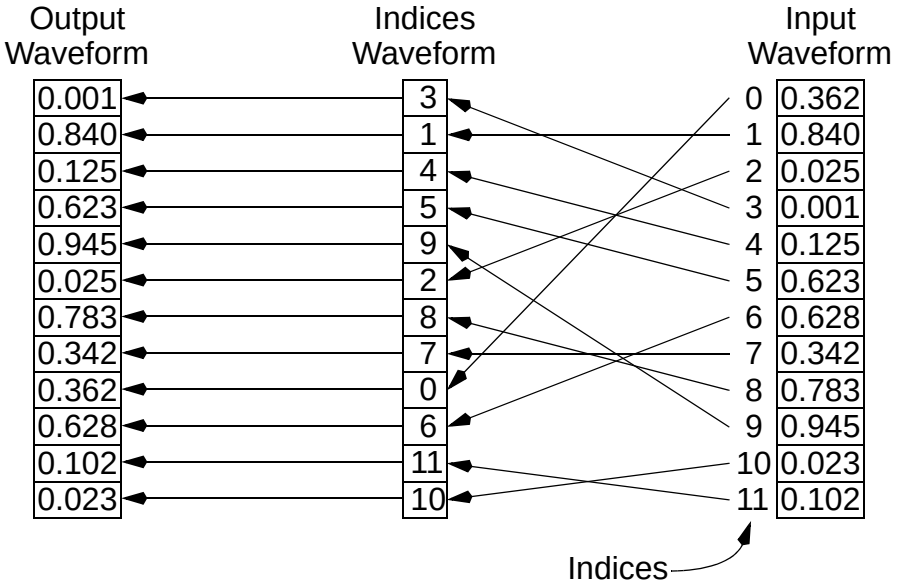


Figure-62: `waveform_reorder()` User Model

To reorder the bits of each sample value see `waveform_bitwise_reorder()`.

Usage

```
void waveform_reorder( Waveform* out_wave,
                       Waveform* in_wave,
                       Waveform* indices );
```

where:

`out_wave` specifies the output waveform. See Description.

`in_wave` specifies the input waveform. See Description.

`indices` specifies the waveform to be used as the reorder map. See Description.

Example

???

3.27.22 Waveform Bitwise Functions

See Waveform Overview.

- `waveform_bitwise_or()`
- `waveform_bitwise_and()`
- `waveform_bitwise_xor()`
- `waveform_bitwise_shift_left()`
- `waveform_bitwise_shift_right()`
- `waveform_bitwise_rotate_left()`
- `waveform_bitwise_rotate_right()`
- `waveform_bitwise_reverse()`
- `waveform_bitwise_reorder()`

3.27.22.1 waveform_bitwise_or()

See Waveform Overview, Waveform Bitwise Functions.

Description

The `waveform_bitwise_or()` function is used to perform a bitwise logical OR operation on the sample values of a specified waveform. Two forms are available:

- Each sample value of the specified waveform is OR'ed with a corresponding sample value of a second specified waveform. If the two waveforms differ in Size, the larger waveform will be truncated to match the size of the smaller waveform.
- Each sample value of the specified waveform is OR'ed with a specified scalar value.

The `waveform_bitwise_or()` function only operates correctly with waveforms defined using the `RLONG_WAVE` one-part notation. When this rule is violated, operation will be

aborted, an error message will be displayed in the appropriate controller window, and BAD_WAVE will be returned in the `out_wave` argument.

Waveforms may contain nonnegative integers up to ($2^{\wedge 53}$ -1) in value i.e. up to 9,007,199,254,740,991. Results of operations involving negative values are undefined.

Usage

```
void waveform_bitwise_or(Waveform* out_wave,
                         Waveform* in_wave1,
                         Waveform* in_wave2 );
void waveform_bitwise_or(Waveform* out_wave,
                         Waveform* in_wave,
                         __int64 scalar );
```

where:

`out_wave` contains the waveform resulting from the bitwise OR operation.

`in_wave1` represents the input waveform. Each sample value of **`in_wave1`** will be AND'ed with the corresponding sample value of **`in_wave2`**.

`in_wave` represents the input waveform. Each sample value of **`in_wave`** will be AND'ed with **`scalar`**.

`in_wave2` contains sample values to be AND'ed with the sample values of **`in_wave1`**. See Description.

`scalar` contain a single sample value to be AND'ed with each sample value of **`in_wave`**. See Description.

Example

???

3.27.22.2 waveform_bitwise_and()

See Waveform Overview, Waveform Bitwise Functions.

Description

The `waveform_bitwise_and()` function is used to perform a bitwise logical AND operation on the sample values of a specified waveform. Two forms are available:

- Each sample value of the specified waveform is AND'ed with a corresponding sample value of a second specified waveform. If the two waveforms differ in Size, the larger waveform will be truncated to match the size of the smaller waveform.
- Each sample value of the specified waveform is AND'ed with a specified scalar value.

The `waveform_bitwise_and()` function only operates correctly with waveforms defined using the `RLONG_WAVE` one-part notation. When this rule is violated, operation will be aborted, an error message will be displayed in the appropriate controller window, and `BAD_WAVE` will be returned in the `out_wave` argument.

Waveforms may contain nonnegative integers up to ($(2^{\wedge 53}$ -1) in value i.e. up to 9,007,199,254,740,991. Results of operations involving negative values are undefined.

Usage

```
void waveform_bitwise_and(  Waveform* out_wave,
                            Waveform* in_wave1,
                            Waveform* in_wave2 );
void waveform_bitwise_and(  Waveform* out_wave,
                            Waveform* in_wave,
                            __int64 scalar );
```

where:

`out_wave` contains the waveform resulting from the bitwise AND operation.

`in_wave1` represents the input waveform. Each sample value of **`in_wave1`** will be AND'ed with the corresponding sample value of **`in_wave2`**.

`in_wave` represents the input waveform. Each sample value of **`in_wave`** will be AND'ed with **`scalar`**.

`in_wave2` contains sample values to be AND'ed with the sample values of **`in_wave1`**. See Description.

`scalar` contain a single sample value to be AND'ed with each sample value of **`in_wave`**. See Description.

Example

???

3.27.22.3 waveform_bitwise_xor()

See Waveform Overview, Waveform Bitwise Functions.

Description

The `waveform_bitwise_xor()` function is used to perform a bitwise logical XOR (exclusive OR) operation on the sample values of a specified waveform. Two forms are available:

- Each sample value of the specified waveform is XOR'ed with a corresponding sample value of a second specified waveform. If the two waveforms differ in Size, the larger waveform will be truncated to match the size of the smaller waveform.
- Each sample value of the specified waveform is XOR'ed with a specified scalar value.

The `waveform_bitwise_xor()` function only operates correctly with waveforms defined using the `RLONG_WAVE` one-part notation. When this rule is violated, operation will be aborted, an error message will be displayed in the appropriate controller window, and `BAD_WAVE` will be returned in the `out_wave` argument.

Waveforms may contain nonnegative integers up to ($(2^{\wedge 53} - 1)$ in value i.e. up to 9,007,199,254,740,991). Results of operations involving negative values are undefined.

Usage

```
waveform_bitwise_xor(       Waveform*out_wave,
                            Waveform* in_wave1,
                            Waveform* in_wave2 );
void waveform_bitwise_xor(  Waveform* out_wave,
                            Waveform* in_wave,
                            __int64 scalar );
```

where:

`out_wave` contains the waveform resulting from the bitwise XOR operation.

`in_wave1` represents the input waveform. Each sample value of **`in_wave1`** will be XOR'ed with the corresponding sample value of **`in_wave2`**.

`in_wave` represents the input waveform. Each sample value of **`in_wave`** will be XOR'ed with **`scalar`**.

in_wave2 contains sample values to be XOR'ed with the sample values of **in_wave1**. See Description.

scalar contain a single sample value to be XOR'ed with each sample value of **in_wave**. See Description.

Example

???

3.27.22.4 waveform_bitwise_shift_left()

See Waveform Overview, Waveform Bitwise Functions.

Description

The `waveform_bitwise_shift_left()` function is used to perform a bitwise left shift operation on the sample values of a specified waveform. Two forms are available:

- Each sample value of the specified waveform is shifted left the number bit positions specified by a corresponding sample value of a second specified waveform. If the two waveforms differ in Size, the larger waveform will be truncated to match the size of the smaller waveform.
- Each sample value of the specified waveform is shifted left the number of bit positions specified by a scalar value.

Bits which are shifted off the left end are lost. The shift fill bit value is zero. If the shift value is negative, the input sample value is shifted *right* by the absolute shift value.

The `waveform_bitwise_shift_left()` function only operates correctly with waveforms defined using the `RLONG_WAVE` one-part notation. When this rule is violated, operation will be aborted, an error message will be displayed in the appropriate controller window, and `BAD_WAVE` will be returned in the `out_wave` argument.

Waveforms may contain nonnegative integers up to ($2^{\wedge 53}$ -1) in value i.e. up to 9,007,199,254,740,991). Results of operations involving negative values are undefined.

Usage

```
void waveform_bitwise_shift_left(Waveform* out_wave,
                                 Waveform* in_wave1,
                                 Waveform* in_wave2 );
```

```
void waveform_bitwise_shift_left(Waveform* out_wave,
                                 Waveform* in_wave,
                                 int shift DEFAULT_VALUE( 1 ) );
```

where:

out_wave contains the waveform resulting from the bitwise left shift operation.

in_wave1 represents the input waveform. Each sample value **in_wave1** will be shifted left the number of bit positions specified by the corresponding sample value of **in_wave2**.

in_wave represents the input waveform. Each sample value of **in_wave** will be shifted left the number of bit positions specified by **shift**.

in_wave2 contains sample values each of which specifies the number of bit positions that the corresponding sample value of **in_wave1** will be left shifted. See Description.

shift is optional, and if used specifies a single value which specifies the number of bit positions that each sample value of **in_wave** will be left shifted. Default = 1. See Description.

Example

???

3.27.22.5 waveform_bitwise_shift_right()

See Waveform Overview, Waveform Bitwise Functions.

Description

The `waveform_bitwise_shift_right()` function is used to perform a bitwise right shift operation on the sample values of a specified waveform. Two forms are available:

- Each sample value of the specified waveform is shifted right the number bit positions specified by a corresponding sample value of a second specified waveform. If the two waveforms differ in Size, the larger waveform will be truncated to match the size of the smaller waveform.
- Each sample value of the specified waveform is shifted right the number of bit positions specified by a scalar value.

Bits which are shifted off the right end are lost. The shift fill bit value is zero. If the shift value is negative, the input sample value is shifted *left* by the absolute shift value.

The waveform_bitwise_shift_right() function only operates correctly with waveforms defined using the RLONG_WAVE one-part notation. When this rule is violated, operation will be aborted, an error message will be displayed in the appropriate controller window, and BAD_WAVE will be returned in the out_wave argument.

Waveforms may contain nonnegative integers up to ($2^{\wedge 53}$ -1) in value i.e. up to 9,007,199,254,740,991. Results of operations involving negative values are undefined.

Usage

```
waveform_bitwise_shift_right(       Waveform* out_wave,
                                    Waveform* in_wave1,
                                    Waveform* in_wave2 );
void waveform_bitwise_shift_right( Waveform* out_wave,
                                    Waveform* in_wave,
                                    int shift DEFAULT_VALUE( 1 ) );
```

where:

out_wave contains the waveform resulting from the bitwise right shift operation.

in_wave1 represents the input waveform. Each sample value **in_wave1** will be shifted right the number of bit positions specified by the corresponding sample value of **in_wave2**.

in_wave represents the input waveform. Each sample value of **in_wave** will be shifted right the number of bit positions specified by **shift**.

in_wave2 contains sample values each of which specifies the number of bit positions that the corresponding sample value of **in_wave1** will be right shifted. See Description.

shift is optional, and if used specifies a single value which specifies the number of bit positions that each sample value of **in_wave** will be right shifted. Default = 1. See Description.

Example

???

3.27.22.6 waveform_bitwise_rotate_left()

See Waveform Overview, Waveform Bitwise Functions.

Description

The `waveform_bitwise_rotate_left()` function is used to perform a bitwise left rotate operation on the sample values of a specified waveform. Two forms are available:

- Each sample value of the specified waveform is rotated left the number bit positions specified by a corresponding sample value of a second specified waveform. If the two waveforms differ in Size, the larger waveform will be truncated to match the size of the smaller waveform.
- Each sample value of the specified waveform is rotated left the number of bit positions specified by a scalar value.

Bits which rotate off the left end are appended to the right end of the sample value. If the rotate value is negative, the input sample value is rotated *right* by the absolute rotate value.

The `waveform_bitwise_rotate_left()` function only operates correctly with waveforms defined using the `RLONG_WAVE` one-part notation. When this rule is violated, operation will be aborted, an error message will be displayed in the appropriate controller window, and `BAD_WAVE` will be returned in the `out_wave` argument.

Waveforms may contain nonnegative integers up to ($2^{\wedge 53}$ -1) in value i.e. up to 9,007,199,254,740,991. Results of operations involving negative values are undefined.

Usage

```
void waveform_bitwise_rotate_left( Waveform* out_wave,
                                   Waveform* in_wave1,
                                   Waveform* in_wave2,
                                   int bitwidth );
void waveform_bitwise_rotate_left( Waveform* out_wave,
                                   Waveform* in_wave,
                                   int bitwidth,
                                   int shift DEFAULT_VALUE( 1 ) );
```

where:

`out_wave` contains the waveform resulting from the left rotate operation.

`in_wave1` represents the input waveform. Each sample value **`in_wave1`** will be left rotated the number of bit positions specified by the corresponding sample value of **`in_wave2`**.

`in_wave` represents the input waveform. Each sample value of **`in_wave`** will be left rotated the number of bit positions specified by **`shift`**.

in_wave2 contains sample values each of which specifies the number of bit positions that the corresponding sample value of **in_wave1** will be left rotated. See Description.

bitwidth specifies the number of bits of the sample value to be rotated. Any bits outside of the designated **bitwidth** are discarded and set to zero in the output waveform sample value.

shift is optional, and if used specifies a single value which specifies the number of bit positions that each sample value of **in_wave** will be left rotated. See Description. Default = 1. If **shift** is larger than **bitwidth**, **shift** is reduced modulo **bitwidth**.

Example

???

3.27.22.7 waveform_bitwise_rotate_right()

See Waveform Overview, Waveform Bitwise Functions.

Description

The `waveform_bitwise_rotate_right()` function is used to perform a bitwise right rotate operation on the sample values of a specified waveform. Two forms are available:

- Each sample value of the specified waveform is rotated right the number bit positions specified by a corresponding sample value of a second specified waveform. If the two waveforms differ in Size, the larger waveform will be truncated to match the size of the smaller waveform.
- Each sample value of the specified waveform is rotated right the number of bit positions specified by a scalar value.

Bits which rotate off the right end are prepended to the left end of the sample value. If the rotate value is negative, the input sample value is rotated *left* by the absolute rotate value.

The `waveform_bitwise_rotate_right()` function only operates correctly with waveforms defined using the `RLONG_WAVE` one-part notation. When this rule is violated, operation will be aborted, an error message will be displayed in the appropriate controller window, and `BAD_WAVE` will be returned in the `out_wave` argument.

Waveforms may contain nonnegative integers up to ($2^{\wedge 53}$ -1) in value i.e. up to 9,007,199,254,740,991. Results of operations involving negative values are undefined.

Usage

```
void waveform_bitwise_rotate_right(Waveform* out_wave,
                                   Waveform* in_wave1,
                                   Waveform* in_wave2,
                                   int bitwidth );
void waveform_bitwise_rotate_right(Waveform* out_wave,
                                   Waveform* in_wave,
                                   int bitwidth,
                                   int shift DEFAULT_VALUE( 1 ) );
```

where:

out_wave contains the waveform resulting from the right rotate operation.

in_wave1 represents the input waveform. Each sample value **in_wave1** will be right rotated the number of bit positions specified by the corresponding sample value of **in_wave2**.

in_wave represents the input waveform. Each sample value of **in_wave** will be right rotated the number of bit positions specified by **shift**.

in_wave2 contains sample values each of which specifies the number of bit positions that the corresponding sample value of **in_wave1** will be right rotated. See Description.

bitwidth specifies the number of bits of the sample value to be rotated. Any bits outside of the designated **bitwidth** are discarded and set to zero in the output waveform sample value.

shift is optional, and if used specifies a single value which specifies the number of bit positions that each sample value of **in_wave** will be right rotated. See Description. Default = 1. If **shift** is larger than **bitwidth**, **shift** is reduced modulo **bitwidth**.

Example

???

3.27.22.8 waveform_bitwise_reverse()

See Waveform Overview, Waveform Bitwise Functions.

Description

The `waveform_bitwise_reverse()` function is used to perform a bitwise order reversal on the sample values of a specified waveform. The LSB bit becomes the MSB bit, the LSB+1 bit becomes the MSB-1 bit, etc.

Note: this function should not be confused with `waveform_reverse()` which reverses the order of the sample values in a specified waveform.

The `waveform_bitwise_reverse()` function only operates correctly with waveforms defined using the `RLONG_WAVE` one-part notation. When this rule is violated, operation will be aborted, an error message will be displayed in the appropriate controller window, and `BAD_WAVE` will be returned in the `out_wave` argument.

Waveforms may contain nonnegative integers up to ($2^{\wedge 53}$ -1) in value i.e. up to 9,007,199,254,740,991. Results of operations involving negative values are undefined.

Usage

```
waveform_bitwise_reverse(     Waveform* out_wave,
                              Waveform* in_wave,
                              int bitwidth );
```

where:

`out_wave` contains the waveform resulting from the bitwise reverse operation.

`in_wave` represents the input waveform.

`bitwidth` specifies the number of bits of the sample value to reverse. Any bits outside of the designated **`bitwidth`** are discarded and set to zero in the output waveform sample value.

Example

???

3.27.22.9 waveform_bitwise_reorder()

See Waveform Overview, Waveform Bitwise Functions.

Description

The `waveform_bitwise_reorder()` function is used to reorder the bits of each sample value of a specified waveform. Note the following:

- The specification for how bits are reordered is defined by a second waveform, called the `bit_indices` waveform. Each output sample value is [re]ordered identically.
- Each output waveform sample value is formed by taking bits from one input waveform sample value, in the order specified by values from the `bit_indices` waveform. Bit positions are numbered from the LSB to the MSB starting with zero.
- The number of bits in each output waveform sample value is determined by the number of values in the `bit_indices` waveform. It is not required that the sample values of the output waveform have the same number of bits as the input waveform.
- Input sample value bits may be duplicated in the output by having a given bit position appear more than once in the `bit_indices` waveform.
- Bits from the input sample can be excluded from the output sample by omitting a given bit position from the `bit_indices` waveform.
- It is possible to force output bit(s) to logic-0 zero or logic-1 using the following values in the `bit_indices` waveform:
  -1 : sets the output sample value bit to logic-0
  -2 : sets the output sample value bit to logic-1

The `waveform_bitwise_reorder()` function only operates correctly with waveforms defined using the `RLONG_WAVE` one-part notation. When this rule is violated, operation will be aborted, an error message will be displayed in the appropriate controller window, and `BAD_WAVE` will be returned in the `out_wave` argument.

Waveforms may contain nonnegative integers up to ($2^{\wedge 53}$ -1) in value i.e. up to 9,007,199,254,740,991. Results of operations involving negative values are undefined.

To reorder the sample values see `waveform_reorder()`.

Usage

```
void waveform_bitwise_reorder(Waveform* out_wave,
                              Waveform* in_wave,
                              Waveform* bit_indices );
```

where:

out_wave contains the waveform resulting from the reorder operation.

in_wave represents the input waveform.

bit_indices specifies the order bits are taken from each sample of **in_wave** and put into **out_wave**. It also controls the number of bits in **out_wave**. See Description.

Examples

Example 1:

This example has the effect of swapping the two four-bit nibbles of the input waveform. Any additional bits of the input waveform are discarded:

```
// Create a bit_indices waveform, containing 8 values.
// The first value specifies which bit from the input waveform will
// become the LSB bit in the output waveform, etc.
long nibble_swap[ 8 ] = { 4, 5, 6, 7, 0, 1, 2, 3 };

// Create a Waveform* container to store this
waveform_set_rlong( bit_indices, // waveform_set_rlong()
                    8,
                    nibble_swap,
                    SCALE_NOYUNITS,
                    0,
                    1,
                    SCALE_NOXUNITS );
waveform_bitwise_reorder( WFout, WFin, bit_indices );
```

Example 2:

This example does the following:

- Discard bits 3, 4, and 5 of the input sample
- Shift bits 0, 1, and 2 three places to the left
- Force output bit 1 to a logic-1
- Force the other vacated bits to logic-0

- Transfer the remaing bits of the ten bit input to the output in place.

```
// Create bit_indices waveform, containing 10 values
long bit_map[ 10 ] = {
   -1,  // Output bit 0: vacated, force to 0
   -2,  // Output bit 1: force to logic-1
   -1,  // Output bit 2: vacated, force to 0
   0,   // Output bit 3: take input bit 0 (shift left 3)
   1,   // Output bit 4: take input bit 1 (shift left 3)
   2,   // Output bit 5: take input bit 2 (shift left 3)
   6,   // Output bit 6: take bit 6 (no change)
   7,   // Output bit 7: take bit 7 (no change)
   8,   // Output bit 8: take bit 8 (no change)
   9    // Output bit 9: take bit 9 (no change)
};
// Create a Waveform* container to store this
waveform_set_rlong( bit_indices, // waveform_set_rlong()
                    8,
                    bit_map,
                    SCALE_NOYUNITS,
                    0,
                    1,
                    SCALE_NOXUNITS );
waveform_bitwise_reorder( WFout, WFin, bit_indices );
```

3.27.23 Waveform Logrithmic Functions

See Waveform Overview.

- waveform_log(), waveform_log10()
- waveform_exp(), waveform_exp10()
- waveform_power()

3.27.23.1 waveform_log(), waveform_log10()

See Waveform Overview, Waveform Logrithmic Functions

Description

The `waveform_log()` function is used to return the natural log of each sample value in a specified waveform. The returned values are in the form of a waveform where each output waveform sample value is the log of the corresponding sample value of the input waveform.

The `waveform_log10()` function is used to return the common log of each sample value in a specified waveform. The returned values are in the form of a waveform where each output waveform sample value is the $\log_{10}$ of the corresponding sample value of the input waveform.

The inverse functions are `waveform_exp(), waveform_exp10()`.

Note the following:

- The input waveform may be specified using the `RRECT_WAVE`, `RLONG_WAVE`, or `CRECT_WAVE` notations. See Waveform Sample Value Notations.
- It is an error to specify an input waveform specified using the `POLAR_WAVE` notation.
- When the input waveform is specified in `RRECT_WAVE`, or `RLONG_WAVE` the output will be `RRECT_WAVE`.
- When the input waveform is complex, i.e. specified in `CRECT_WAVE` notation, the output will be `CRECT_WAVE` and the following equation applies:

$$\log(x + yi) = \frac{\log(x^2 + y^2)}{2} + i \operatorname{atan}\frac{y}{x}$$

Also note the following about the output waveform:

- Y_units is set to `SCALE_VOLTS`
- X_start is set to 0
- X_increment is set to match that of the input waveform
- X_units is set to `SCALE_SECONDS`

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_log( Waveform* out_wave, Waveform* in_wave );
void waveform_log10( Waveform* out_wave, Waveform* in_wave );
```

where:

out_wave identifies the output waveform.

in_wave identifies the input waveform.

Example

???

3.27.23.2 waveform_exp(), waveform_exp10()

See Waveform Overview, Waveform Logrithmic Functions

Description

These are the inverse functions to `waveform_log(), waveform_log10()`.

Note the following:

- The input waveform may be specified using the RRECT_WAVE, RLONG_WAVE, or CRECT_WAVE notations. See Waveform Sample Value Notations.
- It is an error to specify an input waveform with sample values defined using the POLAR_WAVE notation.
- When the input waveform is specified in RRECT_WAVE, or RLONG_WAVE the output will be RRECT_WAVE.
- When the input waveform is specified in CRECT_WAVE the output will be CRECT_WAVE and the following equation applies:

$$e^{x+yi} = e^{x}[\cos(y) + i\sin(y)]$$

Also note the following about the output waveform:

- Y_units is set to SCALE_VOLTS
- X_start is set to 0
- X_increment is set to match that of the input waveform
- X_units is set to SCALE_SECONDS

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_exp( Waveform* out_wave, Waveform* in_wave );
void waveform_exp10( Waveform* out_wave, Waveform* in_wave );
```

where:

out_wave identifies the output waveform.

in_wave identifies the input waveform.

Example

???

3.27.23.3 waveform_power()

See Waveform Overview, Waveform Logrithmic Functions

Description

The waveform_power() function may be used to raise each sample value of a specified waveform to a specified power. The returned values are in the form of a waveform where each output waveform sample value is the the corresponding input waveform sample value raised to the specified power.

Note the following:

- If the power argument is specified as an integer the power calculations are made using interated multiplication.
- If the power argument is specified as an double the power calculations are made according to the following calculation: $e^{power \times \ln(x)}$. This allows fractional powers to be computed, but only when the value to be raised is positive and nonzero. The integer version can compute the powers of any value.
- The input waveform sample values may be specified using the RRECT_WAVE, RLONG_WAVE, or CRECT_WAVE notations. See Waveform Sample Value Notations.
- When the waveform samples are RLONG_WAVE, the integer version of waveform_power() returns RLONG_WAVE values, provided the power is positive, otherwise the values will be RRECT_WAVE.
- The double version of waveform_power() always converts RLONG_WAVEs to RRECT_WAVE.

Also note the following about the output waveform:

- Y_units is set to SCALE_VOLTS
- X_start is set to 0
- X_increment is set to match that of the input waveform
- X_units is set to SCALE_SECONDS

See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
void waveform_power(Waveform* out_wave,
                    Waveform* in_wave,
                    int power );
void waveform_power(Waveform* out_wave,
                    Waveform* in_wave,
                    double power );
```

where:

out_wave identifies the output waveform.

in_wave identifies the input waveform.

Example

???

3.27.24 Waveform Window Functions

See Waveform Overview.

When windowing is applied to a time domain waveform before an FFT is performed, the signals and harmonics get *spread* into more than one FFT bin. The amount of spread is determined by the windowing coefficients applied. The waveform functions which use the FFT fundamental and harmonic bins (waveform_sinad(), waveform_snr(), waveform_thd(), etc.) comprehend windowing and identify FFT bins correctly.

To apply window coefficients the following sequence is used:

- Window coefficients may only be applied to a time domain signaldefined in either `RRECT_WAVE` or `CRECT_WAVE` notation. See Waveform Sample Value Notations.
- A coefficient waveform is created using one of the Waveform Windowing Coefficient Functions. The supported windowing coefficients are represented by a unique function for each coefficient type i.e. `waveform_triangle_window_coefficients()`, `waveform_blackman_harris_window_coefficients()`, etc.
- `waveform_apply_window()` uses the coefficient waveform to apply the windowing.

For the curious...

The unit Bel is named after Alexander Graham Bell, for his work in the audio field. The Bel is a logarithmic scale representing the power ratio between two elements. For example, given two power measurements A and B, the ratio can be expressed in Bels as:

$$\log 10(A/B)$$

The unit Bel is too coarse for most applications, so decibels is used. There are 10 decibels in a Bel, as indicated by the metric prefix (this is the reason dB is abbreviated with a lower case d and a capital B). Thus the ratio above, in decibels, is:

$$10 \times \log 10(A/B)$$

When dealing with volts, the following expression is often seen:

$$20 \times \log 10(x/y)$$

This is because power is relative to volts squared. The correct power ratio is then calculated as:

$$10 \times \log 10(x^2/y^2)$$

As a proof, using the following fundamental identities of logarithms...

$$\log(a/b) = \log(a) - \log(b)$$

$$\log(a^b) = b \times \log(a)$$

...the above expression can be transformed into equivalent math expressions as follows:

$$10 \times [\log 10(x^2) - \log 10(x^2)]$$
$$10 \times [2 \times \log 10(x) - 2 \times \log 10(y)]$$
$$10 \times (2 \times [\log 10(x) - \log 10(y)])$$
$$10 \times 2 \times \log 10(x/y)$$
$$20 \times \log 10(x/y)$$

This shows the the earlier expression is correct, and can be used as a shortcut.

When determining SNR (or THD, SINAD, etc.), if windowing is NOT used, the *signal* is the value from a single FFT bin. For example, let **s** equal the value in the fundamental bin and **n** represent the sum of the squares in the noise bins. **s** is in terms of voltage and **n** is in terms of power, so they can't be directly compared. Two options are used (the 2nd being easier):

$$20 \times \log 10\left(\frac{s}{\sqrt{n}}\right)$$

or

$$10 \times \log 10\left(\frac{s^2}{n}\right)$$

When windowing has been applied to the waveform prior to performing the FFT, the *signal* component is contained in more than one FFT bin. In this situation, the total power of the signal is found by summing the squares of the bins containing the *signal*. This explains why the THD formula, for example, contains *sum_of_squares(signal)*. Thus, any time the term *signal* is used in the context of THD, SNR, SINAD, etc., it may consist of the sum of squares of several FFT bins, and not just a single value.

3.27.24.1 waveform_apply_window()

See Waveform Overview, Waveform Window Functions.

Description

The `waveform_apply_window()` function is used to apply windowing to an existing waveform. See Waveform Window Functions.

Usage

```
void waveform_apply_window( Waveform* out_wave,
                            Waveform* in_wave,
                            Waveform* window_coefficients );
```

where:

`out_wave` specifies the output waveform.

in_wave specifies the waveform to which windowing will be applied.

window_coefficients specifies a waveform containing the window coefficients to be applied to **in_wave**. This waveform can be created using the Waveform Windowing Coefficient Functions functions. See Waveform Window Functions.

Example

???

3.27.24.2 Waveform Windowing Coefficient Functions

See Waveform Overview, Waveform Window Functions.

Description

The functions documented here are used to create window coefficient waveforms. See Waveform Window Functions for details of how a coefficient waveform is used to apply windowing to an existing waveform.

The supported windowing coefficients are:

- Blackman
- Blackman-Harris (3rd order)
- Dolph-Chebyshev. See `waveform_dolph_chebyshev_window_coefficients()` for additional information about this option.
- Hamming
- Hanning
- Triangle

To properly create a windowing coefficient waveform the following are needed:

- Window type, specified by executing one of the functions below
- The Size of the waveform(s) to which the window coefficients will be applied
- A *tuning* parameter (for Dolph-Chebyshev and Kaiser only)

The coefficient waveforms can be created any time, in anticipation of using `waveform_apply_window()` later.

Usage

```
void waveform_blackman_window_coefficients(
            Waveform* out_wave,
            int num_samples );
void waveform_blackman_harris_window_coefficients(
            Waveform* out_wave,
            int num_samples );
void waveform_dolph_chebyshev_window_coefficients(
            Waveform* out_wave,
            int num_samples,
            double alpha );
void waveform_hamming_window_coefficients(
            Waveform* out_wave,
            int num_samples );
void waveform_hanning_window_coefficients(
            Waveform* out_wave,
            int num_samples );
void waveform_triangle_window_coefficients(
            Waveform* out_wave,
            int num_samples );
```

where:

out_wave specifies the output waveform which will store the coefficient samples.

num_samples specifies the desired number of sample values in the output waveform. The number of samples must match the number of samples of the waveform targeted to have windowing applied. See Waveform Window Functions.

alpha, see waveform_dolph_chebyshev_window_coefficients()

Example

???

3.27.24.3 waveform_dolph_chebyshev_window_coefficients()

See Waveform Windowing Coefficient Functions.

The `waveform_dolph_chebyshev_window_coefficients()` function is used to generate coefficients for a Dolph-Chebyshev window.

The Dolph-Chebyshev window differs from other Nextest windows as it is tunable, using the `alpha` parameter, which adjusts the relative levels of frequency sidelobes.

Care must be taken when specifying a value for the `alpha` parameter. If `alpha` is specified too low, the resulting window's waveform will be spread too wide, and hit the left and right side of the array before tapering off to zero. Even if the window appears correct when viewed using MSWT, the user should check for a positive spike at location zero, which indicates the `alpha` value is set too low.

If a complex FFT is performed on the generated coefficients, a main lobe appears with its highest point in index location zero. This quickly tapers down to a very uniform sidelobe level. It is the ratio of the main lobe level to the sidelobe level that defines the `alpha` parameter (α):

$$10^{\alpha} = \frac{MainLobeLevel}{SideLobeLevel}$$

Or in terms of dB, the side lobe level will be at:

$$-20\alpha \text{ dB}$$

When processing real time signals, a real FFT is used, and will result in the side lobes appearing roughly 3dB higher. Therefore, when passing an ideal signal through a Dolph-Chebyshev window of N points, one should not reasonably expect a measured signal-to-noise ratio of better than approximately:

$$(20\alpha - 10\log_{10} \quad N - 6) \text{ dB}$$

3.27.25 Waveform Convolution/Corrleation Functions

See Waveform Overview

Note: the theory behind convolution and correlation is beyond the scope of this manual. For reference see *Theory and Application of Digital Signal Processing*, by Rabiner and Gold, Prentice-Hall, 1975.

- `waveform_convolve_linear()`
- `waveform_convolve_partial()`
- `waveform_convolve_circular()`
- `waveform_correlate_linear()`
- `waveform_correlate_circular()`
- `waveform_autocorrelate_circular()`
- `waveform_covariance()`

3.27.25.1 waveform_convolve_linear()

See Waveform Overview, Waveform Convolution/Corrleation Functions.

Description

The `waveform_convolve_linear()` function is used to ???.

Note the following:

- The input waveforms must be defined using `RRECT_WAVE` or `RLONG_WAVE` notation
- The output waveform will be defined using `RRECT_WAVE` notation.
- The output waveform is determined as follows:

```
int s1 = waveform_get_size( in_wave1); //waveform_get_size()
int s2 = waveform_get_size( in_wave2);
```

$$\text{out\_wave}[i] = \sum_{j=0}^{s2-1} (\text{in\_wave1}[i-j] \cdot \text{in\_wave2}[j]) \qquad 0 \le i < s1 + s2 - 1$$

Note: the theory behind convolution and correlation is beyond the scope of this manual. For reference see Theory and Application of Digital Signal Processing, by Rabiner and Gold, Prentice-Hall, 1975.

Usage

```
void waveform_convolve_linear(Waveform* out_wave,
                              Waveform* in_wave1,
                              Waveform* in_wave2 );
```

where:

out_wave identifies the output waveform.

in_wave1 and **in_wave2** identify the two input waveforms. See Description.

Example

???

3.27.25.2 waveform_convolve_partial()

See Waveform Overview, Waveform Convolution/Corrleation Functions.

Description

The waveform_convolve_partial() function is used to ???

Note the following:

- The input waveforms must be defined using RRECT_WAVE or RLONG_WAVE notation
- The output waveform will be defined using RRECT_WAVE notation.
- The Size of in_wave1 must be greater than or equal to the Size of in_wave2.
- The output waveform is determined as follows:

```
int s1 = waveform_get_size( in_wave1); //waveform_get_size()
int s2 = waveform_get_size( in_wave2);
```

$$\text{out\_wave}[i] = \sum_{j=0}^{s2-1} (\text{in\_wave1}[i + s2 - 1 - j] \cdot \text{in\_wave2}[j]) \qquad 0 \le i < s1 - s2 + 1$$

Note: the theory behind convolution and correlation is beyond the scope of this manual. For reference see Theory and Application of Digital Signal Processing, by Rabiner and Gold, Prentice-Hall, 1975.

Usage

```
void waveform_convolve_partial(Waveform* out_wave,
                               Waveform* in_wave1,
                               Waveform* in_wave2 );
```

where:

out_wave identifies the output waveform.

in_wave1 and **in_wave2** identify the two input waveforms. See Description.

Example

???

3.27.25.3 waveform_convolve_circular()

See Waveform Overview, Waveform Convolution/Corrleation Functions.

Description

The `waveform_convolve_circular()` function is used to implement Finite Impulse filters (FIR) on periodic waveforms.

Note the following:

- The input waveforms must be defined using `RRECT_WAVE` or `RLONG_WAVE` notation.
- The output waveform will be defined using `RRECT_WAVE` notation.
- The output waveform takes units and scaling from the first input waveform.
- The number of sample values in the output waveform (Size) will be equal to the larget of of `in_wave1` or `in_wave2`. See `waveform_get_size()`.
- The output waveform sample values are determined as follows:

```
int size = waveform_get_size(in_wave1);//waveform_get_size()
```

$$\text{out\_wave}[i] = \sum_{j=0}^{size-1} (\text{in\_wave1}[(i - j + size) \bmod \text{size}] \cdot \text{in\_wave2}[j]) \qquad 0 \le i < size$$

- Convolution in the time domain is equivalent to multiplication in the frequency domain. Thus, `waveform_convolve_circular()` can be used on a time domain waveform instead of the series of FFT/multiplication/IFFT operations on the waveform in the frequency domain. However, the latter option may be faster when the waveforms have > 128 sample values.

Note: the theory behind convolution and correlation is beyond the scope of this manual. For reference see Theory and Application of Digital Signal Processing, by Rabiner and Gold, Prentice-Hall, 1975.

Usage

```
void waveform_convolve_circular( Waveform* out_wave,
                                 Waveform* in_wave1,
                                 Waveform* in_wave2 );
```

where:

`out_wave` identifies the output waveform.

`in_wave1` and **`in_wave2`** identify the two input waveforms. See Description.

Example

???

3.27.25.4 waveform_correlate_linear()

See Waveform Overview, Waveform Convolution/Corrleation Functions.

Usage

The `waveform_correlate_linear()` function is used to ???

Note the following:

- The input waveforms must be defined using RRECT_WAVE or RLONG_WAVE notation.
- The output waveform will be defined using RRECT_WAVE notation.
- The number of sample values (Size) of in_wave1 must be greater than or equal to in_wave2. See waveform_get_size().
- The output waveform sample values are determined as follows:

```
int s1 = waveform_get_size( in_wave1); //waveform_get_size()
int s2 = waveform_get_size( in_wave2);
```

$$\text{out\_wave}[i] = \sum_{j=0}^{s2-1} (\text{in\_wave1}[i+j] \cdot \text{in\_wave2}[j]) \qquad 0 \le i < s1 - s2 + 1$$

Description

```
void waveform_correlate_linear(  Waveform* out_wave,
                                 Waveform* in_wave1,
                                 Waveform* in_wave2 );
```

where:

out_wave identifies the output waveform.

in_wave1 and **in_wave2** identify the two input waveforms. See Description.

Example

???

3.27.25.5 waveform_correlate_circular()

See Waveform Overview, Waveform Convolution/Corrleation Functions.

Usage

The waveform_correlate_circular() function is used to ???

Correlating two different waveforms is sometimes referred to as cross correlation. Correlating a waveform to itself is called autocorrelation (see waveform_autocorrelate_circular()).

Note the following:

- The input waveforms must be defined using RRECT_WAVE or RLONG_WAVE notation.
- The output waveform will be defined using RRECT_WAVE notation.
- The Size of in_wave1 and in_wave2 must be the same.
- The output waveform sample values are determined as follows:

$$\text{out\_wave}[i] = \sum_{j=0}^{size-1} (\text{in\_wave1}[(i+j) \bmod \text{size}] \cdot \text{in\_wave2}[j]) \qquad 0 \le i < size$$

-

Description

```
void waveform_correlate_circular(Waveform* out_wave,
                                 Waveform* in_wave1,
                                 Waveform* in_wave2 );
```

where:

out_wave identifies the output waveform.

in_wave1 and **in_wave2** identify the two input waveforms. See Description.

Example

???

3.27.25.6 waveform_autocorrelate_circular()

See Waveform Overview, Waveform Convolution/Corrleation Functions.

Description

The waveform_autocorrelate_circular() function is used to ???

Correlating two different waveforms, sometimes referred to as cross correlation, is done using waveform_correlate_circular(). Correlating a waveform to itself is known as autocorrelation. If the same waveform is speciifed for both inputs to

waveform_correlate_circular() the results are the same as obtained using waveform_autocorrelate_circular().

Note the following:

- The input waveforms must be defined using RRECT_WAVE or RLONG_WAVE notation
- The output waveform will be defined using RRECT_WAVE notation.
- The output waveform sample values are determined as follows:

$$\text{out\_wave}[i] = \sum_{j=0}^{size-1} (\text{in\_wave}[(i+j) \bmod \text{size}] \cdot \text{in\_wave}[j]) \qquad 0 \le i < size$$

- There is no function called waveform_autocorrelate_linear() since this would reduce to just one value which is equal to the sum of the squares of the input values.

Note: the theory behind convolution and correlation is beyond the scope of this manual. For reference see Theory and Application of Digital Signal Processing, by Rabiner and Gold, Prentice-Hall, 1975.

Usage

```
void waveform_autocorrelate_circular( Waveform* out_wave,
                                      Waveform* in_wave );
```

where:

out_wave identifies the output waveform.

in_wave identify the input waveform. See Description.

Example

???

3.27.25.7 waveform_covariance()

See Waveform Overview, Waveform Convolution/Corrleation Functions.

Description

The `waveform_covariance()` function is used to ???

Covariance is the cross correlation of the input waveforms after their arithmetic means have been subtracted.

Note the following:

- The input waveforms must be defined using `RRECT_WAVE` or `RLONG_WAVE` notation.
- The output waveform will be defined using `RRECT_WAVE` notation.

The usage of convolution and correlation functions is usually restricted to small waveform Sizes. Convolution in the time domain is equal to element-by-element multiplication in the frequency domain. This means that when the number of elements in the input arrays grows sufficiently large, it's faster to employ the following sequence to compute the equivalent of `waveform_convolve_circular()`:

```
waveform_real_fft( tmpWF, inWF1 );
waveform_real_fft( outWF, inWF2 );
waveform_multiply( outWF, outWF, tmpWF );
waveform_real_ifft( outWF, outWF );
double scale = (double) waveform_get_size( inWF1 ) / 2.0;
double offset = waveform_arithmetic_mean( inWF1 );
waveform_multiply( outWF, outWF, scale );
waveform_subtract( outWF, outWF, offset );
```

The reason this can be faster is that the convolution and correlation functions have execution times on the order of $O(n^2)$, whereas FFT function execution times are $O(n \log n)$. The Size required before the alternative method above is faster is dependent on the specifics of the CPU being employed, but in one experiment 128 samples (or larger) was computed faster with the FFT method as opposed to the convolution method. And, for non-radix-2 sized arrays, the arrays needed to be larger for the FFT method to be faster since non-radix-2 FFTs take longer than radix-2 FFTs. Waveforms of about 500 samples were therefore required before the FFT method was faster.

FFTs can be employed to implement `waveform_convolve_linear()` as well. The trick is to first zero pad the input arrays such that the FFTs do not include any circular products. It is necessary to at least double their size and it would be beneficial to round the resulting size up to the next power of two.

Usage

```
void waveform_covariance(Waveform* out_wave,
                         Waveform* in_wave1,
                         Waveform* in_wave2 );
```

where:

out_wave identifies the output waveform.

in_wave1 and **in_wave2** identify the two input waveforms. See Description.

Example

???

3.27.26 Waveform Wierd Functions

See Waveform Overview

- vecmem_modify()
- waveform_enob()
- waveform_index_to_time()
- waveform_settling_time()

3.27.26.1 vecmem_modify()

See Waveform Overview.

Description

The vecmem_modify() function may be used to modify pattern data in a Logic Test Pattern, replacing the existing (compiled-in) pattern data with sample values from a specified waveform.

Note the following:

- Pattern data for one or more pin(s) can be modified. A pin list is specified to identify which pins are modified. Every pin in the pin list is modified identically.

- The first vector to be modified in the test pattern is identified using the `vector_offset` argument. Legal values are 1..n, where the last vector can be identified using the `addrs()` function, one version of which returns the number of vectors in a Logic Test Pattern.
- The `vector_stride` argument can be used to specify whether any vectors are to be skipped. Legal values are 1..n where 1 = modify consecutive vectors.
- The number of vectors to be modified is determined by the number of samples in the specified waveform. `waveform_get_size()` can be used to determine the number of samples in a given waveform.

Note: the system software does not check whether the number of sample values in the specified waveform will exceed the scope of the specified test pattern. Careless usage can cause corruption of test patterns other than that specified.

- The sample values of the specified waveform must be integers (`RLONG_WAVE`). The following table shows how VectorStates are mapped to integer values:

Sample Data Value	VectorState	Pattern Token
???	drive_lo	0
???	drive_hi	1
???	tristate	X
???	strobe_lo	L
???	strobe_hi	H

Usage

```
void vecmem_modify( Pattern *pattern,
                    int vector_offset,
                    PinList* pinlist,
                    Waveform* data,
                    int vector_stride DEFAULT_VALUE( 1 ) );
```

where:

`pattern` specifies the pattern to be modified.

`vector_offset` identifies the first vector to be modified. See Description.

`pinlist` identifies the pin(s) for which pattern data will be modified. See Description.

`data` identifies the waveform from which the sample values are taken. See Description.

`vector_stride` is optional, and if used specifies the number of vectors to be skipped between each vector which is modified. Default = 1 i.e. modify consecutive vectors.

Example

???

3.27.26.2 waveform_enob()

See Waveform Overview, Waveform Wierd Functions

Description

ENOB = effective number of bits

The `waveform_enob()` function is used to calculate the ENOB required in an ideal ADC to have a specified signal-to-noise ratio (SNR).

The signal-to-noise ratio attainable by a perfect standard ADC is limited by the ADCs quantization error, which is determined by the number of bits the ADC contains. The expected quantization error (in decibels) from *B* bits is given by the equation:

$$SNR = 10\left[B \times \log_{10}(4) + \log_{10}\left(\frac{3}{2}\right)\right]$$

which is more readily recognized from its popular approximation:

$$SNR = 6.02 \times B + 1.76$$

The `waveform_enob()` function allows us to relate a given SNR value to an ideal ADC. It calculates the effective number of bits (ENOB) required in an ideal ADC to have the specified signal-to-noise ratio. It does this by using the inverse of the above relation, as:

$$SNR = \frac{\frac{decibels}{10} - \log_{10}\left(\frac{3}{2}\right)}{\log_{10}(4)}$$

Usage

```
double waveform_enob( double decibels );
```

where:

decibels specifies the signal-to-noise ratio in dB.

waveform_enob() returns the ENOB of the specified SNR (in dB).

Example

???

3.27.26.3 waveform_index_to_time()

See Waveform Overview, Waveform Wierd Functions

Description

The waveform_index_to_time() function is used to return the X axis time value of a specified waveform sample value i.e. given an index into a waveform's sample array, return the corresponding X axis time value of that index location. Note the following:

- The waveform may be defined using any valid notation (see Waveform Sample Value Notations).
- The time value returned takes into account the waveform's X_increment value and X_start value (see Waveform* Attributes, Waveform Mathematical View). This is effectively
  (X_start + X_increment * index), scaled appropriately for the time units involved.
- If the waveform's X axis does not represent a recognized time scale, the value returned will be calculated as if the X axis units were set to seconds.

Usage

```
MKSTime waveform_index_to_time( Waveform* in_wave, double index );
```

where:

in_wave specifies the input waveform.

index specified the sample value of interest.

`waveform_index_to_time()` returns the X axis time value of the **`index`**th sample value.

Example

???

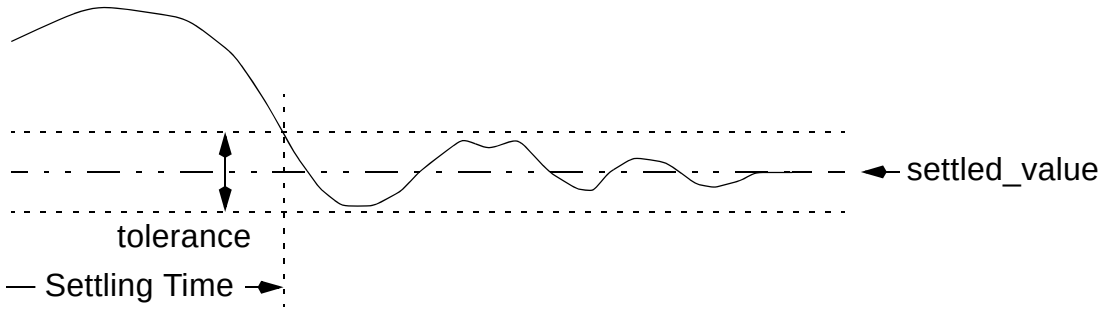
3.27.26.4 waveform_settling_time()

See Waveform Overview, Waveform Wierd Functions

Description

The `waveform_settling_time()` function determines the settling time for a time domain waveform. Note the following:

- The input waveform must be defined using `RRECT_WAVE` or `RLONG_WAVE` notation
- The input waveform must have more than one sample value.
- The `tolerance` parameter must be positive and non-zero and represents how close to the `settled_value` the waveform must come before being considered settled. Both `tolerance` and `settled_value` must be specified in terms of whatever scaling is being used for the Y axis.
- If the `settled_value` parameter is specified (not NULL), the computed settling level is returned in that argument. It is computed on the fly starting at the end of the waveform. The diagram below shows the relationships with an example waveform:



- To give better accuracy, `waveform_settling_time()` uses linear interpolation on the two sample values bounding the crossing of the tolerance line. The interpolated index is then converted to a time value using `waveform_index_to_time()`.

Usage

```
MKSTime waveform_settling_time(Waveform* in_wave,
                    double tolerance,
                    double *settled_value DEFAULT_VALUE( NULL ) );
```

where:

in_wave specifies the input waveform.

tolerance how close to the settled value the waveform must come before being considered settled.

settled_value is optional, and is a pointer to an existing double variable, used to return the calculated settled value. See Description.

`waveform_settling_time()` returns the settling time value.

Example

???

3.27.27 Waveform Compression Functions

See Waveform Overview.

To encode a signal with Mu-law encoding or A-law encoding is to compress it logarithmically. The resulting signal, when quantized and later expanded, yields a better signal-to-noise ratio than a normally quantized signal would. This is a result of the relatively low levels of the signal being amplified in relation to the larger levels. The relative quantization error is thus reduced.

The following functions encode or decode a specified waveform:

- `waveform_mu_law_encode()`
- `waveform_mu_law_decode()`
- `waveform_a_law_encode()`
- `waveform_a_law_decode()`

3.27.27.1 waveform_mu_law_encode()

See Waveform Overview, Waveform Compression Functions

Description

The `waveform_mu_law_encode()` function is used to encode an input waveform using the Mu-law compression method (see intro in Waveform Compression Functions), putting the result into the output waveform. The `waveform_mu_law_decode()` function performs the inverse of this function.

The `waveform_mu_law_encode()` function encodes the input waveform with Mu-law (μ -law) compression. The value of mu (μ) varies the amount of the compression as shown in the equation below. If not supplied, the most common value of 255 is used.

```
V = maximum magnitude of the input signal
x = value to be compressed
y = compressed value
```

$$y = signum(x)\frac{V \cdot \ln\left(1 + \mu\frac{|x|}{V}\right)}{\ln(1+\mu)}$$

Also note the following:

- *V* is assumed to be a value of one. The input waveform should therefore be scaled to be in the range of -1 to +1 before being compressed. Values greater than one will be silently clamped to one and values less than minus one will be silently clamped to minus one.
- The output waveform will also be within the range of -1 to 1.
- The input waveform must be defined using `RRECT_WAVE` notation.
- The output waveform will be in `RRECT_WAVE` notation, and inherit all of the other attributes of the input waveform.

Usage

```
void waveform_mu_law_encode(Waveform* out_wave,
                            Waveform* in_wave,
                            double mu DEFAULT_VALUE( 255.0 ) );
```

where:

out_wave specifies the encoded output waveform.

in_wave specifies the input waveform.

mu is optional, and if used specifies the amount of compression to be performed. Default = 255.

Example

???

3.27.27.2 waveform_mu_law_decode()

See Waveform Overview, Waveform Compression Functions

Description

This is the inverse function of `waveform_mu_law_encode()` i.e. the `waveform_mu_law_decode()` function decompresses a waveform previouslycompressed using `waveform_mu_law_encode()` . The value of mu (μ) varies the amount of the decompression as shown in the equation below. If not supplied, the most common value of 255 is used.

```
V = maximum magnitude of the input signal
x = uncompressed value
y = value being decompressed
```

$$x = signum(y)\frac{V}{\mu}\left[e^{\ln(1+\mu)\frac{|y|}{V}} - 1\right]$$

Also note the following:

- *V* is assumed to be a value of one. The input waveform should therefore have been compressed into the range of -1 to +1. Values greater than one will be silently clamped to one and values less than minus one will be silently clamped to minus one.
- The output waveform will also be within the range of -1 to 1.
- The input waveform must be defined using `RRECT_WAVE` notation.
- The output waveform will be in `RRECT_WAVE` notation, and inherit all of the other attributes of the input waveform.

Usage

```
void waveform_mu_law_decode(Waveform* out_wave,
                            Waveform* in_wave,
                            double mu DEFAULT_VALUE( 255.0 ) );
```

where:

out_wave specifies the decompressed output waveform.

in_wave specifies the input waveform.

mu is optional, and if used specifies the amount of decompression to be performed. Default = 255.

Example

???

3.27.27.3 waveform_a_law_encode()

See Waveform Overview, Waveform Compression Functions

Description

The `waveform_a_law_encode()` function is used to encode an input waveform using the A-law compression method (see intro in Waveform Compression Functions), putting the result into the output waveform. The `waveform_a_law_decode()` function performs the inverse of this function.

The `waveform_a_law_encode()` function encodes the input waveform with A-law compression. A-law compression is achieved differently depending on the range of the input value. The value of A varies the amount of the compression as shown in the equation below. If not supplied, the most common value of 87.6is used:

```
V = maximum magnitude of the input signal
x = value to be compressed
y = compressed value
```

Also note the following:

$$y = \begin{cases} \dfrac{Ax}{1 + \ln A} \text{ for } -\dfrac{V}{A} \le x \le \dfrac{V}{A} \\ signum(x)\dfrac{V\left[1 + \ln\left(A\dfrac{|x|}{V}\right)\right]}{1 + \ln A} \text{ for } \dfrac{V}{A} < |x| \le V \end{cases}$$

- *V* is assumed to be a value of one. The input waveform should therefore be scaled to be in the range of -1 to +1 before being compressed. Values greater than one will be silently clamped to one and values less than minus one will be silently clamped to minus one.
- The output waveform will also be within the range of -1 to 1.
- The input waveform must be defined using RRECT_WAVE notation.
- The output waveform will be in RRECT_WAVE notation, and inherit all of the other attributes of the input waveform.

Usage

```
void waveform_a_law_encode( Waveform* out_wave,
                            Waveform* in_wave,
                            double a DEFAULT_VALUE( 87.6 ) );
```

where:

out_wave specifies the encoded output waveform.

in_wave specifies the input waveform.

a is optional, and if used specifies the amount of compression to be performed. Default = 87.6.

Example

???

3.27.27.4 waveform_a_law_decode()

See Waveform Overview, Waveform Compression Functions

Description

This is the inverse function of `waveform_a_law_encode()` i.e. the `waveform_a_law_decode()` function decompresses a waveform previously compressed using `waveform_a_law_encode()`. The value of A varies the amount of the decompression as shown in the equation below. If not supplied, the most common value of 87.6 is used.

```
V = maximum magnitude of the input signal
x = uncompressed value
y = value being decompressed
```

$$y = \begin{cases} \frac{y(1+\ln A)}{A} \text{ for } -\frac{V}{1+\ln A} \le y \le \frac{V}{1+\ln A} \\ signum(y)\frac{V}{A}e^{(1+\ln A)\frac{|y|}{V}-1} \text{ for } \frac{V}{1+\ln A} < |y| \le V \end{cases}$$

Also note the following:

- *V* is assumed to be a value of one. The input waveform should therefore have been compressed into the range of -1 to +1. Values greater than one will be silently clamped to one and values less than minus one will be silently clamped to minus one.
- The output waveform will also be within the range of -1 to 1.
- The input waveform must be defined using `RRECT_WAVE` notation.
- The output waveform will be in `RRECT_WAVE` notation, and inherit all of the other attributes of the input waveform.

Usage

```
void waveform_a_law_decode( Waveform* out_wave,
                            Waveform* in_wave,
                            double a DEFAULT_VALUE( 87.6 ) );
```

where:

`out_wave` specifies the decompressed output waveform.

`in_wave` specifies the input waveform.

`mu` is optional, and if used specifies the amount of decompression to be performed. Default = 87.6.

Example

???

3.27.28 Waveform FFT Functions

See Waveform Overview

- `waveform_complex_fft(), waveform_complex_ifft()`
- `waveform_real_fft(), waveform_real_ifft()`
- `waveform_real_ifft_even(), waveform_real_ifft_odd()`
- `waveform_set_odd_flag(), waveform_get_odd_flag()`
- FFT Aliasing

3.27.28.1 waveform_complex_fft(), waveform_complex_ifft()

See Waveform Overview, Waveform FFT Functions.

Description

The `waveform_complex_fft()` function processes a specified input waveform and creates an output waveform representing the discrete Fourier transform of the input. Note the following:

- The input waveform may be defined using any notation (see Waveform Sample Value Notations). However, `waveform_complex_fft()` only processes `CRECT_WAVE`, thus if the input waveform is defined using any of the other notations a conversion will be automatically performed.
- The FFT output waveform will be `CRECT_WAVE`, and will be the same Size as the input waveform.
- The input waveform must contain at least eight sample values (Size >= 8). If the input waveform Size is a power of two, a *radix-2* FFT is performed, otherwise a *Chirp-Z* FFT is performed.
- The output waveform X_units is set = ???, and the X scale set = ???.

`waveform_complex_ifft()` is the inverse of `waveform_complex_fft()`:

- The input waveform Type must be CRECT_WAVE.
- The input waveform must contain at least eight sample values (Size >= 8). If the input waveform Size is a power of two, a *radix-2* FFT is performed, otherwise a *Chirp-Z* FFT is performed.
- The FFT output waveform will be CRECT_WAVE, and will be sized as noted above.
- The output waveform X_units is set = ???, and the X scale set = ???.

Also see waveform_real_fft(), waveform_real_ifft().

Usage

```
void waveform_complex_fft( Waveform* out_wave, Waveform* in_wave);
void waveform_complex_ifft( Waveform* out_wave,
                            Waveform* in_wave);
```

where:

out_wave specifies the output waveform to contain the FFT or inverse FFT.

in_wave specifies the input waveform.

Example

???

3.27.28.2 waveform_real_fft(), waveform_real_ifft()

See Waveform Overview, Waveform FFT Functions.

Description

waveform_real_fft() is similar to waveform_complex_fft() except:

- When a *real* waveform is processed (i.e. sample values defined using RRECT_WAVE or RLONG_WAVE), the resulting complex results may be found with fewer calculations i.e the FFT is completed more quickly.
- The FFT output waveform will be CRECT_WAVE, and will be the Sized as follows. If the input waveform is even length (even number of sample values) the output will have one additional bin, called the Nyquist bin. Although a [sine] waveform at the Nyquist frequency cannot be accurately captured, certain waveforms leave energy in this FFT bin, thus if this bin is not maintained the inverse FFT does not

represent an exact inverse operation.
- A real FFT of a waveform of Size 2n returns a CRECT_WAVE of size n+1.
- A real FFT of a waveform of Size 2n+1 returns a CRECT_WAVE of size n+1.
Or stated differently, a real FFT of an input waveform of Size k returns a CRECT_WAVE of size = floor(k/2)+1. Also see waveform_real_ifft_even(), waveform_real_ifft_odd().

- The output waveform will be half the Size of the input waveform. Since the input waveform has no imaginary components, the 2nd half of the complex FFT are not returned (these are equal to the complex conjugate of the first half). If the number of sample values in the input waveform is odd, the output waveform will be:

$$\frac{size-1}{2}$$

- The input waveform must contain at least eight sample values (Size >= 8). If the input waveform size is a power of two, a *radix-2* FFT is performed, otherwise a *Chirp-Z* FFT is performed.
- The output waveform X_units is set = ???, and the X scale set = ???.

waveform_real_ifft() is the inverse of waveform_real_fft():

- The input waveform must be defined using CRECT_WAVE notation.
- The input waveform must contain at least four sample value (Size >= 4)
- The output waveform will be an RRECT_WAVE, twice the size of the input waveform.
- The output waveform X_units is set = ???, and the X scale set = ???.

X_units Usage

```
void waveform_real_fft( Waveform* out_wave, Waveform* in_wave);
void waveform_real_ifft( Waveform* out_wave, Waveform* in_wave);
```

where:

out_wave specifies the output waveform to contain the FFT or inverse FFT.

in_wave specifies the input waveform.

Example

???

3.27.28.3 waveform_real_ifft_even(), waveform_real_ifft_odd()

See Waveform Overview, Waveform FFT Functions.

Description

When performing a real inverse FFT on a CRECT_WAVE input waveform of any given Size, `waveform_real_ifft()` will generate an RRECT_WAVE output waveform with either an even or odd number of samples.

Instead, `waveform_real_ifft_even()` or `waveform_real_ifft_odd()` can be used to explicitly make the choice.

The results of a real FFT operation are tagged with an indicator, the *odd flag*, of whether the original real waveform had an even or odd Size. The *odd flag* = TRUE to indicate an odd size. This flag is used as follows:

- `waveform_real_ifft()` examines the odd flag and calls either `waveform_real_ifft_even()` or `waveform_real_ifft_odd()` as appropriate.
- The flag is used to correctly sort out the aliasing of harmonics that surpassed the Nyquist frequency in several functions, including `waveform_snr()`, `waveform_thd()`, etc.

Also see `waveform_set_odd_flag(), waveform_get_odd_flag()`.

Usage

```
void waveform_real_ifft_even( Waveform* out_wave,
                              Waveform* in_wave );
void waveform_real_ifft_odd(Waveform* out_wave,
                            Waveform* in_wave );
```

where:

`out_wave` specifies the output waveform to contain the FFT or inverse FFT.

`in_wave` specifies the input waveform.

Example

???

3.27.28.4 waveform_set_odd_flag(), waveform_get_odd_flag()

See Waveform Overview, Waveform FFT Functions.

Description

When performing a real inverse FFT on an input waveform of any given Size, `waveform_real_ifft()` will generate an output waveform with either an even or odd number of samples, and the *odd flag* is set accordingly.

The odd flag may be examined using `waveform_get_odd_flag()` or modified using `waveform_set_odd_flag()`.

> Note: the user should not normally need to interact with the odd flag.

Usage

```
void waveform_set_odd_flag( Waveform* obj, BOOL odd_flag );
BOOL waveform_get_odd_flag( Waveform* obj );
```

where:

`obj` identifies the waveform of interest.

`odd_flag` specifies the desired state where TRUE = odd and FALSE = even.

`waveform_get_odd_flag()` returns TRUE if the odd flag is set, otherwise FALSE is returned.

Example

???

3.27.28.5 FFT Aliasing

Given a 128 bin FFT with the fundamental signal value in bin-47, the 2nd harmonic will be in bin-94 (2*47). However, since there are only 128 bins, the 3rd harmonic can't be in bin-141 (3*47). The 3rd harmonic will be in bin-115 (128 - (141-128)). This is called harmonic folding or *aliasing*.

The Waveform Analysis Functions functions which identify FFT harmonic bins (`waveform_sinad()`, `waveform_snr()`, `waveform_thd()`, etc.) comprehend windowing and identify those bins correctly.

3.27.29 Waveform Analysis Functions

See Waveform Overview

- `waveform_average()`
- `waveform_arithmetic_mean()`
- `waveform_clip_upper(), waveform_clip_lower()`
- `waveform_deinterleave()`
- `waveform_eq()`
- `waveform_geometric_mean()`
- `waveform_histogram()`
- `waveform_interleave()`
- `waveform_linear_regression()`
- `waveform_magnitudes()`
- `waveform_median()`
- `waveform_min_max()`
- `waveform_quantize()`
- `waveform_rms()`
- `waveform_sfdr()`
- `waveform_signals_and_noise()`
- `waveform_sinad()`
- `waveform_snr()`

- waveform_standard_deviation()
- waveform_sum_of_squares()
- waveform_thd()
- waveform_variance()

Also see INL & DNL Functions.

3.27.29.1 waveform_average()

See Waveform Overview, Waveform Analysis Functions.

Description

Consider a Waveform* which contains multiple repetitions of a signal; for example, 256 repetitions of a ramp waveform. The waveform_average() function can be used to return a single repetition of that waveform, with each sample value being the average of the corresponding sample values from the original repetitive waveform. Note the following:

- The input waveform may be defined using any notation (see Waveform Sample Value Notations), however the input waveform Type is typically RRECT_WAVE.
- The input waveform should represent a signal that has been repeatedly sampled, a known number of times. For example, if the input waveform consists of 8 copies of a signal, and each copy consists of 256 sample values, the input waveform will contain 2048 total sample values.
- The repetitions argument to waveform_average() specifies the number of of copies of the signal in the input waveform (8 in the previous bullet).
- Given these values, the output waveform will consist of 256 sample values. The first sample of the output waveform will be the average of input sample values 0, 256, 512, 768, 1024, 1280, 1536, and 1792. The second sample of the output waveform will be the average of input sample values 1, 257, 513, 769, 1025, 1281, 1537, and 1793. Etc.
- The output waveform will use the same notation as the input waveform (see Waveform Sample Value Notations) and have the same attributes.

Usage

```
void waveform_average( Waveform* out_wave,
                       Waveform* in_wave,
                       int repetitions );
```

where:

out_wave identifies the output waveform.

in_wave identifies the input waveform.

repetitions specifies the number of repetitions of the signal in **in_wave**. See Description.

Example

???

3.27.29.2 waveform_arithmetic_mean()

See Waveform Overview, Waveform Analysis Functions.

Description

The `waveform_arithmetic_mean()` is used to return arithmetic mean of all samples of the specified waveform. Only waveforms defined using one-part notations (`RRECT_WAVE`, `RLONG_WAVE`) are supported (see Waveform Sample Value Notations).

The `waveform_arithmetic_mean()` operation can be described mathamethically as:

`return` $\frac{\left(\sum_{i=0}^{n-1} Din_i\right)}{n}$ `where:`

`Din = the sample set of the waveform`
`i = index into the set (array index)`
`n = number of samples.`See Waveform Mathematical View and Waveform Units for descriptions of these parameters.

Usage

```
double waveform_arithmetic_mean( Waveform* in_wave );
```

where:

in_wave specifies the waveform of interest.

`waveform_arithmetic_mean()` returns the arithmetic mean of all samples of the specified waveform.

Example

???

3.27.29.3 waveform_clip_upper(), waveform_clip_lower()

See Waveform Overview, Waveform Analysis Functions.

Description

The `waveform_clip_upper()` and `waveform_clip_lower()` functions are used to modify waveform sample values. Note the following:

- Both functions process all sample values of a specified input waveform and put the results into an output waveform of the same Size and defined using the same notation (see Waveform Sample Value Notations).
- `waveform_clip_lower()` sets (clips) sample values which are less than a specified minimum value (the `min_value` argument) to that minimum value.
- `waveform_clip_upper()` sets (clips) sample values which are greater than a specified maximum value (the `max_value` argument) to that maximum value.
- The input waveform must be `RRECT_WAVE` or `RLONG_WAVE`.
- If the input waveform is `RLONG_WAVE`, the output waveform remains `RLONG_WAVE` only if `min_value` is an integer, otherwise the output waveform becomes an `RRECT_WAVE`.

Note that the sequence:

```
waveform_clip_lower( out, in, min_value );
wavefrom_clip_upper( out, out, max_value );
```

results in the same output as:

```
waveform_clamp( out, in, min_value, max_value );//waveform_clamp()
```

Usage

```
void waveform_clip_upper(Waveform* out_wave,
                         Waveform* in_wave,
                         double max_value );
void waveform_clip_lower(Waveform* out_wave,
                         Waveform* in_wave,
                         double min_value );
```

where:

`out_wave` identifies the output waveform.

`in_wave` identifies the input waveform.

`max_value` identifies the clip value (maximum value) used by `waveform_clip_upper()`. See Description.

`min_value` identifies the clip value (minimum value) used by `waveform_clip_lower()`. See Description.

Example

???

3.27.29.4 waveform_deinterleave()

See Waveform Overview, Waveform Analysis Functions.

Description

The `waveform_deinterleave()` function splits a specified input waveform into two output waveforms, as follows:

- Each sample value of the first output waveform consists of every-other sample value of the input waveform, beginning with the first sample.
- Each sample value of the second output waveform consists of every-other sample value of the input waveform, beginning with the second sample.
- If there is an odd number of input waveform samples, the first output waveform will contain one more sample value than the second output waveform.
- The input waveform may be defined using any notation (see Waveform Sample Value Notations). The output waveforms will use the same notation.

Also see waveform_interleave().

Note that the waveform_decimate() function can also be used to *deinterleave*, but is more versatile, since it can be used to deinterleave one waveform into to any number of waveforms.

Usage

```
void waveform_deinterleave( Waveform* out_wave1,
                            Waveform* out_wave2,
                            Waveform* in_wave );
```

where:

out_wave1 and **out_wave2** identify the two output waveforms. See Description.

in_wave identifies the input waveform.

Example

???

3.27.29.5 waveform_eq()

See Waveform Overview, Waveform Analysis Functions.

Description

Usage

```
BOOL waveform_eq( Waveform* in_wave,
                  double scalar,
                  double tolerance DEFAULT_VALUE( 1.0e-7 ) );
BOOL waveform_eq( Waveform* in_wave1,
                  Waveform* in_wave2,
                  double tolerance DEFAULT_VALUE( 1.0e-7 ) );
void waveform_eq( Waveform* out_wave,
                  Waveform* in_wave1,
                  Waveform* in_wave2,
                  double tolerance DEFAULT_VALUE( 1.0e-7 ) );
```

```
void waveform_eq( Waveform* out_wave,
                  Waveform* in_wave,
                  double scalar,
                  double tolerance );
```

where:

in_wave identifies an input waveform in which each sample value is compared against **scalar**.

tolerance specifies ???

in_wave1 and **in_wave2** specify two input waveforms from which corresponding samples are compared.

out_wave identifies an output waveform in which each sample value represents the result of comparing the corresponding sample value from the input waveform. When **scalar** is used, the **in_wave1** sample value is compared against **scalar**. When **in_wave2** is used the **in_wave1** sample value is compared against the corresponding **in_wave2** sample value.

The versions of `waveform_eq()` which return `BOOL` will return `TRUE` when ???

Example

???

3.27.29.6 waveform_geometric_mean()

See Waveform Overview, Waveform Analysis Functions.

Description

The `waveform_geometric_mean()` function is used to return the geometric mean of all sample values in a specified waveform. Note the following:

- Supports sample values defined in `RRECT_WAVE` and `RLONG_WAVE` only. See Waveform Sample Value Notations.
- If any sample value is negative or zero, zero is returned, otherwise the geometric mean is returned.

- If the waveform contains *n* sample values (Size = *n*) the geometric mean is defined to be the n^{th} root of the product of all of the sample values. To reduce the occurrence of numerical overflow this is calculated as:

  exp(arithmetic_mean(log (sample_values)))

Usage

```
double waveform_geometric_mean( Waveform* in_wave );
```

where:

in_wave identifies the input waveform.

See Description for the `waveform_geometric_mean()` return value.

Example

???

3.27.29.7 waveform_histogram()

See Waveform Overview, Waveform Analysis Functions.

Description

The `waveform_histogram()` function analyzes the sample values of a specified input waveform and returns an array of count values which can be used as histogram data.

The following input parameters are required:

- The input waveform, consisting of some number of sample values. Only waveforms defined using a one-part notation (`RRECT_WAVE` or `RLONG_WAVE`) are supported. See Waveform Sample Value Notations.
- Two limits: Low Bound and High Bound. These can be used to exclude values from being counted. In the example below Low = 0.175 and High = 0.625.
- The number of count values (number of buckets) to return. This also determines how the range of sample values between Low Bound and High Bound are divided while being counted. In the example below number of buckets = 11.

`waveform_histogram()` returns the count of exceptions, which are those sample values below the Low Bound and above the High Bound. The histogram data (counts in buckets) is returned as a `Waveform*`, in the `RLONG_WAVE` notation.

The diagram below shows how `waveform_histogram()` operates:

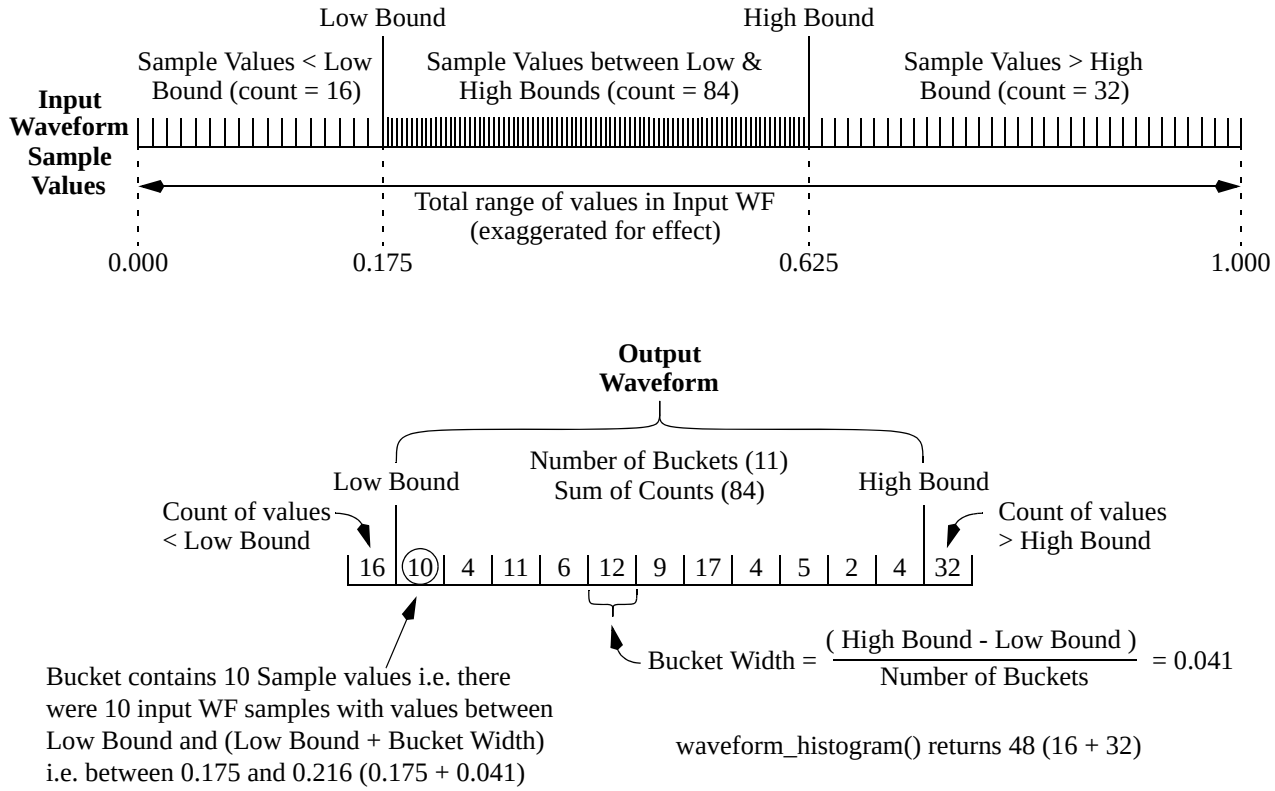


Figure-63: `waveform_histogram()` User Model

Note the following about the diagram above:

- The bucket width is calculated as shown. This determines where each input sample value is counted. Note that a given bucket count can represent the count of a range of input sample values. The example has a bucket width of 0.041.
- Each sample value of the input waveform is analyzed. If it is less than Low Bound or greater than High Bound it is counted as an exception. Values which are NaN (not a number) are also counted as exceptions. *NaN* values occur as a result of calculations (sqrt -1, divide by zero, etc.).
- If the sample is not an exception, the analysis determines in which bucket the sample should be counted. Given there are `[i]` buckets:

```
Bucket[i] = count of all input waveform sample values
   >= low_bound + (i * bucket_width)
and
   < high_bound + (( i + 1 ) * bucket_width)
```

- When all samples are processed, the counts from each bucket are inserted into the output `Waveform*`. The output waveform X-units is set to the input waveform's Y-units. The output waveform's Y-units is set to *counts* (`SCALE_COUNTS`, see Waveform Units).
- `waveform_histogram()` returns the count of exceptions.

Usage

```
int waveform_histogram(Waveform* out_wave,
                       Waveform* in_wave,
                       double low_bound,
                       double high_bound,
                       int num_buckets );
```

where:

`out_wave` specifies the destination for the histogram data. This must be a pointer to an existing `Waveform*` variable. See Description.

`in_wave` specifies the input waveform to be evaluated. **`in_wave`** must be defined using a one-part notation ((`RRECT_WAVE` or `RLONG_WAVE`, see Waveform Sample Value Notations).

`low_bound` and **`high_bound`** are used to exclude values. See Description.

`num_buckets` specifies how many values are to be returned in `out_wave`. See Description.

`waveform_histogram()` returns a count of exception values i.e. a count of values which were NaN (see Description), less than **`low_bound`**, or greater than **`high_bound`**.

Example

???

3.27.29.8 waveform_interleave()

See Waveform Overview, Waveform Analysis Functions.

Description

The `waveform_interleave()` function combines two specified input waveforms into one output waveform, as follows:

- The sample values of the output waveform will consist of alternating sample values from the two input waveforms. The first sample value of the output waveform is the first sample value of the first input waveform. The second sample value of the output waveform is the first sample value of the second input waveform. Etc.
- The two input waveforms must be defined using the same notation (see Waveform Sample Value Notations). `RRECT_WAVE` and `RLONG_WAVE` are considered the same.
- The two input waveforms must either contain the same number of sample values (be the exact same Size), or the first waveform may have one sample value more than the second input waveform.

Usage

```
void waveform_interleave(Waveform* out_wave,
                         Waveform* in_wave1,
                         Waveform* in_wave2 );
```

where:

`out_wave` identifies the output waveform.

`in_wave1` and **`in_wave2`** identify the two input waveforms. See Description.

Example

???

3.27.29.9 waveform_linear_regression()

See Waveform Overview, Waveform Analysis Functions.

Description

The `waveform_linear_regression()` function computes the linear regression parameters of the input waveform using the least squares fit method. The least squares method defines the *best* approximation as the line that produces the minimum sum of squares of the deltas between the actual Y values and the computed Y values.

Note the following:

- The input waveform must be type RLONG_WAVE or RRECT_WAVE and must contain at least two data points.
- The input waveform is not modified.
- The returned slope and intercept values are in terms of the input waveform's X and Y scaling.

Usage

```
void waveform_linear_regression(Waveform* in_wave,
                                double *slope,
                                double *intercept );
```

where:

in_wave specifies the input waveform to be processed.

slope and **intercept** are pointers to two existing double variables used to return the slope and intercept values from the analysis.

Example

???

3.27.29.10 waveform_magnitudes()

See Waveform Overview, Waveform Analysis Functions.

Description

The waveform_magnitudes() function processes each sample value from an input waveform and outputs the magnitude component into the corresponding sample value of the output waveform. Note the following:

- waveform_magnitudes() returns a Waveform* equal in Size to the input waveform.
- The input waveform may be defined using any notation (see Waveform Sample Value Notations). See below for how each notation type is evaluated.
- The output waveform Type is always RRECT_WAVE.

Magnitude is calculated as follows:

- RRECT_WAVE each output sample value = the corresponding input waveform sample value.
- RLONG_WAVE each output sample value = the corresponding input waveform integer sample value converted to real.
- CRECT_WAVE: each output sample value = the absolute value of the corresponding complex input waveform sample value. This is calculated:

  $$|x + yi| = \sqrt{x^2 + y^2}$$

- POLAR_WAVE: the magnitude portion is returned, the phase information is discarded.

Usage

```
void waveform_magnitudes( Waveform* out_wave, Waveform* in_wave );
```

where:

out_wave specified the output waveform.

in_wave specified the input waveform to be processed.

Example

???

3.27.29.11 waveform_median()

See Waveform Overview, Waveform Analysis Functions.

Description

The waveform_median() function is used to return the median value of the sample values of a specified waveform.

For a large set of sample values which exhibit a good distribution, but which may contain some wild outlier values, the median value is often a better quantifier than the mean value (see waveform_geometric_mean()).

Note the following:

- Supports sample values defined using RRECT_WAVE and RLONG_WAVE only. See Waveform Sample Value Notations.

- If the waveform samples are numerically sorted, the median value is the value of the middle element.
- If the waveform contains an even number of samples, the median value is calculated to be the arithmetic mean of the middle two values.

Usage

```
double waveform_median( Waveform* in_wave );
```

where:

in_wave identifies the input waveform.

See Description for the `waveform_median()` return value.

Example

???

3.27.29.12 waveform_min_max()

See Waveform Overview, Waveform Analysis Functions.

Description

The `waveform_min_max()` function is used to identify the minimum and maximum sample values of a specified waveform. The input waveform Type must be `RRECT_WAVE` or `RLONG_WAVE`.

There are two versions of the function.

- The first simply returns the minimum and maximum values.
- The second version also returns the zero-based location (index) of the values. When a minimim and /or maximum value occurs more than once in the waveform, the index points to the first occurrence.

Usage

```
void waveform_min_max( Waveform* in_wave,
                       double *min_value,
                       double *max_value );
```

```
void waveform_min_max( Waveform* in_wave,
                       double *min_value,
                       double *max_value,
                       int *min_index,
                       int *max_index );
```

where:

in_wave identifies the input waveform.

min_value and **max_value** are pointers to existing `double` variables used to return the minimum and maximum sample values identified in **in_wave**.

min_index and **max_index** are pointers to existing `int` variables used to return the index location of the minimum and maximum sample values identified in **in_wave**. See Description.

Example

???

3.27.29.13 waveform_quantize()

See Waveform Overview, Waveform Analysis Functions.

Description

The `waveform_quantize()` function is used to ???. Note the following:

- The input waveform must defined using the `RRECT_WAVE` notation.
- The output waveform will be defined using the `RLONG_WAVE` notation.
- All sample values of the input waveform must be greater than or equal to zero and less than one.
- The `bitwidth` argument to `waveform_quantize()` specifies the number of bits to be used in each sample value of the output waveform. Legal `bitwidth` values must be between 1..54, inclusive.
- Each output sample value is an N-bit representation (N = bitwidth) of the corresponding input sample value, specified as a rounded binary integer. For instance, if the `bitwidth` was specified as 8, then each input sample value would

be multiplied by 2^8 (i.e. 256), and rounded to an integer. The rounding method applied is specifed using the the `rounding_method` argument passed to `waveform_quantize()`. See below.

- Each output sample value is clamped to the range:

  `0 .. (2^bitwidth) - 1`

  This means that the largest input sample value that avoids being clamped is:

$$\frac{(2^{bitwidth} - 1)}{2^{bitwidth}}$$

  i.e. not one (1).

The method used to round each output sample value to an integer is specified using the `rounding_method` argument to `waveform_quantize()`. Legal values are specified using the `RoundingMethod` enumerated type.

Usage

```
void waveform_quantize(  Waveform* out_wave,
                         Waveform* in_wave,
                         int bitwidth,
RoundingMethod rounding_method DEFAULT_VALUE( t_round_to_nearest ) );
```

where:

`out_wave` identifies the output waveform.

`in_wave` identifies the input waveform to be processed.

`bitwidth` describes the number of bits in each output sample. See Description.

`rounding_method` is optional, and if used specifies the method used to round each output sample value to an integer value. Default = `t_round_to_nearest`.

Example

???

3.27.29.14 waveform_rms()

See Waveform Overview, Waveform Analysis Functions.

Description

The `waveform_rms()` function is used to return the root mean square (RMS) of the sample values of a specified waveform. This returns the square root of the arithmetic mean of the squares of the waveform sample values.

Supports sample values defined using `RRECT_WAVE` and `RLONG_WAVE` only. See Waveform Sample Value Notations.

Usage

```
double waveform_rms( Waveform* in_wave );
```

where:

`in_wave` identifies the input waveform.

See Description for the `waveform_rms()` return value.

Example

???

3.27.29.15 waveform_sfdr()

See Waveform Overview, Waveform Analysis Functions.

Description

The `waveform_sfdr()` function is used to determine the *spurious-free-dynamic-range* (SFDR) ratio of a waveform. This is the ratio of the signal to the largest single harmonic or noise component.

Harmonic and noise components are treated identically. And, even though the largest component usually ends up being a harmonic, the expressions below use the term *noise* in reference to both noise and harmonic components.

Note the following:

- The waveform must be the result of an FFT having processed a single sine wave of a coherent frequency (with the usual harmonics, noise, distortion, etc.). See Waveform FFT Functions. Thus, the waveform processed by `waveform_sfdr()` consists of set FFT bins. How these bins are processed to determine SFDR is discussed below.
- The FFT waveform can be in the following notations: `RRECT_WAVE`, `CRECT_WAVE`, and `POLAR_WAVE`, but not `RLONG_WAVE`. See Waveform Sample Value Notations.

`waveform_sfdr()` processes the FFT waveform (bins) as follows:

- The *signal* component of the SFDR analysis is the value in the FFT fundamental bin. This bin may be specified as an argument (`fundamental_bin`) to `waveform_sfdr()`. If not specified, or specified as -1, the largest non-DC bin is assumed to be the fundamental. The value from the fundamental bin is refered to as *F* below.
- Bin-0 is the FFT's DC offset value and is ignored.
- All FFT bins which are not the fundamental or a DC bin are treated as noise bins. This includes harmonic bins.
- The SFDR ratio is returned in dB, i.e.

$$\text{SNR} = 10 * \log10\left(\frac{\text{signal}}{\text{noise}}\right)$$

  where *signal* is the FFT fundamental bin value (F), and *noise* is the largest single noise or harmonic FFT bin value, as noted above.

Aliasing and *Windowing* are handled correctly. See FFT Aliasing and Waveform Window Functions.

Usage

```
double waveform_sfdr(  Waveform* in_wave,
                       int fundamental_bin DEFAULT_VALUE( -1 ) );
```

where:

`in_wave` is the input waveform to be analyzed. This must be the output of an FFT operation, see Description.

`fundamental_bin` is optional, and if used identifies the FFT bin containing the fundamental signal value. See Description for details and default operation.

`waveform_sfdr()` returns the SFDR, in DB.

Example

???

3.27.29.16 waveform_signals_and_noise()

See Waveform Overview, Waveform Analysis Functions.

Description

The `waveform_signals_and_noise()` function may be used to implement a custom noise measurement. Note the following:

- The input waveform should be the result of an FFT. See Waveform FFT Functions.
- Aliasing and windowing are taken into account.

Usage

```
void waveform_signals_and_noise(   Waveform* wave,
                                   DoubleArray *harmonics,
                                   double *noise,
                                   double *spur,
                           int fundamental_bin DEFAULT_VALUE( -1 ),
                           int num_harmonics DEFAULT_VALUE( 9 ) );
```

where:

`wave` identifies the input waveform to be analyzed.

`harmonics` is a `DoubleArray` used to return harmonic information as noted below. Values are in *volts peak squared*. The array is automatically [re]sized based on the value specified for **`num_harmonics`**:

```
harmonics[0]   Signal strength of the DC offset
harmonics[1]   Signal strength of the fundamental
harmonics[2]   Signal strength of the 2nd harmonic
...
harmonics[ num_harmonics + 1] Strength of the last harmonic
```

`noise` is a pointer to an existing `double` variable used to return the total noise strength. This is the sum of squares of all FFT bins except DC, the fundamental, and all considered harmonics.

spur is a pointer to an existing double variable used to return the the largest non-harmonic noise component.

fundamental_bin is optional, and if specified is the FFT bin of the fundamental. If not specified, or specified as -1, the largest non-DC bin is assumed to be the fundamental. Default = -1.

num_harmonics is optional, and if specified is the represents the number of harmonics to be found (in addition to the fundamental). Default = 9.

Example

???

3.27.29.17 waveform_sinad()

See Waveform Overview, Waveform Analysis Functions.

Description

The waveform_sinad() function is used to determine the signal-to-noise-and-distortion (SINAD) ratio of a waveform. Note the following:

- The waveform must be the result of an FFT having processed a single sine wave of a coherent frequency (with the usual harmonics, noise, distortion, etc.). See Waveform FFT Functions. Thus, the waveform processed by waveform_sinad() consists of set FFT bins. How these bins are processed to determine SINAD is discussed below.
- The FFT waveform can be in the following notations: RRECT_WAVE, CRECT_WAVE, and POLAR_WAVE, but not RLONG_WAVE. See Waveform Sample Value Notations.

waveform_sinad() processes the FFT waveform (bins) as follows:

- The *signal* component of the SINAD analysis is the value in the FFT fundamental bin. This bin may be specified as an argument (fundamental_bin) to waveform_sinad(). If not specified, or specified as -1, the largest non-DC bin is assumed to be the fundamental. The value from the fundamental bin is refered to as *F* below.

- SINAD is a power ratio, but FFT bin values (magnitudes) are proportional to voltage. Therefore, the SINAD *noise* and *harmonic* signal strengths are determined by summing the squares of the appropriate FFT bins. However, as noted below, not all bins in the FFT are included.
- Bin-0 is the FFT's DC offset value and is ignored.
- The number of harmonic bins to include may be specified as an argument (`num_harmonics`) to `waveform_sinad()`. If not specified, `num_harmonics` defaults to 9. Then, bins (2 * F), (3 * F),... up to ((`num_harmonics` +1) * F) are included.
- The remaining bins are considered noise. As noted, total noise is the sum of the squares of the values in each of these bins.
- The SINAD ratio is returned in dB, i.e.

$$\text{SINAD} = 10 * \log10\left(\frac{\text{signal}}{\text{noise + harmonics}}\right)$$

  where *signal* is the FFT fundamental bin value (F), *noise* and *harmonics* are calculated as noted above.

Aliasing and *Windowing* are handled correctly. See FFT Aliasing and Waveform Window Functions.

Usage

```
double waveform_sinad( Waveform* in_wave,
                       int fundamental_bin DEFAULT_VALUE( -1 ) );
```

where:

`in_wave` is the input waveform to be analyzed. This must be the output of an FFT operation, see Description.

`fundamental_bin` is optional, and if used identifies the FFT bin containing the fundamental signal value. See Description for details and default operation.

`waveform_sinad()` returns the SINAD, in DB.

Example

???

3.27.29.18 waveform_snr()

See Waveform Overview, Waveform Analysis Functions.

Description

The `waveform_snr()` function is used to determine the signal-to-noise (SNR) ratio of a waveform. Note the following:

- The input waveform must be the result of an FFT having processed a single sine wave of a coherent frequency (with the usual harmonics, noise, distortion, etc.). See Waveform FFT Functions. Thus, the waveform processed by `waveform_snr()` consists of set FFT bins. How these bins are processed to determine SNR is discussed below.
- The FFT waveform can be in the following notations: `RRECT_WAVE`, `CRECT_WAVE`, and `POLAR_WAVE`, but not `RLONG_WAVE`. See Waveform Sample Value Notations.

`waveform_snr()` processes the FFT waveform (bins) as follows:

- The *signal* component of the SNR analysis is the value in the FFT fundamental bin. This bin may be specified as an argument (`fundamental_bin`) to `waveform_snr()`. If not specified, or specified as -1, the largest non-DC bin is assumed to be the fundamental. The value from the fundamental bin is refered to as *F* below.
- SNR is a power ratio, but FFT bin values (magnitudes) are proportional to voltage. Therefore, the SNR *noise* signal strengths are determined by summing the squares of the appropriate FFT bins. However, as noted below, not all bins in the FFT are included.
- Bin-0 is the FFT's DC offset value and is ignored.
- Harmonic bins are not normally considered to be signal or noise and thus should not be included. The number of harmonics to reject may be specified as an argument (`num_harmonics`) to `waveform_snr()`. If not specified, `num_harmonics` defaults to 9. Then, bins (2 * F), (3 * F),... up to ((`num_harmonics` +1) * F) are ignored.
- The remaining bins are considered noise. As noted, total noise is the sum of the squares of the values in each of these bins.

- The SNR ratio is returned in dB, i.e.

$$SNR = 10 * \log10 \left(\frac{signal}{noise} \right)$$

  where *signal* is the FFT fundamental bin value (F) and *noise* is calculated as noted above.
- *Aliasing* and *Windowing* are handled correctly. See FFT Aliasing and Waveform Window Functions.

Usage

```
double waveform_snr(Waveform* in_wave,
                    int fundamental_bin DEFAULT_VALUE( -1 ),
                    int num_harmonics DEFAULT_VALUE( 9 ) );
```

where:

`in_wave` is the input waveform to be analyzed. This must be the output of an FFT operation, see Description.

`fundamental_bin` is optional, and if used identifies the FFT bin containing the fundamental signal value. See Description for details and default operation.

`num_harmonics` is optional, and if used identifies the number of harmonic FFT bins to ignore. See Description for details and default operation.

`waveform_snr()` returns the SNR, in DB.

Example

???

3.27.29.19 waveform_standard_deviation()

See Waveform Overview, Waveform Analysis Functions.

Description

The `waveform_standard_deviation()` function is used to determine the standard deviation of the sample values in a specified waveform.

The standard deviation is simply the square root of the variance:

$$Mean = \bar{X} = \frac{1}{N}\sum X_i$$

$$Variance = \sigma^2 = \frac{1}{N - D_f}\sum (X_i - \bar{X})^2$$

$$StdDev = \sigma = \sqrt{Variance}$$

Also see `waveform_variance()`.

Usage

```
double waveform_standard_deviation(
                      Waveform* in_wave,
                      int degrees_of_freedom DEFAULT_VALUE( 1 ) );
```

where:

`in_wave` identifies the input waveform.

`degrees_of_freedom` is optional, and default = 1. See Description.

See Description for the `waveform_standard_deviation()` return value.

Example

???

3.27.29.20 waveform_sum_of_squares()

See Waveform Overview, Waveform Analysis Functions.

Description

The `waveform_sum_of_squares()` function is used to return the sum of the squares of all sample values. Supports sample values defined using `RRECT_WAVE` and `RLONG_WAVE` only. See Waveform Sample Value Notations.

Usage

```
double waveform_sum_of_squares( Waveform* in_wave );
```

where:

in_wave identifies the input waveform.

Example

???

3.27.29.21 waveform_thd()

See Waveform Overview, Waveform Analysis Functions.

Description

The `waveform_thd()` function is used to determine the total harmonic distortione (THD) of a waveform. Note the following:

- The input waveform must be the result of an FFT having processed a single sine wave of a coherent frequency (with the usual harmonics, noise, distortion, etc.). See Waveform FFT Functions. Thus, the waveform processed by `waveform_thd()` consists of set FFT bins. How these bins are processed to determine THD is discussed below.
- The FFT waveform can be in the following notations: `RRECT_WAVE`, `CRECT_WAVE`, and `POLAR_WAVE`, but not `RLONG_WAVE`. See Waveform Sample Value Notations.

`waveform_thd()` processes the FFT waveform (bins) as follows:

- The *signal* component of the THD analysis is the value in the FFT fundamental bin. This bin may be specified as an argument (`fundamental_bin`) to `waveform_thd()`. If not specified, or specified as -1, the largest non-DC bin is assumed to be the fundamental. The value from the fundamental bin is refered to as *F* below.
- THD is a power ratio, but FFT bin values (magnitudes) are proportional to voltage. Therefore, the THD *harmonics* signal strengths are determined by summing the squares of the appropriate FFT bins.
- Bin-0 is the FFT's DC offset value and is ignored.
- The number of harmonics to include in the analysis may be specified as an argument (`num_harmonics`) to `waveform_thd()`. If not specified, `num_harmonics` defaults to 9. Then, bins (2 * F), (3 * F),... up to ((`num_harmonics` +1) * F) are included in the analysis.

- The THD ratio is returned in dB, i.e.

$$THD = 10 \times \log 10\left(\sum\left(\frac{harmonics}{signal}\right)\right)$$

  where *signal* is the FFT fundamental bin value (F) and *harmonics* is calculated as noted above.
- *Aliasing* and *Windowing* are handled correctly. See FFT Aliasing and Waveform Window Functions.

Usage

```
double waveform_thd(Waveform* in_wave,
                    int fundamental_bin DEFAULT_VALUE( -1 ),
                    int num_harmonics DEFAULT_VALUE( 9 ) );
```

where:

`in_wave` is the input waveform to be analyzed. This must be the output of an FFT operation, see Description.

`fundamental_bin` is optional, and if used identifies the FFT bin containing the fundamental signal value. See Description for details and default operation.

`num_harmonics` is optional, and if used identifies the number of harmonic FFT bins to include in the analysis. See Description for details and default operation.

`waveform_thd()` returns the THD, in DB.

Example

???

3.27.29.22 waveform_variance()

See Waveform Overview, Waveform Analysis Functions.

Description

The `waveform_variance()` function is used to obtain the *variance* of the sample values of a specified waveform.

The arithmetic mean (or average) value of a large group of numbers helps to quantify what the values represent, but does not convey the distribution of the values. *variance* is a

measure of the average deviation from the average, specifically, it is the average of the deviations squared:

$$Mean = \bar{X} = \frac{1}{N}\sum X_i$$

$$Variance = \sigma^2 = \frac{1}{N - D_f}\sum (X_i - \bar{X})^2$$

$$StdDev = \sigma = \sqrt{Variance}$$

where D_f refers to the degrees of freedom. There is a lot of statistical theory behind the meaning of D_f and how to choose its value. Since the average of the data upon which the deviations are based comes from the data itself, setting $D_f = 1$ is usually indicated. For other applications, refer to a good statistical textbook.

Also see `waveform_standard_deviation()`.

Usage

```
double waveform_variance(
                    Waveform* in_wave,
                    int degrees_of_freedom DEFAULT_VALUE( 1 ) );
```

where:

`in_wave` identifies the input waveform.

`degrees_of_freedom` is optional, and default = 1. See Description.

See Description for the `waveform_variance()` return value.

Example

???

3.27.30 INL & DNL Functions

See Waveform Functions, Waveform Analysis Functions.

INL and DNL are key performance specifications for ADCs and DACs, reporting linearity performance as the device output changes from one value to another value, typically across the entire output range of the device.

For a perfect DAC, the transfer function from digital-code-in to analog-voltage-out (or vice versa for an ADC) should be a straight line. For example, to test an eight bit DAC the digital inputs are sequenced between codes 0 through 255, and the analog output is captured using the external hardware. For a perfect DUT, if the sample values are plotted the result will be a straight line, between 0-255. Any deviation from a perfectly straight line indicates some degree of non-linearity. The purpose of the INL/DNL functions is to analyze a histogram of a captured waveform's sample data and report the linearity parameters noted above.

The following functions are used to analyze and return integral and differential nonlinearity information about a specified waveform (histogram):

- `waveform_adc_ramp_inl_dnl()`
- `waveform_adc_sine_inl_dnl()`
- `waveform_dac_ramp_inl_dnl()`

The following INL/DNL information is returned by the analysis:

- INL = Integral Non-Linearity. This is an array of sample values (stored as a `Waveform*`) which represents the ILE for each code value.
- ILE = Integral Linearity Error. This is the maximum magnitude relative input-to-output error.
- ILE index. This is the digital code at which the ILE error occured. Note: the value returned is actually the index into the histogram, see below.
- DNL = Differential Non-Linearity. This is an array of sample values (stored as a `Waveform*`) which represents the DLE for each code value.
- DLE = Differential Linearity Error. This is the largest voltage step error when switching from one code to the next.
- DLE index. This is the digital code at which the DLE error occured. Note: the value returned is actually the index into the histogram, see below.

ILE and DLE values are signed i.e. if just the error magnitudes are needed the `waveform_absolute_value()` function must be applied.

The INL & DNL Functions provide several *line fit* options used when performing ILE analysis:

- `t_least_squares_fit` causes a line to be fitted to the data using a standard least squares algorithm. Such a line may not pass directly through the endpoints, but will have the minimum sum of squared data point deviations. It will generate a statistically better data fit than either the `t_endpoint_fit` or `t_adjusted_fit` methods. This option is usable by `waveform_dac_ramp_inl_dnl()` only.

- t_endpoint_fit results in the maximum ILE being computed as the maximum distance from a straight line intersecting the endpoints of the histogram data.
- t_adjusted_fit improves (reduces) the ILE by moving the Y intercept of the endpoint line such that the line travels through a point midway between the most positive error (point furthest above the line) and the most negative error (point furthest below the line). The slope of the line is not changed. The *adjusted* line is now equidistant from the most positive error and the most negative error. Of these two points, the one that occurs first (i.e. earliest index in sample value array) is returned as the ILE and ILE index:

Expected Usage

Testing an 8-bit ADC:

- The DUT is caused to repeatedly sample and convert a repetative ramp (or sine wave) input signal generated by the external hardware.
- The DUT outputs the corresponding digital samples (codes) which are captured using the Error Catch RAM (ECR).
- The digital output is normalize, using:
  ```
  waveform_rescale( capturedWF, codesWF, 0, 255 )
  ```
  See waveform_rescale(). This is done to ???
- When testing ADC circuits, generate a histogram from the normalized output waveform:
  ```
  waveform_histogram( histoWF, codesWF, -0.5, 255.5, 256 )
  ```
  See waveform_histogram(). Because an 8-bit device is being tested, the resulting histogram has 256 bins, each containing a count representing the number of times each of the 256 potential codes showed up in the captured and normalized sample data. For a perfect DUT, the count in each histogram bin should match the number of ramps sampled. In this example, the histogram output (waveform) is named histoWF.
- Execute the appropriate INL/DNL function to analyze the histogram and return the linearity results. When testing an ADC the function name will include _adc_. For example:
  ```
  waveform_adc_ramp_inl_dnl(  histoWF,
                              &dle, &ile,
                              &dle_index, &ile_index );
  ```
  See waveform_adc_ramp_inl_dnl() or waveform_adc_sine_inl_dnl().

Testing an 8-bit DAC, the process is identical except

- The digital input (waveform) to the DUT comes from a Logic Test Pattern.

- The DUT output waveform is captured using external equipment.
- The `waveform_dac_ramp_inl_dnl()` function is used to perform the analysis.
- The output waveform is not converted into a histogram before being processed by `waveform_dac_ramp_inl_dnl()`.
- The `t_least_squares_fit` line fitting option is available.

3.27.30.1 waveform_adc_ramp_inl_dnl()

See Waveform Overview, Waveform Analysis Functions, INL & DNL Functions.

Description

The `waveform_adc_ramp_inl_dnl()` function is used to analyze and return INL/DNL specifications about a linear ramp waveform (histogram) acquired while testing an ADC circuit. Note the following:

- The input waveform to `waveform_adc_ramp_inl_dnl()` must be a histogram of the waveform samples acquired by the capture instrument. See `waveform_histogram()` and Expected Usage in INL & DNL Functions.
- The histogram waveform must be defined using a one-part notation i.e. `RLONG_WAVE` or `RRECT_WAVE` (see Waveform Sample Value Notations).
- Only two of the line fitting options may be used: `t_endpoint_fit` and `t_adjusted_fit`.

Regarding the transfer functions returned in the `out_inl` and `out_dnl` `Waveform*`s:

- Both the `X_units` and `Y_units` set to match the `X_units` value of `in_wave`.
- The `X_increment` value matches the `in_wave`.
- The output waveforms have 2 fewer samples than the input. The 2 missing values correspond to the first and last samples of the input waveform. The X axis value associated with the first sample of the output waveforms is eqaul to the X axis value of the second sample of the input waveform, but the X increment value remains the same.

More details are documented in INL & DNL Functions.

Usage

```
void waveform_adc_ramp_inl_dnl(  Waveform* in_wave,
                                 Waveform* out_inl,
                                 Waveform* out_dnl,
                                 double *ile,
                                 double *dle,
                                 int *ile_index,
                                 int *dle_index,
            LineFitMethod fit DEFAULT_VALUE( t_adjusted_fit ) );
void waveform_adc_ramp_inl_dnl(  Waveform* in_wave,
                                 double *ile,
                                 double *dle,
                                 int *ile_index,
                                 int *dle_index,
            LineFitMethod fit DEFAULT_VALUE( t_adjusted_fit ) );
```

where:

`in_wave` identifies the target waveform (histogram) to be analyzed.

`out_inl` specifies an output waveform used to return sample values, each representing the ILE value for the corresponding input waveform sample value.

`out_dnl` specifies an output waveform used to return sample values, each representing the DLE value for the corresponding input waveform sample value.

`ile` is a pointer to an existing `double` variable used to return the ILE value from the analysis. The value is signed, thus if just the error magnitude is desired, the test program must apply an absolute operation.

`dle` is a pointer to an existing `double` variable used to return the DLE value from the analysis. The value is signed, thus if just the error magnitude is desired, the test program must apply an absolute operation.

`ile_index` is a pointer to an existing `int` variable used to return the the digital code at which the ILE error occured. **`ile_index`** is actually the zero-based index into the histogram waveform.

`dle_index` is a pointer to an existing `int` variable used to return the the digital code at which the DLE error occured. **`dle_index`** is actually the zero-based index into the histogram waveform.

`fit` is optional, and if used specifies a line fitting option used during the analysis. **`fit`** only affects the ILE determinations. Legal values are of the `LineFitMethod` enumerated type

but only t_adjusted_fit and t_endpoint_fit are valid for use with the waveform_adc_ramp_inl_dnl() function (see INL & DNL Functions). Default = t_adjusted_fit.

Example

???

3.27.30.2 waveform_adc_sine_inl_dnl()

See Waveform Overview, Waveform Analysis Functions, INL & DNL Functions.

Description

The waveform_adc_sine_inl_dnl() function is used to analyze and return INL/DNL specifications about a sine wave waveform (histogram) acquired while testing an ADC circuit. Note the following:

- The input waveform to waveform_adc_sine_inl_dnl() must be a histogram of the waveform samples acquired by the capture instrument. See waveform_histogram() and Expected Usage in INL & DNL Functions.
- The histogram waveform must be defined using a one-part notation i.e. RLONG_WAVE or RRECT_WAVE (see Waveform Sample Value Notations).
- Only two of the line fitting options may be used: t_endpoint_fit and t_adjusted_fit.

Regarding the transfer functions returned in the out_inl and out_dnl Waveform*s:

- Both the X_units and Y_units set to match the X_units value of in_wave.
- The X_increment value matches the in_wave.
- The output waveforms have 2 fewer samples than the input. The 2 missing values correspond to the first and last samples of the input waveform. The X axis value associated with the first sample of the output waveforms is eqaul to the X axis value of the second sample of the input waveform, but the X increment value remains the same.

More details are documented in INL & DNL Functions.

Usage

```
void waveform_adc_sine_inl_dnl(  Waveform* in_wave,
                                 Waveform* out_inl,
                                 Waveform* out_dnl,
                                 double *ile,
                                 double *dle,
                                 int *ile_index,
                                 int *dle_index,
            LineFitMethod fit DEFAULT_VALUE( t_adjusted_fit ) );
void waveform_adc_sine_inl_dnl(  Waveform* in_wave,
                                 double *dle,
                                 double *ile,
                                 int *dle_index,
                                 int *ile_index,
            LineFitMethod fit DEFAULT_VALUE( t_adjusted_fit ) );
```

where:

in_wave identifies the target waveform (histogram) to be analyzed.

out_inl specifies an output waveform used to return sample values, each representing the ILE value for the corresponding input waveform sample value.

out_dnl specifies an output waveform used to return sample values, each representing the DLE value for the corresponding input waveform sample value.

ile is a pointer to an existing `double` variable used to return the ILE value from the analysis. The value is signed, thus if just the error magnitude is desired, the test program must apply an absolute operation.

dle is a pointer to an existing `double` variable used to return the DLE value from the analysis. The value is signed, thus if just the error magnitude is desired, the test program must apply an absolute operation.

ile_index is a pointer to an existing `int` variable used to return the the digital code at which the ILE error occured. **ile_index** is actually the zero-based index into the histogram waveform.

dle_index is a pointer to an existing `int` variable used to return the the digital code at which the DLE error occured. **dle_index** is actually the zero-based index into the histogram waveform.

fit is optional, and if used specifies a line fitting option used during the analysis. **fit** only affects the ILE determinations. Legal values are of the `LineFitMethod` enumerated type

but only t_adjusted_fit and t_endpoint_fit are valid for use with the waveform_adc_sine_inl_dnl() function (see INL & DNL Functions). Default = t_adjusted_fit.

Example

???

3.27.30.3 waveform_dac_ramp_inl_dnl()

See Waveform Overview, Waveform Analysis Functions, INL & DNL Functions.

Description

The waveform_dac_ramp_inl_dnl() function is used to analyze and return INL/DNL specifications about a linear ramp waveform acquired when testing an DAC. Note the following:

- The input waveform to waveform_dac_ramp_inl_dnl() should be a scaled time domain waveform (i.e. X_units = SCALE_SECONDS and Y_units = SCALE_VOLTS) or unscaled time domain waveform (i.e. X_units = SCALE_SECONDS and Y_units = SCALE_CODES).
- The input waveform should be the normalized (see Expected Usage) and not a histogram. This is a representation of the DAC transfer function, ideally a near perfect ramp.
- The input waveform must be defined using a one-part notation i.e. RLONG_WAVE or RRECT_WAVE (see Waveform Sample Value Notations).
- All three line fitting options may be used with waveform_dac_ramp_inl_dnl().

Regarding the transfer functions returned in the out_inl and out_dnl Waveform*s:

- Both the X_units and Y_units set to match the Y_units value of in_wave.
- The first X axis value of the output waveforms will be equal to the *perceived* Y axis value of the first sample of the input waveform.
- The last X axis value of the output waveforms will be equal to the *perceived* Y axis value of the last sample of the input waveform.
- Ideally, the ramp is linear, with uniform step sizes, so the output waveforms X_increment value is set to the calculated uniform step. Therefore the input Y range of values is mapped to the output X range of values.

- The term *perceived* is used to emphasize that an analysis is being performed, which includes a line-fit option. Where the sample points are determined to be depends on the straight line which is calculated through the data. And the straight line which is calculated depends not only on the data, but on the calculation (line fit) method. If `t_endpoint_fit` is used, then the first and last data points define the data range. But if a different line fit method is used, those endpoints may get shifted a bit. This will result in slightly different values along the X axes of the output waveforms.

More details are documented in INL & DNL Functions.

Usage

```
void waveform_dac_ramp_inl_dnl(  Waveform* in_wave,
                                 Waveform* out_inl,
                                 Waveform* out_dnl,
                                 double *ile,
                                 double *dle,
                                 int *ile_index,
                                 int *dle_index,
                LineFitMethod fit DEFAULT_VALUE( t_adjusted_fit ) );
void waveform_dac_ramp_inl_dnl(  Waveform* in_wave,
                                 double *ile,
                                 double *dle,
                                 int *ile_index,
                                 int *dle_index,
                LineFitMethod fit DEFAULT_VALUE( t_adjusted_fit ) );
```

where:

`in_wave` identifies the target waveform (histogram) to be analyzed.

`out_inl` specifies an output waveform used to return sample values, each representing the ILE value for the corresponding input waveform sample value.

`out_dnl` specifies an output waveform used to return sample values, each representing the DLE value for the corresponding input waveform sample value.

`ile` is a pointer to an existing `double` variable used to return the ILE value from the analysis. The value is signed, thus if just the error magnitude is desired, the test program must apply an absolute operation.

dle is a pointer to an existing `double` variable used to return the DLE value from the analysis.The value is signed, thus if just the error magnitude is desired, the test program must apply an absolute operation.

ile_index is a pointer to an existing `int` variable used to return the the digital code at which the ILE error occured. **ile_index** is actually the zero-based index into the histogram waveform.

dle_index is a pointer to an existing `int` variable used to return the the digital code at which the DLE error occured. **dle_index** is actually the zero-based index into the histogram waveform.

fit is optional, and if used specifies a line fitting option used during the analysis. **fit** only affects the ILE determinations. Legal values are of the `LineFitMethod` enumerated type (see INL & DNL Functions). Default = `t_adjusted_fit`.

Example

???

Chapter 4 Test Pattern Programming

- MAR DONE and/or VAR DONE
- Pattern Subroutines
- Error Pipeline Requirements
- Algorithmic Pattern Generator (APG) Configuration
  - Too many to list here.
- Memory Test Patterns
  - Overview
  - Memory Pattern Instruction Format
  - Default Memory Pattern Instruction
  - APG Instruction Execution
  - APG Address Generator Overview
  - YALU Instruction and XALU Instruction
  - COUNT Instruction
  - MAR Instruction
  - CHIPS Instruction
  - DATGEN Instruction
  - UDATA Instruction
  - PINFUNC Instruction
  - USERRAM Instruction
  - Minmax Pattern Example
  - Adaptive Programming Pattern Example
  - Over-programming Controls and Parallel Test
- Logic Test Patterns
  - Overview
  - Logic Vector Syntax
    - Logic Vector Bit Codes
    - 3-bits per Pin
  - Magnum 1/2/2x Logic Pattern Rules
    - LVM Branch/Label Limitations
  - VECDEF Compiler Directive
  - VEC Pattern Instruction
  - RPT Pattern Instruction
  - Optional VEC/RPT Instruction Parameters

4.1 Overview

See Test Pattern Programming.

The main purpose of a test pattern is to:

- Supply logic state information (the 1's and 0's) needed to functionally exercise the DUT.
- Control when pins are strobed and not strobed. Only strobed pins can cause the test pattern to fail and will log errors into the Error Catch RAM (ECR).
- Control the cycle-by-cycle I/O state of each pin.
- Make the cycle-by-cycle Time-sets (TSET).
- Make the cycle-by-cycle Pin Scramble Map selection.
- Make the cycle-by-cycle VIHH Map selection.

The Magnum 1 hardware provides 2 sources of pattern state information:

Table 4.1.0.0-1 Test Pattern Data Sources

Source	Comments
APG	For testing memory devices. Algorithmically generate logic state information in the form of X/Y Addresses, read/write data, and chip selects. The APG also has a Data Buffer Memory (DBM) for storing random read/write data while generating algorithmic addresses and chip selects.
The combined Logic Vector Memory (LVM) / Scan Vector Memory (SVM)	Magnum 1/2/2x use the same memory to store both Logic Test Patterns and Scan Test Patterns.

During test pattern executioneach pin-pair's (see Functional Pin-pairs) data source is selected cycle-by-cycle using a Pin Scramble Map. Note that within a given test pattern these data sources can be used individually (see Logic Test Patterns, Memory Test Patterns, Scan Test Patterns), or combined to generate Mixed Memory/Logic Patterns, to test devices containing both memory and/or logic and/or scan logic.

In addition to supplying drive/expect pattern data the test pattern also provides the following controls:

Table 4.1.0.0-2 Test Pattern Control Applications

Control	Comment
Strobe Control	Per-pin, per-cycle strobe enable/disable. Memory Test Patterns use MAR READ, READUDATA, READV, READZ or NOREAD. Logic Test Patterns and Scan Test Patterns use Logic Vector Bit Codes.
I/O Control	Per-pin, per-cycle drive/tri-state control. Memory Test Patterns use PINFUNC ADHIZ. Logic Test Patterns and Scan Test Patterns use Logic Vector Bit Codes.
Test Pattern Execution Sequence Control	Controls the test pattern execution engine(s). All Maverick-I test patterns are controlled using MAR instructions. Otherwise... Memory Test Patterns are controlled using MAR instructions. Logic Test Patterns are controlled using using VEC/RPT and VAR instructions. Mixed Memory/Logic Patterns may used both methods.
Time-set Selection (TSET#)	Per-cycle selection of cycle period and per-pin Timing Formats and edge times.
VIHH Map Selection (VIHH#)	Per-cycle selection of which pins are driven to the VIHH level.
Pin Scramble Map Selection (PS#)	Per-pin/per-cycle selection of pattern data source.
Trigger DC Comparators and Error Logic and DC A/D Converter	Per-cycle. Using the MAR VCOMP instruction (Memory Test Patterns) or VEC VCOMP, VAR VCOMP, or VPINFUNC VCOMP instructions (Logic Test Patterns), the test pattern can trigger DC Go/NoGo tests or measurements.

Table 4.1.0.0-2 Test Pattern Control Applications *(Continued)*

Control	Comment
Set or modify (tweak) most PE levels	See Controlling PE Levels from the Test Pattern.
Set or modify (tweak) PMU voltages and currents	See Controlling PE Levels from the Test Pattern.
Set or modify (tweak) DPS DC levels	The test pattern can cause the DPS to switch between 2 previously programmed voltages (see `dps()` and `dps_vpulse()`). And, DPS primary voltage and vpulse voltages can be modified. DPS current high/low test limits have limited support. See Controlling PE Levels from the Test Pattern.

Test patterns are used when performing the following types of tests:

Table 4.1.0.0-3 Test Pattern Applications

Test Type	Test Pattern Purpose
`funtest()`	Validate the logic functionality of the DUT. Validate the AC performance of the DUT under specified DC conditions. Functionally program programmable DUTs. Set up the DUT's logic state prior to executing other tests (PMU, DPS current, mixed signal, etc.).
`ac_partest()`	Perform a dynamic PMU test; i.e. while concurrently executing a functional test pattern. Optionally, trigger the DC comparators from the pattern, possibly multiple times.

Table 4.1.0.0-3 Test Pattern Applications *(Continued)*

Test Type	Test Pattern Purpose
ac_test_supply()	Perform a dynamic DPS current test; i.e. while concurrently executing a functional test pattern. Optionally, trigger the DC comparators from the pattern, possibly multiple times.
hv_ac_test_supply()	Perform a dynamic HV voltage or current test; i.e. while concurrently executing a functional test pattern. Optionally, trigger the DC comparators or DC A/D Converter from the pattern, possibly multiple times.
start_pattern(), stop_pattern(), restart()	Supports pattern looping while user code continues to execute.

Test pattern source files are written using a proprietary pattern language, documented in Test Pattern Programming, which is divided into three sections, corresponding to the pattern data sources noted above:

- Memory Test Patterns
- Logic Test Patterns
- Scan Test Patterns

4.2 Magnum 1/2/2x Pattern Features

See Test Pattern Programming.

Note: this section includes information about memory pattern instructions which were first added for Maverick-II and which are also supported on Magnum 1/2/2x. This section does not include new instructions added just for Magnum 1 and/or Magnum 2 and/or Magnum 2x.

See Test Pattern Programming.

The Magnum 1/2/2x test pattern hardware design enables several test pattern features not available using Maverick-I:

- The APG design includes a logic vector execution control engine (VAR Engine), and associated instruction memory (vRAM). This allows execution of logic instructions to be controlled independently of memory pattern instructions (which are controlled by the MAR Engine). And, unlike on Maverick-I, Logic Test Pattern instructions are not stored in MAR Engine instruction memory. The VAR pattern instruction (Logic Test Patterns) controls logic pattern execution, even in Mixed Memory/Logic Patterns. See Magnum 1/2/2x Logic Vector Instructions.
- In Mixed Memory/Logic Patterns, logic instruction selection of Time-set, Pin Scramble Map, and VIHH Map may take precedence over memory instruction selection: the memory vs. logic instruction precedence is independently selectable for each parameter, per-instruction, using the PINFUNC Instruction.
- Similarly logic instructions can take precedence in controlling NOLATCH, RESET, OVER, VCOMP, VPULSE. See Magnum 1/2/2x Logic Vector Instructions.
- Four logic vector hardware counters, controlled using the VCOUNT pattern instruction, are used to control loops (RPT and STARTLOOP/ENDLOOP). They may also be used for conditional branch, and Pattern Subroutine call/return operations, independent of the MAR Engine. See Magnum 1/2/2x Logic Vector Instructions.
- The VUDATA instruction supports UDATA-like operations for logic vectors independent of memory instructions.
- Additional keywords are available for both Magnum 1/2/2x Memory Pattern Instructions and Magnum 1/2/2x Logic Vector Instructions to support these features.
- A 4K APG User RAM is available to save or modify values in various APG registers or to copy values between various APG registers. See APG User RAM.

4.2.1 Magnum 1/2/2x Memory Pattern Instructions

See Magnum 1/2/2x Pattern Features.

Note: this section includes memory pattern instructions which were newly added for Maverick-II and which are also supported on Magnum 1/2/2x. This section does not include new instructions added just for Magnum 1 and/or Magnum 2 and/or Magnum 2xI.

Also see Magnum 1/2/2x Pattern Features and Magnum 1/2/2x Logic Vector Instructions:

```
MAR       VCNTR
PINFUNC   VPS, VTSET, VVIHH, VVPULSE, VVCOMP, VLATCHRESET, VOVER
```

where:

MAR VCNTR is used to specify that a MAR Engine conditional branch decision is to be based on the value in one of the 4 VAR Engine counters, instead of using one of the 60 MAR Engine counters. This allows memory pattern execution to branch based upon the specified VAR Engine counter value, (see VCOUNT). The MAR Engine cannot control the VAR Engine counters, but does receive a signal when the counter selected in the current instruction reaches zero. Using this feature does consume vRAM, to identify the VAR Engine counter, but the pattern must be a Mixed Memory/Logic Patterns for this to be useful anyway. For example:

```
% VCOUNT    COUNT2
  MAR       VCNTR, CJMPNZ, some_label
```

In this example, the MAR instruction will jump to *some_label* if the VAR Engine counter COUNT2 is not zero. Note that the logic vector instruction which causes COUNT2 to reach zero may not be in the same pattern source statement as the MAR instruction which evaluates VAR Engine counter COUNT2 ≠ 0.

PINFUNC VVCOMP selects the VAR Engine as the source of the enable signal to the PMU/DPS comparators, for the current instruction. Effectively disables the MAR VCOMP signal for the current instruction. Default is MAR Engine signal is selected.

PINFUNC VLATCHRESET selects the VAR Engine as the source of both the error flag reset signal and the latch/nolatch signal, for the current instruction. See Error Flag vs. Error Latch. Effectively disables the MAR Engine RESET signal and MAR Engine NOLATCH signal for the current instruction. Default is to select the MAR Engine signals.

PINFUNC VOVER selects the VAR Engine as the source of the over programming inhibit signal, for the current instruction. See Over-programming Controls and Parallel Test. Effectively disables the MAR Engine OVER signal for the current instruction. Default is to select the MAR Engine signal.

PINFUNC VPS selects the VAR Engine as the source of the pin scramble selection (PS#) for the current instruction. See Pin Scramble Map. Effectively disables the MAR Engine PINFUNC pin scramble selection (PS#) for the current instruction. Default is to select the MAR Engine signal.

PINFUNC VTSET selects the VAR Engine as the source of the time-set selection (TS#) for the current instruction. See Time-sets (TSET). Effectively disables the MAR Engine PINFUNC time-set selection (TSET#) for the current instruction. Default is to select the MAR Engine signal.

PINFUNC VVIHH selects the VAR Engine as the source of the VIHH Map selection (VIHH#) for the current instruction. See VIHH Maps. Effectively disables the MAR Engine PINFUNC

VIHH Map selection (VIHH#) for the current instruction. Default is to select the MAR Engine signal.

PINFUNC VVPULSE selects the VAR Engine as the source of the signal used to switch one or more DUT Power Supply(s) to their 2nd voltage level (see dps_vpulse()). Effectively disables the MAR Engine PINFUNC VPULSE signal for the current instruction. Default is to select the MAR Engine signal.

4.2.2 Magnum 1/2/2x Logic Vector Instructions

See Magnum 1/2/2x Pattern Features.

Note: this section includes logic pattern instructions which were newly added for Maverick-II and which are also supported on Magnum 1/2/2x. This section does not include new instructions added just for Magnum 1 and/or Magnum 2 and/or Magnum 2x.

This is a summary of logic pattern instructions unique to Maverick-II and Magnum 1/2/2x; i.e. not usable with Maverick-I. Also see Magnum 1/2/2x Pattern Features and Magnum 1/2/2x Memory Pattern Instructions.

The Maverick-II and Magnum 1/2/2x have independent hardware test pattern execution control engines for the memory pattern (MAR Engine) and the logic pattern (VAR Engine). For logic patterns, the associated logic pattern instruction and option keywords are listed below. These are stored in the VAR Engine vRAM (not MAR Engine uRAM). Except for the VEC and RPT instructions, these instructions are not usable in Maverick-I test patterns. The VEC and RPT instructions are usable on all system types, however, six of optional parameters are not usable on Maverick-I (VPULSE, VCOMP, RESET, LATCH, NOLATCH, OVER).

```
VEC     HL10X..X01LH TSET#, PS#, VIHH#, VPULSE, VCOMP, RESET,
        LATCH, NOLATCH, OVER
RPT 10  HL10X..X01LH TSET#, PS#, VIHH#, VPULSE, VCOMP, RESET,
        LATCH, NOLATCH, OVER
VAR     CSUB{E, NE, A, NA, T, NT, Z, NZ},
        CJMP{E, NE, A, NA, T, NT, Z, NZ},
        CRET{E, NE, A, NA, T, NT, Z, NZ},
        GOSUB, PAUSE, JUMP, INC,
```

```
        DONE, PAUSE, MCNTR
        VPULSE, VCOMP, RESET, LATCH, NOLATCH, OVER, DEFAULT
VCOUNT  COUNT#, INCR, DECR, DEC2, COUNTVUDATA
VUDATA  value
VPINFUNC  TSET#, PS#, VIHH#, VPULSE, VCOMP, RESET, LATCH,
          NOLATCH, OVER
```

where:

The VEC and RPT logic pattern instructions, in addition to specifying the HL10X pattern data, may also specify the optional parameters noted above. Unlike Maverick-I logic patterns, the VEC and RPT instructions in a Maverick-II and Magnum 1/2/2x logic pattern can also specify the optional parameters previously only controllable using memory pattern instructions; i.e. PINFUNC VPULSE, MAR VCOMP, RESET, LATCH, NOLATCH, and OVER. Special rules apply, see Optional VEC/RPT Instruction Parameters. Note that these options can also be specified using the VAR and VPINFUNC instructions (but duplicate definitions in the same instruction are illegal).

The VAR instructions control the VAR Engine similar to how the MAR instruction is used in memory patterns to control execution of the MAR Engine.

In Mixed Memory/Logic Patterns, in cases where the test pattern specifies the same option for the MAR Engine and the VAR Engine, the MAR instruction control is used by default. To select the VAR Engine option requires using specific PINFUNC options (operands), one for each parameter controlled by the VPINFUNC instruction (which are also available to VEC and RPT). The rules are documented in Optional VEC/RPT Instruction Parameters. Also see Magnum 1/2/2x Memory Pattern Instructions.

VAR MCNTR is used to specify that a VAR conditional branch decision is to be based on the value in one of the 60 MAR counters instead of using one of the 4 VAR Engine counters. This allows logic vector execution control to branch based on a specified MAR Engine counter value.

VCOUNT selects and controls one of the 4 VAR Engine counters.

VUDATA is the logic vector equivalent of memory pattern UDATA. Like the UDATA field, VUDATA is used both explicitly and implicitly (see VUDATA Instruction), and thus must not contain user values in some instructions.

VPINFUNC is the logic vector equivalent of of memory pattern PINFUNC. It can be used in logic patterns to select TSET#, PS#, VIHH#, VPULSE, VCOMP, RESET, LATCH, NOLATCH, OVER. For convenience, these same parameters can also be controlled from the VEC/RPT and VAR instructions. As noted above, in Mixed Memory/Logic Patterns, the MAR Engine

PINFUNC selections take precidence, and additional MAR PINFUNC operands are required to enable each of the VPINFUNC selections. See Optional VEC/RPT Instruction Parameters and Magnum 1/2/2x Memory Pattern Instructions.

The following APG memory instruction options (operands) do NOT have logic vector equivalents:

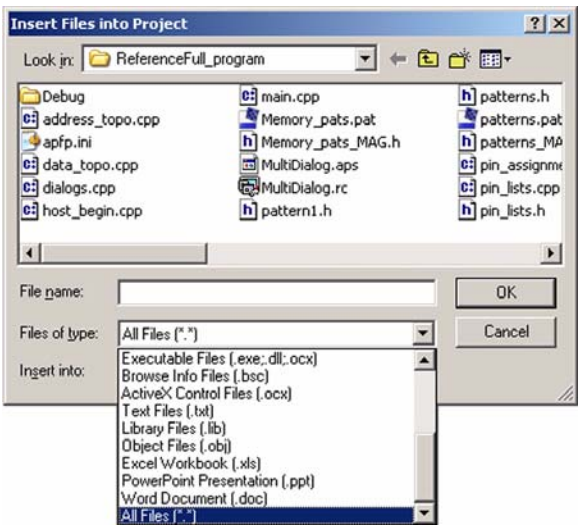
LBDATA, READ, READUDATA, TIMEN, RSTTMR

4.3 Adding a New Pattern File to the Project

See Test Pattern Programming.

Note: this topic and related methods have evolved along with the Nextest software. In software release h2.3.xx/h1.3.xx, a 2nd generation Automated Pattern File Processing (APFP) was introduced, replacing the previous implementation of APFP. The earlier documentation on this topic was deleted, included details of the first generation APFP and the even older methods used before APFP was added.

A test pattern *.pat* file is added to a project using the same method as when adding a C-code *.cpp*, using **Project->Add to Project->Files...**. The following browser is presented:



Note that initially, files with the *.pat* extension are not visible in the browser. In the **Files of type** menu, select **All Files(*)** to include *.pat* files in the display.

The Automated Pattern File Processing (APFP) software and APFP Dialog provide additonal controls for setting options related to compiling test patterns. Once these options are set, APFP manages the pattern build details.

4.3.1 Automated Pattern File Processing (APFP)

See Adding a New Pattern File to the Project.

Note: the APFP implementation changed in software release h2.2.xx/h1.2.xx.

This section includes:

- Overview
- APFP Dialog
  - APFP Button Missing
- Build (Compile) Operation
- APFP Migrating from Older Versions
  - Migrating Programs to h2.3.xx/h1.3.xx
  - Switching Back to Software Prior to h2.3.xx/h1.3.xx

4.3.1.1 Overview

See Adding a New Pattern File to the Project, Automated Pattern File Processing (APFP).

Automated Pattern File Processing (APFP) is the name given to the mechanism which automates many of the details of compiling Magnum 1/2/2x test patterns in a Visual Studio project. In addition, the APFP Dialog contains several tools (buttons) used to perform common tasks related to test patterns.

For the most part, the user adds test pattern source files (*.pat* files) to the test program (to the Visual Studio project), the same as done with C-code *.cpp* files. When the program is compiled (built) the test patterns are compiled too.

In software release h2.2.xx/h1.2.xx APFP was re-implemented (generation two), with the following improvements and features:

APFP was re-implemented (generation two), with the following improvements and features:

- Tighter integration with Visual Studio, including:
  - The Visual Studio build facilities now process pattern files much the same as *.cpp* files. For example, **Build->Rebuild->All** recompiles all test patterns too, **Build->Build** performs an incremental build, etc.
  - The first-generation APFP tool-bar is replaced by a single button  which, once docked to the Visual Studio tool-bar, remains docked. Clicking this button displays the APFP Dialog, which provides controls for various options and utilities.
- The APFP Dialog contains a new utility (button) which removes test pattern *.cpp* and *.h* files from the project. These are the files which are automatically added to the Visual Studio project by the APFP facility.
- The APFP Dialog contains a new utility (button) which will delete from disk all of the files and folders generated by the pattern compiler.
- Support for `#include` statements in test patterns. Important: see APFP Utilities, specifically Clear all Custom Build Steps and Scan Patterns for Dependencies.

The new APFP Dialog does not contain several buttons/features seen in the earlier, first-generation, APFP's four-button tool-bar:

- **Compile All Patterns** is now handled by Visual Studio's build facilities; i.e. **Build->Build** (F7).
- **Recompile All Patterns** is now handled by Visual Studio's build facilities, performing an incremental compile invoked using **Build->Rebuild->All**.
- **Clean All Patterns** is replaced by the Clean Pattern Binaries utility.

Note: APFP expects that the file name of test program executable file (i.e. *.../Debug/myProg.exe*) matches the file name of the Visual Studio project file (i.e. *.../myProg.dsp*). If these file names do not match, APFP execution will enter an end-less loop, continuously compiling test patterns in stub mode.

Note: the Visual Studio's build facility does not detect when folders created by the pattern compiler are deleted from disk (which some users do routinely when archiving a test program). When a given test pattern file (*.*pat*) contains logic/scan patterns the folder created by the pattern compiler contains files which must be present in order for the patterns to be usable in the hardware. When these folders are deleted but the corresponding *.*cpp* file still exists the project build will succeed, with no errors, but the pattern won't load because the required folder contents are gone. To recover, use APFP's Clean Pattern Binaries button before compiling.

Note: Developer Studio service-pack 6 must be installed beginning with software release h2.2.xx/h1.2.xx to address limitations when compiling Logic Test Patterns.

4.3.1.2 APFP Dialog

See Adding a New Pattern File to the Project, Automated Pattern File Processing (APFP).

The APFP dialog is invoked using the  button, added to Visual Studio when *UseRel* is executed. See APFP Button Missing if this button is not automatically displayed.

The APFP Dialog provides access to various controls and utilities which affect the operation of test pattern compilation. The dialog display changes slightly, depending on which software release is being used, to indicate the target hardware platform (Maverick-I/-II vs. Magnum 1, 2, 2x, etc.). This also reflects the contents of the generated test pattern *.cpp* files, which are different for each hardware platform.

The following image shows the APFP dialog displayed when using a Magnum 1 software release.

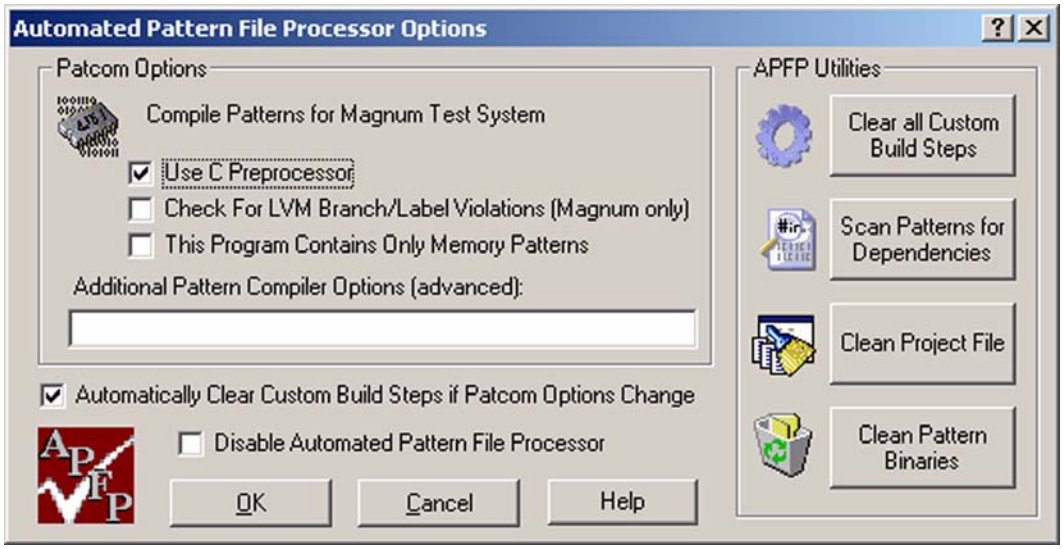


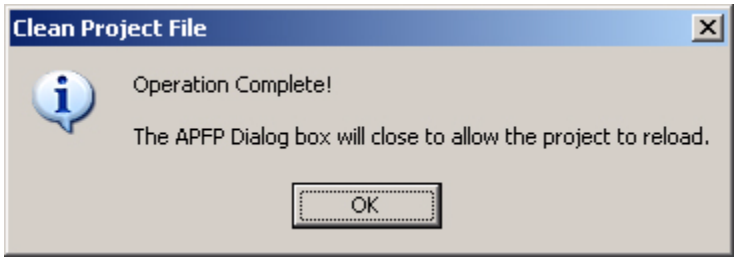
Figure-64: APFP Dialog for Magnum 1

The APFP dialog includes the following controls, in three groups:

- Patcom Options. Affects test pattern compile options. Changes may invoke one or more APFP Utilities.
- APFP Utilities.
- Misc Controls (controls not part of the two previous groups).

Patcom Options

Compile Patterns For :Magnum Test System: indicates that the *.cpp* files generated by Patcom are for Magnum 1 test systems (only). If the selections are changed and the dialog terminated using the **OK** button, the Clean Project File operation is automatically performed, which causes Visual Studio to present the following dialog and reload the project from disk:



Click **Yes**, to complete the process.

- **Use C Preprocessor** : causes the C preprocessor to be used to evaluate `#define` (etc.) statements in test pattern files. If this option is not enabled, the preprocessor facility built into the pattern compiler is used, which is limited in the types of statements which can be used. Note that enabling this feature may cause fatal compiler problems when compiling large Logic Test Patterns (contact Nextest Applications).
- **Check for LVM Branch/Label** : only applies when a Magnum 1/2/2x Logic Test Pattern contains both Pattern Labels and branch instructions. This enables additional rule checks (rarely needed) related to the use of Pattern Labels relative to logic pattern instructions. See LVM Branch/Label Limitations.
- **This Program Contains Only Memory Patterns**: select this when the test program does not contain any Logic, Mixed, or MixedSync test patterns. This allows the pattern compile process to skip several steps, reducing the time necessary to compile the test program and patterns. See Patcom Options. Note that if this option is enabled and the program does contain a logic pattern which contains a VECDEF directive the pattern compiler will generate an error similar to the following:

  ```
  pat.pat(2): error: VECDEF but no pin assignments (-p option
    missing?)
  ```

- **Additional Pattern Compiler Command-line Options**: for advanced applications where pattern compile options are managed explicitly. This is rarely used.

APFP Utilities

These APFP dialog buttons invoke specialized utility functions:

- **Clear all Custom Build Steps** . *MUST BE* manually invoked, by the user, any time any test pattern in the project has had an external dependency file (`#include` statement) added to, modified, or deleted from the pattern. This is the only method which clears the prior record of these dependencies. In most situations, the Scan Patterns for Dependencies must be invoked next, to reestablish any remaining external dependencies. This utility is automatically invoked if any options in the Patcom Options are changed and **OK** is selected.
- **Scan Patterns for Dependencies** . Causes all test patterns (*.pat* files) in the project to be scanned to identify external dependencies; i.e. `#include` statements. Then, a custom build step will be added to any *.pat* file which contains a `#include` statement. Note that this action *must be* manually performed, by the user, any time a `#include` statement is added to, modified, or deleted from a test pattern file in the project. The user should consider invoking this utility after using

the Clear all Custom Build Steps utility (which is used to delete custom build steps added previously). This utility is automatically invoked if the target tester type (for example, Maverick1 vs. Magnum2) is changed in the Patcom Options section of the dialog are changed and **OK** is selected.

- **Clean Project File** . Deletes the test pattern *.cpp* and *.h* files from the test program project. Does not remove test pattern *.pat* files or delete any files from disk. Only the files for the currently selected test system type are affected. This utility is automatically invoked if any options in the **Patcom Options** section of the dialog are changed and **OK** is selected.
- **Clean Pattern Binaries** . Deletes from the disk, the files and folders generated by the pattern compiler. Only the files generated for the selected test system type are affected.

Misc Controls

These APFP controls are not part of the two previous groups:

- **Automatically Clear Custom Build Steps if Patcom Options Change**: default = enabled. Note: only advanced users should disable this. When enabled (highly recommended) causes APFP to update the custom build steps for ALL test patterns when any changes are made to selections in the **Patcom Options** area of the dialog. If disabled, and a Patcom option is changed, the change will not be reflected in any existing custom build steps and a warning will be displayed when the dialog is closed (**OK** button). Any new patterns added to the project after the **Patcom Options** are changed will get the new build settings.
- **Disable Automated Pattern File Processor**: when checked, disables the APFP facility but does not clear any existing custom build steps previously added. This is for users who compile test patterns using user defined methods. Even though this option is checked, if invoked, the Clear all Custom Build Steps utility will perform the actions described. Note that the other APFP Utilities continue to operate as documented. To completely disable APFP and it's effects do the following:
  - Click the **Disable Automated Pattern File Processor** and ensure the option is checked.
  - Click Clear all Custom Build Steps.
  - Click Clean Project File. Note this may prevent pattern compiling until the user defines their own custom built steps for *.pat* files and adds the necessary pattern *.cpp* files to the project.

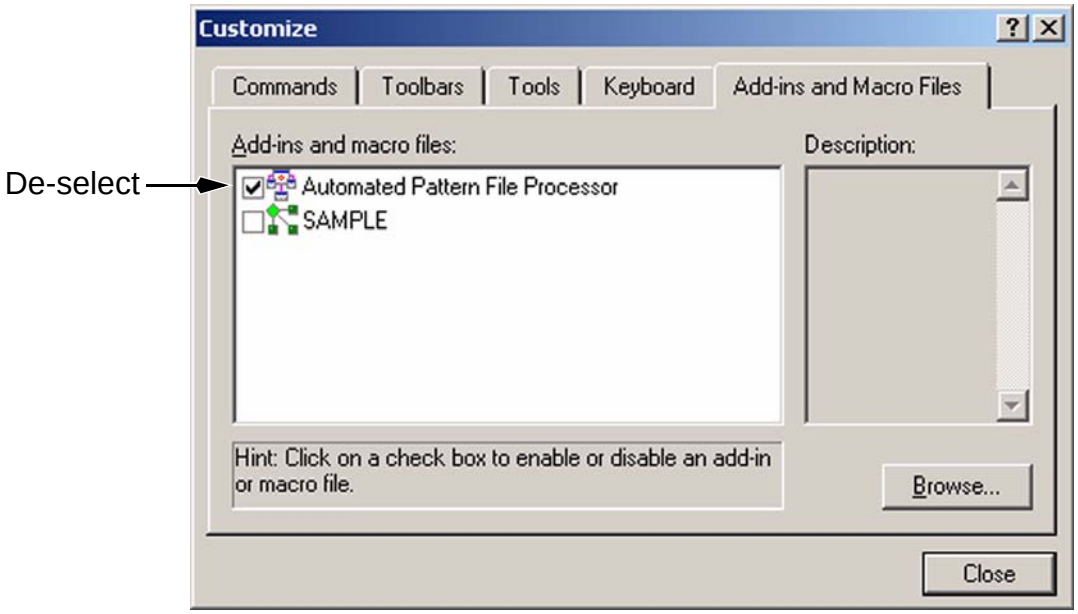
The minimal application of APFP would Scan Patterns for Dependencies then disable APFP (Disable Automated Pattern File Processor). Scanning will insert the appropriate custom build steps and add the pattern *.cpp* files to the project. Then, with APFP disabled, if any pattern stub files are created the 2nd pass build must be invoked/controlled using methods developed by the user because APFP normally does this.

APFP Button Missing

See Automated Pattern File Processing (APFP).

The following procedure describes how to display the button which invokes the APFP Dialog. This will be necessary when the button is terminated by clicking the **X** in the upper-right corner of the button, which is only possible when the APFP dialog button is not docked to a Visual Studio tool-bar:

1. In Visual Studio select **Tools->Customize...**, to display the following dialog:



2. In the *Customize* dialog, de-select the **Automated Pattern File Processor** option and click **Close**.
3. Open this same dialog again, select the **Automated Pattern File Processor** option again and click **Close** again.

4.3.1.3 Build (Compile) Operation

See Adding a New Pattern File to the Project, Automated Pattern File Processing (APFP).

The information below reflects the re-implementation of the Automated Pattern File Processing (APFP) which first shipped in software release h2.2.xx/h1.2.xx.

As compared to the first-generation APFP (see Note:), the test pattern/program build (compile) sequence has changed, generating a different series of compile-time messages and changing how errors and warnings are handled. A one-pass or two-pass build sequence may be used, depending on two options; there are four build scenarios based on whether the `This Program Contains Only Memory Patterns` option is enabled in the APFP Dialog and whether an incremental build is invoked (i.e. whether **Build->Build (F7)**) or **Build->Rebuild All** is invoked:

- Memory Patterns Only: Build->Rebuild All
- Memory Patterns Only: Build->Build (F7)
- Program Contains a Logic Pattern: Build->Rebuild All
- Program Contains a Logic Pattern: Build->Build (F7)

Table 4.3.1.3-1 Memory Patterns Only: Build->Rebuild All

Pass	Step	Performed By	Description
n/a	1	Visual Studio	**Rebuild->Rebuild All** invokes **Build->Clean** which deletes C-compiler-generated files (*.exe*, *.obj*, etc.) and the test pattern *.cpp* files generated by a previous pattern compile, if any.

Table 4.3.1.3-1 Memory Patterns Only: Build->Rebuild All *(Continued)*

Pass	Step	Performed By	Description
1	2	APFP	If no *.pat* files in project, skip to step 7.
	3		The *APFP.ini* file is read or created. The APFP Dialog may be displayed, prompting for options.
	4		Test pattern *.cpp* files are added to the project.
	5		Custom Build Steps for each *.pat* file are added to the project.
	6	Visual Studio	Patcom fully compiles all test pattern (*.pat*) files to generate the corresponding *.cpp* file.
	7		All *.cpp* files are compiled.
	8		Link and generate test program *.exe.*
No pass-2 is required.			

Table 4.3.1.3-2 Memory Patterns Only: Build->Build (F7)

Pass	Step	Performed By	Description
1	1	APFP	If no *.pat* files in project, skip to step 4.
	2		Custom Build Steps for any newly added *.pat* file are added to the project.
	3	Visual Studio	Patcom compiles any pattern *.pat* files which have changed since the last build and generates the corresponding *.cpp* file.
	4		Any *.cpp* files which have changed since last build are compiled.
	5		Link and generate test program *.exe.*
Note that this operation presumes that only incremental changes were made to a test program previously compiled with APFP (custom build steps were previously added, etc.). No pass-2 is required.			

Table 4.3.1.3-3 Program Contains a Logic Pattern: Build->Rebuild All

Pass	Step	Performed By	Description
n/a	1	Visual Studio	**`Rebuild->Rebuild All`** invokes **`Build->Clean`** which deletes C-compiler-generated files (*.exe*, *.obj*, etc.) and the test pattern *.cpp* files generated by a previous pattern compile, if any.
1	2	APFP	The *APFP.ini* file is read or created. The APFP Dialog may be displayed, prompting for options.
	3		Test pattern *.cpp* files are added to the project.
	4		Custom Build Steps for each *.pat* file are added to the project.
	5	Visual Studio	Since the *testprog.exe* file was deleted, Patcom compiles all patterns in *stub mode*, which generates stub *.cpp* files set to an older date than the *.pat* files (see below).
	6		All *.cpp* files are compiled, including stub *.cpp* files.
	7		Link and generate test program *.exe.*
	8	APFP	IF pass-1 generated any ERRORs, stop.
2	9		Since all pattern *.cpp* files are now older than their *.pat* files a pass-2 build is required.
	10		IF pass-1 generated any WARNINGs, display APFP Warning Dialog and Stop if user's response = Stop.
	11	Visual Studio	Patcom fully compiles all pattern *.pat* files, generates new *.cpp* files.
	12		Visual Studio builds all pattern *.cpp* files (other *.cpp* files were built in pass-1).
	13		Link and generate test program *.exe.*

Table 4.3.1.3-4 Program Contains a Logic Pattern: Build->Build (F7)

Pass	Step	Performed By	Description
1	1	APFP	Custom Build Steps for any newly added *.pat* file are added to the project.
	2	Visual Studio	Patcom compiles any pattern *.pat* files which have changed since last build and generates the corresponding *.cpp* file.
	3		All *.cpp* files which have changed since last build are compiled.
	4		Link and generate test program *.exe* file.
	5	APFP	IF pass-1 generated any ERRORs, stop.
2	6		If the Pin Assignment Table has not changed since the last build a pass-2 build is not needed, stop (done).
	7		Otherwise, IF pass-1 generated any WARNINGs, display APFP Warning Dialog and Stop if user's response = stop.
	8	Visual Studio	Because the Pin Assignment Table changed, Patcom fully compiles all pattern *.pat* files, generates *.cpp* files.
	9		Visual Studio compiles all pattern *.cpp* files (other *.cpp* files were built in pass-1 if necessary).
	10		Link and generate test program *.exe* file.
Note that operation presumes that only incremental changes were made to a test program previously compiled with APFP (custom build steps were previously added, etc.).			

When the test program contains logic patterns it is necessary to obtain information from the Pin Assignment Table, to allow the pattern compiler to resolve VECDEF Compiler Directive declarations. This information becomes available after the pass-1 build completes with no errors. Fatal errors must be corrected before the build process can proceed, however, warnings are not fatal and may be ignored (not recommended). Since warnings generated

during pass-1 are cleared once pass-2 begins the following dialog is displayed, allowing the user to decide whether to terminate the build, to address the warnings, or to continue:

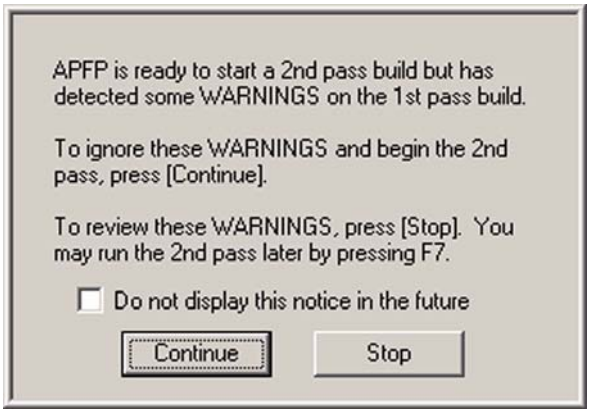


Figure-65: APFP Warning Dialog

Note that if **`Continue`** is selected the pass-1 warnings will not be seen again until a full build is performed, using **`Build->Rebuild All`**.

Note: APFP expects that the file name of test program executable file (i.e. .../Debug/myProg.exe) matches the file name of the Visual Studio project file (i.e. .../myProg.dsp). If these file names do not match, APFP execution will enter an endless loop, continuously compiling test patterns in stub mode.

Note: the Visual Studio's build facility does not detect when folders created by the pattern compiler are deleted from disk (which some users do routinely when archiving a test program). When a given test pattern file (*.pat) contains logic/scan patterns the folder created by the pattern compiler contains files which must be present in order for the patterns to be usable in the hardware. When these folders are deleted but the corresponding *.cpp file still exists the project build will succeed, with no errors, but the pattern won't load because the required folder contents are gone. To recover, use APFP's Clean Pattern Binaries button before compiling.

4.3.1.4 APFP Migrating from Older Versions

See Adding a New Pattern File to the Project, Automated Pattern File Processing (APFP).

This section covers the following topics:

Migrating Programs to h2.3.xx/h1.3.xx

The following procedure assumes that the user is migrating a test program from a Nextest software release prior to h2.3.xx/h1.3.xx which used the first-generation APFP implementation AND that APFP was actually used to control test pattern compilation (the normal case). For those [rare] situations in which the user manually controlled pattern compiling, using manually edited Custom Build Settings, that process does not change.

The second-generation APFP implementation consists of three components:

- A new *APFP.dll*, which replaces the previous version (this file is part of the software release).
- Custom build settings for each test pattern source file (*.pat* file). These are automatically added to each test pattern *.pat* file by the APFP facility when the Scan Patterns for Dependencies button is clicked.
- The APFP Dialog and button, which replaces the first-generation four-button tool-bar.

The following steps document how to migrate to use the second-generation APFP:

1. Close Visual Studio.
2. Install Nextest software revision h2.3.xx/h1.3.xx or later.
3. Execute *UseRel* from this software release.
4. Start Visual Studio and confirm the new APFP button is displayed, if not refer to APFP Button Missing. The button will initially be un-docked but can be dragged to dock to a Visual Studio tool-bar. Note that unlike the previous four-button tool-bar this button will remain docked (until *UseRel* is executed again to change to a different software release). The images below show how the button appears in both situations:

    Un-docked

    Docked

   Note: if the un-docked button is terminated, by clicking the **X** in the upper-right corner, it can be re-displayed using the procedure documented in APFP Button Missing.

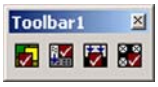
5. For each test program being migrated to h2.3.xx/h1.3.xx or later open the project in Visual Studio.

- If the project contains the *Pattern Processor* sub-project deleted this sub-project. Also delete the *Pattern Processor* folder from the disk (if it exists).
- Display the APFP Dialog and ensure the desired options are selected, in particular make sure the target tester type is correct; i.e. Compile Patterns For Maverick vs. Magnum 1.
- If any test patterns include `#include` statements, in the APFP Dialog click on the Clear all Custom Build Steps utility button. It is safe to always do this provided Scan Patterns for Dependencies is invoked as directed below.
- In the APFP Dialog click on the Clean Project File utility button.
- In the APFP Dialog click on the Scan Patterns for Dependencies utility button. This adds the required custom build steps to each *.pat* file in the project.
- Build the project using **Build->Rebuild All**.

Switching Back to Software Prior to h2.3.xx/h1.3.xx

Once Nextest software release h2.3.xx/h1.3.xx or later has been used, any test patterns compiled in that release will have custom build steps added by APFP. These are not compatible with older software releases and must be removed from every test pattern in every program to be migrated backwards. This can be done manually (not documented), but the controls in the APFP Dialog available in the software release h2.3.xx/h1.3.xx or later reduces both the complexity and potential for errors. Use the following procedure. Note that this procedure assumes that the target earlier release contains the first-generation APFP which contains the four-button tool-bar:

1. Before switching to the older software release (before executing *UseRel* of the older release) identify those test programs which are to be migrated backwards. Then for each of these programs:
   - Open the project in Visual Studio.
   - In the APFP Dialog click the Clear all Custom Build Steps utility button.
   - In the APFP Dialog click the Clean Project File utility button.
   - In the APFP Dialog click the Clean Pattern Binaries utility button.
   - Terminate the APFP Dialog using the **OK** button.
   - In Visual Studio invoke **File->Save All**.
   - In Visual Studio invoke **File->Save Workspace**.
   - Close the project workspace: **File->Close Workspace**.

2. The target older software release DOES require access to files in the h2.3.xx/h1.3.xx or later software release. Do NOT delete this newer release until the switch to the older release has been completed and confirmed as noted below.
3. Close Visual Studio.
4. Execute *UseRel* from the older Nextest software (prior to h2.3.xx/h1.3.xx).
5. Start Visual Studio and confirm that the first-generation APFP four-button tool-bar appears: 

. This confirms that the switch to the APFP methods used in the older release was successful.
6. For each of the programs being migrated backwards:
   - Open the project in Visual Studio.
   - In the four-button APFP tool-bar, click the left-most button to display the APFP Options dialog. Confirm that the desired target system type (Maverick vs. Magnum) is selected. Click **OK**.
   - Build the project.

4.4 Use of #include pattern.h file(s)

See Test Pattern Programming.

The general requirement is that an external declaration of each referenced test pattern must exist in any C-code files which reference the test pattern. For example:

```
extern Pattern *patname;
funtest( patname, finish );
```

Two common methods are used:

- A `#include` statement of a *.h* file which contains one or more `extern Pattern*` declarations. For example, the test program created by the test program wizard contains the *patterns.h* file, which is `#include` into the *tester.h* file, which is itself included in all other test program files. This works correctly in most applications.
- It is also possible to be more selective, to `#include` the *pattern.h* file(s) only in the source file(s) which reference the external test pattern. This can cause a small improvement in compile time.

4.5 Pattern Files and Folders/Directories

See Test Pattern Programming.

Table 4.5.0.0-1 Pattern Files and Directories

Name	Comments
xxx.pat	Test pattern source file, where xxx is user-defined. File name must use the extension .pat, and must reside in the same folder (directory) as the other test program source files (.cpp files). The file can contain one or more test patterns.
xxx_MAG**.cpp**	Pattern C source file, created by the Patcom compiler. **Do NOT edit.** Edits are not useful and will be lost when the pattern is next compiled.
xxx_MAG**/***	Folder containing various pattern compiler output files. Depending on the features used in the pattern being compiled, this folder will have different contents. See Pattern Sub-directory Contents. **Do NOT edit**. Edits are not useful and will be lost when the pattern is next compiled. May be deleted - recompiling the test pattern will recreate the folder and contents.
xxx_MAG**.h**	Pattern external declarations. Created when the pattern is compiled. **Do NOT edit**. Edits are not useful and will be lost when the pattern is next compiled.
test_program.exe	Memory Test Patterns are combined with executable C Code when creating the test program executable.

4.5.1 Pattern Sub-directory Contents

See Pattern Files and Folders/Directories.

Given a pattern file named **Example.pat** compiling will create a folder named **Example_MAG/**. Then, depending on the type of vectors found in **Example.pat** this folder may contain:

- **.svc* containing any scan pattern load information.
- **.lvc* files, one for each pin assignment table in the test program, containing logic vector pattern load information
- Files containing file information used in the build process.

4.6 Compiling Test Patterns

See Test Pattern Programming.

Assuming that the test pattern build settings are properly set up (see Adding a New Pattern File to the Project), pattern compiling will occur automatically, as part of compiling the test program.

The pattern compiler is named Patcom, which is not normally important to the user - Patcom is automatically controlled by the test program compile process. The information below is to help understand the compile process.

When the test pattern does not use `VECDEF` or `SCANDEF` directives pattern compiling is a two step process:

- Patcom reads and processes each **`.pat`** file and creates a corresponding *.cpp* for each **`.pat`** file processed.
- The C compiler compiles the *.cpp* file, along with the other program *.cpp* files, to create the binary pattern file loaded into the test hardware.

A sub-folder (see Pattern Sub-directory Contents) and an external declarations file (*xxx.h*) are also created. Do ***NOT*** edit these created files; edits are not useful and will be lost when the pattern is next compiled.

When the test pattern *does* use `VECDEF` or `SCANDEF` directives pattern compiling is more complex, but still occurs as part of compiling the rest of the test program files:

- Any *.cpp* file which contains any of the following macros is compiled using the C compiler: `PIN_ASSIGNMENTS()`, `PIN_SCRAMBLE()`, `DUT_PIN`. This is done before any pattern files are processed.

Note: PIN_ASSIGNMENTS() and PIN_SCRAMBLE() definitions (code) should be located in source files which contain little other code. The files created by the test program Wizard do this; the user should follow this model when test patterns contain VECDEF or SCANDEF compiler directives.

- Patcom then reads and processes each **.pat** file and creates a corresponding *.cpp* file for each **.pat** file processed. If a given **.pat** file uses VECDEF or SCANDEF the pin assignments and pin scramble *.cpp* file(s) compiled in the previous step are executed to obtain information required to resolve the VECDEF and/or SCANDEF directives.
- The C compiler compiles the *.cpp* file to create the binary pattern file loaded into the test hardware.

4.7 Pattern Loading

See Test Pattern Programming.

By default, compiled test pattern files are loaded automatically as the test program loads. All test patterns compiled into the program will be loaded. See Program Loading and Execution Order.

Optionally, if Pattern Sets are used, the currently defined *pattern set* controls which memory patterns are loaded. And, defining a new pattern set causes new memory patterns to load, over-writing any previously loaded patterns. See Pattern Sets.

As test pattern files are loaded they automatically go through a relocation process. Neither the test program nor the test patterns explicitly specify pattern load addresses. All programmatic references to a test pattern are made via the name specified in the PATTERN statement or to specific instructions identified using a Pattern Label or Pattern Label + offset.

As each test pattern is loaded, any Pattern Initial Conditions instructions are saved in computer memory, and linked to the test pattern using a software pointer. Then, when the pattern is executed (funtest(), ac_partest(), ac_test_supply(), etc.) the system software executes the initial conditions code before starting the pattern generator. Note that the initial conditions are only executed for the main pattern; i.e initial conditions defined in Pattern Subroutiness are not executed.

4.7.1 Pattern Load PATH

See Pattern Loading.

Test patterns are loaded automatically along with the test program.

Memory Test Patterns are compiled into the test program executable file and always load automatically. Logic Test Patterns and Scan Test Patterns are treated differently. Since Logic Test Patterns and Scan Test Patterns are treated the same the remainder of this section will refer to logic patterns only.

For each logic pattern file the pattern compiler generates a sub-directory (folder), containing several files. The directory is named after the pattern source file. For example, given the source file:

```
D:\myPath\myProg\myPat1.pat
```

the pattern compiler will create a directory named:

```
D:\myPath\myProg\myPat1_MAG\
```

and store the compiler generated files in that directory. Note that the compiler always puts these sub-directories in the same location as the pattern source file. The information below anticipates that the user may subsequently move these directories to a different location.

The pattern loader looks for each compiled Logic Test Pattern in the directory of the same name as the pattern source file, in the locations noted below, in the order listed. The last option is the default location:

1. The current directory; i.e. the location of the test program *.exe* file. Using the example above:

   ```
   D:\myPath\myProg\Debug\myPat1_MAG\
   ```

2. In each location defined by the PATTERN_PATH environment variable. The syntax and format of PATTERN_PATH is the same format as the Window's PATH environment variable. PATTERN_PATH can be set in a variety of ways, see Environmental Variables. For example, if the PATTERN_PATH contains the following:

   ```
   D:\somePath
   ```

   The logic pattern could be located at:

   ```
   D:\somePath\myPat1_MAG\
   ```

3. In a sub-directory named after the program executable and located at the same place as the program executable. For example, if the program executable is located at:

```
...anywhere\myProg.exe
```

The patterns for the program can be located at:

```
...anywhere\myProg\myPat1_MAG\
```

This allows multiple test programs to reside in one location, with each program having a separate sub-directory for its patterns, with that the sub-directory named after the program executable.

4. In the default pattern directory. Using the example above:

```
D:\myPath\myProg\Debug\..\myPat1_MAG\
```

This is the normal Developer Studio default case.

4.7.2 Pattern Sets

See Pattern Loading.

Pattern sets were created to provide programmatic control over which Memory Test Patterns are loaded into APG uRAM. This control is provided to allow a given test program to contain more patterns than will fit into 64K uRAM at one time. Using pattern sets, the test program can define and compile more test patterns than will physically fit into APG uRAM and select which ones are loaded during program execution.

Note the following:

- All Memory Test Patterns, Logic Test Patterns, Scan Test Patterns and Mixed Memory/Logic Patterns to be executed in a given test program are compiled along with the test program. This is done automatically presuming that the methods documented in Adding a New Pattern File to the Project are followed. This is a requirement regardless whether or not a given test pattern is subsequently loaded into the hardware.
- When a test program is loaded into the system hardware, by default, all of its test patterns are also loaded. The pattern sets methods documented below may be used to explicitly select which Memory Test Patterns are loaded into the APG's uRAM.
- All Logic Test Patterns in the program are always loaded into Logic Vector Memory (LVM). However, if pattern sets are used any logic patterns to be executed must be included in the pattern set definition too.

Note: when a pattern set includes a logic test pattern from a given *.pat* file then all patterns in that *.pat* file must be included in the pattern set. This is required even when some patterns in the file are not subsequently used or executed. Proper pattern operation may not occur if this rule is violated.

- When pattern sets are used, any test patterns which are not included in a given pattern set cannot be executed. Any attempt to execute a pattern which is not currently loaded will generate a *warning* message and the pattern execution will FAIL.
- User-code can determine if a specified pattern is currently loaded using:

```
BOOL loaded = addrs( myPat, 0 ); // See addrs()
```

The following rules apply:

- Only one pattern set can be loaded at a time, using the `load()` function.
- Executing the `load()` function first clears all previously loaded memory patterns (except the `builtin_xxx` patterns) from the APG's uRAM. It then loads the uRAM with the patterns in the specified pattern set.
- Executing `load()` also clears the list of loaded logic patterns (but they remain loaded). Thus, any logic patterns to be executed must be included in the pattern set definition.
- Using the Resource Find Functions and specifying `S_Pattern` will only display patterns that are currently loaded.
- If user code executes `resource_deallocate()` with `S_Pattern` all patterns known to the test program become visible and a new pattern set must be loaded before executing any functional tests.
- Given the name of a pattern set, the `PatternSet_find()` function can be used to get a pointer to that pattern set.

Note: as noted above, any attempt to use a pattern not currently loaded will generate a runtime *warning* and the pattern execution will FAIL.

Usage

```
PATTERN_SET( ps_name )
ADD_PATTERN( pattern_name )
INCLUDE_PATTERN_SET( ps_name )
```

```
EXTERN_PATTERN_SET( ps_name )
BOOL load( PatternSet *obj );
void add( PatternSet *obj, Pattern *pattern );
```

where:

PATTERN_SET is a Test System Macro used to create a new pattern set. It can only be called globally; i.e. not part of a function or other macro body-code. This creates an `S_PatternSet` resource.

ps_name is the name of the pattern set being created. This becomes a `PatternSet*`, suitable for use as the `obj` argument to `load()` and `add()`.

ADD_PATTERN is a Test System Macro used to add a pattern to a pattern set. It can only be called from within the **PATTERN_SET** macro body code.

pattern_name is the name used in the test pattern's `PATTERN` statement.

INCLUDE_PATTERN_SET is a Test System Macro which allows a previously defined pattern set to be used as a component of a new pattern set. It can only be called from within the **PATTERN_SET** macro body code.

EXTERN_PATTERN_SET is a Test System Macro used to make an external pattern set declaration. It can only be called globally; i.e. not part of a function or other macro.

obj is a pointer to a pattern set. As used by `load()`, it specifies the pattern set to load. As used by `add()` it specifies the pattern set which is being modified.

pattern is a pointer to the pattern being added to **obj**.

Example

This example is somewhat large, and includes code examples from several source files. It demonstrates how pattern sets are created, loaded, and used. Intentional errors are also demonstrated. The information below includes (in order):

- Pattern Sets Code: use the various methods to create pattern sets plus two print routines. *Note: one of the print routines changes the currently loaded pattern set!*
- Sequence and Binning Table Code: controls how example test blocks execute and thus the order of Runtime Output Messages.
- Test Block Code: load pattern sets and execute tests, some of which intentionally generate warning messages. Also calls the print routines created in Pattern Sets Code.
- The Runtime Output Messages seen in UI's SITE output window.

Additional key information is noted for each of these topics below.

Pattern Sets Code

Note the following:

- The various pattern set related macros are used to create several pattern sets.
- One pattern set, `PSet3`, contains no patterns. This has the effect of unloading any currently loaded memory patterns.
- One pattern set, `PSetAdd`, is dynamically created using the `add()` function.
- At the end, two print functions are included which are called from the example Test Block Code. Note that `PatternSetAll_print()` sequentially loads every pattern set, and *leaves the last one loaded*.

```
#include "tester.h"

PATTERN_SET( PSet1 ) {
   output ("Creating pattern set => PSet1");
   ADD_PATTERN( P1_1 )
   ADD_PATTERN( P1_2 )
   ADD_PATTERN( P1_3 )
}

PATTERN_SET( PSet2 ) {
   output ("Creating pattern set => PSet2");
   ADD_PATTERN( P2_1 )
   ADD_PATTERN( P2_2 )
   ADD_PATTERN( P2_3 )
}

PATTERN_SET( PSet3 ) {
   output ("Creating pattern set => PSet3 = no patterns");
}

PATTERN_SET( PSetAll ) {
   output ("Creating pattern set => PSetAll");
   INCLUDE_PATTERN_SET( PSet1 )
   INCLUDE_PATTERN_SET( PSet2 )
   INCLUDE_PATTERN_SET( PSet3 )
   ADD_PATTERN( P4_1)
   ADD_PATTERN( P4_2)
   ADD_PATTERN( P4_3)
}
```

```
PATTERN_SET( PSetAdd ) {
   output ("Creating pattern set => PSetAdd");
   // Make all patterns visible. Also clears user patterns from uRAM
   resource_deallocate ( S_Pattern ) ;
   // Define the pattern set. Only include all patterns that contain
   // '2' in the name
   CStringArray names;
   int count = resource_all_names ( S_Pattern, &names );
   for( int i = 0; i < count; ++i ){
      output("  Adding pattern => %s", names[ i ]);
      if ( names[ i ].Find ( '2' ) >= 0 )
         add ( PSetAdd, Pattern_find ( names[ i ] ) );
   }
}
//=============================================================
// The following user-code will print only the Patterns in the
// currently loaded pattern set
void PatternLoad_print( void ) {
   CStringArray pat_names;
   int count = resource_all_names( S_Pattern, &pat_names );
   output("\nPatternLoad_print");
   output("Name  MAR Length");
   output("----- --- ------");
   for ( int i = 0; i < count; ++i ) {
      DWORD mar, var, mlen;
      addrs( Pattern_find( pat_names[ i ] ), &mar, &var, &mlen );
      output("  %s: %d  %d", pat_names[ i ], mar, mlen );
   }
}
// The following code prints all of the patterns in all
// PATTERN_SETs. Note:during this process, each pattern set is
// sequentially loaded, and the last one  loaded remains loaded  !!!
void PatternSetAll_print( void ){
   CStringArray set_names;
   int count = resource_all_names( S_PatternSet, &set_names );
   output("\nPatternSetAll_print");
   output("Name  MAR Length");
   output("----- --- ------");
   for ( int i = 0; i < count; ++i ) {
```

```
      output ("Loading Pattern Set => %s", set_names[ i ]);
      load( PatternSet_find( set_names[ i ] ) );
      CStringArray pat_names;
      int count1 = resource_all_names( S_Pattern, &pat_names );
      for ( int j = 0; j < count1; ++j ) {
         DWORD mar, var, mlen;
         addrs( Pattern_find( pat_names[ j ] ), &mar, &var, &mlen );
         output("  %s: %d  %d", pat_names[ j ], mar, mlen );
      }
   }
}
```

Sequence and Binning Table Code

This code controls the order the test blocks execute, and thus the order of Runtime Output Messages below.

```
#include "tester.h"
SEQUENCE_TABLE( SeqTab1 ) {
   SEQUENCE_TABLE_INIT
      TEST( TB0,        NEXT, STOP)
      TEST( TBPSet1,    NEXT, STOP)
      TEST( TBPSet2,    NEXT, STOP)
      TEST( TBPSet3,    NEXT, STOP)
      TEST( TBPSetAll,  STOP, STOP)
      TEST( TBPSetAdd,  STOP, STOP)
}
```

Test Block Code

This code has several features warranting special notice:

- For convenience, the test block names are based on the pattern set names used in each test block.
- The `PATTERN_SET()` macro is used to advise the compiler that the associated pattern sets are external; i.e. in another source file. These statements could be moved to the *tester.h* file and deleted here.
- Test block TB0 calls two user-written print routines included in the Pattern Sets Code above (`PatternLoad_print()`, and `PatternSetAll_print()`). Note that `PatternSetAll_print()` *changes the currently loaded pattern set*.

- The `load()` function is used to load a specific pattern set in several test blocks. Then `PatternLoad_print()` is called to print a list of the patterns in the pattern set.
- In both test block `TBPset1` and `TBPset2` the `funtest()` function is called twice; once with a pattern argument which is not currently loaded. This generates the warning messages seen in the Runtime Output Messages.

```
#include "tester.h"
EXTERN_PATTERN_SET( PSet1 )
EXTERN_PATTERN_SET( PSet2 )
EXTERN_PATTERN_SET( PSet3 )
EXTERN_PATTERN_SET( PSetAll )
EXTERN_PATTERN_SET( PSetAdd )

TEST_BLOCK( TB0 ) {
   output("\n================================================");
   output(" Executing Test Block => TB0");
   PatternSetAll_print();
   PatternLoad_print();
   return PASS;
}

TEST_BLOCK( TBPSet1 ) {
   output("\n================================================");
   output(" Executing Test Block => TBPSet1 ");
   load ( PSet1 );
   PatternLoad_print();
   // Next funtest() generates an error message
   output(" Executing Pattern: P2_1 => \\");
   if ( funtest( P2_1, error ) == FALSE)
      output (" FAILED");
   else output (" Passed");
   output(" Executing Pattern: P1_1 => \\");
   if ( funtest( P1_1, error ) == FALSE)
      output (" FAILED");
   else output (" Passed");
   return PASS;
}

TEST_BLOCK( TBPSet2 ) {
   output("\n================================================");
   output(" Executing Test Block => TBPSet2 ");
   load ( PSet2 );
```

```
   PatternLoad_print();
   output(" Executing Pattern: P2_1 => \\");
   if ( funtest( P2_1, error ) == FALSE)
       output (" FAILED");
   else output (" Passed");
   // Next funtest() generates an error message
   output(" Executing Pattern: P1_1 => \\");
   if ( funtest( P1_1, error ) == FALSE)
       output (" FAILED");
   else output (" Passed");
   return PASS;
}
TEST_BLOCK( TBPSet3 ) {
   output("\n================================================");
   output(" Executing Test Block => TBPSet3");
   load ( PSet3 );
   PatternLoad_print();
   return PASS;
}
TEST_BLOCK( TBPSetAll ) {
   output("\n================================================");
   output("Executing Test Block => TBPSetAll");
   load ( PSetAll );
   PatternLoad_print();
   output(" Executing Pattern: P2_1 => \\");
   if ( funtest( P2_1, error ) == FALSE)
       output (" FAILED");
   else output (" Passed");
   output(" Executing Pattern: P1_1 => \\");
   if ( funtest( P1_1, error ) == FALSE)
       output (" FAILED");
   else output (" Passed");
   return PASS;
}
TEST_BLOCK( TBPSetAdd ) {
   output("\n================================================");
   output(" Executing Test Block => TBPSetAdd ");
   load ( PSetAdd );
   PatternLoad_print();
   output(" Executing Pattern: P2_1 => \\");
```

```
    if ( funtest( P2_1, error ) == FALSE)
        output (" FAILED");
    else output (" Passed");
    output(" Executing Pattern: P1_1 => \\");
    if ( funtest( P1_1, error ) == FALSE)
        output (" FAILED");
    else output (" Passed");
    return PASS;
}
```

Runtime Output Messages

Note the following:

- The “Creating...” and “Adding pattern...” messages are generated from Pattern Sets Code.
- Two warning messages are intentionally generated. See introduction to Test Block Code.
- The other output messages are generated by calling the two print routines (`PatternLoad_print()`, and `PatternSetAll_print()`) which are coded in the Pattern Sets Code and called from Test Block Code.
- The output in test block TBPSet3 shows no patterns were loaded.
- The order of messages caused from test block execution is controlled by the Sequence and Binning Table Code. The Sequence & Binning Tablee was executed once.

```
Creating pattern set => PSet1
Creating pattern set => PSet2
Creating pattern set => PSet3
Creating pattern set => PSetAll
Creating pattern set => PSet1
Creating pattern set => PSet2
Creating pattern set => PSet3
Creating pattern set => PSetAdd
  Adding pattern => P2_1
  Adding pattern => P2_2
  Adding pattern => P2_3
  Adding pattern => P1_2
  Adding pattern => P4_2
  Adding pattern => P3_2
The test program is loaded
```

```
TestStarted(1)...
=========================================================
Executing Test Block => TB0
PatternSetAll_print
Name  MAR Length
----- --- ------
Loading Pattern Set => PSet1
  P1_1: 74  1
  P1_2: 75  1
  P1_3: 76  1
Loading Pattern Set => PSet2
  P2_1: 74  1
  P2_2: 75  1
  P2_3: 76  1
Loading Pattern Set => PSet3
Loading Pattern Set => PSetAll
  P2_1: 74  1
  P2_2: 75  1
  P2_3: 76  1
  P1_1: 77  1
  P1_2: 78  1
  P1_3: 79  1
  P4_1: 80  1
  P4_2: 81  1
  P4_3: 82  1
  P3_1: 83  1
  P3_2: 84  1
  P3_3: 85  1
Loading Pattern Set => PSetAdd
  P2_1: 74  1
  P2_2: 75  1
  P2_3: 76  1
  P1_2: 77  1
  P4_2: 78  1
  P3_2: 79  1
PatternLoad_print
Name  MAR Length
----- --- ------
  P2_1: 74  1
  P2_2: 75  1
  P2_3: 76  1
```

```
  P1_2: 77  1
  P4_2: 78  1
  P3_2: 79  1
=====================================================
Executing Test Block => TBPSet1
PatternLoad_print
Name  MAR Length
----- --- ------
  P1_1: 74  1
  P1_2: 75  1
  P1_3: 76  1
Executing Pattern: P2_1 => Warning: Pattern P2_1 has been ignored,
and therefore will not execute properly
Passed
Executing Pattern: P1_1 =>  Passed
=====================================================
Executing Test Block => TBPSet2
PatternLoad_print
Name  MAR Length
----- --- ------
  P2_1: 74  1
  P2_2: 75  1
  P2_3: 76  1
Executing Pattern: P2_1 => Passed
Executing Pattern: P1_1 => Warning: Pattern P1_1 has been ignored,
and therefore will not execute properly
Passed
=====================================================
Executing Test Block => TBPSet3
PatternLoad_print
Name  MAR Length
----- --- ------
=====================================================
Executing Test Block => TBPSetAll
PatternLoad_print
Name  MAR Length
----- --- ------
  P2_1: 74  1
  P2_2: 75  1
  P2_3: 76  1
```

```
  P1_1: 77  1
  P1_2: 78  1
  P1_3: 79  1
  P4_1: 80  1
  P4_2: 81  1
  P4_3: 82  1
  P3_1: 83  1
  P3_2: 84  1
  P3_3: 85  1
Executing Pattern: P2_1 =>  Passed
Executing Pattern: P1_1 =>  Passed
TestDone...bin = builtin_Pass
```

4.8 Pattern Overview and Naming

See Test Pattern Programming.

Test patterns are stored in source files that are separate from the C-code portion of a test program.

The information below applies to Memory Test Patterns, Logic Test Patterns and Mixed Memory/Logic Patterns (Scan Test Patterns are documented separately).

Logic Test Patterns, Memory Test Patterns and Mixed Memory/Logic Patterns have the following main components:

- A `PATTERN( )` statement, which specifies the test pattern's name and optional Pattern Attributes. All references to a given test pattern use the name specified in the PATTERN statement.
- Pattern Initial Conditions (optional).
- Memory pattern instruction(s) and/or logic pattern instructions (test vectors), including Pattern Labels and Comments in Test Patterns.

The `PATTERN` statement identifies the beginning and name of a test pattern or Pattern Subroutine. The pattern statement has the following form:

```
PATTERN( pattern_name, pattern_attributes )
```

where:

pattern_name specifies the pattern name. This is the pattern name (Pattern*) passed to funtest(), ac_partest() or ac_test_supply() and various other Nextest functions which require that a specific pattern be identified.

pattern_attributes are optional, and specify various options. See Pattern Attributes.

If multiple patterns are included in one file, a new PATTERN( ) statement identifies the start of each new pattern. The pattern compiler (Patcom) reads the PATTERN( ) statement and treats everything between it and the next PATTERN( ) statement (or end of file) as one pattern with the specified name.

The PATTERN( ) statement also can be used to define a Pattern Subroutine which appears identical to a stand-alone test pattern.

Details of Pattern Attributes, Pattern Initial Conditions, Memory Test Patterns, and Logic Test Patterns are described separately.

4.8.1 Pattern Attributes

See Pattern Overview and Naming.

Pattern attributes are the mechanism allowing the user to advise the pattern compiler about various pattern options.

Note: the pattern compiler must generate different output for Maverick-I/-II vs. Magnum 1 vs. Magnum 2 vs. Magnum 2x. This is controlled by using the APFP Dialog available via Automated Pattern File Processing (APFP) i.e. not using pattern attributes.

Pattern attributes are optional in all test patterns, but are required to enable various features, some of which are not supported on all Nextest system types, as described below.

There are three pattern attribute types:

- Pattern System Attributes. When compiling patterns for Magnum 1, Magnum 2 or Magnum 2xthe Pattern System Attributes are ignored; i.e. the VAR Engine is always used to execute logic instructions and Double Data Rate (DDR) Mode is solely controlled by the Pattern Rate Attributes.
- Pattern Rate Attributes specify whether the pattern is single data rate (SDR, default) or Double Data Rate (DDR) Mode.

- Pattern Type Attributes specify whether the pattern contains memory pattern instructions only, logic pattern instructions only, or both. Using Magnum 1/2/2x, the `mixedsync` attribute is used in Mixed Memory/Logic Patterns, to ensure lockstep execution. Important rules apply, see Mixed Memory/Logic Patterns.

Pattern attributes may be set using three methods:

- No attributes are specified; i.e. default values are used.
- Setting Attributes Directly, as parameters in the `PATTERN` statement in the pattern source file. See Pattern Overview and Naming.
- Setting Attribute Defaults, using command-line option flags when invoking the pattern compiler (Patcom) from the command line or from a batch file.

When the methods are mixed, the `PATTERN` statement takes precedence.

The tables below summarize the combinations of Pattern System Attributes, Pattern Rate Attributes and Pattern Type Attributes and how they are interpreted by the pattern compiler (Patcom). Each attribute *type* is documented in detail in subsequent sections:

Note: the attributes of a given test pattern and all dependent Pattern Subroutines must be identical. This cannot be enforced by the pattern compiler (Patcom).

Pattern Attributes			Compile		Execution Mode	
System	Type	Rate	Option	Result	System Type	Mode
[none] mav1 mav2	[none]	[none] single	[none] Maverick	OK	Mav1	Mixed, non-DDR
					Mav2	
			Magnum 1		Magnum 1	Memory non-DDR
			Magnum 2		Magnum 2	
			Magnum 2x		Magnum 2x	

Pattern Attributes			Compile		Execution Mode	
System	Type	Rate	Option	Result	System Type	Mode
[none] mav1 mav2	memory	[none] single	[none] Maverick	OK	Mav1	Memory, non-DDR
					Mav2	
			Magnum 1		Magnum 1	
			Magnum 2		Magnum 2	
			Magnum 2x		Magnum 2x	
none mav1	logic	[none] single	[none] Maverick	Error	Mav1	Compile fails
					Mav2	Compile fails
			Magnum 1	OK	Magnum 1	Logic, non-DDR
			Magnum 2		Magnum 2	
			Magnum 2x		Magnum 2x	
mav2	logic	[none] single	[none] Maverick	OK	Mav1	Illegal @ Mav-I
					Mav2	Logic, non-DDR
			Magnum 1		Magnum 1	
			Magnum 2		Magnum 2	
			Magnum 2x		Magnum 2x	

Pattern Attributes			Compile		Execution Mode	
System	Type	Rate	Option	Result	System Type	Mode
none mav1	mixed	[none] single	[none] Maverick	Error	Mav1	Compile fails
					Mav2	Compile fails
			Magnum 1	OK	Magnum 1	Mixed, non-DDR
			Magnum 2		Magnum 2	
			Magnum 2x		Magnum 2x	
mav2	mixed	[none] single	[none] Maverick	OK	Mav1	Illegal @ Mav-I
					Mav2	Mixed, non-DDR
			Magnum 1		Magnum 1	
			Magnum 2		Magnum 2	
			Magnum 2x		Magnum 2x	
none mav1	[none] memory logic mixed	double	[none] Maverick	Error	Mav1	Compile fails
					Mav2	Compile fails
			Magnum 1	OK	Magnum 1	DDR. Memory, Logic or Mixed (based on Type)
			Magnum 2		Magnum 2	
			Magnum 2x		Magnum 2x	

Pattern Attributes			Compile		Execution Mode	
System	Type	Rate	Option	Result	System Type	Mode
mav2	[none] memory logic mixed	double	[none] Maverick	OK	Mav1	Illegal @ Mav-I
					Mav2	DDR. Memory, Logic or Mixed (based on Type).
			Magnum 1		Magnum 1	
			Magnum 2		Magnum 2	
			Magnum 2x		Magnum 2x	
none mav1 mav2	mixedsync	single double	[none] Maverick	Error	Mav1	Compile fails
					Mav2	Compile fails
			Magnum 1	OK	Magnum 1	See Mixed Memory/ Logic Patterns . DDR or non-DDR based on Rate.
			Magnum 2		Magnum 2	
			Magnum 2x		Magnum 2x	

4.8.1.1 Pattern *System* Attributes

See Pattern Overview and Naming, Pattern Attributes.

Pattern *System* attributes specify the system type on which a pattern is intended to be executed. Legal values are:

Table 4.8.1.1-1 Pattern System Attributes

Token	
None	Equivalent to `mav1`
`mav1`	Maverick-I
`mav2`	Maverick-II

Note: when compiling a test pattern for Magnum 1/2/2x (see APFP Dialog) the Pattern System Attributes are ignored. The VAR Engine is always used when the pattern contains logic instructions and Double Data Rate (DDR) Mode is solely controlled by the Pattern Rate Attributes.

Note: the attributes of a given test pattern and all dependent Pattern Subroutines must be identical. This cannot be enforced by the pattern compiler (Patcom).

4.8.1.2 Pattern *Rate* Attributes

The Pattern *Rate* attribute is required to advise the pattern compiler (Patcom) when a test pattern is a Double Data Rate (DDR) Mode pattern . Note the following:

- The Double Data Rate (DDR) Mode rate attribute is used primarily in logic and scan patterns, to advise Patcom that two sets of pattern data are supplied per pattern instruction, one for the DDR A-cycle, one for the DDR B-cycle. Memory patterns do not directly support DDR, however it is possible to use DDR with memory patterns: see DDR Memory Patterns.

The following *Rate* attribute options are supported:

Table 4.8.1.2-1 Pattern Rate Attributes

Attribute	Description
[none]	Same as single

single	Patcom treats the pattern as a *single* data rate pattern (i.e. non-DDR). Each input logic/scan vector contains one token per-DutPin specified in the VECDEF/SCANDEF statement. In Multi-DUT Test Programs, for each input token Patcom generates one output bit for each pin-pair represented by each DutPin specified in the VECDEF statement. In non-Multi-DUT Test Programs Patcom generates one output bit for each DutPin specified in the VECDEF statement. For example:
	`%% VECDEF p1, pin_t, Cs, Clk, p5 ...`
	`PATTERN ( pat_name, single ) // Same as no attribute`
	`% VEC 00001 ... etc ...`

double	Patcom treats the pattern as a Double Data Rate (DDR) Mode pattern. Each input logic/scan vector contains two tokens per-DutPin specified in the VECDEF/SCANDEF statement. In Multi-DUT Test Programs, for each token in the input pattern Patcom generates two output bits for each pin-pair represented by each DutPin specified in the VECDEF/SCANDEF statement (one bit for the DDR A-cycle and one for the B-cycle). In non-Multi-DUT Test Programs Patcom generates two output bits for each DutPin specified in the VECDEF/SCANDEF statement. See DDR Logic Vectors and DDR Scan Vectors. For example:
	`%% VECDEF p1, pin_t, Cs, Clk, p5 ...`
	`PATTERN ( pat_name, double, logic )`
	`% VEC 00001 ... etc ... \`
	`00010 ... etc ...`

Note: the attributes of a given test pattern and all dependent Pattern Subroutines must be identical. This cannot be enforced by the pattern compiler (Patcom).

4.8.1.3 Pattern *Type* Attributes

See Pattern Overview and Naming, Pattern Attributes.

Pattern *Type* attributes are used by the pattern compiler to:

- Determine the memory pattern instructions which are legal for each system type.
- The types of pattern compiler error checks performed.

The following pattern Type attributes are supported:

Table 4.8.1.3-1 Pattern Type Attributes

Token	
None	Equivalent to `memory`
`memory`	More below.
`logic`	
`mixed`	
`mixedsync`	Magnum 1/2/2x only

When no *Type* attribute is specified Patcom considers the pattern to be a `mixed` pattern.

When the `memory` *Type* attribute is set, Patcom treats the pattern as a pure memory pattern; i.e. it does not allow use of the logic pattern or scan pattern instructions. See Magnum 1/2/2x Logic Vector Instructions.

When the `logic` *Type* attribute is set, Patcom treats the pattern as pure logic pattern. Error messages are generated if any memory instructions are detected.

In Magnum 1/2/2x patterns, when the `mixed` *Type* attribute is set, Patcom allows the pattern to use both memory and logic pattern instructions. The VAR Engine controls logic vector execution and the MAR Engine controls memory instruction execution. In a `mixed` pattern any required synchronization between memory pattern and logic pattern execution is entirely the responsibility of the user's test pattern (which can be *very* challenging and difficult to validate or debug). See Mixed Memory/Logic Patterns.

In Magnum 1/2/2x patterns, when the `mixedsync` *Type* attribute is set, Patcom enforces rules which ensure both the memory and logic portions of each pattern instruction execute concurrently (in lockstep). This mode also allows the user to specify default memory and/or

logic instructions, which are applied to any pattern instruction which doesn't explicitly include a memory and/or logic instruction. See Mixed Memory/Logic Patterns.

Note: the attributes of a given test pattern and all dependent Pattern Subroutines must be identical. This cannot be enforced by the pattern compiler (Patcom).

4.8.1.4 Setting Attributes Directly

See Pattern Overview and Naming, Pattern Attributes.

Normally, Pattern Attributes are specified as arguments to the PATTERN statement (see Pattern Overview and Naming). Only one each of the Pattern System Attributes, Pattern Rate Attributes, and Pattern Type Attributes can be specified in a PATTERN statement. The pattern name must be specified first. The other attribute arguments can be listed in any order.

Setting attributes in the PATTERN statement takes precedence over the command line methods documented in Setting Attribute Defaults.

For example:

```
PATTERN ( my_pattern_name, mav2, double, logic )
```

This example sets the mav2 Pattern System Attributes, the double Pattern Rate Attributes, and the logic Pattern Type Attributes.

4.8.1.5 Setting Attribute Defaults

See Pattern Overview and Naming, Pattern Attributes.

Note: using command-line test pattern compilation methods is rarely needed and not documented.

Default pattern attribute values can be specified using Patcom command line option flags.

When used, these defaults apply to all patterns in the source file, or files, listed on the Patcom command line. These flags are overridden by any pattern attributes specified in a PATTERN statement (see Setting Attributes Directly).

Attribute flag values are the same as the options documented in Pattern System Attributes, Pattern Rate Attributes, and Pattern Type Attributes. They are case sensitive when used on the command line, i.e., –MAV2 and –Mav2 are not legal flags.

The attribute value must be prefixed by a dash ('-') character. Multiple flags may be set in a command line, each preceded by a dash, and separated by white space.

For example:

```
patcom –mav2 -logic -single myPatterns.pat
```

4.8.2 Pattern Instruction Identifier (%)

See Pattern Overview and Naming.

Each new test pattern instruction starts with a percent sign (%) token.

The % token acts as a delimiter between pattern instructions to let the pattern compiler detect when a new instruction begins. This is especially important because each pattern instruction can span multiple lines and contain multiple sub-instructions. Thus, a series of pattern generator instructions might look like the following:

```
PATTERN( pattern_name, pattern_attributes )
% First pattern generator instruction
   Sub-instruction
   Sub-instruction
% Second pattern generator instruction
% Third pattern generator instruction
   Sub-instruction
% etc.
```

4.8.3 Comments in Test Patterns

See Pattern Overview and Naming.

Three forms of test pattern comments are supported:

```
/*  Comment  */
```

```
//  Comment
;  Comment
```

Using method 1, the comment text goes between the `/*` and the `*/` and may span multiple lines. Any characters between the `/*` and `*/` are treated as comment text (even carriage returns).

Using method 2, anything on a line to the right of the double slash (`//`) and before a carriage return is considered comment text. If a comment spans multiple lines, *each* line requires a `//`.

Note: the pattern compiler has a known limitation relating to using comments on the same line as a `#define` statement. Until this limitation is corrected DO NOT use a comment on the same line as a `#define`.

Using method 3, anything on a line to the right of the semicolon (`;`) and before a carriage return is considered comment text. If a comment spans multiple lines, *each* line requires a semicolon. This method is included for backward compatibility to legacy test patterns, but is **`discouraged`** when writing new patterns.

4.8.4 Pattern Initial Conditions

See Test Pattern Programming.

There are three methods used to set the initial value of an APG register:

- Execute the appropriate C function(s) in Test Block code.
- Execute the appropriate C function(s) as pattern initial conditions code.
- Use an appropriate pattern instruction/operand to load a specified register from `UDATA` (Memory Test Patterns) or `VUDATA` (Logic Test Patterns) of that instruction.

There are several advantages to executing C function(s) as pattern initial conditions code:

- When looping on a test pattern (see Breakpoint Monitor) the initial conditions code executes for each loop iteration. While this can slow the loop execution speed, it is often necessary for proper pattern operation. C Code in Test Blocks does not execute when looping on a test pattern.
- Initialization code does not consume APG micro RAM (uRAM) leaving more room for other pattern instructions.

Execution of pattern initial conditions code occurs while the pattern is being set up by system software. Execution sequence details are documented in `funtest()` and `start_pattern()`. Note that initial conditions C code is not executed when a given pattern is executed as a Pattern Subroutine from another pattern.

Pattern initial conditions C Code is delimited using the **@{** and **@}** tokens. See Example. The pattern compiler (Patcom) passes any code between **@{** and **@}** directly to the **C** compiler, allowing arbitrary C Code as well as pattern oriented functions to be executed as initial conditions.

The pattern compiler (Patcom) does not support C external function declarations in **.pat** files. Thus, to properly declare a user-written C function such that it can be executed in pattern initial conditions requires that it be declared in the standard *tester.h* file (or a file included there).

The following Nextest functions are commonly found in pattern initial conditions:

APG Register	Function
YMAIN	ymain()
XMAIN	xmain()
YBASE	ybase()
XBASE	xbase()
YFIELD	yfield()
XFIELD	xfield()
AMAIN	amain()
ABASE	abase()
AFIELD	afield()
COUNT# (# = 1 TO 60)	count()
RELOAD# (# = 1 TO 60)	reload()
JAMREG	jamreg()
DATREG	dmain(), dbase()
YINDEX	yindex()
BCKFUN	bckfen()
INTADR	intadr()
TIMER	timer()

A number of other functions are commonly used with the previous functions to set initial condition values based on the size of the address fields: `numx()`, `numy()`, `xmax()`, `ymax()`, `amax()`, etc. These functions, when used in pattern initial conditions code, are evaluated just prior to pattern execution to allow APG configurations to track the current size of the X/Y address fields.

If the `timer()` function is used in Pattern Initial Conditions it must be the last function specified.

Example

```
PATTERN ( my_pattern_name )
@{                      // Initial conditions start here
   ymain( 0 );          // Set ymain register to 0
   count( 1, amax() ); // Load counter #1 = maximum address value
   other_C_code();      // Call user-written C function(s)
@}                      // End of initial conditions
% MAR INC               // Pattern instructions start here
% ...
```

4.8.5 Pattern Labels

See Pattern Overview and Naming.

Labels in a test pattern have the following applications:

- A label can be the destination target for all pattern branching instructions. No absolute addressing (i.e. specifying a physical MAR/VAR address value) is supported.
- A test pattern Pattern Subroutine may be identified using a pattern label, provided the subroutine is in the same source file as the calling instruction. Note that a test pattern, identified using its `PATTERN` name, can also be treated as a subroutine. This method is required when the subroutine is in a different source file than the calling instruction.
- Pattern labels are printed in UI's controller output window when a pattern is stepped using PatternDebugTool. Having a label on each instruction can improve debugging efficiency using this tool.
- A pattern label must be specified by the various C functions which access pattern attributes (`set_tset()`, `get_udata()`, `set_vihh()`, etc.).

Pattern Labels rules:

1. Labels must be a valid C identifier.
2. Labels are **`not`** case sensitive (to be consistent with the rest of the pattern language).
3. Labels **`always`** end with a colon (**`:`**).
4. Labels cannot be reserved words (words predefined in the test programming languages used).
5. Only one label can be used on a given pattern instruction or logic vector.
6. Each label is scoped to the pattern (`PATTERN`) in which it appears.
7. It is illegal to have the same label multiply defined in the same pattern.
8. When a label is used with a pattern instruction, it is required to be the first token immediately following the Pattern Instruction Identifier (%). It is permissible to have other instructions on the same line as the label.
9. The `MAR JUMP` and `VAR JUMP` instructions can only reference a label in the same pattern.
10. The `MAR`/`VAR` conditional branch instructions (`CJMPZ`, `CJMPNE`, etc.) can only reference a label in the same pattern.
11. A label cannot be used on a logic `RPT` vector. See RPT Pattern Instruction.
12. A label can be the same name as the `PATTERN` name (but this is very poor programming practice).
13. A Magnum 1/2/2x pattern Pattern Subroutine which contains only scan vectors must be defined as a `PATTERN()`; i.e. a pattern label cannot be used as a reference to the subroutine. See Scan Test Patterns.
14. Additional rules apply when a Magnum 1/2/2x Logic Test Pattern contains both pattern label(s) and branch instructions. See LVM Branch/Label Limitations and Check for LVM Branch/Label Violations.

Usage

```
% label_x:
  // Other pattern instruction tokens
```

where:

`label_x:` is the label associated with the pattern instruction shown. This label can be used as a jump target, Pattern Subroutine name, or it may not be otherwise used.

Example

The following example shows a pattern label on a memory pattern instruction. In this example, execution loops on this instruction as long as counter 1 (COUNT1) is not zero. This is specified in the MAR instruction with CJMPNZ (conditional jump not zero) to the label **Jump_loop**.

```
% Here:
  COUNT   COUNT1, DECR, AON
  MAR     CJMPNZ, Here
```

The following example shows a pattern label on a logic instruction. Note that the label could be on the same line as the VEC instruction if desired:

```
% Label_1:
  VEC HLXXX1010 XX110101LLL
```

4.8.6 Pattern #Include Files

See Pattern Overview and Naming.

#include statements can be used in test pattern source files just like in standard C programs.

The `#include` syntax is identical in usage to standard C. A typical application is to share Pattern Subroutines (identified using a Pattern Label but not a PATTERN statement) among many patterns or a family of devices.

Usage

```
#include <myPat.pat>
```

This includes the pattern in the file named `myPat.pat` in the current pattern file.

4.8.7 C Preprocessor Support

See Pattern Overview and Naming.

The pattern compiler, Patcom, can optionally use the C preprocessor to support full C-style preprocessor directives. This enables such features as:

- `#define`, `#ifdef`, etc.

- Arguments to macro definitions in test patterns.

Two options are provided:

- Enable the predefined preprocessor, using the following command which is executed before Patcom is invoked:

```
cl /nologo /E /C /X patfilename.pat > .\patfilename.pat.~tmp~
```

- Enable a completely user-defined preprocessor command (more below)

Several methods are available to enable the predefined preprocessor command:

- Select (enable) the Use C Preprocessor option in the APFP Dialog.
- Set the following environment variable:

```
PATCOM_PREPROCESS = 1
```

- Pass the `-E` or `/E` argument when executing Patcom from a command line.

These features are enabled as an option because:

- They increase the time required to compile a test pattern.
- They generate a transient test pattern source file (`./patfilename.pat.~tmp~`), which requires additional disk space.
- The results from the built-in macro and `#include` file processing may differ from that obtained using the C preprocessor, especially in the case of `#include`. Compatibility between the two methods is not guaranteed.

To enable a fully custom preprocessor command set the `PATCOM_CUSTOM_PREPROCESSOR` environment variable to define the command (without including the *patfilename* components). For example, to invoke the same command as the predefined preprocessor command shown above:

```
PATCOM_CUSTOM_PREPROCESSOR = cl /nologo /E /C /X
```

Note: at the time C preprocessor support was added, Patcom was *NOT* enhanced to detect new errors, including any bad syntax, from an optional preprocessor step. To debug complicated macros, execute the preprocessor command manually (from a command line), redirect the result into a text file, and examine that file to observe the output from the preprocessor. The output from a preprocessor step must be syntactically valid pattern instructions.

4.8.7.1 #define

See C Preprocessor Support.

The `#define` compiler directive is used to specify a string substitution to be made by the pattern compiler, providing a similar capability as is available in C code.

Note: the `#define` compiler directive has always been supported by Patcom. The optional C Preprocessor Support uses a different method for processing compiler directives. See C Preprocessor Support.

Note: do ***NOT*** put comments at the end of a `#define` statement. Due to Patcom compiler limitations, side effects may occur. For example:

```
#define MYTERM myval // Do NOT use comments like this.
```

Usage

```
#define string1 string2
```

where:

`#define` is a compiler directive used to perform a string substitution similar to the capability in C.

`string1` is a string that appears in the test pattern.

`string2` is the string that replaces string1 wherever it appears in the pattern.

Examples

In the following logic pattern example, user names are given to `TSET3`, `TEST4`, and `VIHH2` to improve the readability of the pattern. These same `#define` statements may appear in the test program itself so that the same names can be referenced there. When the pattern compiler interprets the vectors, it first substitutes `TSET3` for `ReadTiming`, `TEST4` for `WriteTiming`, and `VIHH2` for `ProgramVoltage`:

```
#define ReadTiming TSET3
#define WriteTiming TSET4
#define ProgramVoltage VIHH2
```

```
% VEC  00000000 XXXXXXXX
% VEC  00001111 HHHHLLLL, ReadTiming
% VEC  00000000 XXXXXXXX
% VEC  10101010 XXXXXXXX, WriteTiming, ProgramVoltage
```

In the following example, the default syntax for specifying CHIPS operands (CS1PT, CS2F, etc.) are replaced with more meaningful tokens: OE_pulse_true, WE_false, CS_true, making the pattern more readable:

```
#define OE_pulse_true  CS1T
#define OE_false       CS1F
#define WE_pulse_true  CS2PT
#define WE_false       CS2F
#define CS_true        CS3T

PATTERN ( some_pat )
   % MAR     INC, NOREAD            // Write data
     CHIPS   OE_false, WE_pulse_true, CS_true
   % MAR     INC, READ              // Read data
     CHIPS   OE_pulse_true, WE_false, CS_true
     PINFUNC ADHIZ
```

4.8.7.2 Newline in Test Pattern Macros

See C Preprocessor Support.

It is possible to embed the newline character (\n) in test pattern macros. For example:

```
#define myMacro % COUNT COUNT4, INCR, AON, \n MAR CJMPNZ label
```

The embedded newline does function correctly with the Test Pattern Line Continuation Character (‘ \’). For example:

```
#define myMacro( op, val) % MAR op \n \
                            UDATA val
```

This feature does not require using the new C Preprocessor Support unless required by other features of the macro, as seen in the 2nd example above, which passes two arguments to the macro.

4.8.8 Test Pattern Line Continuation Character

See Pattern Overview and Naming.

he line continuation character " \" is targeted at splitting long logic vectors across multiple source file text lines. See Logic Vector Syntax.

Note the following:

- C Preprocessor Support requires that the line continuation character be preceded by a space. While this is not required by Patcom's pre-processor it is highly recommended that ALL usage of the line continuation character include a leading space.
- The line continuation character can not be used in a VECDEF Compiler Directive because a VECDEF Compiler Directive can span multiple source file lines without any special syntax. See VECDEF Compiler Directive
- An embedded newline (`\n \`) does function correctly with the line continuation character. See Newline in Test Pattern Macros.

4.9 MAR DONE and/or VAR DONE

See Test Pattern Programming.

Note: this section intentionally contains information about Maverick-I/-II and Magnum 1/2/2x.

The Maverick-I has a single pattern control engine, thus the end of a test pattern is always specified using the `MAR DONE` instruction (even when the pattern otherwise contains only logic pattern instructions). The rest of this section documents Maverick-II and Magnum 1/2/2x operation.

The Maverick-II and Magnum 1/2/2x APGs have two pattern execution control engines:

- The MAR Engine, always used.
- The VAR Engine, used when executing any pattern containing logic instructions.

Thus in Maverick-II and Magnum 1/2/2x, both the MAR Engine and the VAR Engine have the capability to halt pattern execution. This section documents the functionality and rules for

using MAR DONE (Memory Test Patterns and Mixed Memory/Logic Patterns) and/or VAR DONE (Logic Test Patterns and Mixed Memory/Logic Patterns) to end the test pattern, for Maverick-II and Magnum 1/2/2x.

In Maverick-II and Magnum 1/2/2x, from a hardware perspective, the signal path for the pattern DONE signal passes through a MUX (see Done MUX in the VAR Engine), which can select either the VAR DONE signal or the MAR DONE signal, but not both. Pattern execution stops when the pattern instruction generating the DONE signal reaches the DUT. The following rules apply:

- A pattern containing only logic instructions: the Done MUX selects DONE from the VAR Engine; i.e. VAR DONE. This represents pure Logic Test Patterns.
- A pattern containing *any* memory instructions: the Done MUX selects DONE from the MAR Engine; i.e. MAR DONE. This represents both Memory Test Patterns and Mixed Memory/Logic Patterns.

As indicated, for Mixed Memory/Logic Patterns the MAR Engine's DONE signal is always selected. The rest of the information below discusses what occurs when the logic portion of a Mixed Memory/Logic Patterns generates more or less cycles than the memory portion of the pattern.

- If the logic pattern VAR DONE is reached before the memory pattern MAR DONE the last logic vector is repeated until the pattern is halted by the MAR DONE. The active Pin Scramble Map, Time-set and VIHH Map will continue to determine what occurs at the DUT.
- If the memory pattern MAR DONE is reached before the logic pattern VAR DONE pattern execution will stop before all the logic vectors have executed. This is probably bad!
- If the logic portion of the pattern does not include VAR DONE and, in addition, generates fewer cycles than the memory pattern, the t_lvm outputs will be determined by vector memory contents beyond the end of the current logic pattern. This occurs because the default VAR Instruction is INC will point to the VAR address past the last defined vector in the pattern. The vector at this address may be part of some other pattern, or may simply be the residue from system power up. *This is always bad!* Proper operation requires that VAR DONE always be included in all Mixed Memory/Logic Patterns and Logic Test Patterns.

One other important difference exists between Maverick-I vs. Maverick-II and Magnum 1/2/2x logic pattern operation.

- In Maverick-I patterns, the vector address (VAR) is incremented before a VEC instruction executes. In Maverick-II logic patterns and using Magnum 1/2/2x, the VAR Engine increments the vector address (VAR) in preparation for the next

instruction; i.e. after the VEC instruction executes. In most cases the user does not care about this - patterns operate as desired. However, this difference is important at the end of the pattern. At the end of the pattern the control engine in use (MAR or VAR engine) actually causes the last instruction to repeat several times while flushing any failures in the error pipelines back to the APG.

- In Maverick-I, if the last instruction contains both VEC and MAR DONE the vector address will increment several times, while the pipelines are flushed, causing the next few logic vectors in LVM to be executed. These will not be visible in the pattern source file and may be part of some other pattern, or may simply be the residue from system power up. *This is always bad!* Conversely, when the last instruction does not contain VEC; i.e. only MAR DONE, the vector address is not incremented during the error flush cycles, causing the last VEC instruction of the test pattern to be repeated. This is good because the user can control it but this last vector must not generate any strobes.
- Using Maverick-II and Magnum 1/2/2x, the default VAR Instruction is INC. Thus, if the last instruction does not contain VEC (i.e. only VAR DONE) during the error pipeline flush cycles the vector address is incremented, and the next few logic vectors in LVM will be executed. As above, these will not be visible in the pattern source file and may be part of some other pattern, or may simply be the residue from system power up. *This is always bad!* Conversely, if the last instruction contains both VEC and VAR DONE the vector visible in the pattern file will be repeated during the pipeline flush cycles. This is good because the user can control it but this last vector must not generate any strobes.
- The error pipeline flush cycles may generate timed edges at the DUT. The information above can help explain datalogging anomalies and falsely reported errors which can occur when the last vector generates active strobes.

In summary, the correct end of pattern instructions should be:

	Memory	Logic	Mixed
Maverick-I	% MAR DONE	% VEC ... % MAR DONE	% VEC ... % MAR DONE
Maverick-II Magnum 1 Magnum 2 Magnum 2x	% MAR DONE	% VEC ... VAR DONE	% VEC ... VAR DONE MAR DONE

The pattern compiler detects and reports certain errors and warnings related to how MAR DONE and VAR DONE are used vs. a pattern's attributes. Each example below starts with a

simple description, followed by a simple pattern containing the problem (labeled *Incorrect*). Next is the corresponding compiler error message (only the first line of the error message is shown). Last is the correct pattern usage, when appropriate.

1. WARNING: MAR DONE in same instruction as VEC in a Maverick-I pattern:

```
Incorrect:
   PATTERN( mav1_mixed2 ) // Mixed and logic
   %  VEC X
      MAR DONE
LogicPat1.pat(20): warning: MAR DONE within a VEC instruction in
mav1 mode is likely to cause problems
Correct:
   PATTERN( mav1_mixed2 ) // Mixed and logic
   %  VEC X
   %  MAR DONE
```

2. ERROR: logic is only usable in Maverick-II and Magnum 1/2/2x patterns

```
Incorrect:
   PATTERN( mav1_logic3, mav1, logic )
LogicPat1.pat(44): error: logic mode incompatible with mav1
```

3. ERROR: *VAR DONE* is only usable in Maverick-II and Magnum 1/2/2x patterns

```
Incorrect:
   PATTERN( mav1_logic3, mav1, mixed )
   %  VEC X
   %  VAR DONE
LogicPat1.pat(52): error: using APG2 feature while not in APG2
mode: DONE
Correct:
   PATTERN( mav1_logic3, mav1, mixed )
   %  VEC X
   %  MAR DONE
```

4. ERROR: *VAR DONE* requires VEC in Maverick-II and Magnum 1/2/2x patterns

```
Incorrect:
```

```
   PATTERN( mav2_logic1, mav2 )
   %  VEC X
   %  MAR DONE
      VAR DONE
```

```
LogicPat1.pat(66): error: need a VEC bit pattern
```

```
Correct:
   PATTERN( mav2_logic1, mav2 )
   %  VEC X
      MAR DONE
      VAR DONE
```

5. ERROR: *MAR DONE* not usable in Maverick-II and Magnum 1/2/2x pure `logic` patterns

```
Incorrect:
   PATTERN( mav2_logic6, mav2, logic ) // Logic
   %  VEC X
      MAR DONE
      VAR DONE
```

```
LogicPat1.pat(97): error: uRAM bits defined in logic mode: DONE
```

```
Correct:
   PATTERN( mav2_logic6, mav2, logic ) // Logic
   %  VEC X
      VAR DONE
```

4.10 Pattern Subroutines

See Test Pattern Programming.

Much like a computer programming language, pattern subroutines provide for reuse of one or more pattern instructions.

Except for the various subroutine return instructions (`MAR RETURN`, `CRETNZ`, etc. in Memory Test Patterns and `VAR RETURN`, `CRETNT`, etc. in Magnum 1/2/2x Logic Test Patterns) pattern subroutines use the same instructions and syntax used in the main test pattern.

Pattern subroutines can be executed unconditionally, using the `MAR GOSUB` instruction (Memory Test Patterns) and `VAR GOSUB` instruction (Magnum 1/2/2x Logic Test Patterns), or

can be executed conditionally using MAR Conditional Branch-condition Operands (Memory Test Patterns), VAR Multi-DUT Branch-condition Operands and VAR Conditional Branch-condition Operands (Magnum 1/2/2x Logic Test Patterns).

A pattern subroutine may also be called as an interrupt routine when the APG Interrupt Timer count reaches 0.

Note: a subroutine may be identified using a Pattern Label if the subroutine is within the same pattern, or by referring to another PATTERN. The destination of a test pattern jump/branch instruction can only reference a Pattern Label within the same pattern.

A return from a pattern subroutine can be executed unconditionally using the MAR RETURN instruction (Memory Test Patterns) and VAR RETURN instruction (Magnum 1/2/2x Logic Test Patterns), or can be executed conditionally using MAR Conditional Branch-condition Operands (Memory Test Patterns), VAR Multi-DUT Branch-condition Operands and VAR Conditional Branch-condition Operands (Magnum 1/2/2x Logic Test Patterns).

Test patterns which execute subroutines follow the execution sequence outlined below. Note that the MAR GOSUB/RETURN instructions are used here for example purposes:

- The APG instruction containing the MAR GOSUB instruction executes first.
- The first subroutine instruction executes next.
- The other subroutine instructions, if any, execute.
- The subroutine instruction with the RETURN executes.
- The instruction after the original MAR GOSUB instruction executes.

Note that the previous sequence description is somewhat simplified, see APG Instruction Execution.

Pattern subroutines can be identified two ways:

- Within the same test pattern, any instruction with a Pattern Label can be called as a subroutine.
- Any pattern (defined using PATTERN), can be called as a subroutine.

Also note the following:

- A subroutine cannot be called in the last pattern instruction.
- Subroutines can be nested to 3 levels.
- Calling back-to-back pattern subroutines is legal.

- Calling a subroutine which does not return has no side effects; i.e. pattern execution can end within a subroutine due to a DONE instruction (see MAR DONE and/or VAR DONE) or due to stop-on-error with no side effects.
- Pattern subroutine recursion is limited to 3 levels. Go beyond this and an endless loop situation is created (this is always *BAD*).

Note: if a test pattern containing Pattern Initial Conditions code is called as a subroutine the initial conditions code **DOES NOT EXECUTE**.

- Logic pattern subroutines of < 24 execution counts will be loaded into SRAM (see Logic Vector Memory (LVM)). The SRAM can store up to 8K vectors per-pattern. This means a given Logic Test Pattern can define at least 340 unique subroutines which are < 24 execution counts, each consisting of 24 discrete VEC instructions (RPT instructions can decrease the amount of SRAM used). SRAM based subroutines are not subject to the restrictions on pattern branching noted in Magnum 1/2/2x Logic Pattern Rules.
- A Magnum 1/2/2x subroutine which contains pure scan vectors must be defined as a PATTERN( ) because a Pattern Label is not allowed on a pure scan instruction. See Scan Test Patterns.

Example

Example 1:

This example shows the two different methods for defining a subroutine. This example has four parts:

- Pattern File: shows both methods for defining a subroutine.
- Test Block Code: to show how several APG counters were initialized and used to generated the Example Output.
- Example Output: shows the values of the APG counters after pattern execution. Note: COUNT1 is not modified by the initial conditions code seen in two of the pattern subroutines.

Pattern File

```
PATTERN( main_pattern1 )
% label_1:
   COUNT   COUNT1, INCR
   MAR     GOSUB, local_subr1 // Subr = Label in same pattern
```

```
% label_2:
   COUNT   COUNT2, INCR
   MAR     GOSUB, pat_subr1 // Subr = PATTERN
% label_3:
   COUNT   COUNT3, INCR
   MAR     GOSUB, pat_subr1
% label_4:
   MAR     DONE
% local_subr1:
   COUNT   COUNT4, INCR
   MAR     RETURN
PATTERN( pat_subr1 )
@{ // Initial conditions in subroutines do NOT execute
   count( 1, 999 ); // This does NOT execute
@}
% pat_subr1_label_1:
   COUNT   COUNT5, INCR
   MAR     RETURN
```

Test Block Code

The purpose of this is to show how the APG counters are initialized before the pattern executes, and then how the Example Output was generated:

```
   count( 1, 0 ); count( 2, 0 ); count( 3, 0 );
   count( 4, 0 ); count( 5, 0 );
   funtest ( main_pattern1 , finish );
   output("  count 1 => %d (SB=1)", count(1) ); // Should be = 1
   output("  count 2 => %d (SB=1)", count(2) ); // Should be = 1
   output("  count 3 => %d (SB=1)", count(3) ); // Should be = 1
   output("  count 4 => %d (SB=1)", count(4) ); // Should be = 1
   output("  count 5 => %d (SB=2)", count(5) ); // Should be = 2
```

Example Output

```
count 1 =>   1 (SB=1)
count 2 =>   1 (SB=1)
count 3 =>   1 (SB=1)
count 4 =>   1 (SB=1)
count 5 =>   2 (SB=2)
```

Example 2:

This example shows a Magnum 1/2/2x logic pattern calling the same pattern subroutine 3 times:

```
% VEC     XXXX XXXX          // First vector
  VAR GOSUB, sub_pattern     // Execute subroutine next
% VEC     1111 XXXX          // Subroutine returns. Execute
  VAR GOSUB, sub_pattern     // this instruction then
                             // execute the subroutine again
% VEC     1111 XXXX          // Subroutine returns to here. Execute
  VAR GOSUB, sub_pattern     // this instruction then execute
                             // the next instruction
% VEC     1111 XXXX
  VAR     DONE

PATTERN (sub_pattern)
% VEC     0000 HHHH
% RPT 5   0000 XXXX
% VEC     0000 XXXX
% VEC     0000 XXXX
% VEC     0000 XXXX
  VAR RETURN
```

4.11 Error Pipeline Requirements

See APG Controller Engine, MAR Branch Condition Operands.

During test pattern execution, the instruction execution sequence can be conditional, based on whether an error exists; i.e. have any strobes have failed. This is commonly called branch-on-error (or no-error).

A related branch-on-abort (or no-abort) capability allows conditional pattern branching based on whether a fail exists for every DUT being tested in parallel; i.e. branch if all DUTs are bad. The abort signal is derived from the error latch signals, not error flags (more below).

Note: using Magnum 1/2/2x, proper operation of the Abort signal requires that all DUTs be enabled; i.e. all DUTs must be in the Active DUTs Set (ADS). The Abort signal will NOT go active if any DUT(s) are disabled.

In hardware, conditional branch operations can test the state of the error flag signal or abort signal. See PE Error Flag vs. Error Latch Diagram and Error Flag OR Logic. Using Magnum 1/2/2x, conditional branches can also be controlled by ECR Error Counters. As shown in these various diagrams, the error flag signal is the logical OR of the error flags of each pin channel plus all DC Error Flags in all DC Comparators and Error Logic. The error flag signal will be TRUE if any one or more of these other error flags signals is TRUE. The abort signal is TRUE when at least one error *latch* from every group of 8 pins is TRUE.

When using the ECR Error Counters, the branch signal will be TRUE if the associated counter is greater than its comparator value, previously programmed using `ecr_compare_reg_set()`.

Proper test pattern branch-on-error (or no error, on abort, etc.) operation requires that the hardware branch signal be propagated from its source to the APG branch logic. This requires time, specified as a minimum number of pattern cycles (pipeline cycles) executed between the pattern instruction generating the branch signal and the instruction performing the conditional branch operation.

Using Magnum 1/2/2x, the number of required pipeline cycles depends on several factors:

- The cycle period of the pattern instructions used to pipeline the branch signal.
- Does the pattern use the Data Buffer Memory (DBM), in any way; i.e. additional pipeline cycles are required if the test pattern selects the DBM as a data source to be output to the DUT (i.e. `DATGEN BUFBUF`, `DATBUF`, etc.) and/or when using the DBM to acquire generated data (i.e. `DATGEN DBMWR`).
- Does the pattern use the Logic Vector Memory (LVM).
- The source of the branch signal; i.e. branch-on-error or branch based on ECR Error Counters (Total Error Counters, Col Error Counters or Row Error Counters,, see MAR Error-choice Operands.
- How many Sites-per-Controller are specified in the Pin Assignment Table.

The following table shows the minimum number of pipeline cycles required for proper branch operation:

Table 4.11.0.0-1 Magnum 1 Branch-on-error Pipeline Cycle Requirements

<table>
<tr><th>Cycle Period</th><th>Branch-on-error Pipeline Cycles</th><th>Branch-on-TEC Pipeline Cycles</th><th>Branch-on-REC or CEC Pipeline Cycles</th><th>LVM Used?</th><th>DBM Used?</th><th>SITES_PER_CONTROLLER > 1?</th></tr>
<tr><td>20nS</td><td>27</td><td>30</td><td>32</td><td rowspan="15">Add 12

Both used add 24</td><td rowspan="15">Add 12</td><td rowspan="15">Add 1 for each additional site</td></tr>
<tr><td>30nS</td><td>21</td><td>28</td><td>29</td></tr>
<tr><td>40nS</td><td>18</td><td rowspan="3">27</td><td rowspan="2">28</td></tr>
<tr><td>50nS</td><td>17</td></tr>
<tr><td>60nS</td><td rowspan="2">16</td><td rowspan="3">27</td></tr>
<tr><td>70nS</td><td rowspan="4">26</td></tr>
<tr><td>80nS</td><td rowspan="2">14</td></tr>
<tr><td>90nS</td><td rowspan="4">26</td></tr>
<tr><td>100nS</td><td>13</td></tr>
<tr><td>140nS</td><td rowspan="2">12</td></tr>
<tr><td>190nS</td><td rowspan="4">25</td></tr>
<tr><td>230nS</td><td rowspan="2">11</td></tr>
<tr><td>270nS</td><td rowspan="2">25</td></tr>
<tr><td>670nS</td><td>10</td></tr>
</table>

Note the following:

- These are minimum error pipeline cycle requirement values, required for reliable branch operation. No additional cycles are required for cycle periods greater than shown.
- The number of pipeline cycles specified assumes that all error pipeline cycles are set to a cycle period of at least the lenth specified.
- The added cycles for LVM and/or DBM are required in any pattern which uses the option in any way, regardless of whether the instructions related to the branch have any direct involverment or not.

- When using Logic Vector Memory (LVM) (i.e. in Logic Test Patterns or Mixed Memory/Logic Patterns), all error pipeline cycles must be executed using a single instruction which executes immediately prior to the branch instruction. This rule was added 9/8/2008.
- Branch-on-abort has the same pipeline requirements as branch-on-error.

Note: using Magnum 1/2/2x, proper operation of the Abort signal requires that all DUTs be enabled; i.e. all DUTs must be in the Active DUTs Set (ADS). The Abort signal will NOT go active if any DUT(s) are disabled.

- The counter used to control an error pipeline loop can be set to the required cycle count value -1. For example, a simple branch-on-error operation (i.e. 20nS cycle period, no DBM, no LVM, single-site) requires 27 pipeline cycles thus, to use the following example, the COUNT1 counter would be initialized to (27 - 1 = 26):

```
% MAR     READ
  PINFUNC ADHIZ

% here:
  COUNT   COUNT1, DECR, AON
  MAR     CJMPNZ, here

% MAR     CJMPNE, there
```

- The values in the table above do account for strobes delayed into the 2nd cycle. Do NOT reduce the pipeline count when delayed strobes are not used, the situation is not that simple.
- The MAR Error-choice Operands selection also has additional pipeline-related requirements. See Note:.
- Some specific MAR branch-on-error options also have additional pipeline-related requirements. See MAR BOE Type Operands selection.

Note: the following information was added 4/8/2008. It applies to Magnum 1, Magnum 2 and Magnum 2x.

In Magnum 1/2/2x Logic Test Patterns (or mixed patterns) additional rules apply if/when a 2nd (or more) branch logic instruction is based on an error flag which has already been pipelined.

For reference, the most common branch-on-error scenario works as already documented. This can be described as:

```
% VEC or RPT with a strobe
% RPT error pipeline wait loop (conventional)
% VEC Branch-on-error (CJMPE, CJMPNE, CSUBE, etc.)
```

However, the following variation will not work as might be expected. Note the 2nd branch-on-error instruction is using the same pipelined strobe result as the 1st:

```
% VEC/RPT with a strobe
% VEC/RPT error pipeline wait loop (as described above)
% VEC with Branch-on-error // Branches correctly
  // Any number of VEC/RPT instructions without strobes, see Note:
% VEC with Branch-on-error // Won't work as desired, more below
```

Note: this can be 0 or more VEC/RPT instructions, including instructions which branch on things other than error. As described further below, the 2nd branch-on-error above will only operate correctly if immediately preceded by a single-instruction repeat (RPT), of adequate length.

To operate correctly, the 2nd branch in the previous example requires another pipeline loop, called the *LVM error pipeline wait loop* below, as the instruction which executes immediately before the branch instruction, and it MUST be a single instruction loop (RPT or CJMPNZ to same instruction). The minimum loop count value is the value added to the standard error pipeline loop when using LVM:

- 12 cycles for Magnum 1.

For example:

```
% VEC/RPT with a strobe
% VEC/RPT error pipeline wait loop (as described above)
% VEC with Branch-on-error // Branches correctly
  // Any number of VEC/RPT instructions without strobes, see Note:
% RPT 12 LVM error pipeline wait loop // Required before next BOE
% VEC with Branch-on-error
```

When using a mixedsync pattern, to change the branch Error-choice and/or BOE-type requires a MAR instruction and the *LVM error pipeline wait loop* instruction must also select the Error-choice and/or BOE-type used in the following branch instruction:

```
% VEC or RPT with a strobe
% VEC/RPT error pipeline wait loop (as described above)
  MAR select Error-choice and/or BOE-type (or default)
% VEC with Branch-on-error // Branches correctly
  MAR same Error-choice and BOE-type
```

```
   // Any number of VEC/RPT instructions without strobes, see Note:
%  RPT 12 LVM error pipeline wait loop // Required before next BOE
   MAR new Error-choice and/or BOE-type
%  VEC with Branch-on-error
   MAR same Error-choice and BOE-type
```

Finally, the *LVM error pipeline wait loop* must be added for each subsequent branch-on-error instruction based on the original strobe results. Below a sequence of 4 branch-on-error instructions requires 3 additional *LVM error pipeline wait loops*:

```
%  VEC/RPT with a strobe
%  VEC/RPT error pipeline wait loop (as described above)
%  VEC with Branch-on-error // Branches correctly
   // Any number of VEC/RPT instructions without strobes, see Note:
%  RPT 12 LVM error pipeline wait loop // Required before next BOE
%  VEC with Branch-on-error
   // Any number of VEC/RPT instructions without strobes, see Note:
%  RPT 12 LVM error pipeline wait loop // Required before next BOE
%  VEC with Branch-on-error
   // Any number of VEC/RPT instructions without strobes, see Note:
%  RPT 12 LVM error pipeline wait loop // Required before next BOE
%  VEC with Branch-on-error
// Etc.
```

4.12 Algorithmic Pattern Generator (APG) Configuration

See Test Pattern Programming, Algorithmic Pattern Generator (APG).

The APG is used to generate an algorithmic test pattern, commonly used to test memory devices. This section documents the functions used to configured key APG features before executing a Memory Test Pattern which uses that feature. This section includes the following:

- Types, Enums, etc.
- APG Address Mask Functions
- YMAX, XMAX, and AMAX
- Fast Address Axis
- APG Chip Select Polarity Control Function
- APG Chip Select Drive/Strobe Polarity Functions
- APG Data Generator I/O Control Function
- APG Drive/Expect Data Latency Functions
- PE Channel Forced I/O State
- APG Data Register Width Selection Function
- APG JAM Logic Configuration Functions
  - `apg_jam_mode_set(), apg_jam_mode_get()`
  - `apg_jam_ram_set(), apg_jam_ram_get()`
  - `apg_jam_ram_address_set(), apg_jam_ram_address_get()`
- APG User RAM Functions
  - `apg_userram_value_set(), apg_userram_value_get()`
  - `apg_user_ram_address_set(), apg_user_ram_address_get()`
- APG Data Buffer Memory Configuration
- APG Data Inversion Enable Functions
- APG Data Inversion Bank Select Functions
- APG Background Data Inversion Function
- APG Bit-2 Data Inversion Function
- APG Background Bank-A, Bank-B Inversion
- APG Data Topological Inversion (DTOPO) Function

4.12.1 Types, Enums, etc.

See Algorithmic Pattern Generator (APG) Configuration.

Description

The following enumerated types are used in support of various Algorithmic Pattern Generator (APG) Configuration functions.

Usage

The `ChanType` enumerated type is used as an argument to the `adhiz()` function:

```
enum ChanType { RECEIVE, DRIVE, BIDIR };
```

The `BckOperation` enumerated types are used as arguments to the `bckfen()` function, to select an inversion operation:

```
enum BckOperation { force, xeven, xodd, yeven, yodd,
                    xyeven, xyodd, xeven_yodd, xodd_yeven, bit_1,
                    and, or, xor };
```

The `TesterBGFunc` enumerated types are used as arguments to the `bckfen()` function, to identify the complement of an individual X/Y address bit:

```
enum TesterBGFunc { t_x0_bar,  t_x1_bar,  t_x2_bar,  t_x3_bar,
                    t_x4_bar,  t_x5_bar,  t_x6_bar,  t_x7_bar,
                    t_x8_bar,  t_x9_bar,  t_x10_bar, t_x11_bar,
                    t_x12_bar, t_x13_bar, t_x14_bar, t_x15_bar,
                    t_x16_bar, t_x17_bar,
                    t_y0_bar,  t_y1_bar,  t_y2_bar,  t_y3_bar,
                    t_y4_bar,  t_y5_bar,  t_y6_bar,  t_y7_bar,
                    t_y8_bar,  t_y9_bar,  t_y10_bar, t_y11_bar,
                    t_y12_bar, t_y13_bar, t_y14_bar, t_y15_bar,
                    t_bg_on, t_bg_off, t_bgf_na };
```

The DTopoFunc enumerated types are used as arguments to the dtopo() function, to specify a topological inversion function:

```
enum DTopoFunc { xdtopo, ydtopo, xd_and_yd, xd_or_yd, xd_xor_yd };
```

The APGStaticErrorModes enumerated types are used to select the APG's static error mode (see Static Error Choice Functions, Branch-on-error and MAR Error-choice Operands):

```
enum APGStaticErrorModes {  t_errmode1 = 0,
                            t_errmode2 = 1,
                            t_errmode3 = 2,
                            t_errmode4 = 3 };
```

The DataInvControls enumerated type is used to select which Data Inversion Logic inputs are enabled. See APG Data Inversion Enable Functions.

```
enum DataInvControls {
   PAT_INV_EN = 0x0001,
   BIT2_INV_EN = 0x0002,
   DTOPO_INV_EN = 0x0004,
   BCKFN_INV_EN = 0x0008,
   EQFN_INV_EN = 0x0010,
   BCK_A_INV_EN = 0x0020,
   BCK_A_BIT1_INV_EN = 0x0040,
   BCK_A_BIT2_INV_EN = 0x0080,
   BCK_B_INV_EN = 0x0100,
   BCK_B_BIT1_INV_EN = 0x0200,
   BCK_B_BIT2_INV_EN = 0x0400,
};
```

The ApgJamMode enumerated type is used to set and get the APG Data Generator's JAM Register mode. See apg_jam_mode_set(), apg_jam_mode_get():

```
enum ApgJamMode{ t_jam_mode_reg, t_jam_mode_ram };
```

4.12.2 APG Address Mask Functions

See Algorithmic Pattern Generator (APG) Configuration, APG Address Generator.

Description

The numx( ) and numy( ) functions are used to set or get the APG X/Y address masks.

The term *address mask* is used here to refer to the number of APG Address Generator bits enabled on a given axis:

- The X address mask, set using numx( ), determines the number of enabled X address bits.
- The Y address mask, set using numy( ), determines the number of enabled Y address bits.

As indicated, the APG's X and Y address generators can be independently configured to a specific size; i.e. a specific number of address bits are enabled on each axis. This size is typically determined by the number of X and Y address inputs required to correctly test the DUT. During pattern execution, when an X and/or Y address is incremented, it will roll-over to 0x0 when the address reaches the maximum size.

The numx( ) and numy( ) *setter* functions configure the APG's X and Y address generator hardware, and advise the system software as to how many X/Y addresses are being used.

The numx( ) and numy( ) *getter* functions return the currently programmed value.

The maximum number of X address bits = 18*.

The maximum number of Y address bits = 16*.

* the total number of X addresses plus Y addresses cannot exceed 32, thus:

- If numx( ) = 17, numy( ) cannot exceed 15
- If numx( ) = 18, numy( ) cannot exceed 14

This provides for testing devices which up to 18 address bits in one axis with less in the other axis.

During initial program load both numx( ) and numy( ) are set to 0. They are not otherwise set by the system software.

numx( ) and numy( ) are commonly executed in the Site Begin Block, because it is rare that the number of X/Y addresses will change when testing a given DUT.

During pattern execution, the APG Address Generator outputs which are above those enabled using numx( ) and numy( ) will remain at logic-0. It is possible to pin scramble (see Pin Scramble Functions & Macros) these higher addresses to tester channels if a constant logic-0 is useful.

By making numx() and numy() software programmable, it is possible to create test patterns with address schemes which work correctly for almost any DUT address size. And, when debugging test pattern address generation, it is possible to temporarily reduce the X/Y address size (to very small numbers) and single-step the pattern just a few times, typically to evaluate boundary transitions, rather than having to step millions of times.

When the Data Buffer Memory (DBM) is used, the DBM Address Masks are not updated by numx() and numy(). Therefore, any time numx() and numy() are executed the DBM must be reconfigurd (see dbm_config_set()) and reloaded.

Usage

The following functions enable the specified number of X and Y APG Address Generator outputs:

```
void numx( int xbits );
void numy( int ybits );
```

The following functions return the number of address bits currently enabled for the X and Y address generators:

```
int numx();
int numy();
```

where:

xbits specifies the number of X address bits to be enabled. Legal values are 0 to 18. See Description.

ybits specifies the number of Y address bits to be enabled. Legal values are 0 to 16. See Description.

The getter versions of numx() and numy() will return the currently programmed value.

Examples

In this example, Y APG Address Generator outputs Y0 (t_y0) through Y8 (t_y8) are enabled. Outputs Y9 and higher are disabled and will remain at logic-0:

```
numy( 9 );
```

In this example, X address generator outputs X0 (t_x0) through X11 (t_x11) are enabled. Outputs X12 and higher are disabled and will remain at logic-0:

```
numx( 12 );
```

In this example, APG counter 5 (COUNT5) is loaded with the value returned by numx(); i.e. the number of currently enabled X address generator outputs:

```
count( 5, numx() );
```

4.12.3 YMAX, XMAX, and AMAX

See Algorithmic Pattern Generator (APG) Configuration, APG Address Generator.

Description

The X and Y APG Address Generators can be independently configured to enable a specific number of address bits. This is done using the numx() and numy() functions, with the size typically determined by the number of X and Y address inputs required to correctly test the DUT.

The xmax() function can be used to determine the number of unique addresses which can be generated by the APG's X address generator, as currently configured by numx(). The value returned will be $2^{numx()} - 1$.

The ymax() function can be used to determine the number of unique addresses which can be generated by the APG's Y address generator, as currently configured by numy(). The value returned will be $2^{numy()} - 1$.

The amax() function can be used to determine the number of unique addresses which can be generated by the combined X and Y address generators, as currently configured by numx() and numy(). The value returned will be $2^{(numx()+numy())} - 1$.

Memory Test Pattern address generation is typically controlled using APGCounters which, for example, count the number of times an address is modified (incremented, decremented, etc.). If these counter(s) are initialized using xmax(), ymax(), and/or amax(), the pattern will continue to function correctly when the number of used X and/or Y addresses is changed using numx() and/or numy().

Usage

```
int ymax();
int xmax();
```

```
UINT amax();
```

where:

xmax() returns a value between 0 and 2^{18} (262,143), as determined by the currently set numx().

ymax() returns a value between 0 and 22^{16} (65,535), as determined by the currently set numy().

amax() returns a value between 0 and 2^{32}, as determined by the combination of the currently set numx() and numy(). See Description. Even though numx() can be up to 18 (2^{18}) the total combined X + Y addresses is limited to 32 (2^{32}). See APG Address Mask Functions.

Example

The following example uses ymax() to load APG counter 5 (COUNT5) with the maximum number of Y addresses which can be generated as last set using numy():

```
count( 5, ymax() );
```

4.12.4 Fast Address Axis

See Algorithmic Pattern Generator (APG) Configuration, APG Address Generator.

Description

The x_fast_axis() function is used to advise the system software of which APG Address Generator (X or Y) will be generating the address on the fast axis; i.e. the LSB address bits.

Note: x_fast_axis() has no effect on actual test pattern operation: only the test pattern instruction(s) determine whether the X or Y address generator will generate the LSB address bits (fast address axis). Useful results from some Nextest functions requires that the fast axis specified using x_fast_axis(), to match the operation of test patterns.

When manipulating the APG Address Generators, the X or Y address can be treated as the least significant, or fastest changing, address field. In other words, the address at the DUT can be represented as XY, where Y is least significant, or YX, where X is least significant.

Another way to think of the fast, or least significant, axis is that it is the address field (X or Y) that changes fastest in a minmax pattern. This affects the following:

- The operation of the `set_address()` function.
- System software display of APG addresses during execution trace.
- The operation of some Data Buffer Memory (DBM) functions.

The initial program load sets `x_fast_axis( TRUE )`; i.e. the X-axis is fast. The system software does not otherwise modify this value.

When `x_fast_axis()` is used to change the fast axis if the is being used it must be [re]configured (see `dbm_config_set()`) and reloaded. This is required because `dbm_config_set()` records the state returned by `x_fast_axis()`, and this state affects both the DBM hardware configuration and subsequent DBM operation.

Usage

```
void x_fast_axis( BOOL State );
BOOL x_fast_axis();
```

where:

`State` specifies whether the APG X-axis is fast (TRUE) or the Y-axis is fast (FALSE).

The getter version of `x_fast_axis()` returns the currently programmed value.

Example

```
x_fast_axis( TRUE );
output("%s is the fast axis", x_fast_axis() ? "X" : "Y" );
```

4.12.5 APG Chip Select Polarity Control Function

See Algorithmic Pattern Generator (APG) Configuration, APG Chip Select Drive/Strobe Polarity Functions, APG Chip Selects.

Note: these functions are not supported on Maverick-I/-II

Description

The `cs_polarity_set()` function is used to set the active polarity for one specified chip select; i.e. active = logic-1 or logic-0. The polarity of the other chip selects remain unchanged.

The `cs_polarity_get()` function may be used to get the current active chip select polarity for a specified chip select.

The APG has 8 APG Chip Selects, used when testing memory devices, typically on pins which are not address inputs and not data pins; i.e. write enable (WE), output enable (OE), chip select (CS), etc.

During pattern execution, in each tester cycle, each chip select can be independently set to be active or inactive, using the test pattern CHIPS Chip-select-control Operands. All 8 chip selects can generate drive states. `t_cs1` and `t_cs2` can also tri-state and strobe.

The pattern compiler always generates output which assumes all that 8 chip selects are active-low:

- For drive states this means TRUE (`CSnT`, `CSnPT`) = logic-0 = VIL and FALSE (`CSnF`, `CSnPF`) = logic-1 = VIH.
- For strobes states (`t_cs1` and `t_cs2` only) this means strobe TRUE (`CHIPS CSmRDT`) = strobe for logic-0 < VOL and strobe FALSE (`CHIPS CSmRDF`) = strobe for logic-1 > VOH.

There is only one active state for each chip select thus, with regard to `t_cs1` and `t_cs2`, the active polarity is always the same for both drive and strobe.

Usage

```
void cs_polarity_set(TesterFunc chip_select,
                     BOOL active_high DEFAULT_VALUE( TRUE ) );
BOOL cs_polarity_get( TesterFunc chip_select );
```

where:

`chip_select` specifies one chip select to be set. Legal values are of the `TesterFunc` enumerated type but only the chip select values are legal (`t_cs1`..`t_cs8`).

`active_high` is optional and, if used, specifies whether the active state is active-high (TRUE, default) or active-low (FALSE).

`cs_polarity_get()` returns the current active state for a specified chip select. TRUE = active-high, FALSE = active-low.

Example

In the following example, chip selects 2, 4, and 6 are set active-high:

```
cs_polarity_set( t_cs2, TRUE ); // Same as cs_polarity_set(t_cs2);
cs_polarity_set( t_cs4 );
cs_polarity_set( t_cs6 );
```

In the following example, the current active polarity for chip select 1 is retrieved:

```
BOOL active_high = cs_polarity_set( t_cs1 );
```

4.12.6 APG Chip Select Drive/Strobe Polarity Functions

See Algorithmic Pattern Generator (APG) Configuration, APG Chip Select Polarity Control Function, APG Chip Selects.

Description

Note: using Magnum 1/2/2x, an improved method for setting the chip select active polarity is available. See APG Chip Select Polarity Control Function.

The APG has 8 APG Chip Selects, used when testing memory devices, typically on pins which are not address inputs and not data pins; i.e. write enable (WE), output enable (OE), chip select (CS), etc.

During pattern execution, in each cycle, each chip select can be independently set to be active or inactive, using the test pattern CHIPS Chip-select-control Operands. All 8 chip selects can generate drive states. `t_cs1` and `t_cs2` can also strobe.

The pattern compiler always generates output which assumes all that 8 chip selects are active-low:

- For drive states this means TRUE (active) = logic-0 = VIL (`CHIPS CSnT`, `CSnPT`) and FALSE (inactive) = logic-1 = VIH (`CSnF`, `CSnPF`).
- For strobes states (`t_cs1` and `t_cs2` only) this means strobe TRUE = strobe for logic-0 < VOL (`CHIPS CSmRDT`) or strobe FALSE = strobe for logic-1 > VOH (`CHIPS CSmRDF`).

The `cs_active_high()` function or the `cs_read_high()` function (for `t_cs1` or `t_cs2`) may be used to invert this operation; i.e. active = logic-1.

Note the following:

- `cs_active_high()` allows any or all of the 8 chip selects to be set to active-high with a single function execution. `cs_read_high()` only allows `t_cs1` and `t_cs2` to be set.
- There is only one active state for each chip select thus, with regard to `t_cs1` and `t_cs2`, executing either `cs_active_high()` or `cs_read_high()` will set the active state for both drive and strobe for the specified chip selects.
- Using either function, any chip selects not explicitly listed are set to (or left) active-low. Thus, executing `cs_read_high()` always affects 8 chip selects, even though `t_cs3` through `t_cs8` cannot strobe and cannot be set using `cs_read_high()`. See APG Chip Select Polarity Control Function.
- Executing `cs_active_high()` and/or `cs_read_high()` sets a hardware register which causes the control bits for the selected chip select(s) to be high. Any chip select not specified will be set to active-low.

Usage

```
void cs_active_high(TesterFunc Func1 = t_tf_na,
                    TesterFunc Func2 = t_tf_na,
                    TesterFunc Func3 = t_tf_na,
                    TesterFunc Func4 = t_tf_na,
                    TesterFunc Func5 = t_tf_na,
                    TesterFunc Func6 = t_tf_na,
                    TesterFunc Func7 = t_tf_na,
                    TesterFunc Func8 = t_tf_na );
void cs_read_high( TesterFunc Func1 );
void cs_read_high( TesterFunc Func1, TesterFunc Func2 );
```

where:

Func1 though **Func8** are used to specify which of the 8 chip selects are active-high. Legal values are of the `TesterFunc` enumerated type. Using `cs_active_high()` any of the 8 chip select values (`t_cs1`..`t_cs8`) may be specified, in any order. Using `cs_read_high()` one or two values may be specified and are limited to `t_cs1` and `t_cs2`. Any chip select(s) not specified are set active-low.

Example

In the following example, chip selects 2, 4, and 6 are set active-high. All other chip selects remain active-low:

```
cs_active_high( t_cs2, t_cs4, t_cs6 );
```

In the following, only chip select 1 (t_cs1) is set active-high, for both drive and strobe. All other chip selects remain active-low. Since t_cs2 is not listed in the 2nd function it is set to active-low:

```
cs_read_high( t_cs2 );
cs_active_high( t_cs1 );
```

4.12.7 APG Data Generator I/O Control Function

See Algorithmic Pattern Generator (APG) Configuration, APG Data Generator .

Description

The adhiz() function is optionally used to invert the functionality of the test pattern PINFUNC ADHIZ operation.

Note: this function is not supported using Magnum 1, Magnum 2 or Magnum 2x.

4.12.8 APG Drive/Expect Data Latency Functions

See Algorithmic Pattern Generator (APG) Configuration, APG Data Generator .

Description

The expect_delay() function is used to configure expect-data/strobe latency.

Note: Magnum 1 does not support drive-data latency. This section includes some information which applies to Magnum 2/2x also.

Note: as of 2/1/2008, hardware support for expect-data/strobe latency was not implemented in Magnum 1. This note will be removed if/when support is added.

When testing memory devices which have an input data latch and/or output data latch, there may be some latency between when the DUT receives a new address and when the data to be written to that address is actually needed at the DUT or when the data read from that address must be strobed. Here this is called *data latency*.

When designing a memory test pattern it is very desirable that, for a given address, both the address generated in a given cycle and the data to be used at that address be controlled by the same pattern instruction. This requires additional APG hardware to provide for the required data latency. The APG Data Generator hardware can delay the expect-data/strobe generated in a given cycle relative to the address generated in that same cycle. The functions documented here are used to configure this hardware.

By default, no expect-data/strobe latency is enabled; i.e. latency = 0. `expect_delay()` may be used to specify a non-zero expect-data/strobe latency value. When this is done, during pattern execution the APG Data Generator will delay both the expect-data and strobes by up to 7 cycles (Magnum 1), relative to the address and chip selects generated in the same cycle. More below.

Note the following:

- As indicated, during the initial program load, the system software sets expect-data/strobe latency = 0. The system software does not otherwise modify this.
- Any required latency must be configured prior to executing any related test patterns.
- When using expect-data/strobe latency only outputs from the APG Data Generator are delayed. This can only affect pins which are pin scrambled to data generator outputs (i.e. `t_d0` to `t_d35`) (see Pin Scramble Map). These pins should remain continuously scrambled to these data sources to ensure predictable operation.
- Latency values are specified in tester cycle counts, using:
  - The `Delay` argument to `expect_delay()` for expect-data/strobe latency only.
- Expect-data/strobe latency is applied in cycles which are controlled by pattern instructions which do contain `MAR READ`, `READUDATA READV` and `READZ`.
  - If expect-data/strobe latency = 0, the APG does not delay data, strobe enable or tri-state signals relative to the address or chip-select data.
  - If expect-data/strobe latency > 0, the APG delays the data generated by the APG Data Generator (including all data inversions), the strobe enable signals for this data and the tri-state signal (`PINFUNC ADHIZ`) the specified number of cycles relative to the address or chip-select data generated in the same cycle. The delay occurs between the APG and the Pin Scramble MUX.

Also note:

- Per-cycle time-set selection is not delayed. In effect, the time-set selected in the cycle in which the delayed data/strobe is actually issued determines the timing/ format applied to the DUT.
- Per-cycle pin scramble selection is not delayed. Thus, when using latency, in a given cycle, pins which are scrambled to an APG data generator output will receive the delayed data from **n**-cycles earlier, where **n** is the programmed expect-data/strobe latency. The user is responsible for coordinating all test pattern pin scramble selections to obtain the desired delayed data operation.
- When using latency, since there is only one path from the APG Data Generator's outputs to the Pin Scramble MUX, as a test pattern transitions from strobe cycles to drive cycles there is an **n**-cycle overlap of delayed expect-data/strobe and drive-data, where **n** is the number of cycles of expect-data/strobe latency. When switching from strobe cycles to drive cycles, the delayed expect data takes precedence over the drive data (i.e. the drive data is not used). This usually means that there must be **n** *don't care* data cycles between the last strobing cycle and the first *important* drive cycle.
- When using expect-data/strobe latency, the APG Data Generator's I/O control signals are also affected.`MAR LATCH/NOLATCH` and `RESET` are delayed the expect-data/strobe latency value.
- When using expect-data/strobe latency, any strobe(s) on pins scrambled to APG data generator outputs which are generated in instructions closer to `MAR DONE` than the latency value are discarded (i.e. can't fail). For example, given:

```
expect_delay( 3 );
funtest( myPat, finish );
```

The following pattern will use strobes as noted in the comments. Note this example will generate strobes in the `MAR DONE` cycle but that these only occur once (never use any strobe instructions with a `MAR DONE` instruction):

```
PATTERN( myPat, memory )
%  MAR        READ    // Strobes are used, 3 cycles later
   PINFUNC    ADHIZ
%  MAR        READ    // Strobes discarded, READ can't fail
   PINFUNC    ADHIZ
%  MAR        READ    // Strobes discarded, READ can't fail
   PINFUNC    ADHIZ
%  MAR  DONE          // Strobes from 3 cycles earlier, 1 time
```

- When using expect-data/strobe latency, since APG addresses and expect data are both pipe-lined, errors logged to the Error Catch RAM (ECR) are correctly aligned with addresses.

Usage

The following programs the expect-data/strobe latency value:

```
void expect_delay( int Delay );
```

The following returns the currently programmed expect-data/strobe latency value:

```
BYTE expect_delay();
```

where:

`Delay` and **`delay`** specify the number of cycles latency. Legal values are 0 to 7.

The version of `expect_delay()` which returns a `BYTE` value returns the currently programmed expect-data/strobe latency value.

Example

The following example specifies 3 cycles of expect-data/strobe latency:

```
expect_delay( 3 );
```

4.12.9 PE Channel Forced I/O State

See Algorithmic Pattern Generator (APG) Configuration.

Description

It is sometimes convenient to statically set the I/O state of the PE driver on selected pins. For example, when testing a ROM, which can only output data, it may be convenient to specify that the tester pins connected to the ROM outputs always be tri-stated.

The `tri_state()` function forces the specified pins to tri-state.

The `drive_only()` function forces the specified pins to drive.

The `io_enable()` function restores normal I/O operation for the specified pins.

Normal I/O operation is enabled during the initial program load, but is not otherwise changed by the system software.

Note: these functions directly set hardware registers which control the I/O state of the specified pins. These registers take precedence over the I/O signals from the test pattern.

Note: the `tri_state()` function causes the PE Driver to tri-state. The currently set Magnum PE Driver Modes will determine whether the pin(s) are set to VZ, VTT, or open-circuit.

Note: the `drive_only()` function also prevent pins from responding to the VIHH signal from the test pattern; i.e. drive-only pins cannot be set to the VIHH drive level.

Usage

```
void tri_state( PinList* pPinList );
void drive_only( PinList* pPinList );
void io_enable( PinList* pPinList );
```

where:

`pPinList` specifies the pins to be programmed.

Example

The following example sets all pins identified in the pin list named `rom_data_bus_pins` to tri-state:

```
tri_state( rom_data_bus_pins );
```

4.12.10 APG Data Register Width Selection Function

See Algorithmic Pattern Generator (APG) Configuration, APG Data Generator.

Description

The `data_reg_width()` function is used to set or get the data width of the APG data registers (DMAIN, DBASE), which can be configured as one 36-bit wide register or as two 18-bit wide registers.

APG Data Generator shift, rotate, increment and decrement operations operate on either a single 36-bit register or two 18-bit registers, based on how the data register is configured. In 18-bit configuration, the two register halves perform identical operations.

All data registers are set to 36-bits wide at program initialization but is not otherwise modified by system software.

Note: proper Data Buffer Memory (DBM) operation does NOT require that the width configuration of the DBM (set using `dbm_config_set()`) match that of the data register (set using `data_reg_width()` or by default). However, the user is responsible for detailed understanding DBM operation when these widths are different. See DBM Data Widths.

Usage

The following function sets the width of the data register:

```
void data_reg_width( int Width );
```

The following function gets the currently programmed data register width:

```
int data_reg_width();
```

where **`Width`** specifies the desired APG Data Generator width. Legal values are 18 or 36.

`data_reg_width()` returns the currently configured data register width.

Example

The following example configures the APG Data Generator to 18-bit wide configuration. Then retrieves and prints the current value.

```
data_reg_width( 18 );
output(" Width => %d", data_reg_width() );
```

4.12.11 APG JAM Logic Configuration Functions

See Algorithmic Pattern Generator (APG) Configuration, JAM Logic.

This section contains the following:

- `apg_jam_mode_set(), apg_jam_mode_get()`
- `apg_jam_ram_set(), apg_jam_ram_get()`
- `apg_jam_ram_address_set(), apg_jam_ram_address_get()`

Also see DATGEN Dataout Operand.

4.12.11.1 apg_jam_mode_set(), apg_jam_mode_get()

See APG JAM Logic Configuration Functions, JAM Logic.

Note: first available in software release h1.1.23.

Description

The `apg_jam_mode_set()` function is used to configure the APG Data Generator's JAM Logic, specifically to configure the JAM Select MUX This is called the JAM mode. The `apg_jam_mode_get()` function is used to get the currently set JAM mode.

Note the following:

- `apg_jam_mode_set()` sets a single global mode, shared by all APGs in the system. This configures the JAM Select MUX identically for all APGs.
- The default JAM mode is set during the initial program load and selects the JAM Register. This matches the original JAM register operation. The mode is not otherwise changed by the system software.
- The alternate mode selects the JAM RAM; i.e. `t_jam_mode_ram`. This must be done before executing any test pattern which depends on JAM RAM content.

Usage

The following function configures the APG Data Generator's JAM Select MUX for all boards in the system:

```
void apg_jam_mode_set(
        ApgJamMode mode DEFAULT_VALUE( t_jam_mode_reg ) );
ApgJamMode apg_jam_mode_get();
```

where:

mode is optional and, if used, specifies the desired mode. Legal values are of the ApgJamMode enumerated type. Default = t_jam_mode_reg, selecting the original JAM Register.

apg_jam_mode_get() returns the currently set mode.

Example

```
apg_jam_mode_set( t_jam_mode_ram );
ApgJamMode mode = apg_jam_mode_get();
```

4.12.11.2 apg_jam_ram_set(), apg_jam_ram_get()

See APG JAM Logic Configuration Functions, JAM Logic.

Note: first available in software release h1.1.23.

Description

The apg_jam_ram_set() function is used to write values to the APG Data Generator's JAM RAM. The apg_jam_ram_get() function is used to read values from the JAM RAM.

Note the following:

- The JAM RAM can be accessed regardless of the currently set JAM mode (set using apg_jam_mode_set()).

Usage

The following function writes a single value to the JAM RAM on all boards in the system:

```
void apg_jam_ram_set( int addr, __int64 data );
```

The following function writes one or more values to the JAM RAM on all boards in the system:

```
void apg_jam_ram_set(  int addr,
                       Int64Array vals,
                       int count DEFAULT_VALUE( -1 ) );
```

The following function writes a single value to the JAM RAM on one specified board. This may be used when Sites-per-Controller > 1:

```
void apg_jam_ram_set( HSBBoard Board, int addr, __int64 data );
```

The following function writes one or more values to the JAM RAM on one specified board. This may be used when Sites-per-Controller > 1:

```
void apg_jam_ram_set(  HSBBoard Board,
                       int addr,
                       Int64Array vals,
                       int count DEFAULT_VALUE( -1 ) );
```

The following function reads one value from the JAM RAM:

```
__int64 void apg_jam_ram_get( int addr );
```

The following function reads one or more values from the JAM RAM:

```
void apg_jam_ram_get(  int addr,
                       Int64Array &vals,
                       int count DEFAULT_VALUE( -1 ) );
```

The following function reads one value from the JAM RAM on a specified board:

```
__int64 void apg_jam_ram_get( HSBBoard Board, int addr );
```

The following function reads one or more values from the JAM RAM on a specified board::

```
void apg_jam_ram_get(  HSBBoard Board,
                       int addr,
                       Int64Array &vals,
                       int count DEFAULT_VALUE( -1 ) );
```

where:

`addr` specifies the first (or only) JAM RAM address to be written or read. Legal values are 0x0 to 0x3FFF (2^{14} -1).

`data` specifies one value to be written to the JAM RAM at **`addr`**. Only the low 36 bits are used.

`vals` is a previously initialized `Int64Array` from which **`count`** values are written to the JAM RAM beginning at **`addr`**. **`count`** is optional and defaults to -1 = all values in **`vals`** are written to the JAM RAM. Only the low 36 bits of each value are used. Any error related to the

size of the JAM RAM vs. **addr** and **count** values causes a warning to be issued and apg_jam_ram_set() to return without completing the operation.

Board identifiies a specific board to be accessed. This may be used when Sites-per-Controller > 1:

The versions of apg_jam_ram_get() which return __int64 return the value read from **addr**.

Example

```
Int64Array vals = {0xA5695A960, 0x5A96A569F };
apg_jam_ram_set( 0, vals );
apg_jam_ram_get( 0, &vals );
if( int c = vals.GetSize() )
   for( int i = 0; i < c; ++i )
      output(" Val[%d] = %I64d", i, vals.GetAt( i ) );
```

4.12.11.3 apg_jam_ram_address_set(), apg_jam_ram_address_get()

See APG JAM Logic Configuration Functions, JAM Logic.

Note: first available in software release h1.1.23.

Description

The apg_jam_ram_address_set() function is used to set an initial value in the JAM RAM Address Counter. The apg_jam_ram_address_get() function is used to read the current value from the JAM RAM Address Counter.

Note the following:

- apg_jam_ram_address_set() sets a single global value, written to all APGs in use.
- During the initial program load the JAM RAM Address Counter value is set = 0. The address is not otherwise changed by the system software.
- Proper operation of the JAM RAM requires the JAM RAM Address Counter be set before executing any test pattern which depends on JAM RAM content.

- During pattern execution, the JAM RAM Address Counter is incremented, decremented or not modified (hold) by the DATGEN Dataout Operands. The JAM RAM Address Counter is affected in the cycle controlled by a given instruction. For example, if the initial JAM RAM address is set = 0 and the first `DATGEN` instruction increments the JAM RAM address then the first set of data will be from address 1, not 0.
- During pattern execution, the JAM RAM Address Counter will roll-over (or under) when the maximum (or minimum) value is reached.
- The JAM RAM Address Counter can be accessed regardless of the currently set JAM mode (set using `apg_jam_mode_set()`).

Usage

The following function sets the JAM RAM Address Counter on all boards in the system:

```
void apg_jam_ram_address_set( int address );
```

The following functions gets the current JAM RAM Address Counter:

```
int apg_jam_ram_address_get();
```

where:

`address` is the value to be written to the JAM RAM Address Counter. Legal values are 0x0 to 0x3FFF (2^{14} -1).

`apg_jam_ram_address_get()` returns the address read from the JAM RAM Address Counter.

Example

```
apg_jam_ram_address_set( 0 );
int addr = apg_jam_ram_address_get();
```

4.12.12 APG User RAM Functions

See Algorithmic Pattern Generator (APG) Configuration, APG User RAM.

This section includes:

- `apg_userram_value_set(), apg_userram_value_get()`
- `apg_user_ram_address_set(), apg_user_ram_address_get()`

4.12.12.1 apg_userram_value_set(), apg_userram_value_get()

See APG User RAM Functions, APG User RAM.

Description

The `apg_userram_value_set()` function is used to set (write) one or more values in the APG User RAM.

The `apg_userram_value_get()` function is used to get one or more values from the APG User RAM.

Note the following:

- When multiple values are being accessed two parameters are specified:
  - `data_array` is a user defined array containing multiple values to be written to the APG User RAM or to return multiple values read from the APG User RAM.
  - `count` specifies the number of values to be read into the array or written from the array.

  When using `apg_userram_value_set()` and `apg_userram_value_get()`, the combination of starting address, the size of the data array and the `count` value should be consistent vs. the size of the APG User RAM. The system software silently ignores any related errors; i.e. get/set operations silently stop if/ when any errors are detected.

Usage

The following function sets a single URAM address to the specified value:

```
void apg_userram_value_set( int addr, __int64 data );
```

The following function will sequentially load values from the specified array into the URAM, starting at URAM address 1:

```
void apg_userram_value_set( Int64Array &data_array );
```

The following function will load `count` URAM locations, starting at the specified address, with values from the specified array:

```
void apg_userram_value_set( int addr,
                            Int64Array &data_array,
                            int count );
```

The following function returns the value from the specified URAM address:

```
__int64 apg_userram_value_get( int addr );
```

The following function will return `count` values, beginning at the specified address, in the specified data array:

```
void apg_userram_value_get( int addr,
                            Int64Array &data_array,
                            int count );
```

where:

`addr` specifies the (first) APG User RAM location to be accessed. Legal values range from 1-4096.

`data` specifies one value to be written to the APG User RAM.

`data_array` is an existing `Int64Array` used in two contexts:

- In setter functions, **`data_array`** contains the data to be written to the APG User RAM. The array should contain at least **`count`** values, if not, some addresses will not be changed.
- In getter functions, **`data_array`** is used to return data read from the APG User RAM. The array is automatically resized by the system software as needed. Any prior contents are lost. The size and individual elements in the **`data_array`** can be obtained using standard `CArray` member functions.

`count` specifies the number of values to be written (set) or read (get) to/from the APG User RAM. Legal values are 1-4096.

`apg_userram_value_get()` which returns an `__int64` value returns the value read from the specified address.

Example

The following example writes 13 to the first APG User RAM location (address 1):

```
apg_userram_value_set( 0x1, 13);
```

The following example creates an array containing 6 values then writes those values to sequential APG User RAM addresses beginning at location address 199:

```
Int64Array vals;
vals.Add( 1  );
vals.Add( 3  );
vals.Add( 5  );
vals.Add( 7  );
```

```
vals.Add( 11 );
vals.Add( 13 );
apg_userram_value_set( 199, vals, vals.GetSize() );
```

The following example reads and returns the value from APG User RAM address 999:

```
__int64 value = apg_userram_value_get( 999 );
```

The following example reads and returns the value from 10 sequential APG User RAM addresses starting address 29. Any prior values in the `getvals` array are lost:

```
Int64Array getvals;
apg_userram_value_get( 29, getvals, 10 );
```

4.12.12.2 apg_user_ram_address_set(), apg_user_ram_address_get()

See APG User RAM Functions, APG User RAM.

Description

The `apg_user_ram_address_set()` function is used to write an address into the User RAM Address Index Register.

The `apg_user_ram_address_set()` function is used to retrieve the current address from the User RAM Address Index Register.

Usage

The following function writes an address into the User RAM Address Index Register:

```
void apg_user_ram_address_set( int address );
int apg_user_ram_address_get();
```

where:

`address` specifies the address to be written. Legal values range from 0-4095.

Note: test pattern instructions uses explicit tokens to identify a User RAM address. In the pattern language, `URAM1` is equivalent to the address value 0 when using `apg_user_ram_address_set()`.

`apg_user_ram_address_get()` returns the valuer read from the User RAM Address Index Register.

Example

```
apg_user_ram_address_set( 0x0 );
int addr = apg_user_ram_address_get();
```

4.12.13 APG Data Buffer Memory Configuration

See Algorithmic Pattern Generator (APG) Configuration.

Prior to use, the Data Buffer Memory (DBM) must be confgured, as documented in Data Buffer Memory Software (DBM).

4.12.14 APG Data Inversion Enable Functions

See Data Inversion Logic, Algorithmic Pattern Generator (APG) Configuration.

Description

The `apg_datainv_A_enable_set()` and `apg_datainv_B_enable_set()` functions are used to setup the individual data inversion enable bits for the two banks of Data Inversion Logic in the APG Data Generator.

The `apg_datainv_A_enable_get()` and `apg_datainv_B_enable_get()` functions are used to retrieve the current data inversion enable bits configuration for the two banks of Data Inversion Logic.

Note the following:

- As shown in the APG Data Inversion Block Diagram, the Data Inversion Logic contains two independent banks of inversion logic, identified as Data Inversion Bank-A and Data Inversion Bank-B.

- Each bank has a separate set of enable bits, one bit for each of the available data inversions. `apg_datainv_A_enable_set()` and `apg_datainv_A_enable_get()` access the enable bits for the Data Inversion Bank-A. `apg_datainv_B_enable_set()` and `apg_datainv_B_enable_get()` access the enable bits for the Data Inversion Bank-B.
- A single bit-wise value is used to configure the enable bits for each bank. User code logically OR's the desired enable bits to set any combination of data inversions. The `DataInvControls` enumerated type is available to help program these enables, see Example.
- The enable bits must be setup before executing a test pattern which utilizes data inversions; i.e. the enable bits are not controlled from the test pattern.
- The final level of data inversion, controlled by `UDATA` bits and the `DATGEN XORINV` pattern instruction, do not have an equivalent enable bit and thus are not affected by these functions.
- For backwards compatibility, during the initial program load *most* data inversion enable bits are set to the enabled state and the Data Inversion AB Select MUXs are set to route the Data Inversion Bank-A output to all 36 data generator outputs (see APG Data Inversion Bank Select Functions). However, the enables for the Background Bank-A and Background Bank-B data inversions are set to the disable state (the Maverick-I/-II do not include this logic). The system software does not otherwise change the state of these enables.
- In Multi-DUT Test Programs the operation of these functions is not affected by Active DUTs Set (ADS) or Ignored DUTs Set (IDS).

Description

The following functions are used to set the data inversion enable bits:

```
void apg_datainv_A_enable_set( int value );
void apg_datainv_B_enable_set( int value );
```

The following functions are used in multi-site per controller situations to set the data inversion enable bit configuration for a specified Site Assembly Board (when Sites-per-Controller = 1 (default) these functions are equivalent to the two previous versions):

```
void apg_datainv_A_enable_set( HSBBoard Board, int value );
void apg_datainv_B_enable_set( HSBBoard Board, int value );
```

The following functions are used to get the current enable bit configuration for Data Inversion Bank-A or Data Inversion Bank-B:

```
int apg_datainv_A_enable_get();
```

```
int apg_datainv_B_enable_get();
```

The following functions are used in multi-site per controller situations to get the current enable bit configuration for Data Inversion Bank-A or Data Inversion Bank-B for a specified Site Assembly Board. Note that when Sites-per-Controller = 1 (default) these functions are equivalent to the two previous versions):

```
int apg_datainv_A_enable_get( HSBBoard Board );
int apg_datainv_B_enable_get( HSBBoard Board );
```

where:

`value` is a bit-wise value which specifies which data inversions are enabled. The `DataInvControls` enumerated type is available to help program these enables, see Example.

`Board` is used when Sites-per-Controller > 1 to identify a specific Site Assembly Board to be accessed.

`apg_datainv_A_enable_get()` returns a bit-wise value representing the current data inversion enable bit configuration. A logic-1 means the corresponding inversion is enabled. Only the low 5 bits are valid.

`apg_datainv_B_enable_get()` returns a bit-wise value representing the current data inversion enable bits for Data Inversion Logic bank B. A logic-1 means the corresponding inversion is enabled. Only the low 11 bits are valid (see `DataInvControls`).

Example

The following example enables Background data inversion (see APG Background Data Inversion Function) and data topological inversion (see APG Data Topological Inversion (DTOPO) Function) for Data Inversion Bank-B and sets data generator outputs D8..D15 to receive data from Data Inversion Bank-B:

```
apg_datainv_B_enable_set( BCKFN_INV_EN | DTOPO_INV_EN );
apg_datainv_AB_select_set(0xFF00); // apg_datainv_AB_select_set()
```

4.12.15 APG Data Inversion Bank Select Functions

See Data Inversion Logic, Algorithmic Pattern Generator (APG) Configuration.

Description

The `apg_datainv_AB_select_set()` function is used to setup the Data Inversion AB Select MUXs in the APG Data Inversion Logic. This determines which bank (A vs. B) of data inversion is routed each of the 36 APG Data Generator outputs.

The `apg_datainv_AB_select_get()` function is used to retrieve the current Data Inversion AB Select MUXs configuration.

Note the following:

- As shown in the APG Data Inversion Block Diagram, the data generator's Data Inversion Logic contains two independent banks of inversion logic, identified as Data Inversion Bank-A and Data Inversion Bank-B. The Data Inversion AB Select MUXs determine which bank of inversion logic is routed to each of the 36 data generator outputs.
- A single bit-wise value is used to configure these MUXs. See Usage.
- The MUXs must be setup before executing the test pattern; i.e. they are not controlled from the test pattern.
- The final level of data inversion, controlled by `UDATA` bits and the `DATGEN XORINV` pattern instruction, are not affected by these MUXs.
- For backwards compatibility, during the initial program load the Data Inversion AB Select MUXs are set to route data from Data Inversion Bank-A to all 36 data generator outputs (see APG Data Inversion Enable Functions). The system software does not otherwise modify the configuration of these MUXs.
- In Multi-DUT Test Programs the operation of these functions is not affected by Active DUTs Set (ADS) or Ignored DUTs Set (IDS).

Description

The following function is used to set the 36 Data Inversion AB Select MUXs:

```
void apg_datainv_AB_select_set( __int64 value );
```

The following function is used in multi-site per controller situations to set the Data Inversion AB Select MUXs configuration for a specified Site Assembly Board (when Sites-per-Controller = 1 (default) this function is equivalent to the previous version):

```
void apg_datainv_AB_select_set( HSBBoard Board, __int64 value);
```

The following function is used to get the current Data Inversion AB Select MUXs configuration:

```
__int64 apg_datainv_AB_select_get();
```

The following function is used in multi-site per controller situations to get the Data Inversion AB Select MUXs configuration for a specified Site Assembly Board. Note that when Sites-per-Controller = 1 (default) this function is equivalent to the previous version:

```
__int64 apg_datainv_AB_select_get( HSBBoard Board );
```

where:

`value` is a bit-wise value which specifies which data generator outputs (D35..D0) are routed to the Data Inversion Bank-A vs. Data Inversion Bank-B. A logic-1 selects Data Inversion Bank-B for the corresponding APG data generator output. The LSB represents D0, LSB+1 represents D1, etc. Only the low 36 bits are used. See Example.

`Board` is used when Sites-per-Controller > 1 to identify a specific Site Assembly Board to be accessed.

`apg_datainv_AB_select_get()` returns a bit-wise value representing which data generator outputs (D35..D0) are routed to the Data Inversion Bank-A vs. Data Inversion Bank-B. See **`value`** above.

Example

See Example.

4.12.16 APG Background Data Inversion Function

See Data Inversion Logic, Algorithmic Pattern Generator (APG) Configuration.

Description

The `bckfen()` function is used to configure the Background Inversion logic in the APG Data Generator's Data Inversion Logic. A typical application of the background inversion logic is to generate various checkerboard data patterns.

Note the following:

- `bckfen()` must be executed, to configure the underlying hardware, before executing a test pattern which uses DATGEN Background Function Operands.

- The Background Inversion logic is used to conditionally invert data as a function of the X and/or Y address generated by the APG Address Generator. The `bckfen()` function defines a desired inversion logic operation and selects which APG X/Y address bit(s), or their complements, will affect the operation (more below).
- The Background Inversion logic generates a single invert bit which will either invert or not-invert selected data generator outputs, if enabled using the APG Data Inversion Enable Functions.
- When the `bit2fen()` function is used to select a bit2 inversion input this also changes the 2nd bit used by the background inversion. Similarly, the `bit2` argument to `bckfen()` selects the input used for bit2 inversion.
- In Multi-DUT Test Programs the operation of these functions is not affected by Active DUTs Set (ADS) or Ignored DUTs Set (IDS).

The `bckfen()` function has two basic forms:

- Invert based on X/Y address parity. See Background Function Selection: parity options below.
- Invert based on the relative states of two specified X/Y address bits. See Background Function Selection: Address Bit Options below.

The following table describes the data inversion options based on X/Y address parity:

Table 4.12.16.0-1 Background Function Selection: parity options

BckOperation	Description of Inversion Performed
`force`	Unconditionally force a data inversion (default)
`xeven`	Invert data on X address parity = even
`xodd`	Invert data on X address parity = odd
`yeven`	Invert data on Y address parity = even
`yodd`	Invert data on Y address parity = odd
`xyeven`	Invert data on XY address parity = even
`xyodd`	Invert data on XY address parity = odd
`xeven_yodd`	Invert data on X address parity = even AND Y address parity = odd
`xodd_yeven`	Invert data on X address parity = odd AND Y address parity = even

The following table describes the data inversion options which consider individual X/Y address bits, which are identified with separate arguments using `TesterFunc` and/or

TesterBGFunc values. The latter provides for selecting the complement of a specified address bit:

Table 4.12.16.0-2 Background Function Selection: Address Bit Options

BckOperation	Description of Inversion Performed
and	Invert data on address bit1 logically AND'ed with address bit2
or	Invert data on address bit1 logically OR'ed with address bit2
xor	Invert data on address bit1logically XOR'ed with address bit2

Usage

The following function inverts data based on a specified operation. Only operations listed in Background Function Selection: parity options are valid for this version of bckfen():

```
void bckfen( BckOperation Operation );
```

The following functions invert data based on a specified operation and two specified X/Y address bit(s). Only operations listed in Background Function Selection: Address Bit Options are valid for these versions of bckfen():

```
void bckfen( BckOperation Operation, TesterFunc bit1 );
void bckfen( BckOperation Operation, TesterBGFunc bit1 );
void bckfen( BckOperation Operation,
             TesterFunc bit1,
             TesterFunc bit2 );
void bckfen( BckOperation Operation,
             TesterBGFunc bit1,
             TesterFunc bit2 );
void bckfen( BckOperation Operation,
             TesterFunc bit1,
             TesterBGFunc bit2 );
void bckfen( BckOperation Operation,
             TesterBGFunc bit1,
             TesterBGFunc bit2 );
```

where:

Operation specifies the desired data inversion background operation. Legal values are of the BckOperation enumerated type. As indicated above, only certain BckOperation values are valid for each version (overload) of bckfen().

bit1 and **bit2** each identify one X or Y address bit (or its complement) to control inversion. See Background Function Selection: Address Bit Options. Legal values must be of the TesterFunc and/or TesterBGFunc enumerated types. Using TesterFunc only APG address values are legal (t_x0, t_y0, etc.). Inversion on a single address bit can be accomplished using the and operation with the same input specified for **bit1** and **bit2**. See Example 3 below.

Examples

The following example inverts data generator outputs in any cycle in which APG Y address t_y12 AND'ed with the complement of X address t_x4 = TRUE:

```
bckfen( and, t_y12, t_x4_bar );
```

The following example inverts data generator outputs in any cycle in which the APG X address parity = even:

```
bckfen( xeven );
```

The following example inverts the data register generator in any cycle in which the Y address t_y7 is TRUE (logic-1):

```
bckfen( and, t_y7, t_y7 );
```

4.12.17 APG Bit-2 Data Inversion Function

See Data Inversion Logic, Algorithmic Pattern Generator (APG) Configuration.

Description

The bit2fen() function is used to select one X or Y address bit (or its complement) which, when logically TRUE, will invert the output of the APG Data Generator (see Data Inversion Logic). Bit2 inversion generates a single invert bit which will either invert or not-invert selected data generator outputs, if enabled using APG Data Inversion Enable Functions.

Note the following:

- bit2fen() must be executed before executing a test pattern, to configure the underlying hardware.

- Any value specified using `bit2fen()` also affects the operation of the Background data inversion (see APG Background Data Inversion Function), programmed using `bckfen()`. Similarly, any `bit2` argument speciifed using `bckfen()` selects the `bit2` X/Y address (or complement) the same as if using `bit2fen()`.
- `t_bg_off` is used to disable bit2 data inversion. Or, set the appropriate value with the APG Data Inversion Enable Functions.
- In Multi-DUT Test Programs the operation of these functions is not affected by Active DUTs Set (ADS) or Ignored DUTs Set (IDS).

Usage

The following function selects the input address used for bit2 data inversion:

```
void bit2fen( TesterFunc bit2 );
void bit2fen( TesterBGFunc bit2 );
```

where:

`bit2` identifies one X or Y address bit (or its complement) to control inversion. Legal values must be of the `TesterFunc` or `TesterBGFunc` enumerated types. Using `TesterFunc` only APG address values are legal (`t_x0`, `t_y2`, etc.).

Examples

```
bit2fen( t_x0 );
bit2fen( t_y0_bar );
```

4.12.18 APG Background Bank-A, Bank-B Inversion

See Data Inversion Logic, Algorithmic Pattern Generator (APG) Configuration.

Note: as of 1/12/2005, this function and related hardware is under development and subject to change. This note will be deleted when the related features are complete.

Description

The `apg_datainv_A_func_set()` and `apg_datainv_B_func_set()` functions are used to configure the input addess selections and the logic operation performed by Background Bank-A and Background Bank-B data inversions (see Data Inversion Logic).

The output of the main Background data inversion logic is common to both banks (A/B) of Data Inversion Logic. However, Magnum 1/2/2x have two other sets of similar logic, called Background Bank-A and Background Bank-B, which operate much like main Background data inversion but are each constrained to a single data inversion bank. Note the following:

- These functions must be executed before executing a test pattern. There are no test pattern enable bits which affect these data inversion signals.
- Both the Background Bank-A logic and Background Bank-B logic have two inputs, which are selectable from any APG X/Y address or address complement (see Data Inversion Logic). The Background Bank-A input selections are independent of the Background Bank-B inputs.
- Background Bank-A outputs are routed to Data Inversion Bank-A only. Similarly, Background Bank-B outputs are routed to Data Inversion Bank-B only. The selection of which bank of data inversion is routed to each data generator output is specified using APG Data Inversion Bank Select Functions.
- The two X/Y address inputs selected for Background Bank-A may be used directly as inversion control bits for Data Inversion Bank-A. Each bit has an independent enable (see APG Data Inversion Enable Functions). These two inputs are also routed to the Background Bank-A function logic which can perform logical AND, OR or XOR operations on the two inputs to generate a 3rd inversion input to Data Inversion Bank-A, which also has its own enable signal.
- Similarly, The two X/Y address inputs selected for Background Bank-B may be used directly as inversion control bits for Data Inversion Bank-B. Each bit has an independent enable. These two inputs are also routed to the Background Bank-B function logic which can perform logical AND, OR or XOR operations on the two inputs to generate a 3rd inversion input to the Background Bank-B, which also has its own enable signal.
- The APG Bit-2 Data Inversion Function portion of the Background data inversion logic is not duplicated for Background Bank-A or Background Bank-B. Instead, as indicated above, both inputs to the Background Bank-A and Background Bank-B logic are available as direct data inversion controls.
- As indicated above, each of the outputs from Background Bank-A and Background Bank-B have independent enables. During the initial program load these are set to the disable state, for compatibility with the Maverick-I/-II which do not include this logic. The system software does not otherwise change the state of these enables. See APG Data Inversion Enable Functions.

- For compatibility with the legacy APG Background Data Inversion Functions (`bckfen()`) and `bit2fen()`) the X/Y address input selection argument values use two enumerated types: `TesterFunc` (X/Y address selection) and `TesterBGFunc` (X/Y address complement selection). This results in four permutations of the functions below, to allow any combination of both data types.
- The `BckOperation` enumerated type is used to specify the desired logic operation to be performed using the two selected inputs. The functions below can only use the `and`, `or`, `xor` values or `t_bg_off` to disable the logic operation (which is more properly done using the APG Data Inversion Enable Functions).
- The versions of the functions below which include the `Board` argument are only useful when Sites-per-Controller > 1.
- In Multi-DUT Test Programs the operation of these functions is not affected by Active DUTs Set (ADS) or Ignored DUTs Set (IDS).

Usage

The following functions select the two inputs for the Background Bank-A and specify the logic operation to be performed.

```
void apg_datainv_A_func_set(BckOperation Operation,
                            TesterFunc bit1,
                            TesterFunc bit2 );
void apg_datainv_A_func_set(BckOperation Operation,
                            TesterFunc bit1,
                            TesterBGFunc bit2 );
void apg_datainv_A_func_set(BckOperation Operation,
                            TesterBGFunc bit1,
                            TesterFunc bit2 );
void apg_datainv_A_func_set(BckOperation Operation,
                            TesterBGFunc bit1,
                            TesterBGFunc bit2 );
```

The following functions select the two inputs for the Background Bank-B and specify the logic operation to be performed:

```
void apg_datainv_B_func_set(BckOperation Operation,
                            TesterFunc bit1,
                            TesterFunc bit2 );
```

```
void apg_datainv_B_func_set(BckOperation Operation,
                            TesterFunc bit1,
                            TesterBGFunc bit2 );
void apg_datainv_B_func_set(BckOperation Operation,
                            TesterBGFunc bit1,
                            TesterFunc bit2 );
void apg_datainv_B_func_set(BckOperation Operation,
                            TesterBGFunc bit1,
                            TesterBGFunc bit2 );
```

The following functions select the two inputs for the Background Bank-A and specify the logic operation to be performed. These are only useful when Sites-per-Controller > 1:

```
void apg_datainv_A_func_set(HSBBoard Board,
                            BckOperation Operation,
                            TesterFunc bit1,
                            TesterFunc bit2 );
void apg_datainv_A_func_set(HSBBoard Board,
                            BckOperation Operation,
                            TesterFunc bit1,
                            TesterBGFunc bit2 );
void apg_datainv_A_func_set(HSBBoard Board,
                            BckOperation Operation,
                            TesterBGFunc bit1,
                            TesterFunc bit2 );
void apg_datainv_A_func_set(HSBBoard Board,
                            BckOperation Operation,
                            TesterBGFunc bit1,
                            TesterBGFunc bit2 );
```

The following functions select the two inputs for the Background Bank-B and specify the logic operation to be performed. These are only useful when Sites-per-Controller > 1:

```
void apg_datainv_B_func_set(HSBBoard Board,
                            BckOperation Operation,
                            TesterFunc bit1,
                            TesterFunc bit2 );
void apg_datainv_B_func_set(HSBBoard Board,
                            BckOperation Operation,
                            TesterFunc bit1,
                            TesterBGFunc bit2 );
```

```
void apg_datainv_B_func_set(HSBBoard Board,
                            BckOperation Operation,
                            TesterBGFunc bit1,
                            TesterFunc bit2 );
void apg_datainv_B_func_set(HSBBoard Board,
                            BckOperation Operation,
                            TesterBGFunc bit1,
                            TesterBGFunc bit2 );
```

where:

Operation specifies the desired logic operation to be performed using the two selected inputs. Legal values are of the BckOperation enumerated type but only and, or, xor are valid for these functions.

bit1 and **bit2** identify the two inputs, which also become two of the three outputs. Legal values are of the TesterFunc and TesterBGFunc enumerated types (any combination). Using TesterFunc only APG address values are legal (t_x0, t_y2, etc.).

Board is used when Sites-per-Controller > 1 to identify a specific Site Assembly Board (HSB) to be accessed.

Example

```
apg_datainv_A_func_set( and, t_x0, t_y0_bar );
```

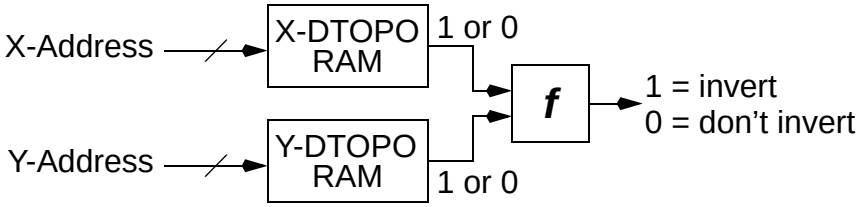
4.12.19 APG Data Topological Inversion (DTOPO) Function

See Data Inversion Logic, Algorithmic Pattern Generator (APG) Configuration.

Description

The dtopo() function is used to configure APG Data Generator topological data inversion options before executing a test pattern which uses the DATGEN DTOPO instruction (see DATGEN Background Function Operands).

The APG Data Generator's DTOPO RAM is used to conditionally invert data as a function of the X/Y address:



The number of X/Y addresses used is based on values set using `numx()` and `numy()`

The X DTOPO RAM is 256Kx1 and the Y DTOP RAM is 64Kx1, each storing one bit for each unique X and Y address. In use, the effective size of each DTOPO RAM is determined by the number of enabled X and Y address bits, set using `numx()` and `numy()`.

During initial program load, both DTOPO RAMs are initialzed to 0; i.e. no DTOPO inversion can occur, regardless of which `dtopo()` operation is selected, and regardless whether the pattern uses the `DATGEN DTOPO` operand or not. The system software does not otherwise modify the DTOPO RAM.

The DTOPO RAMs are loaded via user C code, see APG Data TOPO RAM Load Functions. Then, as the test pattern executes, in any pattern instruction containing the `DATGEN DTOPO` instruction, for each unique X and/or Y address the output of the corresponding DTOPO RAM will be either logic-0 or logic-1. These are the inputs to the DTOPO function block shown above.

The APG Data Inversion Enable Functions and APG Data Inversion Bank Select Functions do affect the use and operation of DTOPO inversion.

In hardware, the DTOPO functional block provides a selectable logic operation, set using the `dtopo()` function. The following table describes the available inversion operations:

Table 4.12.19.0-1 DTOPO Logical Operation Options

DTopoFunc	Description
`xdtopo`	Invert data when the X-DTOPO output is logic-1. The Y-DTOPO RAM output has no effect.
`ydtopo`	Invert data when the Y-DTOPO output is logic-1. The X-DTOPO RAM output has no effect.

Table 4.12.19.0-1 DTOPO Logical Operation Options *(Continued)*

DTopoFunc	Description
xd_and_yd	Invert data when the logical AND of the X-DTOPO output with the Y-DTOPO output is logic-1.
xd_or_yd	Invert data when the logical OR of the X-DTOPO output with the Y-DTOPO output is logic-1.
xd_xor_yd	Invert data when the logical XOR (exclusive OR) of the X-DTOPO output with the Y-DTOPO output is logic-1.

The output of the DTOPO logic is a single invert bit: a logic-1 causes the data to be inverted, logic-0 does not invert data. APG Data Generator outputs are affected. The Data Inversion Logic is as part of the APG Data Generator.

In Multi-DUT Test Programs the operation of these functions is not affected by Active DUTs Set (ADS) or Ignored DUTs Set (IDS).

Usage

```
void dtopo( DTopoFunc Function );
```

where:

Function specifies the desired DTOPO logic operation. Legal values are of the DTopoFunc enumerated type. See DTOPO Logical Operation Options above.

Example

The following example sets the DTOPO logic function to output a logic-1 (i.e. invert APG data) when the output of the X-DTOPO XOR'ed with the output of the Y-DTOPO is a logic-1.

```
dtopo( xd_xor_yd );
```

Also see Example.

4.12.20 APG Data TOPO RAM Load Functions

See Data Inversion Logic, Algorithmic Pattern Generator (APG) Configuration, APG Data Topological Inversion (DTOPO) Function.

Description

The `x_dtopo()`, `y_dtopo()` and `xy_dtopo()` functions are used to load data into the APG's X-DTOPO RAM and/or Y-DTOPO RAM. See APG Data Topological Inversion (DTOPO) Function. Note the following:

- During initial program load, both DTOPO RAMs are initialzed to 0; i.e. no DTOPO inversion can occur, regardless of which function is selected using the functions below, and regardless whether the pattern uses `DATGEN DTOPO` or not. The system software does not otherwise modify the DTOPO RAM.
- Each function execution writes one location of the X-DTOPO RAM, Y-DTOPO RAM, or both. It is common to use these functions in a loop to load the TOPO memory based on an algorithm.
- Only the LSB (1 bit) of each data value is actually used.
- In Multi-DUT Test Programs the operation of these functions is not affected by Active DUTs Set (ADS) or Ignored DUTs Set (IDS).

Usage

The following function writes `XData` to the X-DTOPO RAM and `YDATA` to the Y-DTOPO RAM:

```
void xy_dtopo( int Addr, int XData, int YData );
```

The following function writes `Data` only to the X-DTOPO RAM:

```
void x_dtopo( int Addr, int Data );
```

The following function writes `Data` only to the Y-DTOPO RAM:

```
void y_dtopo( int Addr, int Data );
```

where:

`Addr` is the DTOPO RAM address to be written. This corresponds to the X/Y address which accesses the DTOPO RAM during pattern execution.

`XData` and **`YData`** are the data to be written to that location. Only the LSB (1 bit) is actually used.

Example

The following code loads the X/Y DTOPO RAMs to create the example shown below the code:

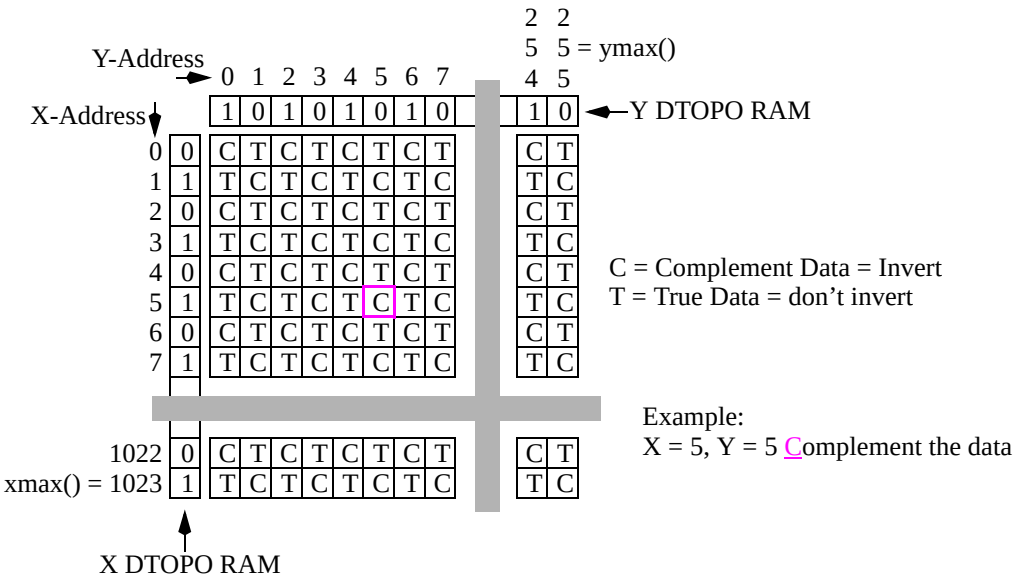
```
int data;
```

```
// Load X-DTOPO RAM
for (int xaddr= 0; xaddr < xmax(); xaddr++) {
   if ( xaddr %2 )     data = 0;
   else                data = 1;
   if( (xaddr / 4) %2) data = ~data & 0x1;
   if( (xaddr / 8) %2) data = ~data & 0x1;
   x_dtopo( xaddr, data );
}
// Load Y-DTOPO
for( int yaddr = 0; yaddr < ymax(); yaddr++) {
   if( yaddr %2 ) y_dtopo( yaddr, 1 );
   else           y_dtopo( yaddr, 0 );
}
```

The diagram below shows how this example might operate, given the following configuration:

- `numx()` = 10. This sets the X-DTOPO sized to 2^{10}; i.e. 0x3FF max.
- The code above initializes the X-DTOPO such that odd X-addresses will output a logic-1.
- `numy()` = 8. This sets the Y-DTOPO sized to 2^{8}; i.e. 0xFF max.
- The code above initializes the Y-DTOPO such that even Y-addresses will output a logic-1.
- The DTOPO function is set to `xd_xor_yd`.

- The cells at the intersection of each X/Y address indicate the resulting inversion states, where the character C is used to indicate inversion (i.e. complement the data) and T is used to indicate no inversion (i.e. true data):



4.12.21 Logical vs. Physical, vs. Electrical Addresses

See Algorithmic Pattern Generator (APG) Configuration, Address TOPO RAM.

The terminology used when discussing memory testing has several decades of tradition. However, the terminology sometimes gets confused when discussing the details of a*ddresses* in the context of memory testing.

Before going into all the details, the good news is that the system software and hardware consistently use only *logical* addresses, with any *physical* address manipulations performed by the Address TOPO RAM(see APG Address Topo RAM Load Functions). This means that, except during the actual hardware's access of the DUT, all discussion of *addressing* is consistent. If the reasoning behind this is already understood, the rest of this section can be skipped.

The rest of this section discusses the distinction between the three commonly used terms: *logical* address, *physical* address, and *electrical* address.

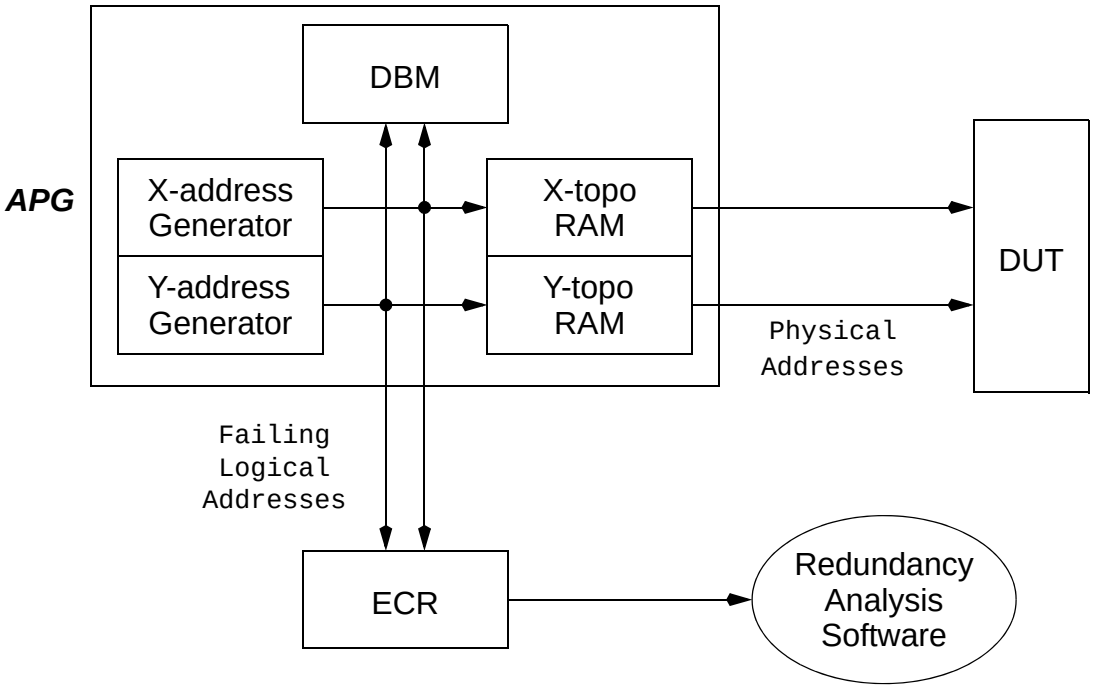
Table 4.12.21.0-1 Address Terminology

Term	Description
Logical Address	Symbolic addresses used to simplify pattern creation, ECR and DBM access, redundancy analysis, etc. Logical address row-0/column-0 symbolically represents the memory cell at the upper-left corner of the DUT die, with row addresses incrementing linearly downwards, and column addresses incrementing linearly to the right. However, by device design, applying row-0/column-0 addresses to the DUT pins may NOT access the physical upper-left corner of the die, and incrementing row/column address signals at the DUT may not linearly access adjacent physical memory elements within the DUT. Topological address scrambling hardware is used to convert logical addresses into physical addresses (see APG Address Topo RAM Load Functions).
Physical Address	The actual address of a given memory element relative to the physical layout of the die. To access the memory element physically at the upper-left corner of the die (physical row-0/ column-0, as seen under a microscope) may require applying an address other than row-0/column-0 at the DUT pins. And, to access adjacent physical addresses may require applying non-adjacent logical row and/or column addresses to the DUT pins.
Electrical Address	A single address value (for example 0x5AF9) representing a single memory element. Consistent with the device data sheet's An..A0 context. To convert an electrical address to logical or physical address requires identifying which electrical address bits are mapped to each row and column address bit. Not commonly used in testing, although failure analysis (FA) of customer returns often requires conversion to logical or physical address.

Note: the term *topological* address is not defined here. It is inherently ambiguous and not used in this document

The X/Y (row/column) addresses generated by the APG Address Generator (prior to entering the Address TOPO RAM) are *logical* addresses. This makes it practical to write very complex test patterns without worrying about *physical* address conversion - let the Address Topological RAM do the conversions.

The failing address information logged to the ECR is also *logical* addresses. And since the redundancy repair analysis software uses ECR information it is also dealing only with *logical* address values.



During functional testing, it is *critical* to correctly convert logical addresses to physical addresses. Without this conversion, the desired fault detection expected from standard Memory Test Patterns will not occur. Logical to physical address conversion is performed in hardware, using the Address TOPO RAM. Properly configured (see APG Address Topo RAM Load Functions), the address TOPO RAM allows the user to view test patterns, DBM addresses, ECR failing address data, and redundancy repair information consistently, using the *logical* address view. Then, during testing, as the pattern executes, the TOPO RAM automatically maps the *logical* pattern addresses to *physical* device addresses, to obtain the desired fault detection.

When performing the actual on-line redundancy repair, proper operation requires that the correct *physical* addresses be applied to the device. After all, it is the replacement of a physically defective memory element (row, column, etc.) with a spare element which repairs the device. Thus, when redundancy repair is performed on the test system, the user must

confirm that the address topological scrambling used to *test the device* is also correct to *repair the device*. And, if not, must correctly configure the Address TOPO RAM for each application.

4.12.22 APG Address Topo RAM Load Functions

See Algorithmic Pattern Generator (APG) Configuration.

Description

The `xtopo()`, `ytopo()` and `xytopo()` functions are used to load the Address TOPO RAMs on the X and/or Y APG Address Generator(s).

Address TOPO RAMs provide address topological scrambling; i.e. the translation of logical address values, as generated by the APG, into the topological addresses needed to correctly test the DUT.

Each function execution writes one location of the X-TOPO RAM, Y-TOPO RAM, or both. It is common to use these functions in a loop to load the TOPO memory based on an algorithm.

Only the addresses enabled by `numx()` and `numy()` are written, regardless of the input addresses specified. For example:

```
numx( 16 );
xtopo( 0x0, 0xFFFF );
output( " XTopo @ Address 0 => 0x%x", xtopo( 0x0 ) );
```

This outputs 0xffff. Then...

```
numx( 10 );
xtopo( 0x0, 0x0 );
output( " XTopo @ Address 0 => 0x%x", xtopo( 0x0 ) );
```

This outputs 0xfc00. Note that only the low 10 address bits were modified.

Usage

The following function writes one value to both the X and Y Address TOPO RAMs:

```
void xytopo( int in_xaddr,
             int in_yaddr,
             int out_xaddr,
             int out_yaddr );
```

The following function writes one value to the X Address TOPO RAM:

```
void xtopo( int in_addr, int out_addr );
```

The following function writes one value to the Y Address TOPO RAM:

```
void ytopo( int in_addr, int out_addr );
```

The following functions writes one or more values to the X or Y Address TOPO RAM:

```
void xtopo( int addr, int len, int* values );
void ytopo( int addr, int len, int* values);
```

The following function returns the topologically scrambled address for the specified input address.:

```
int xtopo( int in_addr );
int ytopo( int in_addr );
```

The following functions are used to get one or more value(s) from the X or Y Address TOPO RAM:

```
BOOL xtopo( int addr, int len, int* values, int* count );
BOOL ytopo( int addr, int len, int* values, int* count );
```

where:

`in_xaddr` and **`in_yaddr`** are TOPO RAM address to be accessed.

`out_xaddr` and **`out_yaddr`** specify the data to be written to **`in_xaddr`** or **`in_yaddr`**.

`addr` specifies the first address to be written or read.

`len` specifies the number of values to be written or read.

`values` is used in two contexts:

- In the setter function, **`values`** is an array of **`len`** values to be written to the X or Y Address TOPO RAM, starting at **`addr`**.
- In the getter function, **`values`** is a pointer to an existing `int` array, of **`len`** or greater size, used to return **`len`** values from the X or Y Address TOPO RAM, starting from **`addr`**.

`count` is a pointer to an existing `int` variable used to return the number of valid **`values`**.

The versions of `xtopo()` and `xtopo()` which return an `int`, return the value read from the Xor Y Address TOPO RAM.

The versions of `xtopo()` and `xtopo()` which return a `BOOL`, return FALSE if an error is detected, otherwise TRUE is returned.

Example

The following example loads both the X and Y Address TOPO RAMs with a simple mapping that inverts the address; i.e. logical 0 = topological 65535, 1 = 65534, etc. The example assumes there are 16 X and 16 Y addresses; most devices will use smaller values, and will have more complex scrambling algorithms.

```
for ( i = 0 ; i < 65536; i++ )
   xtopo( i, 65535 - i );
for ( i = 0 ; i < 65536; i++ )
   ytopo( i, 65535 - i );
```

4.12.23 APG Timer Functions

See Algorithmic Pattern Generator (APG) Configuration, APG Interrupt Timer.

Description

The `timer()` function is used to program the APG Interrupt Timer.

Due to the APG pipeline architecture, the effective timer value is less than the programmed value by:

- The sum of the 8 cycle periods executed prior to any pattern instruction containing the `MAR TIMEN` (Data Buffer Memory (DBM) not used).
- The sum of the 20 cycle periods executed prior to any pattern instruction containing the `MAR TIMEN` (DBM used).

This reduction can be addressed by increasing the programmed timer value by the appropriate amount.

The interrupt timer is set to 10uS during initial program load but not otherwise modified by the system software.

If the `timer()` function is used in Pattern Initial Conditions it must be the last function specified. For example:

```
@{
   ymain( 0 );
   count( 1, amax() );
   // Etc.
   timer( value ); // Must be last
@}
```

Usage

```
void timer( double Value );
```

where:

`Value` specifies the desired interrupt timer value. Units are supported, see Specifying Units. Legal values range from 100nS to 1.6 seconds. Resolution is 100nS. Values which are not in increments of 100nS are silently rounded up to the next higher 100nS value.

Examples

Set the APG Interrupt Timer to 10 mS:

```
timer( 10 MS );
```

Set the APG Interrupt Timer to 4 uS:

```
timer( 4 US );
```

This sets the APG Interrupt Timer to 200 nS. The value is rounded up from 140nS to 200nS as noted above:

```
timer( 140 NS );
```

4.13 Memory Test Patterns

See Test Pattern Programming.

This section includes the following:

- Overview
- Memory Pattern Instruction Format
- Default Memory Pattern Instruction
- APG Instruction Execution
- APG Address Generator Overview
- YALU Instruction and XALU Instruction
  - YALU/XALU SourceA/SourceB Operands
  - YALU/XALU Carry/Borrow Operands
  - YALU/XALU Function Operands
  - YALU/XALU Destination Operands
  - YALU/XALU Addressout Operands
- COUNT Instruction
  - COUNT Counter Operands
  - COUNT Function Operands
  - COUNT Autoreload Operands
- MAR Instruction
  - MAR Default Pattern Instruction
  - MAR Branch Condition Operands
  - Error Pipeline Requirements
  - MAR Address Operand
  - MAR Strobe Control Operands
  - MAR Interrupt Operands
  - MAR Timer Operands
  - MAR Misc Operands
  - MAR BOE Type Operands
  - MAR Error-choice Operands
  - Static Error Choice Functions, Branch-on-error
  - DUT-pin to Tester-pin Connection Requirements
- CHIPS Instruction
  - CHIPS Chip-select-control Operands
  - CHIPS Misc Operands

4.13.1 Overview

See Memory Test Patterns.

The Magnum 1 test system contains two sources of test pattern data:

- Algorithmic Pattern Generator (APG) when executing Memory Test Patterns.
- Combined Logic Vector Memory (LVM) / Scan Vector Memory (SVM) for stored Logic Test Patterns and Scan Test Patterns.

The diagram below shows key Magnum 1 architecture features:

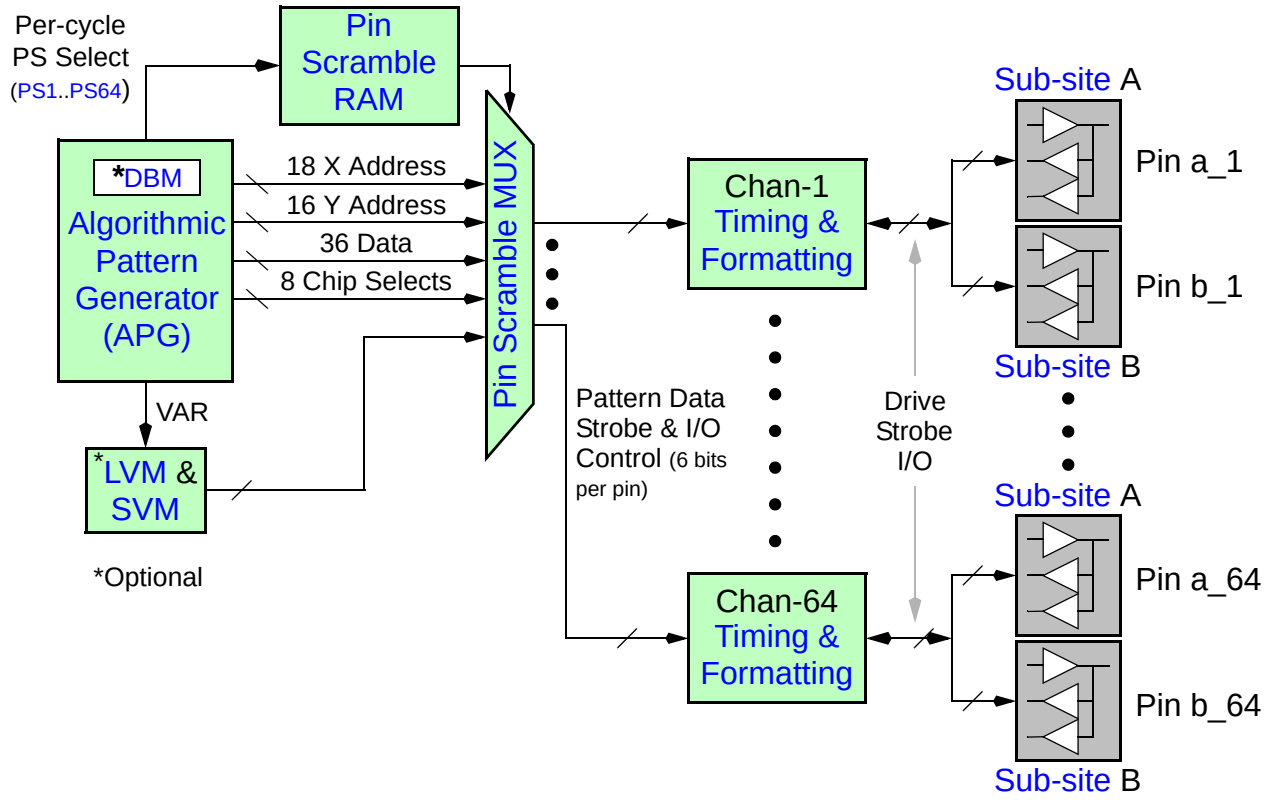


Figure-66: Test Pattern Data Source Hardware Architecture

Using the Pin Scramble MUX, the source of pattern data for each timing channel can be selected on a per-channel/per-cycle basis, at full tester speed. In other words, for any given pin-pair, the source of pattern data can be selected on a per-cycle basis from any APG address, data, or chip select data bit, or one LVM/SVM data bit (2 bits in Double Data Rate (DDR) Mode). This section discusses Memory Test Patterns, also see Logic Test Patterns and Mixed Memory/Logic Patterns.

4.13.2 Memory Pattern Instruction Format

See Memory Test Patterns.

A Memory Test Pattern consists of one or more pattern instructions, written by the user, to control the Algorithmic Pattern Generator (APG), which is the tester hardware which

algorithmically generates binary pattern data repesenting X/Y addresses, data, and chip selects (clocks, enables, etc.) used to functionally test memory devices.

Memory pattern instructions are written using a proprietary language which consists of a set of one or more sub-instructions and associated operand values. Each sub-instruction controls a specific section of the APG hardware.

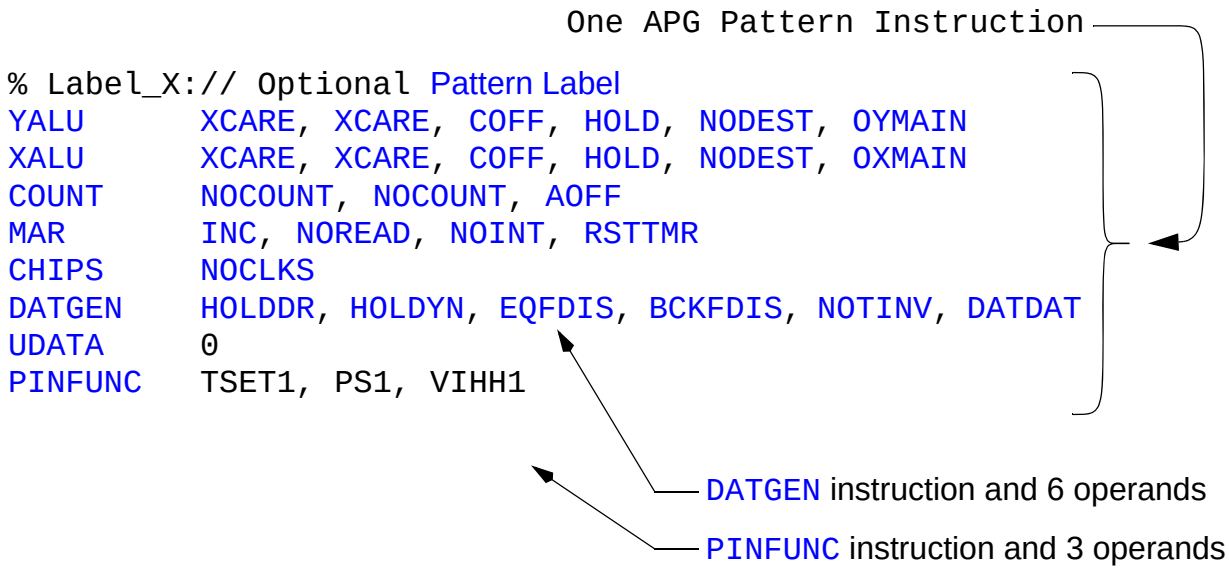
A sample APG instruction is shown below.

Note: on 5/2/03, terminology was changed and standardized, to use the terms:
- *APG instruction*, to refer to all statements between two % (see Pattern Instruction Identifier (%)).
- *Instruction*, to refer to each line (sub-instruction) within an APG instruction (i.e. YALU, CHIPS, UDATA, etc.).
- *Operand* to refer to the parameters specified for each *instruction*.

And, the use of the term *microinstruction* was eliminated.

Every APG instruction starts with a percent sign (%). See Pattern Instruction Identifier (%). Then, on each line, the first token (YALU, XALU, etc.) identifies a sub-section of the APG to be programmed by that portion of the APG instruction. The token(s) to the right of the first are the operand values that control the sub-section of that hardware.

Example



This example encodes the Default Memory Pattern Instruction. Each APG instruction may contain any or all of the sub-instructions (YALU, XALU, COUNT, MAR, CHIPS, DATGEN,

UDATA, and PINFUNC) and they may occur in any order. Only one sub-instruction per line is allowed. The pattern compiler will apply values from the Default Memory Pattern Instruction for any sub-instructions or operands that are omitted.

4.13.3 Default Memory Pattern Instruction

See Memory Test Patterns.

The default memory pattern instruction is a complete APG instruction which explicitly sets the default operands as shown below:

```
% YALU      XCARE, XCARE, COFF, HOLD, NODEST, OYMAIN
  XALU      XCARE, XCARE, COFF, HOLD, NODEST, OXMAIN
  COUNT     NOCOUNT, NOCOUNT, AOFF
  MAR       INC, NOREAD, NOINT, RSTTMR
  CHIPS     NOCLKS
  DATGEN    HOLDDR, HOLDYN, EQFDIS, BCKFDIS, NOTINV, DATDAT
  UDATA     0
  PINFUNC   TSET1, PS1, VIHH1
```

These operands are implicitly added in any APG instruction which does not explicitly include a corresponding value. This allows a given APG instruction to only include those instructions/operands which differ from the default values. For example, the following instruction causes the same result as the default example above:

```
% MAR   INC
```

Using Magnum 1/2/2x, the following operands are NOT enabled by default:

```
MAR       VCNTR
PINFUNC   VPS, VTSET, VVIHH, VVPULSE, VVCOMP,
          VLATCHRESET, VOVER
```

4.13.4 APG Instruction Execution

See Memory Test Patterns.

The APG is a digital-logic state machine, controlled using the pattern language documented in this section. The following rules specify how APG instructions execute:

1. Address generator (XALU, YALU), data generator (DATGEN), chip select (CHIPS), counter (COUNT), PINFUNC, UDATA and MAR instructions affect the tester cycle executing the instruction.
2. APG counter auto-reload operation is described in COUNT Counter Operands.
3. Pattern instruction execution sequence control has three levels of execution priority, based on the MAR instruction operands used in a given instruction. Listed from highest to lowest priority:
   - Subroutine call (GOSUB), including conditional subroutine calls (CSUBE, CSUBNE, CSUBT, etc.) when the specified condition = TRUE.
   - Interrupt (INTEN, INTENADR), if an APG Interrupt Timer interrupt is pending at the start of the instruction. Other, including INC, JUMP, RETURN, DONE, PAUSE and all conditional jump/return options (CJMPE, CJMPNZ, CRETNE, CRETZ, etc.).

Regarding 3., the following rules apply:

- The higher priority option takes precedence. This rule only makes sense in instructions which contain both an interrupt enable (INTEN, INTENADR) plus another branch option (since all other options are all mutually exclusive).
- If an instruction contains both a [conditional] subroutine call and an interrupt enable and the subroutine does execute the interrupt is ignored until a later instruction which contains INTEN or INTENADR without a subroutine call.
- In an instruction which includes both an interrupt enable and another jump/return option, if execution does call the interrupt subroutine execution will return to the target of the other jump/return instruction. When the other jump option is conditional, the target is the result of the conditional evaluation.

For example, given the following pattern:

```
PATTERN( myPat, memory )
%  L1:
   MAR        INC, INTADR          // Interrupt address setup
   UDATA      L5
%  L2: MAR    INC, TIMEN           // Enable timer
   COUNT      COUNT1, COUNTUDATA   // Load UDATA into Counter-1
   UDATA      1
%  L3:
   COUNT      COUNT1, DECR
   MAR        INTEN, TIMEN, CJMPNZ, L3 // Enable timer and interrupt
%  L4: MAR    DONE
```

```
% L5: MAR    RETURN
```

Below, the instruction labels are used to describe execution order. The execution order depends upon whether a timer interrupt is set in instruction L3:

- No interrupt: L1, L2, L3, L3, L4
- Timer Interrupt is set most of a tester cycle prior to the first execution of L3 so an interrupt is pending at the start of L3:
  L1, L2, L3, L5, L3, L4
- Timer Interrupt is set most of a tester cycle before the end of the first execution of L3 so an interrupt is pending at the start of the 2nd execution of L3:
  L1, L2, L3, L3, L5, L4
- Timer Interrupt is set during the second execution of L3 so no interrupt is pending during this instruction:
  L1, L2, L3, L3, L4

In each case the interrupt return address is the instruction that would have been executed next if the interrupt hadn't happened. And in each case L3 is executed twice and Counter-1 is decremented twice.

Examining the 2rd case in more detail, at the start of the first execution of L3 Counter-1 = 1 thus CJMPNZ is TRUE and the tentative next instruction is L3. However, most of a tester cycle before the the first execution of L3 the interrupt timer reaches 0, thus before the start of execution of L3 since the interrupt is TRUE the interrupt subroutine (L5) will execute next. The interrupt subroutine returns to L3 because it was the next instruction identified before the interrupt subroutine executed. So, L3 executes a second time and since Counter-1 is now = 0 (decremented from 1 to 0 in the first L3) the CJMPNZ = FALSE and execution increments to L4. Note that Counter-1 is decremented to -1 after it is tested by the CJMPNZ.

4.13.5 APG Address Generator Overview

See Algorithmic Pattern Generator (APG), Memory Test Patterns.

To test memory devices, an APG must be able to generate complex address sequences.

The APG has two independent but interconnected APG Address Generators:

- X address generator; i.e. typically row addresses
- Y address generator; i.e. typically column addresses

In use, the intersection of a row address and a column address determines which DUT memory cell(s) will be read or written. The dual X/Y address generation architecture provides flexibility in writing Memory Test Patterns.

The heart of each APG Address Generator is an Arithmetic Logic Unit, or ALUALU. In each memory pattern cycle, the ALU takes one or two inputs, identified as SourceA and SourceB, selected from 3 16-bit address register(s) called MAIN, BASE and FIELD, or from UDATA (an arbitrary value for each instruction stored in the APG's uRAM), as well as one Carry/Borrow input from the *other* ALU (X carries to Y, Y carries to X). In each testeer cycle the ALU operates on these inputs performing a user specified function. The result of the ALU's operation is passed through a mask and then placed back in one or more of the 3 address registers. In each pattern cycle, the output of one X address register and one Y address register is selected as the address output from the APG to the DUT in the current cycle. All of these operations are controlled from the test pattern using the YALU Instruction (YALU) and XALU Instruction (XALU). The XALU instruction controls the X address generator, and the YALU instruction controls the Y address generator.

4.13.6 YALU Instruction

See APG Address Generator, Memory Test Patterns, Memory Pattern Instruction Format.

Description

The YALU instruction is used to control the APG's Y address generator. All operand options are summarized in YALU Instruction Operands.

The YALU instruction takes the following form:

```
YALU SourceA, SourceB, Carry/Borrow, Function, Destination(s),
Addressout
```

Rules:

- The YALU instruction may either be completely omitted (the Default Memory Pattern Instruction for YALU is applied) or the first four fields must all be specified; i.e. SourceA, SourceB, Carry/Borrow, and Function. The last two fields (Destination(s) and Addressout), are optional.
- Each YALU instruction specifying the first four operands must include exactly one value for each of the first four operand fields (SourceA, SourceB, Carry/Borrow, and Function)

- Up to three Destination operand values may be specified, identifying any combination of MAIN (DYMAIN), BASE (DYBASE), and/or FIELD (DYFIELD) address registers to be updated with the ALU output.
- If no operand for the Destination field is specified, the default is NODEST, which has the effect of discarding any operation performed by the ALU, and making the values specified in the first four fields insignificant.
- It is not legal to mix NODEST with any other Destination operand value.
- If no operand for the Addressout field is specified, the default destination is the Y MAIN register (OYMAIN).
- When the Function field contains a single-source instruction; i.e. ALL1S, COMP, DECREMENT, DOUBLE, HOLD, INCREMENT, ZERO, the SourceB field must be XCARE.

The default YALU instruction is:

```
YALU    XCARE, XCARE, COFF, HOLD, NODEST, OYMAIN
```

Usage

```
YALU SourceA, SourceB, Carry/Borrow, Function, Destination(s),
Addressout
```

where:

YALU identifes that the Y Address Generator is being programmed.

SourceA specifies the primary input source to the ALU. See YALU/XALU SourceA/SourceB Operands. Default = XCARE.

SourceB specifies the secondary input source to the ALU. See YALU/XALU SourceA/SourceB Operands. Default = XCARE.

Carry/Borrow specifies the carry/borrow option. The state of the carry/borrow bit affects some YALU function operations, see YALU/XALU Carry/Borrow Operands and YALU/XALU Function Operands. Default = COFF.

Function specifies the operation the ALU will perform in the current instruction. See YALU/XALU Function Operands. Default = HOLD.

Destination(s) specifies one or more destination register(s) to be updated (written) with the ALU output in the current instruction. Multiple destinations must be separated by commas. See YALU/XALU Destination(s) Operands. Default = NODEST.

Addressout specifies which Y address register is output as the Y address to the pin scramble MUX in the current instruction. See YALU/XALU Addressout Operands. Default = OYMAIN.

The table below summarizes the operands for each field of the YALU instruction. Default values are indicated using (**D**).

Table 4.13.6.0-1 YALU Instruction Operands

SourceA	SourceB	Carry Borrow	Function	Destination(s)	Addressout
XCARE(**D**) YBASE YFIELD YMAIN YUDATA	XCARE(**D**) YBASE YFIELD YMAIN YUDATA	BBEQMAX BBEQMIN BBNEMAX BBNEMIN BFEQMAX BFEQMIN BFNEMAX BFNEMIN BMEQMAX BMEQMIN BMNEMAX BMNEMIN BOFF BON CBEQMAX CBEQMIN CBNEMAX CBNEMIN CMEQMAX CFEQMAX CFEQMIN CFNEMIN CMNEMAX CFNEMAX CMEQMIN CMNEMIN COFF(**D**) CON	AANDBBAR AND ADD ALL1S ANANDBBAR ANORBBAR AORBBAR COMP DECREMENT DOUBLE HOLD(**D**) INCREMENT NAND NOR OR SUBTRACT XNOR XOR ZERO	DYBASE DYFIELD DYMAIN NODEST(**D**)	OYBASE OYFIELD OYMAIN(**D**)

4.13.7 XALU Instruction

See APG Address Generator, Memory Test Patterns, Memory Pattern Instruction Format.

Description

The XALU instruction is used to control the APG X address generator. All operand options are summarized in XALU Instruction Operands.

Rules:

- The XALU instruction may either be completely omitted (the Default Memory Pattern Instruction for XALU are applied) or the first four fields must be specified; i.e. SourceA, SourceB, Carry/Borrow, and Function. The last two fields (Destination(s) and Addressout), are optional.
- Each XALU instruction specifying the first four operands must include exactly one value for each of the first four operand fields (SourceA, SourceB, Carry/Borrow, and Function)
- Up to three Destination operand values may be specified, identifying any combination of MAIN (DXMAIN), BASE (DXBASE), and/or FIELD (DXFIELD) address registers to be updated with the ALU output.
- If no operand for the Destination field is specified, the default is NODEST, which has the effect of discarding any operation performed by the ALU, and making the values specified in the first four fields insignificant.
- It is not legal to mix NODEST with any other Destination operand value.
- If no operand for the Addressout field is specified, the default destination is the X MAIN register (OXMAIN).
- When the Function field contains a single-source instruction; i.e. ALL1S, COMP, DECREMENT, DOUBLE, HOLD, INCREMENT, ZERO, the SourceB field must be XCARE.

The default XALU instruction is:

```
XALU    XCARE, XCARE, COFF, HOLD, NODEST, OXMAIN
```

Usage

```
XALU SourceA, SourceB, Carry/Borrow, Function, Destination(s),
Addressout
```

where:

XALU identifes that the X Address Generator is being programmed.

SourceA specifies the primary input source to the ALU. See YALU/XALU SourceA/SourceB Operands. Default = XCARE.

SourceB specifies the secondary input source to the ALU. See YALU/XALU SourceA/SourceB Operands. Default = XCARE.

Carry/Borrow specifies the carry/borrow option. The state of the carry/borrow bit affects some YALU function operations, see YALU/XALU Carry/Borrow Operands and YALU/XALU Function Operands. Default = COFF.

Function specifies the operation the ALU will perform in the current instruction. See YALU/XALU Function Operands. Default = HOLD.

Destination(s) specifies one or more destination register(s) to be updated (written) with the ALU output in the current instruction. Multiple destinations must be separated by commas. See YALU/XALU Destination(s) Operands. Default = NODEST.

Addressout specifies which X address register (MAIN, BASE, or FIELD) is output as the X address to the pin scramble MUX in the current instruction. See YALU/XALU Addressout Operands. Default = OYMAIN.

The table below summarizes the operands for each field of the XALU instruction. Default values are indicated using (**D**):

Table 4.13.7.0-1 XALU Instruction Operands

SourceA	SourceB	Carry Borrow	Function	Destination(s)	Addressout
XBASE	XBASE	BBEQMAX	AANDBBAR	DXBASE	OXBASE
XFIELD	XFIELD	BBEQMIN	AND	DXFIELD	OXFIELD
XMAIN	XMAIN	BBNEMAX	ADD	DXMAIN	OXMAIN(**D**)
XCARE(**D**)	XCARE(**D**)	BBNEMIN	ALL1S	NODEST(**D**)	
XUDATA	XUDATA	BFEQMAX	ANANDBBAR		
		BFEQMIN	ANORBBAR		
		BFNEMAX	AORBBAR		
		BFNEMIN	COMP		
		BMEQMAX	DECREMENT		
		BMEQMIN	DOUBLE		
		BMNEMAX	HOLD(**D**)		
		BMNEMIN	INCREMENT		
		BOFF	NAND		
		BON	NOR		
		CBEQMAX	OR		
		CBEQMIN	SUBTRACT		
		CBNEMAX	XNOR		
		CBNEMIN	XOR		
		CMEQMAX	ZERO		
		CFEQMAX			
		CFEQMIN			
		CFNEMIN			
		CMNEMAX			
		CFNEMAX			
		CMEQMIN			
		CMNEMIN			
		COFF(**D**)			
		CON			

4.13.7.1 YALU/XALU SourceA/SourceB Operands

See APG Address Generator, Memory Test Patterns, XALU Instruction, YALU Instruction.

Description

The YALU and XALU instructions take the following form:

```
YALU SourceA, SourceB, Carry/Borrow, Function, Destination(s),
Addressout

XALU SourceA, SourceB, Carry/Borrow, Function, Destination(s),
Addressout
```

The SourceA and SourceB operands select the input(s) to the ALU of the X or Y Address Generator in the current instruction.

Rules:

- When the Function field contains a single source instruction; i.e. ALL1S, COMP, DECREMENT, DOUBLE, HOLD, INCREMENT, ZERO, the SourceB field must be XCARE.

The table below describes the options available for SourceA and SourceB operands to XALU and YALU:

Table 4.13.7.1-1 Y-address Generator SourceA & SourceB Operands

Operand	Purpose
YBASE	Select the BASE address register as input to the Y ALU.
YFIELD	Select the FIELD address register as input to the Y ALU.
YMAIN	Select the MAIN address register as input to the Y ALU.
XCARE	No Y ALU input is selected (default).
YUDATA	Select bits 0-15 of the UDATA value as input to the Y ALU.

Table 4.13.7.1-2 X-address Generator SourceA & SourceB Operands

Operand	Purpose
XBASE	Select the BASE address register as input to the X ALU.
XFIELD	Select the FIELD address register as input to the X ALU.
XMAIN	Select the MAIN address register as input to the X ALU.
XCARE	No X input is selected (default).
XUDATA	Select bits 16-31 of the UDATA value as input to the X ALU.

Note: the XCARE operand is used in both XALU and YALU instructions. The "X" means don't (as in don't care); i.e. it does not refer to the X or Y address generator.

Example

In the following example, the SourceA input for the Y ALU is the Y MAIN address register. The SourceB input for the Y ALU is the first 16 bits of UDATA (.15). The two inputs to the ALU are ADD'ed and the result is placed into the Y FIELD register (DYFIELD). The contents of the Y MAIN register remain unchanged and are output as the APG Y address.

```
% YALU      YMAIN, YUDATA, COFF, ADD, DYFIELD, OYMAIN
  XALU      XFIELD, XCARE, COFF, HOLD, NODEST, OXFIELD
  UDATA     .15 // Same as 0x15
```

The SourceA input for the X ALU is the X FIELD address register, and there is no SourceB specified (XCARE). The contents of the X FIELD register remain unchanged and are output as the APG X address. Since the ALU function is HOLD, the ALU outputs the SourceA input without modification. The address output from the X address generator comes from the FIELD register (OXFIELD).

Note: the *dot* notation used in the UDATA value above (.15) can be used to specify a hexadecimal value. So .15 equals the hex value 0x15. The 0x syntax is also legal (and preferred).

4.13.7.2 YALU/XALU Carry/Borrow Operands

See APG Address Generator, Memory Test Patterns, XALU Instruction, YALU Instruction.

Description

The YALU and XALU instructions take the following form:

```
YALU SourceA, SourceB, Carry/Borrow, Function, Destination(s),
Addressout
XALU SourceA, SourceB, Carry/Borrow, Function, Destination(s),
Addressout
```

The Carry/Borrow operand determines how the carry (or borrow) input to an address generator's ALU is used.

The operation of the YALU/XALU INCREMENT and DECREMENT functions depends upon whether the carry or borrow input is TRUE or FALSE. For example, the INCREMENT function is defined as:

```
INCREMENT : ALU Output = (SourceA + Carry)
DECREMENT : ALU Output = (SourceA - Carry)
```

The DECREMENT function will subtract 1 from the SourceA input if the carry input is FALSE. To simplify things, the borrow input is defined to be TRUE when the carry input is FALSE. Thus, DECREMENT can also be defined as:

```
DECREMENT : ALU Output = (SourceA - Borrow)
```

Some of the Carry/Borrow operands refer to a specific address register. For example, CMEQMAX means set the carry bit TRUE if the *other* address generator's register is equal to its maximum value.

The table below describes the options available for the `Carry/Borrow` operand to XALU and YALU:

Table 4.13.7.2-1 YALU/XALU Carry/Borrow Operands

Operand	Purpose
CON	Carry On Carry input to ALU is forced TRUE
BON	Borrow On Borrow input to ALU is forced TRUE
COFF	Carry Off Carry input to ALU is forced FALSE (default)
BOFF	Borrow Off Borrow input to ALU is forced FALSE
CMEQMAX	Carry Main Equal Max Carry = TRUE if the *other*[1] address generator's MAIN register is equal to its maximum value (set using `xmax()`, `ymax()` or `amax()`)
BMEQMAX	Borrow Main Equal Max Borrow = TRUE if the *other*[1] address generator's MAIN register is equal to its maximum value (set using `xmax()`, `ymax()` or `amax()`)
CFEQMAX	Carry Field Equal Max Carry = TRUE if the *other*[1] address generator's FIELD register is equal to its maximum value (set using `xmax()`, `ymax()` or `amax()`)
BFEQMAX	Borrow Field Equal Max Borrow = TRUE if the *other*[1] address generator's FIELD register is equal to its maximum value (set using `xmax()`, `ymax()` or `amax()`)
CBEQMAX	Carry Base Equal Max Carry = TRUE if the *other*[1] address generator's BASE register is equal to its maximum value (set using `xmax()`, `ymax()` or `amax()`)
BBEQMAX	Borrow Base Equal Max Borrow = TRUE if the *other*[1] address generator's BASE register is equal to its maximum value (set using `xmax()`, `ymax()` or `amax()`)

Note-1: when used in a YALU instruction, the term *other* in the descriptions above refers to X address registers. When used in an XALU instruction, *other* refers to the Y address registers.

Table 4.13.7.2-1 YALU/XALU Carry/Borrow Operands *(Continued)*

Operand	Purpose
CMNEMAX	Carry Main Not Equal Max Carry = TRUE if the *other*[1] address generator's MAIN register is not equal to its maximum value (set using xmax(), ymax() or amax())
BMNEMAX	Borrow Main Not Equal Max Borrow = TRUE if the *other*[1] address generator's MAIN register is not equal to its maximum value (set using xmax(), ymax() or amax())
CFNEMAX	Carry Field Not Equal Max Carry = TRUE if the *other*[1] address generator's FIELD register is not equal to its maximum value (set using xmax(), ymax() or amax())
BFNEMAX	Borrow Field Not Equal Max Borrow = TRUE if the *other*[1] address generator's FIELD register is not equal to its maximum value (set using xmax(), ymax() or amax())
CBNEMAX	Carry Base Not Equal Max Carry = TRUE if the *other*[1] address generator's BASE register is not equal to its maximum value (set using xmax(), ymax() or amax())
BBNEMAX	Borrow Base Not Equal Max Borrow = TRUE if the *other*[1] address generator's BASE register is not equal to its maximum value (set using xmax(), ymax() or amax())
CMEQMIN	Carry Main Equal Min Carry = TRUE if the *other*[1] MAIN register is equal to minimum (always 0)
BMEQMIN	Borrow Main Equal Min Borrow = TRUE if the *other*[1] MAIN register is equal to minimum (always 0)
CFEQMIN	Carry Field Equal Min Carry = TRUE if the *other*[1] FIELD register is equal to minimum (always 0)
BFEQMIN	Borrow Field Equal Min Borrow = TRUE if the *other*[1] FIELD register is equal to minimum (always 0)
CBEQMIN	Carry Base Equal Min Carry = TRUE if the *other*[1] BASE register is equal to minimum (always 0)
BBEQMIN	Borrow Base Equal Min Borrow = TRUE if the *other*[1] BASE register is equal to minimum (always 0)

Note-1: when used in a YALU instruction, the term *other* in the descriptions above refers to X address registers. When used in an XALU instruction, *other* refers to the Y address registers.

Table 4.13.7.2-1 YALU/XALU Carry/Borrow Operands *(Continued)*

Operand	Purpose
CMNEMIN	Carry Main Not Equal Min Carry = TRUE if the *other*[1] MAIN register is not equal to minimum (always 0)
BMNEMIN	Borrow Main Not Equal Min Borrow = TRUE if the *other*[1] MAIN register is not equal to minimum (always 0)
CFNEMIN	Carry Field Not Equal Min Carry = TRUE if the *other*[1] FIELD register is not equal to minimum (always 0)
BFNEMIN	Borrow Field Not Equal Min Borrow = TRUE if the *other*[1] FIELD register is not equal to minimum (always 0)
CBNEMIN	Carry Base Not Equal Min Carry = TRUE if the *other*[1] BASE register is not equal to minimum (always 0)
BBNEMIN	Borrow Base Not Equal Min Borrow = TRUE if the *other*[1] BASE register is not equal to minimum (always 0)
Note-1: when used in a YALU instruction, the term *other* in the descriptions above refers to X address registers. When used in an XALU instruction, *other* refers to the Y address registers.	

Example

```
% YALU    YMAIN, XCARE, CMEQMAX, INCREMENT, DYMAIN, OYMAIN
  XALU    XMAIN, XCARE, CON, INCREMENT, DXMAIN, OXMAIN
```

This example does the following:

- The ALU of the X address generator unconditionally (CON) increments the value in the X MAIN register (XMAIN), storing the result back into the X MAIN register (DXMAIN). And, the X address is output from X MAIN register (OXMAIN).
- The ALU of the Y address generator only increments the value in the Y MAIN register (YMAIN) when the MAIN register of the X address generator reaches max (CMEQMAX). The Y ALU output is stored back into the Y MAIN register (DYMAIN). And, the Y address is output from Y MAIN register (OYMAIN).

In operation, this means the X address will increment fast (every cycle), and the Y address will increment only when the X address reaches the maximum. In both cases, incrementing past the maximum value sets the corresponding register back to 0.

4.13.7.3 YALU/XALU Function Operands

See APG Address Generator, Memory Test Patterns, XALU Instruction, YALU Instruction.

Description

The YALU and XALU instructions take the following form:

```
YALU SourceA, SourceB, Carry/Borrow, Function, Destination(s),
Addressout

XALU SourceA, SourceB, Carry/Borrow, Function, Destination(s),
Addressout
```

The Function operand specifies the desired ALU operation.

There are two basic forms:

- Single-source instruction; i.e. ALL1S, COMP, DECREMENT, DOUBLE, HOLD, INCREMENT, ZERO.
- Dual-source instructions; i.e. the rest.

When a single source instruction is specified, the SourceB field must be XCARE.

The table below describes the options available for the Function operand to XALU and YALU:

Table 4.13.7.3-1 YALU/XALU Function Operands

Operand	Purpose
AANDBBAR	Logically AND SourceA input with inverted SourceB input (A • $\overline{B}$)
ADD	Add SourceA input to SourceB input plus Carry (A + B + Carry)
ALL1S	All ones = force ALU output to maximum (set using xmax(), ymax() or amax()). Ignores both source inputs.
ANANDBBAR	Logically NAND SourceA input with the inverted SourceB input $\overline{(A \bullet \overline{B})}$
AND	Logically AND SourceA input with SourceB input (A • B)

Table 4.13.7.3-1 YALU/XALU Function Operands *(Continued)*

Operand	Purpose
ANORBBAR	Logically NOR SourceA with the inverted SourceB input $\overline{(A + \overline{B})}$
AORBBAR	Logically OR SourceA with the inverted SourceB input $(A + \overline{B})$
COMP	Complement SourceA input $(\overline{A})$
DECREMENT	Subtract Borrow from SourceA input (A - Borrow)
DOUBLE	Double SourceA input (2 * A)
HOLD	Take no action; i.e. ALU output = SourceA (default)
INCREMENT	Add the Carry to SourceA input (A + Carry)
NAND	Logically NAND SourceA input with SourceB input $\overline{(A \bullet B)}$
NOR	Logically NOR SourceA with SourceB input $\overline{(A + B)}$
XOR	Exclusively OR SourceA input with SourceB input $(A \oplus B)$
XNOR	Exclusively NOR SourceA input with SourceB input $\overline{(A \oplus B)}$
OR	Logically OR SourceA with SourceB input (A + B)
SUBTRACT	Subtract SourceB input from SourceA input (A - B - Barrow)
ZERO	Force ALU output to zero. Ignores both source inputs.

Example

In the following example, the Y ALU adds (ADD) the value in the Y MAIN register (YMAIN) to the value in the Y BASE register (YBASE) with Carry forced ON (CON). The output is then put in both the Y BASE (DYBASE) and Y MAIN (DYMAIN) registers. Assuming Y MAIN and Y BASE have an initial value of 0, a 1 will be deposited in both the first time this instruction is executed. The second execution adds 1 + 1 + Carry = 3. The third execution adds 3 + 3 + Carry = 7, etc. The Y BASE (OYBASE) register is selected as the Y address output.

```
% YALU   YMAIN, YBASE, CON, ADD, DYMAIN, DYBASE, OYBASE
```

4.13.7.4 YALU/XALU Destination Operands

See APG Address Generator, Memory Test Patterns, XALU Instruction, YALU Instruction.

Description

The YALU and XALU instructions take the following form:

```
YALU SourceA, SourceB, Carry/Borrow, Function, Destination(s),
Addressout

XALU SourceA, SourceB, Carry/Borrow, Function, Destination(s),
Addressout
```

The `Destination(s)` operand(s) specify which of the 3 address register(s) are updated with the ALU output.

Rules:

- If no operand for the `Destination(s)` field is specified, the default is NODEST, which has the effect of discarding any operation performed by the ALU.
- Any combination of the three address registers can be used as destination registers, with multiple destinations separated by commas.
- It is not legal to mix NODEST with any other `Destination` option.

The table below describes the options available for the `Destination(s)` operand(s) to XALU and YALU:

Table 4.13.7.4-1 YALU/XALU Destination(s) Operands

Operand	Purpose
DYBASE	Writes Y ALU output to the Y BASE register (YALU instruction only)
DYFIELD	Writes Y ALU output to the Y FIELD register (YALU instruction only)
DYMAIN	Writes Y ALU output to the Y MAIN register (YALU instruction only)
DXBASE	Writes X ALU output to the X BASE register (XALU instruction only)
DXFIELD	Writes X ALU output to the X FIELD register (XALU instruction only)
DXMAIN	Writes X ALU output to the X MAIN register (XALU instruction only)
NODEST	No ALU output destination is enabled (default). In effect, this discards the result of any operation performed by the ALU. Usable in both XALU and YALU instructions.

Example

See Example

4.13.7.5 YALU/XALU Addressout Operands

See APG Address Generator, Memory Test Patterns, XALU Instruction, YALU Instruction.

Description

The YALU and XALU instructions take the following form:

```
YALU SourceA, SourceB, Carry/Borrow, Function, Destination(s),
Addressout

XALU SourceA, SourceB, Carry/Borrow, Function, Destination(s),
Addressout
```

The `Addressout` operand specifies which address register is selected as the output from the X or Y address generator. This is the address routed to the DUT via the Pin Scramble MUX.

Rules:

- Using the XALU instruction, if no operand for the `Addressout` field is specified, the X MAIN register (`OXMAIN`) is selected.
- Using the YALU instruction, if no operand for the `Addressout` field is specified, the Y MAIN register (`OYMAIN`) is selected.

The table below describes the options available for the `Addressout` operand to XALU and YALU:

Table 4.13.7.5-1 YALU/XALU Addressout Operands

Operand	Purpose
OYBASE	Y address source is Y BASE register
OYFIELD	Y address source is Y FIELD register
OYMAIN	Y address source is Y MAIN register (default)
OXBASE	X address source is X BASE register
OXFIELD	X address source is X FIELD register
OXMAIN	X address source is X MAIN register (default)

Example

See Example

4.13.8 COUNT Instruction

See Algorithmic Pattern Generator (APG), Memory Test Patterns Memory Pattern Instruction Format.

Description

The APG has 64 hardware Counters (60 are user accessible). Typical applications include:

- Pattern execution loop counters, to control pattern execution conditional branch operations.
- Use as an execution trace flag. A specified counter is incremented or decremented each time the associated instruction executes. After the pattern ends, the counter is read (more below) to determine whether or how many times a given instruction executed.

The test pattern `COUNT` instruction is used to control counter operations.

Note the following:

- Each counter is 32 bits; i.e. 2^{32} = 4,294,967,296 counts.
- Counters are identified as `COUNT1` through `COUNT60`.
- The entire `COUNT` instruction is optional. When omitted, no counters are modified and `COUNT1` is selected for any MAR Instructions which test a counter value.
- When a `COUNT` instruction is specified the `Counter` operand must be specified, identifying one counter (`COUNT1` through `COUNT60`). This counter is the target of the `Function` operand and is the counter evaluated by any MAR Instruction which tests a counter value in the current instruction.
- The `Function` operand is optional and, if specified, identifies the operation performed on the counter identified in the `Counter` field. If not specified the `NOCOUNT` instruction is used.

- Each counter is backed by a 32-bit reload register. During pattern execution, a reload register *may* be used to auto-reload its associated counter when the counter's value equals 0. Automatic reload is optional, enabled using the AON operand in the Autoreload field. Note: counter reload operation is described in COUNT Counter Operands, read this!
- Counter values and/or reload register values can be initialized in a pattern instruction using COUNTUDATA plus the UDATA value. See COUNT Function Operands. The RELOAD# operand will also load the reload register with the UDATA value.
- Counter values and reload register values can also be initialized in user C code or in Pattern Initial Conditions, using the count() and reload() functions. Executing count() also loads a given counter's reload register with the same value. The reload() function can be used to load a reload register without setting the associated counter.
- After a pattern is executed, the value of a given counter can be read using the count() function. A given reload register can be read using reload().

The default COUNT instruction is:

```
COUNT     NOCOUNT, NOCOUNT, AOFF
```

Usage

```
COUNT   Counter, Function, Autoreload
```

where:

Counter specifies which APG counter is selected in the current instruction. Legal values are COUNT1 through COUNT60. See COUNT Counter Operands.

Function specifies the operation to perform on **Counter**, in the current instruction. See COUNT Function Operands.

Autoreload controls automatic reloading of the counter when that counter = 0. Operation is described in COUNT Counter Operands. See COUNT Autoreload Operands.

The table below summarizes the operands for each field of the COUNT instruction. Default values are indicated using (**D**):

Table 4.13.8.0-1 COUNT Instruction Operands

Counter	Function	Autoreload
COUNT# NOCOUNT(**D**) RELOAD#	COUNTUDATA DEC2 DECR INCR NOCOUNT(**D**)	AOFF(**D**) AON
where # = 1 to 60		

Example

In the following example, APG counter 1 (COUNT1) is decremented (DECR) each time this instruction executes. The MAR CJMPNZ instruction causes execution to repeat this instruction (jump to label_X) until COUNT1 decrements to 0. Any time this instruction is executed with COUNT1 = 0 BEFORE COUNT1 is decremented it will be reloaded (AON) to the value in counter 1's reload register (see COUNT Counter Operands):

```
% labelX:
   COUNT   COUNT1, DECR, AON
   MAR     CJMPNZ, label_X
```

4.13.8.1 COUNT Counter Operands

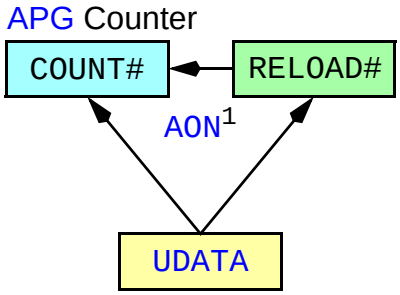
See Algorithmic Pattern Generator (APG), Memory Test Patterns, COUNT Instruction.

Description

The COUNT instruction takes the following form:

```
COUNT Counter, Function, Autoreload
```

The following diagram is used to describe APG counter operation:



Note-1: auto-reload occurs only if AON is specified and if, at the start of the current instruction, COUNT# = 0.

Using Maverick-II and Magnum 1 this behaviour can be modified, more below.

In each COUNT instruction an APG counter is selected, either explicitly, using the COUNT# or RELOAD# operands, or counter-1 (COUNT1) is selected by default (the same as NOCOUNT). The counter selected becomes the *target* of the current COUNT instruction and is the counter evaluated by any MAR Instruction which tests a counter value in the current instruction.

In each COUNT instruction, if AON is specified (see COUNT Autoreload Operands) and if, at the start of instruction execution, the target counter = 0, the counter *function* (see the COUNT Function Operands) is ignored and the target counter is reloaded (auto-reload) from its associated reload register. Otherwise (i.e. either AON is not specified and/or the target counter != 0) the auto-reload does not occur and the counter *function* is executed, with the target counter receiving the result of the operation.

If a RELOAD# operand is used the target counter's reload register is loaded with the UDATA value specified in the current instruction (default = 0). However, when AON is specified and auto-reload does occur the value reloaded into the target counter is different when using Maverick-I/-II and Magnum 1 vs. Magnum 2/2x:

- Using Maverick-I/-II and Magnum 1, if auto-reload does occur, the reload register is copied to the target counter after the UDATA value is transferred to the RELOAD# register.
- Using Magnum 2/2x, if auto-reload does occur, the reload register is copied to the target counter before the UDATA value is transferred to the RELOAD# register.

Note: beginning in software release v2.12.4, using Maverick-II and Magnum 1 some behavior above can be modified using the apg_reload_register_mode_set() function (this function is not usable on Magnum 2/2x but similar functionality can be obtained using the User RAM facility). On Maverick-II this requires APG revision APGMS8 or higher (use *boardrevs.exe* to check). The alternate behavior occurs only when using:

```
COUNT RELOAD#, any_counter_func, AON
```

In the alternate mode, AON is ignored (no auto-reload will occur regardless of the initial value in the target counter), the counter *function* is performed and the result is put into both the target counter and its reload register.

The table below summarizes the Counter operands to COUNT:

Table 4.13.8.1-1 COUNT Counter Operands

Operand	Purpose
COUNT#	Legal values are COUNT1 through COUNT60. Selects the target counter.
NOCOUNT	COUNT1 is the default target counter.
RELOAD#	Legal values are RELOAD1 through RELOAD60. Selects the target counter and affects other COUNT instruction operations, see Description.

Example

The following example decrements APG counter #2 (COUNT2). If, before counter #2 is decremented, its value = 0, counter #2 will be auto-reload'ed (AON) from its reload register; i.e. reload register #2 (see Description):

```
% COUNT COUNT2, DECR, AON
```

The following example loads both counter #4 and reload register #4 with 10:

```
% COUNT RELOAD4, COUNTUDATA, AOFF
  UDATA 10
```

The following example increments APG counter #2 (COUNT2) and loads reload register #2 with 10. If, before counter #2 is incremented, its value = 0, counter #2 will be auto-reload'ed (AON) from its reload register; i.e. reload register #2. The actual value loaded may be different using Maverick-I/-II and Magnum 1 vs. Magnum 2/2x, see Description:

```
% COUNT RELOAD2, INCR, AON
  UDATA 10
```

4.13.8.2 COUNT Function Operands

See Algorithmic Pattern Generator (APG), Memory Test Patterns, COUNT Instruction.

Description

The COUNT instruction takes the following form:

```
COUNT Counter, Function, Autoreload
```

The Function operand specifies the counter operation to be performed on the Counter selected in the current instruction. See COUNT Counter Operands for additional details.

The table below describes the options available for the Function operand to COUNT:

Table 4.13.8.2-1 COUNT Function Operands

Operand	Purpose
COUNTUDATA	Loads the UDATA value into the counter specified in the Counter operand. If Counter = RELOAD#, loads both COUNT# and RELOAD# with the UDATA value. See COUNT Counter Operands for additional details.
DECR	Decrements by 1 the counter specified in the Counter operand
DEC2	Decrements by 2 the counter specified in the Counter operand
INCR	Increments by 1 the counter specified in the Counter operand
NOCOUNT	No counter function is performed; i.e. the counter specified in the Counter operand holds its current value (default).

Example

The following example decrements APG counter #2 (COUNT2). If, before counter #2 is decremented, its value = 0, counter #2 will be reloaded (AON) from its reload register, see COUNT Counter Operands:

```
% COUNT COUNT2, DECR, AON
```

Also see Example, and Example.

4.13.8.3 COUNT Autoreload Operands

See Algorithmic Pattern Generator (APG), Memory Test Patterns, COUNT Instruction.

Description

The COUNT instruction takes the following form:

```
COUNT Counter, Function, Autoreload
```

The Autoreload operand is optional and, if specified, determines whether the target Counter will be reloaded from its associated reload register.

Each APG counter is backed by a 32 bit reload register. During pattern execution, APG counter auto-reload occurs when:

- At the start of the current instruction (i.e. before any counter modifications occurs in the current instruction), the value in the counter selected in the current instruction equals 0. And...
- COUNT AON is specified in the current instruction.

Note: important additional information related to auto-reload operation is documented in COUNT Counter Operands.

The table below describes the options available for the Autoreload operand to COUNT:

Table 4.13.8.3-1 COUNT Autoreload Operands

Operand	Purpose
AOFF	Disables Auto-reload (default)
AON	Enables Auto-reload

Example

The following example decrements APG counter #2 (COUNT2). If, before counter #2 is decremented, its value = 0, counter #2 will be auto-reload'ed (AON) from its reload register; i.e. reload register #2 (see COUNT Counter Operands for additional details):

```
% COUNT COUNT2, DECR, AON
```

In the following example, assuming the first instruction causes counter #2 to decrement to 0, counter #2 will be reloaded as indicated. Note that reloading would occur as noted if counter #2 = 0 regardless of how it reached that value:

```
% COUNT COUNT2, DECR, AON // Reloaded here if =0 before DECR
% COUNT COUNT2, any_function, AOFF // NOT reloaded here
% COUNT COUNT2, any_function, AOFF // NOT reloaded here
% COUNT COUNT2, any_function, AON  // Reload occurs here if =0
```

4.13.9 MAR Instruction

See Algorithmic Pattern Generator (APG), Memory Test Patterns, Memory Pattern Instruction Format.

Description

During pattern execution, APG operation is controlled by the hardware APG Controller Engine (also called the MAR engine), as directed by the MAR instruction in each APG pattern instruction. This instruction also controls various other miscellaneous features (more below).

The term MAR originally referred to the APG's MicroRAM Address Register. In this documentation, the term MAR is also used:

- To represent the pattern instruction which controls memory pattern execution sequence; i.e. MAR instruction.
- When referring to the hardware engine which controls all memory pattern execution; i.e. MAR Engine.
- The value in the APG MAR register. This is the address of the current pattern instruction being executed. Note that, in this context, user code cannot access literal MAR values; all references to specific pattern instructions is done using a Pattern Label, label + offset, or pattern name.

The purposes of the MAR instruction are:

- Control the APG pattern instruction execution sequence (Branch-condition, Address).
- Enable/disable strobes from the APG Data Generator (Strobe-control).
- Control the programmable APG Interrupt Timer (Interrupt, Timer).
- Control miscellaneous features (Misc).
- Control the branch-on-error error type selection (BOE-type).
- Control the error source selection (Error-choice).

The default MAR instruction is:

```
MAR  INC, NOREAD, NOINT, RSTTMR, ERRSRC1
```

Usage

```
MAR  Branch-condition, Address, Strobe-control, Interrupt, Timer,
     Misc, BOE-type, Error-choice
```

where:

`Branch-condition` specifies how the next instruction to be executed is determined. This can be as simple as incrementing the MAR address to the next instruction (`INC`), or a conditional or unconditional branch to an arbitrary instruction, or a conditional or unconditional subroutine call or return. Conditional operations can be based on an APG counter value, or the PASS/FAIL status of an error flag. See MAR Branch Condition Operands.

Note: in Magnum 1/2/2x test patterns, many conditional branch operations are affected by a static setup (see Static Error Choice Functions, Branch-on-error) and the MAR Error-choice Operands.

`Address` identifies the instruction to be executed when **`Branch-condition`** is not `INC`, `PAUSE`, `RETURN`, or `DONE`. An address is specified using a Pattern Label or the name of another test pattern. Rules apply, see MAR Address Operand.

`Strobe-control` enables\disables strobes pin pins which are scrambled (see Pin Scramble) to APG Data Generator outputs in the current instruction. Either all strobes are disabled (`NOREAD`), enabled (`READ`), or a bit-mask can be specified to control strobes individually on each APG Data Generator output (`READUDATA` + `UDATA`). The `data_strobe()` function also affects these same strobes. See MAR Strobe Control Operands.

`Interrupt` allows the real time programmable APG Interrupt Timer to generate an interrupt in the current instruction. See MAR Interrupt Operands.

`Timer` allows the APG Interrupt Timer to count in the current instruction. See MAR Timer Operands.

`Misc` controls several unrelated features. Any combination of the following can be used in a given pattern instruction, in any order. See MAR Misc Operands:

- `RESET` = reset all error flags (see Error Flag vs. Error Latch).
- `NOLATCH`/`LATCH` = controls whether a failing strobe in the current cycle will set an error latch (see Error Flag vs. Error Latch). Default = `LATCH`.

- VCOMP = generates a trigger to strobe the DC Comparators and Error Logic or DC A/D Converter in the DC Test and Measure System. See Dynamic DC Tests.
- OVER = enable the over-programming logic in the current instruction. See Over-programming Controls and Parallel Test.
- CLEARERR: unconditionally clears all Total Error Counters, Row Error Counters and Col Error Counters and IOC Error Counters (see MAR Error-choice Operands). CLEARERR does not clear the PE Error Flags.

BOE-type selects the type of signal used by branch-on-error operation in the *next instruction executed*. This operand is required for some, but not all, MAR branch-on-error operations. See MAR BOE Type Operands.

Error-choice selects between 4 sets of *error* signals (in 3 groups), increasing test pattern branch options as compared to Maverick-I/-II. See MAR Error-choice Operands

The table below summarizes the operands for each field of the MAR instruction. For Magnum 1/2/2x, there are 2 tables; additional Branch-condition and new Error-choice

operands, which only apply to Magnum 1/2/2x, are listed separately, in the 2nd table. Default values are indicated using (**D**):

Table 4.13.9.0-1 MAR Instruction Operands

Branch Condition[3]	Address	Strobe Control	Interrupt	Timer	Misc
INC(**D**) DONE GOSUB JUMP PAUSE RETURN CJMPA CJMPNA CJMPE CJMPNE CJMPT CJMPNT CJMPZ CJMPNZ CRETE CRETNE CRETT CRETNT CRETZ CRETNZ CSUBE CSUBNE CSUBT CSUBNT CSUBZ CSUBNZ	Pattern Label (none)(**D**)	NOREAD(**D**) READ READUDATA READV READZ	INTADR INTEN INTENADR NOINT(**D**)	RSTTMR(**D**) TIMEN	NOLATCH RESET OVER VCOMP VCNTR[1] CLEARERR DEFAULT[2] (none)(**D**)

Notes:
1) Not usable in Maverick-I (only).
2) Usable in Magnum 1/2/2x mixedsync test patterns only.
3) Additional Branch Condition and Error Choice operands are listed in the following table.

Table 4.13.9.0-2 Magnum 1/2/2x Only MAR Instruction Operands[1]

Branch Condition	Error Choice
CSUBNE_ANOTB CSUBNE_BNOTA CSUBNE_ALL CSUBNE_DUT1 ...thru... CSUBNE_DUT8 CSUBE_ANOTB CSUBE_BNOTA CSUBE_ALL CSUBE_DUT1 ...thru... CSUBE_DUT8 CJMPNE_ANOTB CJMPNE_BNOTA CJMPNE_ALL CJMPNE_DUT1 ...thru... CJMPNE_DUT8	ERRSRC1(**D**) ERRSRC2 ERRSRC3 ERRSRC4
Note-1: only usable in Magnum 1/2/2x test patterns. Several other Magnum 1/2/2x only operands appear in the previous table	

Table 4.13.9.0-2 Magnum 1/2/2x Only MAR Instruction Operands[1] *(Continued)*

Branch Condition	Error Choice
CJMPE_ANOTB CJMPE_BNOTA CJMPE_ALL CJMPE_DUT1 ...thru... CJMPE_DUT8 CRETNE_ANOTB CRETNE_BNOTA CRETNE_ALL CRETNE_DUT1 ...thru... CRETNE_DUT8 CRETE_ANOTB CRETE_BNOTA CRETE_ALL CRETE_DUT1 ...thru... CRETE_DUT8	

Note-1: only usable in Magnum 1/2/2x test patterns. Several other Magnum 1/2/2x only operands appear in the previous table

4.13.9.1 MAR Default Pattern Instruction

See Memory Test Patterns, MAR Instruction, Logic Test Patterns, VAR Instruction.

Using Magnum 1/2/2x, Mixed Memory/Logic Patterns are always controlled by independent hardware engines: the MAR Engine for memory instructions and the VAR Engine for logic instructions.

In mixed patterns, all synchronization required between the two engines is normally accomplished using explicit pattern instructions, typically requiring careful attention to details

and ensuring that appropriate memory AND logic instructions are always supplied to both engines in all pattern instructions.

However, default instructions are defined for both the memory engine (see Default Memory Pattern Instruction) and logic engine (VAR INC). Any/all Mixed Memory/Logic Patterns instruction(s) which don't include explicit memory instruction(s) and/or a logic instruction will use the default instruction for the missing instruction.

In test patterns with the mixedsync Pattern Attributes, the DEFAULT operand can be used to redefine the default memory and/or logic instruction.

Note the following:

- Use of the DEFAULT operand is optional, for both MAR and VAR instructions.
- The DEFAULT operand is purely a compile-time operation.
- MAR DEFAULT records the current memory instruction as the new default memory instruction. The default memory instruction is then implicitly added to any subsequent pattern instruction which does not contain an explicit memory instruction.
- VAR DEFAULT records the current logic instruction as the new default logic instruction. The default logic instruction is then implicitly added to any subsequent pattern instruction which does not contain an explicit logic instruction.
- MAR DEFAULT and VAR DEFAULT can be used in the same pattern instruction.
- Either default instruction can be redefined as desired.

4.13.9.2 MAR Branch Condition Operands

See Algorithmic Pattern Generator (APG), Memory Test Patterns, MAR Instruction.

Description

MAR Branch-condition operands are used to control pattern execution sequence. The MAR instruction takes the following form:

```
MAR  Branch-condition, Address, Strobe-control, Interrupt, Timer,
     Misc, BOE-type, Error-choice
```

There are several types of MAR branch condition operands:

- Unconditional, no Address required: INC, DONE, RETURN, PAUSE. See MAR Unconditional Branch-condition Operands.

- Unconditional, Address required: JUMP, GOSUB. See MAR Unconditional Branch-condition Operands.
- Conditional, no Address required: CRETNZ, CRETNE, etc., more below.
- Conditional, Address required: CSUBNE, CSUBNT etc., more below.

There are 3 basic conditional operations:

- Conditional jump (CJMPE, CJMPNZ, etc.)
- Conditional subroutine call (CSUBNE, CSUBNT, etc.)
- Conditional subroutine return (CRETNZ, CRETNE, etc.)

In all 3 cases, the MAR Address operand specifies which instruction is executed next if the condition being evaluated is TRUE (more below). The Address operand is specified as a Pattern Label or a PATTERN name (rules apply, see Note below).

Note: a subroutine may be identified using a Pattern Label if the subroutine is within the same pattern, or by referring to another PATTERN. The destination of a test pattern jump/branch instruction can only reference a Pattern Label within the same pattern.

MAR conditional operations can test a variety of parameters, one per instruction. The parameter to test is identified by the name of the MAR Branch-condition operand. See MAR Conditional Branch-condition Operands and MAR Multi-DUT Branch-condition Operands.

The MAR Multi-DUT Branch-condition Operands provide per-DUT branch options targeted for use in Multi-DUT Test Programs. These options require that the DUT board design adhere to specific DUT-pin to Tester-pin Connection Requirements.

The MAR Multi-DUT Branch-condition Operands provide options which require that the user understand some hardware details and choose between several error source selection options. See MAR Error-choice Operands and MAR BOE Type Operands.

Note: the term *error* is used consistently, even when the selected branch signal represents the output of ECR Counter Comparators. See MAR Error-choice Operands.

As indicated, using conditional operations, pattern execution will branch to the specified Address, or call/return from a subroutine if the condition is TRUE, otherwise execution will continue with the next instruction. If a conditional subroutine call or return is specified and

the condition is TRUE, execution is the same as an unconditional subroutine call or return; i.e. complete the current instruction then call or return.

Several tables below describe the unconditional and conditional options available for the MAR `Branch-condition` operands. A separate table lists MAR Multi-DUT Branch-condition Operands which only apply to Magnum 1/2/2x:

Table 4.13.9.2-1 MAR Unconditional Branch-condition Operands

Operand	Purpose
DONE	Halts the APG MAR Engine; i.e. stops pattern execution. This sends a done signal to the site controller (default). Any specified `Address` is ignored. See MAR DONE and/or VAR DONE.
GOSUB	Calls the subroutine at the specified address (see Note:). See Pattern Subroutines.
INC	Execution proceeds to the next instruction. Any specified `Address` is ignored.

Table 4.13.9.2-1 MAR Unconditional Branch-condition Operands *(Continued)*

Operand	Purpose
JUMP	Execution will unconditionally jump to the specified address (see Note:).
PAUSE	Stops the APG MAR Engine (pauses the pattern) when the instruction containing the PAUSE reaches the DUT. The pattern can be restarted using the `restart()` function. The restarted pattern will continue execution at the instruction immediately following the instruction containing the PAUSE. Restriction: two consecutive pattern instructions containing PAUSE are not allowed. Any specified `Address` is ignored. Note: the pattern generator uses a pipe-line architecture. This means that during the time a test pattern is paused that user-code must not: • Write any APG hardware registers, including Address Generator, Data Generator, Chip-selects, Counters, JAM register, User RAM, UDATA, etc. Register read is OK. • Modify any cycle period values. The details of the pattern generator pipe-line architecture are not documented, because: • The architecture is variable, based on which hardware options are used in any given test program and/or test pattern. • The architecture is subject to change without notice, as needed to add new features, fix problems, etc.
RETURN	Execution returns from a subroutine or interrupt timer routine to the address popped off the stack. Any specified address is ignored. See Pattern Subroutines.

Table 4.13.9.2-2 MAR Conditional Branch-condition Operands

Operand	Purpose
CJMPA	Conditional JuMP on Abort Execution will jump to the specified address (see Note:) if the Abort signal is TRUE at the start of the current instruction. The Abort signal comes from the PE error latches, not error flags, see Error Flag vs. Error Latch. If the Abort signal is FALSE, the next instruction will execute. The Abort signal will be TRUE when all DUTs in a site have one or more set error *latches.* See Error Pipeline Requirements. The Abort signal is useful when testing multiple DUTs; the Abort signal will be TRUE when all DUTs have failed. Using Magnum 1/2/2x, the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and which MAR Error-choice Operands are used in the same instruction. See APG Instruction Execution. Note: using Magnum 1/2/2x, proper operation of the Abort signal requires that all DUTs be enabled; i.e. all DUTs must be in the Active DUTs Set (ADS). The Abort signal will NOT go active if any DUT(s) are disabled.

Table 4.13.9.2-2 MAR Conditional Branch-condition Operands *(Continued)*

Operand	Purpose
CJMPNA	Conditional JuMP on Not Abort Execution will jump to the specified address (see Note:) if the Abort signal is FALSE at the start of the current instruction. The Abort signal comes from the PE error latches, not error flags, see Error Flag vs. Error Latch. If the Abort signal is TRUE, the next instruction will execute. The Abort signal will be FALSE when any one or more DUT(s) in a site do not have a set error *latch*. See Error Pipeline Requirements. The Abort signal is useful when testing multiple DUTs; the Abort signal will be TRUE when all DUTs have failed. Using Magnum 1/2/2x, the operation of this operand is affected/ changed by a static setup (see Static Error Choice Functions, Branch-on-error) and which MAR Error-choice Operands are used in the same instruction. See APG Instruction Execution. Note: using Magnum 1/2/2x, proper operation of the Abort signal requires that all DUTs be enabled; i.e. all DUTs must be in the Active DUTs Set (ADS). The Abort signal will NOT go active if any DUT(s) are disabled.
CJMPE	Conditional JuMP on Error Execution will jump to the specified address (see Note:) if the error signal is TRUE at the start of the current instruction. If the error signal is FALSE, the next instruction will execute. The Error signal will be TRUE if any error flag(s) are set (see Error Flag vs. Error Latch). See Error Pipeline Requirements. Using Magnum 1/2/2x, the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and which MAR Error-choice Operands are used in the same instruction. See APG Instruction Execution.

Table 4.13.9.2-2 MAR Conditional Branch-condition Operands *(Continued)*

Operand	Purpose
CJMPNE	Conditional JuMP on No Error Execution will jump to the specified address (see Note:) if the Error signal is FALSE at the start of the current instruction. If the Error signal is TRUE, the next instruction will execute. The Error signal will be FALSE when no error flag(s) are set (see Error Flag vs. Error Latch). See Error Pipeline Requirements. Using Magnum 1/2/2x, the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and which MAR Error-choice Operands are used in the same instruction. See APG Instruction Execution.
CJMPT	Conditional JuMP on Timer = 0 Execution will jump to the specified address (see Note:) if the APG Interrupt Timer = 0 at the start of the current instruction. If the interrupt timer is not zero, the next instruction will execute. See Note:. See APG Instruction Execution.
CJMPNT	Conditional JuMP on Timer Not = 0 Execution will jump to the specified address (see Note:) if the APG Interrupt Timer is not zero at the start of the current instruction. If the interrupt timer is zero, the next instruction will execute. See Note:. See APG Instruction Execution.
CJMPZ	Conditional JuMP on Zero Execution will jump to the specified address (see Note:) if the counter specified in the COUNT instruction is zero at the start of the current instruction. If the counter is not zero the next pattern instruction will execute. See APG Instruction Execution.
CJMPNZ	Conditional JuMP on Not Zero Execution will jump to the specified address (see Note:) if the counter specified in the COUNT instruction is not zero at the start of the current instruction. If the counter is zero the next instruction will execute. See APG Instruction Execution.

Table 4.13.9.2-2 MAR Conditional Branch-condition Operands *(Continued)*

Operand	Purpose
CRETE	Conditional RETurn on Error Execution returns to the instruction address popped off the stack if the error signal is TRUE at the start of the current instruction. If the Error signal is FALSE, the next instruction will execute. The Error signal will be TRUE if any error flag(s) are set (see Error Flag vs. Error Latch). See Error Pipeline Requirements and Pattern Subroutines. Using Magnum 1/2/2x, the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and which MAR Error-choice Operands are used in the same instruction. See APG Instruction Execution.
CRETNE	Conditional RETurn on No Error Execution returns to the instruction address popped off the stack if the error signal is FALSE at the start of the current instruction. If the Error signal is TRUE, the next instruction will execute. The Error signal will be FALSE when no error flag(s) are set (see Error Flag vs. Error Latch). See Error Pipeline Requirements and Pattern Subroutines. Using Magnum 1/2/2x, the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and which MAR Error-choice Operands are used in the same instruction. See APG Instruction Execution.
CRETT	Conditional RETurn on Timer = 0 Execution returns to the instruction address popped off the stack if the APG Interrupt Timer = 0 at the start of the current pattern instruction. If the interrupt timer is not zero, the next instruction will execute. See Pattern Subroutines. See Note:. See APG Instruction Execution.
CRETNT	Conditional RETurn on Timer Not = 0 Execution returns to the instruction address popped off the stack if the APG Interrupt Timer is not zero at the start of the current pattern instruction. If the interrupt timer is zero, the next instruction will execute. See Pattern Subroutines. See Note:. See APG Instruction Execution.
CRETZ	Conditional RETurn on Zero Execution returns to the instruction address popped off the stack if the counter specified in the COUNT instruction is zero at the start of the current pattern instruction. If the counter is not zero, the next instruction will execute. See Pattern Subroutines and APG Instruction Execution.

Table 4.13.9.2-2 MAR Conditional Branch-condition Operands *(Continued)*

Operand	Purpose
CRETNZ	Conditional RETurn on Not Zero Execution returns to the instruction address popped off the stack if the counter specified in the COUNT instruction is not zero at the start of the current pattern instruction. If the counter is zero, the next instruction will execute. See Pattern Subroutines and APG Instruction Execution.
CSUBE	Conditional SUBroutine call on Error If the error signal is TRUE at the start of the current instruction, calls the specified pattern subroutine (see Note:) and pushes the subroutine return address on the execution stack. If the Error signal is FALSE, the next instruction will execute. The Error signal will be TRUE if any error flag(s) are set (see Error Flag vs. Error Latch). See Error Pipeline Requirements and Pattern Subroutines. Using Magnum 1/2/2x, the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and which MAR Error-choice Operands are used in the same instruction. See APG Instruction Execution.
CSUBNE	Conditional SUBroutine call on No Error If the error signal is FALSE at the start of the current instruction, calls the specified pattern subroutine (see Note:) and pushes the subroutine return address on the execution stack. If the Error signal is TRUE, the next instruction will execute. The Error signal will be FALSE if no error flags are set (see Error Flag vs. Error Latch). See Error Pipeline Requirements and Pattern Subroutines. Using Magnum 1/2/2x, the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and which MAR Error-choice Operands are used in the same instruction. See APG Instruction Execution.
CSUBT	Conditional SUBroutine call on Timer = 0) Calls the specified subroutine (see Note:) if the APG Interrupt Timer = 0 at the start of the current pattern instruction. The subroutine return address is pushed on the execution stack. If the interrupt timer is not 0, the next instruction will execute. See Pattern Subroutines. See Note:. See APG Instruction Execution.

Table 4.13.9.2-2 MAR Conditional Branch-condition Operands *(Continued)*

Operand	Purpose
CSUBNT	Conditional SUBroutine call on Timer Not = 0 Calls the specified subroutine (see Note:) if the APG Interrupt Timer is not 0 at the start of the current pattern instruction. The subroutine return address is pushed on the execution stack. If the timer = 0, the next instruction will execute. See Pattern Subroutines. See Note:. See APG Instruction Execution.
CSUBZ	Conditional SUBroutine call on Zero Calls the specified subroutine (see Note:) if the counter specified in the COUNT instruction is zero at the start of the current pattern instruction. The subroutine return address is pushed on the execution stack. If the counter is not zero, the next instruction will execute. See Pattern Subroutines and APG Instruction Execution.
CSUBNZ	Conditional SUBroutine call on Not Zero Calls the specified subroutine (see Note:) if the counter specified in the COUNT instruction is not zero at the start of the current pattern instruction. The subroutine return address is pushed on the execution stack. If the counter is zero, the next instruction will execute. See Pattern Subroutines and APG Instruction Execution.

Note: it is illegal to use INTEN or INTADR in the same pattern instruction with a MAR conditional branch based on the APG Interrupt Timer (MAR CJMPT, CRETT, etc.). This rule is enforced by the pattern compiler.

The following conditional options are targeted for use in Magnum 1/2/2x Multi-DUT Test Programs. To properly use these options requires the user consider both the static error

choice setup (see Static Error Choice Functions, Branch-on-error) and the MAR Error-choice Operands:

Table 4.13.9.2-3 MAR Multi-DUT Branch-condition Operands

Operand	Purpose
CSUBE_ALL	Conditional SUBroutine Call Error ALL $(A_1 \cdot A_2 \cdot A_n) \cdot (B_1 \cdot B_2 \cdot B_n)$ (see Note below) The subroutine will be called if every DUT on every Sub-site; i.e. all DUTs fail. SeeError Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CSUBNE_ALL	Conditional SUBroutine Call No Error ALL $(\overline{A}_1 + \overline{A}_2 + \overline{A}_n) + (\overline{B}_1 + \overline{B}_2 + \overline{B}_n)$ (see Note below) This is the inverse of CSUBE_ALL i.e. the subroutine will be called if any DUT on any Sub-site doesn't have an error; i.e. all DUTs must have an error to NOT branch. SeeError Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CSUBE_ANOTB	Conditional SUBroutine Call Error Sub-site-A not Sub-site-B $(A_1 + A_2 + A_n) \cdot (\overline{B}_1 \cdot \overline{B}_2 \cdot \overline{B}_n)$ (see Note below) The subroutine will be called if any DUT on Sub-site-A has an error AND no DUTs on Sub-site-B have an error. SeeError Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.

Note: A_1 = Sub-site-A DUT #1 has an error
$\overline{B}_2$ = Sub-site-B DUT #2 has no errors
In these descriptions the term *error* is used consistently, even when the signal represents the output of ECR Counter Comparators.

Table 4.13.9.2-3 MAR Multi-DUT Branch-condition Operands *(Continued)*

Operand	Purpose
CSUBNE_ANOTB	Conditional SUBroutine Call No Error Sub-site-A not Sub-site-B $(\overline{A}_1 \cdot \overline{A}_2 \cdot \overline{A}_n) + (B_1 + B_2 + B_n)$ (see Note below) This is the inverse of CSUBE_ANOTB i.e. the subroutine will be called if no DUTs on Sub-site-A have an error OR any DUT on Sub-site-B has an error. SeeError Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CSUBE_BNOTA	Conditional SUBroutine Call Error Sub-site-B not Sub-site-A $(A_1 \cdot A_2 \cdot A_n) \cdot (B_1 + B_2 + B_n)$ (see Note below) The subroutine will be called if no DUTs on Sub-site-A have an error AND any DUT on Sub-site-B has an error. SeeError Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CSUBNE_BNOTA	Conditional SUBroutine Call No Error Sub-site-B not Sub-site-A $(A_1 + A_2 + A_n) + (\overline{B}_1 \cdot \overline{B}_2 \cdot \overline{B}_n)$ (see Note below) This is the inverse of CSUBE_BNOTA i.e. the subroutine will be called if any DUT on Sub-site-A has an error OR no DUTs on Sub-site-B have an error. SeeError Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CSUBE_DUT1 thru CSUBE_DUT8 See Note:	Conditional SUBroutine Call Error DUT-1 thru DUT-8 Subroutine will be called if the specified DUT has an error. Errors associated with other DUTs have no effect. See Error Pipeline Requirements, Pattern Subroutines and Note:. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.

Note: A_1 = Sub-site-A DUT #1 has an error
$\overline{B}_2$ = Sub-site-B DUT #2 has no errors
In these descriptions the term *error* is used consistently, even when the signal represents the output of ECR Counter Comparators.

Table 4.13.9.2-3 MAR Multi-DUT Branch-condition Operands *(Continued)*

Operand	Purpose
CSUBNE_DUT1 thru CSUBNE_DUT8 See Note:	Conditional SUBroutine Call No Error DUT-1 thru DUT-8 Subroutine will be called if the specified DUT has no errors. Errors associated with other DUTs have no effect. See Error Pipeline Requirements, Pattern Subroutines and Note:. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CRETE_ALL	Conditional Return Error ALL $(A_1 \cdot A_2 \cdot A_n) \cdot (B_1 \cdot B_2 \cdot B_n)$ (see Note below) The subroutine will return if every DUT on every Sub-site has an error; i.e. all DUTs fail. SeeError Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CRETNE_ALL	Conditional Return No Error ALL $(\overline{A}_1 + \overline{A}_2 + \overline{A}_n) + (\overline{B}_1 + \overline{B}_2 + \overline{B}_n)$ (see Note below) This is the inverse of CRETE_ALL i.e. the subroutine will return if any DUT on Sub-site-A doesn't have an error OR any DUT on Sub-site-B doesn't have an error i.e. all DUTs must have an error to NOT return. SeeError Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CRETE_ANOTB	Conditional Return Error Sub-site-A not Sub-site-B $(A_1 + A_2 + A_n) \cdot (\overline{B}_1 \cdot \overline{B}_2 \cdot \overline{B}_n)$ (see Note below) The subroutine will return if any DUT on Sub-site-A has an error AND no DUTs on Sub-site-B have an error. SeeError Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.

Note: A_1 = Sub-site-A DUT #1 has an error
$\overline{B}_2$ = Sub-site-B DUT #2 has no errors
In these descriptions the term *error* is used consistently, even when the signal represents the output of ECR Counter Comparators.

Table 4.13.9.2-3 MAR Multi-DUT Branch-condition Operands *(Continued)*

Operand	Purpose
CRETNE_ANOTB	Conditional Return No Error Sub-site-A not Sub-site-B $(\overline{A}_1 \cdot \overline{A}_2 \cdot \overline{A}_n) + (B_1 + B_2 + B_n)$ (see Note below) This is the inverse of CRETE_ANOTB i.e. the subroutine will return if no DUTs on Sub-site-A have an error OR any DUT on Sub-site-B has an error. SeeError Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CRETE_BNOTA	Conditional Jump Error Sub-site-B not Sub-site-A $(\overline{A}_1 \cdot \overline{A}_2 \cdot \overline{A}_n) \cdot (B_1 + B_2 + B_n)$ (see Note below) The subroutine will return if no DUTs on Sub-site-A have an error AND any DUT on Sub-site-B has an error. SeeError Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CRETNE_BNOTA	Conditional Jump No Error Sub-site-B not Sub-site-A $(A_1 + A_2 + A_n) + (\overline{B}_1 \cdot \overline{B}_2 \cdot \overline{B}_n)$ (see Note below) This is the inverse of CRETE_BNOTA i.e. the subroutine will return if any DUT on Sub-site-A has an error OR no DUTs on Sub-site-B have an error. SeeError Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CRETE_DUT1 thru CRETE_DUT8 See Note:	Conditional Return Error DUT-1 thru DUT-8 Subroutine will return if the specified DUT has an error. Errors associated with other DUTs have no effect. See Error Pipeline Requirements, Pattern Subroutines and Note:. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.

Note: A_1 = Sub-site-A DUT #1 has an error
$\overline{B}_2$ = Sub-site-B DUT #2 has no errors
In these descriptions the term *error* is used consistently, even when the signal represents the output of ECR Counter Comparators.

Table 4.13.9.2-3 MAR Multi-DUT Branch-condition Operands *(Continued)*

Operand	Purpose
CRETNE_DUT1 thru CRETNE_DUT8 See Note:	Conditional Return No Error DUT-1 thru DUT-8 Subroutine will return if the specified DUT has no errors. Errors associated with other DUTs have no effect. See Error Pipeline Requirements, Pattern Subroutines and Note:. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CJMPE_ALL	Conditional Jump Error ALL $(A_1 \cdot A_2 \cdot A_n) \cdot (B_1 \cdot B_2 \cdot B_n)$ (see Note below) The jump will occur if every DUT on every Sub-site has an error i.e. all DUTs fail. SeeError Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CJMPNE_ALL	Conditional Jump No Error ALL DUTs $(\overline{A}_1 + \overline{A}_2 + \overline{A}_n) + (\overline{B}_1 + \overline{B}_2 + \overline{B}_n)$ (see Note below) This is the inverse of CJMPE_ALL i.e. the jump will occur if any DUT on any Sub-site doesn't have an error; i.e. all DUTs must have an error to NOT branch. SeeError Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CJMPE_ANOTB	Conditional Jump Error Sub-site-A not Sub-site-B $(A_1 + A_2 + A_n) \cdot (\overline{B}_1 \cdot \overline{B}_2 \cdot \overline{B}_n)$ (see Note below) The jump will occur if any DUT on Sub-site-A has an error AND no DUTs on Sub-site-B have an error. SeeError Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.

Note: A_1 = Sub-site-A DUT #1 has an error
$\overline{B_2}$ = Sub-site-B DUT #2 has no errors
In these descriptions the term *error* is used consistently, even when the signal represents the output of ECR Counter Comparators.

Table 4.13.9.2-3 MAR Multi-DUT Branch-condition Operands *(Continued)*

Operand	Purpose
CJMPNE_ANOTB	Conditional Jump No Error Sub-site-A not Sub-site-B $(\overline{A}_1 \cdot \overline{A}_2 \cdot \overline{A}_n) + (B_1 + B_2 + B_n)$ (see Note below) This is the inverse of CJMPE_ANOTB i.e. the jump will occur if no DUTs on Sub-site-A have an error OR any DUT on Sub-site-B has an error. SeeError Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CJMPE_BNOTA	Conditional Jump Error Sub-site-B not Sub-site-A $(\overline{A}_1 \cdot \overline{A}_2 \cdot \overline{A}_n) \cdot (B_1 + B_2 + B_n)$ (see Note below) The jump will occur if no DUTs on Sub-site-A have an error AND any DUT on Sub-site-B has an error. SeeError Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.

Note: A_1 = Sub-site-A DUT #1 has an error
$\overline{B_2}$ = Sub-site-B DUT #2 has no errors
In these descriptions the term *error* is used consistently, even when the signal represents the output of ECR Counter Comparators.

Table 4.13.9.2-3 MAR Multi-DUT Branch-condition Operands *(Continued)*

Operand	Purpose
CJMPNE_BNOTA	Conditional Jump No Error Sub-site-B not Sub-site-A $(A_1 + A_2 + A_n) + (\overline{B}_1 \cdot \overline{B}_2 \cdot \overline{B}_n)$ (see Note below) This is the inverse of CJMPE_BNOTA i.e. the jump will occur if any DUT on Sub-site-A has an error OR no DUTs on Sub-site-B have an error. SeeError Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CJMPE_DUT1 thru CJMPE_DUT8 See Note:	Conditional Jump Error DUT-1 thru DUT-8 Jump will occur if the specified DUT has an error. Errors associated with other DUTs have no effect. See Error Pipeline Requirements, Pattern Subroutines and Note:. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CJMPNE_DUT1 thru CJMPNE_DUT8 See Note:	Conditional Jump No Error DUT-1 thru DUT-8 Jump will occur if the specified DUT has no errors. Errors associated with other DUTs have no effect. See Error Pipeline Requirements, Pattern Subroutines and Note:. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
Note: A_1 = Sub-site-A DUT #1 has an error $\overline{B}_2$ = Sub-site-B DUT #2 has no errors In these descriptions the term *error* is used consistently, even when the signal represents the output of ECR Counter Comparators.	

- CRETE_DUT9 is equivalent to CRETE_DUT1
- CJMPE_DUT10 is equivalent to CJMPE_DUT2
- Etc.

This allows the test pattern to reflect that _DUT1 also represents _DUT9 in these static modes, with the MAR Error-choice Operands determining which DUT is actually being tested. See MAR Error-choice Operands and Static Error Choice Functions, Branch-on-error.

4.13.9.3 MAR Address Operand

See APG Controller Engine, Memory Test Patterns, MAR Instruction.

Description

The MAR instruction takes the following form:

```
MAR  Branch-condition, Address, Strobe-control, Interrupt, Timer,
     Misc, BOE-type, Error-choice
```

The MAR instruction Address operand identifies a pattern instruction as the target for a conditional or unconditional branch operation. Note the following:

- An Address is specified using a Pattern Label or a PATTERN name, but not all Branch-condition operations can use both.
- A subroutine address (GOSUB, CSUBE, etc.) can be specified as a Pattern Label in the same pattern or as a PATTERN name.
- A jump address (JUMP, CJMPZ, etc.) can only be specified as a Pattern Label in the same pattern, i.e. not as a PATTERN name.
- The MAR address operand may be omitted for instructions that do not require a target address; i.e. INC, DONE, RETURN, PAUSE. With all other jump and subroutine operands (CSUBE, CJMPZ, JUMP, GOSUB, etc.) if the address is omitted, the default address is the first instruction of the current test pattern.

4.13.9.4 MAR Strobe Control Operands

See APG Controller Engine, Memory Test Patterns, MAR Instruction.

Description

The MAR instruction takes the following form:

```
MAR  Branch-condition, Address, Strobe-control, Interrupt, Timer,
     Misc, BOE-type, Error-choice
```

all pins scrambled to

The MAR Strobe-control operand is used to enable or disable strobes on pins scrambled to the APG Data Generator outputs in the current instruction (see Pin Scramble MUX and Pin Scramble Functions & Macros).

Note: the data_strobe() function must also enable strobes on appropriate APG Data Generator outputs, prior to executing the test pattern.

The Strobe-control operand has no effect on strobes controlled by APG Chip Selects (see CHIPS CSmRDT, CSmRDV, CSmRDZ and CSmRDF).

Note: the last instruction of a pattern (MAR DONE) must not generate any strobes; i.e. do NOT use any of READ, READV, READV, READUDATA, CSmRDT, CSmRDV, CSmRDZ and CSmRDF in the last instruction of a test pattern.

Several Strobe-control operand options are available as noted in the table below:

Table 4.13.9.4-1 MAR Strobe-control Operands

Operand	Purpose
READ	Enables strobes on all pins scrambled to APG Data Generator outputs in the current instruction. Pins must have been previously enabled using the data_strobe() function. Do NOT use READ in the last instruction of a pattern (MAR DONE).
READUDATA	Enables/disables strobes on all pins scrambled to APG Data Generator outputs in the current instruction. Pins must have been previously enabled using the data_strobe() function. The UDATA value in the current instruction is used as a bit-mask, with a logic-1 enabling a strobe and a logic-0 disabling a strobe. UDATA bits 0 through 35 correspond to data generator outputs D0 through D35. This allows strobe control, on a per APG Data Generator output, on a cycle-by-cycle basis. Do NOT use READUDATA to enable strobes in the last instruction of a pattern (MAR DONE).

Table 4.13.9.4-1 MAR Strobe-control Operands

Operand	Purpose
READV	Strobes for data valid (the DUT output voltage is either above voh() or below vol() on all pins scrambled to APG Data Generator outputs in the current instruction. Pins must have been previously enabled using the data_strobe() function. Do NOT use READV in the last instruction of a pattern (MAR DONE).
READZ	Strobes for tri-state (the DUT output voltage is either between voh() and vol()) on all pins scrambled to APG Data Generator outputs in the current instruction. Pins must have been previously enabled using the data_strobe() function. Do NOT use READZ in the last instruction of a pattern (MAR DONE).
NOREAD	Disables strobes on pins scrambled to APG Data Generator outputs in the current instruction. (default)

Example

In the following example, only data generator outputs D4-D7 and D12-D15 have strobes enabled, as controlled using READUDATA plus the UDATA value 0xF0F0 (bits 4-7 and 12-15 = logic-1). Desired operation also assumes that data_strobe() was used to enable strobes for (at least) D4-D7 and D12-D15 prior to executing the test pattern.

```
% MAR        INC, READUDATA
  DATGEN     DATDAT
  UDATA      0xF0F0
  PINFUNC    ADHIZ
```

4.13.9.5 MAR Interrupt Operands

See APG Controller Engine, Memory Test Patterns, MAR Instruction.

Description

The APG contains a real-time interrupt timer. See APG Interrupt Timer for important details about timer operation.

The MAR instruction takes the following form:

```
MAR  Branch-condition, Address, Strobe-control, Interrupt, Timer,
     Misc, BOE-type, Error-choice
```

The table below describes the options available for the MAR Interrupt operands:

Table 4.13.9.5-1 MAR Interrupt Operands

Operand	Purpose
INTEN	If a timer interrupt is pending at the start of an instruction containing this operand, the interrupt subroutine, previously specified using INTADR or INTENADR, will be executed instead of executing the *next* instruction, which is pushed on the stack as the return address from the subroutine.
NOINT	Default. At the start of an instruction containing this operand, the interrupt subroutine will NOT be called, even if an interrupt is pending.
INTADR	Loads the address of the interrupt subroutine, specified in UDATA, into the interrupt address register. The UDATA field may not be used for other purposes in any instruction containing this operand (explicitly or implicitly, see UDATA). At the start of an instruction containing this operand, the interrupt subroutine will NOT be called, even if an interrupt is pending.
INTENADR	Loads the address of the interrupt subroutine, specified in UDATA, into the interrupt address register. The UDATA field may not be used for other purposes in any instruction containing this operand (explicitly or implicitly, see UDATA). If an interrupt is pending at the start of an instruction containing this operand the specified interrupt subroutine will be called instead of executing the next instruction, which is pushed on the stack as the return address from the subroutine.

Note: it is illegal to use INTEN or INTENADR in the same pattern instruction with a MAR conditional branch based on the timer (MAR CJMPT, CRETT, etc.). This rule is enforced by the pattern compiler.

Example

```
// Set the interrupt subroutine address
% MAR     INTADR
  UDATA   Interrupt_routine
```

```
% Loop1: // Enable timer, loop waiting for interrupt
   COUNT   COUNT1, DECR, AON
   MAR     CJMPNZ, Loop1, TIMEN, INTEN
% MAR DONE
// Start of interrupt subroutine
% Interrupt_routine:
% MAR     INC
// ... more instructions as needed
% MAR     RETURN
```

4.13.9.6 MAR Timer Operands

See APG Controller Engine, Memory Test Patterns, MAR Instruction.

Description

The APG contains a real-time interrupt timer. See APG Interrupt Timer.

The MAR instruction takes the following form:

```
MAR  Branch-condition, Address, Strobe-control, Interrupt, Timer,
     Misc, BOE-type, Error-choice
```

The table below describes the options available for the MAR Timer operands:

Table 4.13.9.6-1 MAR Timer Operands

Operand	Purpose
TIMEN	Allows the interrupt timer to count down. If the timer counts-down to 0, an interrupt will be set (pending). See APG Interrupt Timer. Note: this operand must not be used in the first 8 pattern cycles (DBM not used) or 20 pattern cycles (DBM used) of a test pattern. This does not mean 20 pattern instructions, it means 20 tester cycles. Using Magnum 1/2/2x, this operand has an identical effect on VAR Engine interrupt operation. See APG Interrupt Timer. Also review APG Instruction Execution.
RSTTMR	Default. Resets the timer to the value last programmed using the timer() function.

Note: the RSTTMR default resets the timer. The timer has only two conditions, counting down (TIMEN) and reset (RSTTMR or no TIMEN).

4.13.9.7 MAR Misc Operands

See APG Controller Engine, Memory Test Patterns, MAR Instruction.

Description

The MAR instruction takes the following form:

```
MAR  Branch-condition, Address, Strobe-control, Interrupt, Timer,
     Misc, BOE-type, Error-choice
```

The Misc operand is used to control several independent features, thus multiple tokens are allowed in this field:

- Error flag operations (RESET, LATCH, NOLATCH). This requires understanding the two error signal types generated by the hardware. See Error Flag vs. Error Latch.
- Over-programming control (OVER). Used when the test pattern incrementally programs more than one DUT in parallel. See Over-programming Controls and Parallel Test.
- Dynamic DC test pattern trigger(s) (VCOMP). This option triggers the DC Comparators and Error Logic or DC A/D Converter, during Dynamic DC Tests.
- Using Magnum 1/2/2x, specify that a VAR Engine counter is to be tested to determine branch operation (VCNTR).
- Using Magnum 1/2/2x, clear the error source (CLEARERR) specified by the Error-choice operand (see MAR Error-choice Operands).

Note: as noted below, the RESET operand clears the PE error flags and the DC Error Flags in the DC Comparators and Error Logic. The PE error flags have no effect on overall PASS/FAIL result of a functional test. However, the DC Error Flags do effect the overall PASS/FAIL result of Dynamic DC Tests. See Error Flag vs. Error Latch.

The tables below describes the options available for the MAR Misc operands:

Table 4.13.9.7-1 MAR Misc Operands

Operands	Purpose
RESET	Clears all PE error flags, and DC Error Flags. See Error Flag vs. Error Latch. In Magnum 1/2/2x Mixed Memory/Logic Patterns MAR RESET is effectively disabled by PINFUNC VLATCHRESET. Note: RESET (including MAR RESET, VEC RESET, VAR RESET and VPINFUNC RESET) must NOT be used in the same instruction OR the instruction following that using MAR VCOMP, VAR VCOMP, VEC VCOMP, or VPINFUNC VCOMP. Note: RESET may be used in the MAR, VEC/RPT, VAR, VPINFUNC, and CHIPS instruction.
LATCH	Complement of NOLATCH. Any failing strobe(s) generated by an instruction which does not contain an explicit MAR NOLATCH or an explicit VEC/RPT NOLATCH, VAR NOLATCH, or VPINFUNC NOLATCH will set the corresponding PE error latch(es) and cause the test to fail. See Error Flag vs. Error Latch. In Magnum 1/2/2x Mixed Memory/Logic Patterns MAR LATCH is effectively disabled by PINFUNC VLATCHRESET.
NOLATCH	Complement of LATCH. Inhibits failing strobes from setting PE error latches, see Error Flag vs. Error Latch. Also inhibits capturing errors into the Error Catch RAM (ECR). Has no effect on error flags, which controls branch-on-error and stop-on-error decisions. For normal PASS/FAIL testing, the NOLATCH operand is not used, allowing any failing strobes to set the error latches and cause the test to fail. In Magnum 1/2/2x Mixed Memory/Logic Patterns MAR NOLATCH is effectively disabled by PINFUNC VLATCHRESET. Note: NOLATCH may be used in the MAR, VEC/RPT, VAR, and VPINFUNC instructions.

Table 4.13.9.7-1 MAR Misc Operands *(Continued)*

Operands	Purpose
VCOMP	During Dynamic DC Tests, sends a one trigger to the DC Comparators and Error Logic or DC A/D Converter. Desired operation also requires specifying the CompCond argument to the test function (ac_test_supply(), hv_ac_test_supply(), ac_partest()). In Magnum 1/2/2x Mixed Memory/Logic Patterns MAR VCOMP is effectively disabled by PINFUNC VVCOMP. Note: RESET (including MAR RESET, VEC RESET, VAR RESET and VPINFUNC RESET) must NOT be used in the same instruction OR the instruction following that using MAR VCOMP, VAR VCOMP, VEC VCOMP, or VPINFUNC VCOMP. Note: VCOMP may be used in the MAR, VEC/RPT, VAR, and VPINFUNC instruction.
OVER	See Over-programming Controls and Parallel Test. This operand enables special circuitry to inhibit programming stimulus on DUT(s) that have successfully programmed while allowing other DUTs on the same test site to continue programming. In other words, it prevents over-programming. The over_inhibit() function is used to select the programming mechanism that is disabled when the OVER operand is specified. In Magnum 1/2/2x Mixed Memory/Logic Patterns MAR OVER is effectively disabled by PINFUNC VOVER. Note: OVER may be used in the MAR, VEC/RPT, VAR, and VPINFUNC instruction.
VCNTR	In mixed Magnum 1/2/2x patterns, causes a MAR branch decision in the current instruction to test a VAR Engine counter instead of a MAR Engine counter. See Magnum 1/2/2x Memory Pattern Instructions.
CLEARERR	Unconditionally clears all Total Error Counters, Row Error Counters, Col Error Counters and IOC Error Counters. This has the effect of clearing any error signals from these counters. See MAR Error-choice Operands. Does not clear the PE Error Flags.

Table 4.13.9.7-1 MAR Misc Operands *(Continued)*

Operands	Purpose
DEFAULT	Valid only in Mixed Memory/Logic Patterns with the mixedsync attribute (see Pattern Type Attributes). Identifies the current instruction as the default memory instruction to be applied in any subsequent instructions which do not include any explicit memory instructions. See Mixed Memory/Logic Patterns.

Example

The following example resets the PE error flags and DC Error Flags:

```
% MAR RESET
```

In the following instruction, one trigger is sent to the DC Comparators and Error Logic:

```
% MAR INC, VCOMP
```

4.13.9.8 MAR BOE Type Operands

See Algorithmic Pattern Generator (APG), Memory Test Patterns, MAR Instruction.

Note: this section was added in software release h2.0.9.

Description

MAR BOE Type operands are used to select the type of signal used by a branch-on-error operation in the *next instruction executed*. This is required when using *some*, but not all, of the MAR branch-on-error options (more below).

The MAR instruction takes the following form:

```
MAR  Branch-condition, Address, Strobe-control, Interrupt, Timer,
     Misc, BOE-type, Error-choice
```

The rules are as follows:

- Any pattern instruction which performs a branch-on-error operation (i.e. a conditional branch based on errors/failing strobes) must have the appropriate error type selected in the instruction which *executes* immediately preceding the branch instruction. This value changes to 13 instructions when using Logic Test Patterns or Mixed Memory/Logic Patterns. This is done using a MAR BOE-type operand.
- The default APG instruction automatically selects the BOE-type used by the mainstream branch-on-error options (see table below); i.e. the most commonly used branch-on-error operations do not require an explicit BOE-type selection in the prior instruction executed.
- For the other branch-on-error options, the test pattern instruction which executes immediately before the branch instruction must include the appropriate MAR BOE-type selection operand (see table below).
- With regard to BOE-type selection, the branch-on-error instructions can be grouped into error types, as shown in the following table:

Table 4.13.9.8-1 MAR BOE Type Operands

BOE Error Type Selection Operand	MAR Branch-on-error Operand
None (default)	CJMPNE CJMPE CRETNE CRETE CSUBNE CSUBE
ERR_ABORT	CJMPNA CJMPA
ERR_ANOTB	CSUBNE_ANOTB CSUBE_ANOTB CJMPNE_ANOTB CJMPE_ANOTB CRETNE_ANOTB CRETE_ANOTB
ERR_BNOTA	CSUBNE_BNOTA CSUBE_BNOTA CJMPNE_BNOTA CJMPE_BNOTA CRETNE_BNOTA CRETE_BNOTA
ERR_ALL	CSUBNE_ALL CSUBE_ALL CJMPNE_ALL CJMPE_ALL CRETNE_ALL CRETE_ALL

Table 4.13.9.8-1 MAR BOE Type Operands

BOE Error Type Selection Operand	MAR Branch-on-error Operand
ERR_DUT1	CSUBNE_DUT1 CJMPNE_DUT1 CRETNE_DUT1 CSUBE_DUT1 CJMPE_DUT1 CRETE_DUT1
...through...	...through...
ERR_DUT8	CSUBNE_DUT8 CJMPNE_DUT8 CRETNE_DUT8 CSUBE_DUT8 CJMPE_DUT8 CRETE_DUT8

- An explicit `BOE-type` selection in a given pattern instruction directly affects the branch operation of the instruction that executes next, provided that instruction is using a branch-on-error operand. Proper operation requires the correct `BOE-type` selection based on the branch-on-error option executed in the next instruction. The pattern compiler can not check or enforce this rule; i.e. it is the user's responsibility:

  The following is OK because `CJMPNE uses default BOE-type`

  ```
  % MAR     xxx
  % MAR     CJMPNE, myLabel
  ```

  The following is OK because `ERR_DUT1 matches CJMPNE_DUT1`

  ```
  % MAR     xxx, ERR_DUT1
  % MAR     CJMPNE_DUT1, myLabel
  ```

  The following is BAD because `CJMPE_ALL needs ERR_ALL`

  ```
  % MAR     xxx, ERR_DUT1
  % MAR     CJMPE_ALL, myLabel
  ```

  The following is BAD because `CSUBNE_DUT1 needs ERR_DUT1`

```
% MAR     xxx
% MAR     CSUBNE_DUT1, myLabel
```

The following is BAD because CJMPNE needs default

```
% MAR     xxx, ERR_DUT1
% MAR     CJMPNE, myLabel
```

- There are no side effects if a BOE-type selection is repeated for more cycles that are required prior to the instruction which actually performs the branch-on-error; i.e. it is OK to put the BOE-type selection in the error pipeline cycle(s) prior to the branch instruction. For example:

```
% here:
   COUNT   COUNT13, DECR
   MAR     CJMPNZ, here, ERR_DUT1 //OK to use/repeat ERR_DUT1
% MAR     CSUBNE_DUT1, there
```

- It is not legal to include a BOE-type selection operand in an instruction which includes any of the branch-on-error operands:

```
% MAR     CSUBNE_DUT1, myLabel, ERR_DUT1 // BAD
```

- There are no side effects if a BOE-type selection is made in the instruction *prior* to a branch instruction which is not based on error; i.e. branch on counter values, timer state or unconditional execution control operands ignore any prior BOE-type selection.
- The following example shows the minimum requirement for changing BOE-type. This example changes from DUT1 to DUT2:

```
% MAR   CJMPE_DUT1, ...
% MAR   ERR_DUT2
% MAR   CJMPE_DUT2, ...
```

4.13.9.9 MAR Error-choice Operands

See Branch-on-error Logic, MAR Multi-DUT Branch-condition Operands, VAR Multi-DUT Branch-condition Operands.

Description

Using Magnum 1/2/2x, many (most) conditional branch operations of both Memory Test Patterns and Logic Test Patterns depend on:

- A static setup, set using the `mar_error_choice_set()` function, executed from user C-code prior to executing the test pattern. See Static Error Choice Functions, Branch-on-error.
- In Memory Test Patterns (and in the memory instructions of Mixed Memory/Logic Patterns) the MAR `Branch-condition` selection (if used) in each pattern instruction. See MAR Conditional Branch-condition Operands and MAR Multi-DUT Branch-condition Operands.
- In Logic Test Patterns (and in the logic instructions of Mixed Memory/Logic Patterns) the VAR `Branch-condition` selection (if used) in each pattern instruction. See VAR Branch-condition Operands.
- The MAR `Error-choice` selection (explicit or default) in each pattern instruction. In pure Logic Test Patterns the default MAR `Error-choice` value is always selected, more below.
- The MAR `BOE-type` operand selection (explicit or default) in the instructions which executes prior to branch-on-error instruction. This is required when using some, but not all, branch-on-error options, See MAR BOE Type Operands.

The MAR instruction takes the following form:

```
MAR  Branch-condition, Address, Strobe-control, Interrupt, Timer,
     Misc, BOE-type, Error-choice
```

Legal `Error-choice` options are:

Table 4.13.9.9-1 MAR Error-choice Operands

Operand	Comments
ERRSRC1 (default)	Several tables (below) describe how the various combinations of static setup (set using the `mar_error_choice_set()` function) plus these operands and the various conditional branch options operate. See: - Magnum 1 MAR Error-choice Operands, 1 to 8 DUTs per ECR. - MAR Error-choice Operands. - Branch Signal Operand Associations. - Branch Operand Operation: t_errmode1. - Branch Operand Operation: t_errmode2. - Branch Operand Operation: t_errmode3. - Branch Operand Operation: t_errmode4.
ERRSRC2	
ERRSRC3	
ERRSRC4	

Note the following:

- To correctly use the `Error-choice` options requires knowledge of the associated hardware. See Branch-on-error Logic.
- In hardware, the `MAR Error-choice` operand controls the Error Signal MUX which is common to both memory pattern execution and logic pattern execution. See Branch-on-error Logic. As indicated above, in pure Logic Test Patterns the default `MAR Error-choice` value is used.
- The default configuration tests the same PE Error Flags as Maverick-I/-II (see Error Flag vs. Error Latch) and supports testing of up to 4 DUTs per Sub-site.
- In Multi-DUT Test Programs, proper operation of the MAR Multi-DUT Branch-condition Operands and VAR Multi-DUT Branch-condition Operands requires that the DUT board connections to each DUT follow the rules noted in DUT-pin to Tester-pin Connection Requirements.

Note: the following 2 rules were added on 4/21/2006.

During pattern execution, when the `MAR Error-choice` operand must change, proper branch operation requires that the desired error choice be continuously selected for a minimum of 4 cycles (16 using Logic Test Patterns or Mixed Memory/ Logic Patterns) which execute before the instruction which performs the branch operation. This is required to allow time for the new `MAR Error-choice` selection to propagate in the APG hardware. The same `Error-choice` selection must be made in the branch cycle instruction.

- The default APG instruction automatically selects `ERRSRC1`, which is the `MAR Error-choice` operand used by the mainstream branch-on-error options; i.e. the most commonly used branch-on-error operations typically do not require an explicit `MAR Error-choice` operand selection in the prior cycles.
-

The following Magnum 1 table describes how the 8 error signals map to specific DUTs for the various combinations of static error mode and MAR Error-choice. Only 1 table is needed because the relationship between DUT numbers and error signals never change. The table applies when testing 1 to 8 DUTs per sub-site (per ECR); i.e. 2 to 16 DUTs per site). When

fewer DUTs are tested, the error signals are implicitly combined, so the user only needs to consider DUT numbers, never error signals:

Table 4.13.9.9-2 Magnum 1 MAR Error-choice Operands, 1 to 8 DUTs per ECR

		Static Error Mode 1				Static Error Mode 2				Static Error Mode 3				Static Error Mode 4[3]			
MAR Error-choice		ERR SRC1	ERR SRC2	ERR SRC3	ERR SRC4	ERR SRC1	ERR SRC2	ERR SRC3	ERR SRC4	ERR SRC1	ERR SRC2	ERR SRC3	ERR SRC4	ERR SRC1[3]	ERR SRC2	ERR SRC3	ERR SRC4
Sub-site A	ERR1	DUT1	DUT1	DUT1	DUT1	DUT1	DUT1	DUT9	DUT9	DUT1	DUT1	DUT9	DUT9	1/9[2]	1/9[2]	DUT1	DUT9
	ERR3	DUT3	DUT3	DUT3	DUT3	DUT3	DUT3	DUT11	DUT11	DUT3	DUT3	DUT11	DUT11	3/11[2]	3/11[2]	DUT3	DUT11
	ERR5	DUT5	DUT5	DUT5	DUT5	DUT5	DUT5	DUT13	DUT13	DUT5	DUT5	DUT13	DUT13	5/13[2]	5/13[2]	DUT5	DUT13
	ERR7	DUT7	DUT7	DUT7	DUT7	DUT7	DUT7	DUT15	DUT15	DUT7	DUT7	DUT15	DUT15	7/15[2]	7/15[2]	DUT7	DUT15
Sub-site B	ERR2	DUT2	DUT2	DUT2	DUT2	DUT2	DUT2	DUT10	DUT10	DUT2	DUT2	DUT10	DUT10	2/10[2]	2/10[2]	DUT2	DUT10
	ERR4	DUT4	DUT4	DUT4	DUT4	DUT4	DUT4	DUT12	DUT12	DUT4	DUT4	DUT12	DUT12	4/12[2]	4/12[2]	DUT4	DUT12
	ERR6	DUT6	DUT6	DUT6	DUT6	DUT6	DUT6	DUT14	DUT14	DUT6	DUT6	DUT14	DUT14	6/14[2]	6/14[2]	DUT6	DUT14
	ERR8	DUT8	DUT8	DUT8	DUT8	DUT8	DUT8	DUT16	DUT16	DUT8	DUT8	DUT16	DUT16	8/16[2]	8/16[2]	DUT8	DUT16
Branch Signal Type		Err	TEC	REC	CEC	Err	TEC	Err	TEC	REC	CEC	REC	CEC	Err	TEC	Err	Err

Notes:

1) This is also the number of DUTs per 64-pin sub-site.

2) The logical OR of signals from the two DUT numbers shown.

3) Static mode 4 + ERRSRC1 must be used when testing more than 8 DUTs, to test (branch) if all DUTs have (or don't have) an error.

4) The hardware only uses ERRSRC1 when executing pure logic patterns.

Several tables follow:

- MAR Error-choice Operands - describes which signals the hardware selects based on the MAR `Error-choice operand` vs. the static error choice.
- Branch Signal Operand Associations - lists which hardware signal is tested by each branch operand.
- Branch Operand Operation: t_errmode1, Branch Operand Operation: t_errmode2, Branch Operand Operation: t_errmode3 and Branch Operand Operation: t_errmode4 - specify how each branch operand operates for each MAR `Error-choice operand` option.

The following table describes the MAR Error-choice Operands for Magnum 1:

Table 4.13.9.9-3 MAR Error-choice Operands

Static Error Mode	MAR Error-choice	Description
Mode-1 (t_errmode1) Primarily used when testing 1-4 DUTs per Sub-site	ERRSRC1 See Note:	Default. PE Error Flags are routed to Branch Decode Error Logic. Supports 1 to 4 DUTs per Sub-site. Operation matches Maverick-I/-II pattern conditional branch operation. See Mode-4 when testing 4-8 DUTs per Sub-site.
	ERRSRC2 See Note:	TEC Comparator output(s) are routed to Branch Decode Error Logic. Supports 1 to 4 DUTs per Sub-site (each DUT having an independent TEC counter). Enables pattern conditional branch based on whether value(s) in Total Error Counters are greater than the TEC Comparator value (see ecr_compare_reg_set()).
	ERRSRC3 See Note:	REC Comparator output(s) are routed to Branch Decode Error Logic. Supports 1 to 4 DUTs per Sub-site (each DUT having an independent REC counter). Enables pattern conditional branch based on whether value(s) in Row Error Counters are greater than the REC Comparator value (see ecr_compare_reg_set()).
	ERRSRC4 See Note:	CEC Comparator output(s) are routed to Branch Decode Error Logic. Supports 1 to 4 DUTs per Sub-site (each DUT having an independent CEC counter). Enables pattern conditional branch based on whether value(s) in Col Error Counters are greater than the CEC Comparator value (see ecr_compare_reg_set()).
Note: odd numbered DUTs are from Sub-site-A, even numbered DUTs from Sub-site-B.		

Table 4.13.9.9-3 MAR Error-choice Operands *(Continued)*

Static Error Mode	MAR Error-choice	Description
Mode-2 (t_errmode2) Primarily used when testing >4 DUTs per Sub-site. ECR REC Comparators and CEC Comparators are not accessible.	ERRSRC1 See Note:	PE Error Flag for DUTs 1..8 are routed to Branch Decode Error Logic. Supports 1 to 8 DUTs per Sub-site.
	ERRSRC2 See Note:	TEC Comparator outputs for DUTs 1..8 are routed to Branch Decode Error Logic. Supports 1 to 8 DUTs per Sub-site (each DUT having an independent TEC counter). Enables pattern conditional branch based on whether values in Total Error Counters for DUTs 1..8 are greater than the TEC Comparator value (see ecr_compare_reg_set()).
	ERRSRC3 See Note:	PE Error Flag from DUTs 9..16 are routed to Branch Decode Error Logic. Supports 1 to 8 DUTs per Sub-site. Not useful if number of DUTs < 8.
	ERRSRC4 See Note:	TEC Comparator outputs for DUTs 9..16 are routed to Branch Decode Error Logic. Supports 1 to 8 DUTs per Sub-site (each DUT having an independent TEC counter). Enables pattern conditional branch based on whether values in Total Error Counters for DUTs 9..16 are greater than the TEC Comparator value (see ecr_compare_reg_set()). Not useful if number of DUTs < 8.
Note: odd numbered DUTs are from Sub-site-A, even numbered DUTs from Sub-site-B.		

Table 4.13.9.9-3 MAR Error-choice Operands *(Continued)*

Static Error Mode	MAR Error-choice	Description
Mode-3 (t_errmode3) Primarily used when testing >4 DUTs per Sub-site. Error flags are not accessible.	ERRSRC1 See Note:	REC Comparator outputs for DUTs 1..8 are routed to Branch Decode Error Logic. Supports 1 to 8 DUTs per Sub-site (each DUT having an independent REC counter). Enables pattern conditional branch based on whether values in Row Error Counters for DUTs 1..8 are greater than the REC Comparator value (see ecr_compare_reg_set()).
	ERRSRC2 See Note:	CEC Comparator outputs for DUTs 1..8 are routed to Branch Decode Error Logic. Supports 1 to 8 DUTs per Sub-site (each DUT having an independent CEC counter). Enables pattern conditional branch based on whether values in Col Error Counters for DUTs 1..8 are greater than the CEC Comparator value (see ecr_compare_reg_set()).
	ERRSRC3 See Note:	REC Comparator outputs for DUTs 9..16 are routed to Branch Decode Error Logic. Supports 1 to 8 DUTs per Sub-site (each DUT having an independent REC counter). Enables pattern conditional branch based on whether values in Row Error Counters for DUTs 9..16 are greater than the REC Comparator value (see ecr_compare_reg_set()). Not useful if number of DUTs < 8.
	ERRSRC4 See Note:	CEC Comparator outputs for DUTs 9..16 are routed to Branch Decode Error Logic. Supports 1 to 8 DUTs per Sub-site (each DUT having an independent CEC counter). Enables pattern conditional branch based on whether values in Col Error Counters for DUTs 9..16 are greater than the CEC Comparator value (see ecr_compare_reg_set()). Not useful if number of DUTs < 8.
Note: odd numbered DUTs are from Sub-site-A, even numbered DUTs from Sub-site-B.		

Table 4.13.9.9-3 MAR Error-choice Operands *(Continued)*

Static Error Mode	MAR Error-choice	Description
Mode-4 (t_errmode4) Primarily used when testing >4 DUTs per Sub-site.	ERRSRC1 See Note:	PE Error Flags are routed to Branch Decode Error Logic. Supports 1 to 8 DUTs per Sub-site but the logical OR of two DUT's error flags are combined into the output of a given Error Signal MUX. DUTs are paired (OR'ed) 1/9, 2/10, etc.
	ERRSRC2 See Note:	TEC Comparator outputs are routed to Branch Decode Error Logic. Supports 1 to 8 DUTs per Sub-site but the logical OR of two DUT's TEC Comparators are combined into the output of a given Error Signal MUX. DUTs are paired 1/9, 2/10, etc. Enables pattern conditional branch based on whether value(s) in Total Error Counters are greater than the TEC Comparator value (see ecr_compare_reg_set()).
	ERRSRC3 See Note:	PE Error Flags are routed to Branch Decode Error Logic. Supports 1 to 8 DUTs per Sub-site but only error flags for DUTs 1-8 are output from the Error Signal MUX (1/3/5/7 from Sub-site-A, 2/4/6/8 from Sub-site-B).
	ERRSRC4 See Note:	PE Error Flags are routed to Branch Decode Error Logic. Supports 1 to 8 DUTs per Sub-site but only error flags for DUTs 9-16 are output from the Error Signal MUX (DUTs 9/11/13/15 from Sub-site-A, DUTs 10/12/14/16 from Sub-site-B).
Note: odd numbered DUTs are from Sub-site-A, even numbered DUTs from Sub-site-B.		

During pattern execution, in each pattern instruction, the conditional branch operand (see MAR Conditional Branch-condition Operands & MAR Multi-DUT Branch-condition Operands) causes the Branch Signal Selection MUX to select one input, which is routed to the APG Controller Engine to control the conditional branch operation in that instruction. See Note:

The following table lists the branch signal tested by most of the MAR Branch Condition Operands (timer and counter branch signals not included):

Table 4.13.9.9-4 Branch Signal Operand Associations

Branch Signal	Operand
Error	CJMPE, CJMPNE, CRETE, CRETNE, CSUBE, CSUBNE
A_not_B	CSUBNE_ANOTB, CSUBE_ANOTB, CJMPNE_ANOTB, CJMPE_ANOTB, CRETNE_ANOTB, CRETE_ANOTB
B_not_A	CSUBNE_BNOTA, CSUBE_BNOTA, CJMPNE_BNOTA, CJMPE_BNOTA, CRETNE_BNOTA, CRETE_BNOTA
Abort	CJMPA, CJMPNA These test PE error latches, not error flags. See Error Flag vs. Error Latch.
AllDUTs	CSUBNE_ALL, CSUBE_ALL, CJMPNE_ALL, CJMPE_ALL, CRETNE_ALL, CRETE_ALL
DUT1 thru DUT8 (see note)	CSUBNE_DUT1 thru CSUBNE_DUT8 CSUBE_DUT1 thru CSUBE_DUT8 CJMPNE_DUT1 thru CJMPNE_DUT8 CJMPE_DUT1 thru CJMPE_DUT8 CRETNE_DUT1 thru CRETNE_DUT8 CRETE_DUT1 thru CRETE_DUT8

Notes:
1) When using branch-on-DUT options, the terms DUT1 thru DUT8 represents DUT9 thru DUT16 in static mode-2 and static mode-3. See Static Error Choice Functions, Branch-on-error.
2) Except for the first row above, the other branch-on-error options have an additonal pattern requirement, as noted in MAR BOE Type Operands.

Four tables are presented below, one for each static error mode (see Static Error Choice Functions, Branch-on-error. The following general rules apply:

- Odd numbered DUTs are on Sub-site-A, even numbered DUTs on Sub-site-B.
- Pure Logic Test Patterns default to ERRSRC1.
- t_errmode1 + ERRSRC1 = Maverick-I/-II operation with up to 4 DUTs per Sub-site (4 DUTs per 64 pins).

- In the tables below, the term branch is used to refers to any test pattern jump, subroutine call or subroutine return operation.
- Tests of ECR counters (Total Error Counters, Row Error Counters/REC, and Col Error Counters/CEC) are actually testing whether the counter value matches its corresponding counter comparator value (see `ecr_compare_reg_set()`):

Table 4.13.9.9-5 Branch Operand Operation: t_errmode1

MAR Error Choice → / MAR Branch Operand (link to VAR Operand) ↓	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
	Tests error Flags 1-4 DUTs per Sub-site	Tests ECR Total Error Counters 1-4 DUTs per Sub-site	Tests ECR Row Error Counters 1-4 DUTs per Sub-site	Tests ECR Col Error Counters 1-4 DUTs per Sub-site
CJMPNA (CJMPNA)	Branch if at lease one DUT has no errors	Branch if at least one TEC ≠ count	Branch if at least one REC ≠ count	Branch if at least one CEC ≠ count
CJMPA (CJMPA)	Branch if all DUTs have an error	Branch if TEC for all DUTs = count	Branch if REC for all DUTs = count	Branch if CEC for all DUTs = count
CJMPNE (CJMPNE) CRETNE (CRETNE) CSUBNE (CSUBNE)	Branch if no DUTs have an error	Branch if TECs for all DUTs ≠ count	Branch if RECs for all DUTs ≠ count	Branch if CECs for all DUTs ≠ count
CJMPE (CJMPE) CRETE (CRETE) CSUBE (CSUBE)	Branch if at least one DUT has an error	Branch if TECs for at least one DUT = count	Branch if RECs for at least one DUT = count	Branch if CECs for at least one DUT = count

Table 4.13.9.9-5 Branch Operand Operation: t_errmode1 *(Continued)*

MAR Error Choice → / MAR Branch Operand (link to VAR Operand) ↓	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
CSUBNE_ANOTB (CSUBNE_ANOTB) CJMPNE_ANOTB (CJMPNE_ANOTB) CRETNE_ANOTB (CRETNE_ANOTB)	Branch if no DUTs on Sub-site-A have an error OR any DUT on Sub-site-B has an error	Branch if no TECs for DUTs on Sub-site-A = count OR any TEC for a DUT on Sub-site-B = count	Branch if no RECs for DUTs on Sub-site-A = count OR any REC for a DUT on Sub-site-B = count	Branch if no CECs for DUTs on Sub-site-A = count OR any CEC for a DUT on Sub-site-B = count
CSUBE_ANOTB (CSUBE_ANOTB) CJMPE_ANOTB (CJMPE_ANOTB) CRETE_ANOTB (CRETE_ANOTB)	Branch if any DUT on Sub-site-A has an error AND no DUTs on Sub-site-B have an error	Branch if TEC for any DUT on Sub-site-A = count AND no TEC for DUTs on Sub-site-B = count	Branch if REC for any DUT on Sub-site-A = count AND no REC for DUTs on Sub-site-B = count	Branch if CEC for any DUT on Sub-site-A = count AND no CEC for DUTs on Sub-site-B = count
CSUBNE_BNOTA (CSUBNE_BNOTA) CJMPNE_BNOTA (CJMPNE_BNOTA) CRETNE_BNOTA (CRETNE_BNOTA)	Branch if no DUTs on Sub-site-B have an error OR any DUT on Sub-site-A has an error	Branch if no TECs for DUTs on Sub-site-B = count OR any TEC for a DUT on Sub-site-A = count	Branch if no RECs for DUTs on Sub-site-B = count OR any REC for a DUT on Sub-site-A = count	Branch if no CECs for DUTs on Sub-site-B = count OR any CEC for a DUT on Sub-site-A = count

Table 4.13.9.9-5 Branch Operand Operation: t_errmode1 *(Continued)*

MAR Error Choice → / MAR Branch Operand (link to VAR Operand) ↓	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
CSUBE_BNOTA (CSUBE_BNOTA) CJMPE_BNOTA (CJMPE_BNOTA) CRETE_BNOTA (CRETE_BNOTA)	Branch if no DUTs on Sub-site-A have an error AND any DUT on Sub-site-B has an error	Branch if no TECs for DUTs on Sub-site-A = count AND TEC for any DUT on Sub-site-B = count	Branch if no RECs for DUTs on Sub-site-A = count AND REC for any DUT on Sub-site-B = count	Branch if no CECs for DUTs on Sub-site-A = count AND CEC for any DUT on Sub-site-B = count
CSUBNE_ALL (CSUBNE_ALL) CJMPNE_ALL (CJMPNE_ALL) CRETNE_ALL (CRETNE_ALL)	Branch if any DUT on Sub-site-A doesn't have an error OR if any DUT on Sub-site-B doesn't have an error	Branch if TEC for any DUT on Sub-site-A ≠ count OR if TEC for any DUT on Sub-site-B ≠ count	Branch if REC for any DUT on Sub-site-A ≠ count OR if REC for any DUT on Sub-site-B ≠ count	Branch if CEC for any DUT on Sub-site-A ≠ count OR if CEC for any DUT on Sub-site-B ≠ count

Table 4.13.9.9-5 Branch Operand Operation: t_errmode1 *(Continued)*

MAR Error Choice → / MAR Branch Operand (link to VAR Operand) ↓	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
CSUBE_ALL (CSUBE_ALL) CJMPE_ALL (CJMPE_ALL) CRETE_ALL (CRETE_ALL)	Branch if every DUT on Sub-site-A has an error AND every DUT on Sub-site-B has an error	Branch if TEC for every DUT on Sub-site-A = count AND TEC for every DUT on Sub-site-B = count	Branch if REC for every DUT on Sub-site-A = count AND REC for every DUT on Sub-site-B = count	Branch if CEC for every DUT on Sub-site-A = count AND CEC for every DUT on Sub-site-B = count
CSUBNE_DUT1 (CSUBNE_DUT1) ...thru... CSUBNE_DUT8 (CSUBNE_DUT8) CJMPNE_DUT1 (CJMPNE_DUT1) ...thru... CJMPNE_DUT8 (CJMPNE_DUT8) CRETNE_DUT1 (CRETNE_DUT1) ...thru... CRETNE_DUT8 (CRETNE_DUT8)	Branch if the specified DUT has no errors	Branch if the TEC for the specified DUT ≠ count	Branch if the REC for the specified DUT ≠ count	Branch if the CEC for the specified DUT ≠ count

Table 4.13.9.9-5 Branch Operand Operation: t_errmode1 *(Continued)*

MAR Error Choice → / MAR Branch Operand (link to VAR Operand) ↓	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
	Tests error Flags 1-4 DUTs per Sub-site	Tests ECR Total Error Counters 1-4 DUTs per Sub-site	Tests ECR Row Error Counters 1-4 DUTs per Sub-site	Tests ECR Col Error Counters 1-4 DUTs per Sub-site
CSUBE_DUT1 (CSUBE_DUT1) ...thru... CSUBE_DUT8 (CSUBE_DUT8) CJMPE_DUT1 (CJMPE_DUT1) ...thru... CJMPE_DUT8 (CJMPE_DUT8) CRETE_DUT1 (CRETE_DUT1) ...thru... CRETE_DUT8 (CRETE_DUT8)	Branch if the specified DUT has an error	Branch if the TEC for the specified DUT = count	Branch if the REC for the specified DUT = count	Branch if the CEC for the specified DUT = count

Static Error = t_errmode2 (see Static Error Choice Functions, Branch-on-error):

Table 4.13.9.9-6 Branch Operand Operation: t_errmode2

MAR Error Choice → / MAR Branch Operand (link to VAR Operand) ↓	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
CJMPNA (CJMPNA)	Branch if at lease one of DUTs 1-8 has no errors	Branch if at least one TEC for DUTs 1-8 ≠ count	Branch if at lease one of DUTs 9-16 has no errors	Branch if at least one TEC for DUTs 9-16 ≠ count
CJMPA (CJMPA)	Branch if DUTs 1-8 each have an error	Branch if the TEC for each of DUTs 1-8 = count	Branch if DUTs 9-16 each have an error	Branch if DUTs 9-16 each have an error
CJMPNE (CJMPNE) CRETNE (CRETNE) CSUBNE (CSUBNE)	Branch if none of DUTs 1-8 have an error	Branch if the TECs for DUTs 1-8 are all ≠ count	Branch if none of DUTs 9-16 have an error	Branch if the TECs for DUTs 9-16 are all ≠ count
CJMPE (CJMPE) CRETE (CRETE) CSUBE (CSUBE)	Branch if at least one of DUTs 1-8 has an error	Branch if the TEC for at least one of DUTs 1-8 = count	Branch if at least one of DUTs 9-16 has an error	Branch if the TEC for at least one of DUTs 9-16 = count

Table 4.13.9.9-6 Branch Operand Operation: t_errmode2 *(Continued)*

MAR Error Choice → / MAR Branch Operand (link to VAR Operand) ↓	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
CSUBNE_ANOTB (CSUBNE_ANOTB) CJMPNE_ANOTB (CJMPNE_ANOTB) CRETNE_ANOTB (CRETNE_ANOTB)	Branch if none of DUTs 1-8 on Sub-site-A have an error OR any of DUTs 1-8 on Sub-site-B has an error	Branch if no TECs for DUTs 1-8 on Sub-site-A = count OR the TEC for any of DUTs 1-8 on Sub-site-B = count	Branch if none of DUTs 9-16 on Sub-site-A have an error OR any of DUTs 9-16 on Sub-site-B has an error	Branch if no TECs for DUTs 9-16 on Sub-site-A = count OR the TEC for any of DUTs 9-16 on Sub-site-B = count
CSUBE_ANOTB (CSUBE_ANOTB) CJMPE_ANOTB (CJMPE_ANOTB) CRETE_ANOTB (CRETE_ANOTB)	Branch if any of DUTs 1-8 on Sub-site-A has an error AND none of DUTs 1-8 on Sub-site-B have an error	Branch if the TEC for any of DUTs 1-8 on Sub-site-A = count AND no TECs for DUTs 1-8 on Sub-site-B = count	Branch if any of DUTs 9-16 on Sub-site-A has an error AND none of DUTs 9-16 on Sub-site-B have an error	Branch if the TEC for any of DUTs 9-16 on Sub-site-A = count AND no TECs for DUTs 9-16 on Sub-site-B = count
CSUBNE_BNOTA (CSUBNE_BNOTA) CJMPNE_BNOTA (CJMPNE_BNOTA) CRETNE_BNOTA (CRETNE_BNOTA)	Branch if none of DUTs 1-8 on Sub-site-B have an error OR any of DUTs 1-8 on Sub-site-A has an error	Branch if no TECs for DUTs 1-8 on Sub-site-B = count OR the TEC for any of DUTs 1-8 on Sub-site-A = count	Branch if none of DUTs 9-16 on Sub-site-B have an error OR any of DUTs 9-16 on Sub-site-A has an error	Branch if no TECs for DUTs 9-16 on Sub-site-B = count OR the TEC for any of DUTs 9-16 on Sub-site-A = count

Table 4.13.9.9-6 Branch Operand Operation: t_errmode2 *(Continued)*

MAR Error Choice → / MAR Branch Operand (link to VAR Operand) ↓	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
CSUBE_BNOTA (CSUBE_BNOTA) CJMPE_BNOTA (CJMPE_BNOTA) CRETE_BNOTA (CRETE_BNOTA)	Branch if none of DUTs 1-8 on Sub-site-A have an error AND any of DUTs 1-8 on Sub-site-B has an error	Branch if no TECs for DUTs 1-8 on Sub-site-A = count AND the TEC for any of DUTs 1-8 on Sub-site-B = count	Branch if none of DUTs 9-16 on Sub-site-A have an error AND any of DUTs 9-16 on Sub-site-B has an error	Branch if no TECs for DUTs 9-16 on Sub-site-A = count AND the TEC for any of DUTs 9-16 on Sub-site-B = count
CSUBNE_ALL (CSUBNE_ALL) CJMPNE_ALL (CJMPNE_ALL) CRETNE_ALL (CRETNE_ALL)	Branch if any of DUTs 1-8 on Sub-site-A doesn't have an error OR if any of DUTs 1-8 on Sub-site-B doesn't have an error	Branch if the TEC for any of DUTs 1-8 on Sub-site-A ≠ count OR if the TEC for any of DUTs 1-8 on Sub-site-B ≠ count	Branch if any of DUTs 9-16 on Sub-site-A doesn't have an error OR if any of DUTs 9-16 on Sub-site-B doesn't have an error	Branch if the TEC for any of DUTs 9-16 on Sub-site-A ≠ count OR if the TEC for any of DUTs 9-16 on Sub-site-B ≠ count

Table 4.13.9.9-6 Branch Operand Operation: t_errmode2 *(Continued)*

MAR Error Choice → / MAR Branch Operand (link to VAR Operand) ↓	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
CSUBE_ALL (CSUBE_ALL) CJMPE_ALL (CJMPE_ALL) CRETE_ALL (CRETE_ALL)	Branch if DUTs 1-8 on Sub-site-A each has an error AND if DUTs 1-8 on Sub-site-B each has an error	Branch if the TEC for DUTs 1-8 on Sub-site-A each = count AND the TEC for DUTs 1-8 on Sub-site-B each = count	Branch if DUTs 9-16 on Sub-site-A each has an error AND if DUTs 9-16 on Sub-site-B each has an error	Branch if the TEC for DUTs 9-16 on Sub-site-A each = count AND the TEC for DUTs 9-16 on Sub-site-B each = count
CSUBNE_DUT1 (CSUBNE_DUT1) ...thru... CSUBNE_DUT8 (CSUBNE_DUT8) CJMPNE_DUT1 (CJMPNE_DUT1) ...thru... CJMPNE_DUT8 (CJMPNE_DUT8) CRETNE_DUT1 (CRETNE_DUT1) ...thru... CRETNE_DUT8 (CRETNE_DUT8)	Branch if the specified DUT 1-8 has no errors	Branch if the TEC for the specified DUT ≠ count	Branch if the specified DUT 1-8 has no errors. Note that DUT1 actually means DUT9, DUT2 = DUT10, etc.	Branch if the TEC for the specified DUT ≠ count. Note that DUT1 actually means DUT9, DUT2 = DUT10, etc.

Table 4.13.9.9-6 Branch Operand Operation: t_errmode2 *(Continued)*

MAR Error Choice → / MAR Branch Operand (link to VAR Operand) ↓	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
CSUBE_DUT1 (CSUBE_DUT1) ...thru... CSUBE_DUT8 (CSUBE_DUT8) CJMPE_DUT1 (CJMPE_DUT1) ...thru... CJMPE_DUT8 (CJMPE_DUT8) CRETE_DUT1 (CRETE_DUT1) ...thru... CRETE_DUT8 (CRETE_DUT8)	Branch if the specified DUT has an error	Branch if the TEC for the specified DUT = count	Branch if the specified DUT has an error Note that DUT1 actually means DUT9, DUT2 = DUT10, etc.	Branch if the TEC for the specified DUT = count Note that DUT1 actually means DUT9, DUT2 = DUT10, etc.
MAR Branch Operand (link to VAR Operand) ↓	Tests error flags for DUTs 1-8 only	Tests ECR Total Error Counters for DUTs 1-8 only	Tests error flags for DUTs 9-16 only	Tests ECR Total Error Counters for DUTs 9-16 only

Static Error = t_errmode3 (see Static Error Choice Functions, Branch-on-error):

Table 4.13.9.9-7 Branch Operand Operation: t_errmode3

MAR Error Choice → / MAR Branch Operand (link to VAR Operand) ↓	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
	Tests ECR Row Error Counters for DUTs 1-8 only	Tests ECR Col Error Counters for DUTs 1-8 only	Tests ECR Row Error Counters for DUTs 9-16 only	Tests ECR Col Error Counters for DUTs 9-16 only
CJMPNA (CJMPNA)	Branch if the REC for at lease one of DUTs 1-8 ≠ count	Branch if the CEC for at lease one of DUTs 1-8 ≠ count	Branch if the REC for at lease one of DUTs 9-16 ≠ count	Branch if the CEC for at lease one of DUTs 9-16 ≠ count
CJMPA (CJMPA)	Branch if each REC for DUTs 1-8 = count	Branch if each CEC for DUTs 1-8 = count	Branch if each REC for DUTs 9-16 = count	Branch if each CEC for DUTs 9-16 = count
CJMPNE (CJMPNE) CRETNE (CRETNE) CSUBNE (CSUBNE)	Branch if each REC for DUTs 1-8 ≠ count	Branch if each CEC for DUTs 1-8 ≠ count	Branch if each REC for DUTs 9-16 ≠ count	Branch if each CEC for DUTs 9-16 ≠ count
CJMPE (CJMPE) CRETE (CRETE) CSUBE (CSUBE)	Branch if the REC for at least one of DUTs 1-8 = count	Branch if the CEC for at least one of DUTs 1-8 = count	Branch if the REC for at least one of DUTs 9-16 = count	Branch if the CEC for at least one of DUTs 9-16 = count

Table 4.13.9.9-7 Branch Operand Operation: t_errmode3 *(Continued)*

MAR Error Choice → / MAR Branch Operand (link to VAR Operand) ↓	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
	Tests ECR Row Error Counters for DUTs 1-8 only	Tests ECR Col Error Counters for DUTs 1-8 only	Tests ECR Row Error Counters for DUTs 9-16 only	Tests ECR Col Error Counters for DUTs 9-16 only
CSUBNE_ANOTB (CSUBNE_ANOTB) CJMPNE_ANOTB (CJMPNE_ANOTB) CRETNE_ANOTB (CRETNE_ANOTB)	Branch if no RECs for DUTs 1-8 on Sub-site-A = count OR the REC for any of DUTs 1-8 on Sub-site-B = count	Branch if no CECs for DUTs 1-8 on Sub-site-A = count OR the CEC for any of DUTs 1-8 on Sub-site-B = count	Branch if no RECs for DUTs 9-16 on Sub-site-A = count OR the REC for any of DUTs 9-16 on Sub-site-B = count	Branch if no CECs for DUTs 9-16 on Sub-site-A = count OR the CEC for any of DUTs 9-16 on Sub-site-B = count
CSUBE_ANOTB (CSUBE_ANOTB) CJMPE_ANOTB (CJMPE_ANOTB) CRETE_ANOTB (CRETE_ANOTB)	Branch if the REC for any of DUTs 1-8 on Sub-site-A = count AND no REC for DUTs 1-8 on Sub-site-B = count	Branch if the CEC for any of DUTs 1-8 on Sub-site-A = count AND no CEC for DUTs 1-8 on Sub-site-B = count	Branch if the REC for any of DUTs 9-16 on Sub-site-A = count AND no REC for DUTs 9-16 on Sub-site-B = count	Branch if the CEC for any of DUTs 9-16 on Sub-site-A = count AND no CEC for DUTs 9-16 on Sub-site-B = count
CSUBNE_BNOTA (CSUBNE_BNOTA) CJMPNE_BNOTA (CJMPNE_BNOTA) CRETNE_BNOTA (CRETNE_BNOTA)	Branch if no RECs for DUTs 1-8 on Sub-site-B = count OR the REC for any of DUTs 1-8 on Sub-site-A = count	Branch if no CECs for DUTs 1-8 on Sub-site-B = count OR the CEC for any of DUTs 1-8 on Sub-site-A = count	Branch if no RECs for DUTs 9-16 on Sub-site-B = count OR the REC for any of DUTs 9-16 on Sub-site-A = count	Branch if no CECs for DUTs 9-16 on Sub-site-B = count OR the CEC for any of DUTs 9-16 on Sub-site-A = count

Table 4.13.9.9-7 Branch Operand Operation: t_errmode3 *(Continued)*

MAR Error Choice → / MAR Branch Operand (link to VAR Operand) ↓	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
	Tests ECR Row Error Counters for DUTs 1-8 only	Tests ECR Col Error Counters for DUTs 1-8 only	Tests ECR Row Error Counters for DUTs 9-16 only	Tests ECR Col Error Counters for DUTs 9-16 only
CSUBE_BNOTA (CSUBE_BNOTA) CJMPE_BNOTA (CJMPE_BNOTA) CRETE_BNOTA (CRETE_BNOTA)	Branch if no RECs for DUTs 1-8 on Sub-site-A = count AND the REC for any of DUTs 1-8 on Sub-site-B = count	Branch if no CECs for DUTs 1-8 on Sub-site-A = count AND the CEC for any of DUTs 1-8 on Sub-site-B = count	Branch if no RECs for DUTs 9-16 on Sub-site-A = count AND the REC for any of DUTs 9-16 on Sub-site-B = count	Branch if no CECs for DUTs 9-16 on Sub-site-A = count AND the CEC for any of DUTs 9-16 on Sub-site-B = count
CSUBNE_ALL (CSUBNE_ALL) CJMPNE_ALL (CJMPNE_ALL) CRETNE_ALL (CRETNE_ALL)	Branch if the REC for any of DUTs 1-8 on Sub-site-A ≠ count OR if the REC for any of DUTs 1-8 on Sub-site-B ≠ count	Branch if the CEC for any DUTs 1-8 on Sub-site-A ≠ count OR if the CEC for any of DUTs 1-8 on Sub-site-B ≠ count	Branch if the REC for any of DUTs 9-16 on Sub-site-A ≠ count OR if the REC for any of DUTs 9-16 on Sub-site-B ≠ count	Branch if the CEC for any DUTs 9-16 on Sub-site-A ≠ count OR if the CEC for any of DUTs 9-16 on Sub-site-B ≠ count

Table 4.13.9.9-7 Branch Operand Operation: t_errmode3 *(Continued)*

MAR Error Choice → / MAR Branch Operand (link to VAR Operand) ↓	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
	Tests ECR Row Error Counters for DUTs 1-8 only	Tests ECR Col Error Counters for DUTs 1-8 only	Tests ECR Row Error Counters for DUTs 9-16 only	Tests ECR Col Error Counters for DUTs 9-16 only
CSUBE_ALL (CSUBE_ALL) CJMPE_ALL (CJMPE_ALL) CRETE_ALL (CRETE_ALL)	Branch if the RECs for each of DUTs 1-8 on Sub-site-A = count AND the RECs for each of DUTs 1-8 on Sub-site-B = count	Branch if the CECs for each of DUTs 1-8 on Sub-site-A = count AND the CECs for each of DUTs 1-8 on Sub-site-B = count	Branch if the RECs for each of DUTs 9-16 on Sub-site-A = count AND the RECs for each of DUTs 9-16 on Sub-site-B = count	Branch if the CECs for each of DUTs 9-16 on Sub-site-A = count AND the CECs for each of DUTs 9-16 on Sub-site-B = count
CSUBNE_DUT1 (CSUBNE_DUT1) ...thru... CSUBNE_DUT8 (CSUBNE_DUT8) CJMPNE_DUT1 (CJMPNE_DUT1) ...thru... CJMPNE_DUT8 (CJMPNE_DUT8) CRETNE_DUT1 (CRETNE_DUT1) ...thru... CRETNE_DUT8 (CRETNE_DUT8)	Branch if the REC for the specified DUT ≠ count	Branch if the CEC for the specified DUT ≠ count	Branch if the REC for the specified DUT ≠ count. Note that DUT1 actually means DUT9, DUT2 = DUT10, etc.	Branch if the CEC for the specified DUT ≠ count. Note that DUT1 actually means DUT9, DUT2 = DUT10, etc.

Table 4.13.9.9-7 Branch Operand Operation: t_errmode3 *(Continued)*

MAR Error Choice →	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
MAR Branch Operand (link to VAR Operand) ↓	Tests ECR Row Error Counters for DUTs 1-8 only	Tests ECR Col Error Counters for DUTs 1-8 only	Tests ECR Row Error Counters for DUTs 9-16 only	Tests ECR Col Error Counters for DUTs 9-16 only
CSUBE_DUT1 (CSUBE_DUT1) ...thru... CSUBE_DUT8 (CSUBE_DUT8) CJMPE_DUT1 (CJMPE_DUT1) ...thru... CJMPE_DUT8 (CJMPE_DUT8) CRETE_DUT1 (CRETE_DUT1) ...thru... CRETE_DUT8 (CRETE_DUT8)	Branch if the REC for the specified DUT = count	Branch if the CEC for the specified DUT = count	Branch if the REC for the specified DUT = count. Note that DUT1 actually means DUT9, DUT2 = DUT10, etc.	Branch if the CEC for the specified DUT = count. Note that DUT1 actually means DUT9, DUT2 = DUT10, etc.

Static Error = t_errmode4 (see Static Error Choice Functions, Branch-on-error):

Table 4.13.9.9-8 Branch Operand Operation: t_errmode4

MAR Error Choice → / MAR Branch Operand (link to VAR Operand) ↓	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
	Tests error Flags 1-8 DUTs per Sub-site Logical OR of DUTs 1/9, 2/10, 3/11, etc.	Tests ECR Total Error Counters 1-8 DUTs per Sub-site Logical OR of DUTs 1/9, 2/10, 3/11, etc.	Tests error flags for DUTs 1-8 only	Tests error flags for DUTs 9-16 only
CJMPNA (CJMPNA)	Branch if at lease one DUT-pair has no errors	Branch if TECs for at least one DUT-pair are both ≠ count	Branch if at lease one of DUTs 1-8 has no errors	Branch if at lease one of DUTs 9-16 has no errors
CJMPA (CJMPA)	Branch if at least one DUT of each DUT-pair has an error	Branch if TEC for at least one DUT of each DUT-pair = count	Branch if DUTs 1-8 each have an error	Branch if DUTs 9-16 each have an error
CJMPNE (CJMPNE) CRETNE (CRETNE) CSUBNE (CSUBNE)	Branch if no DUTs have an error	Branch if TECs for all DUTs ≠ count	Branch if none of DUTs 1-8 have an error	Branch if none of DUTs 9-16 have an error
CJMPE (CJMPE) CRETE (CRETE) CSUBE (CSUBE)	Branch if at least one DUT has an error	Branch if TEC for at least one DUT = count	Branch if at lease one of DUTs 1-8 has an error	Branch if at lease one of DUTs 9-16 has an error

Table 4.13.9.9-8 Branch Operand Operation: t_errmode4 *(Continued)*

MAR Error Choice → / MAR Branch Operand (link to VAR Operand) ↓	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
	Tests error Flags 1-8 DUTs per Sub-site Logical OR of DUTs 1/9, 2/10, 3/11, etc.	Tests ECR Total Error Counters 1-8 DUTs per Sub-site Logical OR of DUTs 1/9, 2/10, 3/11, etc.	Tests error flags for DUTs 1-8 only	Tests error flags for DUTs 9-16 only
CSUBNE_ANOTB (CSUBNE_ANOTB) CJMPNE_ANOTB (CJMPNE_ANOTB) CRETNE_ANOTB (CRETNE_ANOTB)	Branch if no DUTs on Sub-site-A have an error OR any DUT on Sub-site-B has an error	Branch if no TECs for DUTs on Sub-site-A = count OR any TEC for a DUT on Sub-site-B = count	Branch if none of DUTs 1-8 on Sub-site-A have an error OR any of DUTs 1-8 on Sub-site-B has an error	Branch if none of DUTs 9-16 on Sub-site-A have an error OR any of DUTs 9-16 on Sub-site-B has an error
CSUBE_ANOTB (CSUBE_ANOTB) CJMPE_ANOTB (CJMPE_ANOTB) CRETE_ANOTB (CRETE_ANOTB)	Branch if any DUT on Sub-site-A has an error AND no DUTs on Sub-site-B have an error	Branch if TEC for any DUT on Sub-site-A = count AND no TEC for DUTs on Sub-site-B = count	Branch if any of DUTs 1-8 on Sub-site-A has an error AND none of DUTs 1-8 on Sub-site-B have an error	Branch if any of DUTs 9-16 on Sub-site-A has an error AND none of DUTs 9-16 on Sub-site-B have an error
CSUBNE_BNOTA (CSUBNE_BNOTA) CJMPNE_BNOTA (CJMPNE_BNOTA) CRETNE_BNOTA (CRETNE_BNOTA)	Branch if no DUTs on Sub-site-B have an error OR any DUT on Sub-site-A has an error	Branch if no TECs for DUTs on Sub-site-B = count OR any TEC for a DUT on Sub-site-A = count	Branch if none of DUTs 1-8 on Sub-site-B have an error OR any of DUTs 1-8 on Sub-site-A has an error	Branch if none of DUTs 9-16 on Sub-site-B have an error OR any of DUTs 9-16 on Sub-site-A has an error

Table 4.13.9.9-8 Branch Operand Operation: t_errmode4 *(Continued)*

MAR Error Choice → / MAR Branch Operand (link to VAR Operand) ↓	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
	Tests error Flags 1-8 DUTs per Sub-site Logical OR of DUTs 1/9, 2/10, 3/11, etc.	Tests ECR Total Error Counters 1-8 DUTs per Sub-site Logical OR of DUTs 1/9, 2/10, 3/11, etc.	Tests error flags for DUTs 1-8 only	Tests error flags for DUTs 9-16 only
CSUBE_BNOTA (CSUBE_BNOTA) CJMPE_BNOTA (CJMPE_BNOTA) CRETE_BNOTA (CRETE_BNOTA)	Branch if no DUTs on Sub-site-A have an error AND any DUT on Sub-site-B has an error	Branch if no TECs for DUTs on Sub-site-A = count AND TEC for any DUT on Sub-site-B = count	Branch if none of DUTs 1-8 on Sub-site-A have an error AND any of DUTs 1-8 on Sub-site-B has an error	Branch if none of DUTs 9-16 on Sub-site-A have an error AND any of DUTs 9-16 on Sub-site-B has an error
CSUBNE_ALL (CSUBNE_ALL) CJMPNE_ALL (CJMPNE_ALL) CRETNE_ALL (CRETNE_ALL)	Branch if at least one DUT-pair doesn't have an error	Branch if both TECs for at least one DUT-pair ≠ count.	Branch if at least one of DUTs 1-8 doesn't have an error	Branch if at least one of DUTs 9-16 doesn't have an error

Table 4.13.9.9-8 Branch Operand Operation: t_errmode4 *(Continued)*

MAR Error Choice → / MAR Branch Operand (link to VAR Operand) ↓	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
	Tests error Flags 1-8 DUTs per Sub-site Logical OR of DUTs 1/9, 2/10, 3/11, etc.	Tests ECR Total Error Counters 1-8 DUTs per Sub-site Logical OR of DUTs 1/9, 2/10, 3/11, etc.	Tests error flags for DUTs 1-8 only	Tests error flags for DUTs 9-16 only
CSUBE_ALL (CSUBE_ALL) CJMPE_ALL (CJMPE_ALL) CRETE_ALL (CRETE_ALL)	Branch if at least one DUT of every DUT-pair has an error	Branch if at least one TEC for each DUT-pair = count	Branch if DUTs 1-8 each have an error	Branch if DUTs 9-16 each have an error
CSUBNE_DUT1 (CSUBNE_DUT1) ...thru... CSUBNE_DUT8 (CSUBNE_DUT8) CJMPNE_DUT1 (CJMPNE_DUT1) ...thru... CJMPNE_DUT8 (CJMPNE_DUT8) CRETNE_DUT1 (CRETNE_DUT1) ...thru... CRETNE_DUT8 (CRETNE_DUT8)	Branch if both DUTs of the DUT-pair have no errors. Operand DUT numbers show first DUT of the DUT-pair	Branch if the TEC for both DUTs of the DUT pair ≠ count. Operand DUT numbers show first DUT of the DUT-pair	Branch if the specified DUT doesn't have an error	Branch if the specified DUT doesn't have an error. Note that DUT1 actually means DUT9, DUT2 = DUT10, etc.

Table 4.13.9.9-8 Branch Operand Operation: t_errmode4 *(Continued)*

MAR Error Choice → / MAR Branch Operand (link to VAR Operand) ↓	ERRSRC1	ERRSRC2	ERRSRC3	ERRSRC4
CSUBE_DUT1 (CSUBE_DUT1) ...thru... CSUBE_DUT8 (CSUBE_DUT8) CJMPE_DUT1 (CJMPE_DUT1) ...thru... CJMPE_DUT8 (CJMPE_DUT8) CRETE_DUT1 (CRETE_DUT1) ...thru... CRETE_DUT8 (CRETE_DUT8)	Branch if either DUT of the DUT-pair has an error. Operand DUT numbers show first DUT of the DUT-pair.	Branch if the TEC for either DUT of the DUT-pair = count	Branch if the specified DUT has an error.	Branch if the specified DUT has an error. Note that DUT1 actually means DUT9, DUT2 = DUT10, etc.
Tests error Flags 1-8 DUTs per Sub-site Logical OR of DUTs 1/9, 2/10, 3/11, etc.		Tests ECR Total Error Counters 1-8 DUTs per Sub-site Logical OR of DUTs 1/9, 2/10, 3/11, etc.	Tests error flags for DUTs 1-8 only	Tests error flags for DUTs 9-16 only

4.13.9.10 Static Error Choice Functions, Branch-on-error

See MAR Error-choice Operands, MAR Multi-DUT Branch-condition Operands, VAR Multi-DUT Branch-condition Operands.

Description

The `mar_error_choice_set()` function is used to set the static error choice selection for the Branch Error Choice Logic. See the detailed description in Branch-on-error Logic..

The mar_error_choice_get() function is used to get the currently selected static error choice.

Note the following:

- These functions are only useful if executed after the Error Catch RAM (ECR) has been configured, using ecr_config_set().
- Static error choice option selection is primarily based on two criteria:
  - The number of DUTs being tested per Sub-site:
    - 1-4 DUTs (t_errmode1)
    - 5-8 DUTs (t_errmode2, t_errmode3 and t_errmode4)
  - Whether ECR Error Counters are to affect conditional branch operations, which ECR Counter Comparators are to be tested: TEC Comparator , REC Comparator , or CEC Comparator . Each static mode has different capabilities/limitations; i.e. which ECR Counter Comparators can be tested, see MAR Error-choice Operands.
- The static error mode is set to t_errmode1 during initial program load. The system software does not otherwise change the mode.
- All APGs are set to the same static mode; i.e. in Multi-DUT Test Programs, the Active DUTs Set (ADS) and Ignored DUTs Set (IDS) have no effect on mar_error_choice_set() operation.
- mar_error_choice_get() must not be used in a TDR_BLOCK(). See DUT Board TDR Functions.

Usage

```
void mar_error_choice_set( APGStaticErrorMode mode );
APGStaticErrorMode mar_error_choice_get( );
```

where:

mode specifies the desired static error mode.

mar_error_choice_get() returns the currently set static error mode.

Example

```
mar_error_choice_set( t_errmode1 );
APGStaticErrorMode m = mar_error_choice_get();
```

4.13.9.11 DUT-pin to Tester-pin Connection Requirements

Note: on 7/22/2008 this section was substantially revised, to simplify the information and improve the descriptions.

This section describes how DUT pins must be mapped to tester pins in Multi-DUT Test Programs which capture errors in the Error Catch RAM (ECR) and/or when test patterns use branch-on-error multi-DUT (branch-on-DUT) operations.

Note: this information only applies to DUT pins which will be strobed with errors captured to the ECR and/or with errors which will affect test pattern branch-on-error operations. It also applies when using over-programming facilities (see Over-programming Controls and Parallel Test and Over-programming Control Stimulus Selection). All other DUT pins can connect to any tester channels as desired.

During pattern execution, the functional-fail error signals from individual tester pins are organized per-DUT as outlined in the diagram below. This organization determines how the ECR hardware and related software groups errors per-DUT. Proper ECR operation will not occur if the user's DUT board design does not follow these DUT-pin to tester-pin rules.

These same error signals are routed to the Algorithmic Pattern Generator (APG) for use by test pattern branch-on-error multi-DUT (branch-on-DUT) operations. As above, proper per-DUT branch operations will not occur if the user's DUT board design does not follow these DUT-pin to tester-pin rules.

Important related information is covered in:

- MAR Branch Condition Operands (MAR Multi-DUT Branch-condition Operands).
- VAR Branch-condition Operands (VAR Multi-DUT Branch-condition Operands).

The following diagram shows the DUT configurations supported on Magnum 1. All errors from subsite-A are logged into the A ECR and errors from subsite-B are logged into the B

ECR:

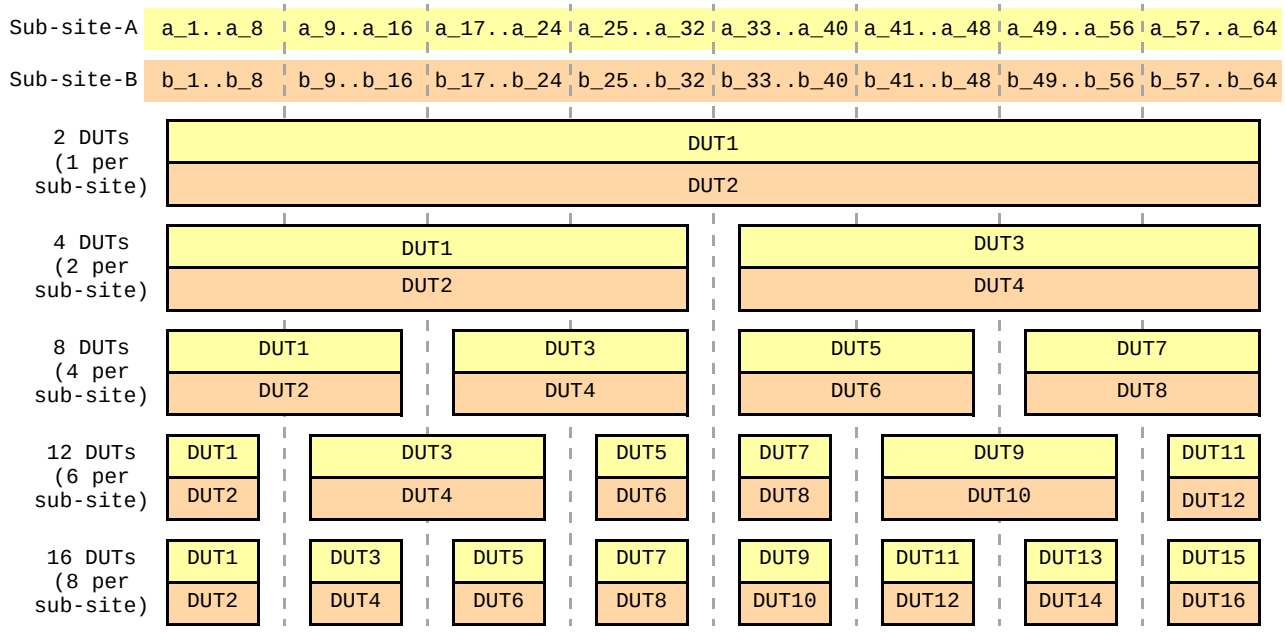


Figure-67: Magnum 1 DUT-pin to Tester-pin Connection Rules (see Note:)

4.13.10 CHIPS Instruction

See APG Chip Selects, Memory Test Patterns, Memory Pattern Instruction Format.

Description

The APG generates 8 signals, called chip selects, used when testing memory devices, typically on pins which are not address or data pins; i.e. output enable (OE), read/write (R/W), chip select (CS), etc.

The CHIPS instruction is used to control chip select state and format selection and to control two other features unrelated to the chip select outputs:

- Send the low 4 UDATA bits to the LBdata pins of the DUT board. See Loadboard Board Data Bits.
- RESET the PE error flags, and DC Error Flags. See Error Flag vs. Error Latch. The MAR RESET, VEC RESET, VAR RESET and VPINFUNC RESET instructions also can be used to reset the error flags.

The default CHIPS instruction is:

```
CHIPS   NOCLKS
```

The entire CHIPS instruction can be omitted if the default operations are appropriate in the current instruction.

The CHIPS instruction takes the form:

```
CHIPS Chip-select-control, Misc
```

where:

Chip-select-control controls the state and format selection of APG Chip Selects.

Misc controls the two other features noted above.

The table below summarizes the available operands for the CHIPS instruction. Default values are indicated using (**D**):

Table 4.13.10.0-1 CHIPS Instruction Operands

Chip Select Control	Misc
CSnT CSnF CSnPT CSnPF CSmHIZ CSmRDT CSmRDF where n = 1-8 and m = 1-2 NOCLKS(D)	LBDATA RESET (none)(D)

Each CHIPS instruction may include up to eight operands from the Chip-select-control column (to control up to 8 chip select outputs), and any combination of operands from the Misc column.

4.13.10.1 CHIPS Chip-select-control Operands

See APG Chip Selects, Memory Test Patterns, CHIPS Instruction.

Description

The APG generates 8 signals, called chip selects, used when testing memory devices, typically on pins which are not address or data pins; i.e. output enable (OE), read/write (R/W), chip select (CS), etc.

The CHIPS instruction is used to control the following parameters of the 8chip selects:

- Drive logic state (TRUE/FALSE)
- Drive format: NRZ level or pulsed (RTO or RTZ)
- I/O state (t_cs1 and t_cs2 chip selects only)
- Strobe control (t_cs1 and t_cs2 chip selects only)

Note the following:

- The first two chip selects (t_cs1 and t_cs2) are fully I/O capable and can strobe; i.e. both are independently able to drive, tri-state, and strobe (with tri-state).
- The other chip selects are drive-only.
- For all chip selects, the drive state (active/inactive) and signal format (NRZ or RTZ/RTO) are controlled by the CHIPS instruction (CSnT, CSnF, CSnPT, CSnPF, NOCLKS).
- For t_cs1 and t_cs2 only, the strobe and tri-state options are controlled using the CHIPS instruction (CSmHIZ, CSmRDT, CSmRDF, CSmRDV, CSmRDZ).

Note: unlike the other APG outputs, the drive format generated on chip select pins is not affected by the format specified when programming timing values using settime(). The programmed edge times do determine when edges occur on chip select pins, but the format (NRZ vs. RTZ vs. RTO) is solely determined by the CHIPS instruction (CSnT, CSnF, CSnPT, CSnPF, NOCLKS) vs. the active state polarity set using APG Chip Select Drive/Strobe Polarity Functions or APG Chip Select Polarity Control Function.

The CHIPS instruction takes the following form:

```
CHIPS Chip-select-control, Misc
```

Each CHIPS instruction may include up to eight operands from the Chip-select-control column (to control up to 8 chip select outputs), and any combination of operands from the Misc column.

The entire CHIPS instruction can be omitted if the default values are acceptable.

Each chip select's active-state polarity (i.e. high or low) is defined using the cs_polarity_set() function. This determines whether a pulsed chip select will generate an RTO or RTZ format, or whether DC TRUE (NRZ) is logic-1 or logic-0. The cs_polarity_get() function can be used to determine the current active-state polarity of one chip select.

It is possible for user-written C Code to get or set (modify) the operands of CHIPS instruction(s) in a specified test pattern using get_chip_select() and set_chip_select().

The table below summarizes the CHIPS Chip-select-control operands which are usable for all 8 chip select:

Table 4.13.10.1-1 CHIPS Chip-select-control Drive Operands

Operand	Purpose
CSnT	Sets chip select-*n* DC TRUE, NRZ format, at timing edge-1.
CSnF	Sets chip select-*n* DC FALSE, NRZ format, at timing edge-1.
CSnPT	Pulses chip select-*n* TRUE, generating an RTO or RTZ format depending on the active state programmed using cs_polarity_set() (or the default active low). The first edge occurs at the RTO/RTZ edge-1 time, the second at the RTO/RTZ edge-2 time.
CSnPF	Pulses chip select-*n* FALSE, generating an RTO or RTZ format depending on the inverse of the active state programmed using cs_polarity_set() (or the default active low). The first edge occurs at the RTO/RTZ edge-1 time, the second at the RTO/RTZ edge-2 time.
NOCLKS	Default. All chip selects drive DC FALSE, NRZ format, at timing edge-1. This is equivalent to: % CHIPS CS1F,CS2F,CS3F,CS4F,CS5F,CS6F,CS7F,CS8F
where **n** = 1 through 8, representing chip selects t_cs1 to t_cs8. The active-state polarity (i.e. high or low) for the chip selects is defined using the cs_polarity_set() function. This determines whether a pulsed chip select will generate an RTO or RTZ format, or whether DC TRUE (NRZ) is logic-1 or logic-0.	

The table below summarizes the CHIPS Chip-select-control operands which are usable only for the first 2 chip selects (t_cs1 and t_cs2):

Table 4.13.10.1-2 CHIPS Chip-select-control Receive Operands

Operand	Purpose
CSmHIZ	Sets chip select-*m* to tri-state. No strobe is generated.
CSmRDT	Tri-state chip select-*m*, and read (strobe) for TRUE data.
CSmRDF	Tri-state chip select-*m*, and read (strobe) for FALSE data.

Table 4.13.10.1-2 CHIPS Chip-select-control Receive Operands *(Continued)*

Operand	Purpose
CSmRDV	Tri-state chip select-*m*, and read (strobe) for valid data (<VOL or >VOH).
CSmRDZ	Tri-state chip select-*m*, and read (strobe) for tri-state (<VOH and > VOL).
where **m** = 1 or 2, representing chip selects t_cs1 or t_cs2. The active-state polarity (i.e. high or low) for the chip selects is defined using the cs_polarity_set() function. This determines whether a strobe-true, for example, is strobing for logic-1 or logic-0.	

Example

In the example below, chip select 1 (t_cs1) is tri-stated, chip select 2 (t_cs2) is strobed for TRUE data, chip select 3 (t_cs3) is set DC TRUE, chip select 4 (t_cs4) is pulsed TRUE, and chip select 5 (t_cs5) is set DC FALSE. By default, the 3 unspecified chip selects are set to DC FALSE:

```
% CHIPS  CS1HIZ, CS2RDT, CS3T, CS4PT, CS5F
```

4.13.10.2 CHIPS Misc Operands

See APG Chip Selects, Memory Test Patterns, CHIPS Instruction.

Description

The CHIPS instruction is used to control the 8 APG chip select signals, and the following two miscellaneous features:

- Send the low 4 UDATA bits to the LBdata pins of the DUT board. See Loadboard Board Data Bits.
- RESET the PE error flags, and DC Error Flags. See Error Flag vs. Error Latch. The MAR RESET, VEC RESET, VAR RESET and VPINFUNC RESET instructions also can be used to reset the error flags.

Note: as noted below, the RESET operand clears the PE error flags and the DC Error Flags in the DC Comparators and Error Logic. The PE error flags have no effect on the overall PASS/FAIL result of a functional test. However, the DC Error Flags do effect the overall PASS/FAIL result of Dynamic DC Tests. See Error Flag vs. Error Latch.

The CHIPS instruction takes the form:

```
CHIPS Chip-select-control, Misc
```

The table below summarizes the CHIPS Misc operands. Any combination of these operands may be used in the same CHIPS instruction:

Table 4.13.10.2-1 CHIPS Misc Operands

Operand	Purpose
LBDATA	Sends bits 0-3 of UDATA to the LBdata bits on the DUT board. See Loadboard Board Data Bits.
RESET	Clears all PE error flags, and DC Error Flags. See Error Flag vs. Error Latch. Using Magnum 1/2/2x, is effectively disabled using VLATCHRESET. Note: RESET (including MAR RESET, VEC RESET, VAR RESET and VPINFUNC RESET) must NOT be used in the same instruction OR the instruction following that using MAR VCOMP, VAR VCOMP, VEC VCOMP, or VPINFUNC VCOMP. Note: RESET may be used in the MAR, VEC/RPT, VAR, VPINFUNC, and CHIPS instruction.

4.13.11 DATGEN Instruction

See APG Data Generator, Memory Test Patterns, Memory Pattern Instruction Format.

The APG Data Generator is used to algorithmically generate the drive/expect (strobe) data, strobe mask, and I/O control signals used to test the data pins of memory devices. These

are pin(s) which are scrambled to APG Data Generator outputs in any given pattern instruction (see Pin Scramble MUX and Pin Scramble Functions & Macros).

The DATGEN instruction is used, in each pattern instruction, to control the following APG Data Generator parameters:

- Selection of which data generator resource is output in the current cycle.
- Selection of two inputs (A & B) to the data generator ALU. This selection is independent of the data output source selection.
- Selection of the functional operation to be performed by the APG Data Generator's ALU (shift, count, etc.).
- Selection of which data register will be updated from the ALU output. This selection is independent of the data output source selection.
- Specification of the various data inversion options.

The following data generator related parameters are not controlled using the DATGEN instruction:

- Specification of I/O state for pins scrambled to data generator outputs, as a group. This is controlled using the ADHIZ operand to the PINFUNC Instruction.
- Specification of strobe enable state for data generator outputs; i.e. strobe or don't strobe. This is controlled using the MAR Strobe Control Operands.

In each pattern cycle, the APG Data Generator outputs 36 bits, D0 through D35 (t_d0 .. t_d35), to the Pin Scramble MUX for use as drive/expect (strobe) data tester channels which are scrambled to data generator outputs. The specific data generator resource which is output is controlled using the DATGEN Dataout Operand, to select from one of the following, per-cycle:

- DMAIN data register.
- DBASE data register.
- JAM Register (a fixed 36-bit value) or the JAM RAM, a 16Kx36-bit RAM. See JAM Logic.
- Data Buffer Memory (DBM), which stores a unique data value for each X/Y address. The DBM is a hardware option.

The selected data generator output is then routed through several stages of Data Inversion Logicbefore reaching the Pin Scramble MUX. In the hardware diagrams, the inversion logic is represented by a series of exclusive OR gates. More below.

The data generator's data registers can each be configured as one 36-bit register or two 18-bit halves. This is controlled using the data_reg_width() function (default = 36-bits). All data registers use the same configuration.

Before being used, the data registers, JAM Register and/or JAM RAM/JAM RAM Address Counter and DBM must be initialized. The following methods are available to perform this initialization:

- From a pattern instruction, the UDATA value can be loaded into the DMAIN or DBASE data register or JAM Register. See DATGEN UDATAJAM and UDATADR. In these instructions the UDATA value cannot be used for other purposes.
- The dmain() and dbase() functions can be used to initialize the DMAIN and DBASE data registers, either prior to pattern execution or via Pattern Initial Conditions.
- The jamreg() function can be used to initialize the JAM Register, either prior to pattern execution or via Pattern Initial Conditions. The apg_jam_ram_set() function is used to initialize the JAM RAM. The apg_jam_ram_address_set() function is used to initialize the JAM RAM Address Counter. See JAM Logic.
- The DBM is initialized as using dbm_file_image_read(), dbm_fill(), dbm_write().

The heart of the APG Data Generator is an Arithmetic Logic Unit, or ALU. In each pattern cycle, the ALU takes one or two data source inputs (from DMAIN, DBASE or UDATA) and performs an operation, specified using the DATGEN Drfunc Operand. The result (ALU output) is placed back into DMAIN or DBASE, as controlled using the DATGEN Dest Operand. Either of these registers can be selected as the data source output to the Pin Scramble MUX as selected using the DATGEN Dataout Operand.

The Data Inversion Logic provides additional flexibility in generating APG data patterns. Data inversion options include:

- Background inversion - inverts all bits from the selected data source based on X/Y address parity plus a logical operation. Used to generate various checkerboard data patterns. Options are specified using DATGEN Background Function Operands. Also see APG Background Data Inversion Function.
- Equality functions - inverts all bits from the selected data source based on contents of the specified X/Y address register (MAIN, BASE, FIELD) plus a logical operation, specified using DATGEN Equality Function Operands. Used to generate various diagonal data patterns.
- Y-index inversion - inverts all bits from the selected data source based on the Y-address plus a logical operation, specified using DATGEN Yindex Operands. Used to modify diagonal data patterns.
- Invert sense - an explicit invert command set using DATGEN Invert Sense Operand. Affects (inverts) all bits from the selected data source.
- UDATA inversion - inverts individual data bits based on the contents of the UDATA field of the current instruction. Set using DATGEN Invert Sense Operand (XORINV).
- Data Topological Inversion (DTOPO) - inverts all bits from the selected data source based on the contents of the DTOPO RAM, which is addressed by the APG's X/Y address outputs. Configured using the APG Data Topological Inversion (DTOPO) Function and APG Data TOPO RAM Load Functions. Enabled in the test pattern using the DTOPO operand, see DATGEN Background Function Operands.

Often, multiple data inversion options are enabled simultaneously. All inversion options except XORINV (UDATA per-bit inversion) operate on all data generator outputs as a group; i.e. only XORINV can selectively invert individual data generator output bits.

When the Data Buffer Memory (DBM) option is installed/used, the DBMWR operand (see DATGEN Dbmwr Operand) can be used to capture the output of the APG Data Generator into the DBM.

The default DATGEN instruction is:

```
DATGEN  SDMAIN, SDMAIN, DMAIN, HOLDDR, HOLDYN, EQFDIS,
        BCKFDIS, NOTINV, DATDAT
```

The DATGEN instruction takes the form:

```
DATGEN SrcA, SrcB, Dest, Drfunc, Yindex, Eqfunc, Bckfunc, Invsns, Dataout, Udatajam, Dbmwr
```

Each DATGEN instruction may include one operand in each field but it is not necessary to include an operand in every field. The order operands are included in an instruction is not important. The entire DATGEN instruction may be omitted if default values are acceptable.

The table below summarizes the operands for each field of the DATGEN instruction. For Magnum 1/2/2x 2 tables are used; additional , and Dest operands, which only apply to Magnum 1/2/2x, are listed separately, in the 2nd table. Default values are indicated using (**D**):

Table 4.13.11.0-1 DATGEN Instruction Operands

Drfunc	Yindex	Eqfunc	Bckfunc	Invsns	Dataout	Udatajam
ADD	CNTDNYN	EQFDIS(D)	BCKDTOPO	INVSNS	BUFBUF	[none](D)
AND	CNTUPYN	XEQB	BCKFEN	NOTINV(D)	BUFDAT	UDATAJAM
CNTDNDR	HOLDYN(D)	XLEB	BCKFDIS(D)	XORINV	BUFJAM	
CNTUPDR	UDATAYN	XLTB	DTOPO		DATBUF	**Dbmwr**
CMPLDR		XYLEBYF			DATDAT(D)	DBMWR
HOLDDR		XYLTBXF			DATJAM	[none](D)
OR		XYLTBYF			JAMBUF	
ROTLDR		XYLEBXF			JAMDAT	
ROTRDR		XEQBORF			JAMJAM	
SHLDR		XEQYPN			BASEBASE	
SHRDR		XEQYBPN			BASEBUF	
SUBTRACT		XYEQB			BASEDAT	
UDATADR		YEQB			BASEMAIN	
XOR		YLEB			BASEJAM	
		YEQBORF			BUFBASE	
		YLTB			BUFMAIN	
					DATBASE	
					MAINBASE	
					MAINBUF	
					MAINMAIN	
					MAINJAM	
					JAMBASE	
					JAMMAIN	
					JAMRAMINCR	
					JAMRAMDECR	
					JAMRAMHOLD	

Table 4.13.11.0-2 DATGEN Magnum 1/2/2x Only Instruction Operands

SrcA	SrcB	Dest	Dataout
SDMAIN(D) SDBASE SUDATA	SDMAIN(D) SDBASE UDATA	DMAIN(D) DBASE	BASEBASE BASEBUF BASEDAT BASEMAIN BASEJAM BUFBASE BUFMAIN DATBASE MAINBASE MAINBUF MAINMAIN MAINJAM JAMBASE JAMMAIN JAMRAMINCR JAMRAMDECR JAMRAMHOLD
SrcA and SrcB refer to the inputs to the APG Data Generator's ALU. Dest refers to the destination of the output of the APG Data Generator's ALU. Dataout refers to the APG Data Generator output selected for the current instruction.			

4.13.11.1 DATGEN Source Operands

See APG Data Generator, Memory Test Patterns, DATGEN Instruction.

Description

The heart of the APG Data Generator is an Arithmetic Logic Unit, or ALU. In each pattern cycle, the ALU takes one or two data source inputs (from DMAIN, DBASE or UDATA) and performs an operation, specified using the DATGEN Drfunc Operand.

The DATGEN instruction takes the following form:

```
DATGEN SrcA, SrcBSrcB, Dest, Drfunc, Yindex, Eqfunc, Bckfunc,
Invsns, Dataout, Udatajam, Dbmwr
```

The two inputs to the ALU are selected using the DATGEN SrcA and SrcB operands.

Rules:

- When the Drfunc field contains a single-source instruction; i.e. CNTDNDR, CNTUPDR, CMPLDR, HOLDDR, ROTLDR, ROTRDR, SHLDR, or SHRDR, the SrcB field is ignored.
- When the Drfunc field contains a two-source instruction; i.e. ADD, AND, OR, SUBTRACT, or XOR, the SrcB field is used. If a SrcB operand is not specified the value defaults to SDMAIN.
- When the Drfunc field contains UDATADR the SrcA field must be SUDATA and SrcB cannot be specified.
- The selected DATGEN SrcA and SrcB operands may modify the Data Register Fill-bit value depending on the Dest and Drfunc operands. See Data Register Fill-bit.

The table below describes the options available for the SrcA and SrcB operands to DATGEN:

Table 4.13.11.1-1 DATGEN SrcA and SrcB Operands

Operand	Purpose
SDMAIN	The DMAIN data register is selected. Default.
SDBASE	The DBASE data register is selected.
SUDATA	The UDATA register is selected.

4.13.11.2 DATGEN Dest Operand

See APG Data Generator, Memory Test Patterns, DATGEN Instruction.

Description

A multiplexer (MUX) on the output of the APG Data Generator's ALU determines, in a given pattern instruction, which data register will be updated with the ALU output.

Rules:

- Only one register may be specified.

- If no Dest value is specified the DMAIN register is written.

The DATGEN instruction takes the following form:

```
DATGEN SrcA, SrcB, Dest, Drfunc, Yindex, Eqfunc, Bckfunc, Invsns,
Dataout, Udatajam, Dbmwr
```

The table below describes the options available for the Dest operand to DATGEN:

Table 4.13.11.2-1 DATGEN Dest Operands

Operand	Purpose
DMAIN	Write the ALU output to the DMAIN register only (default). May also affect the fill bit used during SHLDR./SHRDR. See Data Register Fill-bit.
DBASE	Write the ALU output to the DBASE register only. May also affect fill bit used during SHLDR./SHRDR. See Data Register Fill-bit.

4.13.11.3 DATGEN Drfunc Operand

See APG Data Generator, Memory Test Patterns, DATGEN Instruction.

Description

The Drfunc operand determines the functional operation performed by the APG Data Generator's ALU.

The heart of the APG Data Generator is an Arithmetic Logic Unit, or ALU. In each pattern cycle, the ALU performs the functional operation specified using the Drfunc operand. operating on one or two data source inputs (specified using DATGEN Source Operands). The function result (i.e. ALU output) is placed into the data register specified using the DATGEN Dest Operand.

The DATGEN instruction takes the following form:

```
DATGEN SrcA, SrcB, Dest, Drfunc, Yindex, Eqfunc, Bckfunc, Invsns,
Dataout, Udatajam, Dbmwr
```

Rules:

- When the Drfunc field contains a single-source instruction (i.e. CNTDNDR, CNTUPDR, CMPLDR, HOLDDR, ROTLDR, ROTRDR, SHLDR, or SHRDR), the SrcB operand is ignored.

- When the Drfunc field contains a two-source instruction (i.e. ADD, AND, OR, OR, SUBTRACT, or XOR) both the SrcA and SrcB operands are used. If a SrcB operand is not specified the value defaults to SDMAIN.
- When the Drfunc operand is UDATADR the SrcA field must be SUDATA and SrcB cannot be specified.

The table below describes the options available for the Drfunc operand to DATGEN:

Table 4.13.11.3-1 DATGEN Drfunc Operands

Operand	Purpose
ADD	ADD SrcA + SrcB.
AND	Logically AND SrcA with SrcB.
CNTDNDR	CouNT DowN Data Register Decrements the SrcA input by one. Requires a cycle period of at least 30nS (3 10nS system clocks).
CNTUPDR	CouNT UP Data Register Increments the SrcA input by one. Requires a cycle period of at least 30nS (3 10nS system clocks).
CMPLDR	CoMPLement Data Register Complements the SrcA input, including the Data Register Fill-bit.
HOLDDR	HOLD Data Register No operation (Hold), i.e. the SrcA input is written to the Dest output without modification (default) .
OR	Logically OR SrcA with SrcB.
ROTLDR	ROTate Left Data Register Rotates the SrcA input left. The MSB will rotate into the LSB position. If the SrcA input is configured as two 18-bit halves (see APG Data Register Width Selection Function), this operation is performed on each half independently.
ROTRDR	ROTate Right Data Register Rotates the SrcA input right. The LSB will rotate into the MSB position. If the SrcA input is configured as two 18-bit halves (see APG Data Register Width Selection Function), this operation is performed on each half independently.

Table 4.13.11.3-1 DATGEN Drfunc Operands *(Continued)*

Operand	Purpose
SHLDR	SHift Left Data Register Shifts the SrcA input left. The LSB is filled from the Data Register Fill-bit. If the SrcA input is configured as two 18-bit halves (see APG Data Register Width Selection Function), this operation is performed on each half independently.
SHRDR	SHift Right Data Register Shifts the SrcA input right. The MSB is filled from the Data Register Fill-bit. If the SrcA input is configured as two 18-bit halves (see APG Data Register Width Selection Function), this operation is performed on each half independently.
SUBTRACT	SUBTRACT SrcB from SrcA.
UDATADR	UDATA Data Register Loads UDATA bits 0-35 into the specified Dest and bit-36 into the Data Register Fill-bit associated with Dest. Data is not modified. SrcA must be SUDATA and SrcB cannot be specified.
XOR	Logically XOR SrcA with SrcB.

Data Register Fill-bit

The data register fill-bit supplys the *new* bit required by data register shift instructions SHLDR and SHRDR. This is the value that fills the empty LSB bit in SHLDR instructions and the new MSB bit in SHRDR instructions. Each data register has a separate fill-bit.

The data register fill-bit can be set or modified several ways:

- In a DATGEN instruction which uses UDATADR, bit-36 of the UDATA value sets the fill-bit for the data register identified by the Dest operand.
- In a DATGEN instruction which specifies a SrcA and Dest register, the fill-bit of the Dest register is set to the fill-bit of the SrcA register. If the Drfunc is CMPLDR the fill-bit is also inverted before being written to the Dest register.

4.13.11.4 DATGEN Yindex Operands

See APG Data Generator, Memory Test Patterns, DATGEN Instruction.

Description

The DATGEN Yindex operands are used to control Yindex register component of the Data Inversion Logic.

The Yindex register is used to conditionally invert selected outputs of the APG Data Generator. It is used with the data equality functions XEQYPN and XEQYBPN (see DATGEN Equality Function Operands), to generate diagonal data patterns.

The APG Data Inversion Enable Functions and APG Data Inversion Bank Select Functions do affect the use and operation of DATGEN Yindex.

These operands operate as follows:

- XEQYPN: invert if the X-address = (Y-address + Yindex). The Yindex register is added to the Y-address, then compared to the
  X-address. If they are equal, the output of the data generator is inverted.
- XEQYBPN: invert if the X-address = ($\overline{\text{Y-address}}$ + Yindex). The Yindex register is added to the complement of the Y-address, then compared to the X address. If they are equal, the output of the data generator is inverted.

In general, the Yindex register value is used to shift the position of a diagonal pattern, either right or left, or to move a diagonal the number of columns specified by UDATA bits 0-15. XEQYPN is used to generate a left-to-right diagonal, XEQYBPN is used to generate a right-to-left diagonal. Both options generate *barber-pole* diagonals; i.e. the diagonal wraps around the memory array.

The DATGEN yindex instruction takes the following form:

```
DATGEN SrcA, SrcB, Dest, Drfunc, Yindex, Eqfunc, Bckfunc, Invsns, Dataout, Udatajam, Dbmwr
```

The table below describes the options available for the Yindex operand to DATGEN:

Table 4.13.11.4-1 DATGEN Yindex Operands

Operand	Purpose
CNTDNYN	CouNT DowN YiNdex Decrements the Yindex register.
CNTUPYN	CouNT UP YiNdex Increments the Yindex register.
HOLDYN	HOLD YiNdex Holds the Yindex register at its present value (default)
UDATAYN	UDATA YiNdex Loads the Yindex register from UDATA bits 0-15.

4.13.11.5 DATGEN Equality Function Operands

See APG Data Generator, Memory Test Patterns, DATGEN Instruction.

Description

The DATGEN Eqfunc operands control hardware used to conditionally invert selected outputs of the APG Data Generator based on a comparison of X/Y address registers or X/Y address output + Y-index register. Together, these are used to generate various inverted bit, inverted row(s), inverted column(s) or diagonal data patterns using the Data Inversion Logic.

The APG Data Inversion Enable Functions and APG Data Inversion Bank Select Functions do affect the use and operation of DATGEN Eqfunc.

The DATGEN instruction takes the following form:

```
DATGEN SrcA, SrcB, Dest, Drfunc, Yindex, Eqfunc, Bckfunc, Invsns, Dataout, Udatajam, Dbmwr
```

The DATGEN Eqfunc operand is used to:

- Select which X/Y address registers, or combination of registers, will be compared.

- Specify the boolean comparison operation to be performed.

Rules:

- Comparisons can be made based on the entire X or Y address or based on a specified address register. The specific selection is encoded in the operand name, as described in the table below.
- When AMAIN, ABASE, or AFIELD is specified, the 'A' indicates a comparison based on the combined value of the specified registers of both the X and Y address generators. For example, AMAIN represents both X-MAIN and Y-MAIN registers. The XYEQB, YEQB, XYLTBYF, XYLEBYF, XYLTBXF and XYLEBXF operands order X/Y addresses as specified using `x_fast_axis()`. Similarly, ABASE consists of both X-BASE and Y-BASE registers, ordered as specified using `x_fast_axis()`.
- When a specific address register is specified only that register is used for comparison. For example, when YMAIN is specified, only the MAIN register from the Y address generator is involved in the comparison.

The table below describes the options available for the `Yindex` operand to DATGEN:

Table 4.13.11.5-1 DATGEN Eqfunc Operands

Operand	Purpose
EQFDIS	EQuality Function DISable Disable equality function inversion (default)
XYEQB	Invert on AMAIN = ABASE Inverts the data generator output when AMAIN (the address specified by the combined X-MAIN and Y-MAIN registers) equals ABASE (the address specified by the combined X-BASE and Y-BASE registers). In use, this causes inverted data at a single address.
YEQB	Invert on YMAIN = YBASE Inverts the data generator output when the value in the Y-MAIN register equals the value in the Y-BASE register. In use, this inverts one column of data, generating a vertical stripe.
XEQB	Invert on XMAIN = XBASE Inverts the data generator output when the value in the X-MAIN register equals the value in the X-BASE register. In use, this inverts one row of data, generating a horizontal stripe.

Table 4.13.11.5-1 DATGEN Eqfunc Operands *(Continued)*

Operand	Purpose
YEQBORF	Invert on YMAIN = YBASE or YFIELD Inverts the data generator output when the value in the Y-MAIN register equals the value in either the Y-BASE register or the Y-FIELD register. In use, this generates one vertical stripe (when Y-BASE = Y-FIELD) or two vertical stripes (when the Y-BASE does not equal Y-FIELD).
XEQBORF	Invert on XMAIN = XBASE or XFIELD Inverts the data generator output when the value in the X-MAIN register equals the value in either the X-BASE register or the X-FIELD register. In use, this generates one horizontal stripe (when X-BASE = X-FIELD) or two horizontal stripes (when the X-BASE does not equal X-FIELD).
YLTB	Invert on YMAIN < YBASE Inverts the data generator output when the value in the Y-MAIN register is less than the value in the Y-BASE register. In use, this generates multiple columns of inverted data, up to but excluding the value in the Y-BASE register.
XLTB	Invert on XMAIN < XBASE Inverts the data generator output when the value in the X-MAIN register is less than the value in the X-BASE register. In use, this generates multiple rows of inverted data, up to but excluding the value in the X-BASE register.
YLEB	Invert on YMAIN ≤ YBASE Inverts the data generator output when the value in the Y-MAIN register is less than or equal to the value in the Y-BASE register. In use, this generates multiple columns of inverted data, up to and including the value in the Y-BASE register.
XLEB	Invert on XMAIN ≤ XBASE Inverts the data generator output when the X-MAIN register is less than or equal to the X-BASE register. In use, this generates multiple rows of inverted data, up to and including the value in the X-BASE register.

Table 4.13.11.5-1 DATGEN Eqfunc Operands *(Continued)*

Operand	Purpose
XYLTBYF	Invert on AMAIN < ABASE, Y Fast Inverts the data generator output when AMAIN is less than ABASE, with the Y-axis selected as the fast axis. AMAIN is the concatenation of the X-MAIN and Y-MAIN registers. ABASE is the concatenation of the X-BASE and Y-BASE registers. Specifying Y Fast causes the concatenation to put the Y-MAIN and Y-BASE registers at the low end; i.e. the fast axis is the address axis (X or Y) that is changing most rapidly when the address is incremented. In use, this generates inverted data for all addresses less than that specified by the ABASE value.
XYLEBYF	Invert on AMAIN ≤ ABASE, Y Fast Inverts the data generator output when AMAIN is less than or equal to ABASE, with the Y-axis selected as the fast axis. AMAIN is the concatenation of the X-MAIN and Y-MAIN registers. ABASE is the concatenation of the X-BASE and Y-BASE registers. Specifying Y Fast causes the concatenation to put the Y-MAIN and Y-BASE registers at the low end; i.e. the fast axis is the address axis (X or Y) that is changing most rapidly when the address is incremented. In use, this generates inverted data for all addresses less than or equal to that specified by the ABASE value.
XYLTBXF	Invert on AMAIN < ABASE, X Fast Inverts the data generator output when AMAIN is less than ABASE, with the X-axis selected as the fast axis. AMAIN is the concatenation of the X-MAIN and Y-MAIN registers. ABASE is the concatenation of the X-BASE and Y-BASE registers. Specifying X Fast causes the concatenation to put the X-MAIN and X-BASE registers at the low end; i.e. the fast axis is the address axis (X or Y) that is changing most rapidly when the address is incremented. In use, this generates inverted data for all addresses less than that specified by the ABASE value.

Table 4.13.11.5-1 DATGEN Eqfunc Operands *(Continued)*

Operand	Purpose
XYLEBXF	Invert on AMAIN ≤ ABASE, X Fast Inverts the APG Data Generator's output when AMAIN is less than or equal to ABASE, with the X-axis selected as the fast axis. AMAIN is the concatenation of the X-MAIN and Y-MAIN registers. ABASE is the concatenation of the X-BASE and Y-BASE registers. Specifying X Fast causes the concatenation to put the X-MAIN and X-BASE registers at the low end; i.e. the fast axis is the address axis (X or Y) that is changing most rapidly when the address is incremented. In use, this generates inverted data for all addresses less than or equal to that specified by the ABASE value.
XEQYPN	Invert on XOUT = (YOUT + Yindex) Inverts the data generator output when XOUT equals (YOUT plus Yindex). XOUT is the output of the X address generator, as selected by the XALU Destination operand and YOUT is the output of the Y address generator, as selected by the YALU Destination operand (see YALU/XALU Destination Operands). Yindex is the APG Data Generator's Yindex register. In use, this generates a diagonal data pattern with a slope opposite that obtained using XEQYBPN, with the Yindex register used to offset the diagonal in the Y-axis. Also see DATGEN Yindex Operands.
XEQYBPN	Invert on XOUT = ($\overline{\text{YOUT}}$ + Yindex) Inverts the data generator output when XOUT equals (YOUT plus Yindex). XOUT is the output of the X address generator, as selected by the XALU Destination operand and YOUT is the complement of the Y address generator output, as selected by the YALU Destination operand (see YALU/XALU Destination Operands). Yindex is the APG Data Generator's Yindex register. In use, this generates a diagonal data pattern with a slope opposite that obtained using XEQYPN, with the Yindex register used to offset the diagonal in the Y-axis. Also see DATGEN Yindex Operands.

4.13.11.6 DATGEN Background Function Operands

See APG Data Generator, Memory Test Patterns, DATGEN Instruction.

Description

The DATGEN Bckfunc operands are used to:

- Enable or disable the background inversion logic, used to conditionally invert the output of the APG Data Generator as a function of X and/or Y address. See APG Background Data Inversion Function and Data Inversion Logic.
- Enable or disable the data topological (DTOPO) inversion logic, used to conditionally invert the output of the APG Data Generator based on the contents of two DTOPO RAMs. See Data Inversion Logic.

When enabled, the background inversion logic performs a parity evaluation of the APG's X and/or Y address output. This generates a single bit, which determines whether to invert or not-invert the output of the APG Data Generator. Before the pattern executes, the desired parity operation, and optionally which X/Y address bits to consider, must be specified using the bckfen() function. A typical application of background inversion is to generate various checkerboard data patterns.

The APG Data Generator also contains logic to support data topological (DTOPO) inversion as a function of X/Y address. Using DTOPO inversion, a single invert bit is used to either invert or not-invert the output of the APG Data Generator based on the contents of the DTOPO RAM. During pattern execution, the DTOPO RAM is addressed by the APG's X and Y address generator outputs. To use DTOPO inversion requires the following:

- Initializing the DTOPO RAM using APG Data TOPO RAM Load Functions.
- Specifying which DTOPO RAM output(s) are to be considered and what, if any, logical operation is to be applied. See APG Data Topological Inversion (DTOPO) Function.
- Enable DTOPO inversion using the DTOPO operand in the appropriate test pattern instructions.
- The APG Data Inversion Enable Functions and APG Data Inversion Bank Select Functions do affect the use and operation of background inversion and data topological (DTOPO) inversion.

The DATGEN instruction takes the following form:

```
DATGEN SrcA, SrcB, Dest, Drfunc, Yindex, Eqfunc, Bckfunc, Invsns,
Dataout, Udatajam, Dbmwr
```

The table below describes the options available for the Bckfunc operand to DATGEN:

Table 4.13.11.6-1 DATGEN Bckfen Operands

Operand	Purpose
BCKDTOPO	BaCKground and DTOPO enable Enable both background inversion and DTOPO inversion.
BCKFDIS	BaCKground Function DISable Disable both background inversion and DTOPO inversion (default).
BCKFEN	BaCKground Function ENable Enable the background inversion but not the DTOPO inversion.
DTOPO	DTOPO Enable the DTOPO inversion but not the background inversion.

4.13.11.7 DATGEN Invert Sense Operand

See APG Data Generator, Memory Test Patterns, DATGEN Instruction.

Description

The DATGEN Invsns operands are used to:

- Explicitly invert the output of the APG Data Generator.
- Invert individual APG Data Generator outputs using the UDATA value as an inversion bit-mask.

The APG Data Inversion Enable Functions and APG Data Inversion Bank Select Functions do affect the use and operation of DATGEN Invsns.

The DATGEN instruction takes the following form:

```
DATGEN SrcA, SrcB, Dest, Drfunc, Yindex, Eqfunc, Bckfunc, Invsns,
Dataout, Udatajam, Dbmwr
```

The table below describes the options available for the Invsns operand to DATGEN:

Table 4.13.11.7-1 DATGEN Invsns Operands

Operand	Purpose
INVSNS	INVert SeNSe Unconditionally inverts the output of the APG Data Generator.
NOTINV	NOT INVert Does not invert the output of the APG Data Generator (default)
XORINV	eXclusive OR INVert XORs the output of the APG Data Generator with UDATA bits 0 through 35. Data output D0 is XOR'ed with UDATA bit 0, D1 is XOR'ed with UDATA bit 1, etc. Any UDATA bit which is logic-1 will invert the corresponding output of the APG Data Generator.

4.13.11.8 DATGEN Dataout Operand

See APG Data Generator, Memory Test Patterns, DATGEN Instruction.

Description

The DATGEN Dataout operands are used to select which APG Data Generator resource is output, in the current instruction, to the DUT via the Pin Scramble MUX. Note the following:

- In each tester cycle, the APG Data Generator outputs 36 bits, D0 (t_d0) through D35 (t_d35), to the Pin Scramble MUX, for use as drive/expect (strobe) data, typically on tester channels connected to DUT data bus pins.
- The data generator has several sources which can be selected for output:
  - DMAIN data register.
  - DBASE data register.
  - JAM Register, a single 36-bit value, or the JAM RAM, see JAM Logic.
  - Data Buffer Memory (DBM) - stores a unique data value for each X/Y address. The DBM is a hardware option.
- In hardware, the data generator output is selected by a multiplexer (MUX), which is controlled by the DATGEN Dataout operand in each pattern instruction. The MUX is located before the data inversion logic thus data inversion(s) operate

consistently regardless of which data source is selected. See DATGEN Background Function Operands, DATGEN Invert Sense Operand, DATGEN Equality Function Operands, DATGEN Yindex Operands.

- By design, data source selection is split into two 18-bit halves. Thus it is possible to select, for example, the DMAIN data register to output bits 0-17 and the JAM Register to output bits 18-36. Etc.
- When a test pattern uses the Data Buffer Memory (DBM), if the pattern is paused (MAR PAUSE), the first 20 tester cycles after restarting pattern execution (using `restart()` or `restart_and_wait()`) must not select the DBM as a data source. This rule does not apply when first executing the test pattern using `funtest()` or `start_pattern()`.

The DATGEN instruction takes the following form:

```
DATGEN SrcA, SrcB, Dest, Drfunc, Yindex, Eqfunc, Bckfunc, Invsns,
Dataout, Udatajam, Dbmwr
```

The table below describes the options available for the `Dataout` operand to DATGEN:

Table 4.13.11.8-1 DATGEN Dataout Operands

Src/Src	
BASEBASE	Selects DBASE 35:0.
BASEBUF	Selects DBASE 35:18 and DBM 17:0.
BASEDAT **BASEMAIN**	Selects DBASE 35:18 and DMAIN 17:0.
BASEJAM	When the JAM Register is selected (see `apg_jam_mode_set()`), selects DBASE 35:18 and JAM Register 17:0 and the JAM RAM Address Counter is not modified. When the JAM RAM is selected, selects JAM RAM 35:0 and the JAM RAM Address Counter is not modified.
BUFBASE	Selects DBM 35:18 and DBASE 17:0.
BUFBUF	Selects DBM 35:0.
BUFDAT **BUFMAIN**	Selects DBM 35:18 and DMAIN 17:0.
1) Specific rules apply when using both DATGEN BUF and/or DATGEN DBMWR. See DBM Usage Rules. **2) MAIN and DAT are the same hardware source.**	

Table 4.13.11.8-1 DATGEN Dataout Operands *(Continued)*

Src/Src	
BUFJAM	When the JAM Register is selected (see apg_jam_mode_set()), selects DBM 35:18 and JAM Register 17:0 and the JAM RAM Address Counter is not modified. When the JAM RAM is selected, selects JAM RAM 35:0 and the JAM RAM Address Counter is not modified.
DATBASE MAINBASE	Selects DMAIN 35:18 and DBASE 17:0
DATBUF MAINBUF	Selects DMAIN 35:18 and DBM 17:0.
DATDAT MAINMAIN	Selects DMAIN 35:0. Default data source selection. Note that DATMAIN and MAINDAT are not supported.
DATJAM MAINJAM	When the JAM Register is selected (see apg_jam_mode_set()), selects DMAIN 35:18 and JAM Register 17:0 and the JAM RAM Address Counter is not modified. When the JAM RAM is selected, selects JAM RAM 35:0 and increments the JAM RAM address. Same as JAMRAMINCR when JAM RAM is selected.
JAMBASE	When the JAM Register is selected (see apg_jam_mode_set()), selects JAM Register 35:18 and DBASE 17:0 and the JAM RAM Address Counter is not modified. When the JAM RAM is selected, selects JAM RAM 35:0 but the JAM RAM Address Counter is not modified.
JAMBUF	When the JAM Register is selected (see apg_jam_mode_set()), selects JAM Register 35:18 and DBM 17:0 and the JAM RAM Address Counter is not modified. When the JAM RAM is selected, selects JAM RAM 35:0 but the JAM RAM Address Counter is not modified.
JAMDAT JAMMAIN	When the JAM Register is selected (see apg_jam_mode_set()), selects JAM Register 35:18 and DMAIN register 17:0 and the JAM RAM Address Counter is not modified. When the JAM RAM is selected, selects JAM RAM 35:0 and decrements the JAM RAM Address Counter. Same as JAMRAMDECR when JAM RAM is selected.

1) Specific rules apply when using both DATGEN BUF and/or DATGEN DBMWR. See DBM Usage Rules.
2) MAIN and DAT are the same hardware source.

Table 4.13.11.8-1 DATGEN Dataout Operands *(Continued)*

Src/Src	
JAMJAM	When the JAM Register is selected (see apg_jam_mode_set()) selects JAM Register 35:0 and the JAM RAM Address Counter is not modified. When the JAM RAM is selected, selects JAM RAM 35:0 and holds the JAM RAM Address Counter. Same as JAMRAMHOLD when JAM RAM.
JAMRAMINCR	Same as DATJAM.
JAMRAMDECR	Same as JAMDAT.
JAMRAMHOLD	Same as JAMJAM.
1) Specific rules apply when using both DATGEN BUF and/or DATGEN DBMWR. See DBM Usage Rules. 2) MAIN and DAT are the same hardware source.	

4.13.11.9 DATGEN Udatajam Operands

See APG Data Generator, Memory Test Patterns, DATGEN Instruction.

Description

The DATGEN Udatajam operands are used to load the JAM Register from UDATA bits 0-35.

The DATGEN instruction takes the following form:

```
DATGEN SrcA, SrcB, Dest, Drfunc, Yindex, Eqfunc, Bckfunc, Invsns,
Dataout, Udatajam, Dbmwr
```

The table below describes the options available for the Udatajam operand to DATGEN:

Table 4.13.11.9-1 DATGEN Udatajam Operands

Operand	Purpose
UDATAJAM	Load the JAM Register from UDATA bits 0-35.
[**none**]	Don't load the JAM Register (default).

4.13.11.10 DATGEN Dbmwr Operand

See APG Data Generator, Memory Test Patterns, DATGEN Instruction.

Description

During pattern execution, the output of the APG Data Generator, including all inversions, can be written to the Data Buffer Memory (DBM) option (if installed). This is enabled, per cycle, using `DATGEN DBMWR`.

Any pattern instruction which contains the `DBMWR` operand will write the data generator output, including all data inversions, to the DBM, at the address being output by the X/Y APG Address Generator in the same cycle. Note that the actual DBM address written will be affected when DBM Sequential Mode is used.

This feature is typically used to accumulate multiple data patterns, as they are written into a non-volatile memory (NVM). Then, at any given time, the DBM represents the cumulative data patterns stored in the DUT, allowing the DBM to be used as the data source when reading the DUT. For example, several data patterns, including row and column stripes, diagonal, and a checkerboard, are separately written to an NVM DUT. As each pattern is written to the DUT it is also capture (accumulated) in the DBM using the `DBMWR` operand. To verify that the DUT has correctly stored each test pattern requires reading the accumulation of all patterns written. It is often not practical to algorithmically generate these pattern accumulations, whereas storing the accumulation is practical using the DBM.

Note: specific rules apply when using both `DATGEN DBMWR` and/or `DATGEN BUF`. See DBM Usage Rules.

The `DATGEN` instruction takes the following form:

```
DATGEN SrcA, SrcB, Dest, Drfunc, Yindex, Eqfunc, Bckfunc, Invsns,
Dataout, Udatajam, Dbmwr
```

The table below describes the options available for the Dbmwr operand to DATGEN:

Table 4.13.11.10-1 DATGEN Dbmwr Operands

Operand	Purpose
DBMWR	Write the APG Data Generator output, including all inversions, into Data Buffer Memory (DBM) at the X/Y address output in the same cycle. The DBM address actually written will be affected when DBM Sequential Mode is used.
[**none**]	Do not write to the DBM (default).

4.13.12 UDATA Instruction

See Algorithmic Pattern Generator (APG), Memory Test Patterns, Memory Pattern Instruction Format.

The UDATA (microdata) pattern instruction is used to explicitly specify a UDATA value. The UDATA value represents 37 bits stored in each APG instruction which can be used for various purposes as noted in the UDATA Bit Applications table below.

The UDATA value is used in 3 contexts, which occur in the test pattern on a per-instruction basis:

1. *Explicit* user-defined UDATA value and application. The desired value is specified using the UDATA instruction in the pattern instruction. The application is specified using one of the optional operands to the YALU, XALU, COUNT, MAR, CHIPS, USERRAM and DATGEN instruction. The UDATA Bit Applications table (below) lists these operand options, and which UDATA bits are used in each application.
2. *Implicit* pattern compiler-defined UDATA value and application. The user's pattern instruction must ***not*** specify a UDATA value in the following situations:
   - The first vector of Logic Test Patterns. The UDATA value is implicitly used to set the Vector Address Register (VAR); i.e. the address of the first vector of the pattern.
   - A Logic pattern instruction with the RPT instruction. The UDATA value is implicitly used to set an APG counter to the specified repeat value (-1).

- In a Logic pattern, any vector after a STARTLOOP instruction. The UDATA value is implicitly used to record the VAR as the starting vector of the loop.
- In a Logic pattern, any vector before an ENDLOOP instruction. The UDATA value is implicitly used to set an APG counter to the specified loop count value (-2). This is done during the 1st loop iteration only, subsequent loop iterations decrement the counter and if not zero, jump back to the beginning of the loop.
- In a Logic pattern, any vector with a label. The UDATA value is implicitly used to record the VAR of that vector.
- In any instruction which includes LSENABLE.
- In any USERRAM instruction.
- In mixedsync patterns, any logic instruction with a label.
- In mixedsync patterns, any instruction with a MAR or VAR branch instruction.

3. A mix of explicit and implicit values. This is unique to Controlling PE Levels from the Test Pattern. The UDATA value is used to encode the voltage/current value specified by the user in the UDATA value, with information added by the pattern compiler to identify the hardware being programmed, whether a value is being *set* or *tweaked*, and which voltage or current DAC is being programmed (DPS voltage, VIL, etc.). This applies to the LEVELSET pattern instruction.

Note: user code must NOT use `set_udata()` to modify the UDATA value of these instructions because the implicit information added by the pattern compiler will be corrupted.

An explicit UDATA instruction takes one of the following the forms:

```
UDATA   n
UDATA   value, units, range#
```

The first form sets the UDATA value to **n**, where *n* is in the range 0 to 0x1FFFFFFFF0 hex (37-bits). The table below defines how these bits are used. The second form is unique to Controlling PE Levels from the Test Pattern, and is documented in that section.

The following table documents which UDATA bit positions are used by the pattern instructions noted:

Table 4.13.12.0-1 UDATA Bit Applications

Instruction	UDATA Application (target operand)	Bit Positions Used
YALU	YUDATA	15-0
XALU	XUDATA	33-16
COUNT	RELOAD#	31-0
COUNT	COUNTUDATA	31-0
MAR	READUDATA	35-0
MAR	INTADR	15-0
MAR	INTENADR	15-0
CHIPS	LBDATA	3-0
DATGEN	UDATADR	35-0
DATGEN	Data Register Fill-bit	36
DATGEN	UDATAYN	15-0
DATGEN	XORINV	35-0
DATGEN	UDATAJAM	35-0
USERRAM	LOAD	35-0

As indicated, the YALU, XALU, COUNT, MAR, CHIPS, USERRAM and DATGEN instructions may optionally use the UDATA value, depending on other operands used in the instruction. With the exception of YUDATA and XUDATA the UDATA value will only be applied to one target, based on the operand specified. Thus, it is not possible, for example, to use YUDATA and COUNTUDATA in the same instruction.

Example

The following examples presume 16-bits for both X and Y addresses:

```
% YALU    YUDATA, XCARE, COFF, HOLD, DYMAIN
  XALU    XUDATA, XCARE, COFF, HOLD, DXMAIN
  MAR     INC
  UDATA   0x89ABCDEF
```

In this example, the UDATA value's bits are loaded as follows:

- The hexadecimal value 0xCDEF, corresponding to UDATA bits 15-0, is loaded into the Y-MAIN address register.
- The hexadecimal number 0x89AB, corresponding to UDATA bits 31-16, is loaded into the X-MAIN address register.

4.13.13 PINFUNC Instruction

See Algorithmic Pattern Generator (APG), Memory Test Patterns, Memory Pattern Instruction Format.

The PINFUNC instruction is used to control several unrelated APG options. The table below lists each option and describes how the associated operand is used. All operands are optional and, if used, may be specified in any order. The entire PINFUNC instruction is optional if the default values, indicated below, are acceptable.

The default PINFUNC instruction is:

```
PINFUNC TSET1, PS1, VIHH1, NOADHIZ
```

The PINFUNC instruction takes the form:

```
PINFUNC PS#, VIHH#, TSET#, ADHIZ, VPULSE, VVCOMP, VLATCHRESET,
        VOVER, VPS, VTSET, VVIHH, VVPULSE, VLEVELSET
```

The PINFUNC operands are described below:

Table 4.13.13.0-1 PINFUNC Instruction Operands

Operand	Purpose
PS#	Where # is from 1 to 64. Specifies the Pin Scramble Map to be enabled during the instruction. The Pin Scramble Map determines which data source is mapped to each pin channel. Default = PS1. In Mixed Memory/Logic Patterns PINFUNC PS# is effectively disabled by PINFUNC VPS.
VIHH#	Where # is from 1 to 64. Specifies the VIHH Map to be enabled during the instruction. A VIHH Map specifies which tester pins are switched to the VIHH voltage, with all other pins driving at the normal VIH and VIL levels. Default = VIHH1, which is defined by the system software to disconnect VIHH from all tester pins. VIHH1 cannot be modified by user code. In Mixed Memory/Logic Patterns is effectively disabled by PINFUNC VVIHH.
TSET#	Where # is from 1 to 32. Specifies one Time-sets (TSET) to be enabled during the current instruction. Default = TSET1. In Mixed Memory/Logic Patterns is effectively disabled by PINFUNC VTSET.
ADHIZ	Default operation causes any pin(s) which are pin scrambled to APG Data Generator outputs (see Pin Scramble Macros) to tri-state (excluding ADHIZ causes the pins to drive). The default operation may be inverted using adhiz(). Has no effect on pins which are not pin scrambled to APG data generator outputs.

Table 4.13.13.0-1 PINFUNC Instruction Operands *(Continued)*

Operand	Purpose
VPULSE	Note: this feature is not usable on Magnum 1. This note will be removed when this limitation is corrected. Causes DUT power supplies which have been enabled (see VPulse Function) to switch to the secondary (VPulse) level , set using `dps_vpulse()`. The test pattern must execute multiple instructions each containing VPULSE to allow time for the voltage to stabilize at the DUT. In Mixed Memory/Logic Patterns is effectively disabled by PINFUNC VVPULSE. If pattern execution ends on an instruction containing VPULSE the secondary (VPulse) level remains enabled in hardware. Note: the VPULSE operand may be used in the PINFUNC, VEC/RPT, VAR, and VPINFUNC instructions.
VVCOMP	Note: this feature is not usable on Magnum 1. This note will be removed when this limitation is corrected. Applies to Mixed Memory/Logic Patterns only. Selects the VAR Engine as the source of the DC strobe for the current instruction. In Mixed Memory/Logic Patterns effectively disables the MAR VCOMP DC strobe for the current instruction. See Magnum 1/2/2x Memory Pattern Instructions.
VLATCHRESET	Applies to Mixed Memory/Logic Patterns only. Selects the VAR Engine as the source of both the error flag RESET signal and the LATCH/NOLATCH signal, for the current instruction. See Error Flag vs. Error Latch. In Mixed Memory/Logic Patterns effectively disables the MAR RESET signal and MAR LATCH/NOLATCH signal for the current instruction. See Magnum 1/2/2x Memory Pattern Instructions.

Table 4.13.13.0-1 PINFUNC Instruction Operands *(Continued)*

Operand	Purpose
V0VER	Note: this feature is not usable on Magnum 1. This note will be removed when this limitation is corrected. Applies to Mixed Memory/Logic Patterns only. Selects the VAR Engine as the source of the over programming inhibit signal (0VER) , for the current instruction. In Mixed Memory/Logic Patterns effectively disables the MAR 0VER signal for the current instruction. See Magnum 1/2/2x Memory Pattern Instructions.
VPS	Applies to Mixed Memory/Logic Patterns only. Selects the VAR Engine as the source of the pin scramble selection (PS#) for the current instruction. Effectively disables the PINFUNC PS# selection for the current instruction.See Magnum 1/2/2x Memory Pattern Instructions.
VTSET	Applies to Mixed Memory/Logic Patterns only. Selects the VAR Engine as the source of the time-set selection (TS#) for the current instruction. Effectively disables the PINFUNC TSET# selection for the current instruction. See Magnum 1/2/2x Memory Pattern Instructions.

Table 4.13.13.0-1 PINFUNC Instruction Operands *(Continued)*

Operand	Purpose
VVIHH	Applies to Mixed Memory/Logic Patterns only. Selects the VAR Engine as the source of the VIHH Map selection (VIHH#) for the current instruction. Effectively disables the PINFUNC VIHH# selection for the current instruction. See Magnum 1/2/2x Memory Pattern Instructions.
VVPULSE	Applies to Mixed Memory/Logic Patterns only. Selects the VAR Engine as the source of the signal (VPULSE) used to switch one or more DUT Power Supply(s) to their secondary voltage level for the current instruction. See dps_vpulse(). Effectively disables the PINFUNC VPULSE selection for the current instruction. See Magnum 1/2/2x Memory Pattern Instructions.
VLEVELSET	Used when Controlling Magnum 1 Levels from the Test Pattern in Mixed Memory/Logic Patterns. In the LSENABLE Pattern Instruction determines whether the pin list is stored in the memory pattern hardware or logic pattern hardware. The LEVELSET Pattern Instruction determines whether the UDATA value or the VUDATA value is used as the set/tweak value.

Example

```
% PINFUNC   PS55, VIHH42, TSET2
  MAR       INC
```

4.13.14 USERRAM Instruction

See APG User RAM, Memory Test Patterns, Memory Pattern Instruction Format.

Description

The USERRAM instruction is used to control/use the APG User RAM hardware.

Rules:

- The USERRAM instruction does not have any default operands.

- Values are copied into or out of the APG User RAM at a specified address. In most USERRAM instructions the address is set explicitly, using USERRAM SourceA Operands and USERRAM SourceB Operands. However, using the USERRAM SET/GET URAMINCR and SET/GET URAMDECR instructions, the address is determined by the contents of the User RAM Address Index Register, which is incremented or decremented during use. See User RAM Address Index Register.
- Using USERRAM GET and SET, an APG User RAM address is specified using USERRAM SourceA Operands and an APG Register is specified using a USERRAM SourceB Operands.
- The UDATA field is implicitly used by USERRAM instructions. This means that the UDATA field cannot be used for other purposes, including implicit use by other portions of the same pattern instruction. See UDATA Instruction.
- Many of the other APG pattern instruction operations are not allowed in a pattern instruction which contains a USERRAM instruction. ONLY the following instruction operations are allowed in an instruction which contains a USERRAM instruction:

```
XALU      OXMAIN or OXBASE or OXFIELD
YALU      OYMAIN or OYBASE or OYFIELD
PINFUNC   TSET#, PS#, VIHH#, VTSET, VPULSE, VPS, VVIHH,
          VVCOMP, VVPULSE, VLATCHRESET, VOVER
MAR       VCOMP, LATCH, RESET, OVER, READ
          READZ, READV, NOLATCH, LATCH, RSTTMR, INTEN,
          TIMEN
DATGEN    All data source selection options (DATDAT, etc.)
CHIPS     All options
```

Usage

```
USERRAM <Operation>, <SourceA>, <SourceB>
```

where:

Operation specifies the basic USERRAM operation to be performed: GET, SET. See USERRAM Operation Operands.

SourceA identifies a specific APG User RAM address which, depending on the specified **Operation**, can be the source and/or destination of the **Operation**. See USERRAM SourceA Operands.

SourceB identifies a specific APG hardware register. See USERRAM SourceB Operands.

The table below summarizes the operands for each field of the USERRAM instruction:

Table 4.13.14.0-1 USERRAM Instruction Operands

Operation	SourceA	SourceB		
GET SET	URAM1 to URAM4096 URAMINCR URAMDECR	XMAIN YMAIN	XBASE YBASE	XFIELD YFIELD

Examples

The following example copies the value from APG User RAM address 6 (URAM6) to the APG's XMAIN register:

```
% USERRAM GET, URAM6, XMAIN
```

The following example copies the value from the APG's YMAIN register to APG User RAM address 6 (URAM2):

```
% USERRAM SET, URAM2, YMAIN
```

4.13.14.1 USERRAM Operation Operands

See APG User RAM, Memory Test Patterns, USERRAM Instruction.

Description

The USERRAM instruction takes the following form:

```
USERRAM <Operation>, <SourceA>, <SourceB>
```

The Operation operand specifies the basic USERRAM operation to be performed, one of GET, SET.

Rules:

- There is no default Operation operand.

The table below describes the options available for the USERRAM Operation operand:

Table 4.13.14.1-1 USERRAM Operation Operands

Operand	Purpose	SourceA	SourceB
GET	GET the value from SourceA into SourceB. The Modification operand is not allowed.	APG User RAM	APG Register
SET	SET the value from SourceB into SourceA. The Modification operand is not allowed.	APG User RAM	APG Register

Example

See Examples.

4.13.14.2 USERRAM SourceA Operands

See APG User RAM, Memory Test Patterns, USERRAM Instruction.

Description

The USERRAM instruction takes the following form:

```
USERRAM <Operation>, <SourceA>, <SourceB>
```

The SourceA operand identifies a specific APG User RAM address which, depending on the specified Operation, can be the source and/or destination of the Operation.

Rules:

- SourceA can only be a APG User RAM address (URAM1-URAM4096) or one of URAMINCR or URAMDECR.
- SourceA does not have a default value.

The table below describes the options available for the USERRAM SourceA operand:

Table 4.13.14.2-1 USERRAM SourceA Operands

Operand	Purpose
URAM1 to URAM4096	Explicit APG User RAM address 1. ... to ... Explicit APG User RAM address 4096.
URAMINCR	The SourceA address is taken from the User RAM Address Index Register, which is incremented after the current operation is complete.
URAMDECR	The SourceA address is taken from the User RAM Address Index Register, which is decremented after the current operation is complete.
Note: the apg_user_ram_address_set(), apg_user_ram_address_get() functions use values which start with 0, which is equivalent to URAM1 in the test pattern..	

Example

See Examples.

4.13.14.3 USERRAM SourceB Operands

See APG User RAM, Memory Test Patterns, USERRAM Instruction.

Description

The USERRAM instruction takes the following form:

```
USERRAM <Operation>, <SourceA>, <SourceB>
```

The SourceB identifies the source of the second value used in selected Modification(s).

Rules:

The table below describes the options available for the USERRAM SourceB operand:

Table 4.13.14.3-1 USERRAM SourceB Operands

Operand	APG Register	Used Bit Positions
URAM1 to URAM4096	APG User RAM Address	n/a
XMAIN	X Main	17..0
XBASE	X Base	17..0
XFIELD	X Field	17..0
YMAIN	Y Main	17..0
YBASE	Y Base	17..0
YFIELD	Y Field	17..0

Regarding the table above note the following:

- The Hardware Register identifies which APG resource is accessed for each Operand value.
- Used Bit Position specifies the number of bits which are accessed.

Example

See Examples.

4.13.15 Minmax Pattern Example

See Memory Test Patterns.

Below is an example of a minmax read pattern using the Data Buffer Memory (DBM) as the data source. This is a simple example of reading a ROM code.

```
PATTERN( minmax )
```

```
// Initial Conditions
@{
   count( 1, amax() ); // Load COUNT1/RELOAD1 with max DUT address
   ymain( ymax() );    // Load YMAIN with maximum Y address
   xmain( xmax() );    // Load XMAIN with maximum X address
@}
// Instruction 1
// XALU/YALU increment the DUT address, X fast. The first execution
// will increment the address from xmax()/ymax() to 0x0. MAR CJMPNZ
// causes the instruction to loop until COUNT1 DECRements to zero
// (i.e. amax() cycles +1). MAR READ enables strobes on pins
// scrambled to the APG Data Generator outputs, and since NOLATCH
// is NOT included any failing strobe(s) will set the error latch
// on the associated pin(s) (see Error Flag vs. Error Latch).
// PINFUNC ADHIZ causes pins scrambled to the APG Data Generator
// to tri-state (assuming adhiz() did not invert normal operation).
// DATGEN BUFBUF selects the DBM as the data source for the data
// generator. Three chip selects pulse true. Operands not specified
// are set to values in the Default Memory Pattern Instruction.
%  Lool_Label:
   YALU      YMAIN, XCARE, CMEQMAX, INCREMENT, DYMAIN
   XALU      XMAIN, XCARE, CON, INCREMENT, DXMAIN
   COUNT     COUNT1, DECR, AON
   MAR       CJMPNZ, Lool_Label, READ
   PINFUNC   ADHIZ
   CHIPS     CS1PT,CS2PT,CS3PT
   DATGEN    BUFBUF
// Pattern is done. All undefined instructions/operands are set to
// default values (see Default Memory Pattern Instruction).
%  MAR    DONE
```

The `funtest()` function, executed in test block code, is used execute this `minmax` pattern. This first causes the initial conditions to execute, then starts the APG to execute the pattern instructions. The APG will stop when `MAR DONE` is executed or if a strobe fails (stop on `error`):

```
funtest( minmax, error );
```

4.13.16 Adaptive Programming Pattern Example

See Memory Test Patterns, Over-programming Controls and Parallel Test.

Note: this section should be read only after reviewing Error Flag vs. Error Latch.

Note: this section was written in 1999 and may be somewhat dated. The concepts remain valid, the details are suspect. Check with Nextest Applications for more modern examples.

Below is an example of a pattern that performs adaptive programming of a programmable memory device.

For the purpose of this example, assume this is a simple DUT that has an address bus, a data bus, and three chip selects. This DUT can be put into a special programming mode by applying a high voltage to the programming pin. In this example, the 3rd drive level (VIHH) is used and enabled in the test pattern by selecting VIHH set = `VIHH2` (configured using the VIHH Map Macros in the test program code).

The key section of this example shows how to read the DUT outputs, wait the correct number of cycles for error pipelining, perform a branch-on-error, and clear the error line in preparation for the next read.

The following assumptions are made:

- VIHH1 = VIHH is disabled on all pins (default)
- VIHH2 = VIHH applied to adaptive programming pin
- TSET1 = normal DUT read cycle timing
- TSET2 = 2uS cycle period used for DUT programming cycles
- TSET3 = 30nS cycle period, to generate fast error pipeline clocks
- TSET4 = 600nS cycle period, for VIHH settling time

TSET2 is used to time the duration of the VIHH programming pulse to the DUT (the interrupt timer could be used instead, but it is less accurate at small time values because it is asynchronous to the pattern). TSET3 will be used to pipeline error(s) back to the APG as quickly as possible. TSET4 is used to allow time for VIHH to settle to its programmed value.

In this example, if a given address fails to program after 500 2uS programming pulses the pattern will terminate. The flow chart below shows the adaptive programming pattern:

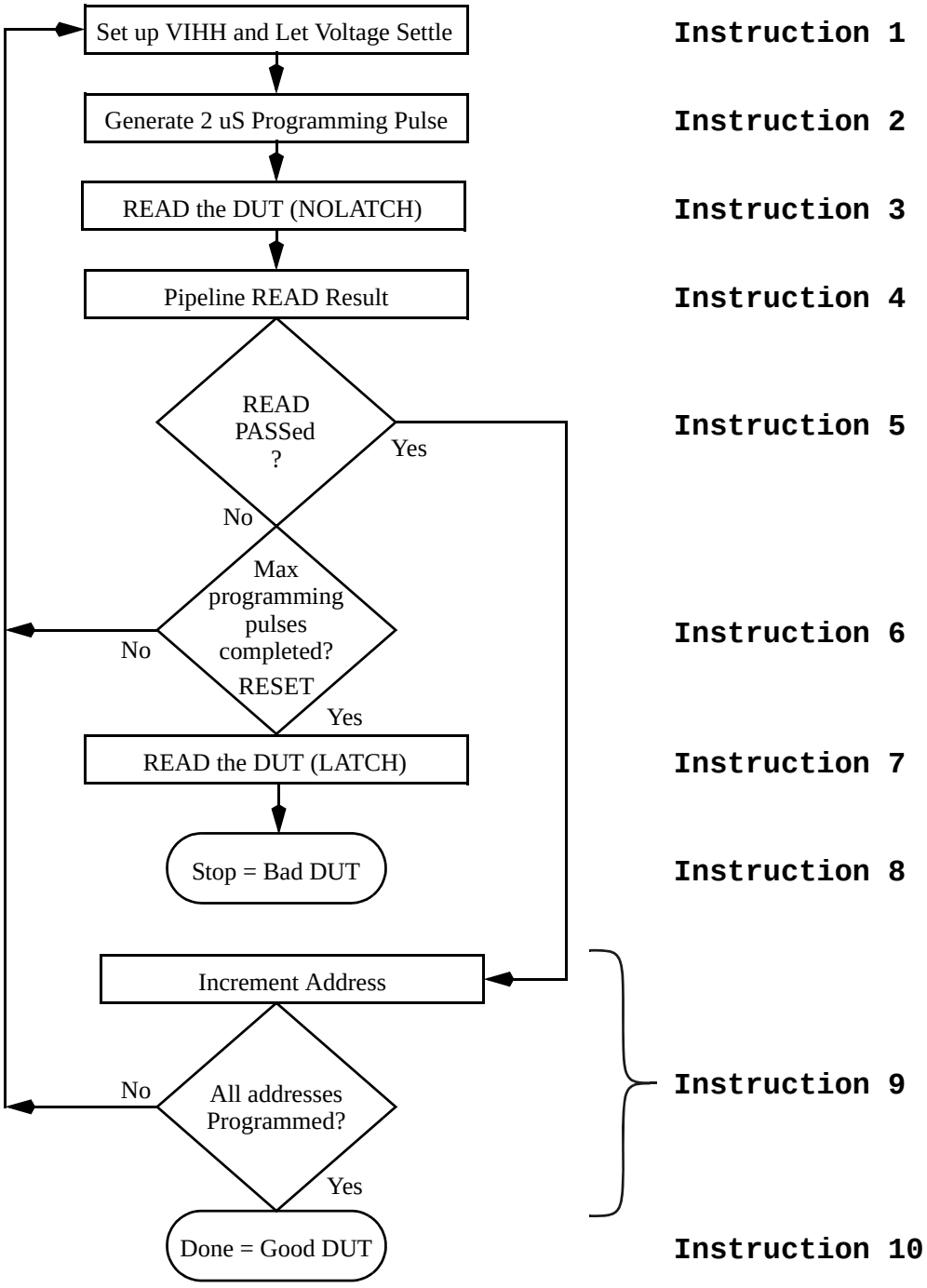


Figure-68: Example Adaptive Programming Flow Chart

```
PATTERN( adaptive_prog_pat )
// Initial Conditions
@{
   // COUNT1 used to increment through every address
   count( 1, amax() );// Load COUNT1/RELOAD1 with max DUT address
   // Load COUNT2/RELOAD2 with (500 -1). COUNT2 stores the maximum
   // number of programming pulses (-1) allowed at a given address
   count( 2, 499 );
   // Load COUNT3/RELOAD3 with (12 -1) = error pipeline length. This
   // is used to loop on an instruction which clocks the error
   // pipeline, to propagate any failing strobes to the APG error
   // logic used to control branch-on-error operations.
   count( 3, 11 );
   amain(0); // Load XMAIN and YMAIN = 0 = 1st address to program.
   // Set the data register to 0. This will be used as the data
   // source for programming and reading the DUT
   datreg( 0 );
@}
// Instruction 1
// Enable VIHH programming voltage by selecting VIHH2. Select TSET4
// to allow VIHH to settle before proceeding.
%  PINFUNC      TSET4, VIHH2
// Instruction 2
// Set CS1 = TRUE to send DUT a single programming pulse.
// The pulse duration is determined by TSET2 cycle period. VIHH2
// remains selected to apply the proper programming voltage. By
// default the data register (DATDAT) is selected as the data
// source. OYMAIN and OXMAIN are the default address generators.
%  Prog_pulse:
   CHIPS        CS1T
   PINFUNC      TSET2, VIHH2
// Instruction 3
// Read (strobe) the DUT to see if this address programmed
// correctly. TSET1 (default) is set up with spec DUT read timing.
// NOLATCH prevents PE error latch(s) from being set; error
// flag(s) will be used for branch-on-error (below) to
// conditionally re-program this same address. PINFUNC ADHIZ causes
// tester pins scrambled to the APG data generator to tri-state.
```

```
% MAR        READ, NOLATCH
  CHIPS      CS2T, CS3T
  PINFUNC    ADHIZ

// Instruction 4
// Loop here to pipeline error(s) from PE to APG, in preparation for
// branch-on-error below. TSET3 used to execute these cycles at
// minimum cycle period, to minimize the waiting time.
% Pipe_loop1:
  COUNT      COUNT3, DECR, AON
  MAR        CJMPNZ, Pipe_loop1
  PINFUNC    TSET3

// Instruction 5
// Branch-on-error based on PASS/FAIL result of earlier READ. If
// address was successfully programmed (no error) branch to the
// instruction with the label Next_address1. If the address was not
// successfully programmed execute the next instruction. Again,
// TSET3 used to minimize time.
% MAR        CJMPNE, Next_address1
  PINFUNC    TSET3

// Instruction 6
// Execution comes here only if the current address did not program
// (READ failed, branch-on-error came here). This instruction
// counts the number of programming pulses which have been applied
// at the current X/Y address. If the max have not been applied
// execution jumps back to apply another pulse. If the max number
// has been applied, the DUT is bad, and execution proceeds to the
// next instruction, which stops the APG. Also, RESET the error
// flags here in preparation for performing another READ.
% COUNT      COUNT2, DECR, AON
  MAR        CJMPNZ, Prog_pulse, RESET
  PINFUNC    TSET3

// Instruction 7
// This instruction is reached only if the DUT failed to
// program after receiving the maximum number of programming
// pulses. The DUT is bad, but the pattern must do a READ LATCH to
// set a PE error latch(es), which determine whether funtest()
// returns PASS or FAIL.
% MAR        READ
  CHIPS      CS2T, CS3T
```

```
// Instruction 8
// End the pattern. This instruction is reached when the DUT
// fails to program correctly.
% MAR     DONE

// Instruction 9
// Increment the address (X fast) and check (COUNT1) to see if all
// addresses have been programmed. If not, jump back to apply the
// programming pulse to the new address. TSET3 for speed.
% Next_address1:
   YALU      YMAIN, XCARE, CMEQMAX, INCREMENT, DYMAIN
   XALU      XMAIN, XCARE, CON, INCREMENT, DXMAIN
   COUNT     COUNT1, DECR, AON
   MAR       CJMPNZ, Prog_pulse
   PINFUNC   TSET3

// Instruction 10
// End the pattern. This instruction is reached when all addresses
// are successfully programmed.
% MAR     DONE
```

The following test block code will execute this pattern. This executes the initial conditions then starts the APG:

```
funtest( adaptive_prog_pat, error );
```

4.13.17 Over-programming Controls and Parallel Test

See Adaptive Programming Pattern Example, Error Flag vs. Error Latch, `error_flag_enable()`, and `over_inhibit()`.

Note: this section was written in 1999 and may be somewhat dated. The concepts remain valid, the details are suspect. Check with Nextest Applications for more modern examples.

When programming some programmable memory devices (Flash, EEPROM, etc.), it is often required that only the minimum amount of programming stimulus be applied; i.e. to not over-program the DUT.

Here, the term over-programming is used in the context of an adaptive programming test pattern, used to program these memory device types. In general, rather than apply the full

spec amount of programming stimulus (pulse count, programming time, etc.) an adaptive programming algorithm will:

- Apply less than spec programming stimulus.
- Read the DUT to see if it programmed successfully (i.e. branch-on-error).
- Repeat the previous 2 steps until either all DUT(s) program successfully or until the spec amount of programming stimulus has been applied. In the latter case, any DUT(s) which have not successfully programmed are considered defective.

See Adaptive Programming Pattern Example for more details.

Over-programming occurs when the programming stimulus continues to be applied even after a device has programmed successfully. When testing a single DUT, this is not difficult (see Adaptive Programming Pattern Example). When testing multiple DUTs in parallel, to prevent over-programming requires specialized hardware, to disable programming stimulus on a per-DUT and per-address basis. This allows DUT(s) which have not successfully programmed to continue to receive stimulus while the other DUT(s) do not, on a per-DUT, per-address basis.

An example of the basic adaptive programming algorithm used to program one DUT is described in the Adaptive Programming Pattern Example: and the Example Adaptive Programming Flow Chart. The latter should be reviewed before proceeding. Note that the model in this flow chart must be adjusted to correctly program mutliple DUTs in parallel (as described below and in Modified Adaptive Programming Pattern Outline). Also, this is the time to review Error Flag vs. Error Latch, to understand these two signals and how they operate.

When testing a single DUT, as indicated in the Example Adaptive Programming Flow Chart, at the appropriate time the test pattern reads (strobes) the DUT outputs (`MAR READ`,`NOLATCH`) at the address currently being programmed, pipelines the result to the APG branch-on-error logic and branches based on whether the current address has been successfully programmed (all strobes = PASS = no error flags are set) or not (one or more strobes = FAIL = one or more error flags are set). If not, the programming loop repeats until either the current address is programmed successfully or until the maximum amount of programming stimulus has been applied (i.e. the DUT is bad). This process continues until all addresses have been successfully programmed or until an address fails to program correctly, at which time the pattern execution stops because the DUT is bad.

As indicated, the Adaptive Programming Pattern Example presumes that a single DUT is being tested. When testing multiple DUTs in parallel, two additional considerations exist:

- When a given DUT fails (i.e. is bad) the test pattern must continue to program the remaining DUTs, until all good DUTs are successfully programmed and all bad DUTs have been identified. However, to ensure the adaptive programming algorithm operates optimally, once a DUT has failed, the algorithm must ignore it while programming the remaining DUT(s). And, usually, the programming stimulus must be inhibited to the bad DUT(s).
- When a given DUT has been sucessfully programmed at the current address the programming stimulus must be inhibited while the remaining DUTs continue to be programmed at that same address. Then, at the next address, the programming stimulus must be re-enabled again for the remaining good DUTs.

As indicated, to ensure the adaptive programming algorithm operates optimally, once a DUT has failed, the programming loop must ignore PASS/FAIL signals from those DUT(s) when determining whether the current address has been successfully programmed. More specifically, the APG's branch-on-error logic must ignore the error *flags* from bad DUT(s). This is done using specialized hardware, by effectively setting the error *flags* for bad DUT(s) = PASS.

A bad DUT is identified when an error *latch* for that DUT is set. In the Modified Adaptive Programming Pattern Outline, this is done in instruction 12. If, after applying the programming stimulus the maximum number of times, if the current address is still not programmed correctly a `MAR READ,LATCH` is executed, which will set one or more error latches for bad DUT(s) (for the good DUTs, the read will PASS and no latches are set). Subsequently, as the pattern continues to program other DUT(s) the error *flags* from bad DUT(s) are ignored because an error *latch* is set for these DUT(s). As indicated above this is done using specialized hardware, which must be enabled using the `error_flag_enable()` function. This same hardware also inhibits the programming stimulus for any DUT with an error latch set (independent of the use of `MAR OVER`, more below).

When testing multiple DUTs in parallel, to prevent the over-programming of individual addresses additional specialized hardware is also required. As each address is programmed, once a given DUT is successfully programmed at that address the programming stimulus must be inhibited for that DUT while the pattern continues to program the remaining DUT(s). Then, at the next address, the programming stimulus must be enabled for all remaining good DUT(s), and the process repeated until all addresses have been programmed (or all DUTs determined to be bad). This requires:

- Identification of the specific programming stimulus being used, to allow it to be inhibited at the appropriate time in the test pattern. This is done using the `over_inhibit()` function.

- Using the MAR OVER token in the test pattern, to cause the hardware to inhibit the programming stimulus for any DUT(s) which *DON'T* have an error *flag* set. Note that in mixed memory/logic patterns, logic instructions can control OVER and LATCH/NOLATCH if enabled using PINFUNC VOVER and/or PINFUNC VLATCHRESET.
- Moving the MAR RESET to ensure it occurs after the programming stimulus has been applied but before the next read occurs. This is required to ensure the error flags from DUTs which do require additional programming are not cleared before the stimulus is applied. Using Magnum 1/2/2x, MAR RESET cannot be used for at least three cycles following the last instruction in which the over-inhibit must be effective. Note that in mixed memory/logic patterns, logic instructions can control RESET if enabled using PINFUNC VLATCHRESET.

The latter benefits from further explanation. During the adaptive programming loop, after the programming stimulus is applied, a MAR READ,NOLATCH is executed, to determine whether the current address has been successfully programmed. READ,NOLATCH only affects the error *flags*, which are *not* set for any DUT(s) which *have been* successfully programmed. Conversely, the error flags *are* set for any DUT(s) which have *not been* successfully programmed. Subsequently, if one or more error flags are set, the programming loop must repeat (branch-on-error), to continue to program DUT(s) which require it. As each loop is repeated, the stimulus is applied, then the error flags are reset and the read/branch process repeats.

Adding MAR OVER to the test pattern causes the hardware to inhibit the programming stimulus (as identified using over_inhibit()), but only for DUT(s) which DON'T have an error flag set; i.e. DUTs which passed the previous MAR READ,NOLATCH. Since the user writes the test pattern which controls when this stimulus is applied and when the error flags are reset they are responsible for adding the MAR OVER token to the pattern and ensuring that MAR RESET is not applied at the wrong time. See Modified Adaptive Programming Pattern Outline.

Note: the APG pipeline model for MAR OVER is quite complex and thus not documented. Instead, it is expected that MAR OVER will be included in all pattern instructions in the program/reset/strobe/pipeline/branch sequence: it is harmless to do so.

Using Magnum 1/2, the special hardware noted above is replicated for every 8 pins and one DPS (i.e. a_1 through a_8, b_1 through b_8, etc). This restricts the minimum DUT size to 8 pins but also works when a DUT spans more than 8 pins. However, additional restrictions apply depending on the programming stimulus being used:

- If using VIHH as the programming stimulus, no additional restrictions exist.

- If using DPS as the programming stimulus, the over-programming inhibit hardware requires that the DPS Output Mode be configured/used in VPulse mode (`t_dps_vpulse`). The over-programming inhibit hardware inhibits the DPS programming stimulus by forcing the DPS to its primary output voltage, over-riding (inhibiting) the secondary output voltage (vpulse).
- If using a normal PE driver signal as the programming stimulus, the over-programming inhibit hardware forces the drive state to VIH (presuming the active programming state to be active-low). This affects BOTH the A/B tester channels which share a given timing generator.

Modified Adaptive Programming Pattern Outline

The memory pattern outline below presumes the use of both `error_flag_enable(FALSE)` and `over_inhibit()`:

Inst	Details
1	Set new address
2	`MAR READ,NOLATCH,OVER` This `READ` is needed to set the error flags (to FAIL) before programming the current address. At each new address this `READ` should fail so that instructions below using `MAR OVER` will apply the programming stimulus. For the very first execution (first address only) all error flags are enabled since no `READ`/`LATCH` has yet occurred to set any error latches, which if `error_flag_enable()` is used, would disable the error flags for bad DUTs. For subsequent loop iterations/addresses, some error flags may be disabled (forced PASS), per DUT, depending on the latched PASS/FAIL result in instruction 12.
3	Pipeline errors, include `MAR OVER`. Required for proper `MAR OVER` operation in next instruction.
4	Apply programming stimulus. Include `MAR OVER`. The programming stimulus will be disabled for some DUT(s), as follows: - Stimulus to DUTs which previously failed `MAR READ`/`LATCH` (instruction 12) are disabled for the duration of the pattern (presuming `error_flag_enable()` and `over_inhibit()` are setup). - Stimulus to DUTs which DON'T have an error flag set (in instructions 4, 12 or 8) is disabled while programming the current address only.
5	If required, loop with stimulus applied. Include `MAR OVER`.

Inst	Details
6	Remove programming stimulus and loop 3 cycles (total) before resetting the error flags. Include MAR OVER. 3 cycles are required to ensure proper over-programming operation.
7	MAR RESET,OVER to reset error flags. See previous instruction.
8	MAR READ,NOLATCH,OVER = adaptive programming read.
9	Pipeline errors. Include MAR OVER. Pipeline cycles are required for proper branch-on-no-error (instruction 10) and proper MAR OVER operation during instructions 4 and 5.
10	Branch-on-no-error (MAR CJMPNE,OVER) to instruction 13 if previous MAR READ,NOLATCH passed (no error flags were set). Remember, the error flag has been disabled for devices which previously failed instruction 12 (i.e. DUTs which have an error *latch* set are bad).
11	Programming limit check. Include MAR OVER Branch to instruction 4 if the maximum stimulus has not been applied.
12	If pattern execution reaches here, at least 1 DUT has not been successfully programmed at the current address. Need to perform MAR READ,LATCH,OVER to set some error latch(es) for those DUTs (they are bad). This will enable the over_inhibit() logic for these DUT(s) while the remaining good DUT(s) continue to be programmed for the remainining addresses. For those devices which fail here, their error flags are disabled (forced PASS) for the duration of the pattern, causing programming stimulus to be disabled during cycles which use the MAR OVER option, preventing any subsequent programming of the bad DUTs.
13	Check if all address have been programmed. If more addresses remain to be programmed go to instruction 1, otherwise fall through to MAR DONE.
14	MAR DONE

4.14 Logic Test Patterns

See Test Pattern Programming.

The Logic Test Patterns sections include the following:

- Overview
- Logic Vector Syntax
  - Logic Vector Bit Codes
  - 3-bits per Pin
- Magnum 1/2/2x Logic Pattern Rules
  - LVM Branch/Label Limitations
- VECDEF Compiler Directive
- VEC Pattern Instruction
- RPT Pattern Instruction
- Optional VEC/RPT Instruction Parameters
- STARTLOOP / ENDLOOP Logic Vector Instructions
- VAR Instruction
  - VAR Branch-condition Operands
  - VAR Address Operand
  - VAR Interrupt Operands
  - VAR Error-control Operands
  - VAR Misc Operands
- VCOUNT Instruction
  - VCOUNT Counter Operands
  - VCOUNT Function Operands
- VPINFUNC Instruction
- VUDATA Instruction
- Sync Loops

4.14.1 Overview

The Magnum 1 test system contains two sources of test pattern data:

- Algorithmic Pattern Generator (APG) when executing Memory Test Patterns.
- Combined Logic Vector Memory (LVM) / Scan Vector Memory (SVM) for stored Logic Test Patterns and Scan Test Patterns.

The diagram below shows key Magnum 1 architecture features:

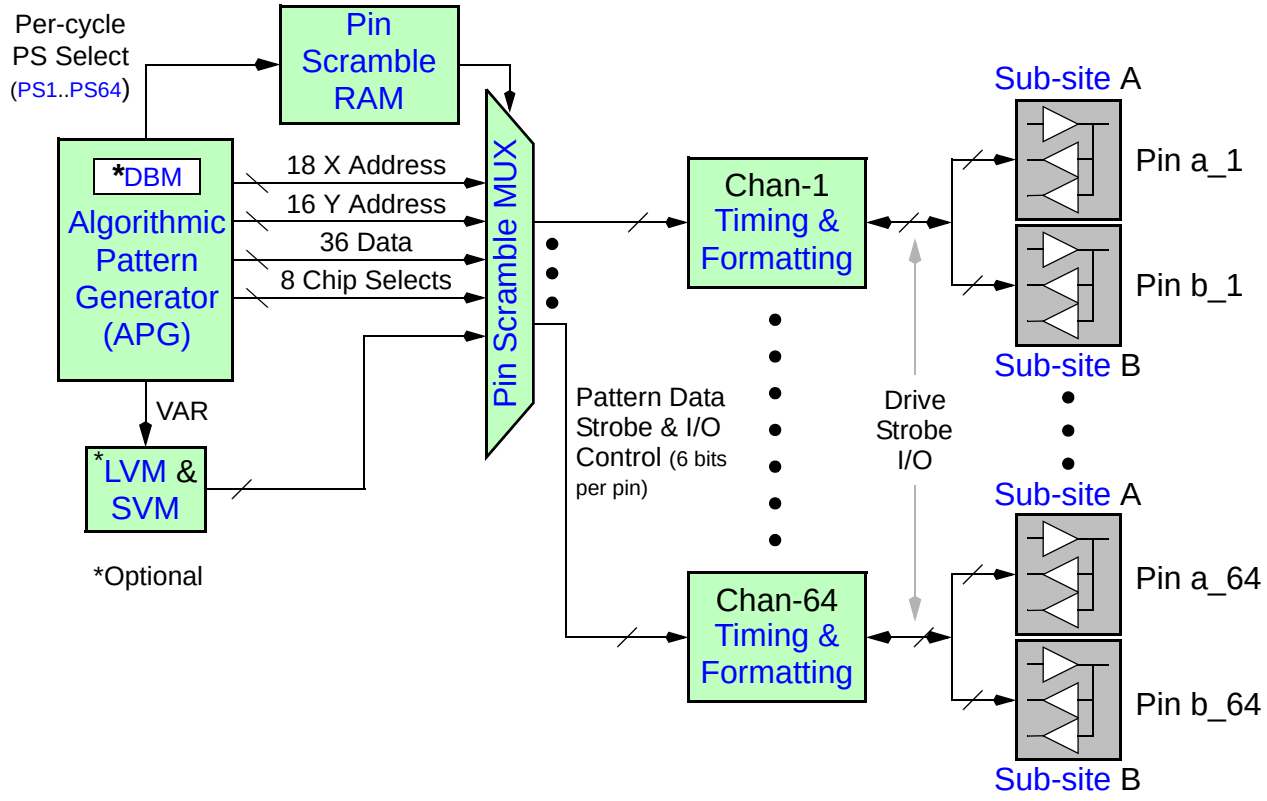


Figure-69: Test Pattern Data Source Hardware Architecture

Using the Pin Scramble MUX, the source of pattern data for each timing channel can be selected on a per-channel/per-cycle basis, at full tester speed. In other words, for any given pin-pair, the source of pattern data can be selected on a per-cycle basis from any APG address, data, or chip select data bit, or one LVM/SVM data bit (2 bits in Double Data Rate (DDR) Mode). This section discusses Memory Test Patterns, also see Logic Test Patterns and Mixed Memory/Logic Patterns.

Memory Test Patterns use user-written *instructions* to control how the APG hardware *algorithmically generates* pattern information. A logic vector pattern defines pattern information to be stored on disk, which is loaded into Logic Vector Memory (LVM)during the test pattern load sequence. The LVM hardware option consists of physical memory (RAM) which can store various numbers of logic test vectors based on the size of the memories installed. The LVM size will grow larger as technology and market demand evolves.

The contents of a simple logic-only test pattern named "myPat1.pat" is shown below:

```
PATTERN( myPat1 )
   %  VEC     00110 HL0XX
   %  RPT 3   00010 HL0XX // Comment
   %  label_1:
      VEC     01110 XX0LL  TSET3, PS9, VIHH2 // Comment
   %  VEC     00110 HL0XX
      VAR     DONE
```

Refering to the example above, myPat1 is the pattern name used in the test program to refer to this pattern. The following example executes the myPat1 pattern to completion and assigns the PASS / FAIL result to the variable "result":

```
int result = funtest( myPat1, finish ); // See funtest()
```

4.14.2 Logic Vector Syntax

See Logic Test Patterns.

A Logic Test Pattern uses pattern instructions and syntax different than used for Memory Test Patterns. It is also possible to mix logic and memory pattern instructions to create Mixed Memory/Logic Patterns.

Note: the term *logic vector* is historical. In this document a logic vector and a logic pattern instruction (delimited by the Pattern Instruction Identifier (%) character) are equivalent and used interchangeably.

The simplest form of a logic vector has three required parts:

```
% INSTRUCTION bit-pattern(s)
```

For example:

```
% VEC     XHL10     // Single vector instruction
% RPT 4   XHL10     // Repeat instruction
```

Note: in any Multi-DUT Test Program testing more than 2 DUTs, a VECDEF Compiler Directive is required (not optional) in any test patterns containing logic instructions. This is not shown above.

Note the following:

- Each logic vector begins with the required Pattern Instruction Identifier (%).
- Each logic vector requires an instruction, either VEC or RPT.
- The bit-pattern(s) use Logic Vector Bit Codes to specify, on a per channel basis, the drive/strobe data, I/O control and strobe enable for each channel (each character/token represents one channel). The token specified encodes the drive/strobe data, I/O control and strobe enable for that channel, see 3-bits per Pin. White-space usage is arbitrary in the bit-pattern field and between other parameters.

A logic instruction may also include several Optional VEC/RPT Instruction Parameters and sub-instructions:

- Pattern Labels may be used in logic patterns, for readability and as conditional branch and subroutine targets. Restrictions exist, see LVM Branch/Label Limitations. For example:

```
% myLabel_1:
     VEC  XHL10   XHL10

% myLabel_2:   RPT 4   XHL10   XHL10
```

- Comments are supported, see Comments in Test Patterns.
- A time-set may be specified, see Time-sets (TSET) and Optional VEC/RPT Instruction Parameters. For example:

```
% VEC     XHL10   XHL10   // Default TSET1
% VEC     XHL10   XHL10,  TSET2
% RPT 4   XHL10   XHL10,  TSET4
```

- A Pin Scramble Map may be specified, see Optional VEC/RPT Instruction Parameters. For example:

```
% VEC     XHL10   XHL10   // Default PS1
% VEC     XHL10   XHL10,  PS12
% RPT 4   XHL10   XHL10,  PS13
```

- A VIHH Map may be specified, see Optional VEC/RPT Instruction Parameters. For example:

```
% VEC     XHL10   XHL10   // Default VIHH1
% VEC     XHL10   XHL10,  VIHH4
% RPT 4   XHL10   XHL10,  VIHH2
```

- Any combination of Time-sets (TSET), Pin Scramble Map, and/or VIHH Map may be specified. For example:

```
% VEC     XHL10   XHL10   // Default TSET1, PS1, VIHH1
% VEC     XHL10   XHL10,  TSET2, PS5, VIHH4
% RPT 4   XHL10   XHL10,  PS2, TSET19 // Default VIHH1
```

- Maverick-II and Magnum 1/2/2x have additional instruction options, see Magnum 1/2/2x Logic Vector Instructions. For example:

```
% VEC       XHL10   XHL10, NOLATCH, VPULSE
  VCOUNT    COUNT3, COUNTVUDATA
  VAR       GOSUB, myLabel_1
  VPINFUNC  TSET4, VIHH3, PS9
  VUDATA    0xFF
```

- The Test Pattern Line Continuation Character (" \") can be used to split long vectors across multiple lines. For example:

```
% VEC     XHL10XHL10   XHL10XHL10 \
          XHL10XHL10   XHL10XHL10, TSET4, PS2
```

- The `#define` compiler directive, can be used in pattern files to improve readability and maintainability.
- Multi-vector loops are possible. See STARTLOOP / ENDLOOP Logic Vector Instructions.
- Pattern Subroutines are supported in logic patterns.

The information above applies to pure Logic Test Patterns; i.e. patterns without explicit memory pattern instructions. However, the entire Memory Test Patterns instruction set can be combined with the Logic Test Patterns instruction set, creating a mixed logic and memory pattern. See Mixed Memory/Logic Patterns.

Example Logic Pattern

```
%% VECDEF p1, p2, p3, p4, p5, p6, p7
PATTERN( myPattern )
% VEC       0011    XXX     // Defaults to TSET1, PS1,VIHH1
STARTLOOP 55                // Multi-vector loop, 55 iterations
% VEC       0101 HLX
% RPT 5     1100 LLH, TSET2// Repeat this vector 5 times,
                            // use timeset TSET2, PS1,VIHH1
% VEC       01HL HHH        // Defaults again
ENDLOOP                     // End of loop
% VEC       10XX XXX
```

```
% VEC        10XX XXX
  VAR        DONE
```

4.14.2.1 Logic Vector Bit Codes

See Overview, Logic Test Patterns.

Description

Each logic test vector includes a series of vector bit codes (tokens), each consisting of a character from Logic Vector Bit Codes. Each character encodes the drive state, I/O control and strobe enable for one timing channel.

For example, the `HL01XXXX` tokens below are the bit codes for 8 timing channels:

```
% VEC   HL01XXXX
```

Unless otherwise specified, the first token corresponds to tester channel 1, the second bit to channel 2, etc. Or, the VECDEF Compiler Directive can used to specify how pattern tokens are mapped to timing channels.

Note: in any Multi-DUT Test Program testing more than 2 DUTs, a VECDEF Compiler Directive is required (not optional) in any test patterns containing logic instructions. This is not shown above.

To improve readability, white space is allowed anywhere in a set of vector bit codes. The following examples result in identical operation:

```
% VEC   HL01XXXX
% VEC   HL01 XXXX
% VEC   HL 01XX XX
% VEC   H L 0 1 XXX X
% VEC   H L \
        0 1 \
        XXX X
```

The following short-hand notation is available:

```
% VEC   (n:bit-pattern)
```

where, **bit-pattern** specifies a set of bit code(s) and **n** is the number of times to repeat bit-pattern in the vector. Multiple sets of (n:bit-pattern) can be included in a test vector.

Usage

Table 4.14.2.1-1 Logic Vector Bit Codes

Code	Pin Function Performed
0	Drive logic-0
1	Drive logic-1
L or l	Tri-state and strobe for logic-0
H or h	Tri-state and strobe for a logic-1
X or x	Tri-state, no strobe
.	Same as X
Z or z	Tri-state and strobe for a tri-state
V or v	Tri-state and strobe for *valid* i.e. logic-1 and logic-0
It is recommended that the lower-case L not be used since it is quite difficult distinguishing it from the numerical 1 (1l1l1).	

Note: it is illegal to strobe in the last vector of a logic pattern;
i.e. the last vector **must not contain** any Z, V, H or L tokens.

Examples

The following vector operates as noted below. The description presumes that the tester channels noted have t_lvm selected in the current Pin Scramble Map. The indicated tester channel numbers presume that a VECDEF is not specified; if one was specified, the VECDEF definition would determine which tester channel was controlled by each pattern token:

```
% VEC   HL01XXXXZV
```

- H = Tri-state tester channels 1a and 1b and strobe for logic-1.
- L = Tri-state tester channels 2a and 2b and strobe for logic-0.
- 0 = Drive logic-0 on tester channels 3a and 3b.
- 1 = Drive logic-1 on tester channels 4a and 4b.

- X = Tri-state tester channelss 5a..8a and 8b..8b.
- Z = Tri-state tester channels 9a and 9b and strobe for tri-state.
- V = Tri-state tester channels 9a and 9b and strobe for *valid* i.e. logic-0 and logic-1.

Note that in Multi-DUT Test Programs, all pattern/timing signals are disabled to DUT(s) which are not in the Active DUTs Set (ADS).

The following two logic instructions are functionally identical:

```
% VEC  (4:01)    (3:1)   0
% VEC  01010101  111     0
```

4.14.2.2 3-bits per Pin

See Logic Vector Bit Codes.

Note: this information is included for historical reference. The references to PEL, PEE, PES originated with the *Megatest Q2* family of test systems. No references are made to the underlying concepts anywhere in the Magnum 1/2/2x software.

In a logic vector, each bit-pattern token (H, L, 0, 1, X, V, Z) actually represents 3-bits. Each character encodes these three bit values for each tester channel:

Table 4.14.2.2-1 Logic Pattern Hardware Bit States

Pin Function	Pattern Token	PEL	PEE	PES
Drive Logic 0	0	0	0	0
Drive Logic 1	1	1	0	0
Tri-state Driver & Strobe Logic 0	L	0	1	1
Tri-state Driver & Strobe Logic 1	H	1	1	1

Table 4.14.2.2-1 Logic Pattern Hardware Bit States

Pin Function	Pattern Token	PEL	PEE	PES
Tri-state Driver & Strobe Valid	V	0	0	1
Tri-state Driver & Strobe Tri-state	Z	1	0	1
Tri-state Driver	X	0	1	0

4.14.3 Magnum 1/2/2x Logic Pattern Rules

See Logic Test Patterns.

The following rules apply to Magnum 1/2/2x Logic Test Patterns and to logic instructions of Magnum 1/2/2x Mixed Memory/Logic Patterns:

- In any Multi-DUT Test Program testing more than 2 DUTs, a VECDEF Compiler Directive is required (not optional) in any test patterns containing logic instructions.
- The RPT instruction cannot be used in the last pattern instruction, the instruction following a STARTLOOP statement or the instruction before a ENDLOOP statement.
- The maximum RPT loop count = 2^{32}.
- A STARTLOOP statement cannot be used before the first pattern instruction or after the last pattern instruction.
- Multiple vector loops (see STARTLOOP / ENDLOOP Logic Vector Instructions) may include up to 3 levels of nesting.
- A STARTLOOP statement must be followed by a VEC instruction. A ENDLOOP statement must be preceded by a VEC instruction.
- Logic Pattern Subroutines of < 24 execution counts will be loaded into SRAM (see Logic Vector Memory (LVM)). The SRAM can store up to 8K vectors per test pattern. This means a given Logic Test Pattern can define at least 340 unique subroutines which are < 24 execution counts, each consisting of 24 discrete VEC instructions (RPT instructions account for multiple execution counts and thus can decrease the amount of SRAM used). SRAM based subroutines are not subject to the LVM Branch/Label Limitations listed below.

- DRAM based subroutines, i.e. subroutines > 24 execution counts, are subject to the LVM Branch/Label Limitations listed below.
- When a logic instruction controls functional strobes, extra error pipeline cycles are required to propagate the error flag signals to the branch logic. This is in addition to what are required for APG error pipelining. See Error Pipeline Requirements.

The Magnum 1/2/2x Logic Vector Memory (LVM) architecture uses DRAM to store both the test pattern logic instructions and logic pattern drive/expect data. During pattern execution, logic instructions are sequentially transferred from DRAM into an instruction FIFO, for synchronization with the tester system clock. See LVM Branch/Label Limitations for rules relating to a rare but important rule related to conditional branch instructions in logic patterns.

4.14.3.1 LVM Branch/Label Limitations

See Logic Test Patterns, Magnum 1/2/2x Logic Pattern Rules, Check for LVM Branch/Label Violations.

The Magnum 1/2/2x Logic Vector Memory (LVM) is DRAM. During pattern execution, logic instructions are sequentially transferred from DRAM into an instruction FIFO, for synchronization with the tester system clock. Because the DRAM access rate is only slightly faster than the maximum pattern data rate it is possible for a very specific sequence of pattern instructions to cause a *FIFO deficit*. If enough consecutive FIFO deficits occur a *FIFO under-run* occurs.

A FIFO under-run means the FIFO runs out of pattern instructions, resulting in faulty pattern execution (bad). Fortunately, this instruction sequence is not typical.

The logic instruction sequence which causes a FIFO deficit consists of a series of 6 consecutive branch instructions, but not all branch options apply (more below). And, it takes many FIFO deficits in very close proximity to cause a FIFO under-run The FIFO under-run conditions are impractical for the pattern compiler to detect directly. Instead, a simpler rule is defined:

> *Any logic pattern branch instruction within 5 execution counts after a pattern instruction with a label may cause a FIFO deficit.*

Note the following:

- Patcom will check this rule if **`Check for LVM Branch/Label Violations`** is enabled in the APFP Dialog, or `-w` is specifed as a command line option to Patcom.

- The Pattern Label is important only if it is being used as a branch target, but the pattern compiler can't know which labels are significant and which are not.
- The rule check is not needed if the logic pattern does not use branch instructions.
- As indicated above, not all logic branch instructions can cause a deficit. The following branch instructions cannot cause a FIFO deficit:
  - Counter-based branch instruction i.e. VAR CJMPNZ, VAR CSUBZ, etc.
  - Unconditional subroutine return i.e. VAR RETURN
  - RPT instruction
  - VAR DONE

  Also, STARTLOOP / ENDLOOP will not generate a FIFO deficit.
- A compiler warning of the rule violation does not mean that a FIFO under-run will occur. As noted above, it takes a series of FIFO deficits in close proximity to cause a FIFO under-run. The compiler warning only advises the user of a potential FIFO deficit.
- A FIFO deficit can never occur if the cycle periods in use are > 240nS.
- Logic subroutines executed from the SRAM are immune from FIFO deficits. See Magnum 1/2/2x Logic Pattern Rules.
- The first instruction in a logic pattern has an implicit label (the PATTERN name) and thus is considered for the purposes of this rule check.
- The instruction after a STARTLOOP statement and before an ENDLOOP statement are not considered to have a label for the purpose of this rule check.
- A RPT instruction and a VAR DONE instruction are not considered to have a label for the purpose of this rule check.
- The rule description uses the term *execution counts*, not *pattern instructions*, since there may be more than one execution count per instruction (RPT for example):
  - Each straight-line vector (VEC) is 1 execution count.
  - Each SVEC (scan vector) is 1 execution count.
  - For RPT instructions, the execution count is the RPT operand value.
  - The instruction after a STARTLOOP statement must be VEC and thus is 1 execution count.
  - The instruction before and ENDLOOP statement must be VEC and thus is 1 execution count.

4.14.4 VECDEF Compiler Directive

See Logic Test Patterns, Logic Vector Syntax, Logic Vector Bit Codes.

Description

VECDEF is a Patcom compiler directive, used to describe the mapping of DUT pins to columns of pattern tokens in a Logic Test Pattern.

Note: in any Multi-DUT Test Program testing more than 2 DUTs, a VECDEF directive is required (not optional) in any test patterns containing logic instructions.

When VECDEF is not used, the columns of VEC pattern tokens are mapped to physical tester channels in numerical order, without regard to whether a given channel is used/connected at the DUT or how the DUT pins are organized (a bus for example). This means that the test pattern must contain a pattern data column for every tester channel (up to the last channel used) even when a given pin isn't actually used. For example, the following example represents an 8-bit data bus connected to tester channels 1,3,4,6,10,12,15,16, leaving tester channels 2, 5, 7, 8, 9, 11, 13, and 14 unused:

```
                          1111111
Tester Channel->  1234567890123456
                  ----------------
          % VEC   1X01X01XX0X1XX10
          % VEC   HXLHXLXXXLXHXXHL
             ... etc ...
```

Note that when the VECDEF is not used, it is not possible when viewing the test pattern to know which pins are not used since the X token is a legitimate pattern token for used pins as well as unused pins (used pin 7 above uses X in the 2nd vector).

Using the VECDEF directive enables the following logic pattern compile-time features:

- Define which DUT pins are used and how they are mapped to the logic pattern's pattern data columns.
- Specify default token values for selected pin(s), allowing the pattern to contain fewer columns.
- Specify different VECDEFs per pin assignment table. See VECDEF per Pin Assignment Table.

When using the VECDEF directive, pins which are not defined in the Pin Assignment Table cannot have corresponding tokens in the test pattern; i.e. unused pins can't appear in the pattern.

Any pins in the Pin Assignment Table which are not included in a VECDEF directive are assigned the 'X' Logic Vector Bit Code.

Multiple VECDEF directives may appear in a pattern. Since VECDEF is a compiler directive, vectors are interpreted in the order in which they appear, not the order in which the are executed. Also note that a VECDEF directive remains in effect across multiple patterns in the same pattern file, but not between pattern files.

A VECDEF directive can span multiple source file lines but the Test Pattern Line Continuation Character (' \') must not be used. Just type <return> to continue the line.

Usage

```
%% VECDEF DUT_pin_name, DUT_pin_name, DUT_pin_name, ...,
   DUT_pin_nameX = [DUT_pin_name,0,1,H,h,L,l,X,x,Z,z,V,v,.],...,
      [0,1,H,h,L,l,X,x,Z,z,V,v,.]
```

where:

%% VECDEF is a compiler directive specifying the format of subsequent vectors.

DUT_pin_name is a user-defined DUT pin name, spelled exactly (case sensitive) as used as argument 1 to the various ASSIGN( ) macros in the Pin Assignment Table. The order the pin names are listed defines the mapping of the pattern data columns to DUT pins in subsequent vectors. A DUT pin name must be listed for each pattern data column and, a pattern data column must be specified for each DUT pin name listed in the VECDEF directive. Arguments for all specified pins must appear before any of the other arguments listed below.

DUT_pin_name = [DUT_pin_name,0,1,H,h,L,l,X,x,Z,z,V,v,.] This form allows a default pattern token value to be specified for one DUT_pin_name. See Logic Vector Bit Codes. Subsequently, this pin will not have a pattern data column in the pattern. The right side of the "**=**" can be a Logic Vector Bit Code value or the DUT_pin_name of a DUT pin which does have a pattern data column in the test pattern. This feature makes it possible to set default values for pins that do not change in a given pattern.

Note: when using VECDEF to specify a default value for a pin it is not legal to include a pattern data column for that pin in any test vectors in which the VECDEF is in effect.

[0,1,H,h,L,l,X,x,Z,z,V,v,.] as the last field of the VECDEF statement specifies a pattern data value that applies to all remaining pins not specified by the previous arguments. These are all Logic Vector Bit Codes.

Example

```
DUT_PIN( dp1 ){}
DUT_PIN( dp2 ){}
DUT_PIN( dp3 ){}
DUT_PIN( dp4 ){}
DUT_PIN( dp5 ){}
DUT_PIN( dp6 ){}
DUT_PIN( dp7 ){}
DUT_PIN( dp8 ){}
DUT_PIN( dp9 ){}
DUT_PIN( dp10 ){}
DUT_PIN( dp11 ){}
DUT_PIN( dp12 ){}
DUT_PIN( dp13 ){}
DUT_PIN( dp14 ){}
DUT_PIN( dp15 ){}
DUT_PIN( dp16 ){}
DUT_PIN( dp17 ){}
DUT_PIN( dp18 ){}
DUT_PIN( dp19 ){}
DUT_PIN( dp20 ){}
DUT_PIN( dp21 ){}
DUT_PIN( dp22 ){}
DUT_PIN( dp23 ){}
DUT_PIN( dp24 ){}
DUT_PIN( dp25 ){}
DUT_PIN( dp26 ){}
DUT_PIN( dp27 ){}
DUT_PIN( dp28 ){}
DUT_PIN( dp29 ){}
DUT_PIN( dp30 ){}
PIN_ASSIGNMENTS( PA4 ){
   ASSIGN_4DUT(  dp1,  a_1,  b_1, a_31, b_31 )
   ASSIGN_4DUT(  dp2,  a_2,  b_2, a_32, b_32 )
   ASSIGN_4DUT(  dp3,  a_3,  b_3, a_33, b_33 )
   ASSIGN_4DUT(  dp4,  a_4,  b_4, a_34, b_34 )
```

```
   ASSIGN_4DUT(  dp5,  a_5,  b_5, a_35, b_35 )
   ASSIGN_4DUT(  dp6,  a_6,  b_6, a_36, b_36 )
   ASSIGN_4DUT(  dp7,  a_7,  b_7, a_37, b_37 )
   ASSIGN_4DUT(  dp8,  a_8,  b_8, a_38, b_38 )
   ASSIGN_4DUT(  dp9,  a_9,  b_9, a_39, b_39 )
   ASSIGN_4DUT( dp10, a_10, b_10, a_40, b_40 )
   ASSIGN_4DUT( dp11, a_11, b_11, a_41, b_41 )
   ASSIGN_4DUT( dp12, a_12, b_12, a_42, b_42 )
   ASSIGN_4DUT( dp13, a_13, b_13, a_43, b_43 )
   ASSIGN_4DUT( dp14, a_14, b_14, a_44, b_44 )
   ASSIGN_4DUT( dp15, a_15, b_15, a_45, b_45 )
   ASSIGN_4DUT( dp16, a_16, b_16, a_46, b_46 )
   ASSIGN_4DUT( dp17, a_17, b_17, a_47, b_47 )
   ASSIGN_4DUT( dp18, a_18, b_18, a_48, b_48 )
   ASSIGN_4DUT( dp19, a_19, b_19, a_49, b_49 )
   ASSIGN_4DUT( dp20, a_20, b_20, a_50, b_50 )
   ASSIGN_4DUT( dp21, a_21, b_21, a_51, b_51 )
   ASSIGN_4DUT( dp22, a_22, b_22, a_52, b_52 )
   ASSIGN_4DUT( dp23, a_23, b_23, a_53, b_53 )
   ASSIGN_4DUT( dp24, a_24, b_24, a_54, b_54 )
   ASSIGN_4DUT( dp25, a_25, b_25, a_55, b_55 )
   ASSIGN_4DUT( dp26, a_26, b_26, a_56, b_56 )
   ASSIGN_4DUT( dp27, a_27, b_27, a_57, b_57 )
   ASSIGN_4DUT( dp28, a_28, b_28, a_58, b_58 )
   ASSIGN_4DUT( dp29, a_29, b_29, a_59, b_59 )
   ASSIGN_4DUT( dp30, a_30, b_30, a_60, b_60 )
}
%% VECDEF dp30, dp29, dp28, dp27, dp26, dp25, dp24, dp23,
          dp22, dp21, dp20, dp19, dp18, dp17, dp16, dp15,
          dp14, dp13, dp12, dp11, dp10, dp9, dp8, dp7,
          dp6, dp5, dp4, dp3, dp2, dp1

PATTERN( LogicPat22, logic )
% VEC 1111111111 1111111111 1111111111, TSET1, PS2
// Etc...
```

4.14.4.1 VECDEF per Pin Assignment Table

See VECDEF Compiler Directive.

Description

The VECDEF Compiler Directive optionally allows the specification of which Pin Assignment Table is associated with a given VECDEF directive. Among other uses, this feature facilitates using a single test program to test devices manufactured with different package types, containing a different number of signal pins.

Compile-time error-checking is performed to ensure that DUT pin vs. pattern data columns of each test vector are correctly defined.

- Each column of pattern data must be mapped to at least one DUT pin listed in the specified Pin Assignment Table. Or, to cause a given pattern data column to be ignored, map that column to `t_na`, as shown in the `one_pin` example below.
- Each DUT pin in the VECDEF Compiler Directive must be mapped to a column of pattern data. As shown in the `four_pins` example below it is possible to assign a given column of pattern data to more than one DUT pin.
- It is legal to list multiple pins from a given pin assignments table for a particular `VECDEF` statement (even though the example below doesn't use this feature).
- Every DUT pin name which appears in a VECDEF ***must*** also be mapped to a `t_lvm` resource in ***at least*** one `PIN_SCRAMBLE()` macro. The pattern compiler checks this. This is enforced because a DUT pin used in a `VECDEF` statement which is not mapped to a `t_lvm` resource can not use logic pattern data.

Usage

The VECDEF Compiler Directive syntax optionally supports using curly-bracket-enclosed specification of a pin assignments table, optionally for each VECDEF statement.

```
%% VECDEF d1, d0 { PinAssignTable1 }
```

Multiple pin assignment tables can be specified using a comma-separated list within the curly-brackets. For example:

```
%% VECDEF d1, d0 { PinAssignTable1, PinAssignmentTable2 }
```

The pattern compiler will only apply the `VECDEF` statement to those Pin Assignment Tables that have been listed. If no pin assignments tables are listed, then the behavior is exactly as described in VECDEF Compiler Directive.

4.14.5 VEC Pattern Instruction

See Logic Test Patterns, Logic Vector Syntax, Logic Vector Bit Codes, VECDEF Compiler Directive.

Description

The VEC pattern instruction is used to define a single logic vector. See Logic Vector Syntax, Logic Vector Bit Codes.

Usage

```
% VEC bit-pattern [, optional parameter(s)]
```

where:

% VEC is a pattern instruction for specifying a logic vector. See Logic Vector Syntax.

bit-pattern is a set of Logic Vector Bit Codes (0, 1, H, L, V, Z or X) that determine pattern data, I/O control, and strobe enable for each tester channel in the current instruction. See VECDEF Compiler Directive for details regarding how the order of bit-pattern tokens is aligned with tester channel numbering.

Note: it is illegal to strobe in the last vector of a logic pattern; i.e. the last vector must not contain any Z, V, H or L tokens.

optional parameter(s) may be used to explicity select a time-set, VIHH Map and Pin Scramble Map for the current instruction. Using Magnum 1/2/2x, additional optional parameters are available. See Optional VEC/RPT Instruction Parameters.

Example

```
% VEC  00001111  HHHHLLLL
% VEC  11110000  LLLLHHHH, TSET3, PS2
```

The first vector will execute for one cycle, then increment to the next test vector. Logic-0 will be driven to the tester pins associated with the first four vector bits and logic-1 will be driven to the tester pins associated with the next four vector bits. The tester pins associated with vector bits 9 through 12 will be strobed for logic-1 (H) and the tester pins associated with vector bits 13 through 16 will be strobed for logic-0 (L). See VECDEF Compiler Directive for details regarding how the order of bit-pattern tokens is aligned with tester channel numbering..

4.14.6 RPT Pattern Instruction

See Logic Test Patterns, Logic Vector Syntax, Logic Vector Bit Codes, VECDEF Compiler Directive.

Description

The RPT pattern instruction is used to define a logic instruction that executes multiple times as specified by a repeat count. Note the following:

- It is illegal to use an explicit Pattern Label on an instruction containing RPT. The pattern compiler automatically adds an implicit Pattern Label to these instructions.
- A RPT may not be used in the first instruction of a pattern.
- Rules also exist for using RPT at the end of a pattern (more below).
- The INTEN and INTENADR interrupt instructions are not allowed in RPT instructions. If timer operations are required, use the MAR instruction with the appropriate MAR Branch Condition Operands which are conditional on timer state (zero or non-zero).

The following operation applies:

- RPT may not be used in the last instruction (vector) of the pattern i.e. with VAR DONE.
- RPT cannot be used in the first instruction after a STARTLOOP or in the instruction just prior to an ENDLOOP. Both STARTLOOP and ENDLOOP use the VUDATA field to store a logic vector address (VAR) value, which conflicts with using the VUDATA to store the RPT count.
- The pattern compiler places the RPT *value* (-1) in the VUDATA field of the instruction thus the VUDATA field cannot be explicitly used in an instruction containing RPT (more below). During pattern execution, the VUDATA value is used to initialize VAR counter #1, which is used as a loop control counter.
- The pattern compiler sets the VAR instruction for a RPT instruction to CJMPNZ to the implicit label added by the compiler to the RPT instruction. Then, during pattern execution, when the counter is >0 execution jumps to this label, repeating the instruction. When the counter reaches 0, pattern execution continues to the next vector. Therefore...

Note: it is not legal to specify an explict VAR execution control operand (INC, GOSUB, CJMPNZ, etc.) in a pattern instruction which contains a RPT.

Usage

```
% RPT n bit-pattern [, optional parameter(s)]
```

where:

% RPT is a pattern instruction for specifying repeat logic vector. See Logic Vector Syntax.

n specifies the number of times the instruction is repeated. Repeat counts may be between 2 and 2^{32}-1 (4,294,967,295 decimal or 0xFFFFFFFF hex).

bit-pattern is a set of Logic Vector Bit Codes (0, 1, H, L, V, Z or X) that determine pattern data, I/O control, and strobe enable for each tester channel in the current instruction. See VECDEF Compiler Directive for details regarding how the order of bit-pattern tokens is aligned with tester channel numbering.

optional parameter(s) may be used to explicity select a time-set, VIHH Map and Pin Scramble Map for the current instruction. Using Magnum 1/2/2x, additional optional parameters are available. See Optional VEC/RPT Instruction Parameters.

Example

In the following example, the RPT vector will execute 32 times, then execution will increment to the next instruction:

```
% VEC       11110000  LLLLHHHH
% RPT 32    00001111  HHHHLLLL // Repeat this 32 times
% VEC       11110000  LLLLHHHH
```

4.14.7 Optional VEC/RPT Instruction Parameters

See Logic Test Patterns, Logic Vector Syntax, Logic Vector Bit Codes, VECDEF Compiler Directive.

Each VEC and RPT logic pattern instruction may optionally specify any or all of the following parameters. Usage rules follow the table:

Table 4.14.7.0-1 Optional Logic Vector Parameters

Parameter	Description
PS#	Where # is from 1 to 64. Selects the Pin Scramble Map to be used during the instruction. The Pin Scramble Map determines which data source is mapped to each timing channel. Default = PS1. In Mixed Memory/Logic Patterns, VEC PS# must be further enabled using PINFUNC VPS.
VIHH#	Where # is from 1 to 64. Selects the VIHH Map to be applied during the instruction. Selecting a non-default VIHH Map specifies which tester pin(s) are switched to the VIHH voltage in the current cycle. The default VIHH Map = VIHH1, which is defined by the system software to switch no pins to the VIHH level. VIHH1 cannot be modified. In Mixed Memory/Logic Patterns, VEC VIHH# must be further using PINFUNC VVIHH.
TS#	Where # is from 1 to 32. Selects the Time-sets (TSET) to be used during the current instruction. Default = TSET1. In Mixed Memory/ Logic Patterns, VEC TS# must be further enabled using PINFUNC VTSET.
VCOMP	During Dynamic DC Tests, sends a one trigger to the or . Desired operation also requires specifying the CompCond argument to the test function (ac_test_supply(), hv_ac_test_supply(), ac_partest()). In Magnum 1/2/2x Mixed Memory/Logic Patterns, VAR VCOMP must be further enabled using PINFUNC VVCOMP. Note: RESET (including MAR RESET, VEC RESET, VAR RESET and VPINFUNC RESET) must NOT be used in the same instruction OR the instruction following that using MAR VCOMP, VAR VCOMP, VEC VCOMP, or VPINFUNC VCOMP. Note: the VCOMP operand may be used in the MAR, VEC/RPT, VAR and VPINFUNC instruction.

Table 4.14.7.0-1 Optional Logic Vector Parameters *(Continued)*

Parameter	Description
RESET	Clears the pin electronics error flags and DC Error Flags used by dynamic PMU and DPS tests. See Error Flag vs. Error Latch and Dynamic DC Tests. In Mixed Memory/Logic Patterns VEC RESET must be further enabled using PINFUNC VLATCHRESET. Note: RESET (MAR RESET, VEC/RPT RESET, VAR RESET and VPINFUNC RESET) must NOT be used in the same instruction OR the instruction following that using MAR VCOMP, VAR VCOMP, VEC VCOMP, or VPINFUNC VCOMP. Note: the RESET operand may be used in the MAR, VEC/RPT, VAR VPINFUNC and CHIPS instruction.
LATCH	Complement of NOLATCH (next). LATCH is the default i.e. NOLATCH must be explicitly specified to over-ride LATCH. In Mixed Memory/ Logic Patterns VEC LATCH must be further enabled using PINFUNC VLATCHRESET. Note: the LATCH operand may be used in the MAR, VEC/RPT, VAR and VPINFUNC instructions.

Table 4.14.7.0-1 Optional Logic Vector Parameters *(Continued)*

Parameter	Description
NOLATCH	Complement of LATCH (default, above). Any failing strobe(s) generated in a cycle which contains an explicit MAR NOLATCH or an explicit VAR NOLATCH will not set the corresponding PE error latch(s). See Error Flag vs. Error Latch. Pattern cycles which include NOLATCH will not capture any errors into the ECR. NOLATCH has no effect on the error flag, which controls test pattern branch-on-error operations. For normal PASS/FAIL testing, the NOLATCH operand is not used, allowing any failing strobes to set the error latches and cause the test to fail. In Mixed Memory/Logic Patterns VEC NOLATCH must be further enabled using PINFUNC VLATCHRESET. Note: the NOLATCH operand may be used in the MAR, VEC/RPT, VAR and VPINFUNC instruction.
OVER	See Over-programming Controls and Parallel Test. This operand enables special PE circuitry to inhibit programming stimulus on DUT(s) that have successfully programmed while allowing other DUTs on the same test site to continue programming. In other words, it prevents over-programming. The over_inhibit() function is used to select the programming mechanism that is disabled when the OVER operand is specified. In Mixed Memory/Logic Patterns VEC OVER must be furtherenabled using PINFUNC VOVER. Note: the OVER operand may be used in the MAR, VEC/RPT, VAR and VPINFUNC instruction.

Table 4.14.7.0-1 Optional Logic Vector Parameters *(Continued)*

Parameter	Description
VPULSE	Causes DUT power supplies which have been enabled (see VPulse Function) to switch to the secondary (VPulse) level , set using `dps_vpulse()`. The test pattern must execute multiple instructions each containing VAR VPULSE to allow time for the voltage to stabilize at the DUT. In Mixed Memory/Logic Patterns, VPINFUNC VPULSE must be further enabled using `PINFUNC VVPULSE`. If pattern execution ends on an instruction containing VPULSE the secondary (VPulse) level remains enabled in hardware. Note: the OVER operand may be used in the MAR, VEC/RPT, VAR and VPINFUNC instruction.

Optional parameters must immediately follow the vector bit-pattern field in a VEC or RPT instruction and are comma delimited. For example:

```
% VEC 110011 LLHHLL, TSET2, PS3, NOLATCH
```

As indicated in the table above, the VCOMP, RESET, LATCH, NOLATCH, OVER and VPULSE parameters can be added to a given pattern instruction using several methods:

- In Memory Test Patterns and Mixed Memory/Logic Patterns, as operands to the MAR instruction (VCOMP, RESET, LATCH, NOLATCH, OVER) and PINFUNC instruction (VPULSE).
- In Logic Test Patterns and Mixed Memory/Logic Patterns, as optional parameters to the VEC and RPT instructions.
- In Logic Test Patterns and Mixed Memory/Logic Patterns, as operands to the VAR and VPINFUNC pattern instruction.

The following rules apply:

- In Magnum 1/2/2x patterns it is possible to specify the optional parameters in both the memory pattern instructions (MAR and PINFUNC) and in the logic pattern instructions (VEC/RPT, VAR, VPINFUNC). This is because the Magnum 1/2/2x has independent pattern execution control engine hardware for the memory pattern (MAR Engine) and the logic pattern (VAR Engine). For example:

```
PATTERN( myPat )
% VEC        HL10X, TSET2, PS2
  VPINFUNC   NOLATCH
```

```
VAR         VIHH4
PINFUNC     VPULSE
MAR         VCOMP, TSET4, PS3
```

However, the optional parameters specified for the VAR Engine (VEC/RPT, VAR, VPINFUNC) have no effect unless further enabled using the appropriate PINFUNC operand (VTSET, VPS, VVIHH, VVPULSE, VVCOMP, VLATCHRESET, VOVER). In the example above, since none of these are specified, the VPINFUNC and VAR instructions have no effect, and time-set (TSET4) and pin scramble (PS3) are selected by the MAR instruction, not the VEC instruction.

4.14.8 STARTLOOP / ENDLOOP Logic Vector Instructions

See Logic Test Patterns, Logic Vector Syntax.

The STARTLOOP and ENDLOOP instructions are not vectors, rather they are compiler directives used to flag the start and end of a multi-vector loop.

Note the following:

- STARTLOOP cannot be used before the first instruction in a pattern.
- ENDLOOP cannot be used before the last instruction in a pattern.
- The pattern instruction immediately following STARTLOOP and immediately preceding ENDLOOP must be VEC.
- The pattern compiler adds an implicit label to the first instruction after each STARTLOOP token, thus the user's pattern may not include an explicit label on these instructions. This label is used as noted below.
- In the pattern instruction immediately preceding ENDLOOP the user's pattern instruction may not specify a VAR instruction operand which controls pattern execution sequence i.e. VAR CJMPNE, VAR CJMPNZ, etc. are illegal (more below).
- the first vector immediately following a STARTLOOP token and the vector immediately preceding an ENDLOOP token must not use any other instruction which contains VUDATA operations i.e. COUNTVUDATA.
- STARTLOOP / ENDLOOP nesting to 3 levels is supported. However, pattern subroutines are not aware of external VAR counter use, which means that a subroutine can use a STARTLOOP/ENDLOOP structure but the subroutine must not be called from within another STARTLOOP/ENDLOOP structure.

- Timer Interrupts (using INTEN or INTENADR) are not allowed between STARTLOOP and ENDLOOP tokens.
- A STARTLOOP cannot immediately follow an ENDLOOP from a previous loop.
- A VAR DONE cannot immediately follow an ENDLOOP token.

the STARTLOOP count value is used, via the VUDATA field of the instruction immediately preceding the ENDLOOP token, to initialize one of the VAR Engine counters (COUNT 1 to COUNT 4) used to control the loop execution.

The hardware loop counter is used (checked) in the instruction immediately preceding the ENDLOOP token. The pattern compiler implicitly adds a VCOUNT instruction to decrement the counter and adds VAR CJMPNZ to branch to the implicit label added by the STARTLOOP instruction.

During execution, when the counter value in the instruction immediately preceding the ENDLOOP token is >0 pattern execution will jump back to the implicit label inserted by compiler i.e. the instruction immediately following the STARTLOOP token. When the counter reaches 0, the vector after the ENDLOOP token executes next.

Usage

```
STARTLOOP count
% VEC   HL10X
// ... Other logic instructions
% VEC   HL10X
ENDLOOP
```

where:

STARTLOOP identifies the start of a multi-vector loop.

count specifies the number of times the loop is repeated. The legal loop count values range from 2 to 2^{32} -1 (4,294,967,295 decimal, or 0xFFFFFFFF hex).

ENDLOOP marks the end of the multi-vector loop.

Example

In the following example, the loop repeats 55 times and contains a single vector repeat which repeats 75 times. This generates 4290 pattern cycles

```
% VEC     00000000 XXXXXXXX // Or RPT
STARTLOOP 55
```

```
% VEC      00000000 XXXXXXXX   // First vector in loop
% VEC      00000000 HHHHLLLL
% RPT 75   00001111 HHHHLLLL
% VEC      00000000 11111111   // Last vector in loop
ENDLOOP
% VEC      11110000 LHLHLHLH   // Or RPT
```

4.14.9 VAR Instruction

See Magnum 1/2/2x Logic Vector Instructions.

The term *VAR* refers to the APG's Vector Address Register. In this documentation, the term VAR is also used:

- When referring to the Magnum 1/2/2x hardware engine which controls Logic Test Pattern execution; i.e. VAR Engine. This only applies to patterns which contain logic instructions.
- To represent the VAR instruction which controls logic pattern execution sequence; i.e. VAR instruction.
- The value in the APG VAR register. This is the address of the logic pattern instruction being executed. Note that the user rarely needs to deal with literal VAR values; all references to specific pattern instructions is done using a Pattern Label or pattern name.

The purposes of the VAR instruction are:

- Control Logic Test Pattern instruction execution sequence (`Branch-condition, Address`)
- Specify an interrupt address used in conjunction with the APG's programmable interrupt timer (`Interrupt`)
- Control miscellaneous features (`Error-control`, `Misc`, more below)

The VAR instruction takes the following form:

```
VAR  Branch-condition, Address, Interrupt, Error-control, Misc
```

where:

Branch-condition specifies how the next logic pattern instruction to be executed will be determined. This can be as simple as incrementing the VAR address to the next instruction (VAR INC), or a conditional or unconditional branch to an arbitrary instruction (VAR CJMPNZ, etc.), or a conditional or unconditional subroutine call, or return (VAR CRETE, etc.). Conditional operations can be based on a VAR counter value (VCOUNT), or the PASS/FAIL status of the error flag. See VAR Branch-condition Operands.

Address specifies the instruction to be executed when a conditional operation is specified or a subroutine is called. When the condition is FALSE the next instruction will execute (i.e. VAR INC). An address is specified using a Pattern Label or the name of another test pattern. See VAR Address Operand

The **Interrupt** operand is not used on Magnum 1/2/2x.

Error-control controls several unrelated features. See VAR Error-control Operands:

- RESET = reset the APG error flag (see Error Flag vs. Error Latch).
- LATCH, NOLATCH = allow any failing strobe(s) in the current cycle to set the PE error latch (LATCH, default) or not (NOLATCH). See Error Flag vs. Error Latch). The use of LATCH and NOLATCH are mutually exclusive in a given pattern instruction.

Misc has two options, see VAR Misc Operands:

- MCNTR is used to specify that a VAR Engine conditional branch decision is to be based on the value in one of the 60 MAR engine counters instead of using one of the 4 VAR engine counters. This allows logic vector execution control to branch based on a specified MAR counter value.
- OVER = enable the over-programming logic in the current instruction. See Over-programming Controls and Parallel Test.
- VCOMP = generate a trigger to strobe the DC Comparators and Error LogicDC Comparators and Error Logic in the DC Test and Measure System. See DPS Dynamic Current Test Functions and PMU Dynamic Test Functions.
- VPULSE causes DUT power supplies which have been enabled (see VPulse Function) to switch to the secondary (VPulse) level , set using dps_vpulse().

The table below summarizes the available operands for the VAR instruction. Default values are indicated using (**D**):

Table 4.14.9.0-1 VAR Instruction Operands

Branch Condition	Address	Interrupt	Error Control	Misc
CJMPA CJMPE CJMPNA CJMPNE CJMPNT CJMPNZ CJMPT CJMPZ CRETE CRETNE CRETNT CRETNZ CRETT CRETZ CSUBE CSUBNE CSUBNT CSUBNZ CSUBT CSUBZ DONE GOSUB INC(**D**) JUMP PAUSE RETURN	Pattern Label (none)(**D**)	(none)(**D**) This operand is not used on Magnum 1 Magnum 2 Magnum 2x	(none)(**D**) LATCH NOLATCH RESET Note: the default (none) selects the LATCH option.	MCNTR OVER VCOMP VPULSE DEFAULT[1] (none)(**D**)
Note-1: Usable in mixedsync test patterns only. See Mixed Memory/Logic Patterns.				

4.14.9.1 VAR Branch-condition Operands

See Magnum 1/2/2x Logic Vector Instructions, VAR Instruction.

All VAR branch operands are used to control pattern execution sequence. The VAR instruction takes the following form:

```
VAR  Branch-condition, Address, Interrupt, Error-control, Misc
```

There are three types of VAR branch condition operands:

- VAR Unconditional Branch-condition Operands the next instruction to be executed, regardless of any conditions. These include `VAR JUMP`, `INC`, `DONE`, `GOSUB`, `RETURN`, `PAUSE`.
- VAR Conditional Branch-condition Operands identify what is tested to determine execution flow, and the address of the instruction to execute if the condition is TRUE. Conditional operations can test a VAR Engine counter value, the state of the error flag (see Error Flag vs. Error Latch), or whether the APG interrupt timer has counted to 0. See APG Instruction Execution and VCOUNT Counter Operands.
- VAR Multi-DUT Branch-condition Operands are used in Multi-DUT Test Programs to branch based on DUT-related test results. Note that these options are only usable in Mixed Memory/Logic Patterns.

A branch address is specified using a Pattern Label or the name of another test pattern. VAR `INC`, `DONE`, `RETURN`, and `PAUSE` do not require a branch address.

Proper branch-on-error and branch-on-abort operations require that the test pattern comply with the Error Pipeline Requirements.

Note: using Magnum 1/2/2x, proper operation of the Abort signal requires that all DUTs be enabled; i.e. all DUTs must be in the Active DUTs Set (ADS). The Abort signal will NOT go active if any DUT(s) are disabled.

Note: a subroutine may be identified using a Pattern Label if the subroutine is within the same pattern, or by referring to another PATTERN. The destination of a test pattern jump/branch instruction can only reference a Pattern Label within the same pattern.

The tables below describe the options available for the `Branch-condition` operands to the VAR instruction. Detailed descriptions of each branch option are included:

Table 4.14.9.1-1 VAR Unconditional Branch-condition Operands

Operand	Purpose
DONE	Halts the VAR Engine. This sends a done signal to the site controller (default)
GOSUB	Call the subroutine at the specified address. See Pattern Subroutines and Note:.
INC	Execution proceeds to the next instruction. Any specified address (label) is ignored. This is the default operand.
JUMP	Jump (branch) to the specified address. See Note:.
PAUSE	stops the VAR Engine (pauses the logic pattern) when the instruction containing the PAUSE reaches the DUT. The pattern can be restarted using the `restart()` function. The restarted pattern will continue execution at the instruction immediately following the instruction containing the PAUSE. Restriction: two consecutive pattern instructions containing PAUSE are not allowed. Note: the pattern generator uses a pipe-line architecture. This means that during the time a test pattern is paused that user-code must not: • Write any APG hardware registers, including Address Generator, Data Generator, Chip-selects, Counters, JAM register, User RAM, UDATA, etc. Register read is OK. • Modify any cycle period values. The details of the pattern generator pipe-line architecture are not documented, because: • The architecture is variable, based on which hardware options are used in any given test program and/or test pattern. • The architecture is subject to change without notice, as needed to add new features, fix problems, etc.
RETURN	Return from a subroutine. See Pattern Subroutines

Table 4.14.9.1-2 VAR Conditional Branch-condition Operands

Operand	Purpose
CJMPA	Conditional JuMP on Abort Execution will jump to the specified address (see Note:) if the Abort signal is TRUE at the start of the current instruction. The Abort signal comes from the PE error latches, not error flags, see Error Flag vs. Error Latch. If the Abort signal is FALSE, the next instruction will execute. The Abort signal will be TRUE when all DUTs in a site have one or more set error *latches.* See Error Pipeline Requirements. The Abort signal is useful when testing multiple DUTs; the Abort signal will be TRUE when all DUTs have failed. Using Magnum 1/2/2x, the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and which MAR Error-choice Operands are used in the same instruction. See APG Instruction Execution. Note: using Magnum 1/2/2x, proper operation of the Abort signal requires that all DUTs be enabled; i.e. all DUTs must be in the Active DUTs Set (ADS). The Abort signal will NOT go active if any DUT(s) are disabled.

Table 4.14.9.1-2 VAR Conditional Branch-condition Operands *(Continued)*

Operand	Purpose
CJMPNA	Conditional JuMP on Not Abort Execution will jump to the specified address (see Note:) if the Abort signal is FALSE at the start of the current instruction. The Abort signal comes from the PE error latches, not error flags, see Error Flag vs. Error Latch. If the Abort signal is TRUE, the next instruction will execute. The Abort signal will be FALSE when any one or more DUT(s) in a site do not have a set error *latch*. See Error Pipeline Requirements. The Abort signal is useful when testing multiple DUTs; the Abort signal will be TRUE when all DUTs have failed. Using Magnum 1/2/2x, the operation of this operand is affected/ changed by a static setup (see Static Error Choice Functions, Branch-on-error) and which MAR Error-choice Operands are used in the same instruction. See APG Instruction Execution. Note: using Magnum 1/2/2x, proper operation of the Abort signal requires that all DUTs be enabled; i.e. all DUTs must be in the Active DUTs Set (ADS). The Abort signal will NOT go active if any DUT(s) are disabled.
CJMPE	Conditional JuMP on Error Execution will jump to the specified address (see Note:) if the error signal is TRUE at the start of the current instruction. If the error signal is FALSE, the next instruction will execute. The Error signal will be TRUE if any error flag(s) are set (see Error Flag vs. Error Latch). See Error Pipeline Requirements. Using Magnum 1/2/2x, the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and which MAR Error-choice Operands are used in the same instruction. See APG Instruction Execution.

Table 4.14.9.1-2 VAR Conditional Branch-condition Operands *(Continued)*

Operand	Purpose
CJMPNE	Conditional JuMP on No Error Execution will jump to the specified address (see Note:) if the Error signal is FALSE at the start of the current instruction. If the Error signal is TRUE, the next instruction will execute. The Error signal will be FALSE when no error flag(s) are set (see Error Flag vs. Error Latch). See Error Pipeline Requirements. Using Magnum 1/2/2x, the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and which MAR Error-choice Operands are used in the same instruction. See APG Instruction Execution.
CJMPT	Conditional JuMP on Timer = 0 Execution will jump to the specified address (see Note:) if the APG Interrupt Timer = 0 at the start of the current instruction. If the interrupt timer is not zero, the next instruction will execute. See Note:. See APG Instruction Execution.
CJMPNT	Conditional JuMP on Timer Not = 0 Execution will jump to the specified address (see Note:) if the APG Interrupt Timer is not zero at the start of the current instruction. If the interrupt timer is zero, the next instruction will execute. See Note:. See APG Instruction Execution.
CJMPZ	Conditional JuMP on Zero Execution will jump to the specified address (see Note:) if the counter specified in the VCOUNT instruction is zero at the start of the current instruction. If the counter is not zero the next pattern instruction will execute. See APG Instruction Execution.
CJMPNZ	Conditional JuMP on Not Zero Execution will jump to the specified address (see Note:) if the counter specified in the VCOUNT instruction is not zero at the start of the current instruction. If the counter is zero the next instruction will execute. See APG Instruction Execution.

Table 4.14.9.1-2 VAR Conditional Branch-condition Operands *(Continued)*

Operand	Purpose
CRETE	Conditional RETurn on Error Execution returns to the instruction address popped off the stack if the error signal is TRUE at the start of the current instruction. If the Error signal is FALSE, the next instruction will execute. The Error signal will be TRUE if any error flag(s) are set (see Error Flag vs. Error Latch). See Error Pipeline Requirements and Pattern Subroutines. Using Magnum 1/2/2x, the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and which MAR Error-choice Operands are used in the same instruction. See APG Instruction Execution.
CRETNE	Conditional RETurn on No Error Execution returns to the instruction address popped off the stack if the error signal is FALSE at the start of the current instruction. If the Error signal is TRUE, the next instruction will execute. The Error signal will be FALSE when no error flag(s) are set (see Error Flag vs. Error Latch). See Error Pipeline Requirements and Pattern Subroutines. Using Magnum 1/2/2x, the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and which MAR Error-choice Operands are used in the same instruction. See APG Instruction Execution.
CRETT	Conditional RETurn on Timer = 0 Execution returns to the instruction address popped off the stack if the APG Interrupt Timer = 0 at the start of the current pattern instruction. If the interrupt timer is not zero, the next instruction will execute. See Pattern Subroutines. See Note:. See APG Instruction Execution.
CRETNT	Conditional RETurn on Timer Not = 0 Execution returns to the instruction address popped off the stack if the APG Interrupt Timer is not zero at the start of the current pattern instruction. If the interrupt timer is zero, the next instruction will execute. See Pattern Subroutines. See Note:. See APG Instruction Execution.
CRETZ	Conditional RETurn on Zero Execution returns to the instruction address popped off the stack if the counter specified in the COUNT instruction is zero at the start of the current pattern instruction. If the counter is not zero, the next instruction will execute. See Pattern Subroutines. See APG Instruction Execution.

Table 4.14.9.1-2 VAR Conditional Branch-condition Operands *(Continued)*

Operand	Purpose
CRETNZ	Conditional RETurn on Not Zero Execution returns to the instruction address popped off the stack if the counter specified in the COUNT instruction is not zero at the start of the current pattern instruction. If the counter is zero, the next instruction will execute. See Pattern Subroutines. See APG Instruction Execution.
CSUBE	Conditional SUBroutine call on Error If the error signal is TRUE at the start of the current instruction, calls the specified pattern subroutine (see Note:) and pushes the subroutine return address on the execution stack. If the Error signal is FALSE, the next instruction will execute. The Error signal will be TRUE if any error flag(s) are set (see Error Flag vs. Error Latch). See Error Pipeline Requirements and Pattern Subroutines. Using Magnum 1/2/2x, the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and which MAR Error-choice Operands are used in the same instruction. See APG Instruction Execution.
CSUBNE	Conditional SUBroutine call on No Error If the error signal is FALSE at the start of the current instruction, calls the specified pattern subroutine (see Note:) and pushes the subroutine return address on the execution stack. If the Error signal is TRUE, the next instruction will execute. The Error signal will be FALSE if no error flags are set (see Error Flag vs. Error Latch). See Error Pipeline Requirements and Pattern Subroutines. Using Magnum 1/2/2x, the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and which MAR Error-choice Operands are used in the same instruction. See APG Instruction Execution.
CSUBT	Conditional SUBroutine call on Timer = 0) Calls the specified subroutine (see Note:) if the APG Interrupt Timer = 0 at the start of the current pattern instruction. The subroutine return address is pushed on the execution stack. If the interrupt timer is not 0, the next instruction will execute. See Pattern Subroutines. See Note:. See APG Instruction Execution.

Table 4.14.9.1-2 VAR Conditional Branch-condition Operands *(Continued)*

Operand	Purpose
CSUBNT	Conditional SUBroutine call on Timer Not = 0 Calls the specified subroutine (see Note:) if the APG Interrupt Timer is not 0 at the start of the current pattern instruction. The subroutine return address is pushed on the execution stack. If the timer = 0, the next instruction will execute. See Pattern Subroutines. See Note:. See APG Instruction Execution.
CSUBZ	Conditional SUBroutine call on Zero Calls the specified subroutine (see Note:) if the counter specified in the COUNT instruction is zero at the start of the current pattern instruction. The subroutine return address is pushed on the execution stack. If the counter is not zero, the next instruction will execute. See Pattern Subroutines and APG Instruction Execution.
CSUBNZ	Conditional SUBroutine call on Not Zero Calls the specified subroutine (see Note:) if the counter specified in the COUNT instruction is not zero at the start of the current pattern instruction. The subroutine return address is pushed on the execution stack. If the counter is zero, the next instruction will execute. See Pattern Subroutines and APG Instruction Execution.

The following conditional options are targeted for use in Magnum 1/2/2x Multi-DUT Test Programs:

Table 4.14.9.1-3 VAR Multi-DUT Branch-condition Operands

Operand	Purpose
CSUBE_ALL	Conditional SUBroutine Call Error ALL $(A_1 \cdot A_2 \cdot A_n) \cdot (B_1 \cdot B_2 \cdot B_n)$ The subroutine will be called if every DUT on all sub-sites have an error; i.e. all DUTs fail. See Error Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and and by the MAR Error-choice Operands selection.
Note: A_1 = Sub-site-A DUT #1 has an error $\overline{B_2}$ = Sub-site-B DUT #2 has no errors	

Table 4.14.9.1-3 VAR Multi-DUT Branch-condition Operands *(Continued)*

Operand	Purpose
CSUBNE_ALL	Conditional SUBroutine Call No Error ALL $(\overline{A}_1 + \overline{A}_2 + \overline{A}_n) + (\overline{B}_1 + \overline{B}_2 + \overline{B}_n)$ This is the inverse of CSUBE_ALL. The subroutine will be called if any DUT on any sub-site doesn't have an error; i.e. all DUTs must have an error to NOT branch. See Error Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CSUBE_ANOTB	Conditional SUBroutine Call Error Sub-site-A not Sub-site-B $(A_1 + A_2 + A_n) \bullet (\overline{B}_1 \bullet \overline{B}_2 \bullet \overline{B}_n)$ The subroutine will be called if any DUT on Sub-site-A has an error AND no DUTs on Sub-site-B have an error. See Error Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CSUBNE_ANOTB	Conditional SUBroutine Call No Error Sub-site-A not Sub-site-B $(\overline{A}_1 \bullet \overline{A}_2 \bullet \overline{A}_n) + (B_1 + B_2 + B_n)$ This is the inverse of CSUBE_ANOTB. The subroutine will be called if no DUTs on Sub-site-A have an error OR any DUT on Sub-site-B has an error. See Error Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CSUBE_BNOTA	Conditional SUBroutine Call Error Sub-site-B not Sub-site-A $(\overline{A}_1 \bullet \overline{A}_2 \bullet \overline{A}_n) \bullet (B_1 + B_2 + B_n)$ The subroutine will be called if no DUTs on Sub-site-A have an error AND any DUT on Sub-site-B has an error. See Error Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
Note: A_1 = Sub-site-A DUT #1 has an error $\overline{B_2}$ = Sub-site-B DUT #2 has no errors	

Table 4.14.9.1-3 VAR Multi-DUT Branch-condition Operands *(Continued)*

Operand	Purpose
CSUBNE_BNOTA	Conditional SUBroutine Call No Error Sub-site-B not Sub-site-A $(A_1 + A_2 + A_n) + (\overline{B}_1 \cdot \overline{B}_2 \cdot \overline{B}_n)$ This is the inverse of CSUBE_BNOTA i.e. the subroutine will be called if any DUT on Sub-site-A has an error OR no DUTs on Sub-site-B have an error. See Error Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CSUBE_DUT1 thru CSUBE_DUT8 See Note:	Conditional SUBroutine Call Error DUT-1 thru DUT-8 Subroutine will be called if the specified DUT has an error. Errors associated with other DUTs have no effect. See Error Pipeline Requirements, Pattern Subroutines and Note:. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CSUBNE_DUT1 thru CSUBNE_DUT8 See Note:	Conditional SUBroutine Call No Error DUT-1 thru DUT-8 Subroutine will be called if the specified DUT has no errors. Errors associated with other DUTs have no effect. See Error Pipeline Requirements, Pattern Subroutines and Note:. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CRETE_ALL	Conditional Return Error ALL $(A_1 \cdot A_2 \cdot A_n) \cdot (B_1 \cdot B_2 \cdot B_n)$ The subroutine will return if every DUT on every Sub-site has an error; i.e. all DUTs fail. See Error Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.

Note: A_1 = Sub-site-A DUT #1 has an error
$\overline{B}_2$ = Sub-site-B DUT #2 has no errors

Table 4.14.9.1-3 VAR Multi-DUT Branch-condition Operands *(Continued)*

Operand	Purpose
CRETNE_ALL	Conditional Return No Error ALL $(\overline{A}_1 + \overline{A}_2 + \overline{A}_n) + (\overline{B}_1 + \overline{B}_2 + \overline{B}_n)$ This is the inverse of CRETE_ALL. The subroutine will return if any DUT on any Sub-site doesn't have an error; i.e. all DUTs must have an error to NOT return. See Error Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CRETE_ANOTB	Conditional Return Error Sub-site-A not Sub-site-B $(A_1 + A_2 + A_n) \bullet (\overline{B}_1 \bullet \overline{B}_2 \bullet \overline{B}_n)$ The subroutine will return if any DUT on Sub-site-A has an error AND no DUTs on Sub-site-B have an error. See Error Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CRETNE_ANOTB	Conditional Return No Error Sub-site-A not Sub-site-B $(\overline{A}_1 \bullet \overline{A}_2 \bullet \overline{A}_n) + (B_1 + B_2 + B_n)$ This is the inverse of CRETE_ANOTB i.e. the subroutine will return if no DUTs on Sub-site-A have an error OR any DUT on Sub-site-B has an error. See Error Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CRETE_BNOTA	Conditional Jump Error Sub-site-B not Sub-site-A $(\overline{A}_1 \bullet \overline{A}_2 \bullet \overline{A}_n) \bullet (B_1 + B_2 + B_n)$ The subroutine will return if no DUTs on Sub-site-A have an error AND any DUT on Sub-site-B has an error. See Error Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.

Note: A_1 = Sub-site-A DUT #1 has an error
$\overline{B}_2$ = Sub-site-B DUT #2 has no errors

Table 4.14.9.1-3 VAR Multi-DUT Branch-condition Operands *(Continued)*

Operand	Purpose
CRETNE_BNOTA	Conditional Jump No Error Sub-site-B not Sub-site-A $(A_1 + A_2 + A_n) + (\overline{B}_1 \bullet \overline{B}_2 \bullet \overline{B}_n)$ This is the inverse of CRETE_BNOTA i.e. the subroutine will return if any DUT on Sub-site-A has an error OR no DUTs on Sub-site-B have an error. See Error Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CRETE_DUT1 thru CRETE_DUT8	Conditional Return Error DUT-1 thru DUT-8 Subroutine will return if the specified DUT has an error. Errors associated with other DUTs have no effect. See Error Pipeline Requirements, Pattern Subroutines and Note:. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CRETNE_DUT1 thru CRETNE_DUT8	Conditional Return No Error DUT-1 thru DUT-8 Subroutine will return if the specified DUT has no errors. Errors associated with other DUTs have no effect. See Error Pipeline Requirements, Pattern Subroutines and Note:. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CJMPE_ALL	Conditional Jump Error ALL $(A_1 \bullet A_2 \bullet A_n) \bullet (B_1 \bullet B_2 \bullet B_n)$ The jump will occur if every DUT on every Sub-site has an error; i.e. all DUTs fail. See Error Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.

Note: A_1 = Sub-site-A DUT #1 has an error
$\overline{B_2}$ = Sub-site-B DUT #2 has no errors

Table 4.14.9.1-3 VAR Multi-DUT Branch-condition Operands *(Continued)*

Operand	Purpose
CJMPNE_ALL	Conditional Jump No Error ALL DUTs $(\overline{A}_1 + \overline{A}_2 + \overline{A}_n) + (\overline{B}_1 + \overline{B}_2 + \overline{B}_n)$ This is the inverse of CJMPE_ALL. The jump will occur if any DUT on any Sub-site doesn't have an error; i.e. all DUTs must have an error to NOT branch. See Error Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CJMPE_ANOTB	Conditional Jump Error Sub-site-A not Sub-site-B $(A_1 + A_2 + A_n) \bullet (\overline{B}_1 \bullet \overline{B}_2 \bullet \overline{B}_n)$ The jump will occur if any DUT on Sub-site-A has an error AND no DUTs on Sub-site-B have an error. See Error Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CJMPNE_ANOTB	Conditional Jump No Error Sub-site-A not Sub-site-B $(\overline{A}_1 \bullet \overline{A}_2 \bullet \overline{A}_n) + (B_1 + B_2 + B_n)$ This is the inverse of CJMPE_ANOTB. The jump will occur if no DUTs on Sub-site-A have an error OR any DUT on Sub-site-B has an error. See Error Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CJMPE_BNOTA	Conditional Jump Error Sub-site-B not Sub-site-A $(\overline{A}_1 \bullet \overline{A}_2 \bullet \overline{A}_n) \bullet (B_1 + B_2 + B_n)$ The jump will occur if no DUTs on Sub-site-A have an error AND any DUT on Sub-site-B has an error. See Error Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
Note: A_1 = Sub-site-A DUT #1 has an error $\overline{B_2}$ = Sub-site-B DUT #2 has no errors	

Table 4.14.9.1-3 VAR Multi-DUT Branch-condition Operands *(Continued)*

Operand	Purpose
CJMPNE_BNOTA	Conditional Jump No Error Sub-site-B not Sub-site-A $(A_1 + A_2 + A_n) + (\overline{B}_1 \bullet \overline{B}_2 \bullet \overline{B}_n)$ This is the inverse of CJMPE_BNOTA. The jump will occur if any DUT on Sub-site-A has an error OR no DUTs on Sub-site-B have an error. See Error Pipeline Requirements and Pattern Subroutines. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CJMPE_DUT1 thru CJMPE_DUT8	Conditional Jump Error DUT-1 thru DUT-8 Jump will occur if the specified DUT has an error. Errors associated with other DUTs have no effect. See Error Pipeline Requirements, Pattern Subroutines and Note:. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and by the MAR Error-choice Operands selection.
CJMPNE_DUT1 thru CJMPNE_DUT8 See Note:	Conditional Jump No Error DUT-1 thru DUT-8 Jump will occur if the specified DUT has no errors. Errors associated with other DUTs have no effect. See Error Pipeline Requirements, Pattern Subroutines and Note:. Note: the operation of this operand is affected/changed by a static setup (see Static Error Choice Functions, Branch-on-error) and which MAR Error-choice Operands are used in the same instruction.
Note: A_1 = Sub-site-A DUT #1 has an error $\overline{B}_2$ = Sub-site-B DUT #2 has no errors	

4.14.9.2 VAR Address Operand

See Magnum 1/2/2x Logic Vector Instructions, VAR Instruction.

The VAR instruction takes the following form:

```
VAR  Branch-condition, Address, Interrupt, Error-control, Misc
```

The address specified in a VAR instruction identifies a jump, branch or subroutine target instruction. This is the instruction target for the VAR Branch-condition operand. It is

specified using a Pattern Label. A subroutine address (GOSUB, CSUBNT, etc.) can be specified as a label in the same pattern source file, or a PATTERN name.

The VAR address operand may be omitted for instructions that do not require a target address; i.e. INC, DONE, RETURN, PAUSE. An address must be specified with all conditional and unconditional jump and subroutine operands (CSUBE, CJMPZ, JUMP, GOSUB, etc.).

4.14.9.3 VAR Interrupt Operands

Note: on 8/4/2008 this operand was un-documented for Magnum 1. The related functionality has not been implemented. All APG Interrupt Timer use requires the use of related MAR Interrupt Operands which requires either a Memory Test Pattern or Mixed Memory/Logic Pattern. This operand is also not supported on Magnum 2/2x.

4.14.9.4 VAR Error-control Operands

See Magnum 1/2/2x Logic Vector Instructions, VAR Instruction.

The VAR instruction takes the following form:

```
VAR  Branch-condition, Address, Interrupt, Error-control, Misc
```

The `Error-control` operand is used to control several independent features:

- Error flag operations. This requires understanding the two error signal types generated by the hardware. See Error Flag vs. Error Latch.
- Over-programming control. See Over-programming Controls and Parallel Test.
- Trigger DC comparators, when testing DPS current or PMU voltage/current.

The table below describes the options available for the VAR `Error-control` operands:

Table 4.14.9.4-1 VAR Error-control Operands

Operands	Purpose
RESET	Clears the pin electronics error flags and DC Error Flags used by dynamic PMU and DPS tests. See Error Flag vs. Error Latch and Dynamic DC Tests. In Mixed Memory/Logic Patterns VEC RESET must be further enabled using PINFUNC VLATCHRESET. Note: RESET (including MAR RESET, VEC RESET, VAR RESET and VPINFUNC RESET) must NOT be used in the same instruction OR the instruction following that using MAR VCOMP, VAR VCOMP, VEC VCOMP, or VPINFUNC VCOMP. Note: RESET may be used in the MAR, VEC/RPT, VAR, VPINFUNC and CHIPS instruction.

Table 4.14.9.4-1 VAR Error-control Operands *(Continued)*

Operands	Purpose
LATCH	Complement of NOLATCH. Any failing strobe(s) generated by an instruction which does not contain an explicit MAR NOLATCH or an explicity VAR NOLATCH will set the corresponding PE error latch(s) and cause the test to fail. See Error Flag vs. Error Latch. In Mixed Memory/ Logic Patterns VEC LATCH must be further enabled using PINFUNC VLATCHRESET. Note: LATCH may be used in the MAR, VEC/RPT, VAR and VPINFUNC instruction.
NOLATCH	Complement of LATCH. In memory patterns, used in conjunction with READ and READUDATA. In logic patterns, affects functional strobe operation. Any pattern instruction which includes MAR NOLATCH and/or VAR NOLATCH prevents any failing strobe(s) from setting the a PE error latch(s). It also inhibits capturing errors into the ECR. Has no effect on the error flag, which controls branch-on-error decisions. For normal PASS/FAIL testing, the NOLATCH operand is not used, allowing any failing strobes to set the error latches and cause the test to fail. See Error Flag vs. Error Latch. In Mixed Memory/Logic Patterns, VAR NOLATCH must be further enabled using PINFUNC VLATCHRESET. Note: NOLATCH may be used in the MAR, VEC/RPT, VAR and VPINFUNC instruction.
CLEARERR	Clear the errorline specified by the error-choice operand (see MAR Error-choice Operands)

Multiple operands are allowed in this field simultaneously, however, LATCH and NOLATCH are mutually exclusive and, as noted above, VCOMP and RESET should not be used together.

Example

The following example resets the PE error flags from the VAR Engine:

```
% VEC      1010HLXX
  PINFUNC VLATCHRESET  // Enable RESET from VAR Engine
  VAR     RESET
```

In the following instruction, the DC comparators are enabled for PASS/FAIL testing. Since a loop counter and conditional jump are used, the comparators will be enabled during the entire time in which this instruction is looping:

```
%  Vloop:// Enable DC comparators in this cycle
   VEC     1010HLXX
   PINFUNC VVCOMP    // Enable VCOMP from VAR Engine
   VCOUNT  COUNT1, DECR
   VAR     CJMPNZ, Vloop, VCOMP
```

4.14.9.5 VAR Misc Operands

See Magnum 1/2/2x Logic Vector Instructions, VAR Instruction.

The VAR instruction takes the following form:

```
VAR  Branch-condition, Address, Interrupt, Error-control, Misc
```

The table below describes the options available for the VAR `misc` operands:

Table 4.14.9.5-1 VAR Misc Operands

Operands	Purpose
MCNTR	Used to specify that a VAR Engine conditional branch decision is to be based on the value in one of the 60 MAR counters instead of using one of the 4 VAR counters. This allows logic vector execution control to branch based on a specified MAR counter value. The logic vector VAR Engine cannot control the MAR counters, but does receive a signal when the selected counter reaches a count of zero. Using this feature does consume MAR uRAM, to identify the MAR counter, but the pattern must be a mixed mode pattern for this to be useful anyway.

Table 4.14.9.5-1 VAR Misc Operands *(Continued)*

Operands	Purpose
OVER	See Over-programming Controls and Parallel Test. This operand enables special PE circuitry to inhibit programming stimulus on DUTs that have successfully programmed while allowing other DUTs on the same test site to continue programming. In other words, it prevents over-programming. The `over_inhibit()` function is used to select the programming mechanism that is disabled when the OVER operand is specified. In Mixed Memory/Logic Patterns, VAR OVER must be further enabled using `PINFUNC VOVER`. Note: OVER may be used in the `MAR`, `VEC/RPT`, `VAR` and `VPINFUNC` instruction.
VCOMP	During Dynamic DC Tests, sends a one trigger to the or . Desired operation also requires specifying the `CompCond` argument to the test function (`ac_test_supply()`, `hv_ac_test_supply()`, `ac_partest()`). In Magnum 1/2/2x Mixed Memory/Logic Patterns, VAR VCOMP must be further enabled using `PINFUNC VVCOMP`. Note: RESET (including MAR RESET, VEC RESET, VAR RESET and VPINFUNC RESET) must NOT be used in the same instruction OR the instruction following that using MAR VCOMP, VAR VCOMP, VEC VCOMP, or VPINFUNC VCOMP. Note: VCOMP may be used in the `MAR`, `VEC/RPT`, `VAR` and `VPINFUNC` instruction.

Table 4.14.9.5-1 VAR Misc Operands *(Continued)*

Operands	Purpose
VPULSE	Causes DUT power supplies which have been enabled (see VPulse Function) to switch to the secondary (VPulse) level , set using `dps_vpulse()`. The test pattern must execute multiple instructions each containing VAR VPULSE to allow time for the voltage to stabilize at the DUT. In Mixed Memory/Logic Patterns, VPINFUNC VPULSE must be further enabled using `PINFUNC VVPULSE`. If pattern execution ends on an instruction containing VPULSE the secondary (VPulse) level remains enabled in hardware. Note: the VPULSE operand may be used in the `PINFUNC`, `VEC/RPT`, VAR and `VPINFUNC` instructions.
DEFAULT	Valid only in Mixed Memory/Logic Patterns with the mixedsync attribute (see Pattern Type Attributes). Identifies the current instruction as the default logic instruction to be applied in any subsequent instructions which do not include any explicit memory instructions. See Mixed Memory/Logic Patterns.

Multiple operands are allowed in this field simultaneously, in any order.

Example

In this example, the VAR instruction will jump to `some_label` if the APG MAR counter COUNT2 is not zero. Note that the APG instruction which causes COUNT2 to reach zero may not be in the same pattern source statement as the VAR instruction which evaluates COUNT2 ≠ 0. However, if it is not and auto-reload occurs (AON), this instruction will never jump to the specified label.

```
%  VEC      1010HLXX
   COUNT    COUNT2, DECR, AON// ID MAR Counter. Uses uRAM
   VAR      MCNTR, CJMPNZ, some_label
```

4.14.10 VCOUNT Instruction

See Magnum 1/2/2x Logic Vector Instructions.

Magnum 1/2/2x APG design includes a logic vector execution control engine (VAR Engine),and associated instruction memory (vRAM). This allows Logic Test Pattern execution to be controlled independently (VAR Engine) of memory patterns (MAR Engine). See Magnum 1/2/2x Memory Pattern Instructions.

The VAR Engine has 4 counters, identified as COUNT1 through COUNT4. The VCOUNT instruction is used to explicitly control VAR engine counter operation. Note the following:

- Each VAR engine counter is 32 bits; i.e. 2^{32} = 4,294,967,296 counts.
- The entire VCOUNT instruction is optional. When omitted, no counters are modified and COUNT1 is selected for any VAR conditional branch operations based on a counter value.
- When a VCOUNT instruction is specified exactly one operand must be specified in both the counter and function operands.
- The 4 VAR counters may be used much like the MAR engine counters; as pattern loop control, as execution trace flags, etc. For example:

```
VCOUNT  COUNT1, DECR
VAR     CJMPZ, label // Jump to label if VAR counter COUNT1 = 0
```

Note: when VAR engine counters are explicitly used to control pattern loops the value assigned is the number of desired loop iterations (n) which is different than when using MAR engine counters, which use n-1.

- The VAR counters in Maverick-II and Magnum 1 do not have reload registers. Magnum 2/2x (and the MAR counters) do have reload registers. A reload register may be used to reload its associated counter when that counter = 0. Reload operation is controlled using the VCOUNT Autoreload Operands.
- The 4 VAR counters can be incremented (INCR), decremented (INCR), decremented by 2 (DEC2) or loaded from the VUDATA field (COUNTVUDATA).
- Using Maverick-II and Magnum 1, VAR counters are used implicitly for RPT and STARTLOOP control and thus must not be used explicitly in vectors with those instructions or inside STARTLOOP/ENDLOOP boundries. VAR counter COUNT4 is always used for RPT. VAR counters COUNT1 through COUNT3 are used for STARTLOOP count control, allowing up to three levels of nesting. Magnum 2/2x have seperate counters for RPT/STARTLOOP control plus 4 seperate VAR counters for explicit user applications.
- Pattern Subroutines are *not aware* of external VAR counters use, which means:

- Calling a pattern subroutine from inside a STARTLOOP is OK, but the subroutine must not contain a STARTLOOP and on Maverick-II and Magnum 1 must not explictly use VAR counters.

  Or...

- If the pattern subroutine must contain a STARTLOOP, the subroutine must not be called from inside a STARTLOOP or on Maverick-II and Magnum 1 from a loop explictly controlled using VAR counters.

  In all cases, on Maverick-II and Magnum 1 explicit use of VAR counters must be consistent with how these counters are used implicitly.

- VAR counters can be accessed (set or get) from C-code or pattern initial conditions using vcount(). See previous sentence.
- VAR counters can be set from a test pattern instruction:

```
VCOUNT    COUNT2, COUNTVUDATA
VUDATA    10
```

  causes the value in the VUDATA field (10) to be transferred to the specified VAR counter (COUNT2).

The VCOUNT instruction takes the following form:

```
VCOUNT counter, function, autoreload
```

where:

counter specifies which VAR engine counter is selected in the current instruction.

function the operation to perform on **counter**.

autoreload controls automatic reloading of the counter when that counter = 0. This field is not used on Maverick-II or Magnum 1. See VCOUNT Autoreload Operands.

The table below summarizes the available operands for each VCOUNT instruction option:

Table 4.14.10.0-1 VCOUNT Instruction Operands

Counter	Function	Autoreload
COUNT# NOCOUNT(**D**)	COUNTVUDATA DECR INCR DEC2 NOCOUNT(**D**)	AOFF(**D**) AON Note: this field is not used on Maverick-II or Magnum 1.

Example

In this example VAR counter 1 (COUNT1) will be decremented (DECR) each time this instruction executes. The VAR CJMPNZ instruction causes execution to repeat this instruction (jump to label_X) until COUNT1 decrements to 0 (see Note:):

```
% label_X:
   VCOUNT  COUNT1, INCR
   VAR     CJMPNZ, label_X
```

4.14.10.1 VCOUNT Counter Operands

See Magnum 1/2/2x Logic Vector Instructions, VCOUNT Instruction.

The VCOUNT instruction takes the following form:

```
VCOUNT counter, function, autoreload
```

The table below describes the options available for the counter operand to the COUNT instruction:

Table 4.14.10.1-1 VCOUNT counter Operands

Operand	Purpose
COUNT#	Selects the target counter. This is the counter affected by the specified VCOUNT function and is the counter tested by VAR conditional branch operations which evaluate a counter value.
NOCOUNT	COUNT1 is selected (default).

4.14.10.2 VCOUNT Function Operands

See Magnum 1/2/2x Logic Vector Instructions, VCOUNT Instruction.

The VCOUNT instruction takes the following form:

```
VCOUNT counter, function, autoreload
```

where:

counter is COUNT1 through COUNT4. This selects one VAR engine counter which will be affected by the selected **function**.

function is one of the operand values from the following table:

Table 4.14.10.2-1 VCOUNT function Operands

Operand	Purpose
COUNTVUDATA	Loads the VUDATA value into the counter# specified by the counter operand. See Note:.
DECR	Decrements by 1 the counter# specified by the counter operand
DEC2	Decrements by 2 the counter specified in the counter operand
INCR	Increments by 1 the counter specified in the counter operand
NOCOUNT	No counter function is performed; i.e. counter# holds its value (default). The selected counter# may still be tested by VAR conditional branch operations.

4.14.10.3 VCOUNT Autoreload Operands

See VCOUNT Instruction.

Description

Note: this field is not used on Maverick-II or Magnum 1.

The VCOUNT instruction takes the following form:

```
VCOUNT counter, function, autoreload
```

The autoreload operand is optional and, if specified, determines whether the specified VAR engine counter will be reloaded from its associated reload register.

Each VAR engine counter is backed by a 32 bit reload register. During pattern execution, counter auto-reload occurs when:

- At the start of the current instruction (i.e. before any counter modifications occurs in the current instruction), the value in the counter selected in the current instruction equals 1. And...

- VCOUNT AON is specified in the current instruction.

The table below describes the options available for the autoreload operand to VCOUNT:

Table 4.14.10.3-1 VCOUNT Autoreload Operands

Operand	Purpose
AOFF	Disables Autoreload (default)
AON	Enables Autoreload

Example

The following example decrements VAR engine counter #2 (COUNT2). If, before counter #2 is decremented, its value = 1, counter #2 will be reloaded (AON) from its reload register; i.e. reload register #2:

```
% VCOUNT COUNT2, DECR, AON
```

In the following example, assuming the first instruction causes counter #2 to decrement to 1, counter #2 will be reloaded as indicated. Note that reloading would occur as noted if counter #2 = 1 regardless of how it reached that value:

```
% VCOUNT COUNT2, DECR, AON // Reloaded here if =1 before DECR
% VCOUNT COUNT2, any_function, AOFF// NOT reloaded here
% VCOUNT COUNT2, any_function, AOFF// NOT reloaded here
% VCOUNT COUNT2, any_function, AON // Reload occurs here if =1
```

4.14.11 VPINFUNC Instruction

See Magnum 1/2/2x Logic Vector Instructions.

The Magnum 1/2/2x APG design includes a logic vector execution control engine (VAR Engine) and associated instruction memory (vRAM). This allows logic pattern execution to be controlled independently (VAR Engine) of memory patterns (MAR Engine). See Magnum 1/2/2x Memory Pattern Instructions.

The VPINFUNC instruction is used to control several unrelated options. The table below lists each operand option and describes how the operand is used.

The VPINFUNC instruction takes the form:

```
VPINFUNC  PS#, VIHH#, TSET#, VCOMP, RESET, LATCH, NOLATCH, OVER,
          VPULSE
```

The VPINFUNC operands are optional and may be specified in any order. The entire VPINFUNC instruction is optional if the default values, indicated below, are acceptable

The VPINFUNC operands are described below::

Table 4.14.11.0-1 VPINFUNC Instruction Operands

Operand	Purpose
PS#	Where # is from 1 to 64. Selects the Pin Scramble Table to be used during the instruction. The Pin Scramble Table determines which data source is mapped to each timing channel. Default = PS1. In Mixed Memory/Logic Patterns, VPINFUNC PS# must be further enabled using PINFUNC VPS.
VIHH#	Where # is from 1 to 64. Selects the VIHH Map to be applied during the instruction. A VIHH Map specifies which tester pins are connected to the VIHH voltage, with all other pins driving at the normal VIH and VIL levels. Default = VIHH1, which is defined by the system software to disconnect VIHH from all tester pins. VIHH1 cannot be modified by the user. In Mixed Memory/Logic Patterns, VPINFUNC VIHH# must be further enabled using PINFUNC VVIHH.
TSET#	Where # is from 1 to 32. Selects the Time-sets (TSET) to be used during the current instruction. Default = TSET1. In Mixed Memory/Logic Patterns, VPINFUNC TSET# must be further enabled using PINFUNC VTSET.

Table 4.14.11.0-1 VPINFUNC Instruction Operands *(Continued)*

Operand	Purpose
VCOMP	During Dynamic DC Tests, sends a one trigger to the or . Desired operation also requires specifying the CompCond argument to the test function (ac_test_supply(), hv_ac_test_supply(), ac_partest()). In Magnum 1/2/2x Mixed Memory/Logic Patterns, VAR VCOMP must be further enabled using PINFUNC VVCOMP. Note: RESET (including MAR RESET, VEC RESET, VAR RESET and VPINFUNC RESET) must NOT be used in the same instruction OR the instruction following that using MAR VCOMP, VAR VCOMP, VEC VCOMP, or VPINFUNC VCOMP. Note: VCOMP may be used in the MAR, VEC/RPT, VAR and VPINFUNC instruction.
RESET	Clears the pin electronics error flags and DC Error Flags used by dynamic PMU and DPS tests. See Error Flag vs. Error Latch and Dynamic DC Tests. In Mixed Memory/Logic Patterns VEC RESET must be further enabled using PINFUNC VLATCHRESET. Note: RESET (including MAR RESET, VEC RESET, VAR RESET and VPINFUNC RESET) must NOT be used in the same instruction OR the instruction following that using MAR VCOMP, VAR VCOMP, VEC VCOMP, or VPINFUNC VCOMP. Note: RESET may be used in the MAR, VEC/RPT, VAR, VPINFUNC and CHIPS instruction.

Table 4.14.11.0-1 VPINFUNC Instruction Operands *(Continued)*

Operand	Purpose
LATCH	Complement of NOLATCH. Any failing strobe(s) generated by an instruction which does not contain an explicit MAR NOLATCH or an explicity VAR NOLATCH will set the corresponding PE error latch(s) and cause the test to fail. See Error Flag vs. Error Latch. In Mixed Memory/ Logic Patterns, VPINFUNC LATCH must be further enabled using PINFUNC VLATCHRESET. Note: LATCH may be used in the MAR, VEC/RPT, VAR and VPINFUNC instruction.

Table 4.14.11.0-1 VPINFUNC Instruction Operands *(Continued)*

Operand	Purpose
NOLATCH	Complement of LATCH. In memory patterns, used in conjunction with READ and READUDATA. In logic patterns, affects functional strobe operation. Any pattern instruction which includes MAR NOLATCH and/or VAR NOLATCH prevents any failing strobe(s) from setting the a PE error latch(s). It also inhibits capturing errors into the ECR. Has no effect on the error flag, which controls branch-on-error decisions. For normal PASS/FAIL testing, the NOLATCH operand is not used, allowing any failing strobes to set the error latches and cause the test to fail. See Error Flag vs. Error Latch. In Mixed Memory/Logic Patterns, VPINFUNC NOLATCH must be further enabled using PINFUNC VLATCHRESET. Note: NOLATCH may be used in the MAR, VEC/RPT, VAR and VPINFUNC instruction.
OVER	See Over-programming Controls and Parallel Test. This operand enables special PE circuitry to inhibit programming stimulus on DUTs that have successfully programmed while allowing other DUTs on the same test site to continue programming. In other words, it prevents over-programming. The over_inhibit() function is used to select the programming mechanism that is disabled when the OVER operand is specified. In Mixed Memory/Logic Patterns, VPINFUNC OVER must be further enabled using PINFUNC VOVER. Note: OVER may be used in the MAR, VEC/RPT, VAR and VPINFUNC instruction.

Table 4.14.11.0-1 VPINFUNC Instruction Operands *(Continued)*

Operand	Purpose
VPULSE	Causes DUT power supplies which have been enabled (see VPulse Function) to switch to the secondary (VPulse) level , set using `dps_vpulse()`. The test pattern must execute multiple instructions each containing VAR VPULSE to allow time for the voltage to stabilize at the DUT. In Mixed Memory/Logic Patterns, VPINFUNC VPULSE must be further enabled using `PINFUNC VVPULSE`. If pattern execution ends on an instruction containing VPULSE the secondary (VPulse) level remains enabled in hardware. Note: the VPULSE operand may be used in the `PINFUNC`, `VEC/RPT`, VAR and `VPINFUNC` instructions.

Example

```
% VEC 1010HLXX
   VPINFUNC  PS55, VIHH42, TSET2
```

4.14.12 VUDATA Instruction

See Magnum 1/2/2x Logic Vector Instructions.

Description

The Magnum 1/2/2x APG design includes a logic vector execution control engine (VAR Engine and associated instruction memory (vRAM). This allows logic pattern execution to be controlled independently (VAR Engine) of memory patterns (MAR Engine). See Magnum 1/2/2x Memory Pattern Instructions.

In a Magnum 1/2/2x logic pattern, the VUDATA field is used in 3 contexts, which occur on a per-instruction basis in the test pattern:

1. Explicit user-defined VUDATA value and application. The desired value is specified using the VUDATA statement in the pattern instruction. The application is specified using the `VCOUNT COUNTVUDATA` operand.

2. Implicit pattern compiler defined VUDATA value and application. The user must ***not*** explicitly use VUDATA statements in these pattern instructions. These include:
   - A logic instruction using the RPT opcode. The VUDATA field is implicitly used to set a VAR counter (COUNT4) to the specified repeat value (-1).
   - The vector after a STARTLOOP statement. The VUDATA field is implicitly used to set a VAR Engine counter (counter COUNT1 through COUNT3, depending on STARTLOOP nesting) to the specified repeat value (-1).
   - The vector preceeding an ENDLOOP statement. The VUDATA field is implicitly used to set the address of the first vector in the loop.
   - In a logic pattern, any vector performing a conditional or unconditional subroutine call or branch i.e. VAR GOSUB, VAR CJMPE, VAR JUMP, etc. The VUDATA field is implicitly used to set jump/call address.

4.14.13 Sync Loops

Sync loops are used to synchronize the DUT outputs to a specific sequence of bits in a functional pattern.

This is accomplished by looping on a set of test vectors, clocking the DUT on each pass through the loop, strobing the desired DUT pins, and branching out of the loop if synchronization occurs.

Note that this example works well when testing a single DUT; i.e. in parallel test applications the test pattern will likely be more complex, typically requiring a separate synchronization loop for each DUT being tested.

Example

```
PATTERN( myPat, logic )
% VEC     XXXX XXXX

STARTLOOP 10
% VEC     1111 XXXX      // Clock the DUT
% VEC     0000 HHHH      // Strobe for 4 bits high
  VAR     NOLATCH
% RPT 12  0000 XXXX      // See Error Pipeline Requirements
% VEC     0000 XXXX
  VAR     CJMPNE, SYNC_LABEL     // Jump out of loop if "no error"
```

```
% VEC     0000 XXXX
  VAR     RESET                  // Reset the error flags
ENDLOOP
% VEC     0000 XXXX
% VAR     DONE                     // End here if sync failed
% SYNC_LABEL:             // Arrive here from CJMPNE
  VEC     0000 XXXX       // first vector after successfully
                          // synchronizing
```

In the example above, the synchronization loop will execute a maximum of 10 times (via STARTLOOP 10). If synchronization is not achieved after 10 attempts, the program terminates by dropping through the ENDLOOP after 10 passes and executing the MAR DONE command. The sync loop in this example starts by clocking the DUT once and then strobing four pins for being high. Since the tester is pipelined, a number of cycles must be inserted (the RPT vector plus a normal VEC) to allow the strobe and error signals to propagate through the pipelines so that a jump on no error condition (CJMPNE) can be properly detected. If there is no error (i.e. the four strobed pins were high.), meaning synchronization has been achieved, the program jumps to SYNC_LABEL and continues executing. Note that SYNC_LABEL is inside the percent sign so that it is clearly associated with a particular test vector.

4.15 Scan Test Patterns

See Test Pattern Programming.

The Magnum 1 test system contains two sources of test pattern data:

- Algorithmic Pattern Generator (APG) when executing Memory Test Patterns.
- Combined Logic Vector Memory (LVM) / Scan Vector Memory (SVM) for stored Logic Test Patterns and Scan Test Patterns.

The diagram below shows key Magnum 1 architecture features:

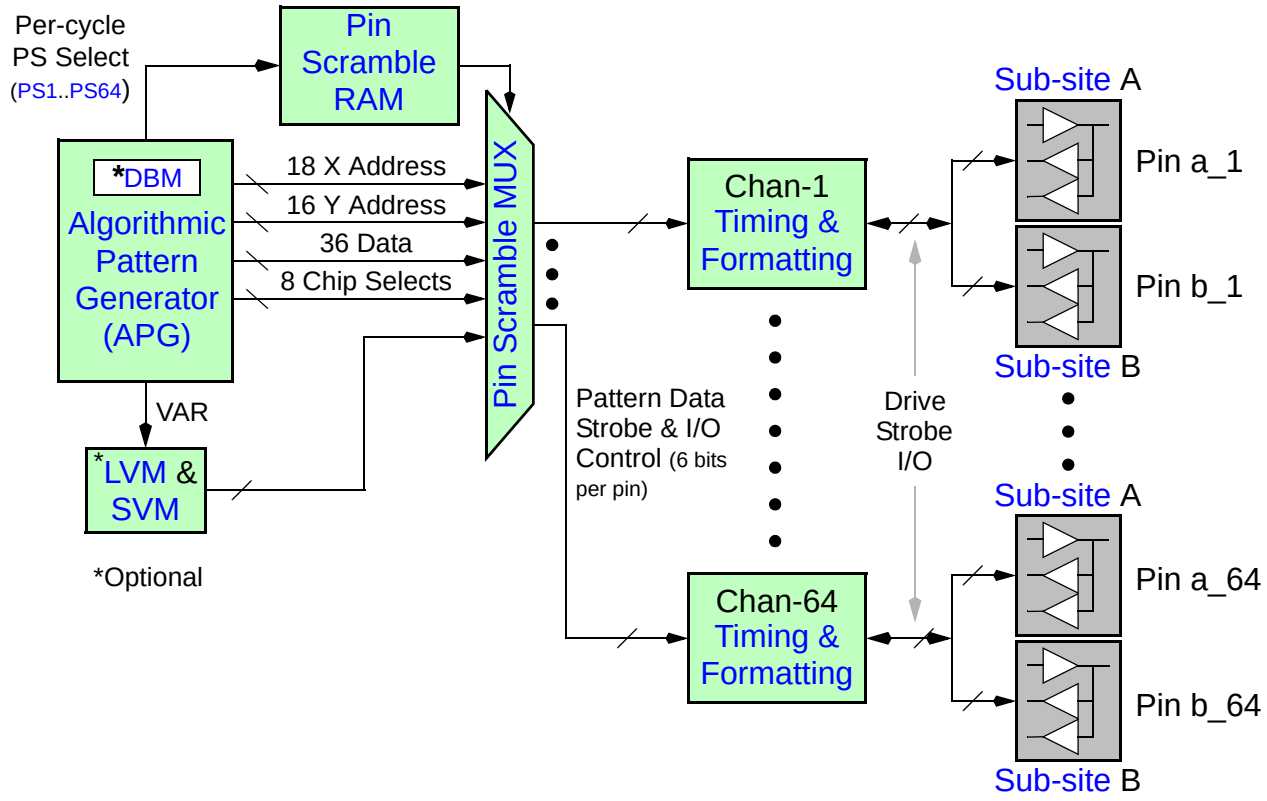


Figure-70: Test Pattern Data Source Hardware Architecture

Using the Pin Scramble MUX, the source of pattern data for each timing channel can be selected on a per-channel/per-cycle basis, at full tester speed. In other words, for any given pin-pair, the source of pattern data can be selected on a per-cycle basis from any APG address, data, or chip select data bit, or one LVM/SVM data bit (2 bits in Double Data Rate (DDR) Mode). This section discusses Memory Test Patterns, also see Logic Test Patterns and Mixed Memory/Logic Patterns.

This section discusses Scan Memory Patterns. also see Memory Test Patterns and Mixed Memory/Logic Patterns.

4.15.1 Overview

See Scan Test Patterns.

Scan test patterns are a source of *stored* pattern data similar to logic vector data. Scan patterns reside in the combined LVM / SVM memory along with logic vectors.

For each Site Assembly Board (each APG), the combined LVM / SVM memory supplies 64 scan data channels. Each channel supplies three logic bits which define:

- Pattern data (drive/strobe data) (PEL)
- I/O control (PEE)
- Strobe control (PES)

The combined Logic Vector Memory (LVM)/Scan Vector Memory (SVM) memory outputs are routed to the timing system via the Pin Scramble MUX, allowing scan pattern data to be mapped to any tester pin(s) on a per cycle basis.

Scan pattern instructions can only be used in Logic Test Patterns (or Mixed Memory/Logic Patterns). This is required because it is the VAR Engine which controls/accesses the combined LVM/SVM.

Each scan pattern always has the following components:

- At lease one VEC/RPT instruction (but typically more, as in any logic pattern)
- One (and only one) SCANDEF directive per pattern
- One or more scan pattern instruction(s) (SVEC)

For example:

```
%%SCANDEF pin9, pin3, pin2, pin13
%  VEC 01HL10X01HL10X01HL10X // Or RPT
% SVEC 0000
% SVEC 0101
% SVEC 1010
```

The SCANDEF directive identifies which DUT pin(s) will have scan data in the subsequent scan instructions (SVEC). The order the DUT pin(s) are listed identifies the mapping of pattern tokens to pins in subsequent SVEC pattern instructions. This allows the pattern compiler to generate a scan pattern which will be loaded into the correct LVM/SVM locations based on the pin(s) to receive scan data. The SCANDEF directive also controls which of the two scan modes is enabled (see SCANDEF Compiler Directive).

A logic VEC or RPT instruction must be executed before the first scan pattern instruction (SVEC) can be executed. During pattern execution, when a scan instruction (SVEC) is encountered the following operation occurs:

- Any pin which appears in the pattern's SCANDEF directive will be controlled by the scan data mapped to that pin. This is true even if the pin is scrambled to t_lvm.

- Similarly, in scan instructions (SVEC), any attempt to scramble pin(s) which appear in the pattern's SCANDEF directive to t_lvm is ignored. It is possible to scramble these pins to APG resources but this is not recommended.
- Any pin which does not appear in the pattern's SCANDEF directive and is scrambled to t_lvm is controlled by the logic instruction (VEC or RPT) executed prior to the SVEC instruction. During SVEC executions, the logic vector address (VAR) does not change and any pin(s) scrambled to t_lvm (except as noted above) receive the same logic pattern data for each scan instruction in the series.
- Any pin(s) scrambled to t_scan but which do not appear in the pattern's SCANDEF directive will receive scan data from the first pin in the SCANDEF directive. However, do not use this technique intentionally; it is technically an error and this operation may change in the future.
- It is legal to strobe pin(s) in the logic instruction (VEC or RPT) prior to a scan instruction.
- A pattern subroutine may contain only scan instructions (SVEC). The pattern instruction which calls the subroutine must have a VEC or RPT instruction. This will control any pin(s) scrambles to t_lvm, as noted above.
- A subroutine which contains only scan vectors must be defined as a PATTERN(); i.e. Pattern Labels are not allowed on a pure scan instruction.
- Pin(s) which are scrambled to APG resources are not affected by scan pattern instructions.

Note: the following usage restriction was added 3/25/05. This restriction is required to ensure that DUT boards and test programs written for Magnum 1 will operate in Magnum-compatible systems being planned for future development.

As note above, each Site Assembly Board supports two ECR's which can be used as a Logic Error Catch (LEC) to capture scan instruction errors: 1 ECR/LEC captures errors from Sub-site A pins and 1 ECR/LEC captures errors from sub-site B pins. Each ECR/LEC can capture errors from up to 36 pins in its associated sub-site, however, to maintain DUT board and test program compatibility with future Magnum-compatible systems, at most a maximum of 18 pins from each group of 32 pins should be captured. In other words:

- From sub-site A, capture any 18 pins from a_1 to a_32
- From sub-site A, capture any 18 pins from a_33 to a_64
- From sub-site B, capture any 18 pins from b_1 to b_32
- From sub-site B, capture any 18 pins from b_33 to b_64

This requires that the user carefully consider which tester pins are connected, via the DUT board, to the DUT pins which are to be captured in the ECR.

4.15.2 SCANDEF Compiler Directive

See Scan Test Patterns, Overview.

Description

SCANDEF is a compiler directive used in conjunction with Scan Test Patterns.

As noted in Overview, Scan patterns always have the following components:

- At lease one VEC/RPT instruction (but typically more, as in any logic pattern)
- One (and only one) SCANDEF directive per pattern
- One or more scan pattern instruction(s) (SVEC)

The SCANDEF directive addresses the following requirements:

- It identifies which DUT pin(s) will have scan data in the subsequent scan instructions (SVEC). See Note:regarding an important hardware scan pin usage rule.
- It determines how the pattern tokens in SVEC instructions are mapped to pins; the first pin in the SCANDEF directive is associated with the first pattern token in the SVEC instruction, etc. This is required for proper pattern loading.
- The form of SCANDEF directive used (there are two) controls which of the two scan pin modes is enabled.
- Note that in scan instructions (SVEC), any attempt to scramble pin(s) which appear in the pattern's SCANDEF directive to t_lvm is ignored. It is possible to scramble these pins to APG resources but this is not recommended.

As indicated, scan patterns have two modes:

- Standard Scan
- Split-I/O Scan

Standard Scan

Using standard scan mode, each scan pin has I/O capability, is represented by a single scan pattern source token (0,1,H,L,X, Z), and stores a 3-bit value in LVM/SVM. Operation is the same as for logic vectors. For example:

```
%% SCANDEF   p9, p2, p7, p0
PATTERN( myPat, logic )
   VEC  HL10XHL10XHL10XHL10XHL10XHL10X
%  SVEC 1HLX
```

In this example, the SCANDEF directive specifies 4 scan pins in subsequent scan instructions. The 3-bits derived from the first pattern token (1) will be stored in the LVM/SVM for pin p9, the 3-bits derived from the second pattern token (H) will be stored in the LVM/SVM for pin p2, etc.

Split-I/O Scan

Split-I/O scan effectively doubles the scan depth possible when a DUT scan pins all connect to a single Site Assembly Board. When a DUT spans Site Assembly Boards (i.e. Sites-per-Controller > 1), the level of compression will vary.

Using split I//O scan, each pin must either be a dedicated scan-in pin or a dedicated scan-out pin. The pattern compiler derives a 3-bit value, stored in LVM/SVM, for a pair of scan pins, each pair consisting of one scan-in pin and one scan-out pin. During execution, the drive and strobe control bits are routed separately to the two pins: the scan-in pin can only drive and the scan-out pin can only tri-state and optionally strobe. For example:

```
PATTERN( myPat, logic )
%% SCANDEF ( p9, p2 ), ( p3, p5 )
   VEC  HL10XHL10XHL10XHL10XHL10XHL10X
%  SVEC 10 HX
```

In this example, the SCANDEF directive specifies 2 scan-in pins and 2 scan-out pins. Scan-in pins may only use the 1/0 pattern tokens. Scan-out pins may only use H/L,Z,X tokens; the V token is not legal in split-I/O scan patterns.

The pattern compiler derives 3-bits from the first scan-in token (1) plus the first scan-out token (H) and stores them in the LVM/SVM for a single pin (the pattern compiler manages the storage details, matching scan-in pins with scan-out pins on the same Site Assembly Board). The process repeats, combining the 2nd scan-in token plus the 2nd scan-out token, etc.

Using the example above, during pattern execution, pin p9 (scan-in) will be set to drive and receive one of the 3 bits as drive-data (1), and pin p3 (scan-out) will tri-state and receive the other 2 bits: one to enable a strobe and the other as strobe data (H = 1).

Using split I//O scan, there must be at least one scan-in pin and one scan-out pin. Then, if the DUT does not have an equal number of scan-in pins vs. scan-out pins the pattern compiler inserts an implicit a_na for input pin(s) or one output pin(s). In the following example, the pattern compiler inserts a_na for the 3rd and 4th scan-out pin.:

```
%% SCANDEF ( p14, p9, p2, p1 ), ( p3, p5 )
%  SVEC 1010 HX
```

SCANDEF directives take effect in the order they appear in the pattern file. It is legal to define more than one SCANDEF directive in a pattern file but it is NOT legal for the SCANDEF to change for a given pattern or any subroutines called by that pattern. The order in which scan instructions (SVECs) are ultimately executed is not affected by the SCANDEF directive used when compiling each SVEC instruction.

It is possible to define a different SCANDEF directive for each Pin Assignment Table in the program (see Usage). When this is done, every Pin Assignment Table in the program must appear in a SCANDEF definition.

Usage

The following syntax is used for Standard Scan patterns. Each scan pin has I/O capability:

```
%% SCANDEF p1, p2, p3, p4
```

The following syntax is used for Split-I/O Scan patterns. Each pin in the first group is a scan-in pin, each pin in the second group is a scan-out pin:

```
%% SCANDEF (inp1, inp2, inp3, inp4 ),( outp1, outp2, outp3, outp4 )
```

The following syntax is used to define a SCANDEF for each Pin Assignment Table for Standard Scan patterns:

```
%% SCANDEF p1, p2, p3, p4 { pa1 }
%% SCANDEF p1, p2, p3, p4 { pa2, pa3}
```

The following syntax is used to define a SCANDEF for each Pin Assignment Table for Split-I/O Scan patterns:

```
%% SCANDEF ( inp1 ),( outp1 ) { pa1 }
%% SCANDEF ( inp1 ),( outp1 ) { pa2, pa3}
```

where:

p1, **p2**, **inp1**, **inp2**, **outp1**, **outp2**, etc. represent DutPin names. They must be used in the Pin Assignment Table and must be signal pins (not DPS, etc.).

pa1, **pa2**, **pa3** represent the names of three Pin Assignment Tables.

Example

See Description.

4.15.3 SVEC Pattern Instruction

See Scan Test Patterns.

Description

SVEC is the scan pattern instruction token. The following rules apply:

- SVEC can only be used in Logic Test Patterns and Mixed Memory/Logic Patterns but not pure Memory Test Patterns.
- A VEC/RPT pattern instruction must be executed before the first SVEC may be executed.
- If the logic instruction prior to SVEC is a RPT, the RPT will execute the specified number of times (per the RPT count value) before execution proceeds to the SVEC instruction.
- No DUT pins receive any scan data during the VEC/RPT instruction, and the SVEC instruction has no provisions for specifying the pattern data for non-scan pins. During the scan cycles, non-scan pins receive pattern data from the preceding VEC/RPT instruction.
- The number of pattern tokens in an SVEC instruction must agree with the number of pin(s) specified in the preceding SCANDEF directive.
- The standard Logic Vector Bit Codes (1, 0, L, H, Z, V, X) are used in Standard Scan instructions. Split-I/O Scan instructions cannot use the V token (strobe for valid).
- In Split-I/O Scan, the pattern compiler counts the tokens to determine which belong to scan-in pins vs. scan-out pins; i.e. there is no specific delimiter used to mark the end of scan-in tokens vs. scan-out tokens.
- SVEC instructions do select a time set and pin scramble table. See Time-sets (TSET) and Pin Scramble Table. When an explicit value is not specified the default values are used (TSET1, PS1).
- In SVEC instructions, any attempt to scramble pin(s) which appear in the pattern's SCANDEF directive to t_lvm is ignored. It is possible to scramble these pins to APG resources but this is not recommended.
- The VIHH Map may not be specified in an SVEC instruction. It remains as set in the preceding VEC/RPT instruction.
- There are no provisions for repeats or looping on SVEC instruction(s).

- An SVEC microinstruction cannot immediately follow a STARTLOOP directive or immediately precede an ENDLOOP directive.
- The VAR RETURN, VAR DONE, and VAR PAUSE instructions may be used in an SVEC instruction. No other VAR options are legal.
- SVEC instructions can be executed as a subroutine, called from the preceding VEC instruction. A subroutine can consist solely of SVEC instructions.
- DDR scan operation is supported. See Usage and Double Data Rate (DDR) Mode.
- The Magnum 1/2/2x scan implementation, both hardware and software, is quite different than for Maverick-I/-II. Maverick-I/-II style test pattern scan syntax; i.e. using SCAN LOAD and SCAN INC is not supported using Magnum 1/2/2x.

Usage

The following syntax is used for non-DDR scan instructions:

```
% SVEC bit-pattern [, optional params]
```

The following syntax is used for DDR scan instructions:

```
% SVEC A-cycle-bit-pattern B-cycle-bit-pattern[, optional params]
```

where:

% SVEC is the pattern instruction for specifying a scan instruction.

bit-pattern is a set of Logic Vector Bit Codes (0, 1, H, L, V, Z or X) that determine pattern data, I/O control, and strobe enable for each tester channel scrambled to scan memory in the current instruction. See SCANDEF Compiler Directive for details regarding how the order of bit-pattern tokens is specified. The V token cannot be used in Split-I/O Scan mode.

A-cycle-bit-pattern and **B-cycle-bit-pattern** are two sets of Logic Vector Bit Codes (0, 1, H, L, V, Z or X) that determine pattern data, I/O control, and strobe enable for each tester channel scrambled to scan memory in the current instruction. See SCANDEF Compiler Directive for details regarding how the order of bit-pattern tokens is specified. The V token cannot be used in Split-I/O Scan mode.

Note: it is illegal to strobe in the last vector of a logic pattern; i.e. the last vector must not contain any Z, V, H or L tokens.

---

optional params may be used to explicitly select a time-set and/or Pin Scramble Map for the current instruction. The VIHH Map may not be specified in an SVEC instruction, it remains as set in the preceding VEC/RPT instruction.

Example

The following example has 16 scan pins and 4 Standard Scan instructions:

```
%%SCANDEF p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12,\
          p13, p14, p15, p16
PATTERN( myPat, logic )
% VEC   HL10XZVHL10XZVHL10XZVHL10XZVHL10XZVHL10XZVHL10XZV
% SVEC  0000000000000000
% SVEC  0101010101010101
% SVEC  1010101010101010
% SVEC  1100110011001100
```

The following example has 4 scan-in pins and 3 scan-out pins 4 Split-I/O Scan instructions:

```
PATTERN( myPat, logic )
% VEC   HL10XZVHL10XZVHL10XZVHL10XZVHL10XZVHL10XZVHL10XZV
%%SCANDEF ( p1, p2, p3, p4 ), ( p5, p6, p7 )
% SVEC  0000 XXX
% SVEC  1010 HHL
% SVEC  0110 LHL
% SVEC  1101 LLH
% SVEC  1111 HHH
```

4.15.4 Datalogging Scan Failures

Not yet documented for Magnum 1/2/2x.

4.16 Mixed Memory/Logic Patterns

See Pattern Overview and Naming, Pattern Attributes.

Note: this section intentionally includes information which applies to Maverick-I, Maverick-II, Magnum 1, Magnum 2 and Magnum 2x test systems.

Any test pattern containing a mix of logic instructions and memory instructions is, by definition, a mixed memory/logic pattern. Maverick-I, Maverick-II, Magnum 1, Magnum 2 and Magnum 2x all support mixed memory/logic patterns.

Mixed memory/logic pattern testing applications include micro controllers with embedded memory and memory devices with embedded logic (like flash memory with on-board micro controller functions). This section documents how the various system types operate and the options and related rules which apply to mixed memory/logic patterns.

A typical logic-only pattern instruction might appear like the following:

```
% VEC   00001111  HHHHLLLL, TSET2, PS4
```

A typical memory-only instruction might appear like the following:

```
%  YALU       YMAIN, XCARE, CMEQMAX, INCREMENT, DYMAIN
   XALU       XMAIN, XCARE, CON, INCREMENT, DXMAIN
   COUNT      NOCOUNT, AOFF
   MAR        INC, NOREAD, NOINT, RSTTMR
   CHIPS      NOCLKS, ADHIZ
   DATGEN     HOLDDR, HOLDYN, EQFDIS, BCKFDIS, NOTINV, DATDAT
   UDATA      0
   PINFUNC    TSET2, PS4
```

The combined mixed pattern instruction would be:

```
%  VEC      00001111  HHHHLLLL
   YALU       YMAIN, XCARE, CMEQMAX, INCREMENT, DYMAIN
   XALU       XMAIN, XCARE, CON, INCREMENT, DXMAIN
   COUNT      NOCOUNT, AOFF
   MAR        INC, NOREAD, NOINT, RSTTMR
   CHIPS      NOCLKS, ADHIZ
   DATGEN     HOLDDR, HOLDYN, EQFDIS, BCKFDIS, NOTINV, DATDAT
   UDATA      0
   PINFUNC    TSET2, PS4
```

It is the Pin Scramble Map selected in each pattern instruction which determines which data source is actually used by a given pin in each tester cycle.

The Maverick-I APG hardware has a single pattern execution control engine, called the MAR engine, which controls all aspects of both the memory and logic execution. This means that Maverick-I mixed patterns always execute in lockstep; i.e. the memory portion and logic portion of a given pattern instruction always execute in sync. Using Maverick-I, because the MAR engine controls everything, any test pattern which contains a logic instruction is effectively a mixed pattern, even when the pattern source file only contains logic

instructions. Thus, in any Maverick-I pattern which contains only logic instructions, the Default Memory Pattern Instruction is implicitly added by Patcom and logic RPT instructions are treated the same as a single-instruction loop written explicitly using a memory pattern instruction.

The Maverick-II hardware includes a second pattern execution control engine, called the VAR engine, which *may* be used to control logic pattern execution. The term *may* is emphasized because the Maverick-II hardware supports two mixed execution modes, designated in the pattern source file using the Pattern Type Attributes: mav1 and mav2.

A mav1 pattern executes on Maverick-II using only the MAR engine, exactly as if using Maverick-I, Therefore, when using both mav1 and mixed pattern attributes the memory portion and the logic portion of each pattern instruction execute in lockstep. Because of this, the following instruction is illegal in a mav1 mixed pattern because the MAR engine cannot both repeat the logic instruction and increment to the next memory instruction:

```
% RPT 3 HL10X
  MAR INC
```

Using Maverick-II, when using both mav1 and mixed pattern attributes there are actually two potential areas of conflict:

- Conflicting pattern execution control instructions, as noted above (INC, JUMP, GOSUB, RETURN, vs. RPT, etc.).
- Any pattern instruction which contains a Pattern Label and a logic instruction (VEC, RPT) implicitly uses the UDATA field, which means the memory instructions in that instruction may not use UDATA. See UDATA Instruction.

On Maverick-II, with both mav2 and mixed pattern attributes a memory/logic pattern uses both the MAR engine and the VAR engine, allowing the logic pattern execution sequence to diverge from the memory pattern. This means that, with respect to the pattern source file, execution of the memory portion of an instruction vs. the logic pattern portion of the same instruction may or may not occur in the same cycle. Using the previous example, the RPT logic instruction will repeat while the MAR INC memory instruction will proceed to the next source instruction; i.e. execution is not in lockstep with the source pattern instructions. In a mav2 mixed pattern any required synchronization between memory pattern and logic pattern execution is entirely the responsibility of the user's test pattern (which can be *very* challenging to validate and debug).

Note that in Maverick-II mixed mode patterns (mav2 mixed) the pattern compiler does provide a very limited synchronization support, by adding MAR INC to any pattern instructions which do include VEC or RPT but don't include an explicit MAR instruction. Patcom does not, however, attempt any other flow control synchronization between the MAR engine and VAR engine. Thus, if a given pattern instruction contains execution control

instructions which are different for the MAR engine vs. the VAR engine, whether explicit or by default, the memory portion and logic portion of the pattern execution will diverge.

Magnum 1/2/2x include both the VAR engine and MAR engine, but do not directly support `mav1 mixed` mode patterns. Instead, the pattern compiler provides a limited version of this capability, called `mixedsync` mode. When compiling a test pattern for Magnum 1/2/2x, the Pattern System Attributes (`mav1`, `mav2`) are ignored, but the Pattern Type Attributes are used. The `memory`, `logic` and `mixed` options operate the same as described above for Maverick-II. The `mixedsync` option, usable on Magnum 1/2/2x, is used to force lockstep execution of the memory portion and the logic portion of each pattern instruction. Restrictions apply because the MAR engine and VAR engine have somewhat different capabilities, in particular some features that are available in pure logic or pure memory patterns are not available in mixed mode patterns. See MixedSync Pattern Rules.

Magnum 1/2/2x: Rules and Restrictions

These rules apply when using Magnum 1/2/2x:

1. Memory pattern execution is controlled by the `MAR` instruction. Logic pattern execution is controlled by the `VAR` instruction. This is true even when explicit `MAR` and/or `VAR` instructions are not specified (defaults apply).
2. Using the `STARTLOOP` compiler directive, the loop count value is stored in the `VUDATA` field of the instruction immediately following the `STARTLOOP` statement. In mixed memory/logic patterns the instruction immediately following a `STARTLOOP` must not contain a user-defined `VUDATA` value (the pattern compiler will overwrite the user's value).
3. The vector address (VAR) of the logic instruction after the `STARTLOOP` statement is stored in the `VUDATA` field of the pattern instruction immediately preceding the `ENDLOOP` compiler directive. Thus in mixed memory/logic patterns, the pattern instruction immediately preceding an `ENDLOOP` must not contain a user specified `VUDATA` value (the pattern compiler will overwrite the user's value).
4. In any Multi-DUT Test Program testing more than 2 DUTs, a `VECDEF` directive is required (not optional) in any test patterns containing logic instructions.

When mixedsync is specified:

- Patcom enforces the MixedSync Pattern Rules which, with respect to the pattern source file, ensures both the memory and logic components of each pattern instruction execute concurrently (in lockstep). This requires the user's test pattern and any Pattern Subroutines executed by that pattern have the `mixedsync` pattern attribute.

- Patcom also adds instructions/operands to any pattern instruction which conforms to the MixedSync Pattern Rules but which does not contain explicit instructions required to ensure the MAR engine and VAR engine remain synchronized. For example, the following instructions contain components explicitly specified in the user's test pattern and components added by Patcom to ensure lockstep operation:

```
% VEC HL10X // User, explicit
  MAR CJMPE // User, explicit
  VAR CJMPE // Added by Patcom
```

  Note that the MAR/VAR engines have copies of the other's counters which are kept syncronized by Patcom. The added instructions are not shown.

```
% here:
  VEC HL10X              // User, explicit
  VCOUNT  COUNT1, DECR   // User, explicit
  VAR     CJMPNZ, here   // User, explicit
  MAR     CJMPNZ, here   // Added by Patcom
```

- The user's test pattern may select a default memory instruction (using `MAR DEFAULT`, more below), which is applied by Patcom to any pattern instruction which doesn't contain any memory pattern instructions. Similarly, the user's test pattern may select a default logic instruction (using `VAR DEFAULT`), which is applied by Patcom to any pattern instruction which doesn't contain any logic pattern instructions. More below.

Also note the following regarding `mixed` and `mixedsync` patterns:

- `mixedsync` is only valid when compiling a test pattern for Magnum 1, Magnum 2 or Magnum 2x. See APFP Dialog.
- When `mixedsync` is used, it is recommended that the user's pattern only specify either the MAR-side execution control instructions/operand or the VAR-side execution control instructions/operand but not both. This allows Patcom to manage the synchronization between the MAR/VAR engines.
- The maximum size of a `mixedsync` pattern (i.e. the number of pattern instructions) is limited to the size of the APG uRAM (64K). However, this only limits the number of memory instructions, it does not limit the number of logic instructions. And, in `mixedsync` mode, Patcom detects duplicate consecutive memory instructions and, when possible, creates a single-instruction loop, thus conserving uRAM.
- Scan vectors are also stored in LVM and execution is controlled by the VAR engine. Thus any pattern that needs to execute scan vectors must be a `logic`, `mixed` or `mixedsync` pattern.

- When a `mav1 mixed` pattern is compiled for Magnum 1/2/2x it will be compiled as if its pattern type attribute was `mixedsync` (more below). No warnings are issued. Since `mav1` does not support the `logic` attribute and Magnum 1/2/2x always uses the VAR engine, effectively all `mav1` patterns become `mixedsync` patterns when compiled for Magnum 1/2/2x.
- When a `mav2 mixed` pattern is compiled for and executed on Magnum 1/2/2x it will operate the same as if executed on a Maverick-II; i.e. any synchronization is the responsibility of the user's pattern instructions.
- Using Maverick-I/-II, the user has no control over the Default Memory Pattern Instruction that Patcom applies to any `mixed` pattern instructions which don't explicitly specify a memory instruction (i.e. any instruction containing only `VEC` or `RPT`). This limitation also applies to `mixed` patterns compiled for Magnum 1/2/2x.
- Magnum 1/2/2x `mixedsync` patterns allow the user to tag a pattern instruction, using `MAR DEFAULT`, which will subsequently be applied, instead of the built-in Default Memory Pattern Instruction, to any pattern instructions which contain no (as in zero) memory instructions; i.e. in instructions which contain none of the following: `YALU`, `XALU`, `COUNT`, `MAR`, `CHIPS`, `DATGEN`, `UDATA`, `PINFUNC` and `USERRAM`m and `LSENABLE` and `LEVELSET` when controlled by the MAR engine. More below.
- Magnum 1/2/2x `mixedsync` patterns allow the user to tag a logic pattern instruction, using `VAR DEFAULT`, which will subsequently be applied to any pattern instructions which contain no (as in zero) logic instructions; i.e. in instructions which contain none of the following: `VEC`, `RPT`, `VAR`, `VCOUNT`, `VPINFUNC`, `VUDATA` and `SVEC`, and `LSENABLE` and `LEVELSET` when controlled by the VAR engine.
- The `MAR`/`VAR` `DEFAULT` operands only affect the pattern compiler, causing it to record the instruction containing the operand for subsequent use. The compiler then applies this default instruction, as noted above, until it is redefined. Instructions containing either/both `DEFAULT` operand(s) otherwise executed normally.
- The scope of `MAR`/`VAR` `DEFAULT` operands is limited to the pattern being compiled. Starting a new pattern (i.e. each `PATTERN( )` statement) resets all default instructions to the system default. This includes patterns which are treated as Pattern Subroutines.
- It is legal to apply `MAR DEFAULT` and `VAR DEFAULT` to the same instruction.
- There is no explicit pattern instruction/operand to cause operation to revert to the system default instruction. However, this can be accomplished indirectly by including `MAR DEFAULT` and/or `VAR DEFAULT` in a pattern instruction which contains no other memory or logic instructions.

- As indicated above, execution of the LSENABLE and LEVELSET instructions can be controlled by either the MAR engine (memory instruction) or the VAR engine (logic instruction). In mixed and mixedsync patterns, default execution is controlled by the MAR engine. To transfer execution control to the VAR engine requires an explicit memory pattern instruction: PINFUNC VLEVELSET. Thus, by default, in mixedsync patterns, LSENABLE and LEVELSET are treated as memory instructions and even though a given pattern instruction only contains logic instructions plus LSENABLE or LEVELSET that instruction will not have the user's default memory instruction applied because it contains memory instructions. Similarly, when an instruction contains PINFUNC VLEVELSET that instruction will also not have the user's default memory instruction applied because it contains a memory instruction. However, if the PINFUNC VLEVELSET instruction is included in the user's default memory instruction, any pattern instruction which doesn't contain any memory instructions will use the user's default and transfer LSENABLE and LEVELSET execution to the VAR engine.

MixedSync Pattern Rules

Note that the mixedsync option applies only to Magnum 1/2/2x.

As indicated above, the purpose of the mixedsync option is to ensure lockstep execution of the memory portion and logic portion of each pattern instruction. And, since the Magnum 1/2/2x hardware cannot do this, it is up to the user's pattern plus the pattern compiler (Patcom). The mixedsync option causes Patcom to output pattern instructions that result in identical execution sequence for both the MAR Engine and VAR Engine, or to issue an error. To do this, Patcom must limit the pattern features used in mixedsync patterns to the intersection of the features available in both memory instructions and logic instructions.

The reason for some of the rules below will be fairly obvious. For example, disallowing execution control instructions which would cause memory pattern execution to diverge from logic pattern execution, and vice-versa. Other rules are required because the MAR Engine and VAR Engine have somewhat different capabilities. For example, the VAR engine implicitly uses the UDATA field to store Pattern Subroutine/branch addresses, whereas the MAR engine stores them separately. This limits the user's ability to use UDATA in mixedsync patterns.

Note also that, for simplicity, the use of a Pattern Label is presumed to indicate that the associated instruction is the target of a conditional or unconditional branch or is treated as a Pattern Subroutine.

The following restrictions apply to mixedsync patterns:

1. Any memory instruction with a label cannot use the UDATA field.
2. Any memory or logic instruction with a label cannot use:
   - MAR or VAR instructions with conditional or unconditional branch or Pattern Subroutine (GOSUB) operands. Note that subroutine RETURN is OK, as is the special case of a memory instruction which loops on itself.
   - COUNT COUNTUDATA and VCOUNT COUNTVUDATA
   - LSENABLE or LEVELSET
   - MAR INTADR or INTENADR (INTEN is OK)
   - STARTLOOP and ENDLOOP
   - USERRAM
   - RPT
3. Any memory pattern instruction with a MAR conditional or unconditional branch or Pattern Subroutine operand cannot also include COUNT COUNTUDATA or VCOUNT COUNTVUDATA.
4. In a given instruction it is illegal to select a MAR counter and a VAR counter which are different. For example:

```
% COUNT   COUNT1, DECR
  VCOUNT  COUNT2, DECR // Illegal, must be COUNT1
```

5. It is illegal to specify a MAR engine control statement and VAR engine control statement which *might* result in a different pattern execution sequence on the two engines. For example:

```
% MAR  CJMPNZ, label
  VAR  CJMPZ, label        // Illegal: CJMPNZ vs. CJMPZ

% MAR  INC
  RPT  3    HL10X          // Illegal: INC vs. RPT

% MAR    CJMPNZ, label
  VAR    CSUBE, label      // Illegal: CJMPNZ vs. CSUBE

% MAR    CJMPNZ, label_1
  VAR    CJMPNZ, label_2   // Illegal: different jump labels
```

   As indicated earlier, when mixedsync is used, the user should specify either MAR-side execution control instructions/operand or VAR-side execution control instructions/operand but not both. Allow Patcom to manage the synchronization between the MAR/VAR engines.

6. It is illegal to specify any operand of the following instructions which do not have an equivalent for both the MAR engine and VAR engine:

   - MAR and VAR
   - COUNT and VCOUNT

   For example:

   ```
   % COUNT  COUNT1, DEC2 // Illegal: VCOUNT doesn't support DEC2
   ```

   This rule does not apply to counter selections; i.e. even though the VAR engine has only four counters (COUNT1 .. COUNT4) and the MAR engine has 64 counters (COUNT1 .. COUNT64) the counter selections in COUNT and VCOUNT instructions operate as desired.

7. It is illegal to use any USERRAM GET instruction that may affect instruction execution sequence, anywhere in the pattern; e.g. changes the value in a counter used for conditional branch operations.

8. When COUNT COUNTUDATA is used in the pattern, if the counter is used for conditional branch operations, anywhere in the pattern, do not modify the UDATA value from the test program (using set_udata()).

A common error made in mixed mode patterns is to specify a Time-set (TSET#), Pin Scramble Map (PS#), VIHH Map (VIHH#), etc. in a logic VEC or RPT instruction without also enabling the selection using the corresponding PINFUNC operand (VTSET, VPS, VOVER, etc.). Without these additional PINFUNC operands, the values specified in the VEC or RPT instruction are not used and the memory instruction selections are used (or the default values are used: TSET1, PS1, VIHH1, etc.). In mixedsync patterns, Patcom detects these and related situations and attempts to correct or warn the user, as shown in the following table. Note that the table uses the time set selection and PINFUNC VTSET but that operation

is similar for each of the parameters which have both MAR-side and VAR-side selections (PS#, VPULSE, TSET#, etc.):

Instruction	Patcom Action	Note
`% PINFUNC TSET8`	None	Uses the default logic instruction.
`% VEC HL10X, TSET8`	Patcom adds `% PINFUNC VTSET` and issues a warning that it was added.	VAR engine time set specified without explicit `PINFUNC VTSET`.
`% PINFUNC TSET8` `VEC HL10X, TSET23`	Patcom warns that TSET23 is not used.	VAR engine TSET specified but not used.
`% PINFUNC TSET8, VTSET`	Patcom warns that TSET8 is not used.	MAR engine TSET specified but not used.
`% PINFUNC VTSET` `VEC HL10X`	TSET1 is used (default) with no warning.	Implicit time set (TSET1).

4.17 Controlling PE Levels from the Test Pattern

See Test Pattern Programming.

Many of the tester's programmable voltages/currents can be set or modified (tweaked) from an executing test pattern.

Note: the hardware implementation, pattern syntax used, and general capabilities are quite different using Maverick-I/-II vs. Magnum 1/2/2x.

This section contains the following:

- changes_voltages()
- level_set_value_change()

4.17.1 Controlling Magnum 1 Levels from the Test Pattern

See Controlling PE Levels from the Test Pattern.

Overview

Many of the tester's programmable voltage/current parameters, i.e. levels, can be set or modified (tweaked) from an executing test pattern. Using Magnum 1/2/2x, this is done using two pattern instructions:

- LSENABLE Pattern Instruction - identifies the target hardware using a pin list.
- LEVELSET Pattern Instruction - specifies the type of operation (set/tweak), the parameter to modify (VIL, DPS voltage, etc.), and a range if required.

Note: important usage details are described separately from the LSENABLE Pattern Instruction and LEVELSET Pattern Instruction sections. See below.

This section covers the following topics:

- Controllable Level Types
- Operation and Rules
- DC Level Response Time
- DPS Considerations
- Pattern Loop Considerations
- Set/Tweak PASS/FAIL Limits
- Set/Tweak in Subroutines
- Setting a Static Pin-state using Level Sets.

Supporting functions include:

- changes_voltages()
- level_set_value_change()

Controllable Level Types

The table below shows the various level parameters which can be controlled using methods documented in this section:

Table 4.17.1.0-1 Levels Controllable from the Test Pattern

Level	Pattern Token	C-Code Set Function
VIL	LS_VIL	vil()
VIH	LS_VIH	vih()
VOL	LS_VOL	vol()
VOH	LS_VOH	voh()
VTT	LS_VTT	vtt()
VZ	LS_VZ	vz()
VIHH	LS_VIHH	vihh()
DPS Primary Voltage	LS_DPS	dps()
DPS Secondary Voltage	LS_DPS_VPULSE	dps_vpulse()
DPS Current Test High Limit	LS_DPS_CURRENT_HIGH	dps_current_high()
DPS Current Test Low Limit	LS_DPS_CURRENT_LOW	dps_current_low()
PMU Force Current	LS_PMU_IPAR_FORCE	ipar_force()
PMU Current Test High Limit	LS_PMU_IPAR_HIGH	ipar_high()
PMU Current Test Low Limit	LS_PMU_IPAR_LOW	ipar_low()
PMU Force Voltage	LS_PMU_VPAR_FORCE	vpar_force()
PMU Voltage Test High Limit	LS_PMU_VPAR_HIGH	vpar_high()
PMU Voltage Test Low Limit	LS_PMU_VPAR_LOW	vpar_low()
PMU Voltage Clamp High	LS_PMU_VCLAMP_POS	vclamp()
PMU Voltage Clamp Low	LS_PMU_VCLAMP_NEG	
PTU Force Current	LS_PTU_IPAR_FORCE	ptu_ipar_force_set()
PTU Current Test High Limit	LS_PTU_IPAR_HIGH	ptu_ipar_high_set()

Table 4.17.1.0-1 Levels Controllable from the Test Pattern *(Continued)*

Level	Pattern Token	C-Code Set Function
PTU Current Test Low Limit	LS_PTU_IPAR_LOW	ptu_ipar_low_set()
PTU Force Voltage	LS_PTU_VPAR_FORCE	ptu_vpar_force_set()
PTU Voltage Test High Limit	LS_PTU_VPAR_HIGH	ptu_vpar_high_set()
PTU Voltage Test Low Limit	LS_PTU_VPAR_LOW	ptu_vpar_low_set()
PTU Voltage Clamp High	LS_PTU_VCLAMP_POS	ptu_vclamp_set()
PTU Voltage Clamp Low	LS_PTU_VCLAMP_NEG	
HV Force Voltage	LS_HV_VOLTAGE	hv_voltage_set()
HV Current Test High Limit	LS_HV_IPAR_HIGH	hv_ipar_high()
HV Current Test Low Limit	LS_HV_IPAR_LOW	hv_ipar_low()
HV Voltage Test High Limit	LS_HV_VPAR_HIGH	hv_vpar_high()
HV Voltage Test Low Limit	LS_HV_VPAR_LOW	hv_vpar_low()
None	_SEL_RT_D0	See Setting a Static Pin-state using Level Sets.

Note: the Per-pin Parametric Test Unit (PTU) supplies various DC levels, including VZ and VIHH, used during functional tests (see Magnum PE Driver Modes), the background voltage optionally used during PMU tests (see back_voltage()) and all levels used to perform PTU voltage/current tests (see PTU Functions).

Operation and Rules

The following two example pattern instructions show the key features used to control levels from the test pattern. Following this is a mix of usage rules and operational descriptions:

```
% LSENABLE  PinListName, TSET# // TSET is optional
% LEVELSET  OPERATION, LevelToken, RANGE, TSET#
  UDATA     VALUE UNITS
     ... or ...
  VUDATA    VALUE UNITS // Requires PINFUNC VLEVELSET
```

Note: the LSENABLE Pattern Instruction and LEVELSET Pattern Instruction must always be used as a contiguous pair. The pattern compiler enforces this rule.

- In Multi-DUT Test Programs, only levels for DUT(s) in the Active DUTs Set (ADS) are affected.
- The LSENABLE Pattern Instruction uses a pin list to identify the target hardware on which a level will be modified. Optionally, a time-set may also be specified to be used in the LSENABLE Pattern Instruction (TSET1 is the default).
- The pin list(s) used in the LSENABLE Pattern Instruction(s) *must be* defined in the test program before test patterns are loaded. See LSENABLE Pattern Instruction.
- An LSENABLE Pattern Instruction implicitly uses the UDATA and VUDATA fields; i.e. they cannot be used for other purposes. More below.
- The LSENABLE Pattern Instruction may not be the first instruction in the test pattern.
- The LEVELSET Pattern Instruction must immediately follow the LSENABLE Pattern Instruction. The pattern compiler enforces this rule.
- The LEVELSET Pattern Instruction specifies the remaining parameters; i.e.:
  - The desired operation: SET or TWEAK.
  - The specific level parameter to be set/tweaked, using pattern tokens from the table above (or see LEVELSET Pattern Instruction).
  - The desired voltage or current value. This is specified in the UDATA field (Memory Test Patterns) or VUDATA field (Logic Test Patterns). See LEVELSET Pattern Instruction for details of Mixed Memory/Logic Patterns.
  - When required, a range selection (see LEVELSET Pattern Instruction).
  - Optionally, a time-set selection (TSET) may also be specified.
- The LEVELSET Pattern Instruction may not be the last instruction in the test pattern.
- The LEVELSET Pattern Instruction may not have a Pattern Label, to prevent accidentally branching to the LEVELSET Pattern Instruction, which requires a label.

Note: in each LEVELSET Pattern Instruction, the compiled UDATA value stores more than just the user specified voltage/current value. And, the LSENABLE Pattern Instruction uses the UDATA value implicitly, to store information related to target hardware. In both cases, it is NOT valid to use `get_udata()` or `set_udata()` to access the UDATA value in these instructions. In particular, using `set_udata()` will corrupt the information generated by the compiler, and *result in incorrect operation*. Instead, the `level_set_value_change()` function is available to modify selected parameters specified in a target LEVELSET Pattern Instruction. There are no provisions for changing the target hardware selections made in a LSENABLE Pattern Instruction.

- The LSENABLE Pattern Instruction and LEVELSET Pattern Instruction may be used in both Memory Test Patterns and Logic Test Patterns (more below).
- During pattern execution, the LSENABLE and LEVELSET operations access the pin scramble RAM in areas which are reserved solely for this purpose. This means that the normal pin scramble operation isn't usable in LSENABLE and LEVELSET instructions, which has the following effects:
  - In an LSENABLE and LEVELSET instruction selecting a pin scramble is illegal. The pattern compiler will issue an error if PINFUNC or VPINFUNC instructions include a pin scramble selection (PS#).
  - During an LSENABLE and LEVELSET execution, the pattern data, strobe enable, I/O control and drive format selection are determined by the instruction executed immediately prior to the LSENABLE, for all pins. However, edge timing values are determined by the time-set selected in the LSENABLE and LEVELSET instruction.
  - During an LSENABLE and LEVELSET execution, pattern operands which change the state of the pattern generator hardware will continue to do so i.e. XALU, YALU and DATGEN can/will modify the APG state, VEC will increment the vector address, etc.
- LSENABLE and LEVELSET instructions may include MAR and VAR instructions but operands which control pattern execution flow are limited to INC; i.e. MAR INC and VAR INC. Other pattern execution flow operands are illegal i.e. all conditional and unconditional branch and subroutine call/return operands, etc. Other MAR/VAR operands operate normally.
- As indicated above, the LSENABLE instruction implicitly uses the UDATA field (Memory Test Patterns) or VUDATA field (Logic Test Patterns) to store pin list information; i.e. UDATA and VUDATA are not available for other applications in

these cycles. In Mixed Memory/Logic Patterns whether UDATA or VUDATA is used is determined by whether PINFUNC VLEVELSET the is specified (more below). See Note:

- The LEVELSET instruction explicitly uses the UDATA value (Memory Test Patterns) or VUDATA value (Logic Test Patterns) to specify the desired voltage or current value. In Mixed Memory/Logic Patterns whether UDATA or VUDATA is used is determined by whether PINFUNC VLEVELSET the is specified (more below). See Note:
- The COUNT and VCOUNT instructions are usable in LSENABLE and LEVELSET instructions although condtional operations based on a counter value are not.
- The PINFUNC/VPINFUNC instruction is usable as follows:
  - In Mixed Memory/Logic Patterns, by default the memory pattern controls level set operation. PINFUNC VLEVELSET may be specified to enable the logic pattern hardware to control level setting; (more below).
  - PINFUNC VLEVELSET may not be specified in pure Logic Test Patterns (no PINFUNC instructions may be specified as they are memory instructions).
  - PS# - Illegal in LSENABLE and LEVELSET instructions. The pin scramble selection is implicit, as described above.
  - VPULSE operates normally.
  - TSET# selection affects only edge times and cycle period value. As noted above, the drive format is determined by the instruction which executes immediately prior to the LSENABLE instruction.
  - Only the first 8 VIHH# maps may be used, otherwise VIHH# operate normally
  - VVCOMP, VLATCHRESET, VOVER, VTSET, VVIHH, VVPULSE, VLEVELSET operate normally.
  - ADHIZ - has no effect in LSENABLE instructions. As indicated above, in LSENABLE instructions the I/O state control for APG data generator outputs is latched and re-used from the instruction executed immediately cycle prior to the LSENABLE instruction.
- As indicated above, each unique LSENABLE pin list is stored in a reserved area of the pin scramble memory. The reserved memory can store up to 64 pin lists, thus a given test pattern may define up to 64 unique LSENABLE pin lists. This is checked by the pattern compiler, however, subroutines are not considered in the compile-time error check, thus runtime software may also report a related error. This is also noted in Set/Tweak in Subroutines.

- In Mixed Memory/Logic Patterns, an additional consideration exists when tweaking a given level parameter. Before a level-setting test pattern executes, for each level type to be modified by the pattern, the system software reads the current value of the *first pin* (only!) in each LSENABLE pin list and stores the value in a special hardware register. Using Magnum 1, separate registers are used for memory patterns vs. logic patterns and a given value is saved in both. Then, during pattern execution, when a level is, for example, tweaked from, for example, the memory hardware (i.e. PINFUNC VLEVELSET is not specified in the LEVELSET Pattern Instruction) the register value is modified by adding the tweak value to the original value stored in the memory pattern's register. The register value is then written to the target hardware, which can consist of multiple pins/DPSs/HVs. This single stored reference value is modified again each time the same level is tweaked from the memory hardware. This raises several issues:
  - If the same level is subsequently tweaked from the logic hardware (i.e. PINFUNC VLEVELSET is specified in the LEVELSET Pattern Instruction) the reference level is read from the logic pattern hardware register, which initially will contain the original value read/saved i.e. it doesn't reflect any changes made from the memory pattern.
  - When the LSENABLE pin list contains multiple pins/DPSs/HVs all are set/tweaked to the same value each time the LEVELSET instruction executes. This is true even if they were originally at different levels.
- The cycle period of any pattern instruction containing LEVELSET must be a minimum of 140nS. Neither the pattern compiler nor the system software can check this; i.e. it is the user's responsibility. **IMPROPER** operation is likely if this rule is violated.
- The minimum time between any 2 pattern instructions containing LEVELSET is 1.5uS. Neither the pattern compiler nor the system software can check this; i.e. it is the user's responsibility. **IMPROPER** operation is likely if this rule is violated.

Note: the time values noted in the two previous bullets are required by the test pattern hardware and do not include the time required for the DC circuitry to react and begin to change the target level. See DC Level Response Time.

DC Level Response Time

The test pattern LEVELSET instruction operates by sending a serial command to the target hardware identified via the LSENABLE instruction. The following table shows the amount of time required for the target hardware output level to *begin* to change. For target levels which are internal to the system, no additional time is required for the level to reach the final

programmed value. However, for external levels (i.e. those that reach the DUT) additional time may be required and this time is normally dependent on both the magnitude of the level change and the enviroment/load presented by the user's DUT board and the DUT. For these levels, additional settling time may be necessary, requiring user characterization, and the table below does not include this added time:

Target Level	Min	Max
VIH VIL VOL VOL VTT, VZ, VCOMP PTU DC Comparators PTU Force Voltage/Current PTU Clamps	31uS	93uS
PMU Force Current DPS Force Voltage HV Force Voltage PMU High/Low Pass/Fail Test Limits	31uS	
PMU Force Voltage	10uS	

DPS Considerations

When setting or tweaking level parameters from DUT Power Supply (DPS) several additional consideration exists:

- Each DPS has two programmable output voltages: primary and secondary (VPulse). The test pattern may modify either value and the selection of which value is output by a given DPS may be changed by the test pattern. More below.
- The low level DPS hardware resolution is different than that allowed by the software.

Regarding the latter, the DPS software resolution is 5mV but the underlying hardware resolution is may be different. When setting or tweaking DPS output voltage the hardware resolution determines the final resolution of a given voltage value. This has two effects:

- If the test pattern is tweaking/setting another voltage parameter along with the DPS voltage and the goal is to have the two levels track each other it is important that the tweak value of both parameters be specified in increments which are evenly divisible by 5mV.
- After the test pattern changes a DPS voltage if the `dps()` getter function is used it may return a value not normally obtainable using the `dps()` setter function.

As indicated above, each DPS has two programmable voltages: primary (set using `dps()`) and VPulse (set using `dps_vpulse()`). But, the DPS output mode is configurable (see `dps_output_mode_set()`). In the default mode (`t_dps_vpulse` mode), both outputs of a given DPS are always set to the primary voltage (set using `dps()`) or to the secondary voltage (set using `dps_vpulse()`). Using `dps_output_mode_set()`, if the mode is set to `t_dps_independent`, the voltage of the two outputs of a given DPS can be programmed independently, using `dps()`. In this mode, programming `dps_vpulse()` has no effect (unless the mode is switched back to `t_dps_vpulse` mode). When modifying a DPS output voltage from the test pattern two level tokens are available:

- `LS_DPS` - selects the primary output voltage for modification. If the target DPS is in `t_dps_vpulse` mode, using `LS_DPS` modifies the primary output voltage only, affecting any DPS(s) which are currently set to output the primary voltage. If a DPS is in `t_dps_independent` mode, using `LS_DPS` modifies both the A and B DPS outputs.
- `LS_DPS_VPULSE` - selects the secondary (VPulse) output voltage for modification. If the target DPS is in `t_dps_vpulse` mode, using `LS_DPS_VPULSE` modifies the secondary output voltage only, affecting any DPS(s) which are currently set to output the secondary (VPulse) voltage. If the DPS is in `t_dps_independent` mode, using `LS_DPS_VPULSE` modifies the DPS B output only.

Pattern Loop Considerations

When a test pattern contains loops which *set or tweak* a level note the following:

- **Be *very cautious***. The normal system software used to program a given level is not used, thus no limits are enforced on how high or low the level can be set or tweaked. It is quite easy for a pattern loop to tweak a voltage/current to the maximum value developed by the hardware, in both the positive or negative direction. This can even exceed the values normally obtained from the test program code. And, it is possible when a hardware limit is reached that the next increment programmed in the pattern loop will set the level to the opposite hardware limit. ***BEWARE!***
- Within the pattern loop, it may be necessary to add instructions to create a delay between `LEVELSET` executions. See Operation and Rules. And, each level parameter type has different slew-rate and settling time characteristics; the desired operation is the responsibility of the user.

Set/Tweak PASS/FAIL Limits

When the test pattern sets or tweaks a PASS/FAIL test limit, special consideration must be given to how these values are used in the hardware.

For example, when `partest()` executes a PMU test, or `test_supply()` executes a DPS current test, three possible PASS/FAIL conditions can be specified: `pass_pcl`, `pass_nicl`, or `pass_ncl`. However, the DC comparators actually only use `pass_pcl` and `pass_ncl`, thus when `pass_nicl` is specified the system software automatically swaps the user's specified high/low limits to create the desired results. The getter API functions and Voltage and Current Tool all handle this transparently when the values are set using the standard functions. However, this swapping is not (cannot be) done when PASS/FAIL limits are controlled from the test pattern e.g. the user must manage this process explicitly when modifying PMU, PTU, DPS and HV test PASS/FAIL limits from the pattern. And, when the PASS/FAIL values are swapped by the user neither the standard *getter* functions nor Voltage and Current Tool know that the values were reversed, and thus won't be (can't be) correct for the swap.

Set/Tweak in Subroutines

As noted earlier, each unique `LSENABLE` pin list is stored in a reserved area of the pin scramble memory. The reserved memory can store up to 64 unique pin lists, thus a given test pattern may define up to 64 unique `LSENABLE` pin lists. This is checked by the pattern compiler, however, subroutines are not considered in the compile-time error check, thus runtime software may also report a related error.

4.17.1.1 LSENABLE Pattern Instruction

See Controlling Magnum 1 Levels from the Test Pattern, Controlling PE Levels from the Test Pattern, DUT-specific Pin Lists.

Description

When Controlling Magnum 1 Levels from the Test Pattern, the target hardware is identified using the `LSENABLE` pattern instruction. Note the following:

- **Important** usage rules are documented in Operation and Rules, Pattern Loop Considerations, Set/Tweak PASS/FAIL Limits and Set/Tweak in Subroutines. Read these!
- The `LSENABLE` instruction `PinListName` operand identifies which hardware is being modified (more below). In Multi-DUT Test Programs, only levels for DUT(s) in the Active DUTs Set (ADS) are affected.
- A given test pattern may define up to 64 unique `LSENABLE` pin lists.

- The PinListName operand specified in an LSENABLE instruction must exactly match the name of a pin list defined in the test program. This pin list *must* be defined before test patterns are loaded i.e. either statically or in a CONFIGURATION() block (before the SITE_BEGIN_BLOCK()). The pattern compiler cannot check this or whether the specified pin list members are valid; this will be checked by the system software as the test patterrns are loaded and executed.
- If the LEVELSET instruction paired with a given LSENABLE instruction uses the PINFUNC VLEVELSET option, the LSENABLE instruction must also use it. This is enforced by the pattern compiler.
- In the test program, the specified pin list must contain only pins of one type:
  - Signal pins (PE pins) only. These are used to identify which pin(s) will be modified when PE levels are adjusted (VOL/VOH, VIL/VIH, VTT/VZ and VIHH), or which PTUs will be modified when PTU levels are adjusted (force voltage/current, PASS/FAIL test limits), or which PMUs are modified when PMU levels are adjusted (force V/I, PASS/FAIL limits, clamps, etc.).
  - DPS pins only. These are used to identify which DPS(s) will be modified when DPS levels are adjusted (output voltage, PASS/FAIL test limits, clamps, etc.) or which PMUs are modified when PMU levels are adjusted (for PMU-on-DPS tests). See DPS Considerations.
  - HV pins only. These are used to identify which HV(s) will be modified when HV levels are adjusted (output voltage, PASS/FAIL test limits, etc.) or which PMUs are modified when PMU levels are adjusted (for PMU-on-HV tests).

  Note that it is the LEVELSET Pattern Instruction parameters which determines which level type (VIL, VOH, etc.) is being modified, the type of adjustment (set or tweak), the desired level value, etc.
- During test program execution, the pin list members may be modified by user C-code. However, the target hardware type represented by these pins (PE vs. DPS vs. HV) must remain the same. This is not checked by the system software. Proper operation is very unlikely if this rule is violated.
- DUT-specific Pin Lists may be used to allow pins of some DUTs to be affected while other pins remain unchanged.

Usage

```
% LSENABLE  PinListName, TSET#
  PINFUNC   VLEVELSET // Optional
```

where:

PinListName identifies a pin list which identifies the target hardware which will be set/tweaked by subsequent LEVELSET Pattern Instruction. Specific rules apply, see Description and Operation and Rules.

TSET# is optional and, if used, specifies the time-set to be selected while the LSENABLE instruction executes. Legal values are TSET1 through TSET32. Default = TSET1.

PINFUNC VLEVELSET is optional and only useful in Mixed Memory/Logic Patterns, see Operation and Rules. If PINFUNC VLEVELSET is used in the LSENABLE instruction is must also be used in the subsequent LEVELSET instruction.

Example

The following Logic Test Pattern sets the primary DPS output voltage to 3.3V on all DPS identified in the pin list named my_vcc_pins (see DPS Considerations):

```
// Test program code
PIN_LIST( my_vcc_pins ){
   PINS2( vcc1, vcc2 )
}
// Test pattern
PATTERN( my_pattern, logic )
// One (minimum) or more logic instructions here
%  VEC          HL10X HL10X HL10X // and so on ....
%  VEC          HL10X HL10X HL10X
   LSENABLE     my_vcc_pins
%  VEC          HL10X HL10X HL10X
   LEVELSET     SET, LS_DPS // TSET1 cycle >= 140nS
   VUDATA       3.3 V
// More logic instructions as needed
```

The following Memory Test Pattern sets the primary DPS output voltage to 3.3V on all DPS identified in the pin list named my_vcc_pins (see DPS Considerations):

```
// Test program code
PIN_LIST( my_vcc_pins ){
   PINS2( vcc1, vcc2 )
}
// Test pattern
PATTERN( my_pattern memory)
// One or more memory instructions here
%  XALU       ...
```

```
   YALU       ...
   COUNT      ...
   MAR        ...
   CHIPS      ...
   DATGEN     ...
   PINFUNC    ...
   UDATA      ...
% LSENABLE     my_vcc_pins
% LEVELSET     SET, LS_DPS // TSET1 cycle >= 140nS
   UDATA       3.3 V
```

The following mixed Memory Test Pattern/Logic Test Pattern sets the primary DPS output voltage to 3.3V on all DPS identified in the pin list named `my_vcc_pins` (see DPS Considerations). The desired level value is taken from the VUDATA field:

```
// Test program code
PIN_LIST( my_vcc_pins ){
   PINS2( vcc1, vcc2 )
}
// Test pattern
PATTERN( my_pattern, mixed )
// One (minimum) or more logic/memory instructions here
% VEC          HL10X HL10X HL10X // and so on ....
   XALU       ...
   YALU       ...
   COUNT      ...
   MAR        ...
   CHIPS      ...
   DATGEN     ...
   PINFUNC    ...
   UDATA      ...
% LSENABLE     my_vcc_pins
   PINFUNC     VLEVELSET // Must match LEVELSET's PINFUNC option
% LEVELSET     SET, LS_DPS // TSET1 cycle >= 140nS
   VUDATA      3.3 V
   PINFUNC     VLEVELSET // Use VUDATA value and pin list from
                         // logic LSENABLE RAM
```

4.17.1.2 LEVELSET Pattern Instruction

See Controlling Magnum 1 Levels from the Test Pattern, Controlling PE Levels from the Test Pattern

Description

The LEVELSET pattern instruction is used to set or modify (tweak) a level from an executing test pattern. Note the following:

- **Important** usage rules are documented in Operation and Rules, Pattern Loop Considerations, Set/Tweak PASS/FAIL Limits and Set/Tweak in Subroutines. Read these!
- The LEVELSET instruction must immediately follow an LSENABLE Pattern Instruction. The pattern compiler enforces this rule. The LSENABLE Pattern Instruction determines which hardware will be modified using a pin list. In Multi-DUT Test Programs, only levels for DUT(s) in the Active DUTs Set (ADS) are affected.
- The LEVELSET instruction causes the specified level parameter to be set or tweaked. In hardware, the process begins immediately but requires a minimum of 140nS to complete (not counting any voltage slew rate and/or settling time). Thus, the minimum cycle period used in a LEVELSET instruction is 140nS. See Operation and Rules.
- The LEVELSET instruction specifies the following parameters:
  - The desired operation: SET or TWEAK.
  - The target level to be modified, see table below.
  - The VALUE to be written, in the UDATA value (Memory Test Patterns) or VUDATA value (Logic Test Patterns).
  - A RANGE value, if required (more below).
- A pattern instruction containing LEVELSET may not have a Pattern Label. This is enforced by the pattern compiler, to prevent [accidentally] branching to a LEVELSET instruction without first executing an LSENABLE Pattern Instruction.
- The LEVELSET instruction can be used in both Memory Test Patterns and Logic Test Patterns. In Mixed Memory/Logic Patterns, by default the VALUE is taken from the UDATA instruction. Using PINFUNC VLEVELSET causes the VALUE to be taken from the VUDATA instruction. Using PINFUNC VLEVELSET has other effects, see

Operation and Rules. When a LEVELSET instruction uses PINFUNC VLEVELSET the preceding (corresponding) LSENABLE instruction must also use it. This is enforced by the pattern compiler.

- As noted above, a RANGE value must be specified if the target level has multiple operating ranges (see Magnum 1/2/2x Test Pattern Set/Tweak Parameter List). The RANGE value is NOT used to set a hardware range, but rather to calculate the correct value to write to the target hardware (a D/A converter (DAC) value). This calculated value is only correct if the target hardware is currently in the range specified.

Note: proper level set operation requires that the RANGE value specified in a LEVELSET instruction be identical to the range value selected in the hardware at the time the test pattern is executed. No related error checks are made; i.e. the user is responsible.
Careless programming can cause damage to the DUT!

Note: unlike Maverick-II, the Magnum 1/2/2x software does not set up initial conditions C-code to set a hardware range i.e. the range set in the hardware at the time the test pattern is executed is not changed by the test pattern.

- Any levels modified by the test pattern remain modfied after the pattern execution ends.

The following table lists the parameters which can be directly set or modified (tweaked) from a Magnum 1/2/2x test pattern:

Table 4.17.1.2-1 Magnum 1/2/2x Test Pattern Set/Tweak Parameter List

Level Parameter Pattern Token	Range Options	Range	Notes
LS_VIL LS_VIH LS_VOL LS_VOH LS_VTT	None	-1V to +7V	LSENABLE Pattern Instruction PinList contains only PE pins.
LS_VZ	None	-1V to +7V	LSENABLE Pattern Instruction PinList contains only PE pins.

Table 4.17.1.2-1 Magnum 1/2/2x Test Pattern Set/Tweak Parameter List

Level Parameter Pattern Token	Range Options	Range	Notes
LS_VIHH	None	0V to +12.5V	
LS_DPS LS_DPS_VPULSE	None	-15V to +15V	**WARNING**: normal limit checks on DPS output voltage are NOT enforced, including a negative voltage output. LSENABLE Pattern Instruction PinList contains only DPS pins. See DPS Considerations. ±600mA only when using the DPS 300mA/600mA DPS Option.
LS_DPS_CURRENT_HIGH LS_DPS_CURRENT_LOW	RANGE1 RANGE2 RANGE3 RANGE4 RANGE5 RANGE6	±4uA ±40uA ±400uA ±4mA ±40mA ±400mAor /±600mA	
LS_PMU_IPAR_FORCE LS_PMU_IPAR_HIGH LS_PMU_IPAR_LOW	RANGE1 RANGE2 RANGE3 RANGE4 RANGE5	±2uA ±20uA ±200uA ±2mA ±20mA	LSENABLE Pattern Instruction PinList contains either PE pins or DPS pins or HV pins (don't mix types).
LS_PMU_VPAR_FORCE	None	-5V to +15V	**WARNING**: normal limit checks on PE pins are NOT enforced. See Pattern Loop Considerations. LSENABLE Pattern Instruction PinList contains either PE pins or DPS pins or HV pins (don't mix types).
LS_PMU_VPAR_HIGH LS_PMU_VPAR_LOW	RANGE1 RANGE2	-2.5V to +4V -5V to +15V	LSENABLE Pattern Instruction PinList contains either PE pins or DPS pins or HV pins (don't mix types).
LS_PMU_VCLAMP_POS	None	-5V to +16V	
LS_PMU_VCLAMP_NEG		-6V to +15V	

Table 4.17.1.2-1 Magnum 1/2/2x Test Pattern Set/Tweak Parameter List

Level Parameter Pattern Token	Range Options	Range	Notes
LS_PTU_IPAR_FORCE LS_PTU_IPAR_HIGH LS_PTU_IPAR_LOW	RANGE1 RANGE2 RANGE3 RANGE4 RANGE5 RANGE6 RANGE7	±2uA ±8uA ±32uA ±128uA ±512uA ±8mA ±32mA	LSENABLE Pattern Instruction PinList contains only PE pins. See Note:.
LS_PTU_VPAR_FORCE LS_PTU_VPAR_HIGH LS_PTU_VPAR_LOW		-2V to +12V	LSENABLE Pattern Instruction PinList contains only PE pins. See Note:.
LS_PTU_VCLAMP_POS		0.5V to +12V	LSENABLE Pattern Instruction PinList contains only PE pins. See Note:.
LS_PTU_VCLAMP_NEG	None	-2V to +11V	
LS_HV_VOLTAGE	None	0V to +28V	LSENABLE Pattern Instruction PinList contains only HV pins.
LS_HV_IPAR_HIGH LS_HV_IPAR_LOW	None	0mA to 8mA	
LS_HV_VPAR_HIGH LS_HV_VPAR_LOW	None	0V to +28V	
_SEL_RT_D0	None	n/a	See Setting a Static Pin-state using Level Sets.

Usage

```
% LEVELSET  OPERATION, LevelToken, RANGE, TSET#, PS#
  UDATA     VALUE UNITS
     and/or
  VUDATA    VALUE UNITS
  PINFUNC   VLEVELSET // Optional
```

where:

OPERATION determines whether the level is being SET or adjusted (TWEAK).

LevelToken identifies the target level being SET/TWEAKed. Legal values are listed in the table above.

RANGE is required when the test parameter has multiple operating ranges. Legal values are RANGE1 through RANGE6. **Careless programming can cause damage to the DUT!,** see Note:. Legal values are shown in Table 4.17.1.2-1 above.

TSET# is optional and, if used, specifies the time-set to be selected while the LEVELSET instruction executes. Legal values are TSET1 through TSET32. Default = TSET1. The cycle period of the time set used in a LEVELSET instruction must be >= 140nS. See Operation and Rules.

PS# is optional and, if used, specifies the pin scramble to be selected while the LEVELSET instruction executes. Legal values are PS1 through PS64. Default = PS1.

VALUE specifies the desired level value to be written. Units must be used (see Specifying Units). **VALUE** is specified as a UDATA operand value (Memory Test Patterns) or a VUDATA operand value (Logic Test Patterns). In mixed memory/logic patterns both UDATA and VUDATA may specify a value, however, to use the VUDATA value requires specifying PINFUNC VLEVELSET.

UNITS specifies a units macro to be applied to **VALUE**. Units must be used, see Specifying Units.

PINFUNC VLEVELSET is optional and only useful in Mixed Memory/Logic Patterns. See Description. If PINFUNC VLEVELSET is used in the LEVELSET instruction is must also be used in the prior LSENABLE instruction.

Example

See Example.

4.17.1.3 Setting a Static Pin-state using Level Sets

See Controlling Magnum 1 Levels from the Test Pattern, Controlling PE Levels from the Test Pattern

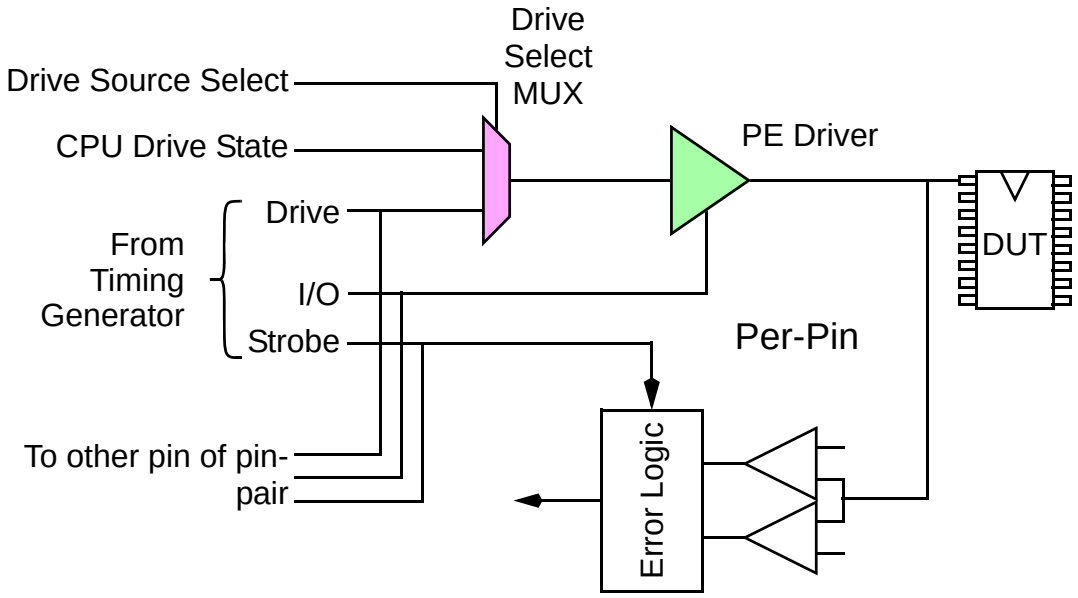
Description

This section documents how to use the Controlling PE Levels from the Test Pattern facility to set one or more tester signal pins to drive a static state (CPU Drive State), effectively disabling the drive signals from the timing system to the PE driver on those pins.

In general, using the test pattern Level Set pattern instructions noted below, selected pins may be set to drive a static logic state, as set in the test program prior to pattern execution. When in this state, these pins will not receive drive edges from their associated timing generators (strobe and I/O edges are not affected). Later, in the same pattern, normal operation can be restored using the same Level Set instructions.

The target application allows the test pattern to disable/enable selected signals to selected DUTs by disabling the drive high/low signals of key pins of each DUT and, instead, cause these pin(s) to drive a logic state set prior to executing the test pattern.

The following simplified diagram is used to describe this operation:



Normally, during pattern execution, the drive signal from a pin's timing generator passes through that pin's Drive Select MUX to control the pin's PE driver. In this situation, the Drive Source Select signal shown above selects the Drive signal from the timing generator, allowing normal operation.

The Drive Source Select signal is normally only changed in two situations:

- The `pin_dc_state_set()` function can be used, without executing a test pattern, to cause specified pins to drive high (VIH), low (VIL), super-voltage (VIHH) or to tri-state. Optionally, the `hold_state` argument can be used inhibit drive edges during subsequent test pattern executions. The `pin_dc_state_set()` function sets both the CPU Drive State signal and the Drive Source Select signal for the specified pin(s). Using `pin_dc_state_set()`, each pin of a given pin-pair can be controlled independently.

- The Drive Source Select signal for each pin can also be controlled from the test pattern, using the Controlling PE Levels from the Test Pattern facility (more below). This allows the test pattern to determine whether a given pin's driver receives the drive signals from its associated timing generator or is set to the CPU Drive State, as programmed using `pin_dc_state_set()` before the test pattern is executed. Note that strobe and I/O edges are not affected and that each pin of a given pin-pair can be configured separately.

To allow the test pattern to manipulate the Drive Source Select signal (correctly) requires two things:

- Execute `pin_dc_state_set()` (twice, more below) to configure the CPU Drive State signal for the target pins. This sets the desired static pin state (high/VIH, low/VIL, super voltage/VIHH or tri-state) prior to pattern execution.
- Add the appropriate Level Set instructions to the test pattern, to cause the Drive Source Select signal to switch between the normal timing generator drive signal and the CPU Drive State (this controls the Drive Select MUX selection). As noted below, these pattern instructions must include a delay to allow adequate time for the Drive Select MUX configuration to change before the test pattern depends on the change.

In the following example, `pin_dc_state_set()` is used to configure the pins in the pin list named `pl_WE`:

```
pin_connect( pl_WE  ); // Normally already in the test program
pin_dc_state_set( pl_WE, t_vih, TRUE );    // REQUIRED, more below
pin_dc_state_set( pl_WE, t_vih, FALSE );   // REQUIRED, more below
```

Note that the `pin_dc_state_set()` must be executed twice for each pin list:

- First execution, with the `hold_state` argument = `TRUE`, latches the CPU Drive State signal for the specified pin(s). In this example, the drive-high (VIH) state is specified (`t_vih`). But, specifying `hold_state` = `TRUE` also changes the Drive Source Select signal to select the CPU Drive State, which inhibits signals from the timing generators. To restore normal TG operation requires that `pin_dc_state_set()` be executed again...
- Second execution, with the `hold_state` argument = `FALSE`. This restores normal timing generator operation without changing the latched CPU Drive State value.

The following example test pattern causes the pins in the pin list named `pl_WE` to switch to the static driver state (CPU Drive State) set in the code above. In this state, these pins will not receive driver signals (edges) from their associated timing generators:

```
% LSENABLE  pl_WE             // Select target pins
% LEVELSET  SET, _SEL_RT_D0   // _SEL_RT_D0 = Drive Source Select
  UDATA     0                 // 0 = select CPU Drive State
% waitA:                      // Wait 2uS for pin config to
  COUNT     COUNT1, DECR, AON // actually change. Setup
  MAR       CJMPNZ, waitA     // COUNT1 accordingly
```

The following instructions restores normal operation for the pins in the pin list named `pl_WE`; i.e. to receive driver signals (edges) from their associated timing generators:

```
% LSENABLE  pl_WE             // Select target pins
% LEVELSET  SET, _SEL_RT_D0   // _SEL_RT_D0 = Drive Source Select
  UDATA     1                 // 1 = select TG drive signal
% waitB:                      // Wait 2uS for pin config to
  COUNT     COUNT1, DECR, AON // actually change. Setup
  MAR       CJMPNZ, waitB     // COUNT1 accordingly
```

Also note:

- It is the user's responsibility to restore normal driver operation (the system software does not do this). This can be done as shown in the second pattern example above or by executing `pin_dc_state_set()` with the `hold_state` argument = FALSE; i.e.

  ```
  pin_dc_state_set( pl_WE, t_vih, FALSE );
  ```

- Note that the test pattern must include a 2uS delay, to allow adequate time for the `LEVELSET` instruction to change the Drive Select MUX selection. In the examples above, this is shown as a separate delay instruction using counter `COUNT1`, however this delay can also be obtained using a long cycle period in the `LEVELSET` instruction. This delay is the user's responsibility.
- Using this feature only the `SET` option to the `LEVELSET` pattern instruction is valid (`TWEAK` is not valid).
- In Multi-DUT Test Programs the Active DUTs Set (ADS) determines which pin(s) are enabled at any given time.

4.17.1.4 changes_voltages()

See Controlling Magnum 1 Levels from the Test Pattern, Controlling PE Levels from the Test Pattern.

Description

The `changes_voltages()` function can be used to determine if a specified test pattern is Controlling PE Levels from the Test Pattern.

Usage

```
BOOL changes_voltages( Pattern *obj );
```

where:

`obj` identifies the target pattern.

`changes_voltages()` returns TRUE if the specified pattern (**`obj`**) is Controlling PE Levels from the Test Pattern, otherwise FALSE is returned.

Example

```
BOOL cv = changes_voltages( myPat );
if( cv == TRUE )  output(" myPat does control levels");
else              output(" myPat does NOT control levels");
```

4.17.1.5 level_set_value_change()

See Manipulating Tester Hardware, Controlling PE Levels from the Test Pattern.

Note: first available in software release h3.3.xx.

Description

The `level_set_value_change()` function is used to modify the DC parameter *value* of a specified target instruction in test patterns which set/tweak PE levels. See Controlling Magnum 1 Levels from the Test Pattern.

The following 2 example patterns (one memory, one logic) are used to describe the operation of `level_set_value_change()`:

```
PATTERN( myPat, memory)
% MAR     INC  // One or more memory instructions here
// ...
% target_label:
```

```
   LSENABLE     someDPSpins
%  LEVELSET     SET, LS_DPS
   UDATA        3.3 V

PATTERN( myPat, logic )
%  VEC  HL10X HL10X HL10X // One or more logic instructions here
// ...
%  target_label:
   VEC  HL10X HL10X HL10X
   LSENABLE     someDPSpins
%  VEC  HL10X HL10X HL10X
   LEVELSET     SET, LS_DPS
   VUDATA       3.3 V
```

In these examples, the LEVELSET instruction sets the primary DPS voltage (LS_DPS) to 3.3V, on the DPS identified by the someDPSpins pin list specified in the LSENABLE instruction. Operational details are documented in Controlling Magnum 1 Levels from the Test Pattern. Given these example patterns, the level_set_value_change() function may be used to change the 3.3V value to a different value. Subsequently, executing either pattern will set the DPS to the new value in these instructions.

Note the following:

- The obj argument to level_set_value_change() identifies the pattern containing the target pattern instruction to be modified. The pattern specified by obj must include at least one set of LSENABLE/LEVELSET instructions.
- The label and delta arguments identify the target pattern instruction to be modified, as follows:
  - The pattern instruction to be modified must be a LEVELSET instruction.
  - label is a Pattern Label attached to a pattern instruction preceding the target instruction.
  - delta specifies an offset from label and is used (required here) when the target pattern instruction does not have a label. In this application a non-zero delta value is always required because the LEVELSET instruction is not allowed to have a Pattern Label (to prevent accidentally branching to a LEVELSET, which must always be preceding by a LSENABLE instruction). Using either example pattern above, the target instruction can be identified with label = target_label and delta = 1.

- One version (overload) of `level_set_value_change()` includes a `range` arguement. This version MUST be used when modifying any parameter type which requires that a range argument be specified in the test pattern LEVELSET Pattern Instruction. The system type cannot enforce this. The table below lists these parameter types.

Note: abnormally high voltage or current values (positive and negative) may be generated if this rule is violated. Proper operation requires that the range specified using `level_set_value_change()` match the range specified in the test pattern (which must match the range currently programmed in the target hardware at the time the pattern executes, see LEVELSET Pattern Instruction). The system software cannot enforce this, the user is responsible!

The following table lists the parameter types which require that a `range` argument be specified, see previous note:

Level-set Parameter	Description
LS_DPS_CURRENT_HIGH LS_DPS_CURRENT_LOW	DPS Current Test Pass/Fail Test Limits
LS_PMU_IPAR_FORCE LS_PMU_IPAR_HIGH LS_PMU_IPAR_LOW	PMU Force Current - Warning see Note:. PMU Current Test Pass/Fail Test Limit PMU Current Test Pass/Fail Test Limit
LS_PMU_VPAR_HIGH LS_PMU_VPAR_LOW	DPS Voltage Test Pass/Fail Test Limits
LS_PTU_IPAR_FORCE LS_PTU_IPAR_HIGH LS_PTU_IPAR_LOW	PTU Force Current - Warning see Note:. PTU Current Test Pass/Fail Test Limit PTU Current Test Pass/Fail Test Limit

Usage

```
BOOL level_set_value_change(  Pattern *obj,
                              LPCTSTR label,
                              int delta,
                              double value );
```

```
BOOL level_set_value_change(  Pattern *obj,
                              LPCTSTR label,
                              int delta,
                              double value,
                              Range range );
```

where:

obj identifies the pattern to be modified. See Description.

label and **delta** identify the instruction to be modified. See Description.

value specifies the new value. Units are supported.

range specifies the range used to calculate the underlying hardware value written to the pattern. Legal values are of the Range enumerated type. This argument is required when the parameter type is included in the table above. The specified range must match the value specified in the target pattern instruction (which must match the range currently programmed in the target hardware at the time the pattern executes, see LEVELSET Pattern Instruction). See Note:.

`level_set_value_change()` returns TRUE if no errors occurred otherwise FALSE is returned.

Example

```
BOOL ok = level_set_value_change( myPat, "target_label", 1, 2.1 V );
```

Chapter 5 Redundancy Analysis (RA)

Note: the functions used to configure and use the Error Catch RAM (ECR), including for RA applications, were re-implemented and re-named for Magnum. See Error Catch RAM Software. This chapter only refers to the Magnum ECR functions.

Note: the legacy (Maverick-I/-II) Redundancy Analysis functions continue to operate as documented however, they have ZERO built-in parallel test support and no support for using Magnum's 64 B-pins (see Functional Pin-pairs).

DO NOT mix the Maverick RA functions with the Magnum RA functions: they DO NOT inter-operate.

For all new Magnum test programs, it is highly recommended that the new RA functions, documented in this section, be used.

The RA topic is organized into the following main sections:

- Overview and Concepts
  - RA Pseudo-Code Example
  - RA Data and Lists
  - RaErrorPosition
  - RA vs. Magnum 1/2 Parallel Test
  - Must-repair vs. Sparse-repair
- Spares For Repair
  - Spare Rows, Spare Columns
  - Per I/O Spares
  - Per-I/O Spare Mask
  - Rows-Used-Together(RUT), Columns-Used-Together(CUT)
  - Spare Segments
- RA Software
  - Too many functions to list here

- Magnum RA vs. Maverick-I/-II RA
  - Magnum vs. Maverick RA Functions

5.1 Overview and Concepts

This section includes a substantial overview of RA concepts. Also see RA Data and Lists, Must-repair vs. Sparse-repair, Logical vs. Physical, vs. Electrical Addresses.

Memory devices often contain spare circuits which can be used to replace (repair) defective memory elements during the manufacturing process. This repair is commonly called *redundancy repair*.

Prior to performing a redundancy repair, a *redundancy analysis* (RA) must be made which considers test failure data, the device architecture and available spare repair elements, rules about how these spare elements can be used, etc. This chapter documents the software used to perform the redundancy analysis, and how the necessary model of the device architecture, and available spare row/column elements, is described in user-written software.

Note: Nextest can not document the actual device repair process or software because each device is unique. The primary output of the Nextest RA software is a Repair List which is used by user-written software to actually repair the device. More below.

Redundancy Analysis (RA) reads and analyzes errors captured in the Error Catch RAM (ECR) during the execution of one or more functional memory test pattern(s). RA then generates 2 basic outputs:

- An overall result status (obtained using `ra_result_get()`); i.e. does the DUT need repair and is it repairable given the available spare resources?
- The Repair List containing specific repair details; i.e. which spare element(s) were allocated to repair the device.

To use RA, the user's test program must do the following (note, these steps are described in greater detail below):

1. Configure the ECR (typically once)
2. Configure the RA subsystem (typically once)

3. Define the DUT's segment (block) configuration (typically once)
4. Define the DUT's spare resources configuration (typically once)
5. Clear the ECR
6. Reset the RA Data and Lists
7. Execute one or more functional tests and log errors to the ECR
8. Invoke RA
9. Examine and use the RA results
10. Repeat steps 5. through 9. for each new set of DUT(s) being tested, OR, to re-test and re-repair DUT(s) which were previously repaired.

In Magnum Multi-DUT Test Programs, user code only needs to define the segment (block) architecture and spares architecture for a single DUT, the system software then manages the details separately for each DUT defined in the Pin Assignment Table. The parallel test architecture of the Magnum hardware and software allows optimized hardware operations and software RA to provide the performance needed in production test applications.

The following outline describes, in more detail, the configuration steps noted above (the RA Pseudo-Code Example implements this outline in even more detail):

- Execute `ecr_config_set()` to configure the ECR, normally to match the DUT's X/Y address and data architecture. This must be done before the DUT's RA configuration is defined (next) and before any functional tests which capture errors to the ECR are executed.
- Execute `ra_config_set()` to initialize the RA subsystem and specify various options.
- Use `ra_segment_make()`, one or more times, to define the DUT's RA Segment architecture. This tells the RA software the number and size of each memory segment (block) in the DUT. In Multi-DUT Test Programs, the system software duplicates the segments for each DUT in the program.
- Use `ra_spare_row_make()` and `ra_spare_col_make()` to create one or more spare row(s) and/or spare column(s).
- Use `ra_spare_add()` to formally associate each spare created with one or more segments. This creates the Spares List which, during RA, tracks which spares are available to make a repair. In Multi-DUT Test Programs, the system software duplicates the Spares List for each DUT in the program. Linked Segments may be created when a given spare is associated with more than one segment.

This normally completes the ECR and RA configuration. These steps are usually only performed once, typically in the Site Begin Block.

To actually perform an RA involves the following sequence, which will execute in Test Blocks executed by the Sequence & Binning Table:

- Execute `ecr_all_clear()` to delete errors from any previously tested DUTs from the ECR.
- Execute `ra_reset()` to reset the various RA lists (Error List, Spares List, Repair List, Unusable List, Bad Segment List) for all DUTs.
- Use `funtest()` with the `fullec` option to execute one or more functional memory test patterns and log errors to the ECR.
- Execute `ra_execute()` to perform the RA, which does the following:
  - Scans (reads) the Error Catch RAM (ECR) and caches the errors for each segment of each active DUT (each DUT in the Active DUTs Set (ADS)). This is done once, for all active DUT(s).
  - Individually analyze the errors for each segment of each active DUT, to determine the overall status of each DUT and generate the Repair List. User code may use Redundancy Call-back Functions to modify various analysis operations or replace Nextest-defined functionality with user-written code.
  - When a repairable failure is detected identify which spare element is to be used to make the repair and move it from the Spares List to the Repair List. This must consider which spares are currently available (Spares List). User code can use Redundancy Call-back Functions to enforce device-specific rules regarding how spares may be used or limited, move spares to the Unusable List, etc. More below.
  - For each active DUT, if an unrepairable segment is identified add it to the Bad Segment List and stop analyzing the DUT if the `max_bad_segments` limit is exceeded (see `ra_config_set()`).
- User code then performs the following for each active DUT:
  - Execute `ra_result_get()` to get the overall RA result status for the DUT.
  - If the status indicates the DUT needs repair and is repairable use various Repair List Functions to get the Repair List.
  - Either Repair the DUT or export the Repair List for off-line repair. This is device specific and cannot be done by the RA software.
  - If appropriate, use `ra_repair_done()` to update the Repair List in preparation for re-test and re-repair. This changes the state of any repairs in the Repair List from pending to done (see Repair List).

As noted above, the RA Pseudo-Code Example implements this outline in even more detail.

When all active DUT(s) have been processed, if appropriate, re-test and re-repair those DUT(s) which were just repaired. The process stops when:

- A given DUT has been fully repaired; i.e. it passes the appropriate tests.
- The DUT is determined to be unrepairable; i.e. defects remain and the device can't be repaired with the available spares resources.
- User code exits the process, for other reasons.

In the latter two cases the DUT is usually considered to be defective.

As mentioned above, actual device repair may be done *off-line*, typically using laser hardware and software developed by others, or *online* using user-written software in the test program performing the RA. When done off-line, the Repair List must be exported (via user-written code) to capture the repairs identified by the RA. It is important to note that actual repair is highly device specific, thus Nextest RA software is not used to actually repair the DUT.

Selecting a specific spare element to make a given repair can be a complex process since multiple repair options often exist and spare elements are often constrained, by the device design, in how they may be used to repair the device. For example:

- A single spare row and/or column may not be available. Spare rows (or columns) are often grouped such that only multiple rows (or columns) can be replaced as a unit. When this occurs, the spare elements can be defined as having 2 or more rows and/or columns *used together* (see Rows-Used_Together, Columns-Used_Together).
- Only a subset of addresses and/or data bits can be repaired using a given spare element. Per I/O Spares are used in the latter situation.
- Spare elements may be shared between segments, which may or may not create Linked Segments.
- There may be a larger number of spare rows vs. columns (or vice versa). Spare selection priority must be user programmable.
- ... Etc. ..

Note: most redundancy requirements can be addressed with just a few of the functions documented in this Redundancy Analysis (RA) chapter. Conversely, many of the functions documented here will only be used in those rare occasions when the user needs to customize portions of the RA operation. The documentation for these rarely used functions has a ***special note*** indicating that the function is not commonly used in mainstream redundancy applications. The following functions are commonly used to implement redundancy analysis and repair:

`ecr_config_set()` (see Error Catch RAM Software)
`ecr_all_clear()` (see Error Catch RAM Software)
`ra_config_set()`
`ra_segment_make()`
`ra_spare_row_make(), ra_spare_col_make()`
`ra_spare_add()`
`ra_execute()`
`ra_result_get()`
`ra_repaired_row_count_get(), ra_repaired_col_count_get()`
`ra_repaired_rows_get(), ra_repaired_cols_get()`
`ra_repair_done()`
`ra_reset()`
`ra_dump()` (during debug)

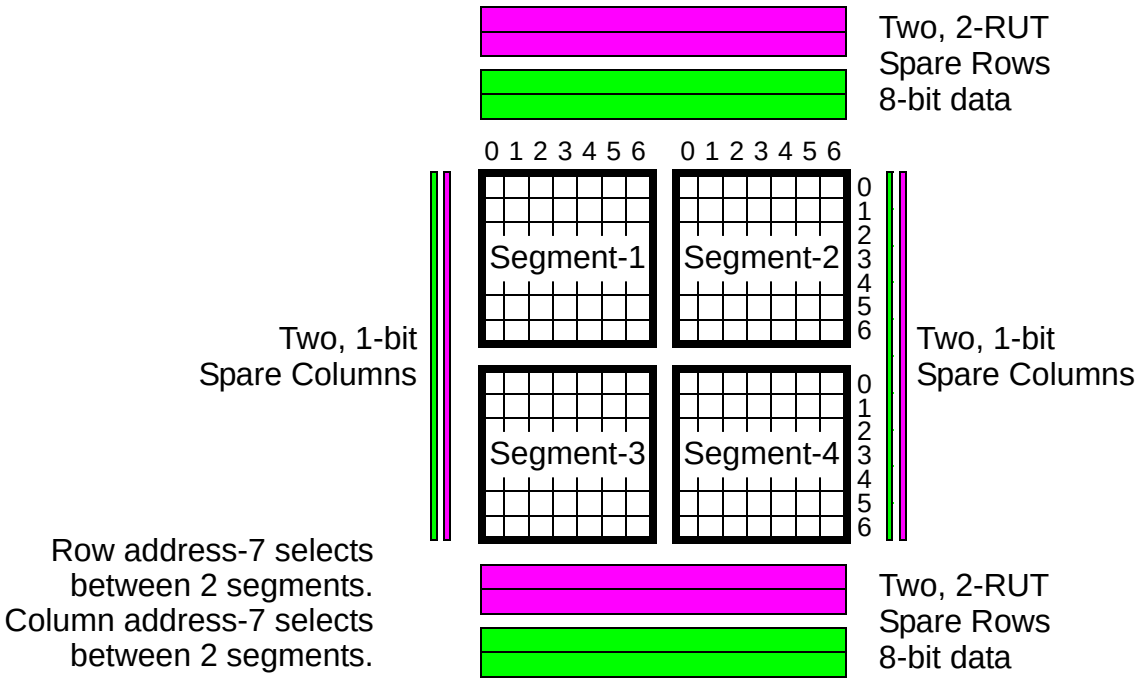
5.1.1 RA Pseudo-Code Example

See Redundancy Analysis (RA), Overview and Concepts.

Note: the detailed example below is also described in the Overview and Concepts section, in two levels of detail: very basic, more detailed.

In this example, the DUT has 8X and 8Y addresses, 8 data, organized in 4 segments, each with 128 rows x 128 columns. Each pair of segments have two 1-bit-wide per-I/O spare columns (see Per I/O Spares) and two spare rows, each spare consisting of 2 rows; i.e. each

row spare is a 2-RUT spare (see Rows-Used-Together(RUT), Columns-Used-Together(CUT)):



This example is divided into two sections:

- Code which is typically executed once to configure the ECR and RA software and to describe the DUT's segment and redundancy architecture.
- Code which is executed for each set of DUTs being tested; i.e. each time the Sequence & Binning Table is executed. This retrieves errors from the ECR, performs the RA, allocates spares for repair, etc.

The following code is normally executed only once. Consider doing this in the Site Begin Block:

```
// Setup the ECR in preparation for use
ecr_config_set( 8, 8, data_pins); // ecr_config_set()

// Initialize the RA subsystem to configure and use the
// ECR Mini-RAM. Must be done after ecr_config_set() and before any
// test patterns are executed to capture errors in the ECR
ra_config_set( TRUE, TRUE ); // ra_config_set()
```

```
// The DUT architecture has 4 segments, each 128 rows by 128
// columns. Define one DUT for RA; the system software will
// automatically create the segments for every DUT in the program.
RaSegment S1, S2, S3, S4;
S1 = ra_segment_make( 0, 127, 0, 127 ); // ra_segment_make()
S2 = ra_segment_make( 0, 127, 128, 255 );
S3 = ra_segment_make( 128, 255, 0, 127 );
S4 = ra_segment_make( 128, 255, 128, 255 );

// Columns are PerIO spares, 1 bit wide. seg_length of '2' is
// required to denote that the spare covers 2 segments
RaSpareCol col1 = ra_spare_col_make(0x1, 2);
RaSpareCol col2 = ra_spare_col_make(0x1, 2);
RaSpareCol col3 = ra_spare_col_make(0x1, 2);
RaSpareCol col4 = ra_spare_col_make(0x1, 2);
// Add the spare columns to the appropriate segments
ra_spare_add(S1, col1);
ra_spare_add(S3, col1);
ra_spare_add(S1, col2);
ra_spare_add(S3, col2);
ra_spare_add(S2, col3);
ra_spare_add(S4, col3);
ra_spare_add(S2, col4);
ra_spare_add(S4, col4);

// Spare rows are 2-RUT spares. mask = 0 specifies full width spare.
// seg_length of 2 is required to denote that the spare covers 2
// segments. numRUT of 2 says this spare replaces 2 rows. Leaving
// blanksBetweenRUT at the default value denotes that the 2 rows
// replaced are adjacent to each other.
RaSpareRow row1 = ra_spare_row_make(0, 2, 2);
RaSpareRow row2 = ra_spare_row_make(0, 2, 2);
RaSpareRow row3 = ra_spare_row_make(0, 2, 2);
RaSpareRow row4 = ra_spare_row_make(0, 2, 2);
// Add the spare rows to the appropriate segments
ra_spare_add(S1, row1);
ra_spare_add(S2, row1);
ra_spare_add(S1, row2);
ra_spare_add(S2, row2);
ra_spare_add(S3, row3);
```

```
ra_spare_add(S4, row3);
ra_spare_add(S3, row4);
ra_spare_add(S4, row4);
```

The pseudo-code below is typically executed each time the Sequence & Binning Table executes. Comments indicate where preparations are made to re-test/re-repair those DUTs which were repaired:

```
// Reset the RA lists (Error List, Spares List, Repair List,
// Unusable List, Bad Segment List) in preparation for testing a
// new set of DUTs, and clear all previous errors from the ECR.
// Consider doing this in a Before-testing Block.
ra_reset();
ecr_all_clear();

// Execute other test blocks as desired...

// Setup DC, AC, DPS etc. for functional test(s) and RA.

// Execute one or more memory pattern(s) to test the active DUT(s)
// and log errors to the ECR
funtest( myPattern1, fullec ); // funtest()
funtest( myPattern2, fullec ); // Etc... errors accumulate in ECR

// Perform RA to retrieve and cache errors from the ECR for each
// DUT in the Active DUTs Set (ADS), then analyze the errors for
each
// active DUT.
ra_execute(); // Arguments not shown, see ra_execute() for details

// Check the RA result for each active DUT
{ // Extra braces required to set ActiveDutIterator scope
ActiveDutIterator duts; // For each active DUT...
while ( duts.More() ) {
   // Get the RA status result for this DUT
   RaResult status = ra_result_get();

   DutNum d = active_dut_get(); // For printing only
   switch( status ){
      case t_ra_good :
         output(" t_dut%d had NO ERRORS, needs no repair", d );
      break;
      case t_ra_unrepairable :
         output(" t_dut%d had errors, is NOT REPAIRABLE", d );
      break;
      case t_ra_repairable :
```

```
            output(" t_dut%d had errors, IS REPAIRABLE", d );
            DoRepairHere();// User code to repair this device or export
                           // the Repair List for off-line repair
            ra_repair_done(); // Prepare for re-test/re-repair
            if( verbosity == 999 ) ra_dump();
          break;
          case t_ra_not_analyzed :
            output(" ERROR: ADS was modified after the RA was done");
          break;
        }
      } // while(duts.More())
      } // Extra braces required to set ActiveDutIterator scope

      // Re-test and re-repair as appropriate.
```

Note the following:

- The actual repair of each DUT must be done by user-written code, represented above by `DoRepairHere()`. If the repair is done off-line, or by another test program, this code must export the Repair List for later use (see `ra_repaired_row_count_get()`, `ra_repaired_col_count_get()`, `ra_repaired_row_get()`, `ra_repaired_col_get()`, `ra_repaired_rows_get()`, `ra_repaired_cols_get()`).
- `ra_repair_done()` moves spares from the pending Repair List to the done Repair List, leaving the done Repair List available to see what was already repaired. This is only required to prepare for re-test and re-repair of the same set of DUTs.
- The `verbosity` variable above is user-defined (elsewhere) and represents the fact that `ra_dump()` is not normally used during production test.

5.1.2 RA Data and Lists

See Redundancy Analysis (RA), Overview and Concepts.

The RA software uses several lists to manage and track errors, repairs and spare elements. In general, separate lists are maintained for each RA Segment. And, in Multi-DUT Test Programs, the RA software automatically creates and maintains these lists for each DUT in the program.

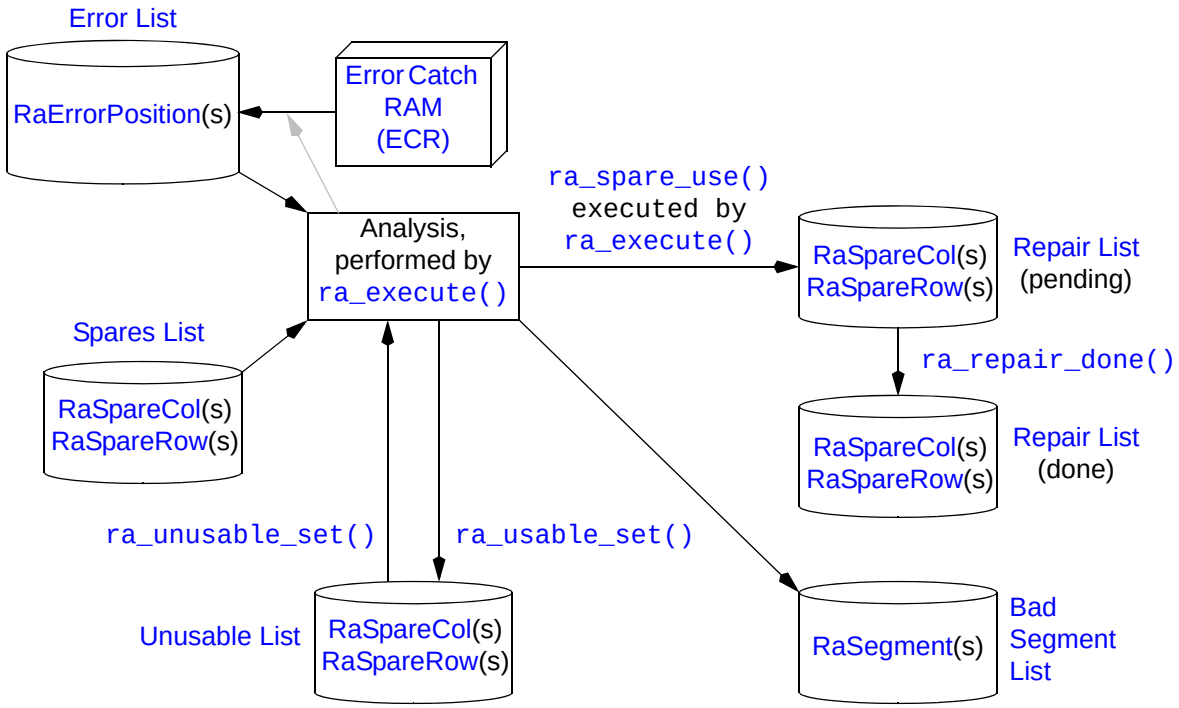
- **`Error List`**: in this document, for convenience and clarity, the term `Error List` is used to represent the errors scanned (read) from the Error Catch RAM (ECR) and cached for each segment of each active DUT. These are the errors actually analyzed during the RA. This list is populated by `ra_execute()`, using `ra_scan_area_callback()` or the user's call-back if an RaScanAreaCallbackFunc Call-back Function is registered. The `Error List` is cleared using `ra_reset()` or `ra_segment_reset()` (`ra_reset()` calls `ra_segment_reset()` for all segments).
- **`Spares List`**: the list of spare elements (`RaSpareRow`s and/or `RaSpareCol`s) which are currently available to repair each segment. At any given time, these are the *unused* spares which have not been moved to the **`Repair List`** or the **`Unusable List`**. The **`Spares List`** for a given segment is implicitly created by `ra_spare_add()`, which associates the spare rows/columns created using `ra_spare_row_make(), ra_spare_col_make()` with one or more segment(s).
- **`Repair List`**: the main output of redundancy analysis (RA); i.e. a list of the spare rows and/or columns allocated, during RA, to repair each segment. The **`Repair List`** actually stores a list of `RaSpareRow`(s) and/or a list of `RaSpareCol`(s). Each `RaSpareRow` identifies both the spare row allocated for a repair and records the row it replaces. When the DUT supports Per I/O Spares, the spare also records the Per-I/O Spare Mask identifying what data bits the spare replaces. The same applies to `RaSpareCol` vs. columns. During RA, spares are moved from the **`Spares List`** to the **`Repair List`** by the `ra_spare_use()` function which is called, as necessary, by `ra_execute()`. It is the **`Repair List`** which user code either uses to actually repair the device or exports to perform off-line repair. To access the Repair List, to repair a DUT or to export the list, user code uses `ra_repaired_row_count_get(), ra_repaired_col_count_get()` and `ra_repaired_row_get(), ra_repaired_col_get()` or `ra_repaired_rows_get(), ra_repaired_cols_get()`. The `ra_what_repaired_row_get(), ra_what_repaired_col_get()` functions may be used to identify which spare was allocated to replace a specified row or column. Spare elements in the **`Repair List`** have two states:
  - **`Pending`** - identified repairs not yet consummated
  - **`Done`** - completed repairs retained for reference

  Once repairs are complete, before the device is retested, if the test strategy allows additional repairs the `ra_repair_done()` function is used to change the state of the spares in the Repair List from *pending* to *done*.
- **`Unusable List`**: a list of spare elements which have not been used but become unusable due to enforcement of the DUT's redundancy architectural rules. This is done by the system software when dealing with shared spares (see Linked

Segments) and may also be done using Redundancy Call-back Functions. Spare elements are moved between the **`Spares List`** and **`Unusable List`** using the `ra_usable_set()` and the `ra_unusable_set()` functions. `ra_reset()` and `ra_segment_reset()` move spares from the **`Unusable List`** back to the **`Spares List`**.

- **`Bad Segment List`**: a list of segments which have been determined to be unrepairable. See Spare Segments.

The following diagram shows how RA information is moved between the different lists:



`ra_reset()` clears all lists, but not the ECR, in preparation for testing new DUT(s)

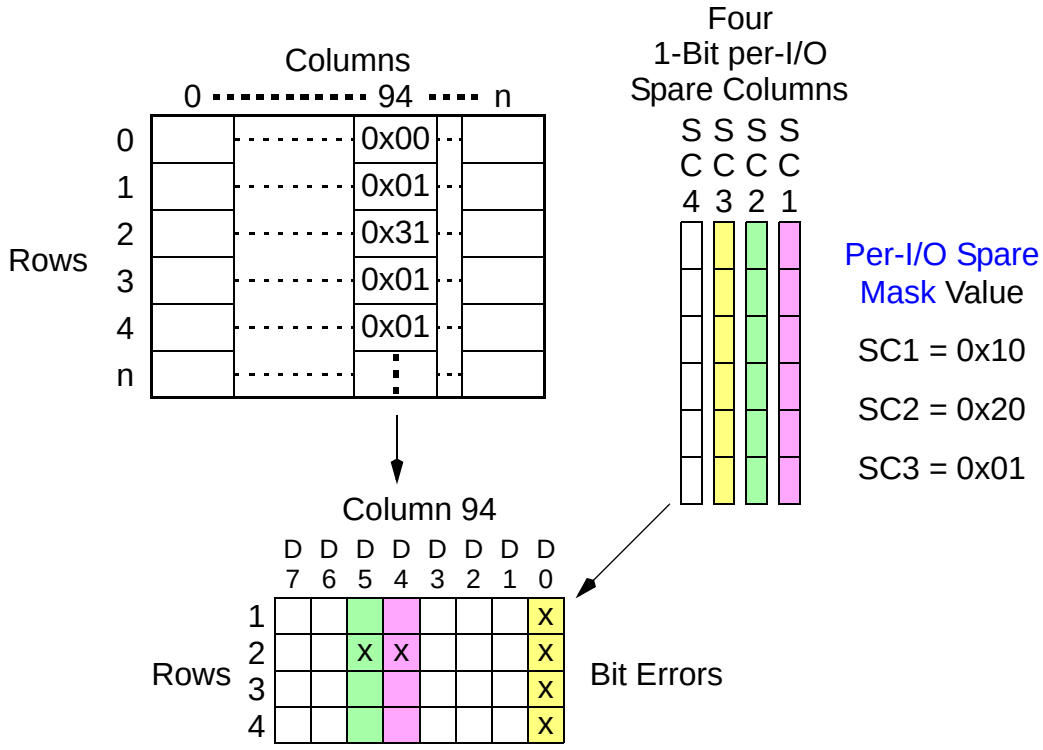
5.1.3 RaErrorPosition

See Redundancy Analysis (RA), Overview and Concepts.

The Magnum RA software supports Per I/O Spares i.e. spare row(s) and/or spare column(s) which replace a subset of a given row's or column's data bits.

If the RA software did not support Per I/O Spares, a given repair could be specified using an integer to identify which spare row or spare column is allocated to replace a given bad row or

bad column. However, Per I/O Spares require that the RA software also identify which data bits a given spare row or spare column will replace. For example, consider a segment which has four 1-bit per-I/O spare columns and a segment with the errors in column 94 shown below:



In this scenario, three 1-bit per-I/O spare columns are required to repair the segment:

- Spare column SC1 is allocated to replace bit-4 i.e. mask = 0x10
- Spare column SC2 is allocated to replace bit-5 i.e. mask = 0x20
- Spare column SC3 is allocated to replace bit-0 i.e. mask = 0x01

Therefore, the repair made by a given spare requires specifying two components:

- A bad row or column number; in the previous example column 94.
- A Per-I/O Spare Mask; in the previous example 0x31.

In the Magnum RA software, these are represented as an `RaErrorPosition`:

```
typedef struct RaErrorPosition {
   int       rcnum;    // Bad row or column number
   __int64   mask;     // Mask position within that row or col
} RaErrorPosition;
```

By definition, an RaErrorPosition stores two things:

- A (potentially bad) row or column e.g. the RaErrorPosition rcnum value.
- A bit-mask e.g. the RaErrorPosition mask value. See Per-I/O Spare Mask.

An RaErrorPosition is used in the following contexts:

- Each spare row (RaSpareRow) and spare column (RaSpareCol) stores an RaErrorPosition, recording the bad row or column and data bits the spare is allocated to replace. More below.
- Some RA functions require an RaErrorPosition as an input argument or return an RaErrorPosition, or both.
- Arrays of RaErrorPositions (i.e. RaErrorPosArray) are also used as input to or returned as output from some RA functions. More below.

Note that the errors stored in the Error List are not represented as an RaErrorPosition. However, all of the RA functions which retrieve errors from the Error List return error information only as an RaErrorPosition or RaErrorPosArray.

As noted above, each spare row (RaSpareRow) and spare column (RaSpareCol) stores an RaErrorPosition, targeted at recording the bad row/column and data bits the spare will replace. These operate as follows:

- When a spare is initially created (see ra_spare_row_make(), ra_spare_col_make()), its RaErrorPosition rcnum value = -1 and RaErrorPosition mask = the initial Per-I/O Spare Mask value of the spare.
- Subsequently, but before the spare is actually allocated for a repair (by ra_spare_use() via ra_execute()), the rcnum and mask values may remain as initialized or may reflect transient values assigned by various analysis functions (see ra_what_repaired_row_get(), ra_what_repaired_col_get()) This applies to all spares in the Spares List and Unusable List.
- When a spare is allocated for repair it is moved from the Spares List to the Repair List. At that time, the spare's RaErrorPosition rcnum value records which bad row or column the spare was allocated to repair and the RaErrorPosition mask value records which data bits the spare will replace (see Per-I/O Spare Mask).

- Some RA functions require an `RaErrorPosition` or an `RaErrorPosArray` as input, via an argument of the same type. In most cases, the value(s) supplied will be obtained using an RA function which returns an `RaErrorPosition` or `RaErrorPosArray`; i.e. user code will not normally be required to explicitly identify `rcnum` or `mask` value(s).
- In this document, the term `RaErrorPosition` is used in context to refer to error(s) obtained, using the various RA functions, from the Error List. This is done to emphasize that, using Magnum RA software, an error usually consists of the two component parts noted above.
- When a segment has Per I/O Spares, a given bad row or column may be represented by more than one `RaErrorPosition`, each with a different Per-I/O Spare Mask.
- When a segment does not have Per I/O Spares, a given bad row or column will be represented by a single `RaErrorPosition`, with a Per-I/O Spare Mask reflecting all data bits of the DUT.
- When a segment has Per I/O Spares, and every bit in a given row or column has an error, the number of `RaErrorPosition`(s) needed to represent that row or column will match the number of valid Per I/O Spares masks for the spares which can be used to replace it
- Non-per-I/O spares are treated as per-I/O spares that replace the full data width of the bad row or bad column.
- Row `RaErrorPosition`s are usually different than column error positions for the same error address. This is especially true when the row-vs-column spares have a different data-width.
- RA functions which analyze and return repair results return values in the terms of `RaErrorPosition`s.
- The `ra_spare_current_mask_set(), ra_spare_current_mask_get()` functions may be used to set and get a spare's current `RaErrorPosition` `mask` value of a spare row or column. As indicated above, this is normally not necessary (for advanced users only).
- The `ra_spare_mask_count_get()` and `ra_spare_mask_get()` functions can be used to sequentially get each legal mask value for a given spare.

5.1.4 RA vs. Magnum 1/2 Parallel Test

See Redundancy Analysis (RA), Overview and Concepts, Magnum 1, 2 & 2x Parallel Test.

The Magnum Test System software formally supports parallel test; i.e. concurrently testing multiple DUTs in parallel. The general features and operation are described in Magnum 1, 2 & 2x Parallel Test. Issues specific to Redundancy Analysis (RA) are described below.

The following information applies to Multi-DUT Test Programs; i.e. test programs which define more than one DUT in the Pin Assignment Table:

- Except as noted below, all *setter* functions operate on all DUT(s) in the Active DUTs Set (ADS) and all *getter* functions return a value from the first DUT in the Active DUTs Set (ADS). This is the general paradigm for all Nextest functions in Multi-DUT Test Program.
- `ra_config_set()` should be executed once regardless of the number of DUT(s) defined in the Pin Assignment Table. The Active DUTs Set (ADS) and Ignored DUTs Set (IDS) have no effect on `ra_config_set()`.
- User code must create/define segments for a single DUT, using `ra_segment_make()`. The system software then automatically duplicates the segment configuration for each DUT in the Pin Assignment Table. The segment (`RaSegment`) returned by `ra_segment_make()` is from the first DUT in the Active DUTs Set (ADS) but may be used elsewhere to identify the segment for any DUT, with the Active DUTs Set (ADS) determining which DUT(s) are affected or accessed.
- User code must create/define spare row(s) and/or spare column(s) for a single DUT, using `ra_spare_row_make()` and `ra_spare_col_make()`. The system software then automatically duplicates these spare(s) for each DUT in the test program. The spare row (`RaSpareRow`) returned by `ra_spare_row_make()` and the spare column (`RaSpareCol`) returned by `ra_spare_col_make()` is from the first DUT in the Active DUTs Set (ADS) but may be used elsewhere to identify a spare, for any DUT with the Active DUTs Set (ADS) determining which DUT(s) are affected or accessed.
- User code associates (adds) spare row(s) and/or spare column(s) to a segment using `ra_spare_add()`. The system software automatically duplicates these spare-vs-segment associations for each DUT in the Pin Assignment Table. The `RaSegment` returned by `ra_segment_make()` is used to identify the target segment and the spare row (`RaSpareRow`) returned by `ra_spare_row_make()` or spare column (`RaSpareCol`) returned by `ra_spare_col_make()` identifies the target spare.
- The `ra_execute()` function performs RA on all DUTs in the Active DUTs Set (ADS).

- The `ra_reset()` and `ra_segment_reset()` functions have an optional argument (`all_duts`) used to specify whether its operation applies to all DUTs in the Pin Assignment Table (default) or only to the DUT(s) in the Active DUTs Set (ADS).
- The Redundancy Call-back Functions operate identically for all DUTs in the Pin Assignment Table; i.e. it is not possible to register Redundancy Call-back Functions on a per-DUT basis. It is, however, possible to change which function is registered and thus executed at any given time.
- `ra_usable_set()` and `ra_unusable_set()` move the specified spare for the first DUT in the Active DUTs Set (ADS). This is unconventional as compared to most other setter functions.
- `ra_repair_done()` moves repairs in the pending Repair List to the done Repair List for the first DUT in the Active DUTs Set (ADS).

5.1.5 Must-repair vs. Sparse-repair

See Redundancy Analysis (RA), Overview and Concepts.

`ra_execute()` first scans (reads) errors from the ECR and caches them in the Error List, separately for each segment of every active DUT. Rather than store all errors scanned (read) from the ECR the RA software performs a preliminary analysis as the errors are read, caching only the minimum number of errors necessary, thus both reducing the amount of data stored for each DUT and improving performance.

`ra_execute()` then performs the detailed redundancy analysis on each DUT individually in two phases:

- Must-repair
- Sparse-repair

Must-repair is performed initially and has two goals:

- Identify non-repairable DUTs as early as possible, to reduce time wasted testing bad devices.
- Identify any rows or columns which can *only* be repaired one way; i.e. using only a spare row or only a spare column.

For example, if the RA detects 5 failures in a particular row (by definition in 5 distinct columns) if only three spare columns are available it is not possible to repair the device using spare columns. This repair *must* use a spare row and thus is logged as a Must-repair

row. A spare row is selected to make the repair and moved from the Spares List to the Repair List, and the associated failures are removed from the analysis (removed from the Error List). Must-repair columns are treated similarly.

Once the initial Must-repair phase has completed, only failures which were not identified as Must-repair remain in the Error List. Provided the device is still repairable, these remaining failures will be further analyzed by the Sparse-repair algorithm.

Only failures not previously identified as Must-repair are considered during the Sparse-repair evaluation. However, since the consumption of a spare element changes the conditions used to detect a Must-repair row or column, Must-repair is performed again any time a Sparse-repair is identified.

A more detailed description of the entire analysis and repair process is outlined in the `ra_execute()` function subsection.

5.2 Spares For Repair

See Redundancy Analysis (RA), Overview and Concepts, RA Software.

This section covers the following topics:

- Spare Rows, Spare Columns
- Per I/O Spares
- Per-I/O Spare Mask
- Rows-Used-Together(RUT), Columns-Used-Together(CUT)
- Spare Segments

5.2.1 Spare Rows, Spare Columns

See Redundancy Analysis (RA), Overview and Concepts, Spares For Repair.

Spare rows and/or spare columns and/or Spare Segments, also called spare elements or just spares, are the device specific resources available to repair a defective DUT.

In the RA software, one or more spare row(s) and/or column(s) are defined/created using `ra_spare_row_make(), ra_spare_col_make()`. To be used during RA, each spare

must be associated with one or more segments, using `ra_spare_add()`. Note that Spare Segments are treated differently, without formal creation and require user code support.

Then, during the RA performed by `ra_execute()`, specific spares are identified to repair (replace) each defective row and/or column. If the RA consumes all the spares before a DUT is completely repaired that DUT is considered unrepairable; i.e. bad.

As indicated, each spare is associated with one or more specified segments. In some device architectures a given spare spans more than one segment. When this occurs the associated segments are considered *linked*. Linked Segments change how the RA actually analyzes errors and allocates spares since the spare actually affects multiple segments. When a spare causes Linked Segments, using that spare will, by definition, replace the same row(s) or column(s) in all segment(s) to which it is linked.

A spare can be shared between segments without linking those segments. This means that the spare can be used to repair one of the segments which share it. This is described in more detail in Linked Segments.

In some DUT architectures, a single spare row or column *element* may replace more than one actual row or column. For example, when a single spare row element replaces 4 actual rows, the RA software will treat this as 4 *rows used together*, or RUT. Similarly, columns used together are called CUT. See Rows-Used-Together(RUT), Columns-Used-Together(CUT). To operate properly, the RA software must be aware of RUT and/or CUT features in the DUT architecture. This is specified when each spare is created, using `ra_spare_row_make(), ra_spare_col_make()`. Note that ECR address compression is not required to support use of RUTs and/or CUTs (unlike Maverick-I/-II).

The Magnum RA software supports Per I/O Spares; i.e. spare row(s) and/or column(s) which replace some but not all data bits. The support for Per I/O Spares is the reason Per-I/O Spare Masks exist.

Each spare records an RaErrorPosition; i.e. which row or column it is replacing and the Per-I/O Spare Mask which records which data bits it is replacing. Each spare also has a record of the number of rows/columns it replaces (i.e., RUTs and CUTs). When a given spare resides in the Spares List or Unusable List, this information is not very useful; however, after the spare is allocated for a repair and moved to the Repair List, the RaErrorPosition records what the spare repaired. It is the Repair List which user code accesses to actually repair the DUT.

5.2.2 Per I/O Spares

See Overview and Concepts, Spare Rows, Spare Columns, Spares For Repair.

Per-I/O spare

As indicated in Spares For Repair, some device architectures have spare rows and/or columns that replace less than a segment's full data width. In this document and in the RA software these are called *per-I/O spares*.

Except as noted, per-I/O spares have the same attributes as regular spares:

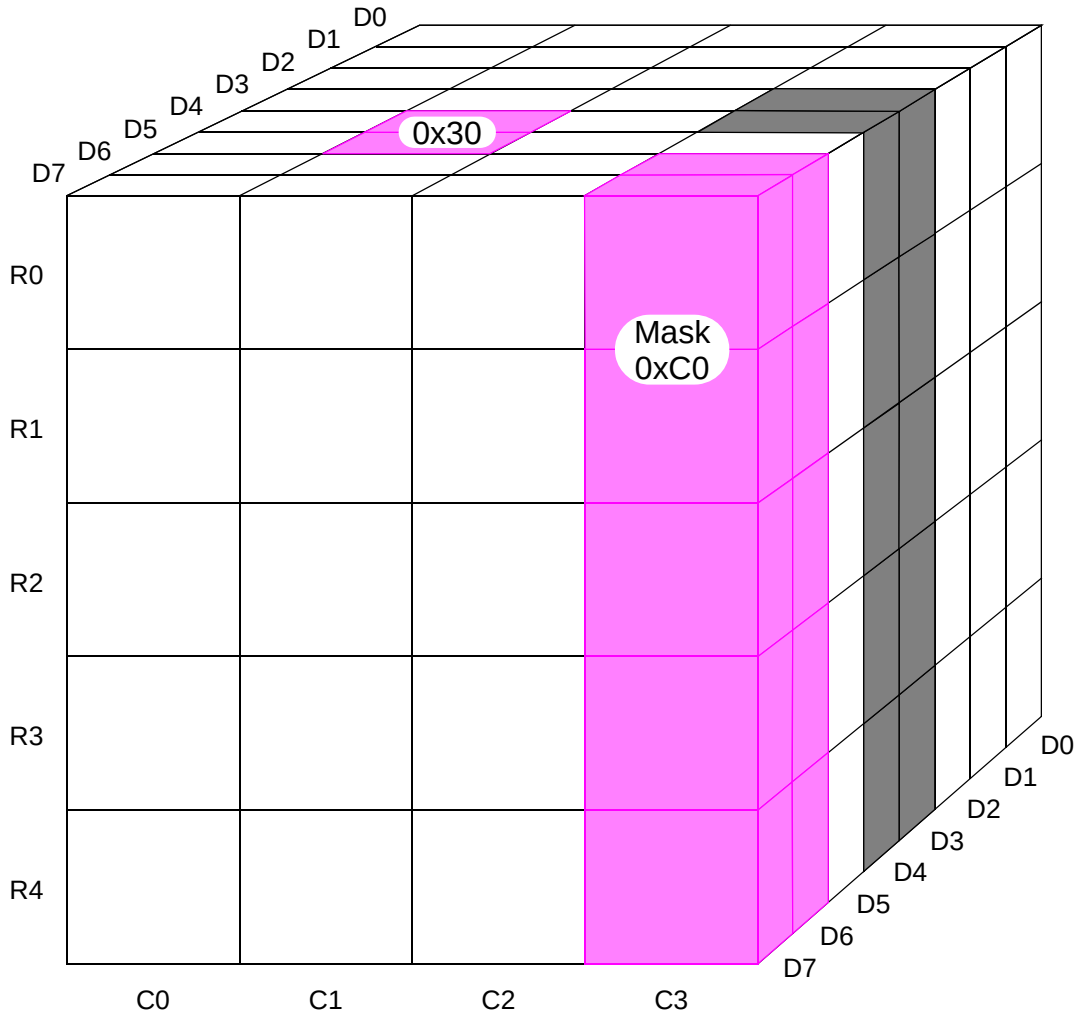
- A DUT can have per-I/O spare rows, per-I/O spare columns or both.
- All spares are created using the same methods; i.e. `ra_spare_row_make()`, `ra_spare_col_make()`. The only difference between per-I/O spares and non-per-I/O spares is the Per-I/O Spare Mask definition for a given spare. A spare with a Per-I/O Spare Mask that includes the full data-width of a segment is a non-per-I/O spare. Conversely, any spare with a Per-I/O Spare Mask that doesn't include the full data-width of the segment is a per-I/O spare.
- Per-I/O spares can span segments, which creates Linked Segments. Per-I/O spares can be shared between segments (see Linked Segments).
- Per-I/O spares can be columns-used-together (CUTs) and/or rows-used-together (RUTs). See Rows-Used-Together(RUT), Columns-Used-Together(CUT).
- All spare rows associated with a given segment must all the same data width. Similarly, All spare columns associated with a given segment must all the same data width. This means that it is not legal, in the same segment, to have a spare row 16 bits wide, for example, and a second spare row 1 bit wide. Different segments, however, may have different per-I/O widths.

Per-I/O spares have attributes and usage rules which do not apply to regular spares:

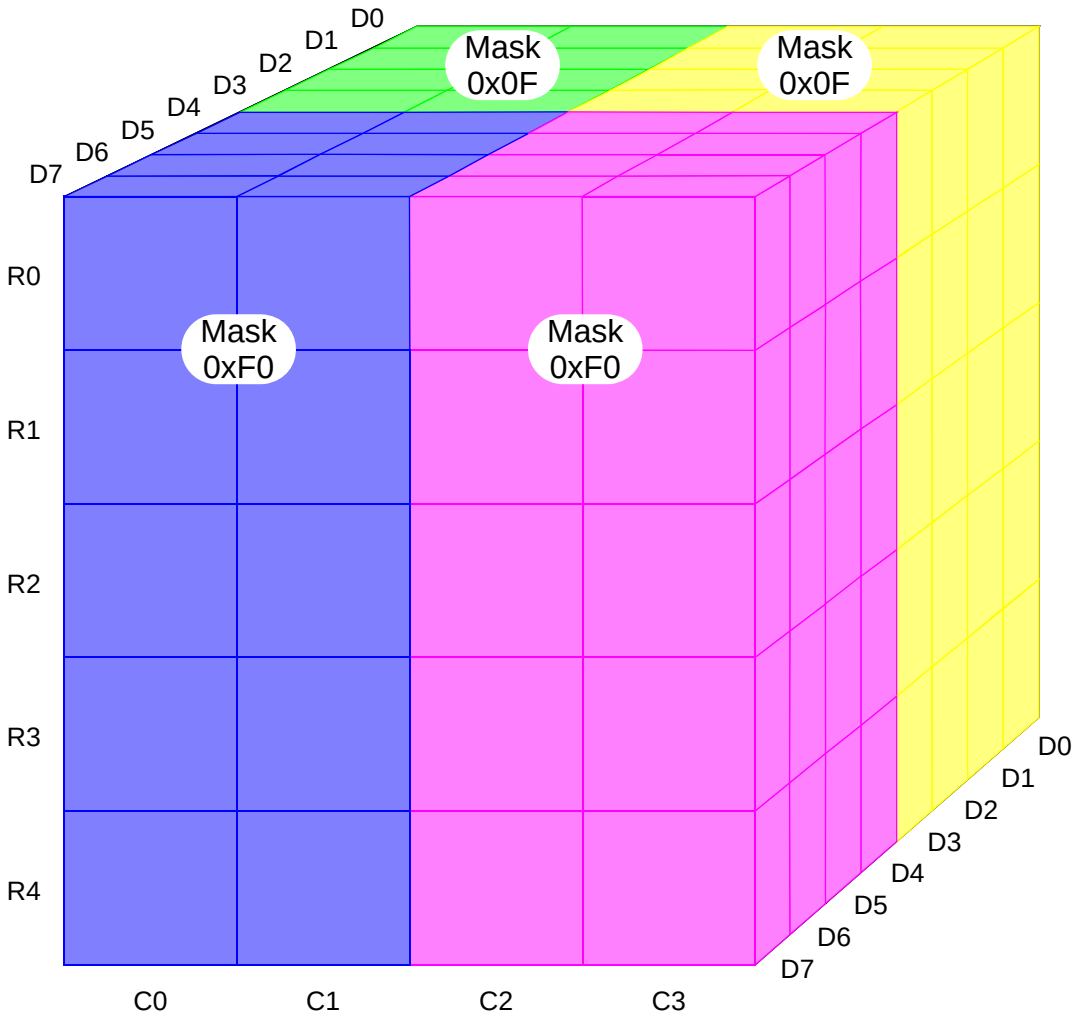
- Per-I/O spares may be 1 or more bit(s) wide; i.e. multi-bit per-I/O spares are supported.
- Multi-bit per-I/O spares need not replace adjacent data bits. These are called non-adjacent multi-bit per-I/O spares. For example, a per-I/O spare 2 bits wide could have 1 bit in between the replaced bits. Given a DUT with an 8-bit data width, this spare could be used to replace bits D0+D2 or D1+D3 or D4+D6 or D5+D7. In another example, given a device with 2 per-I/O spares each 4 bits wide with a 1-bit gap, one spare can replace bits D0, D2, D4 and D6, the other spare can replace bits D1, D3, D5 and D7.

- All bits replaced using a multiple-bit per-I/O spare must reside within one row address or one column address. For example, using a 4 bit wide per-I/O spare column to replace column-17 all 4 bits must reside in column-17 and no bits may reside in column-16 or column-18. For a device with this same spare architecture but having 2 columns-used-together (CUT = 2) both columns of the CUT would replace same bits.
- As indicated above, normally, a given per-I/O spare can potentially be used to replace different sets of data bits at a given address. This can be visualized as shifting the spare to cover different sets of data bits. A *mask* is used to specify (and record) which bits are replaced by a given per-I/O spare. See Per-I/O Spare Mask and RaErrorPosition.
- By default, a one-bit-wide per-I/O spare can replace any data bit of the segment with which it is associated. Other per-I/O spare configurations have additional constraints. For example, the diagram below shows how a per-I/O spare column with 2 adjacent bits might be used to make 3 different repairs:
  - The repairs shown in magenta are valid; i.e. D7/D6 of column C3 and D5/D4 of column C1. Many other valid repairs are also possible but are not shown for clarity. The mask value for both repairs is shown.

- The repair shown in grey is invalid: a two-adjacent-bit per-I/O spare can only replace D0/D1 or D2/D3 or D4/D5 or D6/D7 but not D4/D3:



- The following example shows the valid repairs possible using a 2-CUT (see Rows-Used-Together(RUT), Columns-Used-Together(CUT)) 4-bit per-I/O spare column:



- By default, the RA software requires that a given per-I/O spare row must be able to replace any legal set of bits (more below) of a bad row. The same rule applies to spare columns. Any device-specific limitations on spare usage, including per-I/O spares, must be handled by a user-written RaRowUseOK or RaColUseOK call-back function (see Redundancy Call-back Functions).

5.2.3 Per-I/O Spare Mask

See Per I/O Spares, Redundancy Analysis (RA), Overview and Concepts.

As indicated in Per I/O Spares, a given per-I/O spare may be used to replace different combinations of data bits at a given address. For example, a single data bit per-I/O spare could replace D0 or D1 or D2, etc. For a device with 8-bit data width, this one spare could potentially make 1 of 8 different repairs.

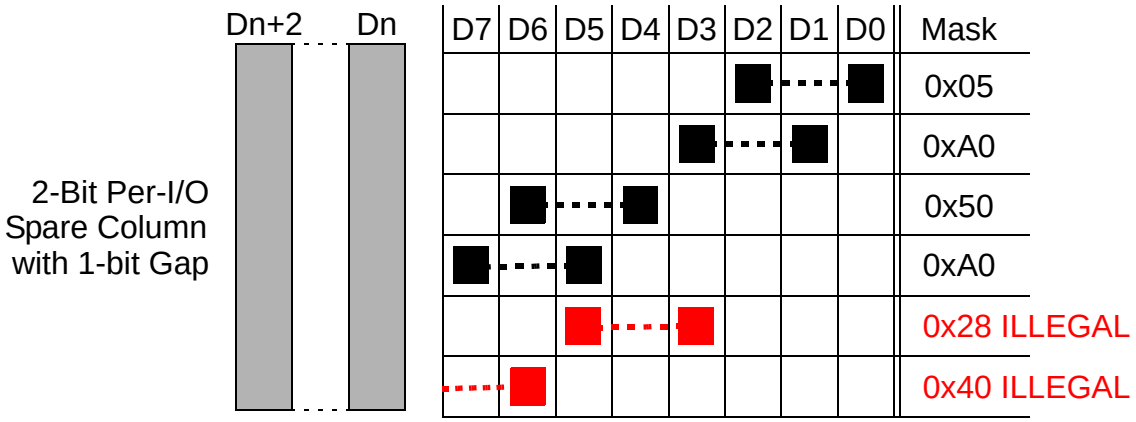
Each spare has an integral RaErrorPosition, which records the bad row or bad column the spare replaces and which data bits the spare replaces. During the RA, when a given per-I/O spare is allocated to make a repair, the RA software sets the spare's RaErrorPosition `rcnum` value to the bad row or column it will replace and the spare's RaErrorPosition `mask` value to the data bits the spare will replace. The mask value is more generally called a per-I/O spare mask.

As indicated, the per-I/O spare mask records which data bits the spare will replace. This can be visualized as shifting the spare to repair the identified data bits with the mask representing both the shifted position and which bits are being replaced.

Note the following:

- The data bits that a given spare is able to replace are specified using the mask argument to `ra_spare_row_make(), ra_spare_col_make()`. This sets the spare's initial RaErrorPosition mask value.
- A per-I/O spare replacing a single data bit will have an initial mask of 0x1. If the spare is subsequently used to replace the LSB data bit (D0) its mask will remain 0x1. If the spare is used to replace the D1 its mask will shift left to 0x2. If the spare is used to replace the D2 its mask will shift left twice to 0x4. Etc. The mask is manipulated during the RA as each spare is allocated to make a given repair.
- A per-I/O spare replacing 2 adjacent data bits will have an initial mask of 0x3. For a DUT with 8-bit data it can be shifted to 0x0C, 0x30 and 0xC0.
- A per-I/O spare replacing 4 adjacent data bits will have an initial mask of 0xF. For a DUT with 16-bit data it can be shifted to 0x000F, 0x00F0, 0x0F00 and 0xF000.
- Etc.

- For example, a per-I/O spare 2-bits wide with a 1-bit gap between the replaced bits is used to repair a DUT with an 8-bit data width:



  In this example, the spare can make one of 4 repairs: D0+D2, D1+D3, D4+D6 and D5+D7. The table above shows the mask for each possible repair. The RA performed by `ra_execute()` will identify both which spare to use to make a given repair and, when a per-I/O spare is selected, will set that spare's RaErrorPosition `mask` value to record which data bit(s) are being repaired by the spare.

- When a multi-bit per-I/O spare replaces non-adjacent bits several rules apply (which actually apply to all Per I/O Spares):
  - The RA software expects that the spare may be used in any mask position. Any device-specific limitations on spare usage, including per-I/O spares, must be handled by a user-written RaRowUseOK or RaColUseOK call-back function (see Redundancy Call-back Functions).
  - The bits replaced by a given spare cannot overlap bits of any other spare. For example, in the example above, a mask of 0x28 is not legal because D3 is already covered by the mask 0xA0.
  - All bits replaced by a given spare must remain within the data width of the segment being repaired. In the example above, the mask 0x40 is illegal because the upper data bit is outside the data width of the ECR address.

The RA software knows which mask positions are valid for each spare. For example:

Initial Mask Value	Segment's Data Width	Valid Mask Positions
0x1	8	0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80
0x3	8	0x03, 0x0C, 0x30, 0xC0
0x5	8	0x05, 0x0A, 0x50, 0xA0
0x11	16	0x0011, 0x0022, 0x0044, 0x0088, 0x1100, 0x2200, 0x4400, 0x8800
0x55	16	0x0055, 0x00AA, 0x5500, 0xAA00

In this document, the phrase *mask position* is frequently used. A mask position is defined to be one of the valid mask values for each spare created. Thus, the second table entry above has 4 valid mask positions: `0x03`, `0x0C`, `0x30`, `0xC0`.

Each spare row and column records an RaErrorPosition:

- When a spare is initially created (see `ra_spare_row_make()`, `ra_spare_col_make()`), its RaErrorPosition `rcnum` value = -1 and RaErrorPosition `mask` = the initial mask value of the spare.
- Subsequently, but before the spare is actually allocated for a repair (by `ra_spare_use()` via `ra_execute()`), the `mask` values may remain as initialized or may reflect transient values assigned by various analysis functions. This applies to spares in the Spares List and Unusable List.
- When a spare is allocated for repair it is moved from the Spares List to the Repair List. At that time, the spare's RaErrorPosition `rcnum` value records which bad row or column the spare was allocated to repair and the RaErrorPosition `mask` value records which data bits the spare will replace.

Note that spares which are not per-I/O spares will have a mask equal to the data width of the DUT and this mask value will never change.

5.2.4 Rows-Used-Together(RUT), Columns-Used-Together(CUT)

See Spares For Repair, Overview and Concepts, Redundancy Analysis (RA).

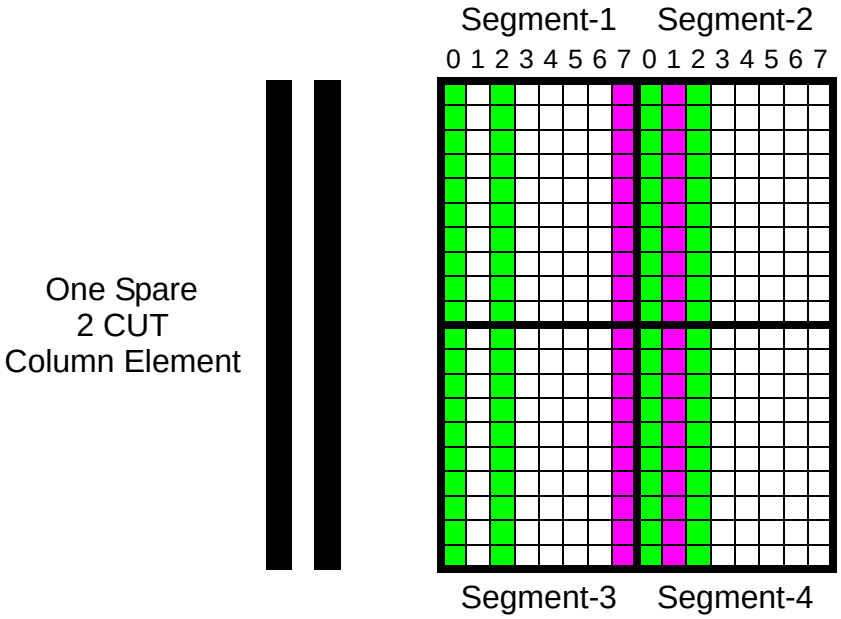
```
Rows-Used_Together    RUT
Columns-Used_Together CUT
```

The redundancy architecture of some devices include spare elements which replace more than one row or column. Thus, if a given row or column must be replaced, the repair actually replaces multiple rows or multiple columns. The terms *rows-used-together* (RUT) and *columns-used-together* (CUT) are used when the DUT design includes spare element(s) which replace multiple rows or columns.

To properly perform the repair analysis, the RA software must be aware of each spare's RUT or CUT attributes. Note the following:

- The cardinality of all spare rows associated with a given segment must be identical (see `ra_spare_add()`). Likewise for spare columns. In this context, *cardinality* refers to the number of rows or columns replaced by a RUT or CUT. Within a given segment, RUTs and CUTs may not be intermixed with individual spare rows and columns.
- `ra_spare_row_make(), ra_spare_col_make()` have optional arguments used to specify the number of RUT rows or CUT columns; i.e. the number of rows or columns the spare being created will replace.
- The second and later rows in a RUT and/or columns in a CUT need not be immediately adjacent to the first. Additional arguments to `ra_spare_row_make(), ra_spare_col_make()` are used to specify the number of non-replaced rows or columns between each replaced row or column; i.e. the gap between the RUTs or CUTs.
- When the RUT spare or CUT spare includes a gap, the RA software requires that the rows or columns be evenly spaced; i.e., with the same number of unreplaced rows or columns between each row or column.
- The rows replaced by a RUT spare cannot extend vertically into an adjacent segment. The columns replaced by a CUT spare cannot extend horizontally into an adjacent segment. For example, the diagram below shows a 2 CUT spare which, during the RA, links Segment-1 with Segment-3 and Segment-2 with Segment-4. For example clarity only, this spare column element has a 1-bit gap between the replaced columns. This spare can be used to replace columns 0+2,

1+3, 4+6 or 5+7 in any segment. The example shows 2 repairs in green which are legal. The repair shown in magenta is illegal because the spare columns can't cross horizontally between segments:



- There is no maximum numbers of rows allowed in a RUT or columns in a CUT. However, declaring a RUT spare or CUT spare with a very large number of rows or columns as a substitute for creating Spare Segments will impact RA performance.
- Rows or columns that are part of a RUT or CUT may not be used individually. The RA software considers a RUT and CUT to be a single replaceable unit.
- The number of rows or columns in a RUT or CUT and their configuration must be *regular*; i.e. the RUT must evenly and completely *tile* the segment. For example:
  - No RUT placement can overlap any part of another RUT placement. Similarly for CUTs. For example, in a 2-row RUT where the rows are adjacent, this one RUT can replace either row addresses 0+1, OR 2+3, OR 4+5, etc. This RUT may not replace addresses 1+2, OR 3+4, etc.
  - All rows or columns in a segment must be replaceable by a valid RUT or CUT placement. For example. it is illegal for a RUT to replace 3 adjacent rows in a segment containing 16 rows, because row 16 is not covered by a possible RUT placement.

- RUTs and CUTs can be Per I/O Spares. When a RUT or CUT is a per-I/O spare, the Per-I/O Spare Mask position for the spare applies identically to all rows or columns in the spare. Stated another way, to replace, for example, D5 in a row with a RUT spare, D5 will be replaced in each row replaced by that RUT spare.

5.2.5 Spare Segments

See Spares For Repair, Overview and Concepts.

A spare segment replaces an entire rectangle (block) of addresses instead of just replacing row(s) or column(s).

In memory devices, spare segments are generally implemented in two different forms:

- Some devices (e.g., NOR flash) have actual spare segments that are used to physically replace a bad segment, much like a spare row is used to replace a bad row, etc.
- Some devices (e.g., NAND flash) are designed with extra segments. In these devices a bad segment is not actually replaced. Instead, the device stores a list of bad segments which are subsequently not used, much like the bad block list used in a disk drive. In these devices, as long as a sufficient number of good segments exist the device is considered good.

The Magnum RA software supports spare segments somewhat differently than spare rows and columns, as follows:

- `ra_config_set()` has an optional argument (`max_bad_segs`) used to specify the number of unrepairable segments allowed before the DUT is declared unrepairable (bad). The default value of 0 means no bad segments are allowed; i.e., the DUT is declared unrepairable when the first unrepairable segment is identified. For devices which have spare segment support a non-zero `max_bad_segs` value may be specified.
- `ra_max_bad_segments_set()` function can also be used to set the value of the number of unrepairable segments allowed. This is targeted at re-test/re-repair applications and changes the value temporarily. See Note:. The `ra_max_bad_segments_get()` function may be used to get the current `max_bad_segs` value.

- During the RA (`ra_execute()`) if, at any time, it is determined that a defective segment cannot be repaired using the available spare rows and/or columns the RA analysis tags the segment as unrepairable. When the DUT has no spare segment support (i.e. `max_bad_segs` = 0) the RA stops because the DUT is not repairable.
- If the DUT does have spare segment support; i.e. `max_bad_segs` > 0:
  - RA software adds the unrepairable segment to the Bad Segment List.
  - If the `max_bad_segs` limit has been reached, the RA for the DUT stops.
  - If any shared spares (see Linked Segments) had been allocated to repair the bad segment they are returned to the Spares List. Note that this cannot be done for spares which link segments (see Linked Segments).
- After RA is complete, user code can access the Bad Segment List to retrieve any bad segments and act accordingly. Note that this is the extent of spare segment support; i.e. it is up to user code to determine how to use the contents of the Bad Segment List.
  - The `ra_bad_segments_count_get()` function can be used to determine the number of the bad segment(s) in the Bad Segment List.
  - The `ra_bad_segment_get()` function can be used to get a bad segment from the Bad Segment List.

  These functions are useful during a test-repair-retest cycle, particularly if repair is performed off-line.

5.3 RA Software

See Redundancy Analysis (RA).

This section includes the following RA topics:

- Types, Enums, etc.
- RA Configuration - must do this first!
- RA Segment - define DUT's segment architecture
- RA Spares - define spare rows & columns and associate with segment(s)
- RA Execution And Results - perform RA and get the overall result
- Repair List Functions - use or export the detailed RA results
- Redundancy Call-back Functions - replace built-in operations for special needs

- Magnum vs. Maverick RA Functions - for migration reference only

---

5.3.1 Types, Enums, etc.

See Redundancy Analysis (RA), RA Software.

Description

The following data and enumerated types are used in Redundancy Analysis (RA) applications:

Functions, MACROs & Keyword

Usage

The `RaSegment` data type is used in the RA Software to represent a segment. This is an opaque object.

The `RaSpareRow` data type is used in the RA Software to represent a spare row. This is an opaque object.

The `RaSpareCol` data type is used in the RA Software to represent a spare column. This is an opaque object.

The `PointFailureArray` is used to store one or more errors (a `PointFailure`) read from the Error Catch RAM (ECR):

```
struct PointFailure {
   DWORD row, col;
   __int64 data;
};
typedef CArray< PointFailure, PointFailure & > PointFailureArray;
```

As indicated, the `PointFailureArray` is based on the `CArray` class, defined by Microsoft. The various `CArray` member functions can be used to access a `PointFailureArray`. Use the MSDN on-line documentation.

The `RaSpareRowArray` is used to store one or more spare rows (`RaSpareRow`):

```
typedef CArray< RaSpareRow, RaSpareRow & > RaSpareRowArray;
```

As indicated, the RaSpareRowArray is based on the CArray class, defined by Microsoft. The various CArray member functions can be used to access a RaSpareRowArray. Use the MSDN on-line documentation.

The RaSpareColArray is used to store one or more spare columns (RaSpareCol):

```
typedef CArray< RaSpareCol, RaSpareCol & > RaSpareColArray;
```

As indicated, the RaSpareColArray is based on the CArray class, defined by Microsoft. The various CArray member functions can be used to access a RaSpareColArray. Use the MSDN on-line documentation.

The RaErrorPosition struct is a key structure used throughout the RA software. see RaErrorPosition:

```
typedef struct RaErrorPosition {
   int       rcnum;    // Row or column number
   __int64   mask;     // Mask position within that row or col
} RaErrorPosition;
```

The RaErrorPosArray is used to store one or more error positions (RaErrorPosition):

```
typedef CArray< RaErrorPosition, RaErrorPosition& > RaErrorPosArray;
```

As indicated, the RaErrorPosArray is based on the CArray class, defined by Microsoft. The various CArray member functions can be used to access a RaErrorPosArray. Use the MSDN on-line documentation.

The RaSpareRowPosition and RaSpareColPosition structs are used to store a combination of bad row or column and the mask indicating which data bits are defective plus a spare row (RaSpareRow) or column (RaSpareCol) being considered to replace the bad row or column:

```
typedef struct RaSpareRowPosition {
   int          rownum;   // Bad row
   __int64      mask;     // Bad row mask
   RaSpareRow   row;      // Spare row to repair rownum
} RaSpareRowPosition;
typedef struct RaSpareColPosition {
   int          colnum;   // Bad column
   __int64      mask;     // Bad column mask
   RaSpareCol   col;      // Spare column to repair colnum
} RaSpareColPosition;
```

The RaSpareRowPosArray is used to store one or more spare row positions (RaSpareRowPosition). The RaSpareColPosArray is used to store one or more spare column positions (RaSpareColPosition)::

```
typedef
CArray< RaSpareRowPosition, RaSpareRowPosition& > RaSpareRowPosArray;
typedef
CArray< RaSpareColPosition, RaSpareColPosition& > RaSpareColPosArray;
```

As indicated, the RaSpareRowPosArray and RaSpareColPosArray are based on the CArray class, defined by Microsoft. The various CArray member functions can be used to access a RaSpareRowPosArray and RaSpareColPosArray. Use the MSDN on-line documentation.

The RaResult enumerated type is used as the overall RA result returned by ra_result_get():

```
enum RaResult {t_ra_good,
               t_ra_repairable,
               t_ra_unrepairable,
               t_ra_not_analyzed };
```

5.3.2 RA Configuration

See Redundancy Analysis (RA), Overview and Concepts, RA Software.

Before an RA can be performed, the RA software must be configured. Various options are controlled or retrieved using arguments to the following functions:

- ra_config_set()
- ra_config_get()

5.3.2.1 ra_config_set()

See Overview and Concepts, RA Software, RA Configuration.

Note: this function is commonly used in redundancy applications. See Note:

Description

The `ra_config_set()` function is used to configure the RA Software before use. Note the following:

- `ra_config_set()` must be executed after the Error Catch RAM (ECR) is configured (using `ecr_config_set()`).
- `ra_config_set()` must be executed before any other RA-related functions are executed.
- Arguments to `ra_config_set()` are used to select various options which affect both how the ECR hardware is used and configured and how various RA operations are performed.
  - `use_miniram` specifies whether the ECR Mini-RAM is used when performing the RA. The ECR Mini-RAM was created primarily to make RA faster. By configuring the ECR Mini-RAM to match the segment architecture of the DUT, the ECR Mini-RAM will indicate exactly which segment(s) in the DUT contain errors and thus need to be analyzed during RA. The ECR's Row RAM or Column RAM are not used when the ECR Mini-RAM is used, and vice-versa.
  - `autoconfig_mini` specifies whether the user wants the system software to automatically configure the ECR Mini-RAM to match the segment architecture of the DUT (as defined using `ra_segment_make()` one or more times). Setting `autoconfig_mini` TRUE will overwrite any previous execution of `ecr_miniram_config_set()`. The ECR Mini-RAM is actually configured during the first `funtest()` execution after `ra_config_set()`; thus all segments must be defined before `funtest()` is executed after `ra_config_set()`.
  - `max_bad_segs` specifies the number of bad segments allowed before a DUT is declared unrepairable. See Spare Segments.
  - The `scan_func` argument registers a user-written RaScanAreaCallbackFunc Call-back Function call-back function which will execute during the RA to scan errors from the main ECR array and add them to the Error List. If used, this replaces the built-in method (`ra_scan_area_callback()`) with user-written code. Setting this argument to 0 causes the built-in method to be used (and un-registers any previously registered user-written call-back).
  - The `scan_rc_func` argument registers a user-written RaScanRCFunc call-back function which will execute during the RA to scan errors from the ECR's Row RAM or Column RAM. These RAMs are only used when the ECR Mini-RAM is not enabled (see above). If registered, this replaces the built-in method with user-written code. Setting this argument to 0 causes the built-in method to be used (and un-registers any previously registered user-written call-back).

- In Multi-DUT Test Programs, `ra_config_set()` should be executed once, regardless of the number of DUT(s) defined in the Pin Assignment Table. The Active DUTs Set (ADS) and Ignored DUTs Set (IDS) have no effect on `ra_config_set()`. See RA vs. Magnum 1/2 Parallel Test.
- If any of the following RA configuration options need to be modified, it is necessary to re-execute `ra_config_set()` first:
  - Segment definitions (`ra_segment_make()`)
  - Spare definitions (`ra_spare_row_make(), ra_spare_col_make()`)
  - Association of spare(s) with segment(s) (`ra_spare_add()`)
  - Segment definitions (`ra_spare_add()`)

  Note that executing `ra_config_set()` resets all of these definitions, thus it is necessary to completely redefine these parameters after executing `ra_config_set()`.

Usage

```
void ra_config_set(
            BOOL use_miniram DEFAULT_VALUE( TRUE ),
            BOOL autoconfig_mini DEFAULT_VALUE( FALSE),
            int max_bad_segs DEFAULT_VALUE( 0),
            RaScanAreaCallbackFunc scan_func DEFAULT_VALUE( 0),
            RaScanRCFunc scan_rc_func DEFAULT_VALUE( 0) );
```

where:

`use_miniram` is optional and, if used, determines whether the ECR Mini-RAM is used (TRUE) or not used (FALSE, default) when performing the RA.

`autoconfig_mini` is optional and, if used, specifies whether the system software automatically configures the ECR Mini-RAM (TRUE) to match the segment architecture of the DUT or not (FALSE, default). If FALSE is specified user code must execute `ecr_miniram_config_set()` to configure the ECR Mini-RAM. The **`use_miniram`** value must be TRUE for **`autoconfig_mini`** TRUE to be useful.

`max_bad_segs` is optional and, if used, specifies the number of bad segments allowed before a DUT is declared unrepairable. Default = 0 = no bad segments are allowed; i.e. a DUT is declared unrepairable when the first unrepairable segment is identified. This value is set for all DUTs in the test program; i.e. the Active DUTs Set (ADS) and Ignored DUTs Set (IDS) are ignored.

`scan_func` is optional and is used to register a user-defined RaScanAreaCallbackFunc function. See Description and Redundancy Call-back Functions.

scan_rc_func is optional and is used to register a user-defined RaScanRCFunc call-back function. See Description and Redundancy Call-back Functions.

Example

The following example enables the use of the ECR Mini-RAM and causes the system software to automatically configure the ECR Mini-RAM to match the segment architecture of the DUT:

```
ra_config_set( TRUE, TRUE );
```

5.3.2.2 ra_config_get()

See RA Segment, Redundancy Analysis (RA)

Note: this function is not used in most redundancy applications. See Note:

Description

The ra_config_get() function is used to retrieve the current value of the parameters set up using ra_config_set().

These parameters are not set per-DUT; i.e. in Multi-DUT Test Programs the Active DUTs Set (ADS) as no effect on this function.

Usage

```
void ra_config_get(
            BOOL* use_miniram DEFAULT_VALUE( 0),
            BOOL* autoconfig_mini DEFAULT_VALUE( 0),
            int *max_bad_segs DEFAULT_VALUE( 0),
            RaScanAreaCallbackFunc* scan_func DEFAULT_VALUE( 0),
            RaScanRCFunc* scan_rc_func DEFAULT_VALUE( 0) );
```

where:

use_miniram is optional and, if specified, must be the address of an existing BOOL variable used to return whether the ECR Mini-RAM is being used during the RA performed by ra_execute(). Specify 0 if this value is not needed.

autoconfig_mini is optional and, if specified, must be the address of an existing BOOL variable used to return whether the ECR Mini-RAM was automatically configured by the system software to match the segment architecture of the DUT (TRUE) or not (FALSE). This value is undefined if **use_miniram** returns FALSE. Specify 0 if this value is not needed.

max_bad_segs is optional and, if specified, must be the address of an existing int variable used to return the number of bad segments allowed before a DUT is deemed unrepairable. Specify 0 if this value is not needed.

scan_func is optional and returns a pointer to an RaScanAreaCallbackFunc call-back function (see Redundancy Call-back Functions). Returns 0 if the user has not registered a RaScanAreaCallbackFunc Call-back Function. Specify 0 if this value is not needed.

scan_rc_func is optional and returns a pointer to a RaScanRCFunc call-back function (see Redundancy Call-back Functions). Returns 0 if the user has not registered a RaScanRCFunc. Specify 0 if this value is not needed.

Example

The following example retrieves the current autoconfig_mini and scan_rc_func values:

```
BOOL auto;
RaScanRCFunc scanrc;
ra_config_get( 0, &auto, 0, 0, &scanrc );
```

5.3.3 RA Segment

See Redundancy Analysis (RA), RA Software.

In the context of the DUT, a *segment* represents a physical block of addresses. Commonly, memory devices are partitioned into physical segments (blocks) where a subset of rows and/or columns and/or data bits are visibly (i.e. under a microscope) or logically distinct. A DUT will typically either have 1 segment or some power-of-2 number of segments.

In the context of the RA software a *segment* represents a rectangular block of contiguous addresses which are a separately repairable subcomponent of the DUT; i.e. a segment is an area of the DUT which has its own spare repair elements. It is common for redundant (spare) elements to, by design, be constrained to repairing less than the whole DUT; i.e. a segment. It is also possible for spares to be shared between multiple segments and to create Linked Segments.

In order to properly perform a redundancy analysis (RA) the DUT's segment architecture must be defined, using `ra_segment_make()` once for each segment in the DUT.

Note: the RA software requires that at least one segment be created.

Some DUTs have Spare Segments, which doesn't affect the DUT's RA segment description and are thus discussed as a separate topic.

During RA (`ra_execute()`), each segment will be analyzed separately and will have spare resource allocation tracked separately. When Linked Segments exist, the RA software adjusts the analysis and spare allocation accordingly.

The functions directly related to segment are:

Always Used	Rarely Used (see Note:)
`ra_segment_make()`	`ra_segment_config_get()`
	`ra_segment_count_get()`
	`ra_segment_get()`
Used When DUT Has Spare Segments	`ra_segment_id_get()`
`ra_bad_segments_count_get()`	`ra_segment_lookup()`
`ra_bad_segment_get()`	`ra_segment_linkage_count_get()`
	`ra_max_bad_segments_set()`
	`ra_max_bad_segments_get()`

5.3.3.1 Linked Segments

See RA Segment, Redundancy Analysis (RA).

Definitions:

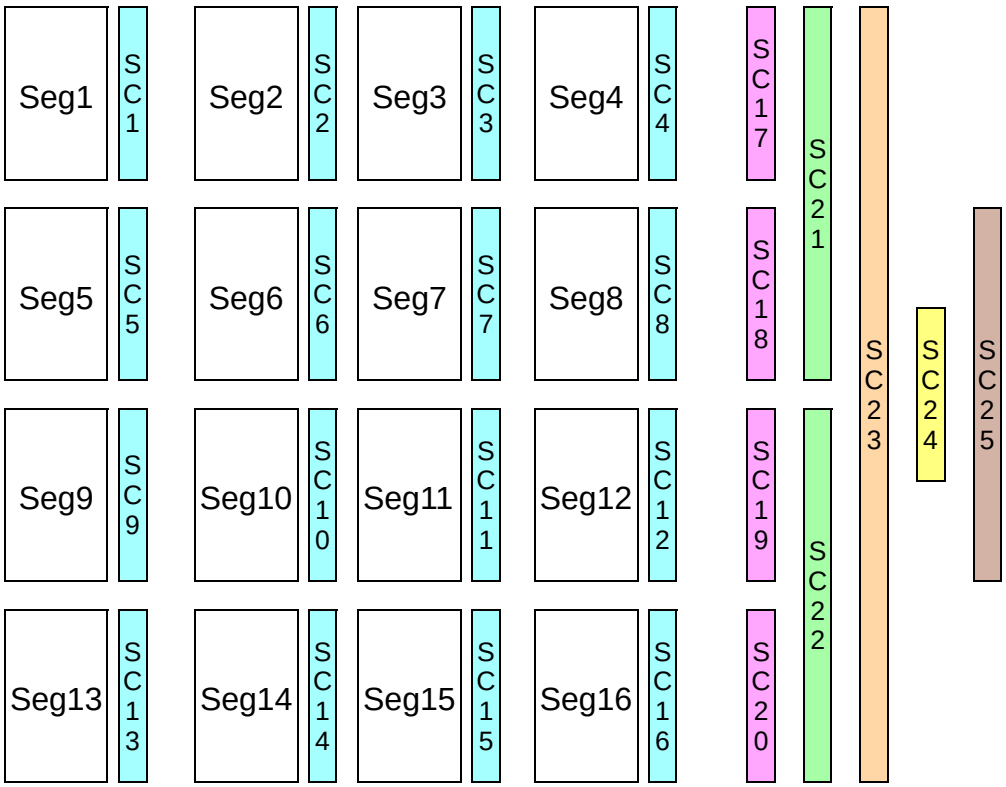
- When a given spare (row or column) will replace parts of more than one segment that spare *links* those segments i.e. linked segments are created.
- A *shared* spare is a spare which can replace a part of one of possibly several segments; i.e. that spare is *shared* between those segments.
- It is possible for a given spare row or spare column to be shared and to link segments

The distinction between *linked* and *shared* is important:

- Both concepts are fundamentally based on the DUT architecture; i.e. how the DUT's spare rows and/or spare columns may be applied to repair one or more DUT segments.
- Proper RA operation requires that linked segments and shared spares be clearly defined.

A given spare can be added to more than one segment (see `ra_spare_add()`). When this is done, it is the *length* of the spare which differentiates between *linked* and *shared*.

The `seg_length` argument to `ra_spare_row_make(), ra_spare_col_make()` specifies the length of the spare being created, in segments. A given spare may be added (`ra_spare_add()`) to more than one segment which may, or may not, create linked segments, depending on the `seg_length` value. The following diagram explains this graphically. Note that it is very unlikely that a real-world DUT would have all of the spare options shown:



In this diagram:

- Spare columns SC1 through SC16 each have a `seg_length` of 1 and will be added to only one segment (`ra_spare_add()`). These spares may only repair the single segment to which they are added and no linked segments are created.

- Spares SC17 through SC20 also have a `seg_length` of 1 but will each be added to 4 segments: SC17 can only repair one of Seg1 through Seg4 and SC18 can only repair one of Seg5 through Seg8, etc. These segments are not linked segments because these spares may only repair errors in one segment.
- Spares SC21 and SC22 have a `seg_length` of 2 and each will be added to 8 segments. SC21 can repair any 2 vertical segments between Seg1 and Seg8. This will link Seg1 with Seg5, or Seg2 with Seg6, or Seg3 with Seg7, etc. Similarly, SC22 can repair any 2 vertical segments between Seg9 and Seg16. This will link Seg9 with Seg13, or Seg10 with Seg14, etc.
- Segment SC23 has a `seg_length` of 4 and will be added to all 16 segments. It can repair any vertical set of 4 segments. This will link Seg1/Seg5/Seg9/Seg13, etc.
- Spare SC24 has a `seg_length` of 1 and will be added to all 16 segments. It can repair any single segment. This does not cause any linked segments.
- Spare SC25 has a `seg_length` of 2 and will be added to all 16 segments. It can repair any vertical pair of segments in the upper half (Seg1/Seg5, Seg3/Seg7, etc.) or any vertical pair of segments in the lower half (Seg9/Seg13, etc.). The RA software does not allow SC25 to replace, for example, SC5/SC9, etc.). This spare does create linked segments.

During RA, when a repair is identified for a segment which is linked to other segment(s) the Spares List, Repair List, and Unusable List for all linked segments are updated. When a repair using a shared spare is identified, the RA software puts the spare into the Unusable List for all other segments.

The following functions directly support segment linkage:

- `ra_segment_linkage_count_get()`

This function is not commonly used in most redundancy applications.

5.3.3.2 ra_segment_make()

See RA Segment, Redundancy Analysis (RA).

Note: this function is commonly used in redundancy applications. See Note:

Description

The `ra_segment_make()` function is used to describe the DUT's segment (block) architecture for the RA software. See RA Segment. Each segment description specifies the row and column ranges in that segment.

Note: the RA software requires that at least one segment be created.

In Multi-DUT Test Programs, user code must create/define segments for a single DUT. The system software then automatically duplicates the segment configuration for each DUT in the Pin Assignment Table. The segment (`RaSegment`) returned by `ra_segment_make()` is from the first DUT in the Active DUTs Set (ADS) but may be used elsewhere to identify the segment for any DUT with the Active DUTs Set (ADS) determining which DUT(s) are affected or accessed. See RA vs. Magnum 1/2 Parallel Test. The Active DUTs Set (ADS) and Ignored DUTs Set (IDS) have no effect on `ra_config_set()`. See RA vs. Magnum 1/2 Parallel Test.

The `ra_segment_config_get()` function may be used to get the values set using `ra_segment_make()` for a specified segment.

Usage

```
RaSegment ra_segment_make(  int rmin, int rmax,
                            int cmin, int cmax,
        RaRowAvailableFunc row_available_func DEFAULT_VALUE( 0 ),
        RaColAvailableFunc col_available_func DEFAULT_VALUE( 0 ),
                            BOOL by_row = -1);
```

where:

`rmin` and **`rmax`** identify the first and last row (inclusive) of the segment being created.

`cmin` and **`cmax`** identify the first and last column (inclusive) of the segment being created.

`row_available_func` is optional, and is used to register a user-written RaRowAvailableFunc call-back function which, if used, executes during RA to obtain an array of spare rows that are currently usable to repair the segment being created. Set this argument = 0 (default) to get the available spares directly from the Spares List.

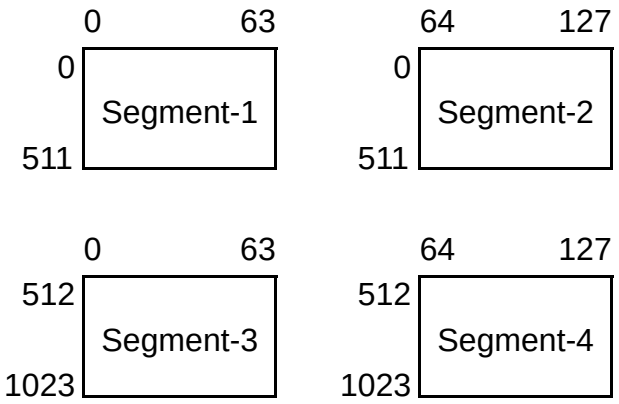
`col_available_func` is optional, and is used to register a user-written RaColAvailableFunc call-back function which, if used, executes during RA to obtain an array of spare columns that are currently usable to repair the segment being created. Set this argument = 0 (default) to get the available spares directly from the Spares List.

by_row is optional and, if used, allows the ECR scan direction to be set on a per-segment basis: TRUE = scan rows fast, FALSE = scan columns fast. The default value (-1) causes the ECR to be scanned in the direction set using `x_fast_axis()`. This parameter has no effect if the ECR Mini-RAM is enabled (see `ra_config_set()`).

`ra_segment_make()` returns the segment created. In Multi-DUT Test Programs, the segment is returned for the first DUT in the Active DUTs Set (ADS) but may be used elsewhere to identify the segment for any DUT, with the Active DUTs Set (ADS) determining which DUT(s) are affected or accessed. See RA vs. Magnum 1/2 Parallel Test.

Example

This example shows one DUT containing 4 segments, each with 64 columns and 512 rows:



The following code example is used to set up this configuration:

```
RaSegment S1 = ra_segment_make( 0, 511, 0, 63);
RaSegment S2 = ra_segment_make( 0, 511, 64, 127 );
RaSegment S3 = ra_segment_make( 512, 1023, 0, 63 );
RaSegment S4 = ra_segment_make( 512, 1023, 64, 127 );
```

5.3.3.3 ra_segment_config_get()

See RA Segment, Redundancy Analysis (RA)

Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_segment_config_get()` function can be used to retrieve several key parameters (previously set using `ra_segment_make()`) for a specified segment.

Usage

```
void ra_segment_config_get(
        RaSegment s,
        int* rmin DEFAULT_VALUE( 0),
        int* rmax DEFAULT_VALUE( 0),
        int* cmin DEFAULT_VALUE( 0),
        int* cmax DEFAULT_VALUE( 0),
        RaRowAvailableFunc *row_available_func DEFAULT_VALUE( 0),
        RaColAvailableFunc *col_available_func DEFAULT_VALUE( 0),
        BOOL * by_row DEFAULT_VALUE( 0));
```

where:

s is the segment of interest.

rmin and **rmax** are optional and, if used, are the addresses of a user-defined `int` variable used to return `rmin` and/or `rmax` value(s) currently set for segment **s**. Specify 0 if these values are not wanted.

cmin and **cmax** are optional and, if used, are the addresses of a user-defined `int` variable used to return `cmin` and/or `cmax` value(s) currently set for segment **s**. Specify 0 if these values are not wanted.

row_available_func is optional and, if used, is the address of an existing `RaRowAvailableFunc` variable used to return the address of the user registered `RaRowAvailableFunc` call-back function. Specify 0 if this value is not wanted. The value returned will be 0 if the user code has not registered a `RaRowAvailableFunc` call-back function.

col_available_func is optional and, if used, is the address of an existing `RaColAvailableFunc` variable used to return the address of the user registered `RaColAvailableFunc` call-back function. Specify 0 if this value is not wanted. The value returned will be 0 if the user code has not registered a `RaColAvailableFunc` call-back function.

by_row is optional and, if used, is the address of an existing BOOL variable used to return the ECR scan direction: TRUE if scan-by-row, FALSE if scan-by-column. Specify 0 if this value is not wanted. The value returned has no effect if the ECR Mini-RAM is enabled (see `ra_config_set()`).

In Multi-DUT Test Programs the values are retrieved from the first DUT in the Active DUTs Set (ADS).

Example

The following example uses `ra_segment_make()` to define one segment, S1, and then uses `ra_segment_config_get()` to get only the row and column min/max values:

```
RaSegment S1 = ra_segment_make( 0, 128, 0, 256 );
int rmin, rmax, cmin, cmax;
ra_segment_config_get( S1, &rmin, &rmax, &cmin, &cmax );
```

5.3.3.4 ra_segment_count_get()

See RA Segment, Redundancy Analysis (RA)

Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_segment_count_get()` function is used to return either:

- The number of segments created.
- The number of segments associated with a spare row or spare column (see `ra_spare_add()`).

RA segments are created using `ra_segment_make()` and are stored in a list. `ra_segment_count_get()` gets the number of segments in the list. `ra_segment_count_get()` is targeted for use with `ra_segment_get()` to sequentially get each segment in the list or each segment associated with a specified spare.

In Multi-DUT Test Programs, the value is returned for the first DUT in the Active DUTs Set (ADS), however, this should be the same for all DUTs.

Usage

The following function returns the number of segments currently defined for one DUT:

```
int ra_segment_count_get();
```

The following functions return the number of segments associated with the specified spare row or column:

```
int ra_segment_count_get( RaSpareRow r );
int ra_segment_count_get( RaSpareCol c );
```

where:

r and **c** identify the spare row or spare column of interest.

`ra_segment_count_get()` returns either the number of segments created or the number of segments associated with the specified spare.

Example

```
int num_segs = ra_segment_count_get();
for( int i = 0; i < num_segs; ++i ){
   RaSegment s = ra_segment_get( i );
   // Do something with the segment returned
}
```

5.3.3.5 ra_segment_get()

See RA Segment, Redundancy Analysis (RA)

Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_segment_get()` function is used with `ra_segment_count_get()` to sequentially get each segment created or each segment associated with a specified spare row or spare column (see `ra_spare_add()`).

RA segments are created using `ra_segment_make()` and are stored in a list. `ra_segment_get()` is targeted for use with `ra_segment_count_get()` to sequentially get each segment in the list or each segment associated with a specified spare.

In Multi-DUT Test Programs, the segment is returned for the first DUT in the Active DUTs Set (ADS).

Usage

```
RaSegment ra_segment_get( int index );
RaSegment ra_segment_get( RaSpareRow r, int index );
RaSegment ra_segment_get( RaSpareCol c, int index );
```

where:

index is the zero-based value which specifies which segment is returned.

r and **c** identify the spare row or spare column of interest.

`ra_segment_get()` returns the *index*'th segment created or the *index*'th segment associated with the specified spare. NULL is returned if index is out of range.

Example

See Example.

5.3.3.6 ra_segment_id_get()

See RA Segment, Redundancy Analysis (RA).

Note: this function is not used in most redundancy applications. See Note:. It is included for backwards compatibility with Maverick-I/-II RA. New programs should use `ra_segment_count_get()` and `ra_segment_get()`.

Description

The `ra_segment_id_get()` function returns a segment's ID. Each RA segment has an integer ID (see `ra_segment_lookup()`) which is typically only useful to sequentially process every segment of a DUT.

Usage

```
int ra_segment_id_get( RaSegment s );
```

where:

s identifies the segment of interest.

In Multi-DUT Test Programs, the segment is returned for the first DUT in the Active DUTs Set (ADS).

Example

```
int id = ra_segment_id_get( S1 );
```

5.3.3.7 ra_segment_lookup()

See RA Segment, Redundancy Analysis (RA)

Note: this function is not used in most redundancy applications. See Note:. It is included for backwards compatibility with Maverick-I/-II RA. New programs should use `ra_segment_count_get()` and `ra_segment_get()`.

Description

The `ra_segment_lookup()` function returns a pointer to a segment given its ID. Each RA segment has an integer ID (see `ra_segment_id_get()`) which is typically only useful to sequentially process every segment of a DUT.

Usage

```
RaSegment ra_segment_lookup( int id );
```

where:

`id` identifies the segment of interest. See Description.

`ra_segment_lookup()` returns a pointer to the specified segment. NULL is returned if **`id`** is not a valid segment ID. In Multi-DUT Test Programs, the segment is returned for the first DUT in the Active DUTs Set (ADS).

Example

```
RaSegment s = ra_segment_lookup( ra_segment_id_get( S1 ) );
```

5.3.3.8 ra_max_bad_segments_set(), ra_max_bad_segments_get()

See Redundancy Analysis (RA), Spare Segments.

Note: these functions are not used in most redundancy applications. See Note:

Description

The `ra_max_bad_segments_set()` function sets the number of unrepairable segments allowed before a DUT is considered unrepairable during RA. `ra_config_set()` also sets this value (more below). In Multi-DUT Test Programs the value is set for all DUTs the Active DUTs Set (ADS).

`ra_max_bad_segments_set()` is used primarily in re-test/re-repair situations, to temporarily reduce the `max_bad_segs` allowed when some spare segments have been consumed in previous repairs.

Note: the change made using `ra_max_bad_segments_set()` is temporary; executing `ra_reset()` **always** restores the `max_bad_segs` value to the value originally set using `ra_config_set()`.

The `ra_max_bad_segments_get()` function is used to get the current `max_bad_segs` value. This is the value initially set using `ra_config_set()` or the temporary value set using `ra_max_bad_segments_set()`.

Usage

```
void ra_max_bad_segments_set( int max_bad_segs );
int ra_max_bad_segments_get();
```

where:

`max_bad_segs` specifies the number of unrepairable segments allowed before a DUT is considered unrepairable. A value of 0 means no unrepairable segments are allowed before declaring the DUT unrepairable. A warning is issued if **`max_bad_segs`** is negative, or if the value exceeds the number segments defined for the DUT. In Multi-DUT Test Programs the value is set for all DUT(s) in the Active DUTs Set (ADS). See Note: above.

`ra_max_bad_segments_get()` returns the current `max_bad_segs` value. In Multi-DUT Test Programs the value is returned for the first DUT in the Active DUTs Set (ADS).

Example

The following code sets the max_bad_segs value to 0, then gets that value and prints it:

```
ra_config_set( TRUE, TRUE, 0 ); // ra_config_set()
output(" Max Bad Segs => %d", ra_max_bad_segments_get() );
```

The following code sets the max_bad_segs value to 2, then gets that value and prints it:

```
ra_max_bad_segments_set( 2 );
output(" Max Bad Segs => %d", ra_max_bad_segments_get() );
```

The following code resets the max_bad_segs value to that last set using ra_config_set():

```
ra_reset();
output(" Max Bad Segs => %d", ra_max_bad_segments_get() );
```

5.3.3.9 ra_segment_linkage_count_get()

See RA Segment, Redundancy Analysis (RA)

Note: this function is not used in most redundancy applications. See Note:

Description

The ra_segment_linkage_count_get() function is used to determine the number of segments linked to a specified segment. Segments are linked when a spare with a length greater than 1 is associated with more than one segment. See Linked Segments and ra_spare_add().

Usage

```
int ra_segment_linkage_count_get( RaSegment s );
```

where:

s identifies the segment of interest.

ra_segment_linkage_count_get() returns the number of segments linked to segment **s**. In Multi-DUT Test Programs, the value is returned for the first DUT in the Active DUTs Set (ADS).

Example

In the following example `ra_segment_linkage_count_get()` will return 2; i.e. two segments are linked to segment S2:

```
RaSegment S1 = ra_segment_make(); // Parameters not shown
RaSegment S2 = ra_segment_make(); // Parameters not shown
RaSegment S3 = ra_segment_make(); // Parameters not shown
RaSpareCol C1 = ra_spare_col_make();
ra_spare_add( S1, C1 );
ra_spare_add( S2, C1 );
ra_spare_add( S3, C1 );
// Other code to do other things...
int count = ra_segment_linkage_count_get( S2 );
```

5.3.4 RA Spares

See Redundancy Analysis (RA), RA Software.

The functions related to RA spares are:

Commonly Used	Rarely Used (see Note:)
ra_spare_row_make()	ra_spare_config_get()
ra_spare_col_make()	ra_usable_set()
ra_spare_add()	ra_unusable_set()
	ra_spare_row_count_get(), ra_spare_col_count_get()
	ra_spare_id_get()
	ra_spare_row_lookup(), ra_spare_col_lookup()
	ra_spare_row_get(), ra_spare_col_get()
	ra_spare_rows_get(), ra_spare_cols_get()
	ra_spare_rownum_set(), ra_spare_rownum_get()
	ra_spare_colnum_set(), ra_spare_colnum_get()
	ra_spare_position_set(), ra_spare_position_get()
	ra_spare_mask_count_get(), ra_spare_mask_get()
	ra_spare_current_mask_set(), ra_spare_current_mask_get()
	ra_shortest_spare_row_get(), ra_shortest_spare_col_get()

5.3.4.1 ra_spare_row_make(), ra_spare_col_make()

See RA Spares, RA Software.

Note: these functions are used in most redundancy applications. See Note:

Description

The `ra_spare_row_make()` function is used to create a spare row.

The `ra_spare_col_make()` function is used to create a spare column.

Spare rows and columns cannot be used for repair until associated with one or more segment(s), using `ra_spare_add()`.

The `seg_length` argument to these functions is used to specify the length, in segments, of the spare being created. This is required when a given spare can be used to repair more than one segment and/or the spare will create Linked Segments.

In Multi-DUT Test Programs, user code must create/define spare row(s) and/or spare column(s) for a single DUT. The system software then automatically duplicates these spare(s) for each DUT in the test program. The spare row (`RaSpareRow`) returned by `ra_spare_row_make()` and the spare column (`RaSpareCol`) returned by `ra_spare_col_make()` is from the first DUT in the Active DUTs Set (ADS), but may be used elsewhere to identify a spare for any DUT with the Active DUTs Set (ADS) determining which DUT(s) are affected or accessed. See RA vs. Magnum 1/2 Parallel Test. The Active DUTs Set (ADS) and Ignored DUTs Set (IDS) have no effect on `ra_spare_row_make()` and `ra_spare_col_make()`.

The configuration details of a specified spare can be retrieved using `ra_spare_config_get()`.

Usage

```
RaSpareRow ra_spare_row_make(
                        __int64 mask DEFAULT_VALUE( 0 ),
                        int seg_length DEFAULT_VALUE( -1 ),
                        int numRUT DEFAULT_VALUE( 1 ),
                        int blanksBetweenRUT DEFAULT_VALUE( 0 ),
                        RaRowUseOK okFunc DEFAULT_VALUE( 0 ) );
```

```
RaSpareCol ra_spare_col_make(
                        __int64 mask DEFAULT_VALUE( 0 ),
                        int seg_length DEFAULT_VALUE( -1 ),
                        int numCUT DEFAULT_VALUE( 1 ),
                        int blanksBetweenCUT DEFAULT_VALUE( 0 ),
                        RaColUseOK okFunc DEFAULT_VALUE( 0 ) );
```

where:

mask is optional and, if specified, identifies which data bit(s) the spare can replace. The value 0 means the spare replaces all data bits in the associated segment(s). A value other than 0 signifies Per I/O Spares and the RA software generates the set of Per-I/O Spare Mask values and sets the spare's mask to the initial value. The low order mask bit represents D0, etc. Only the low 36 bits are used.

seg_length is optional and, if used, specifies the length of the spare, in segments. See Description and Linked Segments. The default **seg_length** value (-1) sets the spare's length to match the number of contiguous segments to which it is later added using `ra_spare_add()`; i.e. Linked Segments are created if the spare is added to more than one segment.

numRUT and **numCUT** are optional and, if used, specifies the number of rows or columns that the spare will replace (see Rows-Used-Together(RUT), Columns-Used-Together(CUT)). Default = 1; i.e. the spare can replace one row or one column.

blanksBetweenRUT and **blanksBetweenRUT** is optional and, if used, specifies the number of non-replaced rows or columns between the rows or columns replaced by this spare; i.e. the gap between rows/columns replaced by the spare. Default = 0. Legal values are 0, 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, etc.

okFunc is optional and is used to register a user-defined `RaRowUseOK` or `RaColUseOK` call-back function (see Redundancy Call-back Functions). As RA occurs (`ra_execute()`), and the need for a repair is identified, the process of choosing which spare element will be used to make the repair will execute each spare element's call-back function as part of the decision process. If the call-back function returns FALSE, that spare will not be used to make that repair. There are no default `RaRowUseOK` or `RaColUseOK` call-back functions; i.e. any spare in the Spares List will be considered usable if otherwise valid for a given repair. The built-in `ra_exclusive()` function is available as an example call-back.

`ra_spare_row_make()` and `ra_spare_col_make()` return the spare created. In Multi-DUT Test Programs, the spare is returned for the first DUT in the Active DUTs Set (ADS).

The following errors are checked:

- The number of mask bits exceeds the segment's data width.

- **numRUT/numCUT** is less than 1.
- **numRUT/numCUT** is 1, but blanksBetweenRUT/CUT is not 0.
- **blanksBetweenRUT/blanksBetweenCUT** is not a legal value.

Example

Example 1:

The following example creates one spare row which can replace the full data width of any row of the segment(s) with which is associated:

```
RaSpareRow r = ra_spare_row_make();
```

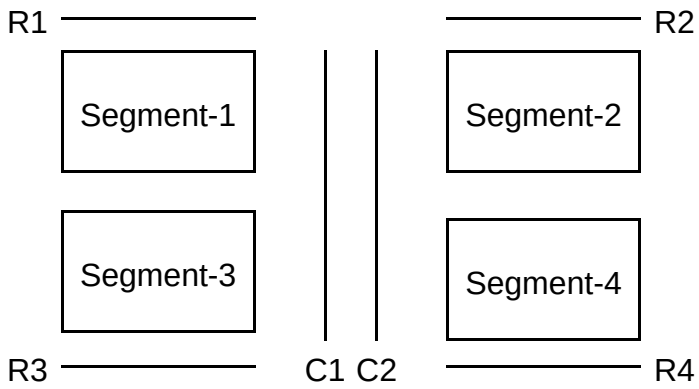
Example 2:

The following example creates a 2-CUT per-I/O spare column (see Per I/O Spares). The mask indicates it can replace D0+D2 or D1+D3, etc. There is a 1-bit gap between the two columns this spare can replace and the user's `myColRules()` function is registered as a `RaColUseOK` call-back.

```
RaSpareCol c = ra_spare_col_make( 0x5, -1, 2, 1, myColRules );
```

Example 3:

In the following example, the DUT has 4 segments. Each segment has one spare row which can only be used to repair one segment. The DUT also has two spare columns which are shared between all four segments and which link segments 1/3 and segments 2/4 (see Linked Segments):

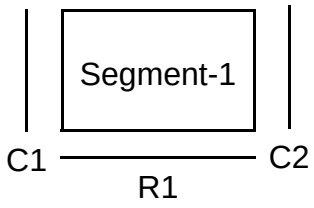



```
RaSegment S1 = ra_segment_make(...); // Details not included
RaSegment S2 = ra_segment_make(...);
RaSegment S3 = ra_segment_make(...);
RaSegment S4 = ra_segment_make(...);
```

```
ra_spare_add( S1, ra_spare_row_make() ); // ra_spare_add()
ra_spare_add( S2, ra_spare_row_make() );
ra_spare_add( S3, ra_spare_row_make() );
ra_spare_add( S4, ra_spare_row_make() );

// Spare columns are shared among all segments and link 2 segments
RaSpareCol C1 = ra_spare_col_make( 0, 2 );
RaSpareCol C2 = ra_spare_col_make( 0, 2 );

ra_spare_add( S1, C1 ); // ra_spare_add()
ra_spare_add( S2, C1 );
ra_spare_add( S3, C1 );
ra_spare_add( S4, C1 );
ra_spare_add( S1, C2 );
ra_spare_add( S2, C2 );
ra_spare_add( S3, C2 );
ra_spare_add( S4, C2 );
```

Example 4:

This example shows two user-created `RaColUseOK` call-back functions, named `odd()` and `even()`, and how they are used during spare creation. This device has one segment, with one spare row (R1) and two spare columns (C1, C2). Spare column C1 can only replace odd columns (1,3, 5, etc.) and spare column C2 can only replace even columns (0, 2, 4, etc.).

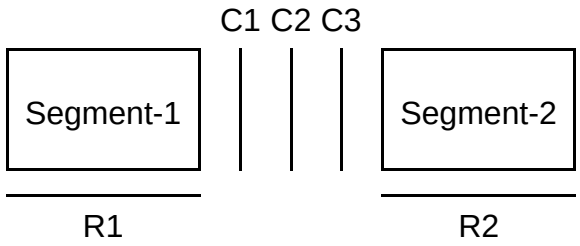



```
BOOL odd( RaSegment s, RaSpareCol spare_col ){ //For RaColUseOK
   // c is UNUSED in this example
   // s is UNUSED in this example
   return ( spare_col % 2 ); // TRUE for odd columns
}

BOOL even( RaSegment s, RaSpareCol spare_col ){//For RaColUseOK
   // c is UNUSED in this example
   // s is UNUSED in this example
   return !( spare_col % 2 ); // FALSE for odd columns
}

RaSegment S = ra_segment_make(...); // Details not included
```

```
RaSpareCol C1 = ra_spare_col_make( 0, 1, 1, 0, odd );
RaSpareCol C2 = ra_spare_col_make( 0, 1, 1, 0, even );
RaSpareRow R1 = ra_spare_row_make();

ra_spare_add( S, C1 ); // ra_spare_add()
ra_spare_add( S, C2 );
ra_spare_add( S, R1 );
```

Example 5:

In the following example, the DUT has two segments, each with one spare row (R1). Three spare columns (C1, C2, C3) are shared between the two segments, and can be used to repair either segment, with the restriction that no more than two spare columns can be used to repair any one segment. The user-written call-back function named `max_two()` shows how to enforce this restriction.




```
// RaColUseOK call-back function
BOOL max_two( RaSegment s, RaSpareCol spare_col ){
   if( ra_repaired_col_count_get( s ) < 2 ) return TRUE;
   else {
      ra_unusable_set( s, spare_col ); // ra_unusable_set()
      return FALSE;
   }
}

RaSegment S1 = ra_segment_make(...); // Details not included
RaSegment S2 = ra_segment_make(...); // Details not included

RaSpareCol C1 = ra_spare_col_make( 0, 1, 1, 0, max_two );
RaSpareCol C2 = ra_spare_col_make( 0, 1, 1, 0, max_two );
RaSpareCol C3 = ra_spare_col_make( 0, 1, 1, 0, max_two );
RaSpareRow R1 = ra_spare_row_make();
RaSpareRow R2 = ra_spare_row_make();

ra_spare_add( S1, C1 ); // ra_spare_add()
ra_spare_add( S1, C2 );
ra_spare_add( S1, C3 );
```

```
ra_spare_add( S2, C1 );
ra_spare_add( S2, C2 );
ra_spare_add( S2, C3 );
ra_spare_add( S1, R1);
ra_spare_add( S2, R2);
```

5.3.4.2 ra_spare_add()

See RA Spares, RA Software.

Note: this function is commonly used in redundancy applications. See Note:

Description

The `ra_spare_add()` function is used to associate a spare row or column with a segment and causes the spare to be added to the Spares List. Spare elements which are not in the Spares List are not usable during RA; i.e. they can't/won't be allocated to make a repair.

Note the following:

- All spares of the same type (i.e., spare rows or spare columns) associated with a given segment must be the same data width. It is not legal, for example, to add a 1-bit wide spare row to a segment that already has a 16-bit wide spare row.
- Adding a given spare row or spare column to more than one segment may create Linked Segments, depending on the *length* of the spare. See Linked Segments.
- The cardinality of all spare rows associated with a segment must be identical. Likewise for spare columns. In this context, *cardinality* refers to the number of rows or columns replaced by a RUT or CUT. See Rows-Used-Together(RUT), Columns-Used-Together(CUT). Within a given segment, RUTs and CUTs may not be intermixed with individual spare rows and columns.
- In Multi-DUT Test Programs, user code only needs to add spare row(s) and/or spare column(s) to a segment for a single DUT. The system software automatically duplicates these spare-vs.-segment associations for each DUT in the Pin Assignment Table. See RA vs. Magnum 1/2 Parallel Test. The Active DUTs Set (ADS) and Ignored DUTs Set (IDS) have no effect on `ra_spare_add()`.

Usage

```
BOOL ra_spare_add( RaSegment s, RaSpareRow r );
```

```
BOOL ra_spare_add( RaSegment s, RaSpareCol c );
```

where:

s identifies the target segment.

r and **c** identify the spare to be added to segment **s**. This spare must have been previously created using `ra_spare_row_make(), ra_spare_col_make()`.

`ra_spare_add()` returns TRUE if no errors occur. FALSE is returned if **s**, **r**, or **c** are NULL or if the data width of the spare being added is different than any previous spares (of the same type) added to the segment. FALSE is also returned if the cardinality rule noted in Description is violated.

Example

```
RaSegment S1 = ra_segment_make(...); // Details not included
RaSpareCol C1 = ra_spare_col_make(...); // Details not included
RaSpareCol C2 = ra_spare_col_make(...); // Details not included
RaSpareCol C3 = ra_spare_col_make(...); // Details not included
RaSpareCol C4 = ra_spare_col_make(...); // Details not included
ra_spare_add( S1, C1 );
ra_spare_add( S1, C2 );
ra_spare_add( S1, C3 );
ra_spare_add( S1, C4 );
```

5.3.4.3 ra_spare_config_get()

See RA Spares, RA Software.

Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_spare_config_get()` function can be used to retrieve the configuration attributes, set using `ra_spare_row_make(), ra_spare_col_make()`, of a specified spare row or spare column.

In Multi-DUT Test Programs, the information is returned for the first DUT in the Active DUTs Set (ADS).

Usage

```
void ra_spare_config_get(RaSpareRow r,
                         __int64* mask DEFAULT_VALUE( 0 ),
                         int* seg_length DEFAULT_VALUE( 0 ),
                         int* numRUT DEFAULT_VALUE( 0 ),
                         int* blanksBetweenRUT DEFAULT_VALUE( 0 ));
void ra_spare_config_get(RaSpareCol c,
                         __int64* mask DEFAULT_VALUE( 0 ),
                         int* seg_length DEFAULT_VALUE( 0 ),
                         int* numCUT DEFAULT_VALUE( 0 ),
                         int* blanksBetweenCUT DEFAULT_VALUE( 0 ));
```

where:

r and **c** identify the spare row or column of interest.

mask is optional and, if used, is the address of an existing `__int64` variable used to return the Per-I/O Spare Mask attribute. Specify 0 if this parameter is not desired.

seg_length is optional and, if used, is the address of an existing `int` variable used to return the length of the spare, in segments. See Description and Linked Segments. Specify 0 if this parameter is not desired.

numRUT and **numCUT** are optional and, if used, are the address of an existing `int` variable used to return the number of Rows-Used-Together(RUT), Columns-Used-Together(CUT). Specify 0 if this parameter is not desired.

blanksBetweenRUT and **blanksBetweenCUT** are optional and, if used, are the address of an existing `int` variable used to return the number of rows or columns between each RUT or CUT; i.e. the gap between RUTs/CUTs.

Example

The following example returns a value for each attribute of the specified row:

```
__int64 mask;
int numruts, rutgap;
RaSpareRow r = ra_spare_row_make( ... ); // Details not included
ra_spare_config_get( r, &mask, &numruts, &rutgap);
```

The following example returns only the number of RUTs value for the specified row:

```
ra_spare_config_get( r, 0, &numruts );
```

5.3.4.4 ra_usable_set()

See RA Execution And Results, Repair List Functions, RA Software.

Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_usable_set()` function is used to move a specified spare row or column, associated with a specified segment, from the Unusable List back to the Spares List. This allows any subsequent RA processing to allocate that spare for a repair.

By default, all spare elements defined are initially usable. Spare elements become unusable by calling the `ra_unusable_set()` function, typically during an RaRowUseOK or RaRowUseOK call-back function (see Redundancy Call-back Functions).

In Multi-DUT Test Programs, the spare is moved for the first DUT in the Active DUTs Set (ADS). This is unconventional as compared to most other setter functions.

`ra_usable_set()` has no effect on spares already moved to the Repair List.

Executing `ra_reset()` and `ra_segment_reset()` returns spares in the Unusable List to the Spares List, typically in preparation for testing new DUT(s).

Usage

```
BOOL ra_usable_set( RaSegment s, RaSpareRow r );
BOOL ra_usable_set( RaSegment s, RaSpareCol c );
```

where:

s identifies the segment of interest.

r and **c** identify the spare row or column to be moved from the Unusable List back to the Spares List. A warning is issued if **r** or **c** is not a valid spare for **s** or if **r** or **c** is not currently in the Unusable List.

These functions return TRUE if the designated spare element is successfully moved from the Unusable List list to the Spares List list. FALSE is returned if the specified spare is not currently in the Unusable List.

Example

```
RaSegment S1 = ra_segment_make(); // Parameters not shown
RaSpareRow R1 = ra_spare_row_make();
ra_spare_add( S1, R1 ); // ra_spare_add()
// Other code here which must have moved R1 to the Unusable List
BOOL ok = ra_usable_set( S1, R1 );
if( !ok ) output(" WARNING: ra_usable_set() returned FALSE");
```

5.3.4.5 ra_unusable_set()

See RA Execution And Results, Repair List Functions, RA Software.

Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_unusable_set()` function is used to move a specified spare row or column from the Spares List to the Unusable List. Any subsequent RA processing cannot allocate that spare for a repair.

The `ra_unusable_set()` function has two general applications:

- By default, all spare elements defined are considered usable during the RA. If the device architecture limits the universal application of a spare element, a user-defined `RaRowUseOK` and/or `RaRowUseOK` call-back function may be used to enforce the application rules (see RaRowUseOK & RaColUseOK Call-back Functions).
- When retesting a previously repaired device sometimes the original Spares List, Repair List and Unusable List are lost, as occurs when off-line repair is performed. Some device designs allow the test program to read back which spare elements were consumed in previous repairs. When it is possible, this information can be used to move those spares from the Spares List to the Unusable List.

The version of `ra_unusable_set()` with the `RaSegment` argument makes the specified spare unusable for only the specified segment. The version without the `RaSegment` argument makes the spare unusable for all segment(s) which are linked by the spare or which share the spare (see Linked Segments).

In Multi-DUT Test Programs, the spare is moved for the first DUT in the Active DUTs Set (ADS). This is unconventional as compared to most other setter functions.

A spare can be made usable again using the `ra_usable_set()` function.

Executing `ra_reset()` and `ra_segment_reset()` returns spares to the Spares List, typically in preparation for testing new DUT(s).

`ra_unusable_set()` has no effect on spares in the Repair List.

Usage

```
BOOL ra_unusable_set( RaSpareRow r );
BOOL ra_unusable_set( RaSpareCol c );
BOOL ra_unusable_set( RaSegment s, RaSpareRow r );
BOOL ra_unusable_set( RaSegment s, RaSpareCol c );
```

where:

r and **c** identify the spare row or column to be moved from the Spares List to the Unusable List.

s identifies the segment of interest. A warning is issued if **r** or **c** is not a valid spare for **s**.

These functions return TRUE if the designated spare element is successfully moved from the Spares List to the Unusable List. FALSE is returned if the specified spare is not currently in the Spares List.

Example

```
RaSegment S1 = ra_segment_make(); // Parameters not shown
RaSpareCol C1 = ra_spare_col_make();
ra_spare_add( S1, C1 ); // ra_spare_add()
// Other code here which must not have moved C1 from the Spares List
BOOL ok = ra_unusable_set( S1, C1 );
if( !ok ) output(" WARNING: ra_unusable_set() returned FALSE");
```

5.3.4.6 ra_spare_row_count_get(), ra_spare_col_count_get()

See RA Spares, RA Software.

Note: these functions are not used in most redundancy applications. See Note:

Description

The `ra_spare_row_count_get()` function is used to get a count of the number of spare row(s) associated with a specified segment (see `ra_spare_add()`) from the Spares List or Unusable List. Similarly, the `ra_spare_col_count_get()` function is used to get a count of the number of spare column(s) associated with a specified segment. from the Spares List or Unusable List.

These functions are used in conjunction with `ra_spare_row_get()` and `ra_spare_col_get()` to sequentially process each spare row or column associated with a specified segment.

Usage

```
int ra_spare_row_count_get( RaSegment s,
                            BOOL usable DEFAULT_VALUE( TRUE ) );
int ra_spare_col_count_get( RaSegment s,
                            BOOL usable DEFAULT_VALUE( TRUE ) );
```

where:

s identifies the segment of interest.

usable is optional and, if used, specifies whether the spares counted are in the Spares List (TRUE, default) or Unusable List (FALSE).

`ra_spare_row_count_get()` returns the number of spare rows associated with segment **s**. In Multi-DUT Test Programs, the value is returned for the first DUT in the Active DUTs Set (ADS).

`ra_spare_col_count_get()` returns the number of spare columns associated with segment **s**. In Multi-DUT Test Programs, the value is returned for the first DUT in the Active DUTs Set (ADS).

Example

In the following example, the `for()` loop will sequentially return the 3 spare columns associated with segment S1:

```
RaSegment S1 = ra_segment_make(); // Parameters not shown
RaSpareCol C1 = ra_spare_col_make();
RaSpareCol C2 = ra_spare_col_make();
RaSpareCol C3 = ra_spare_col_make();
ra_spare_add( S1, C1 ); // ra_spare_add()
ra_spare_add( S1, C2 );
ra_spare_add( S1, C3 );
// Other code to do other things...
for (int i = 0; i < ra_spare_col_count_get( S1 ); ++i ){
   RaSpareCol sparecol = ra_spare_col_get( S1, i );
   // Do something with sparecol
}
```

5.3.4.7 ra_spare_row_get(), ra_spare_col_get()

See RA Spares, RA Software.

Note: these functions are not used in most redundancy applications. See Note:

Description

The `ra_spare_row_get()` function is used in conjunction with `ra_spare_row_count_get()` to sequentially get the spare row(s), associated with a specified segment, from the Spares List or Unusable List.

Similarly, the `ra_spare_col_get()` function is used in conjunction with `ra_spare_col_count_get()` to sequentially get the spare column(s), associated with a specified segment, from the Spares List or Unusable List.

The `ra_repaired_row_get(), ra_repaired_col_get()` functions may be used to get spares from the pending Repair List or done Repair List.

Usage

```
RaSpareRow ra_spare_row_get(RaSegment s,
                            int index,
                            BOOL usable DEFAULT_VALUE( TRUE ) );
```

```
RaSpareCol ra_spare_col_get(RaSegment s,
                            int index,
                            BOOL usable DEFAULT_VALUE( TRUE ) );
```

where:

s identifies the segment of interest.

index is the zero-based index specifying which spare to return.

usable is optional and, if used, specifies whether the spares are retrieved from the Spares List (TRUE, default) or Unusable List (FALSE).

`ra_spare_row_count_get()` returns the **index**'th spare row associated with segment **s**. NULL is returned if the **index** is out of range. In Multi-DUT Test Programs, the spare row is returned for the first DUT in the Active DUTs Set (ADS).

`ra_spare_col_get()` returns the **index**'th spare column associated with segment **s**. NULL is returned if the **index** is out of range. In Multi-DUT Test Programs, the spare column is returned for the first DUT in the Active DUTs Set (ADS).

Example

See Example.

5.3.4.8 ra_spare_id_get()

See RA Spares, RA Software.

Note: this function is not used in most redundancy applications. See Note:. It is included for backwards compatibility with Maverick-I/-II RA. New programs should use `ra_spare_row_count_get()` with `ra_spare_row_get()` or `ra_spare_col_count_get()` with `ra_spare_col_get()`.

Description

The `ra_spare_id_get()` function returns a spare row's or spare column's ID. Each RA spare has an integer ID which is typically only useful to sequentially process every spare row or spare column of a DUT. See `ra_spare_row_lookup(), ra_spare_col_lookup()`.

Usage

```
int ra_spare_id_get( RaSpareRow r );
int ra_spare_id_get( RaSpareCol c );
```

where:

r and **c** identify the spare row or column of interest.

`ra_spare_id_get()` returns the integer ID of the specified row or column. In Multi-DUT Test Programs, the information is returned for the first DUT in the Active DUTs Set (ADS).

Example

```
int id = ra_spare_id_get( R1 );
```

5.3.4.9 ra_spare_row_lookup(), ra_spare_col_lookup()

See RA Spares, RA Software.

Note: these functions are not used in most redundancy applications. See Note:

Description

The `ra_spare_row_lookup()` function returns a pointer to a spare row given its ID.

The `ra_spare_col_lookup()` function returns a pointer to a spare column given its ID.

Each RA spare has an integer ID (see `ra_spare_id_get()`) which is typically only useful to sequentially process every spare row or spare column of a DUT.

Usage

```
RaSpareRow ra_spare_row_lookup( int id );
RaSpareCol ra_spare_col_lookup( int id );
```

where:

id is the ID number of the spare row or column of interest.

`ra_spare_row_lookup()` returns a pointer to the spare row identified by ID. NULL is returned if the **`id`** is invalid. In Multi-DUT Test Programs, the information is returned for the first DUT in the Active DUTs Set (ADS).

Example

See Example

5.3.4.10 ra_spare_rows_get(), ra_spare_cols_get()

See RA Spares, RA Software.

> Note: these functions are not used in most redundancy applications. See Note:

Description

The `ra_spare_rows_get()` function is used to get a list of the available spare row(s) associated with a specified segment from the Spares List or the Unusable List.

The `ra_spare_col_get()` function is used to get a list of the available (unused) spare column(s) associated with a specified segment from the Spares List or the Unusable List.

Usage

```
RaSpareRowArray& ra_spare_rows_get(
                    RaSegment s,
                    RaSpareRowArray& rows,
                    BOOL usable DEFAULT_VALUE( TRUE ) );
RaSpareColArray& ra_spare_cols_get(
                    RaSegment s,
                    RaSpareColArray& cols,
                    BOOL usable DEFAULT_VALUE( TRUE ) );
```

where:

`s` identifies the segment of interest.

`rows` and **`cols`** are a user-defined `RaSpareRowArray` or `RaSpareColArray`, used to return the list of spare rows or columns associated with segment **`s`**. These arrays are automatically resized by the system software as needed. Any prior contents are lost. The

size and individual elements in the `RaSpareRowArray` or `RaSpareColArray` can be obtained using standard `CArray` member functions.

`usable` is optional and, if used, specifies whether the spare should be retrieved from the Spares List (TRUE, default) or the Unusable List (FALSE).

Both functions return the array passed in. In Multi-DUT Test Programs, the list of spares is returned for the first DUT in the Active DUTs Set (ADS).

Example

The following example uses `ra_spare_rows_get()` to get a list of the 3 spare rows associated with segment S1:

```
RaSegment S1 = ra_segment_make(); // Parameters not shown
RaSpareRow R1 = ra_spare_row_make();
RaSpareRow R2 = ra_spare_row_make();
RaSpareRow R3 = ra_spare_row_make();
ra_spare_add( S1, R1 );
ra_spare_add( S1, R2 );
ra_spare_add( S1, R3 );
// Other code to do other things...
RaSpareRowArray rows = ra_spare_rows_get( S1 , rows );
for( int r = 0; r< rows.GetSize(); ++r ){
   RaSpareRow my_row = rows[r]; // or = rows.GetAt( r );
   // Do something with the my_row
}
```

5.3.4.11 ra_spare_colnum_set(), ra_spare_colnum_get()

See 5.3.4.12 (next).

5.3.4.12 ra_spare_rownum_set(), ra_spare_rownum_get()

See RA Spares, RA Software.

Note: these functions are not used in most redundancy applications. See Note:

Description

Each spare row and spare column stores the following information about what they repair:

- The row or column being replaced
- A Per-I/O Spare Mask indicating which data bits are being replaced

Note that these values are not normally useful until a given spare has been allocated by the RA (`ra_spare_use()` via `ra_execute()`) to make a repair.

The `ra_spare_rownum_set()` function can be used to set the row being replaced by a specified spare row. Similarly, `ra_spare_colnum_set()` can be used to set the column being replaced by the specified spare column. In Multi-DUT Test Programs, the value is set for all DUT(s) in the Active DUTs Set (ADS).

Note that `ra_spare_position_set()` and `ra_spare_position_set()` can be used to set both the replaced row or column and the Per-I/O Spare Mask by a specified spare.

The `ra_spare_rownum_get()` and `ra_spare_colnum_get()` functions are used to retrieve the replaced row or column for a specified spare. In Multi-DUT Test Programs, the value is returned for the first DUT in the Active DUTs Set (ADS).

Usage

```
void ra_spare_rownum_set( RaSpareRow r, int rownum );
void ra_spare_colnum_set( RaSpareCol c, int colnum );
int ra_spare_rownum_get( RaSpareRow r );
int ra_spare_colnum_get( RaSpareCol c );
```

where:

`r` and **`c`** identify the target spare row or column.

`rownum` and **`colnum`** specify the row or column replaced by spare row **`r`** or spare column **`c`**.

`ra_spare_rownum_get()` returns the row replaced by spare row **`r`**.

`ra_spare_colnum_get()` returns the column replaced by spare row **`c`**.

Example

```
RaSpareRow r = ra_spare_row_make(...); // Details not included
ra_spare_rownum_set( r, 4 );
int replaced_row = ra_spare_rownum_get( r );
```

5.3.4.13 ra_spare_position_set(), ra_spare_position_get()

See RA Spares, RA Software.

Note: these functions are not used in most redundancy applications. See Note:

Description

Each spare row and spare column stores the following information about what they repair:

- The row or column being replaced
- A Per-I/O Spare Mask indicating which data bits are being replaced

The `ra_spare_position_set()` function can be used to set both the row or column being replaced and the data bits being replaced by a specified spare row or column. The two values are passed using an `RaErrorPosition` variable: see Example. In Multi-DUT Test Programs, the value is set for all DUT(S) in the Active DUTs Set (ADS).

`ra_spare_position_get()` can be used to get both the row or column being replaced and the data bits being replaced by a specified spare row or column. Note that these values are not normally useful until a given spare has been allocated by the RA (`ra_spare_use()` via `ra_execute()`) to make a repair.

Usage

```
RaSpareRow ra_spare_position_set(RaSpareRow r,
                                 RaErrorPosition& pos);
RaSpareCol ra_spare_position_set(RaSpareCol c,
                                 RaErrorPosition& pos);
RaErrorPosition ra_spare_position_get( RaSpareRow r );
RaErrorPosition ra_spare_position_get( RaSpareCol c );
```

where:

r and **c** identify the spare row or column of interest.

pos is a user-defined `RaErrorPosition` variable previously initialized with the desired row or column and mask values being replaced by **r** or **c**.

`ra_spare_position_set()` returns the `RaSpareRow` or `RaSpareCol` with its `RaErrorPosition` values set to the values in **pos**.

`ra_spare_position_get()` returns an `RaErrorPosition` with the values currently set for **r** or **c**.

Example

The following example sets the replaced row and Per-I/O Spare Mask for spare row `r`:

```
RaSpareRow r = ra_spare_row_make(...); // Details not included
RaErrorPosition pos;
pos.mask = 0xFF;
pos.rcnum = 17;
ra_spare_position_set( r, pos );
```

The following example gets the replaced row and Per-I/O Spare Mask for spare row `r`:

```
RaErrorPosition pos = ra_spare_position_get( r );
```

5.3.4.14 ra_spare_mask_count_get()

See RA Spares, RA Software.

> Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_spare_mask_count_get()` function is used to return the number of valid Per-I/O Spare Mask values for a specified spare row or spare column.

`ra_spare_mask_count_get()` can be used in conjunction with `ra_spare_mask_get()` to sequentially obtain each valid Per-I/O Spare Mask value for a specified spare row or spare column.

The value returned for a spare which replaces all data bits is 1.

Usage

```
int ra_spare_mask_count_get( RaSpareRow r );
int ra_spare_mask_count_get( RaSpareCol c );
```

where:

r and **c** identify the target spare row or column.

`ra_spare_mask_count_get()` returns the number of valid Per-I/O Spare Mask values for **r** or **c**. In Multi-DUT Test Programs, the value is returned for the first DUT in the Active DUTs Set (ADS) (although the value should be the same for all DUTs).

Example

The following example creates a 2-bit per-I/O spare row, which replaces 2 adjacent bits. It then sequentially sets the Per-I/O Spare Mask for spare row `r` to the 4 legal spare mask values for this spare (assuming row `r` is associated with a segment with an 8-bit data-width). The value returned by `ra_spare_mask_count_get()` will be 4, representing 4 valid data masks: 0x03, 0x0C, 0x30 and 0xC0:

```
RaSpareRow r = ra_spare_row_make( 0x3 );
for( int c = 0; c < ra_spare_mask_count_get( r ); ++c ){
   __int64 mask = ra_spare_mask_get( r, c );
   output(" Mask Position %d => 0x%I64x", c, mask );
}
```

5.3.4.15 ra_spare_mask_get()

See RA Spares, RA Software.

Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_spare_mask_get()` function is used in conjunction with `ra_spare_mask_count_get()` to sequentially obtain each valid Per-I/O Spare Mask value for a specified spare row or spare column.

Note: the Per-I/O Spare Mask value of the `RaSpareRow` or `RaSpareRow` argument to `ra_spare_mask_get()` is modified by `ra_spare_mask_get()`, to the `index`'th legal mask value.

Usage

```
__int64 ra_spare_mask_get( RaSpareRow r, int index );
__int64 ra_spare_mask_get( RaSpareCol c, int index );
```

where:

r and **c** identify the target spare row or column.

index is the zero-based index specifying which mask value to return.

`ra_spare_mask_get()` returns the **index**'th Per-I/O Spare Mask value for spare row **r** or spare column **c**. NULL is returned if **index** is out of range. In Multi-DUT Test Programs, the value is returned for the first DUT in the Active DUTs Set (ADS) (although the value should be the same for all DUTs). See Note: above.

Example

The following example creates a 2-bit per-I/O spare row, with a 1-bit gap between bits being replaced. Assuming this row is associated with a segment with an 8-bit data-width the value returned by `ra_spare_mask_count_get()` will be 4 and `ra_spare_mask_get()` will sequentially return the 4 mask values: 0x05, 0x0A, 0x50 and 0xA0. For each loop iteration, the Per-I/O Spare Mask of spare row R1 will be set to the same mask value being returned by `ra_spare_mask_get()`:

```
RaSpareRow R1 = ra_spare_row_make( 0x5 ); // ra_spare_row_make()
for( int i= 0; i < ra_spare_mask_count_get( R1 ); ++i ){
   __int64 mask = ra_spare_mask_get( R1, i );
   // Do something with the mask or the modified spare row ...
}
```

5.3.4.16 ra_spare_current_mask_set(), ra_spare_current_mask_get()

See Per-I/O Spare Mask, RA Spares, RA Software.

Note: these functions are not used in most redundancy applications. See Note:

Description

The `ra_spare_current_mask_set()` function is used to set the Per-I/O Spare Mask for one specified spare row or spare column.

Note: normally, the RA performed by `ra_execute()` sets the Per-I/O Spare Mask for a given spare, as that spare is allocated to make a repair and moved from the Spares List to the Repair List. `ra_spare_current_mask_set()` is available for advanced users only.

The `ra_spare_current_mask_get()` function is used to get the current Per-I/O Spare Mask for a specified spare row or column.

Usage

```
void ra_spare_current_mask_set( RaSpareRow r, __int64 mask );
void ra_spare_current_mask_set( RaSpareCol c, __int64 mask );
__int64 ra_spare_current_mask_get( RaSpareRow r );
__int64 ra_spare_current_mask_get( RaSpareCol c );
```

where:

r and **c** identify the target spare row or column.

mask is the desired Per-I/O Spare Mask value. Legal values are determined by the RA software based on how the spare was defined, see `ra_spare_row_make()` or `ra_spare_col_make()`, and Per I/O Spares. The legal mask values for a given spare can be obtained using `ra_spare_mask_get()` and `ra_spare_mask_count_get()`. A warning is issued if **mask** is not one of the legal masks for the specified spare row or column. In Multi-DUT Test Programs the mask is set for all DUT(s) in the Active DUTs Set (ADS).

`ra_spare_current_mask_get()` returns the current Per-I/O Spare Mask for the specified spare row or column. The low order mask bit represents D0, etc. Only the low 36 bits are used. In Multi-DUT Test Programs the mask for the first DUT in the Active DUTs Set (ADS) is returned.

Example

```
RaSpareRow R1 = ra_spare_row_make( 0x3 );
__int64 mask = ra_spare_current_mask_get( R1 ); // Returns 0x3

// Lots of missing details re RA config, etc.
// Setup and Execute RA...
ra_execute(); // Args not shown
```

```
// If spare row R1 was allocated for a repair it was moved from
// the Spares List to the pending Repair List. The mask now reflects
// which data bits were actually repaired by R1:
__int64 mask = ra_spare_current_mask_get( R1 );
```

5.3.4.17 ra_shortest_spare_row_get(), ra_shortest_spare_col_get()

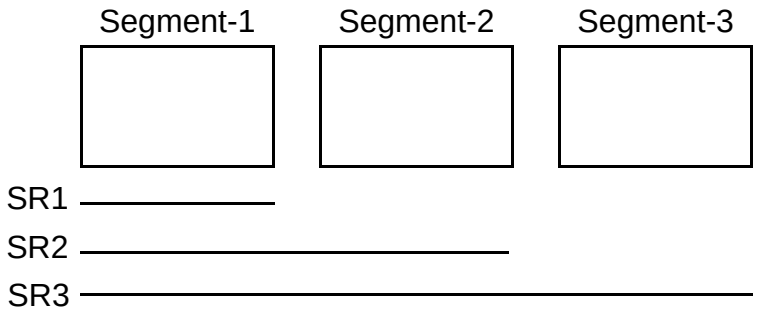
See RA Spares, RA Software.

Note: these functions are not used in most redundancy applications. See Note:

Description

The `ra_shortest_spare_row_get()` function considers the spare row(s) currently available in the Spares List and identifies and returns the spare row with the fewest linkages beyond the specified segment. See Linked Segments. Similarly, the `ra_shortest_spare_col_get()` function returns the spare column with the fewest linkages beyond the specified segment.

In this context, spare elements are sorted from shortest to longest based upon linkage. In the following example, spare row SR1 is the shortest spare row for Segment-1, but spare row SR2 is the shortest for Segment-2:



Usage

```
RaSpareRow ra_shortest_spare_row_get( RaSegment s );
RaSpareCol ra_shortest_spare_col_get( RaSegment s );
```

where:

s identifies the segment of interest.

`ra_shortest_spare_row_get()` returns the shortest spare row associated with segment **s**. In Multi-DUT Test Programs the spare for the first DUT in the Active DUTs Set (ADS) is returned.

`ra_shortest_spare_col_get()` returns the shortest spare column associated with segment **s**. In Multi-DUT Test Programs the spare for the first DUT in the Active DUTs Set (ADS) is returned.

Example

```
RaSpareRow sr = ra_shortest_spare_row_get( s );
```

5.3.5 RA Execution And Results

See Redundancy Analysis (RA), RA Software.

The following functions are directly related to RA execution and obtaining RA results:

Commonly Used
- ra_execute()
- ra_result_get()
- ra_reset()
- ra_repaired_rows_get()
- ra_repaired_cols_get()

Used Mostly During Debug
- ra_dump()
- ra_segment_dump()
- ra_spare_dump()

Rarely Used (see Note:)
- ra_error_count_get()
- ra_must_repair()
- ra_must_repair_needed()
- ra_segment_reset()
- ra_repair_done()
- ra_bad_segments_count_get()
- ra_bad_segment_get()
- ra_segment_repair_done()
- ra_error_add()
- ra_scan_area_callback()
- ra_scan_area_callback_func_set()
- ra_scan_area_callback_func_get()
- ra_scan_rc_func_set()
- ra_scan_rc_func_get()
- ra_worst_row_get()
- ra_worst_col_get()
- ra_best_row_wipeout()
- ra_best_col_wipeout()
- ra_wipeout_get()
- ra_spare_rows_required()

ra_spare_cols_required()
ra_failed_rows_count_get()
ra_failed_cols_count_get()
ra_worst_rows_get()
ra_worst_cols_get()
ra_row_wipeout()
ra_col_wipeout()

5.3.5.1 ra_execute()

See RA Execution And Results, RA Software.

Note: this function is commonly used in redundancy applications. See Note:

Description

The `ra_execute()` function performs a redundancy analysis (RA) to determine whether each active DUT is repairable given the available spare resources (Spares List). For repairable DUT(s) `ra_execute()` moves the spare(s) allocated for repair from the Spares List to the pending Repair List.

In Multi-DUT Test Programs, RA is performed concurrently on each DUT in the Active DUTs Set (ADS) with separate results and Spares List/Repair List kept for each DUT. See RA vs. Magnum 1/2 Parallel Test.

Note the following:

- `ra_execute()` is executed as part of a series of steps used to implement a complete RA solution. The information below deals with how `ra_execute()` operates and how this operation may be modified. Prior to using `ra_execute()` the concepts and details of the entire RA process must be well known. Read and understand Overview and Concepts, RA Pseudo-Code Example and Spares For Repair.
- `ra_execute()` should be executed only after one or more functional memory test patterns have been executed and logged errors to the ECR.
- Key RA functionality is implemented as Redundancy Call-back Functions. Default call-back functions are built-in but, in special situations, can be replaced by user-written functions.

- The following description outlines how ra_execute() operates and identifies the various Redundancy Call-back Functions available to allow user code to affect RA:

foreach DUT(s) in the Active DUTs Set (ADS) (ADS)
  foreach segment with an error in the ECR Mini-RAM:

  Scan the main ECR for this segment, for all DUT(s) in the ADS, and cache the errors in the Error List. For performance, this step is done only once for each segment, with cached errors used thereafter. While scanning, basic repair limits are assessed to detect when a segment can't be repaired because gross limits are exceeded. Scanning for unrepairable segment(s) is halted before all errors are cached.

  foreach DUT in the ADS with errors in this segment:
    Do Must-repair analysis on this segment for this DUT
      Call-back:
        RaMustRepairFunc: performs the Must-repair analysis and allocates spare elements for repair
        RaRowAvailableFunc: determines which spare rows are usable to repair this segment.
        RaColAvailableFunc: determines which spare columns are usable to repair this segment

      Must-repair analysis executes ra_spare_use() if a repair is identified
        Call-backs:
          RaRepairFunc: selects which spare to use and moves it from the Spares List to the Repair List
          RaRowUseOK: approves use of a specific spare row
          RaColUseOK: approves use of a specific spare column

      If the segment is unrepairable and the maximum unrepairable segments limit is reached, stop all RA for this DUT.

    Do Sparse-repair analysis on this segment for this DUT
      Call-back:
        RaSparseFunc: performs the Sparse-repair analysis and allocates spare elements for repair
        RaEvalFunc(): indirectly determines which segment to process next during Sparse-repair

      Call ra_spare_use() if a repair is identified
        Call-backs:
          RaRepairFunc: selects which spare to use and moves it

```
      from the Spares List to the Repair List
   RaRowAvailableFunc: determines which spare rows are
      usable to repair this segment.
   RaColAvailableFunc: determines which spare columns
      are usable to repair this segment
   RaRowUseOK: approves use of a specific spare row
   RaColUseOK: approves use of a specific spare column
```

```
If the segment is unrepairable and the maximum unrepairable
segments limit is reached, stop all RA for this DUT.
```

The `ra_execute()` process is as follows:

- If the ECR Mini-RAM is enabled (see `ra_config_set()`) it determines which segments have errors which are to be scanned from the ECR. If not, the Row RAM and Column RAM are used.
- Using `ra_scan_area_callback()`, errors are read from the ECR for all DUT(s) in the ADS and cached in the Error List for each segment.
- While the ECR scan is occurring, basic repair limits are assessed to detect when a segment can't repaired because gross limits are exceeded. Scanning for unrepairable segment(s) is halted before all errors are cached. This analysis has two goals:
  - Identify non-repairable devices as early as possible
  - Identify any rows or columns which can *only* be repaired one way; i.e. using only a spare row or only a column.
- The `ra_execute()` function will immediately stop analyzing errors for a given DUT if, at any time, it determines that the device is unrepairable. This will occur any time the number of unrepairable segments exceeds the maximum limit set using `ra_config_set()`. Unrepairable segments are always added to the Bad Segment List (see Spare Segments).
- No actual repairs are identified during the ECR scan process. Once the scan is complete, errors for repairable segments are analyzed in two phases:
  - Must-repair
  - Sparse-repair (with Must-repair again as necessary). More below.
- Any time a repair is identified the `ra_spare_use()` function is called to move the identified spare row or spare column from the Spares List to the Repair List. Before this occurs, if user code has registered an `RaRepairFunc` call-back function it will be executed to select which spare is to be used. And, if user code has registered

an RaRowUseOK and/or RaColUseOK call-back function one of these will be executed to enforce spare usage rules; i.e. validate that the selected spare is actually usable.

- Any time a repair is identified and a spare is allocated to make that repair, the Must-repair analysis must be performed again because the consumption of a spare element changes the conditions used to detect a Must-repair row or column. The failures associated with the Must-repair just identified are removed from the Error List and the remaining failures are re-analyzed.
- Once Must-repair has completed for all segments of all DUT(s) in ADS the Sparse-repair analysis process executes, IF:
  - Failures remain which were not repaired during Must-repair, and...
  - The device is still repairable.

  Each segment with unrepaired failures is processed again, with Must-repair and Sparse-repair performed as needed.
- By default the built-in Sparse-repair software is executed. User code can replace this operation by registering an RaSparseFunc Call-back Function. This is for very advanced applications only.
- By default, Sparse-repair begins by identifying the *worst* segment. This default can be modified using a user-written call-back function to replace the default RaSparseFunc function (see Redundancy Call-back Functions).
- When Sparse-repair identifies a new repair, and a spare element is allocated to make that repair, Must-repair analysis is done again. This is needed because the consumption of a spare element may cause errors which previously had multiple repair options to become Must-repair. Thus, Sparse-repair is interspersed with Must-repair until all segments are error-free, or a segment is determined to be unrepairable.
- During both Must-repair and Sparse-repair the Repair List keeps a record of which spare elements were allocated to repair each error. Assuming the DUT had errors, and the analysis determined that it is repairable, the Repair List holds the spare(s) information needed to actually repair the device.
- When a repair involves Linked Segments, the use of a linked spare element impacts multiple segments. The Error List, Spares List, Repair List, and Unusable List for all segments linked by the spare are updated to reflect repairs made in the current segment.
- Once RA has completed for all segments of all DUT(s) in the ADS, the ra_execute() function returns. User code then executes ra_result_get() to determine whether a given DUT had no failures (good), had failures and is not repairable (bad), had failures and is repairable (needs repair), or was not analyzed.

Note: the actual device repair is ***not*** performed by the `ra_execute()` function. This must be performed by user-written software. It is the contents of the *pending* Repair List which contain the RA repair results; i.e. which spare rows and spare columns were allocated to repair each segment of each DUT. Repair List contents are accessed using Repair List Functions.

- The Repair List is maintained until `ra_reset()` is executed. This allows test->repair->retest->re-repair... etc. to continue until either the device is successfully repaired or all spare elements are consumed and the device is considered unrepairable.
- During both Must-repair and Sparse-repair, user-written code may execute `ra_unusable_set()` to move unusable spares from the Spares List to the Unusable List and `ra_usable_set()` to move spares back from the Unusable List to the Spares List. This is done using one of the Redundancy Call-back Functions. Spares in the Unusable List are not considered available to repair the DUT. The Unusable List list is cleared using `ra_reset()` or `ra_segment_reset()`.

Usage

```
void ra_execute(
   RaSparseFunc sparse_func DEFAULT_VALUE( ra_row_first_sparse ),
   RaEvalFunc() eval_func DEFAULT_VALUE( ra_linear_eval() ),
   RaRepairFunc row_to_use_func DEFAULT_VALUE( ra_shortest_row_use ),
   RaRepairFunc col_to_use_func DEFAULT_VALUE( ra_shortest_col_use ),
   RaMustRepairFunc must_repair_func DEFAULT_VALUE( ra_must_repair ) );
```

where:

`sparse_func` is optional and registers a user-written call-back function that will be executed instead of the default RaSparseFunc Call-back Function. See Redundancy Call-back Functions. Setting the argument to 0 restores the default call-back (`ra_row_first_sparse`) and un-registers any user-defined call-back function.

`eval_func` is optional and registers a user-written call-back function that will be executed instead of the default RaEvalFunc Call-back Function. See Redundancy Call-back Functions. Setting the argument to 0 restores the default call-back (`ra_linear_eval()`) and un-registers any user-defined call-back function.

`row_to_use_func` and **`col_to_use_func`** are optional and register a user-written call-back function that will be executed instead of the default RaRepairFunc Call-back Functions. See Redundancy Call-back Functions. Setting the argument to 0 restores the default call-

back (ra_shortest_row_use or ra_shortest_col_use) and un-registers any user-defined call-back function.

must_repair_func is optional and registers a user-written call-back function that will be executed instead of the default RaMustRepairFunc Call-back Function. See Redundancy Call-back Functions. Setting the argument to 0 restores the default call-back (ra_must_repair) and un-registers any user-defined call-back function.

Example

See RA Pseudo-Code Example

5.3.5.2 ra_result_get()

See RA Execution And Results, RA Software.

Note: this function is commonly used in redundancy applications. See Note:

Description

After ra_execute() has performed a redundancy analysis (RA), the ra_result_get() function is used to retrieve the overall RA result for one DUT, or for a specified segment of one DUT. The possible results are:

- The DUT or segment has no defects and needs no repair (t_ra_good)
- The DUT or segment has defects and is repairable (t_ra_repairable)
- The DUT or segment has defects but is not repairable (t_ra_unrepairable)
- The DUT or segment was not analyzed (t_ra_not_analyzed)

When getting results for a specific segment:

- The segment will be reported as good (t_ra_good) if no repairs are applied to it.
- A segment will be reported as repairable (t_ra_repairable) if one or more spares have been allocated to repair it.
- A segment which itself has NO errors will be reported as repairable (t_ra_repairable) when a spare which links the segment to another segment is used to repair any of the other segments.

Usage

```
RaResult ra_result_get();
RaResult ra_result_get( RaSegment s );
```

where:

s identifies the segment of interest.

`ra_result_get()` returns an `RaResult`. In Multi-DUT Test Programs, the result is returned for the first DUT in the Active DUTs Set (ADS).

Example

See RA Pseudo-Code Example.

5.3.5.3 ra_error_count_get()

See RA Execution And Results, RA Software.

> Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_error_count_get()` function is used to get a count of remaining failures (i.e. unrepaired) for a DUT or a segment. Two versions are available:

- Get the number of failing addresses or failing data bits for one DUT
- Get the number of failing addresses or failing data bits for a specified segment

The values returned by these function represent the number of errors in the Error List for a specified segment, not necessarily the number of errors currently in the ECR. If errors have not yet been scanned (retrieved) from the ECR, the values returned by these functions will be 0. The number of errors returned by these functions will differ from the number of errors in the ECR for several reasons:

- The RA algorithm is optimized to minimize the number of errors read (`ra_scan_area_callback()` via `ra_execute()`) from the ECR; i.e. don't get errors that aren't needed. When the RA algorithm determines that the number of errors in a row or column exceeds the number of available spare columns or rows, no more errors are read/stored in the Error List for that row or column.

- As the RA proceeds, errors are removed from the Error List as the algorithm allocates spares to repair bad row(s) and/or column(s).

Usage

```
int ra_error_count_get( BOOL bit_count DEFAULT_VALUE( FALSE) )
int ra_error_count_get(RaSegment s,
                       BOOL bit_count DEFAULT_VALUE( FALSE) );
```

where:

`bit_count` is optional and, if specified, determines whether the value returned is the number of failing addresses (FALSE, default) or number of failing bits (TRUE).

`s` identifies a segment of interest.

`ra_error_count_get()` returns the remaining number (i.e. unrepaired) of failing addresses or failing data bits for a DUT or the specified segment. In Multi-DUT Test Programs, the value is returned for the first DUT in the Active DUTs Set (ADS).

Example

The following example returns the number of remaining error addresses for the first DUT in the Active DUTs Set (ADS):

```
int count = ra_error_count_get();
```

The following example returns the number of remaining error bits for the first DUT in the Active DUTs Set (ADS):

```
int count = ra_error_count_get( TRUE );
```

The following example returns the number of remaining error addresses for segment S1 of the first DUT in the Active DUTs Set (ADS):

```
RaSegment S1 = ra_segment_make(); // Parameters not shown
// Lots of code not shown...
int count = ra_error_count_get( S1 );
```

5.3.5.4 ra_dump()

See RA Execution And Results, RA Software.

Note: this function is commonly used in redundancy applications, during debug. See Note:

Description

The `ra_dump()` function is used to output information about both the RA configuration and RA results. The output is displayed in the controller output window of the site executing the function.

In Multi-DUT Test Programs the output is from the first DUT in the Active DUTs Set (ADS).

Note that `ra_dump()` executes `ra_segment_dump()` for each segment of the DUT.

Usage

```
void ra_dump( BOOL hex DEFAULT_VALUE( FALSE ) );
```

where:

hex is optional and, if used, determines whether numerical values are output in decimal (FALSE, default) or hexadecimal (TRUE). Mask values are always output in hexadecimal.

Example

The following is example `ra_dump()` output showing the various information generated:

```
-----------------------------------------------------
Number of Segments Per DUT: 16
Number of Spare Rows Per DUT: 16
Number of Spare Columns Per DUT: 14
Dut #0: Repairable
 Unrepairable Segments Found = 0, Unrepairable Segments Allowed = 0
    Segment #1: Repairable
        rmin = 0, rmax = 1023, cmin = 0, cmax = 255
        Spare Rows - Unused:
                0 Remaining or Defined
        Spare Rows - Unusable:
                0 Remaining or Defined
        Row Repairs - Pending:
            Spare Row #1:
                Current Mask Position = 0xFF, R/C Number = 333
                Number of Mask Positions = 1
                Length in Segments = -1, Number of RUT/CUT = 1,
```

```
Blanks Between RUT/CUT = 0
        Row Repairs - Done:
                0 Remaining or Defined
        Spare Columns - Unused:
            Spare Column #1:
                Current Mask Position = 0x1, R/C Number = -1
                Number of Mask Positions = 8
                Length in Segments = 1, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
            Spare Column #9:
                Current Mask Position = 0x1, R/C Number = -1
                Number of Mask Positions = 8
                Length in Segments = 2, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
            Spare Column #10:
                Current Mask Position = 0x1, R/C Number = -1
                Number of Mask Positions = 8
                Length in Segments = 2, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
            Spare Column #13:
                Current Mask Position = 0x1, R/C Number = -1
                Number of Mask Positions = 8
                Length in Segments = 4, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
            Spare Column #14:
                Current Mask Position = 0x1, R/C Number = -1
                Number of Mask Positions = 8
                Length in Segments = 1, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
        Spare Columns - Unusable:
            Spare Column #2:
                Current Mask Position = 0x4, R/C Number = 800
                Number of Mask Positions = 8
                Length in Segments = 1, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
        Column Repairs - Pending:
                0 Remaining or Defined
        Column Repairs - Done:
                0 Remaining or Defined
    Segment #2: Repairable
        rmin = 0, rmax = 1023, cmin = 256, cmax = 511
```

```
        Spare Rows - Unused:
                0 Remaining or Defined
        Spare Rows - Unusable:
                0 Remaining or Defined
        Row Repairs - Pending:
            Spare Row #2:
                Current Mask Position = 0xFF, R/C Number = 1023
                Number of Mask Positions = 1
                Length in Segments = -1, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
        Row Repairs - Done:
                0 Remaining or Defined
        Spare Columns - Unused:
            Spare Column #1:
                Current Mask Position = 0x1, R/C Number = -1
                Number of Mask Positions = 8
                Length in Segments = 1, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
            Spare Column #9:
                Current Mask Position = 0x1, R/C Number = -1
                Number of Mask Positions = 8
                Length in Segments = 2, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
            Spare Column #10:
                Current Mask Position = 0x1, R/C Number = -1
                Number of Mask Positions = 8
                Length in Segments = 2, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
            Spare Column #13:
                Current Mask Position = 0x1, R/C Number = -1
                Number of Mask Positions = 8
                Length in Segments = 4, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
            Spare Column #14:
                Current Mask Position = 0x1, R/C Number = -1
                Number of Mask Positions = 8
                Length in Segments = 1, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
        Spare Columns - Unusable:
            Spare Column #2:
                Current Mask Position = 0x4, R/C Number = 800
```

```
                Number of Mask Positions = 8
                Length in Segments = 1, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
        Column Repairs - Pending:
                0 Remaining or Defined
        Column Repairs - Done:
                0 Remaining or Defined
... snip ...
-------------------------------------------------
```

5.3.5.5 ra_segment_dump()

See RA Execution And Results, RA Software.

Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_segment_dump()` function is used to output information about the configuration and RA results for a specified segment. The output is displayed in the controller output window of the site executing the function.

In Multi-DUT Test Programs the output is from the first DUT in the Active DUTs Set (ADS).

Note that `ra_dump()` executes `ra_segment_dump()` for each segment of the DUT.

Usage

```
void ra_segment_dump(  RaSegment s,
                       BOOL hex DEFAULT_VALUE( FALSE ));
```

where:

s identifies the segment of interest.

hex is optional and, if used, determines whether numerical values are output in decimal (FALSE, default) or hexadecimal (TRUE). Mask values are always output in hexadecimal.

Example

The following is example `ra_segment_dump()` output showing the various information generated:

```
Segment #1: Repairable
        rmin = 0, rmax = 1023, cmin = 0, cmax = 255
        Spare Rows - Unused:
                0 Remaining or Defined
        Spare Rows - Unusable:
                0 Remaining or Defined
        Row Repairs - Pending:
            Spare Row #1:
                Current Mask Position = 0xFF, R/C Number = 333
                Number of Mask Positions = 1
                Length in Segments = -1, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
        Row Repairs - Done:
                0 Remaining or Defined
        Spare Columns - Unused:
            Spare Column #1:
                Current Mask Position = 0x1, R/C Number = -1
                Number of Mask Positions = 8
                Length in Segments = 1, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
            Spare Column #9:
                Current Mask Position = 0x1, R/C Number = -1
                Number of Mask Positions = 8
                Length in Segments = 2, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
            Spare Column #10:
                Current Mask Position = 0x1, R/C Number = -1
                Number of Mask Positions = 8
                Length in Segments = 2, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
            Spare Column #13:
                Current Mask Position = 0x1, R/C Number = -1
                Number of Mask Positions = 8
                Length in Segments = 4, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
            Spare Column #14:
                Current Mask Position = 0x1, R/C Number = -1
```

```
                Number of Mask Positions = 8
                Length in Segments = 1, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
        Spare Columns - Unusable:
            Spare Column #2:
                Current Mask Position = 0x4, R/C Number = 800
                Number of Mask Positions = 8
                Length in Segments = 1, Number of RUT/CUT = 1,
Blanks Between RUT/CUT = 0
        Column Repairs - Pending:
                0 Remaining or Defined
        Column Repairs - Done:
                0 Remaining or Defined
```

5.3.5.6 ra_spare_dump()

See RA Spares, RA Software.

Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_spare_dump()` function is used to output information about a specified spare row or spare column, including which defective row(s) or column(s) were repaired by the spare row or column. The output is displayed in the controller output window of the site executing the function.

In Multi-DUT Test Programs the output is from the first DUT in the Active DUTs Set (ADS).

Each spare records the RaErrorPosition it is replacing; i.e. the bad row or column and the Per-I/O Spare Mask indicating which data bits it is replacing.

Usage

```
void ra_spare_dump( RaSpareRow r );
void ra_spare_dump( RaSpareCol c );
```

where:

r and **c** identify the spare row or column of interest.

Example

The following is an example of `ra_spare_dump()` output showing the information generated for two spare rows:

```
Spare Row #2:
   Current Mask Position = 0xFFFF, R/C Number = 12
   Number of Mask Positions = 1
   Number of RUT/CUT = 1, Blanks Between RUT/CUT = 0
Spare Row #1:
   Current Mask Position = 0xFFFF, R/C Number = 67
   Number of Mask Positions = 1
   Number of RUT/CUT = 1, Blanks Between RUT/CUT = 0
```

5.3.5.7 ra_must_repair()

See RA Execution And Results, Repair List Functions, RA Software.

Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_must_repair()` function performs RA on a specified segment and identifies the spare(s) to repair all Must-repair rows and columns in the given segment.

`ra_must_repair()` is also the default RaMustRepairFunc Call-back Function executed during the RA invoked using `ra_execute()`. `ra_must_repair()` executes `ra_spare_use()` when a repair is identified to move the allocated spare from the Spares List to the Repair List.

The `ra_row_func` and `ra_col_func` arguments allow user code to register `RaRepairFunc` call-back functions (see Redundancy Call-back Functions) to be executed instead of the built-in functions. These functions are used, when multiple row (column) spares exist, to determine which specific spare row (column) is selected to make a given repair.

In Multi-DUT Test Programs, the Must-repair analysis is performed on the first DUT in the Active DUTs Set (ADS) only.

Usage

The following function is the built-in RaMustRepairFunc Call-back Function, executed during `ra_execute()`:

```
BOOL ra_must_repair(   RaSegment s,
                       RaSpareRowArray& spare_rows_avail,
                       RaSpareColArray& spare_cols_avail,
RaRepairFunc row_to_use_func DEFAULT_VALUE( ra_shortest_row_use ),
RaRepairFunc col_to_use_func DEFAULT_VALUE( ra_shortest_col_use )
);
```

where:

s identifies the segment to analyze.

spare_rows_avail and **spare_cols_avail** are arrays containing the spare rows and columns that the Must-repair analysis can consider usable for repair during the analysis.

row_to_use_func is optional and, if used, specifies a user-written `RaRepairFunc` call-back function which can be used to determine the strategy for deciding which spare row is selected to make a given repair. See Description and Redundancy Call-back Functions. If no argument (or NULL) is specified the built-in `ra_shortest_row_use` function is used.

col_to_use_func is optional and, if used, specifies a user-written `RaRepairFunc` call-back function which can be used to determine the strategy for deciding which spare column is selected to make a given repair. See Description and Redundancy Call-back Functions. If no argument (or NULL) is specified the default call-back `ra_shortest_col_use` is used.

`ra_must_repair()` returns FALSE if the segment is determined to be unrepairable, otherwise TRUE is returned.

Example

???

5.3.5.8 ra_must_repair_needed()

See RA Execution And Results, Repair List Functions, RA Software.

Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_must_repair_needed()` function performs a limited Must-repair analysis and simly returns TRUE if a Must-repair condition exists in the specifed segment. In Multi-DUT Test Programs, the analysis is performed on the first DUT in the Active DUTs Set (ADS) only.

Usage

The following function performs a limited Must-repair RA to determine whether a Must-repair is required, given the specified spare rows and columns. It does not return any other analysis results:

```
BOOL ra_must_repair_needed( RaSegment s,
                            RaSpareRowArray& spare_rows_avail,
                            RaSpareColArray& spare_cols_avail );
```

where:

`s` identifies the segment to analyze.

`spare_rows_avail` and **`spare_cols_avail`** are arrays containing the spare rows and columns that the Must-repair analysis can consider usable for repair during the analysis.

`ra_must_repair_needed()` returns TRUE if a Must-repair is identified otherwise FALSE is returned.

Example

???

5.3.5.9 ra_reset()

See RA Execution And Results, RA Software.

Note: this function is commonly used in redundancy applications. See Note:

Description

The `ra_reset()` function resets the Error List, Repair List, Spares List, Unusable List and Bad Segment List, typically in preparation for testing a new set of DUT(s).

In Multi-DUT Test Programs, by default, `ra_reset()` resets these lists for all DUTs in the Pin Assignment Table; i.e. the Active DUTs Set (ADS) and Ignored DUTs Set (IDS) are ignored. Setting the optional `all_duts` argument to FALSE can be used to reset the lists for only the DUT(s) in the Active DUTs Set (ADS). See RA vs. Magnum 1/2 Parallel Test.

Note that `ra_reset()` executes `ra_segment_reset()` for each segment of the appropriate DUT(s).

Note: any change made using `ra_max_bad_segments_set()` is temporary; executing `ra_reset()` **always** restores the `max_bad_segs` value to the value originally set using `ra_config_set()`.

Usage

```
void ra_reset( BOOL all_duts DEFAULT_VALUE( TRUE ) );
```

where:

`all_duts` is optional and only useful in Multi-DUT Test Programs. If used, **`all_duts`** determines whether the Error List, Repair List, Spares List, Unusable List and Bad Segment List are reset for all DUTs in the program (TRUE, default) or for only the DUT(s) in the Active DUTs Set (ADS) (FALSE).

Example

```
ra_reset();
```

5.3.5.10 ra_segment_reset()

See RA Execution And Results, RA Software.

Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_segment_reset()` function resets the Error List, Repair List, Spares List, Unusable List and Bad Segment List for a specified segment.

In Multi-DUT Test Programs, by default, `ra_segment_reset()` reset the lists for only the DUT(s) in the Active DUTs Set (ADS). Setting the optional `all_duts` argument to TRUE

can be used to resets these lists for all DUTs in the Pin Assignment Table; i.e. the Active DUTs Set (ADS) and Ignored DUTs Set (IDS) are ignored. See RA vs. Magnum 1/2 Parallel Test.

Note that `ra_reset()` executes `ra_segment_reset()` for each segment of the appropriate DUT(s).

Usage

```
void ra_segment_reset( RaSegment s,
                       BOOL all_duts DEFAULT_VALUE( FALSE ) );
```

where:

`s` identifies the segment to be reset.

`all_duts` is optional and only useful in Multi-DUT Test Programs. If used, **`all_duts`** determines whether the specified segment is reset for all DUTs in the program (TRUE, default) or for only the DUT(s) in the Active DUTs Set (ADS) (FALSE).

Example

The following example resets every segment for all DUT(s) in the Active DUTs Set (ADS):

```
for( int i = 0; i < ra_segment_count_get(); ++i )
   ra_segment_reset( ra_segment_get( i ), FALSE );
```

5.3.5.11 ra_repair_done()

See RA Execution And Results, RA Software.

Note: this function is commonly used in redundancy applications. See Note:

Description

The `ra_repair_done()` function is used to advise the RA software that the pending repairs in the Repair List have been made to the DUT. This moves these repairs from the pending Repair List to the done Repair List. Note the following:

- `ra_repair_done()` is normally executed before re-testing a [previously] repaired device, to prepare the Repair List for a new RA. Conversely, executing `ra_repair_done()` is not useful except when a DUT is going to be retested and re-repaired.
- `ra_repair_done()` has no effect on the Error List, Spares List, Unusable List or Bad Segment List.
- Note that `ra_repair_done()` calls `ra_segment_repair_done()` for each spare in each segment's Repair List.
- In Multi-DUT Test Programs, only the Repair List for the first DUT in the Active DUTs Set (ADS) is processed.

Usage

```
BOOL ra_repair_done();
```

where:

`ra_repair_done()` returns TRUE if the Repair List was successfully updated, otherwise FALSE is returned, which will occur in Multi-DUT Test Programs if the Active DUTs Set (ADS) contains no DUTs.

Example

```
if( ! ra_repair_done() )
   output(" ERROR: ra_repair_done() returned FALSE");
```

5.3.5.12 ra_spare_use()

See RA Execution And Results, RA Software.

Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_spare_use()` function performs several operations:

- Executes the RaRowUseOK or RaRowUseOK call-back function (if any) to determine whether the specified spare may be used to make the target repair. If the call-back returns FALSE `ra_spare_use()` immediately terminates, returning FALSE>

- It moves a specified spare row or spare column from the Spares List to the Repair List (pending) for a specified segment; i.e. *use* the specified spare to make a repair.
- It updates the Spares List for any segment(s) which are linked by the specified spare (see Linked Segments).
- It also removes the repaired errors from the Error List for the specified segment and any segments linked by the spare (see Linked Segments).

Normally, `ra_spare_use()` is executed by `ra_execute()` as a given spare is identified, during the RA, to make a repair. User code may use `ra_spare_use()` within an RaRepairFunc Call-back Function and/or RaSparseFunc Call-back Function (see Redundancy Call-back Functions).

Note: each spare associated with one or more segment(s) (see `ra_spare_add()`) stores an RaErrorPosition, used to record the bad row or column and Per-I/O Spare Mask the spare is repairing. When user code executes `ra_spare_use()`, these parameters *must be set* for the specified spare (using `ra_spare_position_set()` or one of the other functions that set the spare's RaErrorPosition) BEFORE executing `ra_spare_use()`. Failure to do this will likely result in improper RA operation elsewhere.

In Multi-DUT Test Programs, the operation applies only to the first DUT in the Active DUTs Set (ADS).

Usage

```
BOOL ra_spare_use( RaSegment s, RaSpareRow r );
BOOL ra_spare_use( RaSegment s, RaSpareCol c );
```

where:

s identifies the segment of interest.

r and **c** identify the spare row or spare column to be *used*; i.e. moved from the Spares List to the Repair List. IMPORTANT: see the Note: above.

`ra_spare_use()` returns TRUE if the spare was successfully used.

Example

See RaSparseFunc Call-back Function Example.

5.3.5.13 ra_bad_segments_count_get()

See RA Execution And Results, RA Software.

Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_bad_segments_count_get()` function can be used to determine the number of the bad segment(s) in the Bad Segment List. See Spare Segments and `ra_bad_segment_get()`.

Usage

```
int ra_bad_segments_count_get();
```

where `ra_bad_segments_count_get()` returns the number of bad segments in the Bad Segment List. In Multi-DUT Test Programs the value is returned for the first DUT in the Active DUTs Set (ADS).

Example

```
for( int i = 0; i < ra_bad_segments_count_get(); ++i){
   RaSegment s = ra_bad_segment_get( i ); // ra_bad_segment_get()
   // Do something with the bad segment
}
```

5.3.5.14 ra_bad_segment_get()

See RA Execution And Results, RA Software.

Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_bad_segment_get()` function can be used to retrieve a bad segment from the Bad Segment List. See Spare Segments.

The number of bad segments in the Bad Segment List can be determined using `ra_bad_segments_count_get()`.

Usage

```
RaSegment ra_bad_segment_get( int index );
```

where:

`index` specifies the zero-based index into the Bad Segment List.

`ra_bad_segment_get()` returns an RaSegment from the index'th position of the Bad Segment List or NULL if index is out of range. In Multi-DUT Test Programs, the value is returned for the first DUT in the Active DUTs Set (ADS).

Example

See Example.

5.3.5.15 ra_segment_repair_done()

See RA Execution And Results, RA Software.

> Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_segment_repair_done()` function is used to move the specified spare from the pending Repair List to the done Repair List for a specified segment.

In Multi-DUT Test Programs, the spare is moved for the first DUT in the Active DUTs Set (ADS).

Normally, `ra_segment_repair_done()` is called by `ra_repair_done()` for each segment of each active DUT. The `ra_segment_repair_done()` function is used when user-written redundancy routines are used instead of the built-in versions.

`ra_segment_repair_done()` has no effect on the Error List, Spares List, Unusable List or Bad Segment List.

Usage

```
BOOL ra_segment_repair_done( RaSegment s, RaSpareRow r);
BOOL ra_segment_repair_done( RaSegment s, RaSpareRow c);
```

where:

s identifies the target segment.

r and **c** identify the spare row or spare column to be changed from pending to done in the Repair List.

`ra_segment_repair_done()` returns TRUE if the specified spare is successfully moved from pending to done. A warning is issued if **r** or **c** is not in the pending Repair List for segment **s**.

Example

```
RaSegment s = ra_segment_make(); // Details not included
RaSpareCol c = ra_spare_col_make();
// Assume c was allocated for a repair and placed in the Repair List
BOOL ok = ra_segment_repair_done( s, c );
```

5.3.5.16 ra_error_add()

See RA Execution And Results, RA Software.

> Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_error_add()` function is used to add errors to the Error List for a specified segment.

In Multi-DUT Test Programs, the error is added to the Error List for the first DUT in the Active DUTs Set (ADS).

Usage

```
BOOL ra_error_add( RaSegment s, int row, int col, __int64 data );
```

where:

s identifies the segment of interest.

row and **col** identify the target X and Y address in the Error List at which the error(s) are to be added.

data specifies which data bits are to have errors. **data** operates like a bit-mask with a logic-1 in the mask indicating an error on the corresponding data bit and a logic-0 indicating no error on that data bit. The low order mask bit represents D0, etc. Only the low 36 bits are used.

`ra_error_add()` returns FALSE if the **row**/**col** address does not fall within segment **s** or if any of the specified data bits are not valid for segment **s**.

Example

```
BOOL ok = ra_error_add( S1, 10, 20, 0xA5 );
```

5.3.5.17 ra_scan_area_callback()

See RA Execution And Results, RA Software.

Note: early Magnum RA software used the `ra_scan()` function. This was replaced by the function documented here. Any use of the `ra_scan()` will be flagged with a runtime warning and the desired RA operation will not occur.

Description

The `ra_scan_area_callback()` function is the built-in RaScanAreaCallbackFunc Call-back Function which is executed by `ra_execute()` to scan (read) errors from the Error Catch RAM (ECR) for a specified segment and put those errors into the Error List.

For special situations, it is possible for user code to register a user-written call-back function which will be executed instead of `ra_scan_area_callback()`. Special rules apply - see RaScanAreaCallbackFunc Call-back Function. A user call-back is registered using `ra_config_set()` or `ra_scan_area_callback_func_set()`.

In Multi-DUT Test Programs, the scan operation retrieves errors for the first DUT in the Active DUTs Set (ADS) but the system software manages the Active DUTs Set (ADS) during the execution of `ra_execute()`.

Usage

```
BOOL ra_scan_area_callback( PointFailure* fails,
                            int count,
                            __int64& scanmask);
```

where:

failures is a pointer to an array of PointFailure(s) containing **count** elements. The system allocates and manages memory for 16K PointFailure at a time, which is the maximum number of errors scanned from the ECR at one time. If necessary, ra_scan_area_callback() will be executed multiple times to process the errors for a given segment.

scanmask is the addess of an __int64 variable which is not intended for user applications and thus not documented further.

ra_scan_area_callback() returns TRUE as long as un-scanned errors remain in the ECR for the given segment.

Example

None

5.3.5.18 ra_scan_area_callback_func_set() , ra_scan_area_callback_func_get()

See RA Execution And Results, RA Software.

Note: these functions are not used in most redundancy applications. See Note:

Description

By default, the ra_scan_area_callback() function is used, by ra_execute(), to read (scan) errors from the Error Catch RAM (ECR). See Redundancy Call-back Functions.

The ra_scan_area_callback_func_set() function may be used to register a user-written call-back function which will be executed instead of ra_scan_area_callback(). This allows user-written code to process/filter errors read from the ECR before they are added to the Error List. See RaScanAreaCallbackFunc Call-back Function.

The ra_scan_area_callback_func_get() function will return the current user-registered call-back function, if any, or 0.

In Multi-DUT Test Programs a single RaScanAreaCallbackFunc Call-back Function is registered at any given time; i.e. the Active DUTs Set (ADS) or Ignored DUTs Set (IDS) have no effect on which call-back is executed.

Usage

```
void ra_scan_area_callback_func_set(
        RaScanAreaCallbackFunc area_func );

RaScanAreaCallbackFunc ra_scan_area_callback_func_get();
```

where:

area_func is the user-written call-back function to be executed in place of ra_scan_area_callback(). A user-written RaScanAreaCallbackFunc call-back function must conform to the prototype documented in RaScanAreaCallbackFunc Call-back Function.

ra_scan_area_callback_func_get() returns the currently registered call-back function, if any. 0 is returned if the user has not registered a RaScanAreaCallbackFunc call-back function.

Example

The following code sets the call-back function to a user-written function named myScanFunc():

```
ra_scan_area_callback_func_set( myScanFunc );
```

The following code restores operation to use the original built-in function:

```
ra_scan_area_callback_func_set( 0 );
```

5.3.5.19 ra_scan_rc_func_set(), ra_scan_rc_func_get()

See RA Execution And Results, RA Software.

Note: these functions are not used in most redundancy applications. See Note:

Description

The `ra_scan_rc_func_set()` function may be used to register a user-written call-back function which will be executed instead of the built-in function to read (scan) errors from the ECR's Row RAMs and Column RAMs. See Redundancy Call-back Functions and RaScanRCFunc Call-back Function.

The `ra_scan_rc_func_get()` function will return the currently user-registered call-back function, if any, or 0.

Note that the ECR's Row RAMs and Column RAMs are not used when the ECR Mini-RAM is enabled (see `ra_config_set()`).

In Multi-DUT Test Programs a single RaScanRCFunc Call-back Function call-back is registered at any given time; i.e. the Active DUTs Set (ADS) or Ignored DUTs Set (IDS) have no effect on which call-back is executed.

Usage

```
void ra_scan_rc_func_set( RaScanRCFunc rc_func );
RaScanRCFunc ra_scan_rc_func_get();
```

where:

`rc_func` is the user-written call-back function to be executed instead of the built-in function. A user-written `RaScanRCFunc` call-back function must conform to the prototype documented in RaScanRCFunc Call-back Function.

`ra_scan_rc_func_get()` returns the currently registered call-back function, if any. 0 is returned if the user has not registered a `RaScanRCFunc` call-back function.

Example

The following code sets the RC scan call-back function to a user-written function named `myRCScanFunc()`:

```
ra_scan_rc_func_set( myRCScanFunc );
```

The following code restores operation to use the original built-in function:

```
ra_scan_rc_func_set( 0 );
```

5.3.5.20 ra_worst_row_get(), ra_worst_col_get()

See RA Execution And Results, RA Software.

Note: these functions are not used in most redundancy applications. See Note:

Description

Given a list (array) of bad rows in a specified segment the `ra_worst_row_get()` function will:

- Identify the (worst) bad row which, if replaced, will eliminate the maximum number of column failures in the specified segment. Linked Segments are considered in the evaluation.
- Consider the spare rows available in the Spares List and identify and return a spare row which can be used to replace the worst bad row. The RaRowAvailableFunc & RaColAvailableFunc Call-back Function are used during the analysis.
- Return the number of RaErrorPosition(s) which would be partially or completely repaired by replacing the identified worst row with the recommended spare row. If the segment has per-I/O spare columns, this number may not be the number of repaired columns and may also not be the number of data bits repaired; it is a count of RaErrorPosition(s).

Also note:

- The list of bad rows to be analyzed must be defined, by user code, in an `RaErrorPosArray`. See `ra_failed_rows_get()`.
- The spare selection analysis does consider Rows-Used-Together(RUT), Columns-Used-Together(CUT).
- The identified spare's RaErrorPosition is updated:
  - `rcnum` = the identified worst bad row
  - `mask` = which bits the identified spare will replace
  See RaErrorPosition and Per-I/O Spare Mask.
- The spare rows or columns considered by the analysis are filtered by the RaRowAvailableFunc call-back function, if any, specified via `ra_segment_make()`. See RaRowAvailableFunc & RaColAvailableFunc Call-back Function.

- `ra_worst_row_get()` only performs an analysis and recommends a spare for repair, it does not modify the Spares List or Repair List (see `ra_spare_use()`).

The function `ra_worst_col_get()` operates similarly on columns.

In Multi-DUT Test Programs, the analysis is performed to the first DUT in the Active DUTs Set (ADS).

Note: two other functions named `ra_worst_rows_get()` and `ra_worst_cols_get()` have very similar names, but have quite different functionality.

Usage

```
RaSpareRow ra_worst_row_get(RaSegment s,
                            RaErrorPosArray &rows,
                            int *count );
RaSpareCol ra_worst_col_get(RaSegment s,
                            RaErrorPosArray &cols,
                            int *count );
```

where:

`s` identifies the segment of interest.

`rows` and **`cols`** are a user-defined `RaErrorPosArray` containing the bad row or column information. The user code must have previously initialized the contents of **`rows`** or **`cols`**. See Description.

`count` is the address of an existing `int` variable used to return the number of errors which will be eliminated if the returned spare is actually used to repair the specified segment.

`ra_worst_row_get()` returns the spare row identified to repair the worst bad row specified in **`rows`**. The spare's RaErrorPosition is modified as noted in the Description above.

`ra_worst_col_get()` returns the spare column identified to repair the worst bad column specified in **`cols`**. The spare's RaErrorPosition is modified as noted in the Description above.

Example

```
int count;
RaErrorPosArray pos_array;
ra_failed_rows_get( S1, pos_array ); // ra_failed_rows_get()
RaSpareRow r = ra_worst_row_get( S1, pos_array, &count );
```

5.3.5.21 ra_best_row_wipeout(), ra_best_col_wipeout()

See RA Execution And Results, RA Software.

Note: these functions are not used in most redundancy applications. See Note:

Description

The `ra_best_row_wipeout()` function identifies the spare which completely repairs the most columns in a specified segment.

The `ra_best_col_wipeout()` function identifies the spare which completely repairs the most rows in a specified segment.

Two versions of these functions are provided. Only the row version is described, but the operation is similar for columns:

- Given a segment and a target bad row RaErrorPosition identify which spare row completely repairs the most columns.
- Given a segment and an array of bad column RaErrorPositions (e.g. an initialized `RaErrorPosArray` identifying bad columns) identify which spare column completely repairs the most rows. The list of bad columns to be analyzed must be defined, by user code, in an `RaErrorPosArray`. See `ra_failed_cols_get()`.

Also note:

- The spare rows or columns considered by the analysis are filtered by the RaRowAvailableFunc call-back function, if any, specified via `ra_segment_make()`. See RaRowAvailableFunc & RaColAvailableFunc Call-back Function.
- Linked Segments and Rows-Used-Together(RUT), Columns-Used-Together(CUT) are considered in the evaluation.

- The identified spare's RaErrorPosition is updated:
  - `rcnum` = the row or column which will be replaced by the recommended spare.
  - `mask` = which data bits the identified spare will replace
  See RaErrorPosition and Per-I/O Spare Mask.
- `ra_best_row_wipeout()` only performs an analysis and recommends a spare for repair, it does not modify the Spares List or Repair List (see `ra_spare_use()`).

In Multi-DUT Test Programs, the analysis is performed on the first DUT in the Active DUTs Set (ADS).

Usage

Given a bad row `RaErrorPosition`, identify the spare row which completely repairs the most columns:

```
RaSpareRow ra_best_row_wipeout(  RaSegment s,
                                 RaErrorPosition rownum );
```

Given an `RaErrorPosArray` containing bad column `RaErrorPosition`s, identify the spare column which completely repairs the most rows:

```
RaSpareCol ra_best_row_wipeout(  RaSegment s,
                                 RaErrorPosArray &cols );
```

Given a bad column `RaErrorPosition`, identify the spare column which completely repairs the most rows:

```
RaSpareCol ra_best_col_wipeout(  RaSegment s,
                                 RaErrorPosition colnum );
```

Given an `RaErrorPosArray` containing bad row `RaErrorPosition`s, identify the spare row which completely repairs the most columns:

```
RaSpareRow ra_best_col_wipeout(  RaSegment s,
                                 RaErrorPosArray &rows );
```

where:

`s` identifies the segment of interest.

`rownum` and **`colnum`** identify the bad row or column RaErrorPosition.

`rows` or **`cols`** are an array of bad rows or columns RaErrorPositions e.g. an `RaErrorPosArray`.

The first version of `ra_best_row_wipeout()` returns the spare row (`RaSpareRow`) which completely repairs the most columns in the specified bad row RaErrorPosition. The spare's RaErrorPosition is set as described in Description above.

The second version of `ra_best_row_wipeout()` returns the spare column (`RaSpareCol`) which completely repairs the most rows in the specified array of bad columns. The spare's RaErrorPosition is set as described in Description above.

The first version of `ra_best_col_wipeout()` returns the spare column (`RaSpareCol`) which completely repairs the most rows in the specified bad column RaErrorPosition. The spare's RaErrorPosition is set as described in Description above.

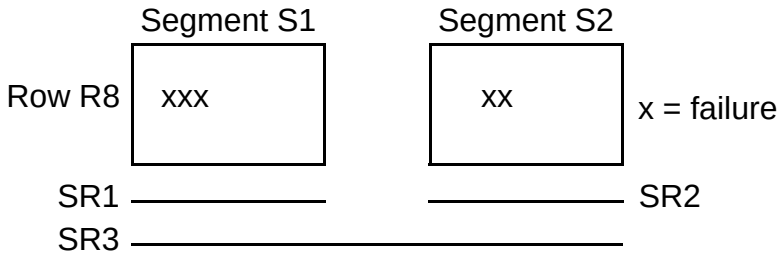
The second version of `ra_best_col_wipeout()` returns the spare row (`RaSpareRow`) which completely repairs the most columns in the specified array of bad rows. The spare's RaErrorPosition is set as described in Description above.

Both functions return NULL and a warning is issued, in the following situations:

- **s** is NULL.
- If an RaErrorPosition `rcnum` value is out of range of the specified segment.
- If an RaErrorPosition `mask` position is invalid; i.e. it doesn't match one of the valid Per-I/O Spare Masks of the available spares.

Example

The following example shows 2 segments which share one spare row (SR3) and each have a local spare row. Both segments store 8-bit data. Errors in both segments are indicated:



The following code would return SR3 with its RaErrorPosition `rcnum` value = 8 and its `mask` = 0xFF:

```
RaErrorPosition bad_row;
bad_row.rcnum = 8;
bad_row.mask = 0xFF;
RaSpareRow r = ra_best_row_wipeout( S1, bad_row );
```

5.3.5.22 ra_wipeout_get()

See RA Execution And Results, RA Software.

Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_wipeout_get()` function is used to determine the number of RaErrorPosition(s) which would be completely repaired in a specified bad row or bad column using a specified spare column or row. This function only performs an analysis.

When evaluating the use of a spare row to repair a bad row, the number of column RaErrorPosition(s) which would be completely repaired is returned. Conversely, when evaluating the use of a spare column to repair a bad column, the number of row RaErrorPosition(s) which would be completely repaired is returned.

Rows-Used-Together(RUT), Columns-Used-Together(CUT) are considered in the evaluation. When the spare includes Rows-Used-Together(RUT), Columns-Used-Together(CUT) the returned number of potential RaErrorPositions repaired includes those in the second, third, etc. row or column replaced by the RUT or CUT.

Linked Segments are considered in the evaluation. When the spare spans more than one segment the returned number of potential RaErrorPositions repaired includes those in the second, third, etc. segment which shares the specified spare.

In Multi-DUT Test Programs, the analysis is performed on the first DUT in the Active DUTs Set (ADS).

Usage

```
int ra_wipeout_get( RaSpareRow r, RaErrorPosition& rowpos );
int ra_wipeout_get( RaSpareCol c, RaErrorPosition& colpos );
```

where:

`r` and **`c`** identify the spare row or spare column to be considered in the evaluation.

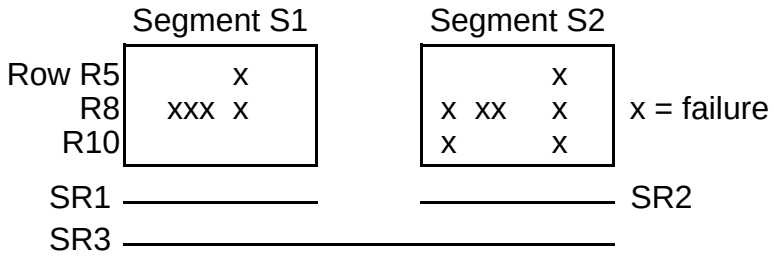
`rowpos` identifies a bad row RaErrorPosition to be considered in the analysis.

`colpos` identifies a bad column RaErrorPosition to be considered in the analysis.

`ra_wipeout_get()` returns the number of RaErrorPosition(s) which would be completely repaired if the specified spare were actually used to repair the specified RaErrorPosition.

Example

The following example shows 2 segments which are linked (see Linked Segments) by one spare row (SR3) and each have a local spare row. Both segments store 8-bit data. Errors in both segments are indicated:



Given the following RaErrorPosition...

```
RaErrorPosition bad_row;
bad_row.rcnum = 8; // R8
bad_row.mask = 0xFF;
```

The following would return 3 because spare SR1 would completely repair only 3 columns in segment S1:

```
int c = ra_wipeout_get( SR1, bad_row );
```

The following would return 2 because spare SR2 would completely repair only 2 columns in segment S2:

```
int c = ra_wipeout_get( SR2, bad_row );
```

The following would return 5 because spare SR3 would completely repair 5 columns in the two segments linked by SR3:

```
int c = ra_wipeout_get( SR3, bad_row );
```

5.3.5.23 ra_spare_rows_required(), ra_spare_cols_required()

See RA Execution And Results, RA Software.

Note: these functions are not used in most redundancy applications. See Note:

Description

The `ra_spare_rows_required()` function is used to determine the number of additional spares needed to repair a specified (bad) row.

The `ra_spare_cols_required()` function is used to determine the number of additional spares needed to repair a specified (bad) column.

These functions only perform an analysis.

When a given segment does not have per-I/O spare rows (see Per I/O Spares) each bad row may be replaced by a single spare row. Similarly for columns. When a segment does have per-I/O spare rows a given bad row may require several spare rows to repair all of the bad data bits in that row. Similarly for columns. These functions may be used to determine how many additional spares are needed to repair a specified row or column.

The term *additional* is used above to indicate that errors for which a repair has already been identified (and for which a spare has been allocated) are not considered by `ra_spare_rows_required()` or `ra_spare_cols_required()`.

In Multi-DUT Test Programs, the information is returned for the first DUT in the Active DUTs Set (ADS).

Usage

```
int ra_spare_rows_required( RaSegment s, int rownum );
int ra_spare_cols_required( RaSegment s, int colnum );
```

where:

`s` identifies the segment of interest.

`rownum` and **`colnum`** identify the target (bad) row or column.

`ra_spare_rows_required()` returns the number of spare row(s) required to repair all remaining bad data-bits in **`rownum`**. The value returned will range from 0 (no errors in **`rownum`**) to the total number of Per-I/O Spare Mask positions for **`rownum`**. Note that this function can return a value which exceeds the number of spares currently available for repair.

`ra_spare_cols_required()` returns the number of spare column(s) required to repair all remaining bad data-bits in **`colnum`**. The value returned will range from 0 (no errors in **`colnum`**) to the total number of Per-I/O Spare Mask positions for **`colnum`**. Note that this function can return a value which exceeds the number of spares currently available for repair.

Example

```
int spare_rows_reqd = ra_spare_rows_required( S1, 35 );
```

5.3.5.24 ra_failed_rows_count_get(), ra_failed_cols_count_get()

See RA Execution And Results, RA Software.

Note: these functions are not used in most redundancy applications. See Note:

Description

The `ra_failed_rows_count_get()` function returns the spare column that repairs the most rows in a specified column, irrespective of whether the entire row is repaired. The number of rows which the spare column repairs is returned in the `count` parameter.

The `ra_failed_cols_count_get()` function returns the spare row that repairs the most columns in a specified row, irrespective of whether the entire column is repaired. The number of columns which the spare row repairs is returned in the `count` parameter.

These functions only perform an analysis.

For example, assume row 6 has errors at column positions 9, 15, and 17. Executing `ra_failed_cols_count_get()` on row 6 will return 3 in the `count` parameter, even if columns 9, 15, and 17 have errors in other rows.

The returned `RaSpareRow` or `RaSpareCol` records the RaErrorPosition it would repair if the spare was actually used to make the identified repair.

Linked Segments and Rows-Used-Together(RUT), Columns-Used-Together(CUT) are considered in the evaluation. When the spare includes Rows-Used-Together(RUT), Columns-Used-Together(CUT) the number of errors returned includes the errors replaced by the second, third, etc. row or column in the RUT or CUT.

The spare rows or columns considered by these functions are first filtered by the RaRowAvailableFunc or RaRowAvailableFunc call-back functions, if any, specified via `ra_segment_make()`. See RaRowAvailableFunc & RaColAvailableFunc Call-back Function.

In Multi-DUT Test Programs, the analysis is performed on the first DUT in the Active DUTs Set (ADS).

Usage

```
RaSpareCol ra_failed_rows_count_get(  RaSegment s,
                                      RaErrorPosition& colpos,
                                      int *count );
RaSpareRow ra_failed_cols_count_get(  RaSegment s,
                                      RaErrorPosition& rowpos,
                                      int *count );
```

where:

s identifies the target segment.

rowpos and **colpos** identify the bad row or column RaErrorPosition to be considered in the analysis.

count is the address of an existing `int` variable used to return the number of rows or columns that would be repaired if the identified spare was actually used to make the repair. See Description.

`ra_failed_rows_count_get()` returns the spare column identified to make the repair. The returned `RaSpareCol` records the RaErrorPosition it would repair if the spare was actually used to make the repair.

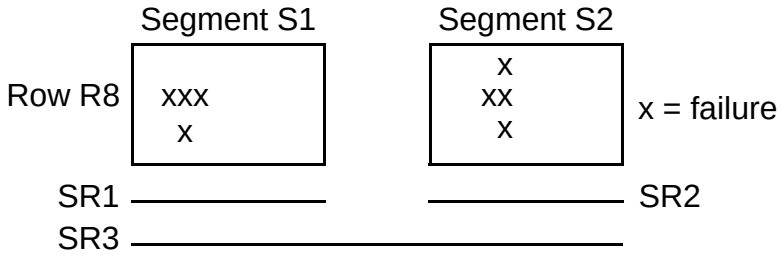
`ra_failed_cols_count_get()` returns the spare row identified to make the repair. The returned `RaSpareRow` records the RaErrorPosition it would repair if the spare was actually used to make the repair.

Both functions return NULL and a warning is issued, in the following situations:

- **s** or **count** is NULL.
- If an RaErrorPosition `rcnum` value is out of range of the specified segment.
- If an RaErrorPosition `mask` position is invalid; i.e. doesn't match one of the valid Per-I/O Spare Masks of the available spares.

Example

The following example shows 2 segments which are linked by spare row (SR3) and each have a local spare row. Both segments store 8-bit data. Errors in both segments are indicated:



The following code would return SR3 with its RaErrorPosition `rcnum` = 8 and `value` = 0xFF. The returned `count` value will be 5:

```
RaErrorPosition bad_row;
bad_row.rcnum = 8;
bad_row.mask = 0xFF;
int count;
RaSpareRow r = ra_failed_cols_count_get( S1, bad_row, &count );
```

5.3.5.25 ra_failed_rows_get(), ra_failed_cols_get()

See RA Execution And Results, RA Software.

Note: these functions are not used in most redundancy applications. See Note:

Description

The `ra_failed_rows_get()` and `ra_failed_cols_get()` functions are used to retrieve various failed row and column information from the Error List. Errors are returned as RaErrorPosition(s).

Several versions of these functions are provided (see Usage):

- The first versions below return the number of row RaErrorPosition(s) or column RaErrorPosition(s) in the Error List for a specified segment.

- The second version below returns an array containing all failing row or column RaErrorPosition(s) in the Error List for a specified segment.
- The third version below returns the number of row RaErrorPosition(s) or column RaErrorPosition(s) in the Error List for a specified segment, filtered by a specified input RaErrorPosition.
- The last version below returns an array containing all failing row or column RaErrorPosition(s) in the Error List for a specified segment, filtered by a specified input RaErrorPosition.

These four versions can be further divided into two sets:

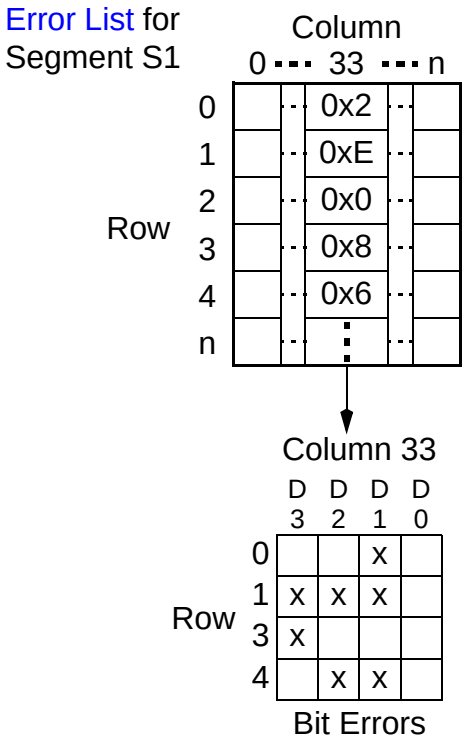
- The first two versions return information for all row RaErrorPosition(s) or column RaErrorPosition(s) in the Error List for a specified segment.
- The second two versions return information which is filtered by a specified input row RaErrorPosition or column RaErrorPosition.

In both cases, it is RaErrorPosition(s) which are retrieved and/or counted and it is the Per-I/O Spare Mask of the spare row(s) and spare column(s) associated with the segment which determines the (maximum) number of RaErrorPosition(s) which will be returned for a given row or column.

For example, if the DUT has Per I/O column spares, the maximum number of errors returned for each row equals the number of Per-I/O Spare Mask positions for the spare column; e.g. in a segment with N rows and M column mask positions, the largest possible number of failed rows returned is (N*M).

The following diagram is used to show the operation of the first two versions (see Usage below) of `ra_failed_rows_get()` and `ra_failed_cols_get()`. Segment, S1, stores a 4-bit data width (for diagram convenience). The example segment has four 1-bit per-I/O

spare columns and 2 full-data-width spare rows. The Error List for the segment contains the errors noted below. For simplicity, all errors occur in column-33:



The (unfiltered) row RaErrorPositions are:

rcnum	mask
0	0xF
1	0xF
3	0xF
4	0xF

The mask = 0xF because the row spares replace the full 4-bit data width.

The (unfiltered) column RaErrorPositions are:

rcnum	mask
33	0x2
33	0x4
33	0x8

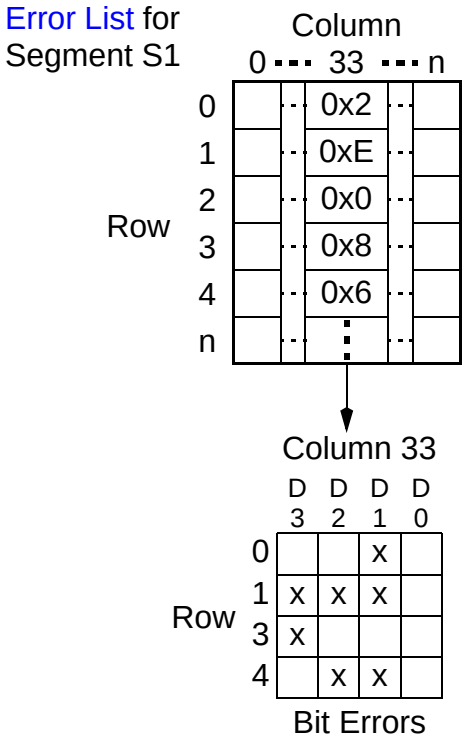
The mask values reflect the Per-I/O Spare Mask values for the spare columns available to replace columns in this segment.

The first version of `ra_failed_rows_get()` returns 4, the second returns an `RaErrorPosArray` containing the four row RaErrorPositions noted above.

The first version of `ra_failed_cols_get()` returns 3, the second returns an `RaErrorPosArray` containing the three column RaErrorPositions noted above.

The other two versions of `ra_failed_rows_get()` and `ra_failed_cols_get()` filter the values returned from the Error List by a specified row RaErrorPosition or column

RaErrorPosition. The following diagram shows this operation. The errors in the Error List are identical to the previous example:



Given:

```
RaErrorPosition col;
col.rcnum = 33;
col.mask = 0x2; // i.e. D1
RaErrorPosArray rows;
ra_failed_rows_get( S1, col, &rows );
```

The returned rows array will contain the following RaErrorPosition values:

rcnum	mask
0	0xF
1	0xF
4	0xF

As in the previous example, the mask = 0xF because the row spares replace the full 4-bit data width.

```
RaErrorPosition row;
row.rcnum = 1;
row.mask = 0xF; // All Databits
RaErrorPosArray cols;
ra_failed_cols_get( S1, row, &cols );
```

The returned cols array will contain the following RaErrorPosition values:

rcnum	mask
33	0x2
33	0x4
33	0x8

The mask values reflect the Per-I/O Spare Mask values for the spare columns available to replace columns in this segment.

The third version of `ra_failed_rows_get()` returns 3, the fourth returns an `RaErrorPosArray` containing the three row RaErrorPositions noted above.

The third version of `ra_failed_cols_get()` returns 3, the fourth returns an `RaErrorPosArray` containing the three column RaErrorPositions noted above.

In Multi-DUT Test Programs, the information is returned for the first DUT in the Active DUTs Set (ADS).

Usage

The following functions return the number of failing rows or columns in the specified column or row:

```
int ra_failed_rows_get ( RaSegment s);
int ra_failed_cols_get( RaSegment s );
```

The following functions return an array containing all failing rows or columns in the specified segment:

```
void ra_failed_rows_get ( RaSegment s, RaErrorPosArray& rows );
void ra_failed_cols_get ( RaSegment s, RaErrorPosArray& cols );
```

The following functions return the number of failing row or column RaErrorPosition(s) in the specified segment as filtered by a specified RaErrorPosition:

```
int ra_failed_rows_get ( RaSegment s, RaErrorPosition col );
int ra_failed_cols_get ( RaSegment s, RaErrorPosition row );
```

The following functions returns an array of failing row or column RaErrorPosition(s) in the specified segment as filtered by a specified RaErrorPosition:

```
void ra_failed_rows_get( RaSegment s,
                         RaErrorPosition col,
                         RaErrorPosArray& rows );
void ra_failed_cols_get( RaSegment s,
                         RaErrorPosition row,
                         RaErrorPosArray& cols );
```

where:

`s` identifies the segment of interest.

`rows` and **`cols`** are an existing `RaErrorPosArray` array used to return zero or more row RaErrorPosition(s) or column RaErrorPosition(s). The array will be automatically resized as necessary. Any prior contents are lost. The number of elements and each element in the `RaErrorPosArray` can be obtained using standard `CArray` member functions.

`col` specifies a column RaErrorPosition for which failed row errors are to be filtered.

row specifies a row RaErrorPosition for which failed column errors are to be filtered.

The versions of `ra_failed_rows_get()` and `ra_failed_cols_get()` which return an `int` are returning the number of row RaErrorPosition(s) or column RaErrorPosition(s).

Example

See Description.

5.3.5.26 ra_worst_rows_get(), ra_worst_cols_get()

See RA Execution And Results, RA Software.

> Note: these functions are not used in most redundancy applications. See Note:

Description

The `ra_worst_rows_get()` function:

- Identifies the row(s) that have the most failures in the specified segment.
- Returns the number of failures in the identified worst row.
- Returns an array of RaErrorPosition(s) (`RaErrorPosArray`). Each element represents a bad row + Per-I/O Spare Mask which has the number of failures matching the most failures value.

The function `ra_worst_cols_get()` operates similarly on columns.

Rows-Used-Together(RUT), Columns-Used-Together(CUT) do not affect the evaluation performed by `ra_worst_rows_get()` or `ra_worst_cols_get()`.

> Note: two other functions named `ra_worst_row_get()` and `ra_worst_col_get()` have very similar names, but have quite different functionality.

Usage

```
int ra_worst_rows_get( RaSegment s, RaErrorPosArray &rows );
int ra_worst_cols_get( RaSegment s, RaErrorPosArray &cols );
```

where:

s identifies the segment of interest.

rows and **cols** are a user-defined RaErrorPosArray used to return a list of worst rows or columns. See Description. The RaErrorPosArray is automatically sized by the system software and any prior contents are lost. Standard CArray member functions can be used to obtain the size and individual elements of **rows** and **cols**. See Example.

ra_worst_rows_get() and ra_worst_cols_get() return the number of errors in the identified worst row or column (*not* the number of elements in the RaErrorPosArray). In Multi-DUT Test Programs, the operation is performed only on the first DUT in the Active DUTs Set (ADS).

Example

The following example gets a list of the worst row(s) for segment s, then retrieves each element of the list:

```
RaErrorPosArray rows;
int count = ra_worst_rows_get( s, rows );
for( int i = 0; i < rows.GetSize(); ++i){
   RaErrorPosition badrow = rows[ i ];
   // Do something with badrow
}
```

5.3.5.27 ra_row_wipeout(), ra_col_wipeout()

See RA Execution And Results, RA Software.

Note: these functions are not used in most redundancy applications. See Note:

Description

The ra_row_wipeout() and ra_col_wipeout() functions target identifying how many errors could be repaired using a specified spare row or spare column.

Two overloads of each function are available. The bulleted descriptions below refer to the ra_row_wipeout() function only, but ra_col_wipeout() operates similarly for columns (swap both the words row and column):

- Identify and return the number of row RaErrorPosition(s) which would be completely repaired by replacing a specified (bad) column with a specified spare column.
- Given an array of bad column RaErrorPosition(s) and an array of spare column(s) analyze each combination of spare column vs. bad column RaErrorPosition to identify which combination completely repairs the most row RaErrorPosition(s). The identified combination is returned in the `best_cols` argument. If a tie exists, `best_cols` contains all combinations which resulted in the same number of completely repaired row RaErrorPosition(s). The function returns the total number of row RaErrorPosition(s). which would be repaired if any one of the repairs in `best_cols` was implemented.

Linked Segments are considered in the evaluation.

The spare rows or columns considered by these functions are first filtered by the RaRowAvailableFunc or RaRowAvailableFunc call-back functions, if any, specified via `ra_segment_make()`. See RaRowAvailableFunc & RaColAvailableFunc Call-back Function.

In Multi-DUT Test Programs the analysis is performed for the first DUT in the Active DUTs Set (ADS).

Usage

```
int ra_row_wipeout( RaSegment s,
                    RaSpareCol spare_col,
                    RaErrorPosition& bad_col );
int ra_row_wipeout( RaSegment s,
                    RaSpareColArray& spare_cols,
                    RaErrorPosArray& bad_cols,
                    RaSpareColPosArray& best_cols );
int ra_col_wipeout( RaSegment s,
                    RaSpareRow spare_row,
                    RaErrorPosition& bad_row );
int ra_col_wipeout( RaSegment s,
                    RaSpareRowArray& spare_rows,
                    RaErrorPosArray& bad_rows,
                    RaSpareRowPosArray& best_rows );
```

where:

s identifies the segment of interest.

spare_col and **spare_row** identify one spare column or spare row to be considered in the analysis.

bad_col and **bad_row** identify one bad RaErrorPosition to be considered in the analysis.

spare_cols and **spare_rows** are an array identifying one or more spare column(s) or spare row(s) to be considered in the analysis.

bad_cols and **bad_rows** are an array identifying one or more bad column or row RaErrorPosition(s) to be considered in the analysis.

best_cols and **best_rows** are an existing array used to return the detailed results of the analysis; i.e. one or more combinations of spare row + row RaErrorPosition or spare column + column RaErrorPosition identified during the analysis. See Descripion.

The first version of `ra_row_wipeout()` returns the number of rows which would be completely repaired if the specified bad column RaErrorPosition were repaired using the specified spare column.

The second version of `ra_row_wipeout()` returns the total number of row RaErrorPosition(s). which would be completely repaired if any one of the repairs in **best_cols** was implemented.

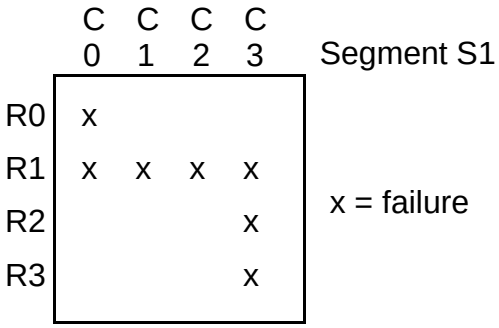
The first version of `ra_col_wipeout()` returns the number of columns which would be completely repaired if the specified bad row RaErrorPosition were repaired using the specified spare row.

The second version of `ra_col_wipeout()` returns the total number of column RaErrorPosition(s). which would be completely repaired if any one of the repairs in **best_rows** was implemented.

Examples

Example 1:

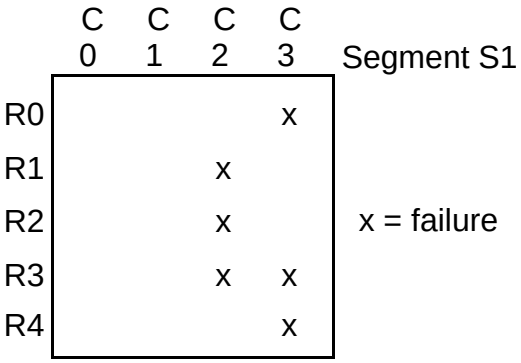
This example uses the first overload of `ra_row_wipeout()`. Replacing column 3 will completely repair 2 rows (R2 & R3 but not R0 or R1). Assume this segment has an 8-bit data width and each spare column replaces all 8 bits:




```
RaSpareCol SC2 = ra_spare_col_make( 0xFF ); // ra_spare_col_make()
ra_spare_add( S1, SC2 );                    // ra_spare_add()
RaErrorPosition pos;
pos.rcnum = 3; // Column 3
pos.mask = 0xFF;
int rows_fixed = ra_row_wipeout( S1, SC2, pos ); // rows_fixed = 2
```

Example 2:

This example uses the second overload of `ra_row_wipeout()`:




```
RaSpareCol SC1 = ra_spare_col_make( 0xFF ); // ra_spare_col_make()
RaSpareCol SC2 = ra_spare_col_make( 0xFF );
ra_spare_add( S1, SC1 ); // ra_spare_add()
ra_spare_add( S1, SC2 );
```

```
RaSpareColArray spare_cols;
spare_cols.Add = SC1;
spare_cols.Add = SC2;

RaErrorPosArray bad_cols;
int num = ra_worst_cols_get( S1, bad_cols ); // ra_worst_cols_get()

RaSpareColPosArray best_cols;
int count = ra_row_wipeout( S1, spare_cols, bad_cols, best_cols );

for( int i = 0; i < best_cols.GetSize(); ++i ){
   // Use each best col spare and bad RaErrorPosition pair
}
```

This example shows a tie; i.e. replacing C2 and C3 both result in completely repairing two (different) rows. The function returns two things:

- `count` = 2; i.e. 2 rows can be completely repaired (R1 & R2 or R0 & R4)
- The number of elements in `best_cols[]` = 4:
  - `best_cols[0].colnum` = 2; i.e. column 2
  - `best_cols[0].mask` = 0xFF
  - `best_cols[0].col` = SC1
  - `best_cols[1].colnum` = 3; i.e. column 3
  - `best_cols[1].mask` = 0xFF
  - `best_cols[1].col` = SC1
  - `best_cols[2].colnum` = 2; i.e. column 2
  - `best_cols[2].mask` = 0xFF
  - `best_cols[2].col` = SC2
  - `best_cols[3].colnum` = 3; i.e. column 3
  - `best_cols[3].mask` = 0xFF
  - `best_cols[3].col` = SC2

5.3.6 Repair List Functions

See Redundancy Analysis (RA), RA Software.

The main output of the RA (`ra_execute()`) is a Repair List, containing a list of spare rows and/or spare columns which have been allocated to repair a segment. Each spare in the

Repair List stores an RaErrorPosition used to record the bad row or column the spare is replacing and the Per I/O Spares mask which identifies which data bits the spare is replacing. In Multi-DUT Test Programs a separate Repair List is automatically maintained for each DUT in the test program.

It is user-written code which either uses the Repair List to repair the DUT on-line, or to export the Repair List for use in off-line repairs. The following functions are used to access the Repair List:

Commonly Used	Rarely Used (see Note:)
ra_repaired_row_count_get()	ra_repaired_row_get()
ra_repaired_col_count_get()	ra_repaired_col_get()
ra_repaired_rows_get()	ra_what_repaired_row_get()
ra_repaired_cols_get()	ra_what_repaired_col_get()
	ra_spare_repaired_errors_get()

5.3.6.1 ra_repaired_row_count_get(), ra_repaired_col_count_get()

See RA Execution And Results, Repair List Functions, RA Software.

Note: these functions are used in most redundancy applications. See Note:

Description

The ra_repaired_row_count_get() function is used to get the number of row repairs from the Repair List for a specified segment. The ra_repaired_col_count_get() function is used to get the number of column repairs from the Repair List for a specified segment.

These functions can be used in conjunction with ra_repaired_row_get(), ra_repaired_col_get() to sequentially get all repairs from the Repair List. Or, all repairs can be retrieved using ra_repaired_rows_get(), ra_repaired_cols_get().

The Repair List contains information in two states: *pending* repairs and *done* repairs. ra_repaired_row_count_get() and ra_repaired_col_count_get() have an optional argument (pending) used to select which count is returned. See Repair List. Using pending = FALSE it is possible to access previously made repairs from the done Repair List (see ra_repair_done()).

Usage

```
int ra_repaired_row_count_get(
                              RaSegment s,
                              BOOL pending DEFAULT_VALUE( TRUE ));
int ra_repaired_col_count_get(
                              RaSegment s,
                              BOOL pending DEFAULT_VALUE( TRUE ));
```

where:

s is the segment if interest.

pending is optional and, if used, specifies whether the count of pending repairs (TRUE, default) is returned or the count of done repairs (FALSE) is returned.

`ra_repaired_row_count_get()` returns the count of pending or done row repairs from the Repair List for segment **s**. In Multi-DUT Test Programs the value returned is for the first DUT in the Active DUTs Set (ADS).

`ra_repaired_row_count_get()` returns the count of pending or done columns repairs from the Repair List for segment **s**. In Multi-DUT Test Programs the value returned is for the first DUT in the Active DUTs Set (ADS).

Example

The following example sequentially returns all pending row repairs from the Repair List for segment S1 of the first DUT in the Active DUTs Set (ADS):

```
RaSegment S1 = ra_segment_make(); // Params defined elsewhere
for( int i = 0; i < ra_repaired_row_count_get( S1 ); ++i)
   RaSpareRow r = ra_repaired_row_get( S1, i );
```

The following example sequentially returns all done column repairs from the Repair List for segment S1 of the first DUT in the Active DUTs Set (ADS):

```
for( int i = 0; i < ra_repaired_col_count_get( S1, FALSE); ++i)
   RaSpareCol c = ra_repaired_col_get( S1, i, FALSE );
```

5.3.6.2 ra_repaired_row_get(), ra_repaired_col_get()

See RA Execution And Results, Repair List Functions, RA Software.

Note: these functions are not used in most redundancy applications. See Note:

Description

The `ra_repaired_row_get()` function is used to retrieve one row repair (`RaSpareRow`) from the Repair List for a specified segment. The `ra_repaired_col_get()` function is used to retrieve one column repair (`RaSpareCol`) from the Repair List for a specified segment.

Each returned spare row or column records the RaErrorPosition the spare is allocated to repair.

These functions can be used in conjunction with `ra_repaired_row_count_get()`, `ra_repaired_col_count_get()` to sequentially get all repairs from the Repair List. Or, all repairs can be retrieved using `ra_repaired_rows_get()`, `ra_repaired_cols_get()`.

The Repair List contains information in two states: *pending* repairs and *done* repairs. `ra_repaired_row_get()` and `ra_repaired_col_get()` have an optional argument (`pending`) used to select which repair is returned. See Repair List.

Note: these function names are very similar to `ra_repaired_rows_get()`, `ra_repaired_cols_get()`.

Usage

The following function returns the index'th row repair (`RaSpareRow`) from the pending Repair List or done Repair List for the specified segment:

```
RaSpareRow ra_repaired_row_get(
                            RaSegment s,
                            int index,
                            BOOL pending DEFAULT_VALUE( TRUE ));
```

The following function returns the index'th column repair (`RaSpareCol`) from the pending Repair List or done Repair List for the specified segment:

```
RaSpareCol ra_repaired_col_get(
                            RaSegment s,
                            int index,
                            BOOL pending DEFAULT_VALUE( TRUE ));
```

where:

s identifies the segment of interest.

index is the zero-based index into the Repair List and identifies which repair is to be returned.

pending is optional and, if used, specifies whether the repair from the pending Repair List is to be returned (TRUE, default) or the repair from the done Repair List is returned (FALSE).

ra_repaired_row_get() returns the index'th RaSpareRow from the Repair List for segment **s**. NULL is returned if **index** is out of range. In Multi-DUT Test Programs the value is returned for the first DUT in the Active DUTs Set (ADS).

ra_repaired_col_get() returns the index'th RaSpareCol from the Repair List for segment **s**. NULL is returned if **index** is out of range. In Multi-DUT Test Programs the value is returned for the first DUT in the Active DUTs Set (ADS).

Example

See Example.

5.3.6.3 ra_repaired_rows_get(), ra_repaired_cols_get()

See RA Execution And Results, Repair List Functions, RA Software.

Note: these functions are used in most redundancy applications. See Note:

Description

The ra_repaired_rows_get() function is used to retrieve all row repair(s) from the Repair List for a specified segment. The ra_repaired_cols_get() function is used to retrieve all column repair(s) from the Repair List for a specified segment.

Each returned spare row or column records the RaErrorPosition the spare is allocated to repair.

The Repair List contains information in two states: *pending* repairs and *done* repairs. ra_repaired_rows_get() and ra_repaired_cols_get() have an optional argument (pending) used to select which repairs are returned. See Repair List.

Note: these function names are very similar to ra_repaired_row_get(), ra_repaired_col_get().

Usage

```
RaSpareRowArray& ra_repaired_rows_get(
                    RaSegment s,
                    RaSpareRowArray& rows,
                    BOOL pending DEFAULT_VALUE( TRUE ) );
RaSpareColArray& ra_repaired_cols_get(
                    RaSegment s,
                    RaSpareColArray& cols,
                    BOOL pending DEFAULT_VALUE( TRUE ) );
```

where:

s identifies the segment of interest.

rows and **cols** are a user-defined RaSpareRowArray or RaSpareColArray, used to return the list of spare rows or columns used to repair segment **s**. These arrays are automatically resized by the system software as needed. Any prior contents are lost. The size and individual elements in the RaSpareRowArray or RaSpareColArray can be obtained using standard CArray member functions. In Multi-DUT Test Programs the repairs are returned for the first DUT in the Active DUTs Set (ADS).

pending is optional and, if used, specifies whether repairs from the pending Repair List are to be returned (TRUE, default) or the repairs from the done Repair List are to be returned (FALSE).

ra_repaired_rows_get() returns the RaSpareRowArray array passed in.

ra_repaired_cols_get() returns the RaSpareColArray array passed in.

Example

The following example gets all of the pending column repairs for segment S1 of the first DUT in the Active DUTs Set (ADS) into the cols array then sequentially gets one repair at a time from that array:

```
RaSegment S1 = ra_segment_make(); // Params defined elsewhere
RaSpareColArray cols = ra_repaired_cols_get( S1, cols );
for( int c = 0; c < cols.GetSize(); ++c){
```

```
    RaSpareCol spare_col = cols[c];
    // Do something with spare_col
}
```

5.3.6.4 ra_what_repaired_row_get(), ra_what_repaired_col_get()

See RA Execution And Results, Repair List Functions, RA Software.

Note: these functions are not used in most redundancy applications. See Note:

Description

The `ra_what_repaired_row_get()` function is used to access the Repair List to determine which spare row(s) were allocated by the RA to repair a specified bad row of a specified segment. The `ra_what_repaired_col_get()` function is used to access the Repair List to determine which spare column(s) were allocated by the RA to repair a specified bad column of a specified segment.

When the DUT has Per I/O Spares, a given bad row may have been repaired by multiple spares, with each repair correcting defects for a subset of data bits. For this reason, two versions of each function are provided:

- An RaErrorPosition is specified and the function returns a single spare which repairs it. The spare's RaErrorPosition is set to match the input.
- A bad row number only or bad column number only is specified and the function returns an array containing all spares used to repair that row or column. The RaErrorPosition `rcnum` value of the returned spare(s) is set to the bad row or column number and the RaErrorPosition `mask` value is set to the appropriate Per-I/O Spare Mask of the returned spare.

In Multi-DUT Test Programs, the operation returns information for the first DUT in the Active DUTs Set (ADS).

Usage

The following functions return a single spare row or column, from the Repair List, used to repair the specified row or column RaErrorPosition in the specified segment:

```
RaSpareRow ra_what_repaired_row_get(
            RaSegment s,
            RaErrorPosition row,
            BOOL pending DEFAULT_VALUE( TRUE ) );
RaSpareCol ra_what_repaired_col_get(
            RaSegment s,
            RaErrorPosition col,
            BOOL pending DEFAULT_VALUE( TRUE ) );
```

The following functions return an array containing all spare row(s) or column(s), from the Repair List, used to repair the specified row or column of the specified segment:

```
void ra_what_repaired_row_get(RaSegment s,
                              int rownum,
                              RaSpareRowArray& rows,
                             BOOL pending DEFAULT_VALUE( TRUE ) );
void ra_what_repaired_col_get(RaSegment s,
                              int colnum,
                              RaSpareColArray& cols,
                             BOOL pending DEFAULT_VALUE( TRUE ) );
```

where:

s identifies the segment of interest.

row and **col** identify the bad row or column RaErrorPosition to be considered. Both components are packaged in a user-defined `RaErrorPosition` variable.

rownum and **colnum** identify the bad row or column to be considered.

`rows` and `cols` are a user-defined array used to return a list of spare rows or spare columns used to repair **row** or **col**. The array is automatically resized by the system software as needed. Any prior contents are lost. The size and individual elements in the `RaSpareRowArray` or `RaSpareColArray` can be obtained using standard `CArray` member functions.

pending is optional and, if used, specifies whether the repair information is retrieved from the pending Repair List (TRUE, default) or the done Repair List (FALSE).

`ra_what_repaired_row_get()` returns the spare row used to repair the specified RaErrorPosition.

`ra_what_repaired_col_get()` returns the spare column used to repair the specified RaErrorPosition.

Both functions return NULL, and issue a warning, when:

- **s** is NULL.
- The RaErrorPosition `rcnum` value does not reside in segment **s**.
- The RaErrorPosition `mask` value is not a valid mask position for the spare being used for the repair.
- The `RaSpareRow` or `RaSpareCol` used to repair the specified RaErrorPosition is not found in the Repair List list.

Example

The following example uses the first form of `ra_what_repaired_row_get()`:

```
RaSegment S1 = ra_segment_make(); // Params defined elsewhere
RaErrorPosition bad_row;
bad_row.rcnum = 8;
bad_row.mask = 0xFF;
RaSpareRow r = ra_what_repaired_row_get( S1, bad_row );
```

The following example uses the second form of `ra_what_repaired_row_get()`:

```
RaSpareRowArray rows;
ra_what_repaired_row_get( S1, 8, rows );
```

5.3.6.5 ra_spare_repaired_errors_get()

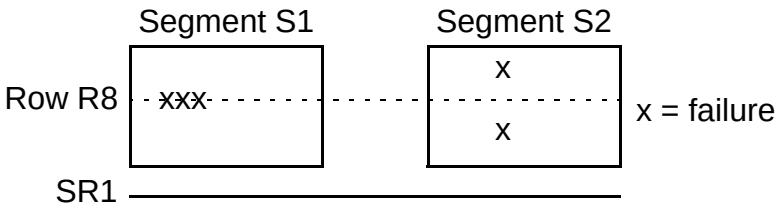
See RA Execution And Results, Repair List Functions, RA Software.

Note: this function is not used in most redundancy applications. See Note:

Description

The `ra_spare_repaired_errors_get()` function is used to determine if a specified segment has errors which would be repaired by a specified spare row or spare column. This is targeted at situations in which the specified spare row/column causes Linked Segments and the spare can be selectively applied to each segment to which it is linked.

For example:



In this example, spare row SR1 can be used to replace one row in both segments S1 and S2 (spare row SR1 thus creates Linked Segments). However, since the DUT's redundancy architecture allows this spare row to be selectively applied to each segment it is desirable to know which segment(s) have errors which would be repaired by the spare (normally, it is not desirable to replace a row/column which doesn't have errors). In this example, row R8 has errors in segment S1 but not in segment S2. Thus, it is desirable to replace row R8 only in segment S1. In this example, `ra_spare_repaired_errors_get()` will return TRUE for segment S1 and FALSE for segment S2.

Usage

```
BOOL ra_spare_repaired_errors_get( RaSegment s, RaSpareRow r);
BOOL ra_spare_repaired_errors_get( RaSegment s, RaSpareCol c);
```

where:

s identifies the target segment.

r and **c** identify the spare row or spare column. A warning is issued if **r** or **c** is not associated with segment **s** (see `ra_spare_add()`).

`ra_spare_repaired_errors_get()` returns TRUE if **r** or **c** would repair errors in segment **s**.

Example

```
if( ra_spare_repaired_errors_get( S1, SR1 ) )
   output(" Segment S1 has errors repairable by SR1");
```

5.3.7 Redundancy Call-back Functions

See Redundancy Analysis (RA), RA Software.

Note: these functions are not used in most redundancy applications. See Note:

Overview

Call-back functions are C-functions, written by the user or by Nextest, which are not directly called from user-written code.

Instead, a call-back function is *registered* with the system software and will be automatically executed by a Nextest-written function at the appropriate time. The RA call-back functions allow user code to customize the operation of certain built-in RA operations, by registering user-written functions that will be executed at certain well-defined times during the execution of Nextest-defined RA functions. In simple terms, this allows Nextest software to call user-written C code.

Several key redundancy operations are implemented as call-back functions. Normally, the built-in call-back functions, written by Nextest, are executed, but if necessary, to customize the redundancy software, a user-written function can be registered to be executed instead.

The following redundancy call-backs are available for RA customization:

- RaRowAvailableFunc & RaColAvailableFunc Call-back Function
- RaSparseFunc Call-back Function
- RaEvalFunc Call-back Function
- RaRepairFunc Call-back Function
- RaRowUseOK & RaColUseOK Call-back Functions
- RaMustRepairFunc Call-back Function
- RaScanRCFunc Call-back Function
- RaScanAreaCallbackFunc Call-back Function

These are individually documented below (follow the links). In addition, review the pseudo-code in `ra_execute()` to graphically view where these call-backs are invoked in the RA process.

5.3.7.1 RaRowAvailableFunc & RaColAvailableFunc Call-back Function

See Redundancy Analysis (RA), Redundancy Call-back Functions.

Note: these functions are not used in most redundancy applications. See Note:

Description

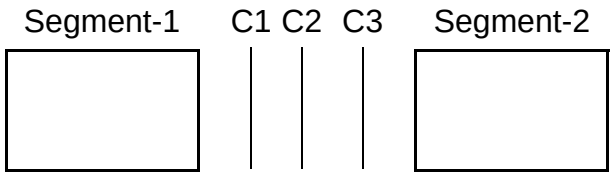
These call-back functions are optionally registered using the `ra_segment_make()` function. If registered, they are executed by `ra_execute()` during Must-repair (more below).

If used, these call-backs allow a user-written call-back function to return an array of spare rows and/or spare columns that are currently usable to repair the specified (bad) row or column in the segment being processed. When not used, the available spares are obtained from the Spares List. There are no default `RaRowAvailableFunc` or `RaColAvailableFunc` call-back functions.

These call-backs execute during Must-repair and provide a method for user-written code to implement spare usage constraints (rules) at a high level (the RaRowUseOK & RaColUseOK Call-back Functions are used at a per-row or per-column level).

The call-back must return an array containing the available (usable) spare rows or columns which can be used to repair the segment being processed.

For example, consider a device with 3 spare columns (C1, C2, C3 below) that are shared among 2 segments, but are constrained such that only 2 spare columns can be used to repair a given segment.



During Must-repair the default operation will allow all 3 columns in the Spares List to be used. To implement the rule noted above requires a user-written `RaColAvailableFunc` call-back function which would consider how many spare columns are currently available in the Spares List but would only return a maximum of 2.

Note: the RA system software executes these call-backs. When executed during Must-repair and Sparse-repair the `pos` argument is NULL. At other times (`ra_worst_row_get()`, `ra_best_row_wipeout()`, etc.), the `pos` argument identifies an RaErrorPosition being considered for repair. User written call-backs must anticipate both applications.

In Multi-DUT Test Programs a single RaRowAvailableFunc and a single RaColAvailableFunc call-back is registered at any given time; i.e. the Active DUTs Set (ADS) or Ignored DUTs Set (IDS) have no effect on which call-back is executed.

Usage

A user-written RaRowAvailableFunc call-back function must conform to the following prototype design:

```
RaSpareRowArray& (*RaRowAvailableFunc)(
                    RaSegment s,
                    RaErrorPosition* pos,
                    RaSpareRowArray& rows);
```

A user-written RaColAvailableFunc call-back function must conform to the following prototype design:

```
RaSpareColArray& (*RaColAvailableFunc)(
                    RaSegment s,
                    RaErrorPosition* pos,
                    RaSpareColArray& cols);
```

where:

s identifies the segment being processed when the call-back is invoked.

pos identifies the (bad) row RaErrorPosition or column RaErrorPosition to be considered. User-written call-backs must handle a NULL **pos** value, see Note:.

rows and **cols** are an array created by the RA software. The call-back code must clear the array, then add to it all of the spares that are available to repair **pos**, taking into account that pos may be NULL.

The RaRowAvailableFunc() call-back function must return the RaSpareRowArray passed as the **rows** argument.

The RaColAvailableFunc() call-back function must return the RaSpareColArray passed as the **cols** argument.

5.3.7.2 RaSparseFunc Call-back Function

See Redundancy Analysis (RA), Redundancy Call-back Functions.

Note: this function is not used in most redundancy applications. See Note:

Description

This call-back function is optionally passed as an argument to the `ra_execute()` function.

If used, a user-written Redundancy Call-back Functions replaces the built-in default function which performs Sparse-repair.

The call-back function determines the precedence and/or criteria for using spare elements for a given repair, and when a spare is allocated for a repair executes `ra_spare_use()` to move that spare from the Spares List to the Repair List.

Any time a Sparse-repair is identified and a spare allocated, the Sparse-repair call-back executes a Must-repair analysis again.

The built-in `RaSparseFunc` call-back functions are shown in the table below, with the default noted. The Examples below contain code which implements these call-backs, as examples for user-written versions:

Table 5.3.7.2-1 Built-in RaSparseFunc Functions

Option	Comment
`ra_row_first_sparse`	Uses a spare row if any are available, otherwise uses a spare column (default).
`ra_col_first_sparse`	Uses a spare column if any are available, otherwise uses a spare row.
`ra_row_pref_sparse`	Identify the row or column with the maximum number of failures, and use a spare row or column accordingly. Use a spare row if there is a tie.
`ra_col_pref_sparse`	Identify the row or column with the maximum number of failures, and use a spare row or column accordingly. Use a spare column if there is a tie.

Usage

A user-written `RaSparseFunc` call-back function must conform to the following prototype design:

```
BOOL (*RaSparseFunc)( RaSegment s );
```

where:

s identifies the segment to process when the call-back is invoked.

The RaSparseFunc call-back function must return FALSE if the segment is not repairable and TRUE if repairable.

Example

The following code shows the built-in RaSparseFunc call-back functions (last) and most (but not all) supporting functions (first). These are included here as examples for design of a user-written call-back function:

```
static RaSpareRow find_row_repair( RaSegment s, int &len ){
   RaErrorPosArray pos_array;
   // Get the list of failing rows in the segment s
   ra_failed_rows_get ( s, pos_array ); // ra_failed_rows_get()
   // Use ra_worst_row_get() to identify and return the best spare
   // row to fix it.
   return ra_worst_row_get(s, pos_array, &len);
}
static RaSpareCol find_col_repair( RaSegment s, int &len ){
   RaErrorPosArray pos_array;
   // Get the list of worst columns in the segment s
   ra_failed_cols_get(s, pos_array); // ra_failed_cols_get()
   // Use ra_worst_col_get() to identify and return the best spare
   // column to fix it.
   return ra_worst_col_get(s, pos_array, &len);
}
// This function is only called after a call to repair_col has
// failed. That failure means that a column repair wasn't possible
// (possibly due to an RaColUseOk function). This function will try
// to repair a row that has an error in the column that we just
// failed to repair.
static BOOL repair_row_in_col( RaSegment s ) {
   int len;
   // Find the worst column and the spare to fix it
   RaSpareCol spare_col = find_col_repair( s, len );
   if (! spare_col) return FALSE;
   // Find a row that fixes an error in spare_col
   RaSpareRow spare_row =
      find_row_in_col_repair(s, ra_spare_position_get(spare_col));
```

```
   if (! spare_row) return FALSE;
   // Use the spare just identified
   return ra_spare_use(s, spare_row); // ra_spare_use()
}

// This function is only called after a call to repair_row has
// failed. That failure means that a row repair wasn't possible
// (possibly due to an RaRowUseOk function). This function will try
// to repair a column that has an error in the row that we just
// failed to repair.
static BOOL repair_col_in_row( RaSegment s ) {
   int len;
   // Find the worst row and the spare to fix it
   RaSpareRow spare_row = find_row_repair( s, len );
   if (!spare_row) return FALSE;
   // Find a column that fixes an error in spare_row
   RaSpareCol spare_col =
      find_col_in_row_repair(s, ra_spare_position_get(spare_row));
   if (! spare_col) return FALSE;
   // Use the spare just identified
   return ra_spare_use(s, spare_col); // ra_spare_use()
}

static BOOL repair_row( RaSegment s ) {
   int len;
   // Find the worst row and the spare to fix it.
   RaSpareRow spare = find_row_repair( s, len );
   // Use the spare just identified
   return ra_spare_use( s, spare ); // ra_spare_use()
}

static BOOL repair_col( RaSegment s ) {
   int len;
   // Find the worst column and the spare to fix it.
   RaSpareCol spare = find_col_repair( s, len );
   // Use the spare just identified
   return ra_spare_use( s, spare ); // ra_spare_use()
}

DLL_PUBLIC BOOL ra_row_first_sparse( RaSegment s ) {
   // First, try to repair a row, if that fails, try to repair a
   // column with a failure in the failing row. If THAT fails, just
```

```
   // repair any column.
   return repair_row(s) || repair_col_in_row(s) || repair_col(s);
}
DLL_PUBLIC BOOL ra_col_first_sparse( RaSegment s ) {
   // First, try to repair a column, if that fails, try to repair
   //  a row with a failure in the failing column. If THAT fails,
   // just repair any row.
   return repair_col(s) || repair_row_in_col(s) || repair_row(s);
}
```

5.3.7.3 RaEvalFunc Call-back Function

See Redundancy Analysis (RA), Redundancy Call-back Functions.

Note: this function is not used in most redundancy applications. See Note:

Description

This call-back function is optionally passed as an argument to the `ra_execute()` function.

If used, this user-written Redundancy Call-back Functions replaces the built-in default call-back function (`ra_linear_eval()`) to indirectly determine which segment to process next during Sparse-repair.

The call-back function does this by returning a positive floating point value representing the *badness* of the specified segment, based on user-defined criteria. The worst such segment will be the one next selected for repair. Note that only segments with errors remaining will be passed to the `RaEvalFunc`.

The Example 2: below document the code implementing `ra_linear_eval()` as an example call-back. `ra_linear_eval()` returns a larger float value according to the following criteria:

- The segment has no spares left for repair; i.e. the segment is unrepairable and the RA should likely stop.
- The segment with highest ratio of failed rows (`R`) and columns (`C`) to spare rows (`r`) and columns (`c`); i.e. `R+C/r+c`.

Usage

A user-written RaEvalFunc call-back function must conform to the following prototype design:

```
float (*RaEvalFunc)( RaSegment s );
```

where:

s identifies the segment to evaluate when the call-back is invoked.

The RaEvalFunc call-back function must return a value as described above.

Examples

Example 1:

This example shows the simplest possible RaEvalFunc, which would perform Sparse-repair in the natural segment order:

```
float simplest_evalfunc( RaSegment s ) { return 1.0; }
```

Example 2:

The following code implements the built-in ra_linear_eval() function, and is included here as an example for design of a user-written function:

```
// This function is an example of an RaEvalFunc(). It rates the
// "badness" of a segment based upon the ratio of errors in the
// segment to the number of available spares.
DLL_PUBLIC float ra_linear_eval( RaSegment s ) {
   const float UNREPAIRABLE = 1000000.0;
   int r = ra_spare_row_count_get( s ); // ra_spare_row_count_get()
   int c = ra_spare_col_count_get( s ); // ra_spare_col_count_get()
   int R = ra_failed_rows_get( s ); // ra_failed_rows_get()
   int C = ra_failed_cols_get( s ); // ra_failed_cols_get()
   if ( R + C == 0 ) // No errors to repair
      return -1;
   if ( r + c == 0 ) // No spares left
      return UNREPAIRABLE;
   return float( R + C ) / float( r + c );
}
```

5.3.7.4 RaRepairFunc Call-back Function

See Redundancy Analysis (RA), Redundancy Call-back Functions.

Note: this function is not used in most redundancy applications. See Note:

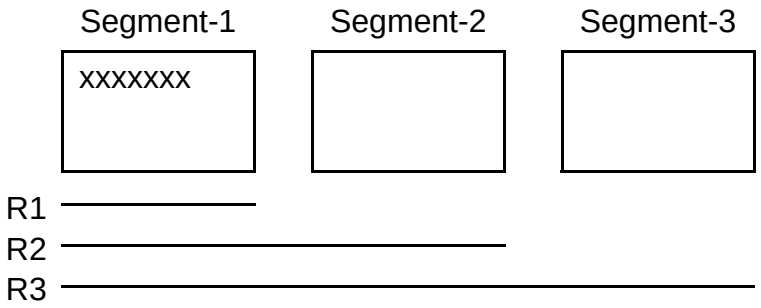
Description

This call-back function is optionally passed as an argument to `ra_execute()` and `ra_must_repair()`.

If used, during Must-repair these Redundancy Call-back Functions determine the strategy for deciding which spare element is selected to make a given repair and for moving that spare from the Spares List to the Repair List (using `ra_spare_use()`).

Different call-back functions may be registered for rows vs. columns. The built-in default `RaRepairFunc` call-back functions are: `ra_shortest_row_use` and `ra_shortest_col_use`. These are documented in the Examples below.

This diagram shows how the term *shortest* is applied:



In this example, a bad row (failures shown as **`xxxxxxxx)`**, in Segment-1 is a Must-repair row. Spare rows R1, R2, and R3 are available to make this repair. If no user-written row `RaRepairFunc` is registered the built-in `ra_shortest_row_use()` is used and spare row R1 will be chosen to repair the bad row.

Note: if a user-written call-back successful identifies a spare row or spare column to repair the specified error it must execute `ra_spare_use()` to move that spare from the Spares List to the Repair List.

Usage

A user-written RaRepairFunc call-back function must conform to the following prototype design:

```
BOOL (*RaRepairFunc)( RaSegment s, RaErrorPosition &pos );
```

where:

s identifies the segment to process when the call-back is invoked.

pos specifies the bad row RaErrorPosition or column RaErrorPosition to be repaired.

The RaRepairFunc call-back function must return TRUE when the function succeeds in allocating a spare to repair the specified error; otherwise FALSE must be returned.

Examples

The following examples implement the built-in ra_shortest_row_use() call-back function and 2 supporting functions. This is included here as an example for design of a user-written function:

```
// This function tries to repair "pos" with the shortest spare row
// available
static BOOL shortest_row_use( RaSegment s, RaErrorPosition &pos ) {
   RaSpareRow r = ra_shortest_spare_row_get( s );
   return r ?
      ra_spare_use( s, ra_spare_position_set( r, pos ))
      :
      FALSE;
}
// This function tries to repair "pos" with ANY spare row
// available. Used when shortest_row_use() fails
static BOOL any_row_use( RaSegment s, RaSpareRow &pos ) {
   int num_rows = ra_spare_row_count_get( s );
   for (int i = 0; i < num_rows; i++) {
      RaSpareRow r;
      r = ra_spare_row_get( s, i );    // ra_spare_row_get()
      r = ra_spare_position_set(r, pos); //ra_spare_position_set()
      if( ra_spare_use( s, r ) )       // ra_spare_use()
         return TRUE;
   }
   return FALSE;
}
```

```
// This function uses the "shortest" row to repair the error
// position "pos". RaRepairFunc call-back functions must only
// return FALSE if pos cannot be fixed by ANY spare row, thus we
// first try to use the shortest spare row. If that fails, we will
// try to repair it with other spare rows.
DLL_PUBLIC BOOL ra_shortest_row_use(  RaSegment s,
                                      RaSpareRow &pos ) {
   return shortest_row_use(s, pos) || any_row_use(s, pos);
}
```

5.3.7.5 RaRowUseOK & RaColUseOK Call-back Functions

See Redundancy Analysis (RA), Redundancy Call-back Functions.

Note: this function is not used in most redundancy applications. See Note:

Description

These call-backs are optionally passed as an argument to the `ra_spare_row_make()`, `ra_spare_col_make()` functions.

If used, these user-written Redundancy Call-back Functions execute as the RA is about to allocate a given spare to repair a specific bad row or column (i.e. move the spare from the Spares List to the Repair List). This allows user code to decide if the selected spare row or column should actually be used, typically to enforce DUT-specific repair constraints. A given spare row or column will not be used (not moved to the Repair List) unless the `RaRowUseOK` or `RaColUseOK` call-back returns TRUE.

The call-back is executed any time `ra_spare_use()` is called, which is typically done via `ra_execute()`. More specifically, `ra_spare_use()` is executed in all of the built-in `RaSparseFunc` and `RaRepairFunc` call-back functions (see RaSparseFunc Call-back Function and RaRepairFunc Call-back Function).

The RaErrorPosition of the spare row or spare column passed in to the call-back must already have been set.

A user call-back must also consider whether the specified spare has Rows-Used-Together(RUT), Columns-Used-Together(CUT) and adapt accordingly.

No default call-back is registered however three built-in call-backs function are available as examples (see below):

- `ra_exclusive`
- `ra_single_spare_row_per_address`
- `ra_single_spare_col_per_address`

Note that the RaRowAvailableFunc & RaColAvailableFunc Call-back Functions should be used to affect when a spare is *considered* for a repair.

Usage

A user-written `RaRowUseOK` call-back function must conform to the following prototype design:

```
BOOL (*RaRowUseOK)( RaSegment s, RaSpareRow row );
```

A user-written `RaColUseOK` call-back function must conform to the following prototype design:

```
BOOL (*RaColUseOK)( RaSegment s, RaSpareCol col );
```

where:

`s` identifies the segment being processed when the call-back is invoked.

`row` and **`col`** are the spare row or spare column being considered for use to make a repair. The spare's `RaErrorPosition` values will be set to the proposed repair before the call-back is executed.

The `RaRowUseOK` and `RaColUseOK` call-back function must return `TRUE` when the proposed **`row`** or **`col`** is OK to use to repair the segment, otherwise `FALSE` must be returned.

Example

Example 1:

The following example code implements the built-in `ra_single_spare_col_per_address()` function. It is included here as an example of the design of a user-written call-back function:

```
// This function is useful in devices that allow only a single spare
// column to be used to repair a given bad column. This implies
// Per I/O Spares. It returns FALSE if a spare column has already
// been allocated to fix the specified column.
```

```
BOOL ra_single_spare_col_per_address( RaSegment s, RaSpareCol c ){
   int colnum = ra_spare_colnum_get( c );) // ra_spare_colnum_get()
   RaSpareColArray cols;
   ra_what_repaired_col_get(s, colnum, cols);
      // ra_what_repaired_col_get()
   return ( cols.GetSize() > 0) ? FALSE : TRUE;
}
```

5.3.7.6 RaMustRepairFunc Call-back Function

See Redundancy Analysis (RA), Redundancy Call-back Functions.

Note: this function is not used in most redundancy applications. See Note:

Description

This call-back is executed by `ra_execute()`, and `ra_scan_area_callback()` to perform the Must-repair analysis. Arguments to these functions allow a user-written call-back function to be executed instead of the built-in default function (`ra_must_repair()`).

The `ra_row_func` and `ra_col_func` arguments allow an `RaRepairFunc` call-back to be registered for both spare row selection and spare column selection.

Note: a user-written `RaRepairFunc` must execute `ra_spare_use()` when a spare row or column is allocated for a repair, to move the spare from the Spares List to the Repair List.

Usage

A user-written `RaMustRepairFunc` call-back function must conform to the following prototype design:

```
BOOL (*RaMustRepairFunc)(RaSegment s,
                         RaSpareRowArray& spare_rows_avail,
                         RaSpareColArray& spare_cols_avail,
   RaRepairFunc ra_row_func DEFAULT_VALUE( ra_shortest_row_use ),
   RaRepairFunc ra_col_func DEFAULT_VALUE( ra_shortest_col_use ));
```

where:

s identifies the segment being processed when the call-back is invoked.

spare_rows_avail and **spare_cols_avail** are arrays containing the spare rows and columns that the Must-repair analysis can consider usable for repair during the analysis.

ra_row_func is optional and, if used, specifies a user-defined RaRepairFunc call-back function which will determine the strategy for deciding which spare row is selected to make a given repair. See Description and Redundancy Call-back Functions.

ra_col_func is optional and, if used, specifies a user-defined RaRepairFunc call-back function which will determine the strategy for deciding which spare column is selected to make a given repair. See Description and Redundancy Call-back Functions.

ra_must_repair() must return FALSE if the segment is determined to be unrepairable, otherwise TRUE must be returned.

Example

This call-back is too complex to provide an example.

5.3.7.7 RaScanRCFunc Call-back Function

See Redundancy Analysis (RA), Redundancy Call-back Functions.

Note: this function is not used in most redundancy applications. See Note:

Description

This call-back is optionally registered by ra_config_set() or using ra_scan_rc_func_set().

If registered, this call-back function will be used instead of the built-in method to scan (read) the Row RAM or Column RAM during execution of ra_execute(). This allows the user to design a custom scan function for reading these RAMs during RA.

This call-back is not executed when the ECR Mini-RAM is enabled (see ra_config_set()); i.e. the ECR's Row RAM or Column RAM are not used when the ECR Mini-RAM is used.

In Multi-DUT Test Programs a single RaScanRCFunc call-back is registered at any given time; i.e. the Active DUTs Set (ADS) or Ignored DUTs Set (IDS) have no effect on which call-back is executed.

The currently registered call-back can be retrieved using `ra_scan_rc_func_get()`.

Usage

A user-written RaScanRCFunc call-back function must conform to the following prototype design:

```
int (*RaScanRCFunc)(RaSegment s,
                    BOOL by_row,
                    int min,
                    int max,
                    __int64 mask,
                    PointFailureArray &failures );
```

where:

s identifies the segment being processed when the call-back is invoked.

by_row specifies whether the ECR's Row RAM (TRUE) or Column RAM (FALSE) is being scanned.

min and **max** specify the starting and ending addresses to be read.

mask identifies which data bits are scanned. A logic-1 in a given bit position enables the corresponding data bit to be scanned.

failures is a user-defined `PointFailureArray` used to return any failures read. The system software will resize the array as necessary. Any prior array contents are over-written.

The RaScanRCFunc call-back must return the number of elements in the **failures** array.

Example

???

5.3.7.8 RaScanAreaCallbackFunc Call-back Function

See Redundancy Analysis (RA), Redundancy Call-back Functions.

Note: this function is not used in most redundancy applications. See Note:

Note: early Magnum RA software used the RaScanAreaFunc call-back function. This was replaced by the function documented here. Any use of the RaScanAreaFunc will be flagged with a runtime warning and the desired RA operation will not occur.

Description

This call-back is optionally registered by `ra_config_set()` and `ra_scan_area_callback_func_set()`.

If registered, the user's call-back function will be executed during `ra_execute()` instead of the built-in `ra_scan_area_callback()` function. Note the following:

- The call-back function receives errors scanned from the ECR by the system software. User code may modify (filter) the errors received before they are added to the Error List (more below).
- The user's call-back may be executed more than once for a given segment. Up to 16K errors are received each time the call-back is executed by the system software. The `DWORD row/col` parameters of each `PointFailure` element can be used to determine which segment is being scanned.
- The `__int64 data` parameter of each `PointFailure` element includes errors for any/all DUT(s) which have errors logged to a given ECR. In Multi-DUT Test Programs, any DUT(s) in the Active DUTs Set (ADS) at the time a functional test logged errors to the ECR may have errors in a given element. If necessary, user code must shift/mask the appropriate bits for a given DUT.
- The user's call-back MUST return using the following syntax (exactly as shown). This causes the (potentially filtered) errors in the `fails` array to be properly added to the Error List:

  ```
  return ra_scan_area_callback( fails, count, scanmask );)
  ```

  where:

  `fails` is the `PointFailure` array containing the errors.

  `count` specifies the number of elements in **`fails`**.

  `scanmask` is the unmodified `scanmask` parameter passed into the user's call-back.

Note: proper RA operation *REQUIRES* that the user's call-back function return by calling `ra_scan_area_callback()` as noted above.

The currently registered call-back can be retrieved using `ra_scan_area_callback_func_get()`.

In Multi-DUT Test Programs a single `RaScanAreaCallbackFunc` call-back is registered at any given time; i.e. the Active DUTs Set (ADS) or Ignored DUTs Set (IDS) have no effect on which call-back is executed.

Usage

A user-written `RaScanAreaCallbackFunc` call-back function must conform to the following prototype design:

```
BOOL ra_scan_area_callback( PointFailure* fails,
                            int count,
                            __int64& scanmask);
```

where:

`fails` is a pointer to an array of `PointFailure`(s) containing **`count`** elements.

`scanmask` is the addess of an `__int64` variable which is not intended for user applications but must be passed to `ra_scan_area_callback()` when executed from the user's call-back (which is required as noted in the Description).

`ra_scan_area_callback()` must return TRUE if un-scanned errors remain in the ECR for the current segment.

Example

None

5.4 Magnum RA vs. Maverick-I/-II RA

See Redundancy Analysis (RA), RA Software.

This section outlines some key differences between Maverick RA and Magnum RA. This should only be interesting to users who have used Maverick RA. Note that the topic of Maverick-to-Magnum RA migration is too complex to provide a step-by-step tutorial; the user

must comprehend, in detail, both their device's redundancy architecture and repair options and the Maverick RA solution to comprehend the information in this section.

- The Maverick ECR does not have an ECR Mini-RAM. Thus, the ECR scan performance using Maverick does not benefit from this hardware and support for Spare Segments is totally implemented in user code and methods.
- Maverick RA does not include integrated parallel test support. Using Maverick, user code is required to create an individual DIE object for each DUT and explicitly invoke the RA process for each DIE.
- Maverick RA does not include Per I/O Spares support; i.e. Per I/O Spares on Maverick is totally implemented by the user.
- Maverick RA does not include direct support for Rows-Used-Together(RUT), Columns-Used-Together(CUT). Instead, ECR address compression is used, which works well for RA applications but degrades the resolution when displaying errors in BitmapTool.
- The Magnum RA software does not include or need the DIE or `error_engine` data types seen in Maverick RA software. In Magnum, a DUT is mostly equivalent to a Maverick DIE and all Maverick `error_engine` operations are integrated into the Magnum segment object model.
- Using Magnum in Multi-DUT Test Programs the Active DUTs Set (ADS) affects RA operations. In general, setter functions affect all DUT(s) in the ADS, whereas getter functions retrieve information for the first DUT in the ADS.
- In Magnum in Multi-DUT Test Programs, the RA performed by ra_execute() operates on all DUTs in the ADS. More importantly, the operations which scan (read) errors from the ECR are optimized to read a given ECR address only once, with any errors read cached as appropriate for each active DUT.

At the end of this section, a table of Magnum vs. Maverick RA Functions is provided. This provides only the most basic function cross-reference information. As noted above, the user must comprehend, in detail, both their device's redundancy architecture and repair options and the Maverick RA solution to comprehend the information in this section.

5.4.0.1 Magnum vs. Maverick RA Functions

See Magnum RA vs. Maverick-I/-II RA.

This section provides a very limited correlation of Maverick RA functions to Magnum RA functions. See Magnum RA vs. Maverick-I/-II RA.

As stated at the start of this chapter:

- DO NOT mix the Maverick RA functions with the Magnum RA functions: they DO NOT inter-operate.
- For all new Magnum test programs, it is highly recommended that the new RA functions be used.

Maverick ECR Functions

The following table lists selected Maverick functions associated with the ECR. These are now part of the RA software and become DUT-centric, not ECR-centric:

Maverick Function	Magnum Function	Notes
error_count()	ra_error_count_get()	
set_scan_area()	ra_scan_area_callback_func_set()	
set_scan_x()	ra_scan_rc_func_set()	
get_scan_area()	ra_scan_area_callback_func_get()	
get_scan_x()	ra_scan_rc_func_get()	
scan_area()	ra_scan_area_callback()	
error_count()	ra_error_count_get()()	

Maverick DIE Functions

The following table lists the Maverick RA DIE functions with the Magnum function which performs an equivalent or related operation. Using Magnum, a DUT is equivalent to a Maverick DIE, thus the DIE object is gone. Magnum DUT(s) are automatically created/ defined in the Pin Assignment Table. Since the DIE is gone, some Maverick functions below do not have exact equivalents in Magnum RA:

Maverick Function	Magnum Function	Notes
reset()	ra_reset()	
repair()	ra_execute()	
no_repair()	ra_result_get()	

Maverick Function	Magnum Function	Notes
repair_done()	ra_repair_done()	
segment_head()	ra_segment_count_get()	
segment_next()	ra_segment_get()	
segment_count()	ra_segment_count_get()	
dump()	ra_dump()	
add()	None	No DIE equivalent. Spares are added to segments using ra_spare_add().
make_die()	Obsolete	Related parameters are set using ra_config_set()
die_ecr()	Obsolete	
id()	Obsolete	
lookup_die()	Obsolete	

Maverick Segment Functions

The following table lists the Maverick RA Segment functions with the Magnum function which performs an equivalent or related operation. Using Magnum, the Maverick error_engine is merged into the segment object. Since the error_engine is gone, some Maverick functions below do not have exact equivalents in Magnum RA:

Maverick Function	Magnum Function	Notes
make_segment()	ra_segment_make()	
add()	ra_spare_add()	
reset()	ra_segment_reset()	
scan()	ra_scan_area_callback()	
must_repair()	ra_must_repair()	
no_repair()	ra_result_get()	
repair_done()	ra_segment_repair_done()	

Maverick Function	Magnum Function	Notes
unusable()	ra_unusable_set()	
usable()	ra_usable_set()	
error_count()	ra_error_count_get()	
linkage_count()	ra_segment_linkage_count_get()	
failed_rows_count() failed_cols_count()	ra_failed_rows_count_get() ra_failed_cols_count_get()	
spare_row_head() spare_col_head()	ra_spare_row_count_get() ra_spare_col_count_get()	Used with ra_spare_row_get()/ ra_spare_col_get() to iterate over all spare rows/columns.
spare_row_next() spare_col_next()	ra_spare_row_get() ra_spare_col_get()	
spare_row_count() spare_col_count()	ra_spare_row_count_get() ra_spare_col_count_get()	
repaired_row_head() repaired_col_head()	ra_repaired_row_count_get() ra_repaired_col_count_get()	Used with ra_repaired_row_get()/ ra_repaired_col_get() to iterate over all spare rows/ columns.
repaired_row_next() repaired_col_next()	ra_repaired_row_get() ra_repaired_col_get()	
what_repaired_row() what_repaired_col()	ra_what_repaired_row_get() ra_what_repaired_col_get()	
repaired_row_count() repaired_col_count()	ra_repaired_row_count_get() ra_repaired_col_count_get()	

Maverick Function	Magnum Function	Notes
data_mask() row_min() row_max() col_min() col_max()	ra_segment_config_get()	
shortest_spare_row() shortest_spare_col()	ra_shortest_spare_row_get() ra_shortest_spare_col_get()	
best_row_wipeout() best_col_wipeout()	ra_best_row_wipeout() ra_best_col_wipeout()	
worst_row() worst_col()	ra_worst_row_get() ra_worst_col_get()	
dump()	ra_segment_dump()	
id()	ra_segment_id_get()	
lookup_segment()	ra_segment_lookup()	
segment_die()	Obsolete	
segment_engine()	Obsolete	

Maverick Spare Functions

The following table lists the Maverick RA Spare functions with the Magnum function which performs an equivalent or related operation. Using Magnum, the Maverick `error_engine` is merged into the segment object. Since the `error_engine` is gone, some Maverick functions below do not have exact equivalents in Magnum RA:

Maverick Function	Magnum Function	Notes
make_spare_row() make_spare_col()	ra_spare_row_make() ra_spare_col_make()	
use()	ra_spare_use()	
usable()	ra_usable_set()	
unusable()	ra_unusable_set()	
wipeout()	ra_wipeout_get()	

Maverick Function	Magnum Function	Notes
dump()	ra_spare_dump()	
id()	ra_spare_id_get()	
lookup_spare_row() lookup_spare_col()	ra_spare_row_lookup() ra_spare_col_lookup()	

Maverick error_engine Functions

The following table lists the Maverick RA `error_engine` functions with the Magnum function which performs an equivalent or related operation. Using Magnum, the Maverick `error_engine` is merged into the segment object. Since the `error_engine` is gone, some Maverick functions below do not have exact equivalents in Magnum RA:

Maverick Function	Magnum Function	Notes
reset()	ra_segment_reset()	
worst_rows() worst_cols()	ra_worst_rows_get() ra_worst_cols_get()	
failed_rows() failed_cols()	ra_failed_rows_get() ra_failed_cols_get()	
row_wipeout() col_wipeout()	ra_row_wipeout() ra_col_wipeout()	
best_row_wipeout() best_col_wipeout()	ra_best_row_wipeout() ra_best_row_wipeout()	
repair_row() repair_col()	Obsolete	
make_error_engine()	Obsolete	
add()	Obsolete	
total()	Obsolete	
must_repair()	Obsolete	

Maverick Function	Magnum Function	Notes
dump()	Obsolete	
id()	Obsolete	
lookup_error_engine()	Obsolete	

Chapter 6 Interactive Tools

6.1 *UI* - User Interface

This section covers the following topics:

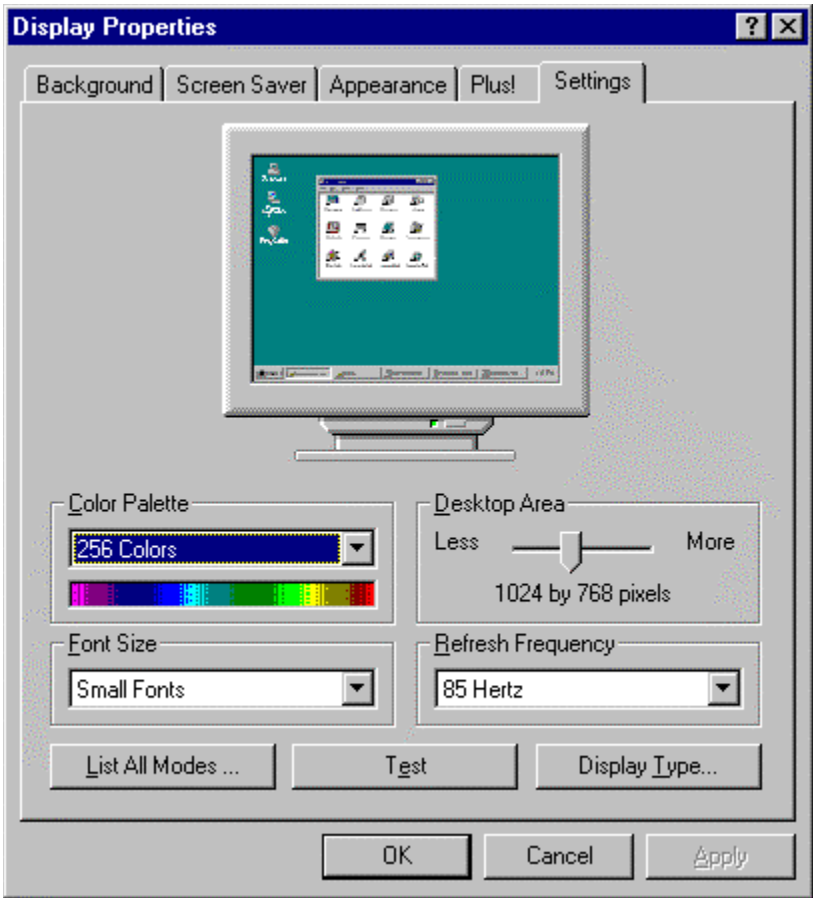
6.1.1 *UI* Overview

The *User Interface* (UI) is a graphical interface used to load, execute, and interact with test programs.

6.1.2 Before Starting *UI*

There are several system level configurations and processes which must be set up correctly before starting *Ui*.

1. *Ui* requires the video display resolution setting to be at least 1024x768 for some of the engineering tools to work properly. This is set up from Display Properties dialog, invoked using the right-mouse button from the display background:



2. *Ui* requires the TCP/IP protocol to be installed during the Windows NT network installation.

3. Each test site controller must be executing the program *MonitorApp*. This is normally started automatically when the user logs-in. Otherwise, you may see the following error messages:

If either of the above error messages are displayed, confirm that *MonitorApp* is running on each Site controller. On a personal tester (PT), this is done by pressing ***CTRL+ALT+DEL*** key combination to invoke the Task Manager, and confirming that `MonitorApp.exe` is seen in the process list. If *MonitorApp* is not running, it must be restarted; see Terminating & Restarting MonitorApp.

6.1.2.1 ui.ini File

The `ui.ini` file stores various configuration states each time UI is *terminated normally*. These states are used to configure UI the next time it is started: display size, location, windows displayed, Controller List entries, etc. Controller List entries are manually configured when using a multi-site systems (GT, VT, ST). Note the following:

- Any time UI is terminated, the `ui.ini` file is either created or updated if it already exists.
- Deleting `ui.ini` file will reset the initial UI configuration to default values. This must be done while UI is not running.

Executing UseRel will prompt the user with the following:

```
Copy previous_release\ui.ini to new_release\ui.ini? [n]
```

Entering **`Y`** (yes) will copy the existing `ui.ini` file, thus preserving the previous configuration. Entering **`N`** or <**`Enter`**> will not copy the `ui.ini` file and operation will be the same as prior releases.

This choice is not presented in the following situations:

- If the `ui.ini` file is already present in the software release being switched-to i.e. the option is only presented the first time a given release is first used.
- There is no previously installed release from which to copy the `ui.ini` file.

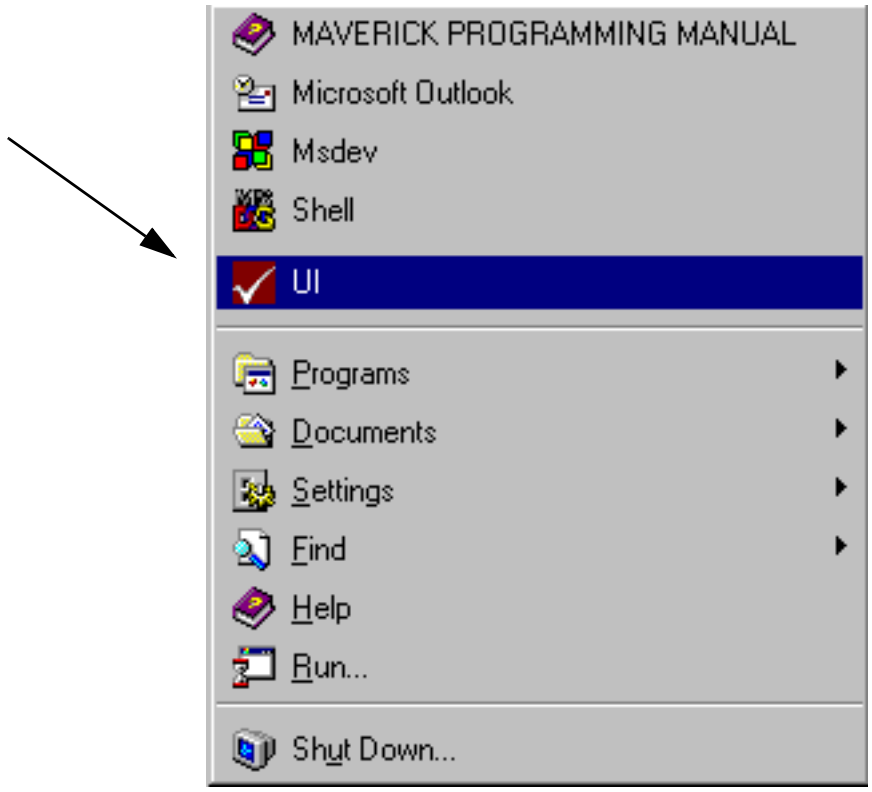
- The new release is same as the previous release.

6.1.3 Starting UI from Windows

UI can be started using several methods:

- Starting UI from a Command Line
- Clicking on the *Ui* icon in the Window's *Start Menu*.
- Beginning in software release h1.1.23, executing the *StartUI.exe* program located in the software release's *Utils*\ folder. This method only allows one instance of UI to be started; it otherwise operates the same as UI, and is not further documented.

To start UI from *Windows* select the UI icon in the *Start Menu*, as show below



Note: it is ***HIGHLY*** recommended that shortcuts to UI ***NOT*** be used, on the desktop or elsewhere. When installing Nextest software or switching between software releases shortcuts are ***NOT*** changed... but, the Start menu is modified to execute the correct version of UI.

6.1.4 Starting UI from a Command Line

UI can be started using two methods:

- Starting UI from Windows : common method
- Starting UI from a command line : for special applications only

UI can be started from a *Windows* command line by typing `UI<ret>`.

This should always work correctly since the **`path`** to UI is automatically set up when the Nextest software is installed.

To support test floor automation, a variety of input parameters can be passed to UI when it is started from a command line. In simple terms, this allows a command line or for more complex situations, a batch file, to start UI, load a test program, set user variable values, start testing, unload the test program, and terminate UI. When testing is complete, control will automatically return to the batch file. See UI User Variables

Usage

To start UI from a line command type:

```
UI<ret>
```

This starts UI the same as if it was started from the *Windows Start* menu.

6.1.5 Magnum 1/2/2x Simulation Setup

Note: the following information applies when using software releases h1.1.23 or later.

The following procedures may be used to setup the local computer in preparation for executing a Magnum 1/2/2x test program in simulation mode i.e. without using actual test

system hardware. See Simulation Configuration Errors for problem symptoms seen when this procedure is not done correctly.

Three methods are documented:

- MagnumSimulation.bat Method: execute this batch file to interactively to set the number of sites to be used and the Magnum system type to be simulated. The batch file then deals with all the setup details (which are outlined in the Hostmon Method, below).
- SIMULATED_SITES Method: pre-set the number of sites and Magnum system type using environment variables. Recommended for use on computers which will never be connected to actual hardware AND which always simulate the same number of sites and same Magnum system type.
- Hostmon Method: this method is for those who prefer to manually perform the steps automated using the MagnumSimulation.bat Method.

MagnumSimulation.bat Method

The *MagnumSimulation.bat* file may be executed to interactively select the Magnum system type and the number of sites to be simulated. The batch file then deals with the other details as outlined in the Hostmon Method. Note the following:

Note: as newer Magnum system types were developed the *MagnumSimulation.bat* file was modified to prompt the user to select which system type is to be simulated: 1, 2 or 3, corresponding to Magnum 1, Magnum 2 and Magnum 2x.

- This procedure must be performed even when simulating a single site.
- As more sites are enabled computer performance may be noticeably impacted.
- *MagnumSimulation.bat* is located in the Nextest software release's *Utils*\ directory. It may be convenient to create a desktop shortcut to this file.
- *MagnumSimulation.bat* executes UseRel from the Nextest software release which was last used. The release information is read from the Windows registry. If this release is not located as specified in the registry *MagnumSimulation.bat* will fail.
- The Windows shell opened by *MagnumSimulation.bat* is terminated automatically. A separate MonitorApp dialog, normally presended by UseRel, will be presented for each site being simulated.
- *MagnumSimulation.bat* is not backwards compatible with software releases prior to h1.1.23. UseRel was modified to support *MagnumSimulation.bat*.

SIMULATED_SITES Method

This method preset the number of sites and Magnum system type using environment variables. Recommended for use on computers which will never be connected to actual hardware AND which always simulate the same number of sites and same Magnum system type.

Before proceeding, note the following:

- This procedure should *not* be used on computers which are connected to Magnum test system hardware.
- If used, this procedure must be performed even when simulating a single site.

Setup procedure:

- This method requires that 2 environment variables be configured before executing UseRel. See Environmental Variables.
  - Set the SIMULATED_SITES environment variable to the number of sites to be simulated. Legal values are 1 to 10. As more sites are enabled computer performance may be noticeably impacted.
  - Set the SIMULATED_HD environment variable to the Magnum system type being simulated:

Magnum System Type	SIMULATED_HD value
Magnum 1	1
Magnum 2	2
Magnum 2x	3

- Execute UseRel from the target Nextest software release.

Hostmon Method

Before proceeding, note the following:

- Using this method the SIMULATED_SITES environment variable should ***not*** be set as defined in SIMULATED_SITES Method above.
- This procedure will operate correctly on computers which are connected to Magnum test system hardware. However, to revert to using the hardware requires additional steps, as outlined in Reconfiguring for Hardware Use.

- If used, this procedure must be performed even when simulating a single site.
- As more sites are enabled computer performance may be noticeably impacted.
- The steps below must be performed in the order noted.
- When executing the test program in Site Debug Mode (in Developer Studio) step-1. through step-6. below must be performed before starting Developer Studio.

Setup procedure:

1. The environment variable `SIMULATED_HD` must NOT be set to 0. More below.
2. Execute UseRel*.bat* from the Nextest Magnum software release to be used. Note that this is only required if UseRel was not already executed during the current logged-in session.
3. Start a Windows command shell.
4. Change directory to the location of the Nextest software release being used, typically something like: *C:\Nextest\v2.10.1*, etc.
5. Change directory to the *Bin\Misc* folder. The file of interest is named *hostmon.exe*.
6. Type the following command:

   ```
   hostmon /e:SIMULATED_HD=t /n
   ```

   where:

   t represents the Magnum system type (1 = Magnum 1, 2 = Magnum 2, 3 = Magnum 2x).

   n represents the number of sites to be used/enabled during the simulation.

   For example: to simulate Magnum 1 with 1 site:

   ```
   hostmon /e:SIMULATED_HD=1 /1
   ```

   To simulate Magnum 2 with 3 sites:

   ```
   hostmon /e:SIMULATED_HD=2 /3
   ```

   To simulate Magnum 2x with 2 sites:

   ```
   hostmon /e:SIMULATED_HD=3 /2
   ```

   Executing *hostmon* as shown above does the following:

   - Executes *hostmon /halt*. This terminates all instances of *hostmon*, *sitemon* and *monitorapp* (more below).

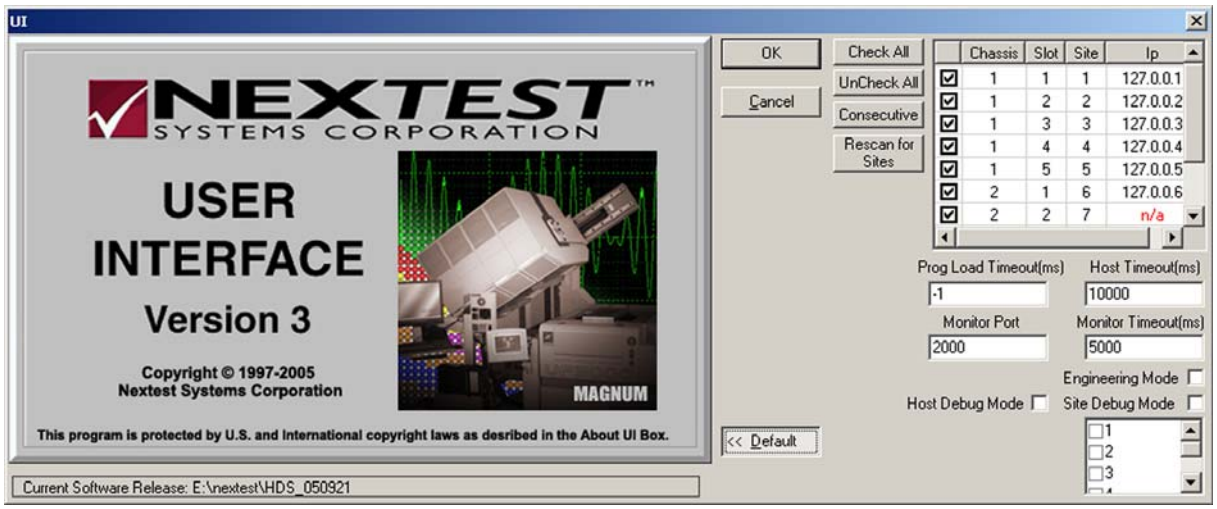
- If SIMULATED_HD is not currently set to 0, the following environment variables are set/unset as shown. This is done in *hostmon*'s environment (i.e. the system environment variable settings are not modified). UI will inherit this environment from *hostmon*:
  - Set SIMULATED_HD=t
  - Set SIMULATED_SITES=n
  - Unset SIMULATED_PTI
  - Unset SIMULATED_APG
  - Unset SIMULATED_PE

  Note that if SIMULATED_HD is set = 0 prior to executing *hostmon* the Nextest software presumes that a Maverick simulation is desired and the environment variables noted above are not modified.
- Starts one instance of the *hostmon* process, which will see the environment variables as setup in the previous step. Only one *hostmon* process is ever needed.
- Starts **n** instances of the *monitorapp* process. One is needed for each site to be simulated.
- Starts **n** instances of the *sitemon* process. One is needed for each site to be simulated.

The following dialog will appear after 1-2 seconds. A separate dialog is presented for each site being simulation. These dialogs may be displayed behind other windows:

Either click **Hide Window** or ignore this dialog, it will disappear after 10 seconds.

7. Start UI and confirm the upper right display area contains an IP Address (127.0.0.1, etc.) for each site to be used/enabled during the simulation. In the example below, 6 sites are enabled:



Note: it is OK to modify this configuration, to use less sites. To use more sites requires repeating the procedure above, starting with step-6.

Note: if the *hostmon* process is not running when UI is started, the Maverick-I/-II version of UI will be started and all of the other graphic tools will be configured for Maverick-I/-II use. The various graphic tools have different features when using Magnum 1/2/2x vs. Maverick-I/-II.

Note: *hostmon /halt* may be used to terminate all *hostmon*, *sitemon* and *monitorapp* processes executing on the current computer.

Reconfiguring for Hardware Use

When a computer which is connected to a Magnum 1/2/2x test system is configured for simulation the following steps are required to revert to using the test system hardware:

1. Start a Windows command shell.
2. Change directory to the location of the Nextest software release being used, typically something like: *C:\Nextest\v2.10.1*, etc.

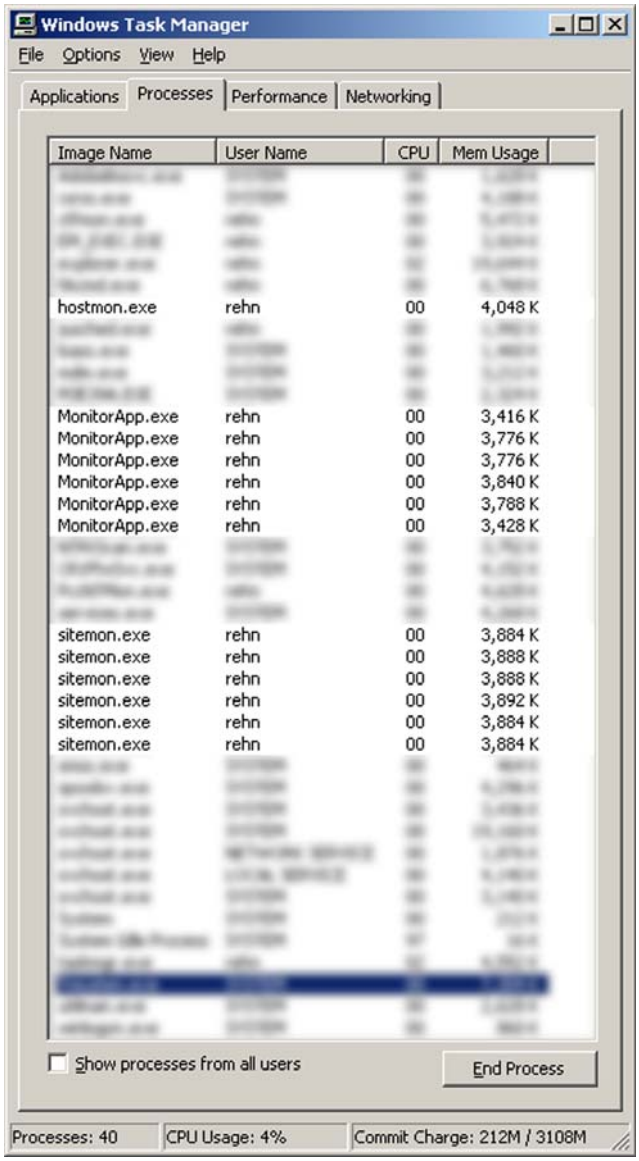
3. Change directory to the *Bin\Misc* folder. The file of interest is named *hostmon.exe*.
4. Type the following command:

   ```
   hostmon /halt
   ```

   This command terminates all instances of *hostmon*, *sitemon* and *monitorapp* on the local computer.
5. Execute UseRel from the target Nextest software release.
6. Restart (reboot) the Site Assembly computers. This can normally be done by executing *StartServer* from the *RBOOT* folder found on the desktop. However, if this fails to restart all sites correctly it may be necessary to cycle the power to each system chassis.

Simulation Configuration Errors

For reference, using the Windows Task Manager, the following processes are important to Magnum simulation. This example shows 6 sites being simulated:



Note: if the *hostmon* process is not running when UI is started, the Maverick-I/-II version of UI will be started and all of the other graphic tools will be configured for Maverick-I/-II use. The various graphic tools have different features when using Magnum 1/2/2x vs. Maverick-I/-II.

If the steps noted above are not performed prior to starting UI some or all of the following error symptoms may be seen.

- In the following image note that no valid IP addresses are shown in the upper right area:

- If UI's OK button is then clicked the following dialog will be displayed:

6.1.5.1 SimulationMode()

Description

The `SimulationMode()` function can be used to determine if the test program is executing on hardware or in simulation mode.

Usage

```
BOOL SimulationMode( );
```

`SimulationMode()` returns TRUE if the program is executing in simulation mode (not on hardware), otherwise FALSE is returned.

Example

```
if( ! SimulationMode() ){
   // Code which requires actual hardware support
}
```

6.1.6 UI Initial Display

The initial *Ui* display is shown below:

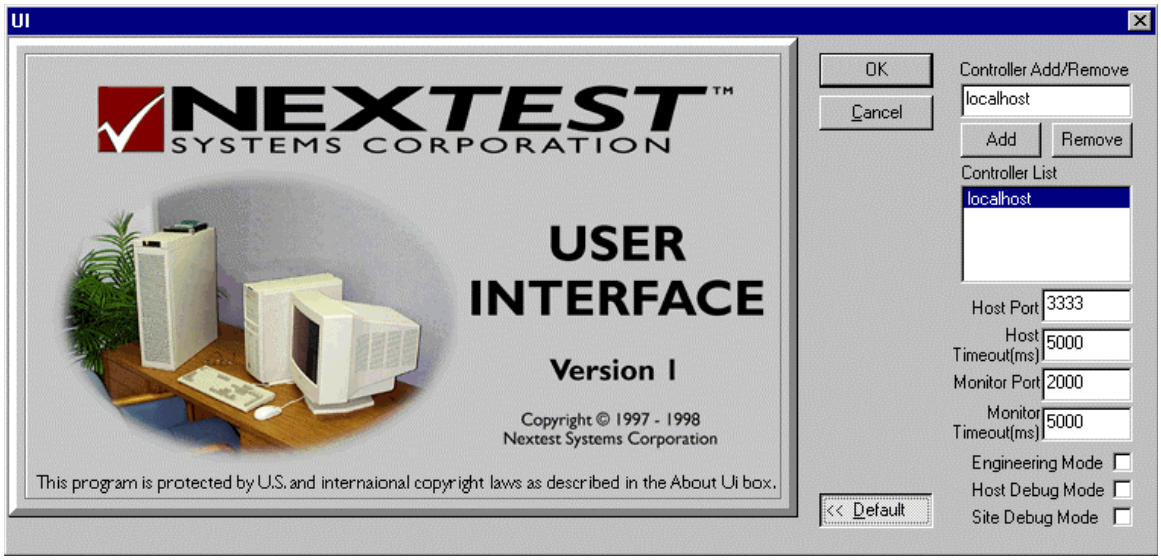
Over time, the Version number and picture seen here will change. However, the controls remain the same.

For production operations, click on the **OK** button, or press the **Enter** key on the keyboard to continue. During program development, click the advanced button. This enables various engineering and program debugging modes which may not be appropriate for production

operations. Clicking Advanced>> also allows setting advanced host to test site controller communication.

6.1.7 UI Advanced Option Controls

The advanced controls are shown below:



Engineering Mode

Check this item to enable various tools available in Ui to read and modify hardware values such as voltage, current, timing or pattern generator registers.

Host Debug Mode

Check this item to enable execution of Host process code (i.e. `CONFIGURATION()`, `HOST_CONFIGURATION()`, `HOST_BEGIN_BLOCK()`, `HOST_END_BLOCK()` and `INITIALIZATION_HOOK()`) within the *Microsoft Developer Studio* debug environment. See Host/Site/Tool Debug Mode(s).

Site Debug Mode

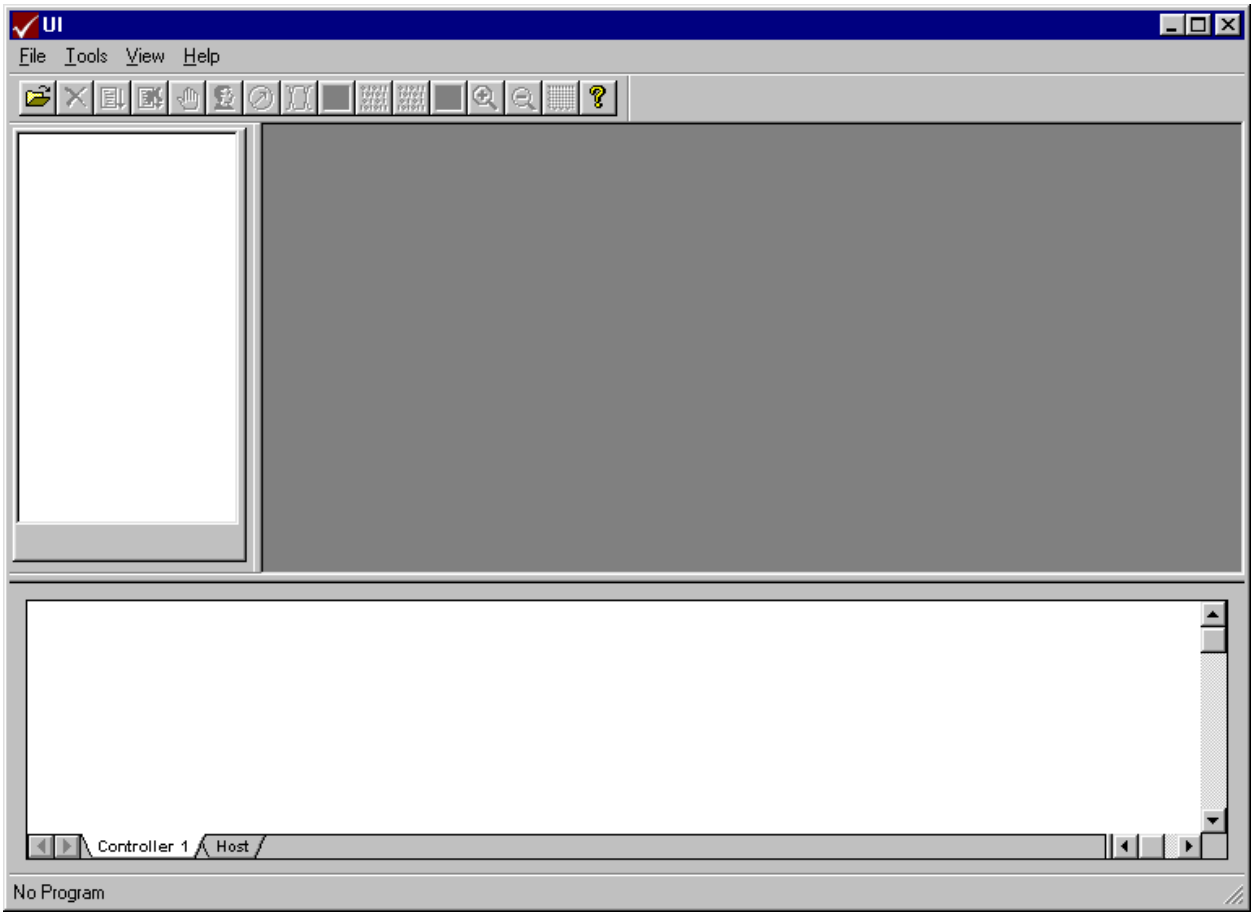
Check this item to enable execution of Site process code (i.e. `CONFIGURATION()`, `SITE_CONFIGURATION()`, `SITE_BEGIN_BLOCK()`, `SITE_END_BLOCK()`, `INITIALIZATION_HOOK()`, `SEQUENCE_TABLE()`, and all `TEST_BLOCK` related code) within the *Microsoft Developer Studio* debug environment. See Host/Site/Tool Debug Mode(s).

The other input fields in the advanced section of the *Ui* startup are for changing the TCP/IP parameters for test site controller connections.

6.1.8 UI Main Display

Note: each time *Ui* is terminated it records the size and position of the main window and all sub-windows. This is done so that *Ui* appears the same the next time it is started. These values are stored in the `ui.ini` file in the windows root directory (*C:\Windows* for *Windows 95*, or *C:\Winnt* for *Windows NT*). This file can be deleted to restore the default window configuration.

For the first invocation of *Ui*, or if the file `ui.ini` is deleted, the *Ui* default window will appear as shown below. This is the default display before a test program has been loaded.

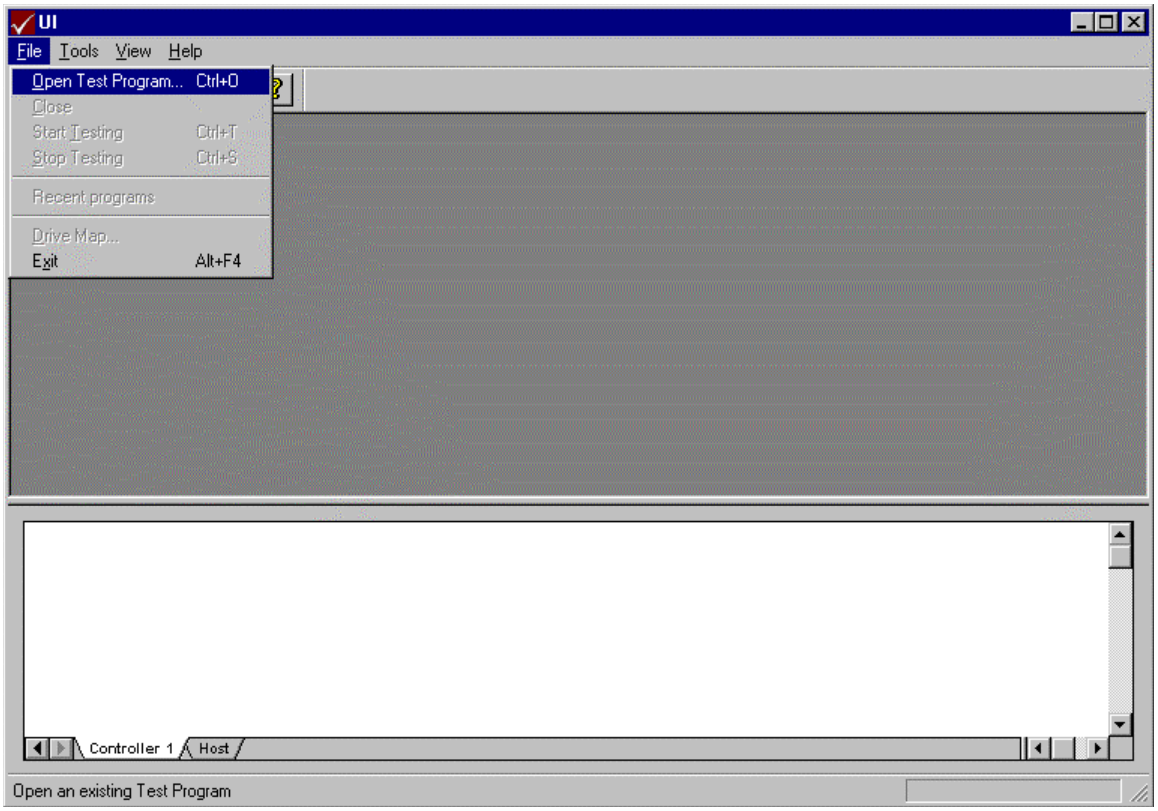


6.1.8.1 UI File Menu

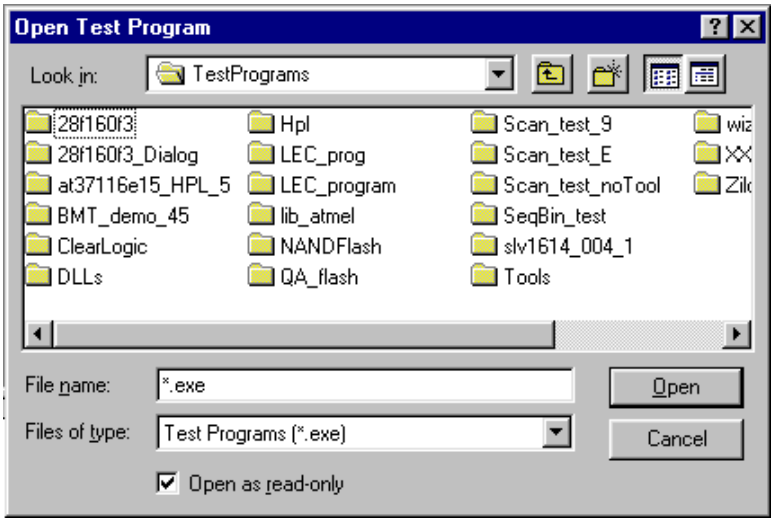
Test programs are loaded and unloaded from the **File** menu.

The **Open Test Program** menu item opens a standard file dialog for selecting the test program to be loaded (see below). In addition, a list of the most recently loaded test programs is maintained in the **File** menu. Double clicking a test program name in the list of

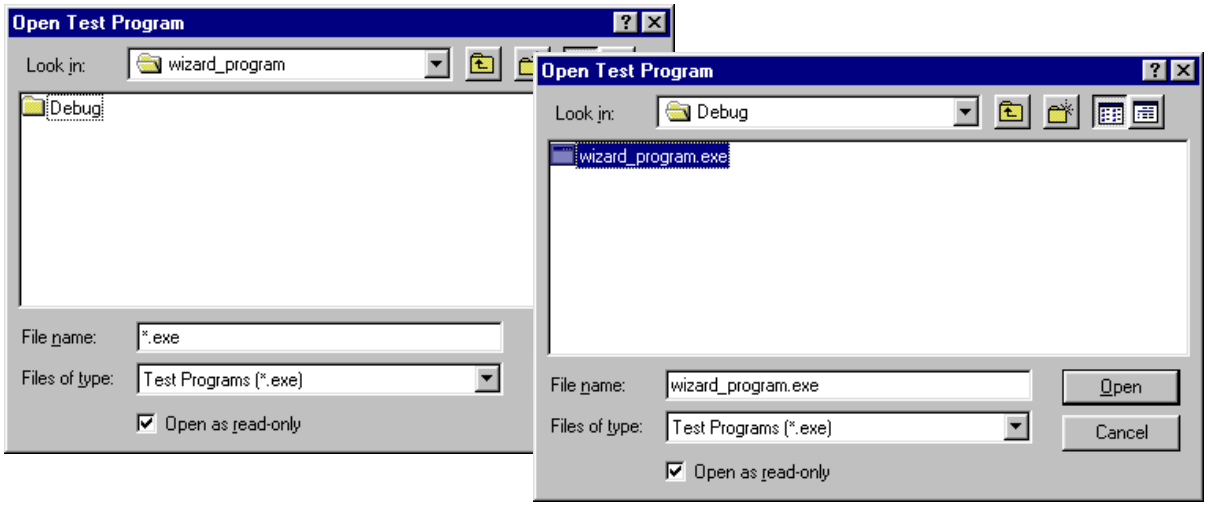
recent programs will load that program. A test program may also be loaded with the **`Ctrl+O`** keyboard shortcut:



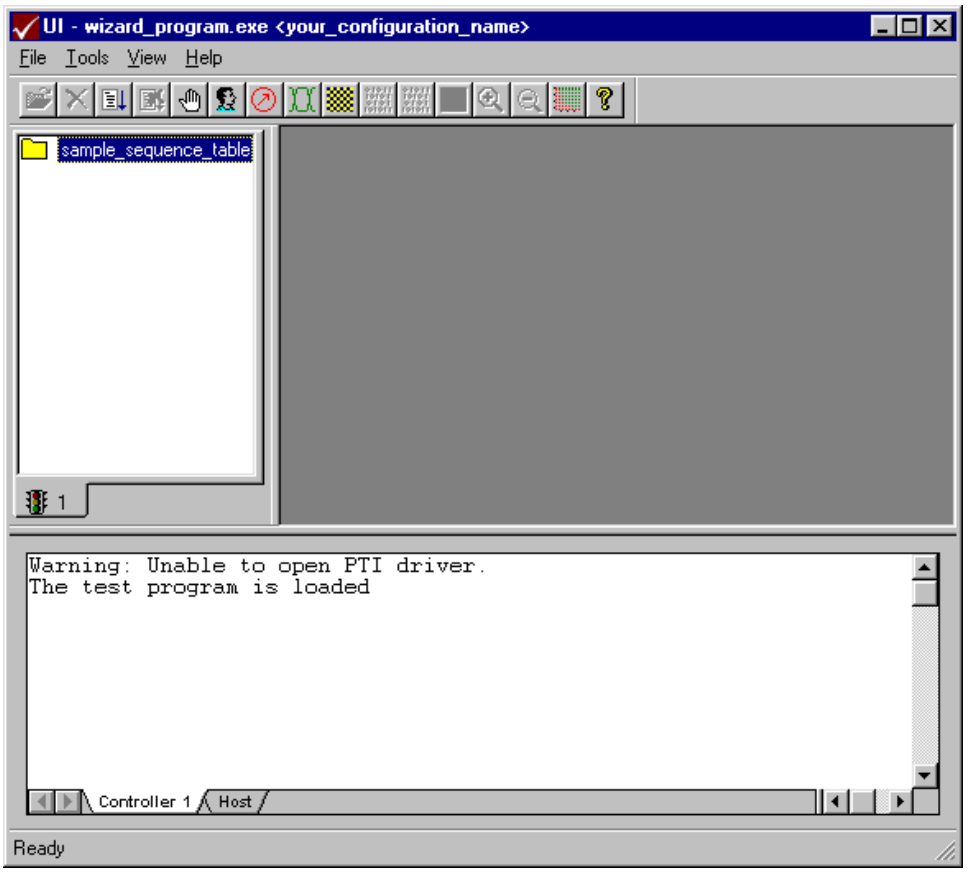
This will display the standard Microsoft file selection browser. Note in the *File name* cell that *Ui* wants an executable file name (*.exe). Locate and select the folder containing the test program to be loaded.



Then, select the *Debug* folder which contains the test program's executable file (*.exe). If the *Debug* folder doesn't exist you must compile the test program to create it and the executable file.



After the test program is loaded, the program name is displayed on the title bar of *Ui* along with the name of the `CONFIGURATION()`.



The message “Warning: Unable to open PTI driver” is normal when using UI without a connection to a test system.

While the program is loaded, the Sequence & Binning Table can be executed interactively using several methods, including **File: Start Testing**. See Executing the Sequence and Binning Table for more details.

An executing Sequence and Binning table can be aborted with the **Stop Testing** menu item or by typing the **Ctrl+S** keyboard shortcut. These two methods for halting a running test program are typically used as a last resort while debugging a test program stuck in an endless loop.

Note: the **Stop Testing** function now works reliably if the program is set in an infinite loop from the Breakpoint Monitor.

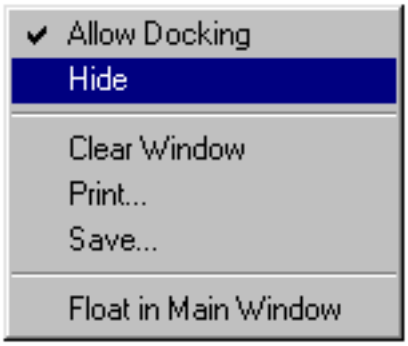
The **Close** item unloads the test program from all active test site controllers which is required before a new test program is loaded in the site controllers.

Exit (keyboard shortcut **Alt+F4**) will quit *Ui* altogether. If a test program is loaded, it will unload the test program first before exiting *Ui*. However, it is always a good idea to unload a test program using **Close** first before exiting *Ui*.

UI may be exited via the **File: Exit** menu item, by typing the **Alt+F4** keyboard shortcut, or by clicking on the window close button at the top right of the *Ui* window.

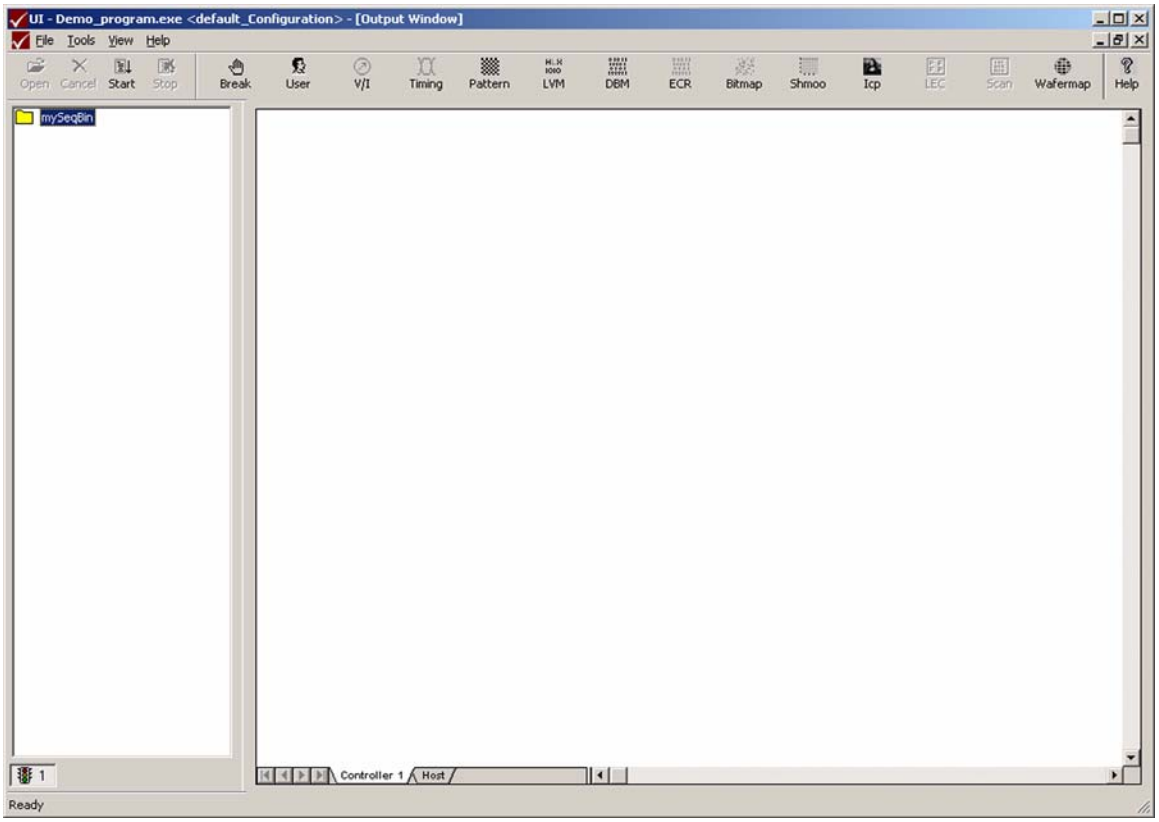
6.1.8.2 UI Window Hide and Dock

Using the right-mouse in one of the *Ui* sub-windows will display the menu below. Floating a sub-window in the main display makes it movable and resizable independent of the main display. Docking a sub-window causes it to lock in place relative to the main window. Sometimes the state of these sub-windows can get confusing - delete *C:\winnt\Ui.INI* and restart *Ui* to restore default operation,



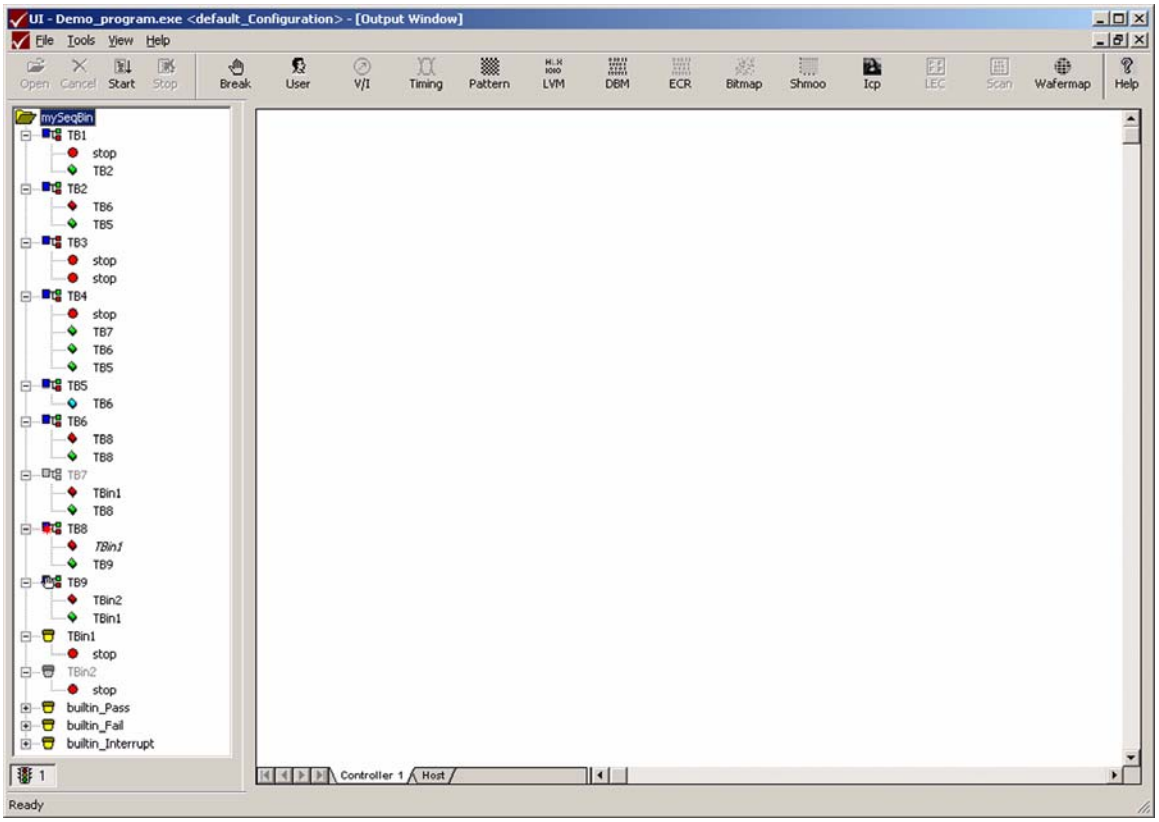
6.1.9 UI Sequence and Binning sub-window

Below, the Sequence & Binning Table table is minimized - double click on the folder symbol to expose the details of the test sequence.



UI displays a separate table tab for each site controller. Test result pass/fail status is shown as an indicator on the tab for each site. Simliar indicators are found each tab in the Breakpoint Monitor display.

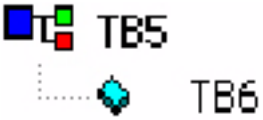
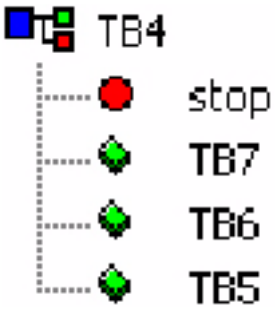
The test flow implemented via the Sequence and Binning table is now graphically represented. The small icons are further described below. Some of the icons shown were first available in software release h2.2.7/h1.2.7:

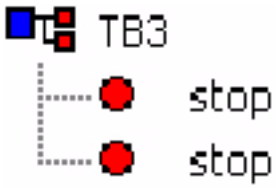


The image above and those below reflect the Sequence and Binning table code below. Note that the labels used in the code are translated into test blocks in the UI display and that the FAIL branch from TB8 was manually modified in UI (the code and the display don't match):

```
SEQUENCE_TABLE( mySeqBin){
   SEQUENCE_TABLE_INIT
   TEST(        TB1,    NEXT,   STOP                )
   TEST(        TB2,    L5,     L6                  )
   TEST(        TB3,    STOP,   STOP                )
   TEST4(       TB4,    NEXT,   SKIP,   L7,   STOP  )
   CALLL(  L5,  TB5                                 )
   TESTL(  L6,  TB6,    SKIP,   SKIP                )
   TESTL(  L7,  TB7,    NEXT,   L10                 )
   TEST(        TB8,    NEXT,   NEXT                )
   TEST(        TB9,    NEXT,   SKIP                )
```

```
   BINL(   L10, TBin1,  STOP                    )
   BIN(         TBin2,  STOP                    )
}
```

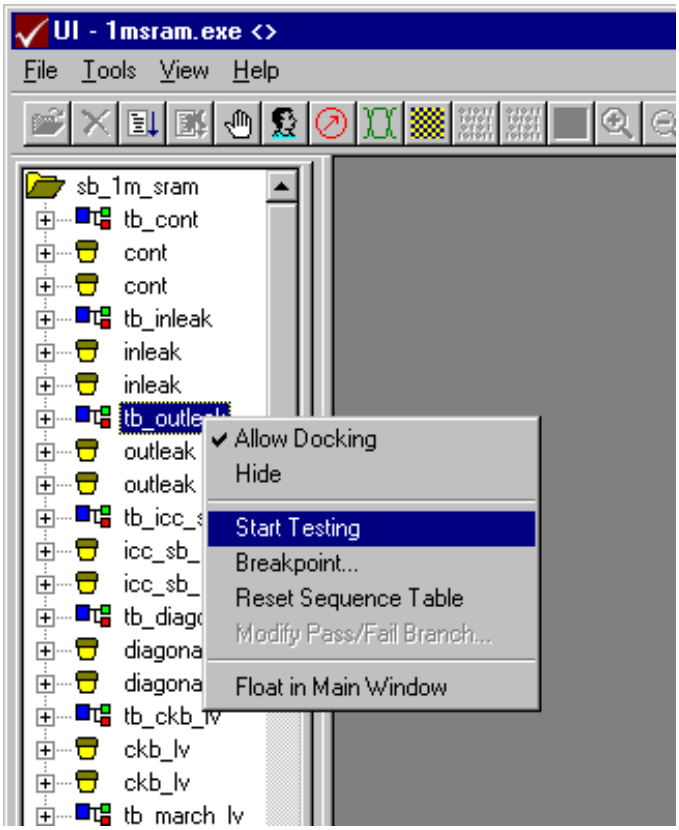
Icon	Purpose
	Sequence/Binning Table Symbol
	Test block symbol
	Test bin symbol
	PASS branch action
	FAIL branch action
	Unconditional branch action
	STOP branch action
	`CALL( TB5 )`// CALL always = unconditional branch
	`TEST( TB2, L5, L6 )` // Two branches = PASS/ FAIL Same as `TEST2(...)`.
	`TEST4( TB4, NEXT, SKIP, L7, STOP )`// 4 branches

Icon	Purpose
	`BINL( L10, TBin1, STOP )` // Test bin with STOP action
	`TEST( TB3, STOP, STOP )` // All branches STOP (new in software release h2.2.7 /h1.2.7)
	Test block execution is skipped. Caused by setting Skip breakpoint using Breakpoint Monitor. (new in software release h2.2.7 /h1.2.7)
	Test bin execution is skipped. Caused by setting Skip breakpoint using Breakpoint Monitor. (new in software release h2.2.7 /h1.2.7)
	Test block execution may stop due to breakpoint (Break-Before, Break-After or Loop). (new in software release h2.2.7 /h1.2.7)
	Test block branch was modified in UI. Note italicized label. (new in software release h2.2.7 /h1.2.7)

6.1.9.1 Modifying the Sequence and Binning Table

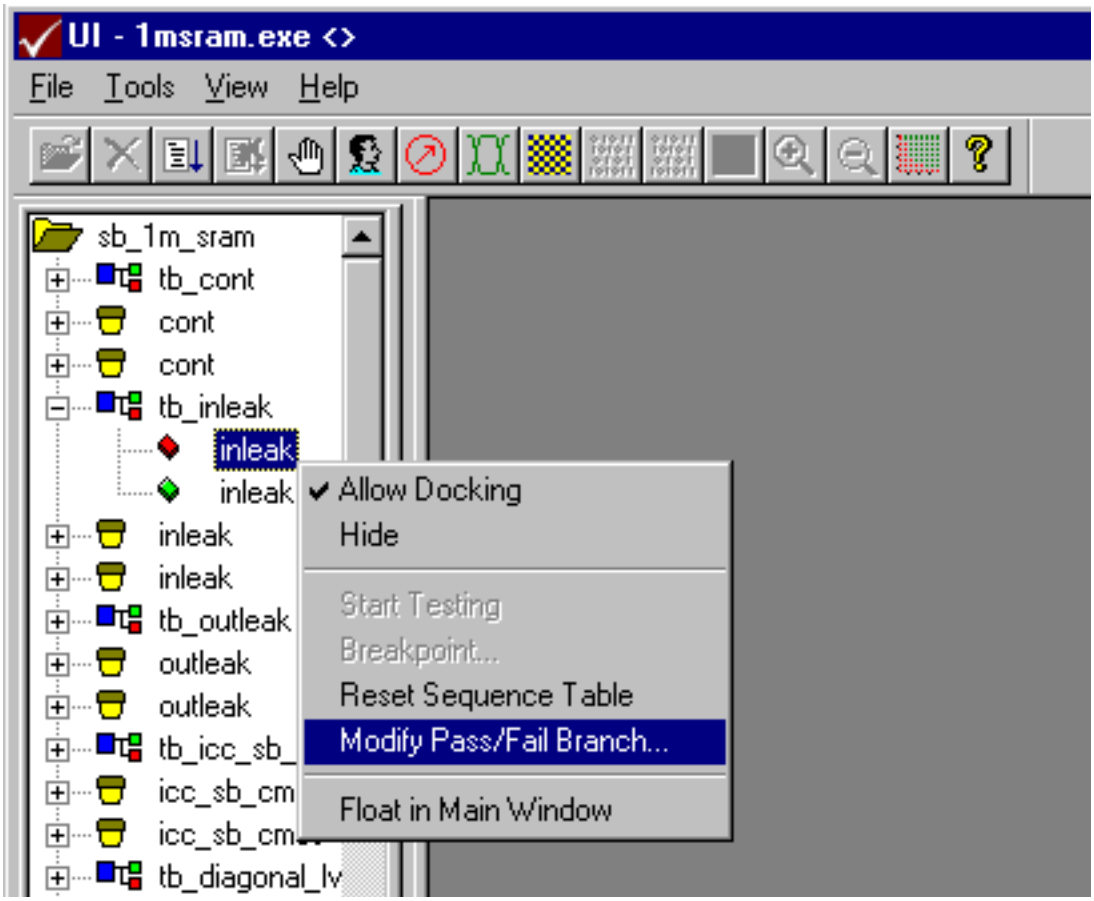
It is possible to modify the Sequence and Binning table from *Ui*, if Engineering Mode is enabled.

If a *Test Block* is selected, clicking the right mouse will display the following dialog, allowing the Sequence and Binning table to be executed starting at the selected test, skipping earlier tests in the sequence.

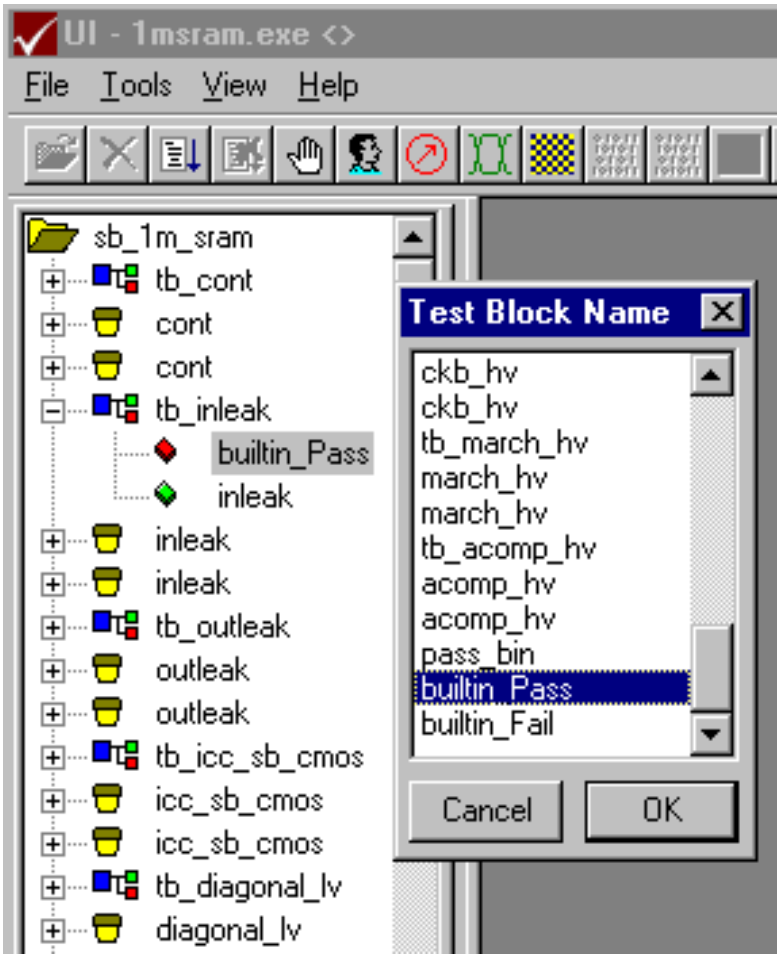


This does not modify the Sequence and Binning table.

If a *Bin* is selected, the following dialog is displayed, allowing the branch target to be modified, which modifies the test flow.



If this option is selected an additional menu of existing test blocks and bins is displayed to allow selection of the desired target.



This example shows the effect of selecting *builtin_Pass* and clicking **OK**.

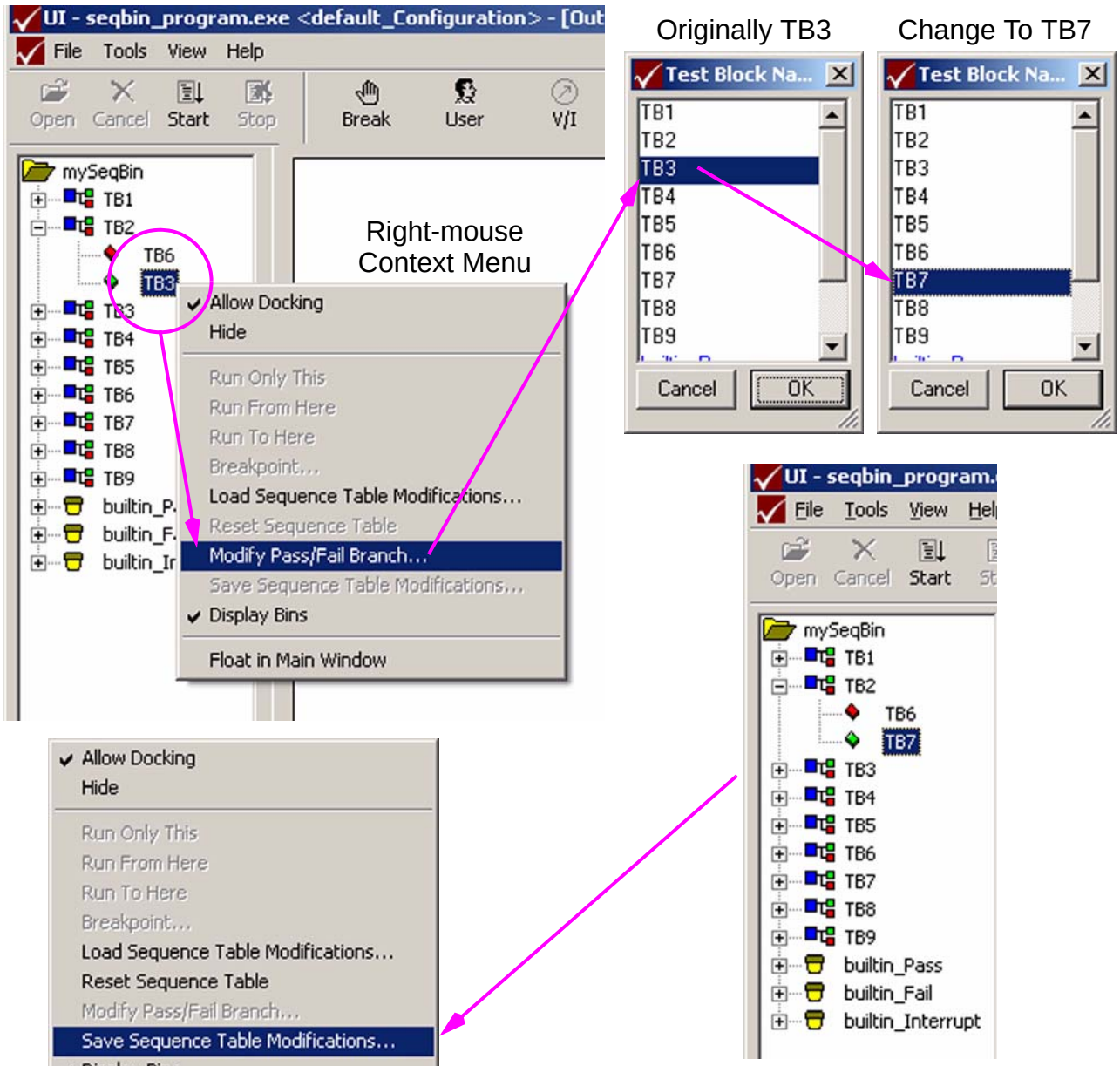
The contents of the *Test Block Name* menu are listed in the order of the original Sequence and Binning table. Since a given test block can occur multiple times in the sequence caution must be used not to create an endless loop. If this occurs, select **File: StopTesting** to interrupt the loop. The STOP action was added to the list in software release h2.2.7/h1.2.7.

Any number of changes can be made, but they are only valid for the duration of the program load session. And, the Sequence and Binning table can be restored to the original compiled configuration by using the right-mouse selecting *Reset Sequence Table*.

6.1.9.2 Save/Load Sequence/Binning Table Modifications

Note: first available in software release h2.2.7/h1.2.7.

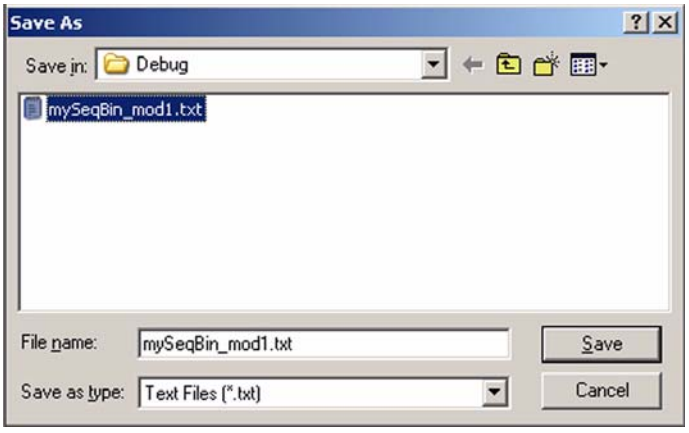
In UI, it is possible to modify the Sequence & Binning Table at run-time, see Modifying the Sequence and Binning Table. It is also possible to save and later reload these modifications, as shown below:



Select **`Save Sequence Table Modifications...`** to save changes to disk file.

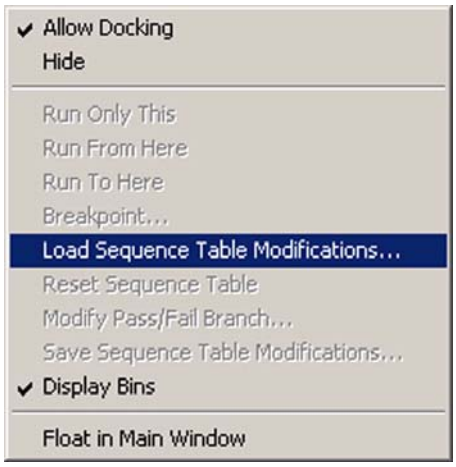
Once the Sequence & Binning Table is modified using the controls noted above it is possible to save the changes to a file on disk. Then, later, the saved file can be loaded to reproduce the edits saved earlier. Note the following:

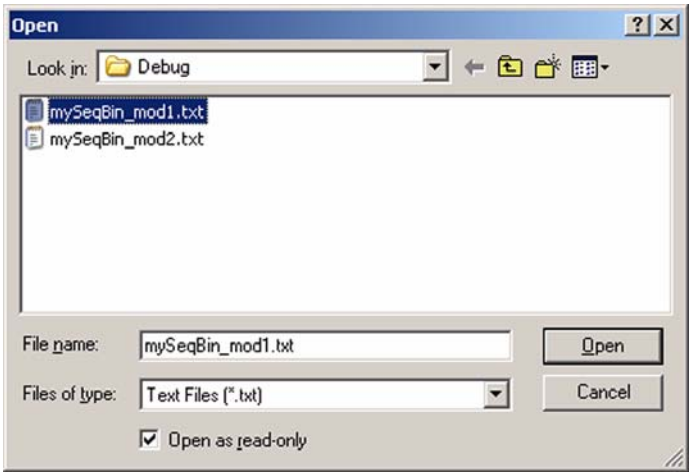
- After the test program is initially loaded, the **Save Sequence Table Modifications** menu selection is disabled until the Sequence & Binning Table is actually modified, either by using the **Modify Pass/Fail Branch...** control or by loading a previously saved file using the **Load Sequence Table Modifications...** menu selection.
- The **Save Sequence Table Modifications** menu selection is also disabled any time the **Reset Sequence Table** selection is invoked, to restore the Sequence & Binning Table to the original as-loaded configuration.
- Any time the **Save Sequence Table Modifications** selection is used all of the modifications made since the program was loaded or since the last **Reset Sequence Table** are captured to the specified disk file.
- Invoking **Save Sequence Table Modifications** displays a standard Windows file browser. The default location is the current test program's Debug\ folder:



- The file created by **Save Sequence Table Modifications** file is an ASCII (*.txt*) file. However, the format of this file is not documented and may change in future software releases.

- To load a previously saved file, use the **`Load Sequence Table Modifications...`** menu selection, and select the desired save file via the browser:





Note that one or more warning dialogs will be displayed in the following situations:

- If a file is loaded which contains format errors. This can occur if a saved file contains components which were subsequently deleted or renamed in the test program, or if the user has manually edited the file and introduced errors.
- If a file is loaded which does not result in any changes to the Sequence & Binning Table:

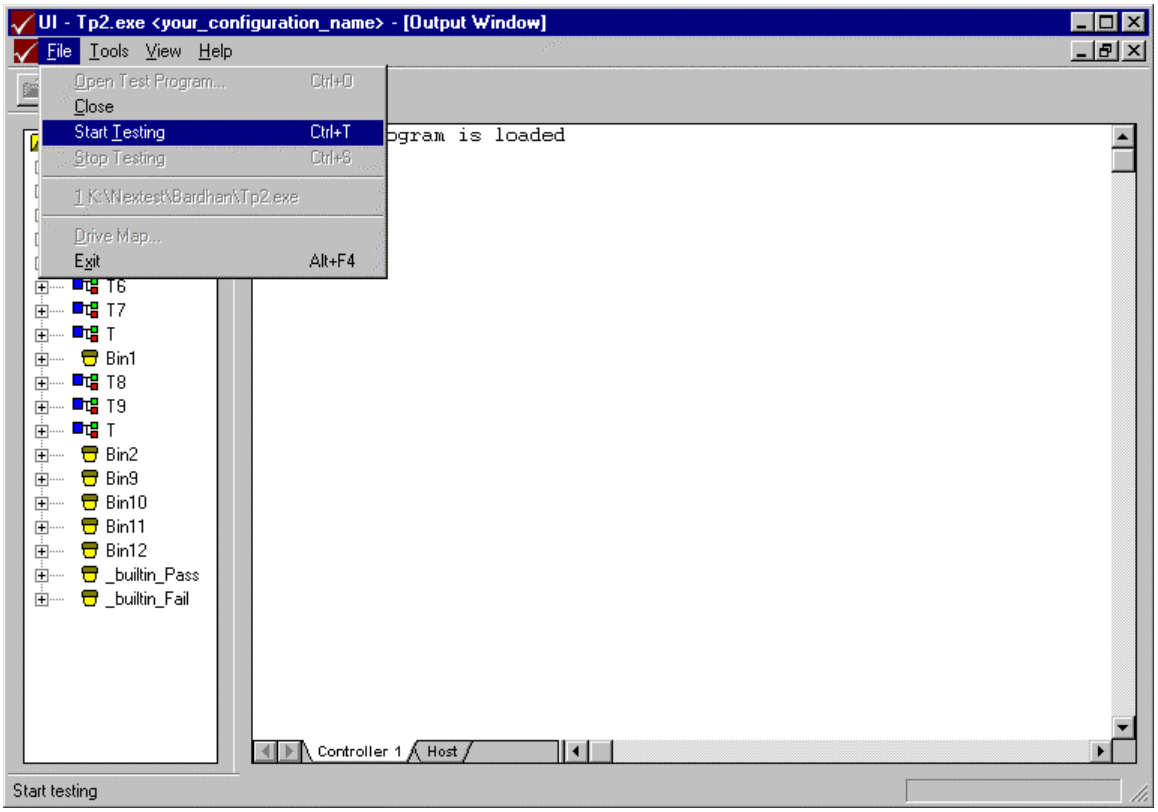
6.1.9.3 Executing the Sequence and Binning Table

The Sequence and Binning table can be interactively executed using several methods:

- Select **File: Start Testing**
- Typing the **Ctrl+T** keyboard shortcut
- Clicking the Start Test button from the Breakpoint Monitor.
- Using the right-mouse in the UI Sequence and Binning window. See Modifying the Sequence and Binning Table.
- A start test signal from external equipment (handler or prober).

When external handlers or probers are used, they generate the start test signal, which must be handled by C-code resident in (or called from) the Host Begin Block. This *auto start* signal initiates the start test directly bypassing UI.

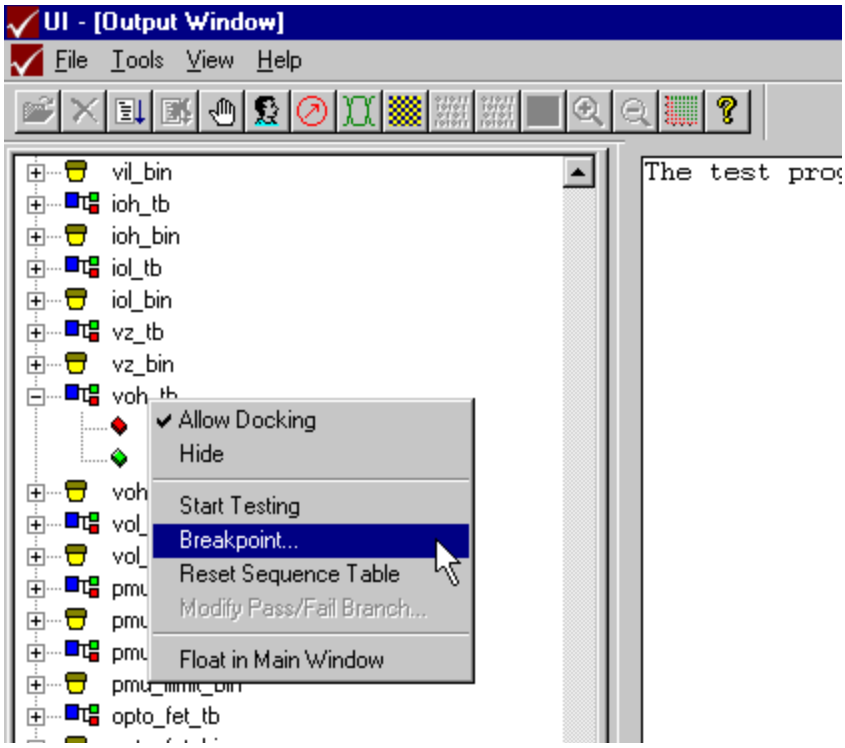
Generating *start test* using UI's **File: Start Testing** menu option:



An executing Sequence and Binning table can be aborted with the **Stop Testing** menu item or by typing the **Ctrl+S** keyboard shortcut. These are typically used as a last resort while debugging a test program stuck in an endless loop.

6.1.9.4 Starting the Breakpoint Monitor

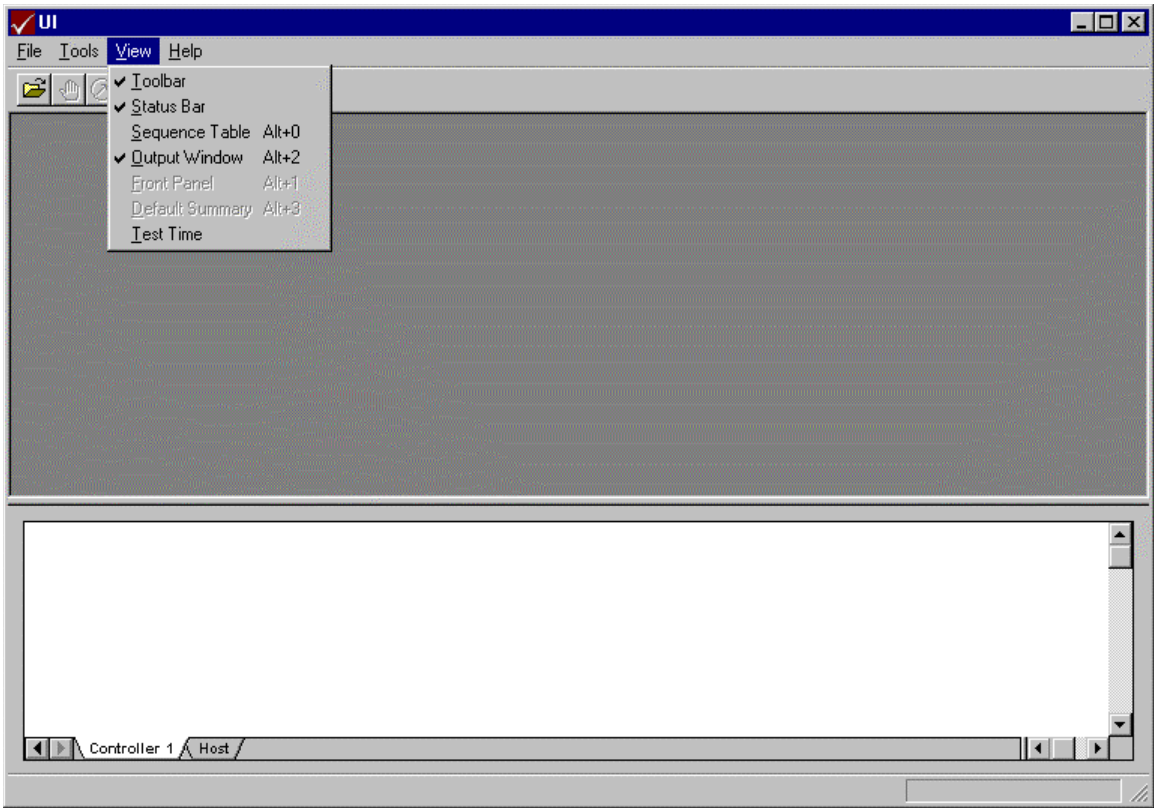
From the UI *Sequence View*, the user can now invoke the Breakpoint monitor by right-clicking on a Test Blocks and selecting **Breakpoint** from the pop up menu. This will display the Breakpoint monitor with the selected test inserted into the test block field. The user can then choose the desired Breakpoint Actions on that test.



6.1.10 *Ui* View Menu

The **View** menu of *Ui* will look like the following, when *Ui* is invoked for the first time before any test program is loaded. All the items in the View menu are read-only except the clear

button in the default summary, which is available only in engineering mode. The grayed out items are not available until a test program is loaded.:



Items can be check-marked or unmarked by selecting the item from the menu or by using the keyboard shortcut. The output window for example, will toggle between hiding and reappearing when **Alt+2** keys are pressed together repeatedly.

Toolbar: Has an icon for each item in the Tools menu (default is ON).

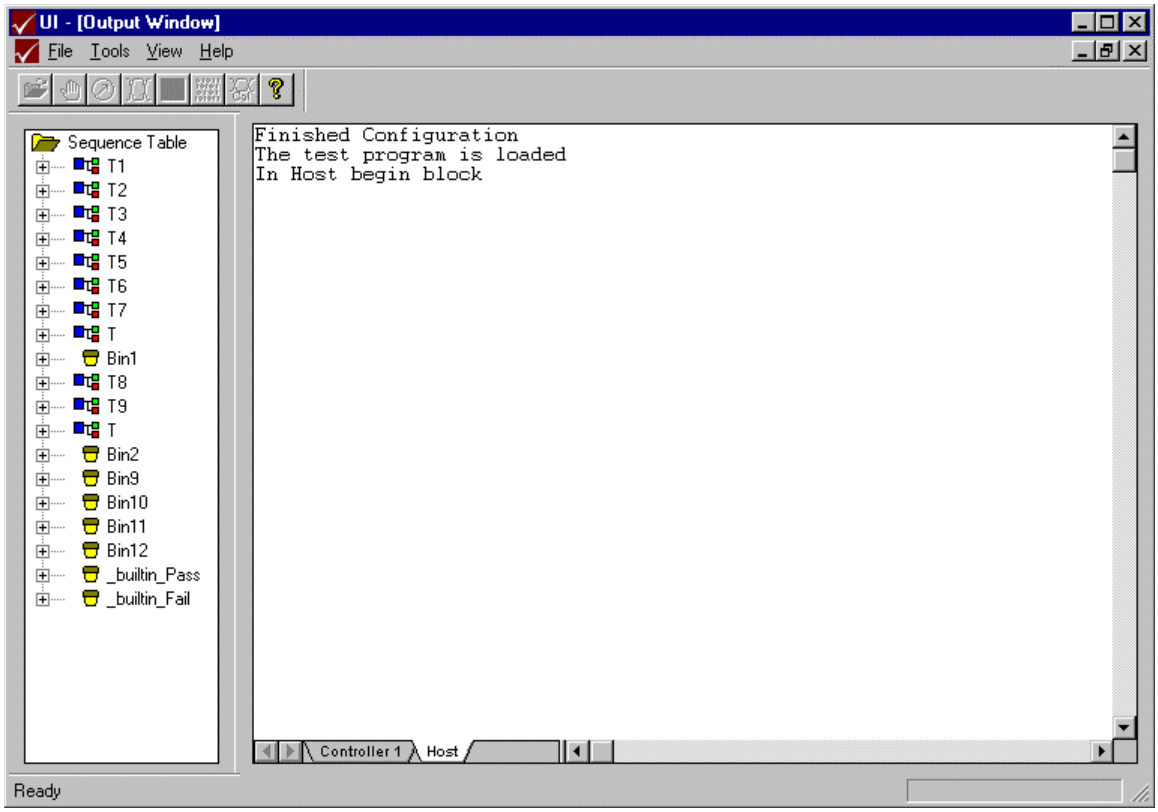
Status Bar: The bar at the bottom of the *Ui* window to display program status information such as *Ready*, *Running*, stopped at a Breakpoint etc. on the right hand side of the status window. Status bar also displays the test time in the box at the left hand side of the status bar if **Test Time** option is checked menu (default is ON).

Sequence Table: This option is turned on (default is OFF) after a test program is loaded. The shortcut key for this option is **Alt+0**.

Output Window: This option is to turn on/off (default is ON) the console window for test program output, system errors and warning messages. There is one tab for the Host window and one per test site controller. The context menu (available by right clicking on the window) allows the user to clear, print or save the content of the window to a file. It is also possible to

hide the window from the context menu. However, the window reappears when new text arrives from the test program. . The shortcut key for this option is **Alt+2**.

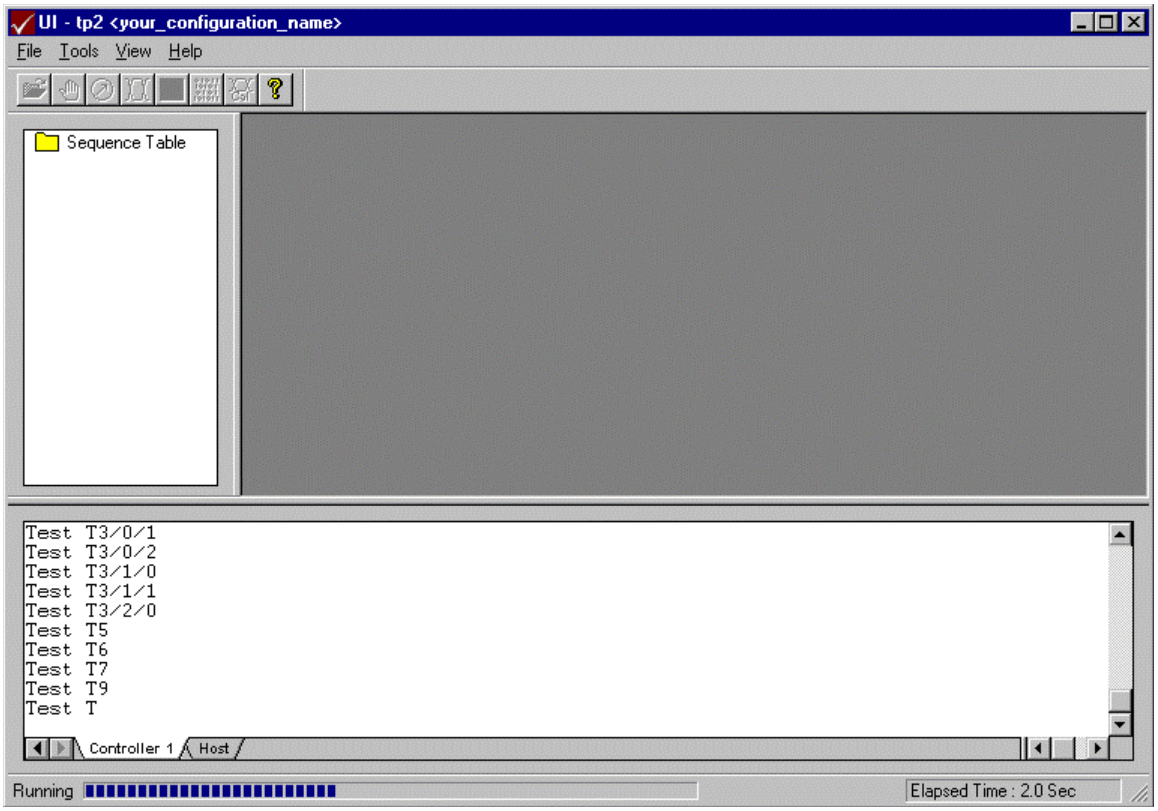
In the example below, the output from the Host Begin Block is displayed in the Host output window (note the Host tab is selected):



Front Panel: see FrontPanelTool. The shortcut key for this option is **Alt+1**.

Default Summary: see SummaryTool. The shortcut key for this option is **Alt+3**.

Test Time: This option toggles the running test time display in the status bar (default is OFF). The resolution of this stopwatch is 500ms.



Note the following:

- These time values are not very accurate. *Ui* starts the timer with **Start Testing** and stops it when the test done signal is received from all active site controllers.
- To get an accurate test time, the Elapsed Time facilities should be used from within the test program.

6.1.10.1 UI Output Window

The following standard Windows features are usable in UI's output window:

- Select text to be copied, printed, or saved to file. Use the mouse, or standard Windows keyboard shortcuts (Ctrl-A, shifted arrows, control-shift Home/End, etc.). Select all text in the output window using **Select All** via the right-mouse button, or Ctrl-A.
- Copy selected text to the clipboard, using the right-mouse button, or Ctrl-C.
- Search in the output window using **Find...** via the right-mouse, or using Ctrl-F. **Find Next** uses the **F3** button, **Find Previous** uses the **Shift-F3** key.
- Print all or selected text using the **Print...** right-mouse button.
- As before, it is remains possible to **Save** the entire output window to a disk file. Now, if text is selected, only that text is saved to the disk file.
- By default, invoking **Save** or **Save-all** saves the selected information a text (.txt.) Beginning in software release h3.3.xx, some font attributes may be manipulated (color, font size, bold, italic, underline, etc.), see Output/Warning/Fatal Text Format Options. And, messages generated by `warning()` and `fatal()` will be displayed using red text. Beginning in this release the **Save** and **Save-all** selections allow saving messages in Rich-text format (*.rtf*) to allow the saved text to include these font attributes (saving as plain text does not).

Note: it is not possible to *Cut* text from the output window. And, **Clear Window** only clears the entire window.

6.1.11 *Ui* Tools Menu

The **Tools** Menu on the *Ui* menu bar has the engineering tools necessary for debugging test programs. All of the menu items except **Option** are disabled in the normal mode. All of the tools are available in the engineering mode once a test program is successfully loaded. In some cases, optional hardware has to be present for the tool to be enabled (DBMTool for example). Each tool is also represented as an icon on the *Ui* toolbar. There are three ways to invoke a tool:

- Use the mouse to select the appropriate item from the **Tools** pull down menu.
- Click on the tool icon on the *Ui* toolbar.
- Use the keyboard shortcut. For example, **Ctrl+K** for the Breakpoint Monitor.

6.1.12 User Menus in UI

The `menu_add()` function is used to add a user-defined icon to UI - User Interface menu bar. Clicking on the menu item will invoke the body code of an associated CSTRING_VARIABLE User Variables.

The `menu_add()` function also supports the following optional features:

- Initial enable/disable state of the menu item
- user-defined accelerator key specification
- Timeout

These are described in more detail in the Usage section.

Clicking on the menu item does not disable it. The `menu_enable()` function can be used to both enable and disable a menu item. See Example.

The `menu_delete()` function can be used to remove the menu item from the tool bar.

Note: once a menu item is deleted any code which attempts to access that item will no longer function correctly (program may crash).

The C code which adds/deletes a menu item to UI - User Interface can execute in a Host, Site, or User Tools process.

Usage

```
BOOL menu_add( LPCTSTR menu_description,
               VariableProxy variable,
               DWORD timeout DEFAULT_VALUE( INFINITE ),
               BOOL enable DEFAULT_VALUE( TRUE ),
               LPCTSTR accel DEFAULT_VALUE( 0 ) );
BOOL menu_add( LPCTSTR menu_description,
               VariableProxy variable,
               LPCTSTR accel,
               BOOL enable DEFAULT_VALUE( TRUE ),
               DWORD timeout DEFAULT_VALUE( INFINITE ) );
BOOL menu_delete( LPCTSTR menu_description );
```

```
BOOL menu_enable( LPCTSTR menu_description,
                  BOOL enable );
```

where:

`menu_add()` will add **`menu_description`** to UI's menu bar.

`menu_enable()` will enable or disable **`menu_description`** in UI's menu bar.

`menu_delete()`will delete **`menu_description`** from UI's menu bar.

`menu_description` is a string describing the desired menu or submenu name. The '`/`' character may be used in the string to delimit menu vs. sub-menu relationship. The '`-`' character can be used in the string as a separator between menu items. The Ampersand '`&`' can be added in front of any given letter to underline that character indicating a short-cut key.

`variable` is a user-created `CSTRING_VARIABLE`. When a particular menu item is selected it is the body code of the corresponding `CSTRING_VARIABLE` which is executed, in the process which added the menu to UI. The value of this user variable will be the actual item selected in the user menu, as a string. The prompt for the user variable is displayed in the UI status bar.

`timeout` is optional, and is the amount of time to wait before notifying the user that the user variable body code execution did not complete.

`enable` is optional, and specifies whether the menu item is enabled (TRUE) or disabled (FALSE).

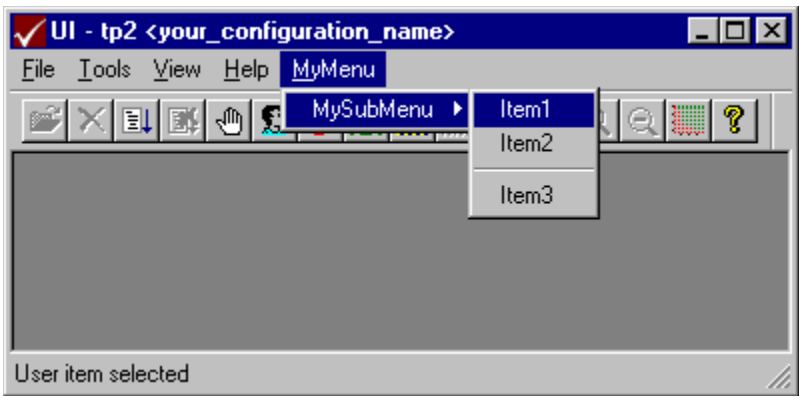
`accel` is optional, and if used specifies an acceleration key to be linked to the menu item. The format must be:
[C][A]-key, where C (Control) and A (Alt) are optional and key is a case-sensitive alphanumeric character. The - (dash) delimits the C/A keys from the following key, and must be included in the definition but is not typed to invoke the accelerator. Invalid character combinations are reported as a warning in the UI Host output window. See Example 2:

All 3 functions return TRUE if the operation was successful and FALSE if any errors occur.

Example

Example 1:

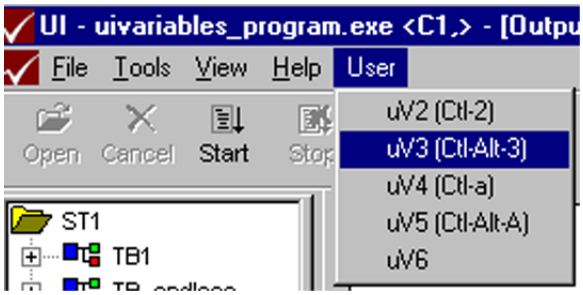
The following user menus will be added to UI using the C-code which follows:




```
CSTRING_VARIABLE( csv, "", "Select user item" ) {
   if ( csv == "Item1" ) {} // Execute this code if Item1 is selected
   if ( csv == "Item2" ) {} // Execute this code if Item2 is selected
   if ( csv == "Item3" ) {} // Execute this code if Item3 is selected
}
HOST_BEGIN_BLOCK( your_host_block_name ) {
   ... other code here ...
   // Ampersand(&) can be added in front of a letter to specify a
   // mnemonic key. In the following example, Alt+M will select
   // MyMenu from Ui top menu. Care should be taken in choosing a
   // mnemonic key, to avoid collision with existing mnemonic keys
   // from Ui
   menu_add("&MyMenu/MySubMenu/Item1", csv);
   menu_add("&MyMenu/MySubMenu/Item2", csv);
   menu_add("&MyMenu/MySubMenu/-", csv);
   menu_add("&MyMenu/MySubMenu/Item3", csv);
   ... other code here ...
}
```

Example 2:

The following example uses the accelerator features:




```
CSTRING_VARIABLE( UV2, "?", "UV2") {output("  UV2 => %s", UV2 );}
CSTRING_VARIABLE( UV3, "?", "UV3") {output("  UV3 => %s", UV3 );}
CSTRING_VARIABLE( UV4, "?", "UV4") {output("  UV4 => %s", UV4 );}
CSTRING_VARIABLE( UV5, "?", "UV5") {output("  UV5 => %s", UV5 );}
CSTRING_VARIABLE( UV6, "?", "UV6") {output("  UV6 => %s", UV6 );}

HOST_BEGIN_BLOCK( your_host_block_name ) {
   ... other code here ...
   menu_add ("UV2 (Ctl-2)", UV2, INFINITE, TRUE, "C-2" );
   menu_add ("UV3 (Ctl-Alt-3)", UV3, INFINITE, TRUE, "CA-3" );
   menu_add ("UV4 (Ctl-a)", UV4, INFINITE, TRUE, "C-a" );
   menu_add ("UV5 (Ctl-Alt-A)", UV5, INFINITE, TRUE, "C-A" );
   menu_add ("UV6", UV6, INFINITE, TRUE, "C-@" ); // Bad
   ... other code here ...
}
```

Typing the following key combinations will execute the body code of the associated user variable. Keys must be pressed at the same time. The - (dash) is not typed. Note that the accelerator specified for UV6 is invalid (as an example):

```
Control-2 invokes UV2 body code

Control-Alt-3 invokes UV3 body code

Control-a invokes UV4 body code

Control-Alt-Shift-A invokes UV5 body code

Control-Shift-2 does nothing. A warning is issued during program
load.
```

6.1.13 User Icons in UI Tool Bar

Description

The `toolbar_add()` function is used to add a user-defined icon to UI's tool bar. Clicking on the icon will invoke the body code of an associated CSTRING_VARIABLE User Variables.

One or two bitmap images may be defined to represent an icon. These are identified as *Cold* and *Hot* where the cold image is the default display and is replaced by the hot image when the mouse focus moves to the icon.

The `toolbar_add()` function also supports the following optional features:

- user-defined accelerator key
- Initial enable/disable state of the icon
- Timeout

These are described in more detail in the Usage section.

If defined, the prompt argument (parameter 3) of the CSTRING_VARIABLE definition is displayed as a tool-tip when the mouse focus moves to the icon.

Clicking on the icon does not disable it. The `toolbar_enable()` function can be used to both enable and disable a user icon. See Example.

The `toolbar_delete()` function can be used to remove the icon from the tool bar.

Note: once an icon is deleted any code which attempts to access the icon will no longer function correctly (program may crash). See example.

The C code which adds/deletes an icon to the toolbar can execute in a Host, Site, or User Tools process.

Usage

```
BOOL toolbar_add( LPCTSTR toolbar_description,
                  VariableProxy variable,
                  int hot DEFAULT_VALUE( 0 ),
                  int cold DEFAULT_VALUE( 0 ),
```

```
                    LPCTSTR accel DEFAULT_VALUE( 0 ),
                    BOOL enable DEFAULT_VALUE( TRUE ),
                    DWORD timeout DEFAULT_VALUE( INFINITE ) );
  BOOL toolbar_enable( LPCTSTR toolbar_description, BOOL enable );
  BOOL toolbar_delete( LPCTSTR toolbar_description );
```

where:

`toolbar_add()` will add an icon to UI's tool bar.

`toolbar_enable()` will enable or disable the specified icon in UI's tool bar.

`toolbar_delete()` will remove an icon from UI's tool bar. Once an icon is deleted any code which attempts to access the icon will no longer function correctly (program may crash). See example.

`toolbar_description` identifies the icon and is the label displayed under the icon.

`variable` is a user-defined `CSTRING_VARIABLE`. When the icon is clicked the body code is executed, in the process which added the icon to UI. The value of this user variable is not affected by clicking on the icon. The prompt for the user variable is displayed as the tool-tip for the icon.

`hot` and **`cold`** are both optional, and identify one or two bitmaps which are displayed as icons in the tool bar. Each bitmap must be 16 pixels wide, 15 pixels tall, and the number of colors must be 16 (these can be set using the icon properties using the bitmap editor). The first bitmap is the default icon, the second bitmap is displayed, replacing the first, when the mouse focus is put on the icon.

`accel` is optional, and if used specifies an acceleration key to be linked to the tool bar item. The format must be:
[C][A]-key, where C (Control) and A (Alt) are optional and key is a case-sensitive alphanumeric character. The - (dash) delimits the C/A keys from the following key, and must be included in the definition but is not typed to invoke the accelerator. Invalid character combinations are reported as a warning in the UI Host output window.

`enable` is optional, and specifies whether the icon is enabled (`TRUE`) or disabled (`FALSE`). Clicking on a disabled icon has no affect.

`timeout` is optional, and is the amount of time to wait before notifying the user that the user variable body code execution did not complete.

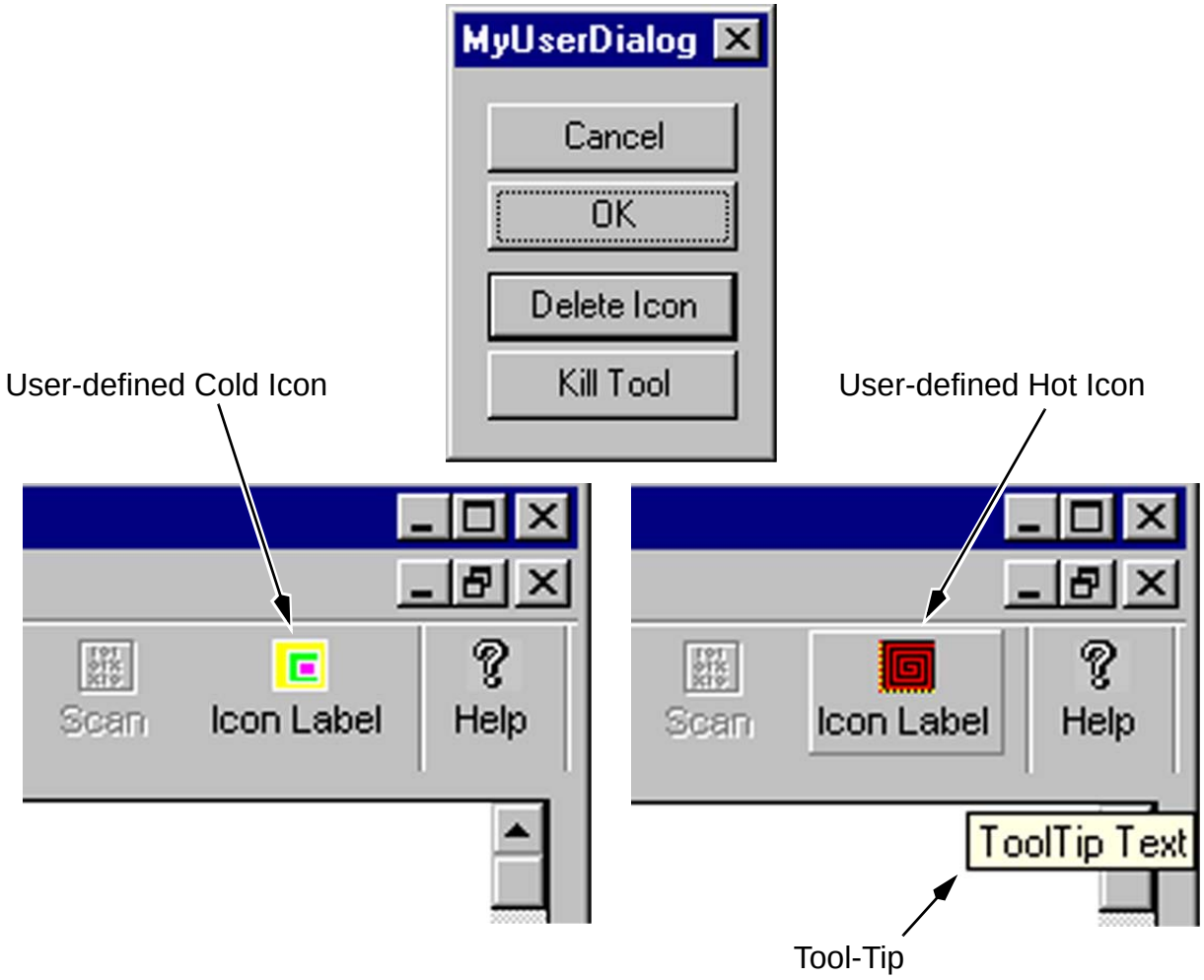
All 3 functions return `TRUE` if the operation was successful and `FALSE` if any errors occur.

Example

The code adds an icon to UI's tool bar. The icon is labeled *Icon Label*. When mouse focus is placed on the icon the user-defined tool-tip (*Tool-tip Text*) is displayed. The graphic dialog resource definition and the two bitmap definitions are up to the user.

The code also implements a simple user tool containing a dialog with 4 buttons:

- Cancel - built-in button which terminates the dialog
- OK - built-in button which terminates the dialog
- Delete Icon - removes the icon from the tool bar. Note the `icon_valid` variable is used to prevent subsequent attempts to access the deleted icon. This is the responsibility of the user's code.
- Kill Tool - user-defined button which terminates the tool:



```
#include "tester.h"
#include "resource.h"
EXTERN_DIALOG( D1 )

static BOOL icon_valid = FALSE;

// Body code executes when icon is clicked
CSTRING_VARIABLE( InvokeDialog,"","ToolTip Text") {
   run_modeless( D1 );
   icon_valid = TRUE;
   BOOL ok = toolbar_enable("Icon Label", FALSE );
   if( ! ok ) output("ERROR: toolbar_add() returned FALSE");
}

// Handle Cancel button. Order of body code is important.
VOID_VARIABLE( MyCancelFunc,"") {
   if( ! icon_valid ) return;
   BOOL ok = toolbar_enable("Icon Label", TRUE );
   if( ! ok ) output("ERROR: toolbar_add() returned FALSE");
   focus(variable); // Terminate dialog, must be last
}

// Handle OK button. Order of body code is important.
VOID_VARIABLE( MyOkFunc,"") {
   if( ! icon_valid ) return;
   BOOL ok = toolbar_enable("Icon Label", TRUE );
   if( ! ok ) output("ERROR: toolbar_add() returned FALSE");
   focus(variable); // Terminate dialog, must be last
}

// Delete icon
VOID_VARIABLE( MyDelIcon,"") {
   BOOL ok = toolbar_delete("Icon Label");
   if( ! ok ) output("ERROR: toolbar_delete() returned FALSE");
   icon_valid = FALSE;
}

// Terminate tool
VOID_VARIABLE( MyKillTool,"") { testprogexit(); }

DIALOG( D1 ) {
   CONTROL( IDCANCEL,       MyCancelFunc );
   CONTROL( IDOK,           MyOkFunc );
```

```
   CONTROL( IDC_DELICON,    MyDelIcon);
   CONTROL( IDC_KILLTOOL,   MyKillTool );
}
TOOL_BEGIN_BLOCK( my_TBB_name ) {
   BOOL ok = toolbar_add( "Icon Label",
                          InvokeDialog,
                          IDB_HOT,
                          IDB_COLD,
                          "C-1",
                          TRUE );
   if( ! ok ) output("ERROR: toolbar_add() returned FALSE");
}
```

6.1.14 Host/Site/Tool Debug Mode(s)

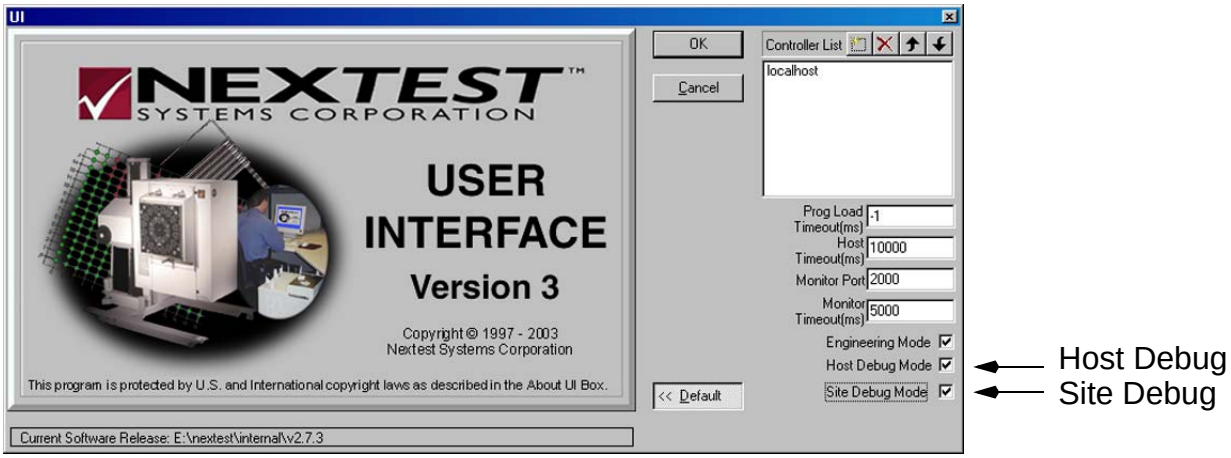
Note: on 10/1/01 information in the section titled Debugging With Developer Studio was moved to this section.

Description

Microsoft Developer Studio (*MsDev*) provides powerful source debug facilities, none of which are documented here. What is covered here are the basic steps used to enable debugging a Magnum 1/2/2x test program using Developer Studio.

Note: the information included here regarding Site debug applies to PT only i.e. a single site system. Contact your Nextest Applications Support specialist for information on debugging Site code on multi-site test systems (ST, VT, GT), including all Magnum 1/2/2x system configurations.

The process of enabling MsDev debug begins when UI is first started, using UI Advanced Option Controls to enable **`Host Debug Mode`**, **`Site Debug Mode`**, or both:



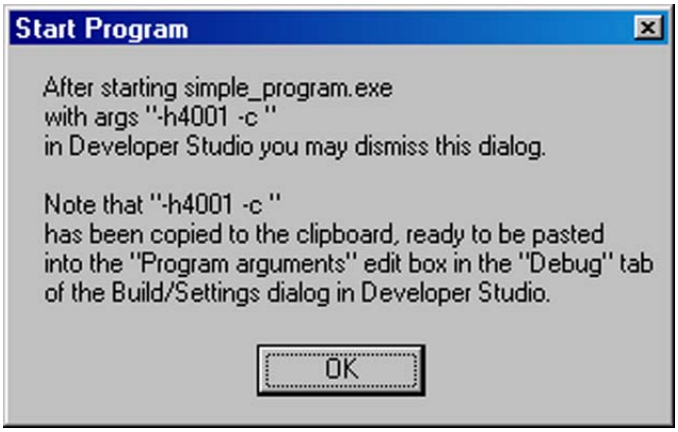
A separate instance of *MsDev* must be running for each process (Host and/or Site) which is to be debugged.

When a test program is loaded in UI the following steps must be followed to complete the enabling process.

Note: the following steps assume both Host and Site Debug Modes are enabled. When only one is enabled, ignore the steps which apply to the other process.

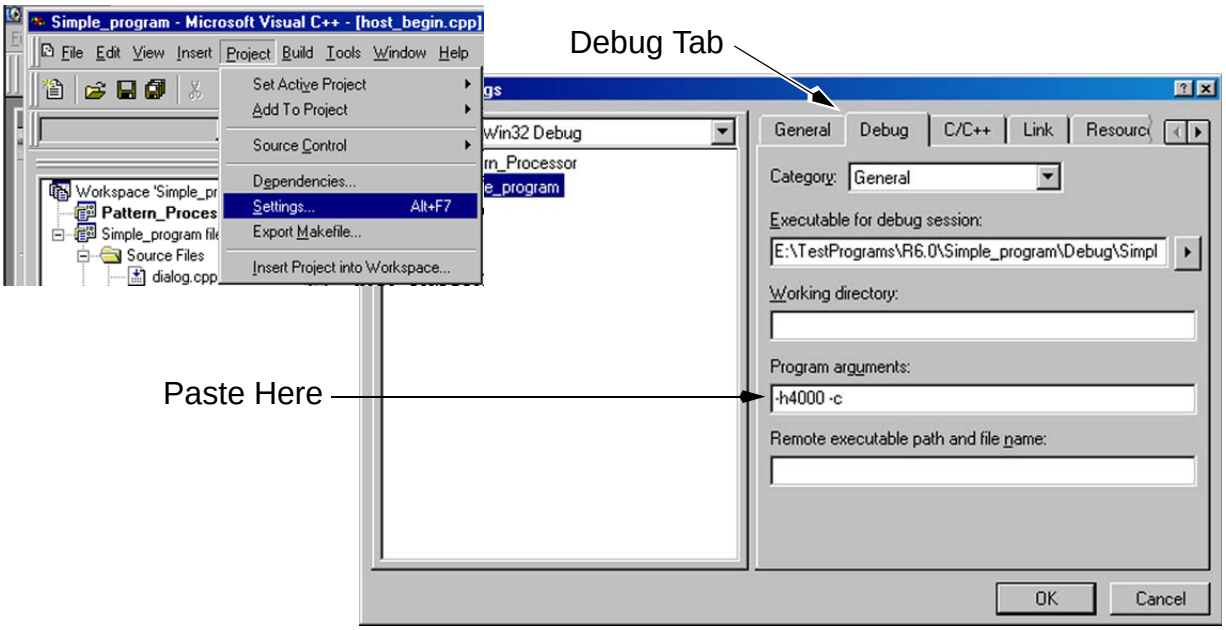
Instructions are also included to perform User Tool Debug.

1. Note the following dialog is created (sometimes it starts minimized or hidden behind other windows). This dialog is from UI, and is referring to a Host Debug Mode step:



Read these instructions but ***DO NOT*** click OK yet. Note that some very useful information has been copied to the clipboard, for use in the following step. To reduce confusion, you should complete the following step before copying anything else to the clipboard.

2. Locate the instance of MsDev which is to be used to debug Host process code. Click on **`Project > Settings...`** and select the **`Debug`** tab. In this dialog, click in the **`Program arguments`** window and invoke paste (**`^v`**). Confirm that the information noted in the previous dialog is correctly pasted into this window (as shown). Click **`OK`**..

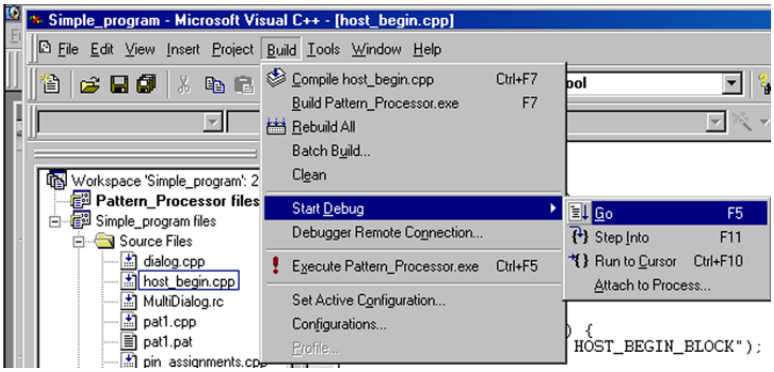


3. Before proceeding, set any breakpoints in Host code as needed. The basic method is to click on the desired line of source code and press the **F9** key to insert a breakpoint. Read the MsDev documentation for details on using and managing breakpoints.

Note: ***do not*** confuse breakpoints used when debugging source code with those set using the Breakpoint Monitor - they serve completely different purposes.

---

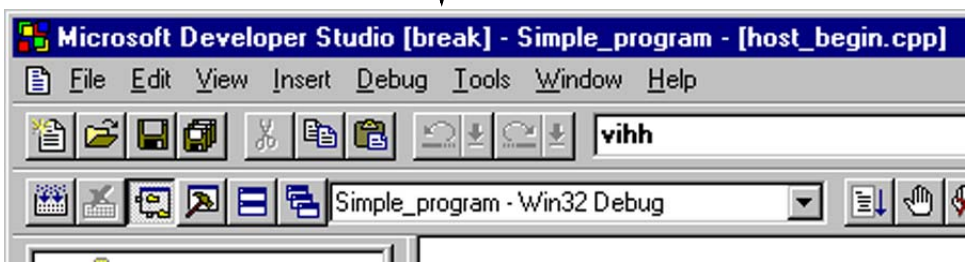
4. When ready to proceed select **Build > Start Debug -> Go**. Or, press the F5 key:



5. Next, locate the dialog shown in step-1, and click **OK**. This completes the setup of the Host Debug steps. The test program will begin to load and execute in the Host process (but not the Site yet), which includes all code in `CONFIGURATION()`, `HOST_CONFIGURATION()`, `HOST_BEGIN_BLOCK()`, and `INITIALIZATION_HOOK()`.

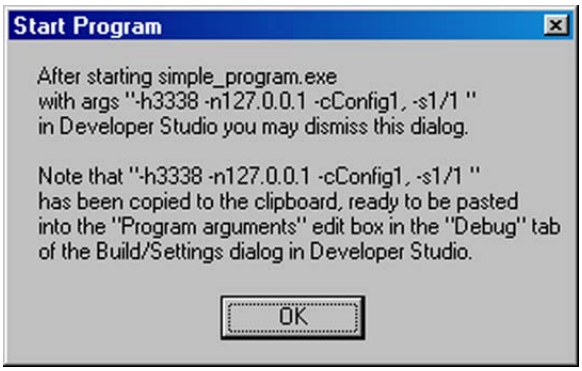
Note: if source debug breakpoints are set, Host code execution may stop at one of these breakpoints. When this ocurs, it may appear as a non responsive test program i.e. hung, frozen, etc. Look at the upper border of MsDev to determine the execution state of the Host process. To restart (i.e. not single step) press the **F5** key:

Stopped at a breakpoint



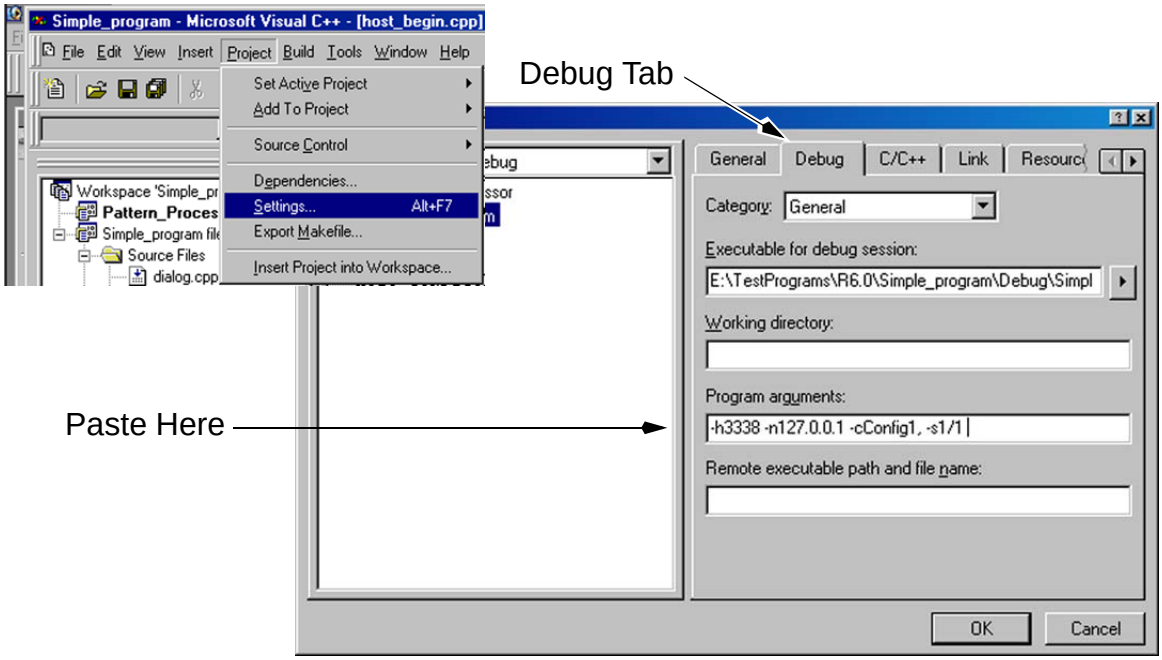
Not stopped

6. The Site Debug set up process begins when the following dialog is displayed (sometimes it starts minimized or hidden behind other windows). This dialog is from UI, and is referring to a Site Debug Mode step. Note that it is very similar to that seen in step-1, but that the text copied to the clipboard is quite different:



   Read these instructions but ***DO NOT*** click OK yet. As before, some very useful information has been copied to the clipboard, for use in the following steps. To reduce confusion, you should complete the following steps before copying anything else to the clipboard.

7. Locate the instance of MsDev which is to be used to debug Site process code. Click on **`Project > Settings...`** and select the **`Debug`** tab.



In this dialog, click in the **Program arguments** window and invoke paste (**^v**). Confirm that the information noted in the previous dialog is correctly pasted into this window (as shown). Click **OK**.

8. Before proceeding, set any breakpoints in Site code as needed.

Note: as before, ***do not*** confuse breakpoints used when debugging source code with those set using the Breakpoint Monitor - they serve completely different purposes.

---

9. When ready to proceed select **Build > Start Debug -> Go**. Or, press the F5 key.
10. Then locate the dialog shown in step-6, and click **OK**. This completes the setup of the Site Debug steps. The test program will begin to load and execute in the Site process, which includes all code in `CONFIGURATION()`, `SITE_CONFIGURATION()`, and `SITE_BEGIN_BLOCK()`, and `INITIALIZATION_HOOK()`. When Start Testing is invoked the Sequence & Binning Table will execute which causes `TEST_BLOCK`s to execute, etc. Unloading the test program causes the `SITE_END_BLOCK()` to execute.

Note: if source debug breakpoints are set, Site code execution may stop at one of these breakpoints. When this occurs, it can appear as a non responsive test program i.e. hung, frozen, etc. Look at the upper border of MsDev to determine the execution state of the Site process. To restart (i.e. not single step) press the **F5** key:

Stopped at a breakpoint

Not stopped

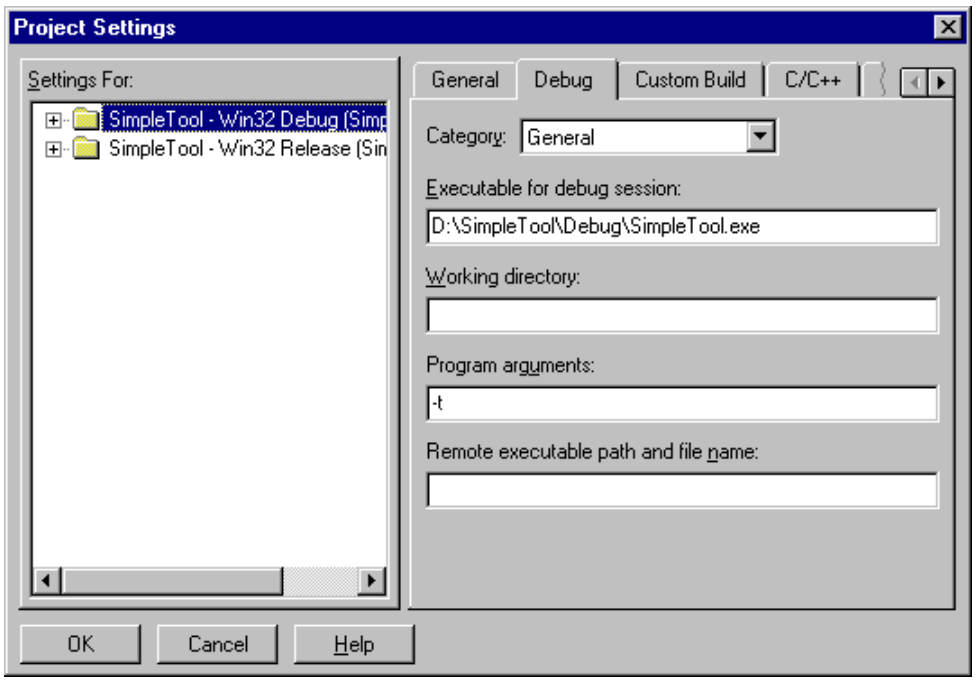
11. This completes the setup of both Host and Site Debug Modes. MsDev has extensive debug capabilities which are not documented here, including:
    - Setting and managing breakpoints
    - Stepping through source code
    - Examining and modifying variable values
    - Skipping code execution
    - etc.
12. Unloading the test program will terminate the debug session for the test program currently loaded. Note however that the debug modes set when UI was started remain in effect until UI is terminated.

6.1.14.1 User Tool Debug

These steps are performed prior to starting the User Tools. The method documented here to invoke the user tool can be used even in situations where the tool would normally be started from a User Menus in UI.

Debugging User Tool code applies to code executing in the tool process, but not to code which is executed in a Site or Host process.

1. Locate the instance of MsDev which is to be used to debug Tool process code. Click on **`Project > Settings...`** and select the **`Debug`** tab.



   In this dialog, click in the **`Program arguments`** window and invoke type **`-t`**. Click **`OK`**.

2. Before proceeding, set any breakpoints in Tool code as needed. The basic method is to click on the desired line of source code and press the **`F9`** key to insert a breakpoint. Read the MsDev documentation for details on using and managing breakpoints.

3. When ready to proceed select **`Build > Start Debug -> Go`**, or press the F5 key.

4. The user tool code will load and execute in the tool process, which includes all code in `CONFIGURATION()`, `TOOL_CONFIGURATION()`, `TOOL_BEGIN_BLOCK()`, and `INITIALIZATION_HOOK()`.

Note: if source debug breakpoints are set, Tool code execution may stop at one of these breakpoints. When this ocurs, it may appear as a non responsive tool i.e. hung, frozen, etc. Look at the upper border of MsDev to determine the execution state of the tool process. To restart (i.e. not single step) press the **F5** key:

Stopped at a breakpoint

Not stopped

6.2 UI Tool Persistence

See Interactive Tools.

Note: this release continues adding support for persistent attributes for the various UI tools. The initial release only supported WafermapTool. Support for additional tools are added in this release and additional tools will be added in future releases.

When UI is terminated (normally) the following attributes may be recorded (made persistent) for each UI tool which was started in the current session. Later, when UI and these tools are restarted, the tool will be automatically configured with these recorded values. This is commonly known as *tool attribute persistence*:

Tool	Saved Attributes
BitmapTool[1]	Location, Advanced State
BreakpointTool[1]	Location, Display X/Y Size
DBMTool[1]	Location, Display X/Y Size
ECRTool[1]	Location, Display X/Y Size
PatternTool[1]	Location
SearchTool[1] ShmooTool[1]	Location (these are the same tool)
LVMTool[1]	Location, Display X/Y Size
Voltage/Current Tool[1]	Location, Display X/Y Size
TimingTool[1]	Location, Display X/Y Size

Tool	Saved Attributes
UserVariableTool[1]	Location, Display X/Y Size
WafermapTool[2]	Location, Display X/Y Size, Display Mode, Cross-hair and Marked-Die mode, Zoom factor.
Note-1: the persistent attributes for these tools are recorded in the *Bin/ui.ini* file. Note-2: the persistent attributes for these tools are recorded in the *Bin/uiconfig/*.ini file (more below). Other tools will be added in the future.	

UI's tool display paradigm is somewhat different than other Windows applications. UI's tools will always be in one of the following states:

- Not started: the tool process is not running
- Started: the tool process is running. The tool is in one of the following states:
  - Tool is visible
  - Tool is visible but minimized; it is seen in the taskbar
  - Tool is hidden; i.e. not displayed and not seen in the taskbar

In normal use, the latter state is entered by first starting the tool and then clicking the cancel button (the  in the upper-right corner of the tool). As indicated, this does not terminate the tool process. Instead, it hides the display; the tool does not appear in the task bar nor on the display. Starting the tool again makes it visible again. This operation allows the tool to be hidden without losing any information.

When UI is terminated, the persistence facility notes which tool(s) are running and may record the attributes noted above. This will occur even if the tool is hidden. It does not occur for any tool which does not have a currently running process.

Note that a given tool's hide/show state is not persistent.

When UI is terminated (normally) the persistence facility records a given tool's persistent attributes (see table above). At this time, some tools use the legacy method for recording persistence (*Bin/ui.ini* file, see Note-1 in the table above) but newer tools use a more versatile method (see Note-2 in the table above) which uses two possible file locations:

1. The current test program's *Debug\uiconfig****.ini* file, if the file exists and is not read-only.
2. The current Nextest software release's *Bin\uiconfig****.ini* file, if the file is not read-only and the previous file does not exist.

Note the following:

- The *Debug\uiconfig* and *Bin\uiconfig* folders and *.ini* files are new, more below.
- Only one file is accessed, per UI tool.
- The order the files are listed above is the order in which the persistence facility looks for an *.ini* file. The first file located is the file accessed.
- The file order shown above is the order used to save persistent attributes AND to configure a given UI tool when it is first started.
- The first option above is only used if the specified *uiconfig* folder AND appropriately named ****.ini* file exists. This folder and these *.ini* files are never created by the Nextest software. The folder\file of the second option are initially configured by the software installation (more below).
- Using either option, a tool's ****.ini* file is actually named after the tool:
  - *WafermapTool.ini*

  Note that this list will get larger in future software releases, see Note:.
- To enable the first option above, the user must create both the *uiconfig* folder and the desired *.ini* file(s) in that folder. The folder name and the *.ini* file name are not case sensitive. However, it is recommended that the initial file(s) be copied from the software release's *Bin\uiconfig* folder, to ensure a valid file is created.
- Using the first option, the test program's owner has control over the write permission for each ****.ini* file. Using the second option (i.e. the (*SWrelease\Bin\uiconfig****.ini)* only the system administrator can change an *.ini* file's write permission. In all cases, if a given *.ini* file is set to read-only that file will not be updated when UI terminates and no warning is issued. This mechanism can be used to prevent each user from modifying any of the persistent attribute values recorded in a given *.ini* file.
- Regardless of which method is used, when UI is terminated normally, if a given tool's *.ini* file is not read-only it will be updated with each running tool's current persistent attribute values.
- The *ui.ini* file, located in each Nextest software release's *Bin* folder, will continue to record persistent attributes for some tools, until all are migrated to the new methods. It also continues to record UI's window size and location, site IP addresses, etc.
- It is expected that some tools (most?) will use the second option (i.e. the *SWrelease\Bin\uiconfig****.ini)* and a few selected tools will use the first option. The second option should be used for tools which are to be displayed the same, regardless of which test program is in use. The first option is useful when a given

tool's initial configuration needs to change based on the test program configuration. For example, different die or wafer sizes when using WafermapTool, different pin usage when using FrontPanelTool, etc.

- Waveftool's (MSWT) persistence file (*mswt.ini*) was in use before this unified tool persistence facility was invented. The existing paradigm will continue to operate as previously designed. However, if the existing *mswt.ini* file is deleted the new methods will be utilized.

Beginning with this release, as a new Nextest software release is installed (*UseRel*) it will perform the following actions related to the persistence facility:

- Add the *uiconfig*\ folder to the release's *Bin*\ folder.
- Add the appropriate tool ****.ini* file to this *uiconfig*\ folder, one for each of the tools noted above. Each file's content will be the tool's default attribute values.
- As in all Nextest software installations, the user (installer) will then be prompted whether the *ui.ini* file from the current Nextest software release should be copied to the release being installed, to replace the default file. If the user responds YES the *ui.ini* file from the previous release is copied to the *Bin*\ folder, over-writing the default file.
- In the future, the previous step will also copy the entire *Bin**uiconfig*\ folder, and all of the *.ini* files in that folder, from the existing release to the new release, thus copying any per-tool *.ini* file(s) which exist in the previous release.

6.3 BitmapTool

BitmapTool allows the user to analyze memory device failures by visually displaying an accurate representation of the topology of the device's passing/failing data bits on the computer monitor.

Note: the optional Error Catch RAM (ECR) must be installed in each site controller for BitmapTool to function - all pass/fail information and APG X/Y address information is read from the ECR for display in BitmapTool .

By default, BitmapTool does not modify the ECR information before generating the display. This normally results in displaying a *logical* view of the DUT (see Logical vs. Physical, vs. Electrical Addresses). To obtain a *physical* view additional software must be written to define a Bitmap Schemes which describes the desired physical topology.

The BitmapTool example display below shows failures from 2 ECRs (2 sites)..

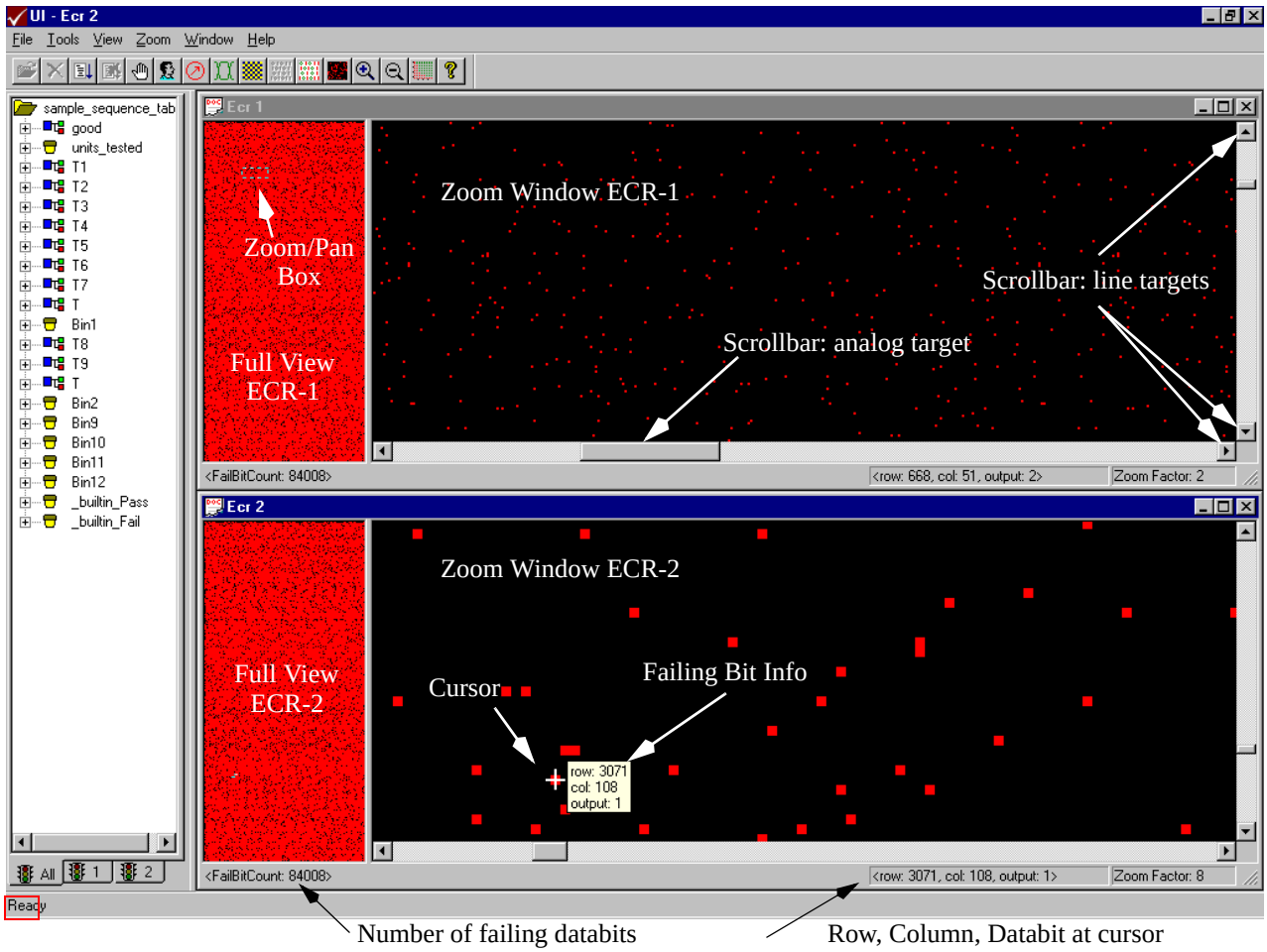


Figure-71: BitmapTool Display

Each ECR has its own display window, which is divided into two resizable panes. The left pane shows a whole device view. The right pane is a scrollable zoomed view of the zoom/pan box seen in the whole device view.

The Bitmap uses color to represent various states:

- Red = failed databits
- Black = passed databits
- Grey represents view area outside the device boundary.

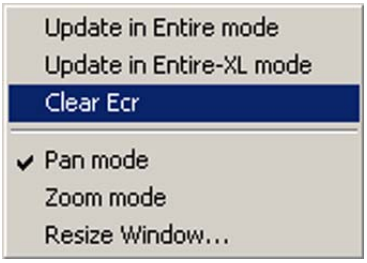
The colors used for pass/fail can be changed. See BitmapTool Control Dialog.

The zoom/pan box can be used to either change the zoom factor or to pan the display. See BitmapTool Control Dialog for controls. In either case, the right portion of the screen is the zoom window, which represents the contents of the zoom/pan box.

The following controls are enabled by click-and-hold the right-mouse button within the full view window:

Changes the mode of the zoom/pan box. When Pan mode is selected the zoom/pan box can only be moved, which causes the view in the zoom window to pan accordingly. When Zoom mode is selected only the size of the zoom/pan box can be changed which then sets the zoom factor. Also see `BitmapTool Control Dialog` and `ui_BitmapPan`.

Controls enabled by click-and-hold the right-mouse button within the full view window:

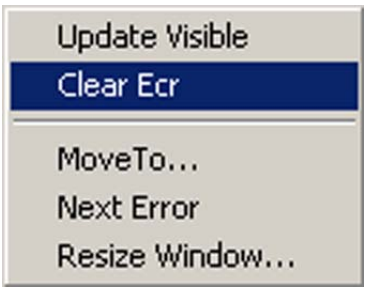


Select and release the **`Update in Entire mode`** option causes the entire contents of the ECR to be re-displayed in BitmapTool.

Select and release the **`Update in Entire XL mode`** option causes selected contents of the ECR to be re-displayed in BitmapTool. See BitmapTool Display Mode.

Select and release the **`Clear ECR`** control to clear the ECR. This does not cause BitmapTool to update its display.

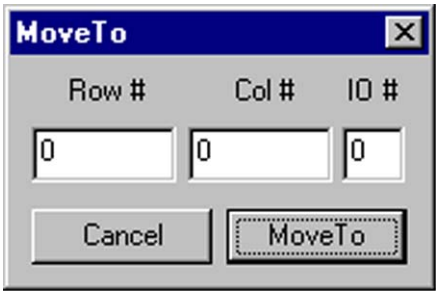
Controls enabled by click-and-hold the right-mouse button within the zoom window:



Select and release the **`Update Visible`** option causes the selected contents of the ECR to be re-displayed in BitmapTool. See BitmapTool Display Mode.

Select and release the **`Clear ECR`** control to clear the ECR. This does not cause BitmapTool to update its display.

Select and release the **`MoveTo...`** option causes the following dialog to be displayed:



The **MoveTo** dialog provides a method for moving (scrolling) the zoom window to a specific row, column, and databit. In general terms, the zoom/pan box will be moved such that its upper/left corner is located at the specified row, column, and I/O (databit), provided that the bottom and right borders of the box have room to move within the display. The zoom window tracks the zoom/pan box. In other words, the display will only move to the extent that the zoom/pan box can move; once the box reaches the right or bottom display limits no further movement will occur. The position of a given row, column, and databit in the display is determined by the currently selected Bitmap Schemes.

Select and release The **Next Error** option causes zoom/pan box to move to the next failing databit in the display. As noted for **MoveTo**, this occurs only to the extent that the zoom/pan box has room to move.

At the bottom/left of each ECR window, the status bar displays the total failed ***bit*** count (*not* the failed *address* count).

At the bottom/right of each ECR window the status bar always displays row, column, and databit corresponding to the cursor position in the zoom window. This information is displayed for both passing and failing bits. As the cursor is moved to a failed bit, a popup window displays the corresponding device row, column, and output information. Both sets of values are affected by the currently selected Bitmap Schemes. In addition, user-written code can affect the row, column, and data values displayed in the status bar, as the cursor is moved in BitmapTool. See BitmapTool Callback Macros.

6.3.1 ECR Setup

The ECR must be set up from the test program before BitmapTool can be used from UI. See Error Catch RAM (ECR) for details.

In Memory Test Patterns applications the ECR ***accumulates*** failure information, even across `funtest()` executions. Therefore, it is necessary to explicitly clear the ECR as needed. Then, executing `funtest()` using the `fullec` argument will cause the ECR to capture failure information. For example:

```
clear( my_ecr );
BOOL status = funtest( pattern, fullec )
```

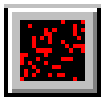
Note that there are other argument options to `funtest()` which affect the use of the ECR. See Pattern Execution Stop Condition Options for details.

6.3.2 Invoking BitmapTool

BitmapTool is typically invoked:

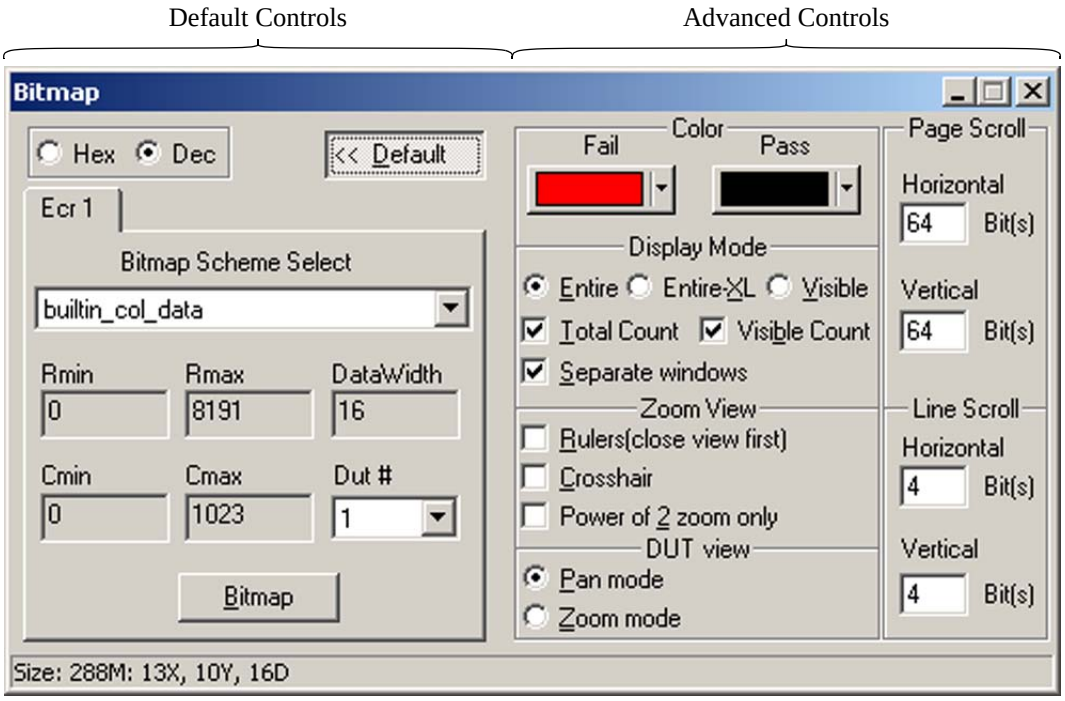
- At a UI *Breakpoint* after a functional test (see Breakpoint Monitor)
- From a *Breakpoint* after a Test Blocks
- After an execution of the Sequence & Binning Table.

To start BitmapTool:

- Click on the BitmapTool icon from the UI toolbar 
- Type keyboard shortcut **`Ctrl+B`**
- Choose **`Tools: BitmapTool…`**
- From user C-code. See BitmapTool UI Variables

6.3.3 BitmapTool Control Dialog

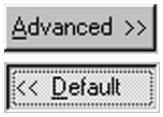
The BitmapTool control panel appears when BitmapTool is invoked from UI's tool bar:



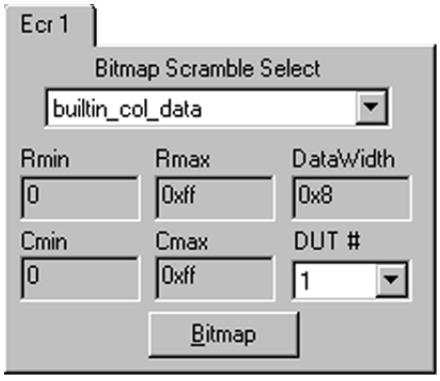
Note: dialog shown was using software release v.2.6.3

The controls in this dialog are described below:

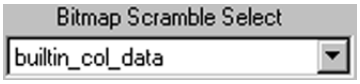
`Hex` / **`Decimal`** selection. This determines whether various integer values displayed in both BitmapTool and the control dialog use a hexadecimal or decimal radix. Also see `ui_BitmapDialogDecMode`.



`Default` / **`Advanced`** selection. Determines whether the advanced options are displayed in the control panel. The **`Advanced`** button is displayed when the default control panel is displayed; clicking it causes the advanced options to be displayed.



ECR selection tab, and associated controls.
When using a single site system only one tab will be displayed (as shown = ECR 1). When using a multi-site system multiple tabs will be displayed allowing selection of the ECR from one site. The other controls in this display then only affect information displayed from that site.



Bitmap Scramble Select pull-down menu. This control is used to select one Bitmap Schemes, which then determines how ECR information is translated (scrambled) before being displayed in BitmapTool. The built-in schemes provide for displaying a logical view of the Bitmap data (see Logical vs. Physical, vs. Electrical Addresses). User-defined Bitmap schemes, documented in Bitmap Schemes, can be used to create a physical view of the Bitmap data. The default built-in `builtin_col_data` scheme named is shown.

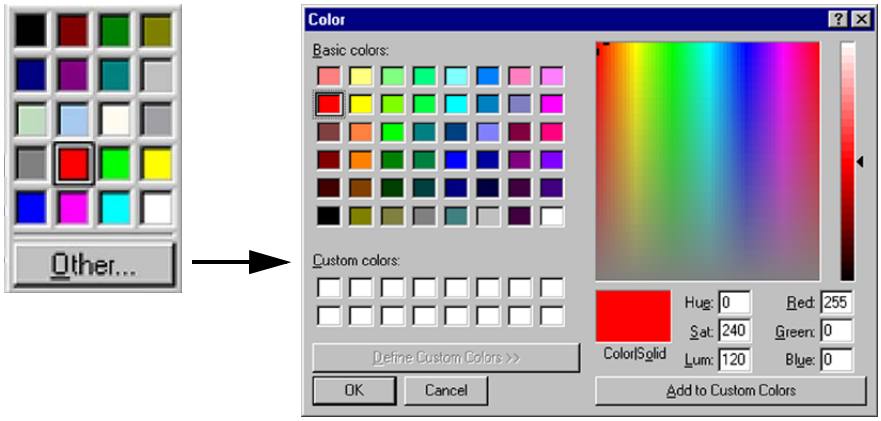
These are read-only values indicating the minimum/maximum ECR rows and columns affecting the display, and the number of databits displayed at each display row/column. Note that if the currently selected Bitmap Schemes does not display a contiguous series of rows and columns that the amount of data displayed in BitmapTool may not correlate with calculations performed using these values.

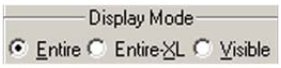
DUT number selection pull-down menu. This control allows selection of which DUT will affect the information displayed in BitmapTool. In Multi-DUT Test Programs, the number of DUTs selectable is determined by the number of DUTs defined in the Pin Assignment Table. If > 1, failures for only the selected DUT will be displayed in BitmapTool. Also see `ui_BitmapdutNo`.

`Bitmap` update button. This button updates the Bitmap display. See BitmapTool Display Mode for deails of different update mode options. Also see `ui_BitmapDisplay`.

The following controls appear in the Advanced section of the dialog, when the **Advanced** button is selected.

Color selection controls. These controls can be used to change the default colors used to display passing and failing databits in BitmapTool. Due consideration must be given to color interactions when using Bitmap Overlays (see Bitmap Overlay Colors). Clicking on either of these controls will display the basic color palette, shown on the left below. Clicking on the Other button in that dialog will display the advanced color palette shown on the right below.





Controls several display update related features. See BitmapTool Display Mode.

Controls whether the total failed bit count and visible failed bit count values are collected and displayed. See Fail Count Enable Controls.

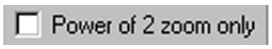
Controls whether the two BitmapTool display windows are connected or displayed separately. See BitmapTool Separate Window Option.

Controls which affect only the right window in BitmapTool (the zoomed view), or only the left window (the full DUT view).

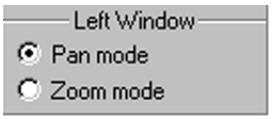
Enable or disable the display of rulers in the zoom window (only) (see Example BitmapTool Display with Overlays). To take effect BitmapTool must be closed and restarted. Rulers allow more exact positioning of the cursor within the window. Ruler scale values represent a count of databits

in each axis. The data association (with columns or rows) as specified by the currently selected Bitmap Schemes will affect one axis. Also see `ui_BitmapRulers`.

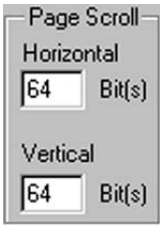
Enable or disable cross hairs in the zoom window (only). Cross hairs allow more exact positioning of the cursor within the window. Also see `ui_BitmapCrossHair`.



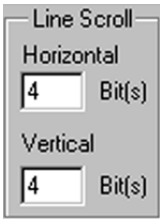
Enable or disable the *Power of 2 zoom* mode. When enabled, the zoom factor can only be changed as a power of 2. When disabled the zoom factor is an analog value.Also see `ui_BitmapZoom2`.



Set the mode of the zoom/pan box in the left window (full DUT view). When Pan mode is selected the zoom/pan box can only be moved, which causes the view in the zoom window to pan accordingly. When Zoom mode is selected only the size of the zoom/pan box can be changed which then determines the zoom factor. Also see `ui_BitmapPan`.



Used to specify the amount the zoom window will scroll when the left mouse is clicked in the page area of a scroll bar. The value is in databits, which are displayed using a different number of pixels depending on the current zoom factor. Default values are shown. Also see `ui_BitmapPageHScroll` and `ui_BitmapPageVScroll`.



Used to specify the amount the zoom window will scroll when the left mouse is clicked on either line scroll target of a scroll bar. The value is in databits, which are displayed using a different number of pixels depending on the current zoom factor. Default values are shown. Also see `ui_BitmapLineHScroll` and `ui_BitmapLineVScroll`.

6.3.3.1 BitmapTool Display Mode

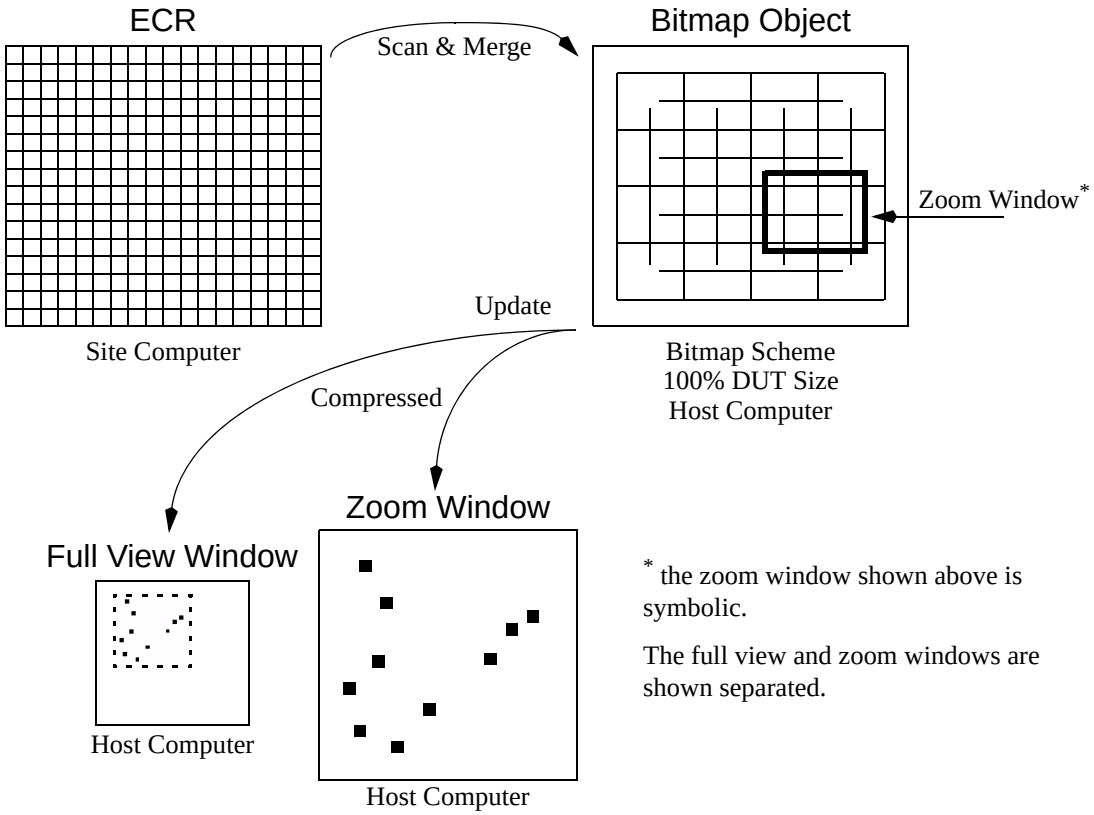
In general, the BitmapTool display mode determines how failure information is read from the ECR, processed, and displayed in BitmapTool. The following options are provided:

Entire Mode

Entire XL Mode

Visible Mode

The following model is used to describe the three modes:



Note the following:

1. In all update modes, fail data is read (scanned) from the ECR and transferred from the site computer to a software bitmap object on the host computer. The bitmap object represents fail data for 1 DUT from 1 ECR, massaged into 1 bitmap scheme.
2. The bitmap object is populated differently based on the selected update mode. This affects update performance and the completeness of the data in the bitmap object (more below).
3. Information from the bitmap object is used to paint the two bitmap displays.
4. In the full view window, for most memory devices, the video display isn't large enough to display a unique pixel for each databit. Thus, the bitmap software must determine which bits from the bitmap object are compressed into a given display pixel. If any one or more of those bits = FAIL the pixel is shown as a fail. This is called display compression.
5. In Entire mode, the contents of the ECR for an entire DUT is scanned, and all failures are merged into the bitmap object, organized for the currently selected Bitmap Schemes. .

This mode ensures that the full view window and the zoom window always display a complete and current view of the fail data, including the Total and Visible count values if enabled (see BitmapTool Visible Fail Count Display). Since the bitmap object is complete, scrolling or panning the zoom window does not require accessing the ECR for additional fail data.

6. In Visible mode, only the area of the ECR represented by the zoom window is scanned, thus only failures from that area are merged into the bitmap object. This improves update performance because less processing is involved and less data is handled. Both the full view window and the zoom window are updated but only the failures in the zoom/pan box are updated in the full view window. Since the bitmap object is incomplete, scrolling or panning the zoom window does require accessing the ECR for additional fail data.

Note: using Visible Mode mode and Entire XL Mode, when the ECR contents change it is HIGHLY recommended that the Bitmap button be clicked, to update both the full view and zoom windows, and the Total and Visible fail data bit count(s).

---

7. In Entire XL Mode the entire full view window and the zoom window are updated, but the ECR scan algorithm operates differently than that Entire mode. In simple terms, a map is made of which ECR bits can affect a given full view window pixel. Using this map, the ECR scan software can stop as soon as the first failure affecting a given pixel is identified. This has several effects:

    - Bitmap update performance is improved when the size of the full view window is relatively small, the DUT is large, and has many failing bits. Entire mode may be faster if failures are sparse, and when a very complex bitmap scheme is in use.
    - The creation of the map consumes time which is not required in Entire or Visible mode. However, the map is cached, and only needs to be updated when the full view window is resized, the zoom window is panned/or scrolled, or when the pan/scroll box is changed in the full view window.
    - For pixels outside the zoom window, only the first fail detected per-pixel is merged into the bitmap object. This means the bitmap object may not contain every failure captured in the ECR. This is important as noted blow. For pixels within the zoom window, all failures are merged from the ECR to the bitmap object.
    - Because bitmap object may be incomplete, if the zoom/pan box is changed in the full view window, or if the zoom window is scrolled, the ECR must be partially scanned again, to obtain a complete picture of failures to be displayed in the

zoom window. And, when a partial scan is performed, if the ECR contents have changed since the previous scan, the resulting display will be invalid (and the user will not know it).

- Because the bitmap object may be incomplete, to obtain an accurate Total fail count (see Fail Count Enable Controls) requires additional time, which reduces the benefit of Entire XL Mode, .
- The `ui_BitmapDisplayMode` user variable (see UI User Variables) can be used to programmatically set this mode. It must be used prior to starting BitmapTool.
- The `ui_BitmapVisibleSize` user variable is available to programmatically set the size of the zoom window. It must be used prior to starting BitmapTool.

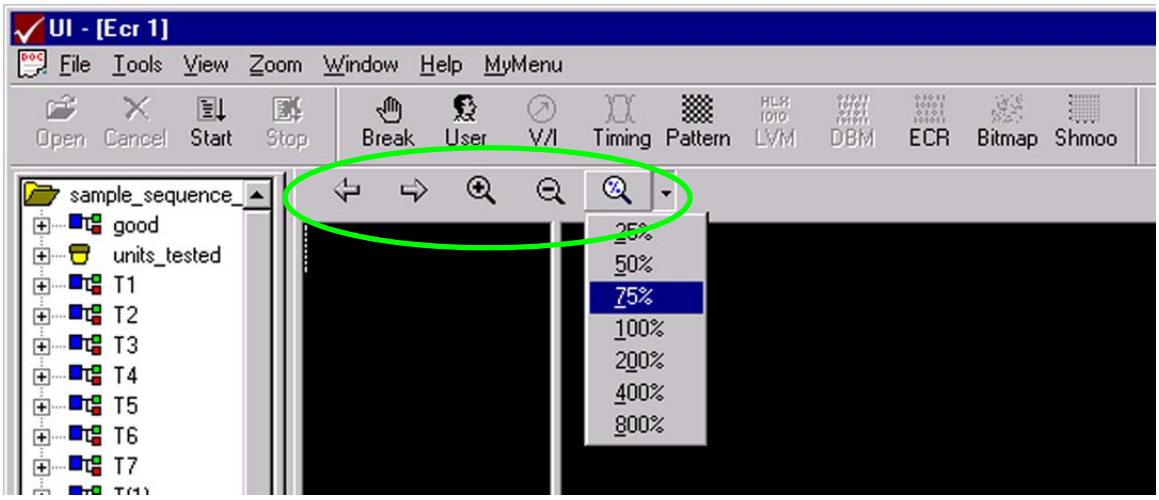
Note: the BitmapTool Display Mode controls and `ui_BitmapDisplayMode` set a static display mode, whereas using the right mouse in the BitmapTool zoom window is a transient event.

8. As noted above, the time spent scanning the ECR to obtain the total fail bit count (displayed in the bottom edge of the BitmapTool display) can be substantial. It is possible to disable this, using the Fail Count Enable Controls. It is also possible for user code to determine the count value displayed (see `ui_BitmapTotalFailBitCount`) or the entire string displayed (see `ui_BitmapTotalVisibleFailBitString`, `ui_BitmapTotalFailBitString`, or `ui_BitmapVisibleFailBitString`).

6.3.4 BitmapTool Zoom Controls

BitmapTool has the following controls used to afffect the display Zoom settings:

- BitmapTool Zoom selections::



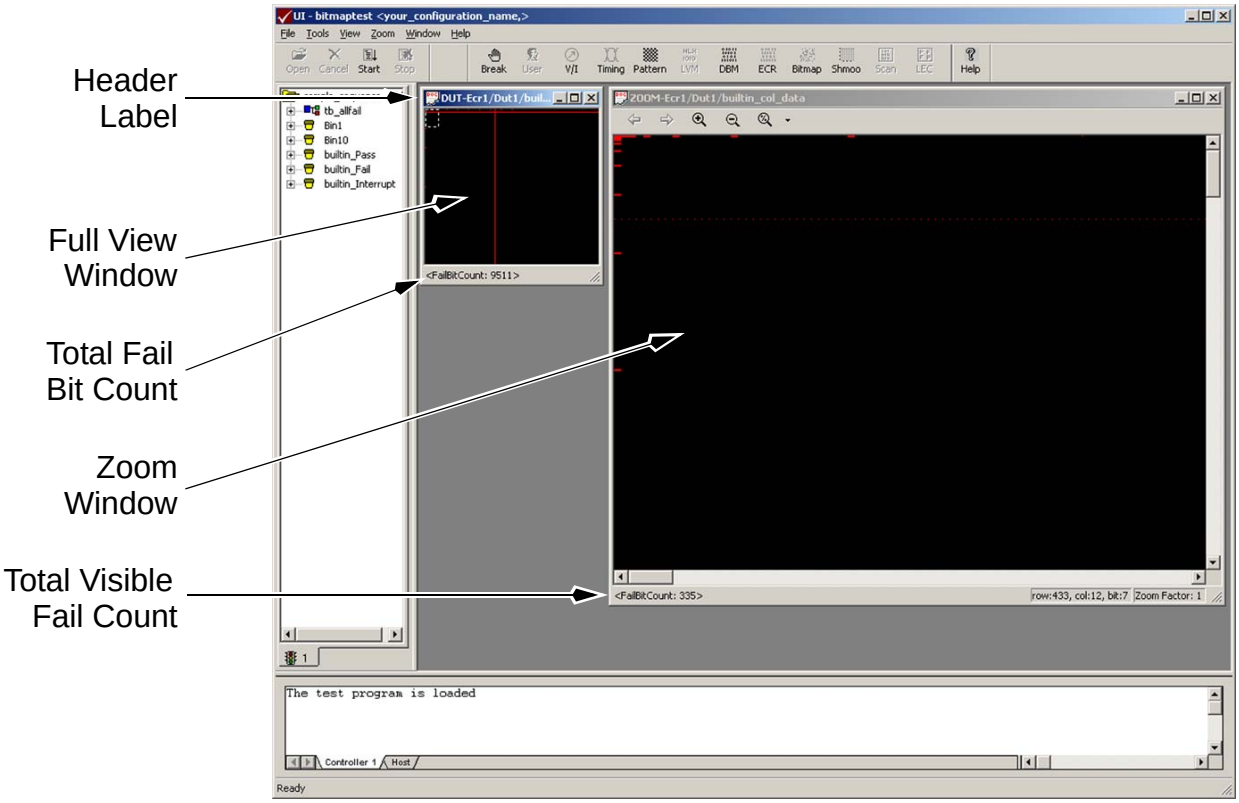
- The plus and minus buttons zoom in/out one increment.
- The right button can be used two ways:

  - Left-click to unconditionally return to 100% view.

  - Right mouse to display the menu shown above.
- The left arrow is used to zoom to the previous zoom value. The right arrow to zoom back. These arrows are initially disabled, until a zoom-in or zoom-out is performed. The arrows may be disabled when the BitmapTool is resized or anytime either scroll bar is removed from BitmapTool (which occurs when it is resized). The arrows are enabled again when a zoom-in or zoom-out is performed.
- The `ui_BitmapVisibleSize` user variable is available to programmatically set the size of the zoom window. It must be used prior to starting BitmapTool.

6.3.5 BitmapTool Separate Window Option

BitmapTool has two graphic display windows, the full view window and the zoom window. These can now be resized, positioned (within UI), and updated independently. The image below shows separate full view and zoom windows:



Note the following:

- By default, BitmapTool will start in the original configuration, with the full view and zoom windows tiled into a single GUI display. To separate the windows, in the BitmapTool Control Dialog select the Separate Zoom view check box. Then, terminate and restart BitmapTool. The selection is persistent, saved in the *UI.ini* file.
- BitmapTool windows which are already open are not affected when the Separate Zoom view check box is modified.
- When Separate Zoom view is enabled, a new set of windows will be created any time one of the following is changed and the Bitmap button is clicked:
  - ECR tab selection

- DUT #
- Bitmap scheme selection

This allows multiple sets of windows to be displayed simultaneously.

- When the Bitmap button is clicked only those window(s) which match the currently selected ECR/DUT/Bitmap Scheme are updated.
- The header title of each window will include related information. For example:

- As before, resizing, panning, or scrolling the zoom window, or moving the zoom/pan box in the full view window, will re-read the pertinent sections of the ECR and update the failures displayed in the zoom window, the failures displayed in the zoom/pan box, and the Visible fail count value (see BitmapTool Visible Fail Count Display).
- When Separate Zoom view is enabled, the visible fail count string can be modified using `ui_BitmapVisibleFailBitString` and the total fail count string can be modified using `ui_BitmapTotalFailBitString`. When Separate Zoom view is disabled, the total/visible fail count string can be modified using `ui_BitmapTotalVisibleFailBitString`.
- When Separate Zoom view is enabled and the Zoom window is closed, the zoom/pan box normally seen in the full view window disappears.
- When Separate Zoom view is disabled, both Total fail count and Visible fail count values are displayed in the same location, at the bottom left of the window border. When Separate Zoom view is enabled, the full view window will only display the Total fail count, and the zoom window will only display the Visible fail count. See BitmapTool Visible Fail Count Display and Fail Count Enable Controls.
- When Separate Zoom view is enabled, if either or both windows are closed, and the Bitmap button is clicked, both windows will be re-displayed. To update a single window without updating (or opening) the other window, use the right mouse button to select Update in the desired window. Note that the full view window has two update options: Entire Mode and Entire XL Mode display modes (see BitmapTool Display Mode).
- When Separate Zoom view is enabled, the size of the full view window directly affects the update performance in Entire XL Mode. Similarly, the size of the zoom window affects performance of Visible Mode and Entire XL Mode (although not as much as the size of the full view window). This also applies when the window display is combined, but since resizing affects both the full view and zoom windows it is less useful as a performance improvement technique.

6.3.6 BitmapTool Visible Fail Count Display

Two fail count values are displayed simultaneously:

- Total = all fails
- Visible = fails bounded by the zoom window

Windows Combined

Separate Main Window = total

Separate Zoom Window = visible

Note the following:

- Total count is displayed when enabled via the Fail Count Enable Controls. Total count is accurate when the display is updated in either Entire Mode or Entire XL Mode. Conversely, when the display is updated in Visible Mode mode, the Total count may not be correct, depending on whether the contents of the ECR have changed since the last time the main window was updated in Entire Mode or Entire XL Mode.
- Visible count is displayed when enabled via the Fail Count Enable Controls. Visible count is always accurate.
- Resizing, panning, or scrolling the zoom window will re-read the pertinent sections of the ECR and update the Visible count but not the Total count. This is important when the contents of the ECR have changed since the last time the Total count was updated i.e. the value may be stale.

6.3.7 Fail Count Enable Controls

The BitmapTool Control Dialog has added controls which can disable the *collection* and display of Total and Visible fail count values:

More important than the display aspect is the performance improvement obtained when the count values are not collected. In particular, disabling the Total fail count option will

noticeably reduce display update time in Entire Mode and Entire XL Mode. Disabling Visible fail count has less impact on update performance.

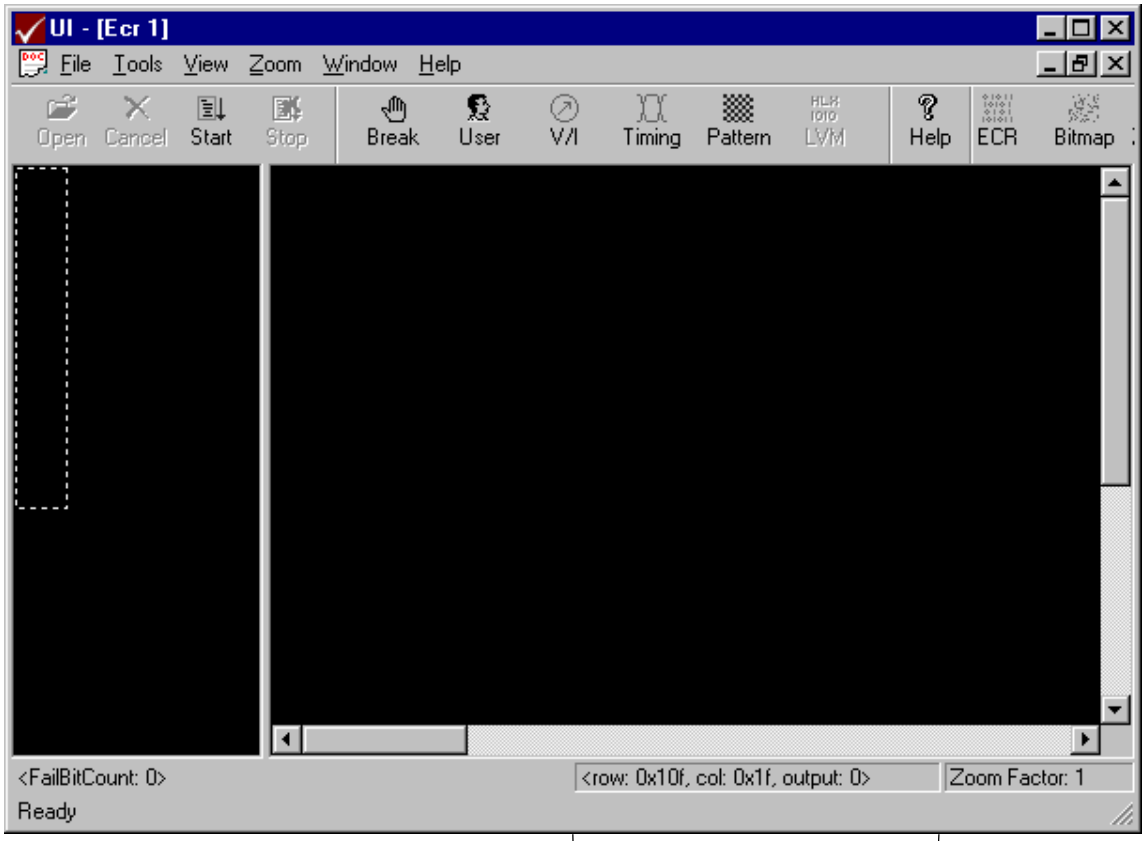
When a given count is disabled the associated FailBitCount value is not displayed in the BitmapTool display.

6.3.8 BitmapTool Callback Macros

The `TRANSLATE_BITMAP_INFO()`, `TRANSLATE_BITMAP_INFO_5()` and `TRANSLATE_BITMAP_INFO_7()` macros are BitmapTool call-back macros are available to:

- Modify the row/column/data coordinate values displayed in BitmapTool corresponding to the mouse position in BitmapTool zoom window.
- Capture these row/column/data value(s) when the mouse is left clicked in the BitmapTool zoom window.

The default row, column and data coordinate values are seen below, in the lower right corner of the BitmapTool display:



Default row, Column, and Data coordinates corresponding to mouse position in zoom window

The BitmapTool call-back macros are tied to mouse cursor movement, and left-button mouse clicks in the BitmapTool zoom window. If either macro is defined, the macro body code executes automatically as the cursor is moved, or the left mouse button is clicked in the BitmapTool zoom display area. Code within the macro body can be used as noted above.

Only one of these macros should be defined in any given test program. They must be declared at a global level i.e. not within a function or within the body code of another macro.

A new call-back event is not sent to a site (to execute the macro body code) until execution of the previous event was complete. This has the effect of discarding those events which occur during that time. This improves performance, prevents overwhelming the call-back, and makes mouse-to-site synchronization reliable, at the expense of missing some events when the mouse is moved quickly.

Usage

```
TRANSLATE_BITMAP_INFO( DWORD row,
                       DWORD col,
                       DWORD data,
                       DWORD clicked ) {}
TRANSLATE_BITMAP_INFO_5( DWORD row,
                         DWORD col,
                         DWORD data,
                         DWORD clicked,
                         CString message) {}
TRANSLATE_BITMAP_INFO_7( DWORD row,
                         DWORD col,
                         DWORD databit,
                         DWORD clicked,
                         CString message,
                         DWORD bitmap_row,
                         DWORD bitmap_col )
```

where:

TRANSLATE_BITMAP_INFO is a Test System Macro used to denote the start of the code block.

TRANSLATE_BITMAP_INFO_5 is similar to **TRANSLATE_BITMAP_INFO** except that it requires 5 arguments.

TRANSLATE_BITMAP_INFO_7 is similar to **TRANSLATE_BITMAP_INFO** except that it requires 7 arguments.

row, **column**, and **data** are variables used within the macro body code to represent the row/column/data coordinate values corresponding to the mouse position in the BitmapTool zoom display area. The values actually displayed in BitmapTool are the final values assigned to these variables within the macro body code.

clicked is TRUE any time the left mouse button is clicked in the BitmapTool zoom display.

message allows the entire value displayed in BitmapTool to be defined within the macro body code. When not used, the structure of the displayed message is the default row, column, and data values. When **message** is used it completely replaces the display values.

bitmap_row and **bitmap_col** are read-only and represent the cursor position within the BitmapTool zoom display area. Values are in screen-space context with the upper left corner

= 0/0. Resolution is one databit which, depending on the current zoom factor, can represent one or many pixels.

None of the argument variables exist outside the scope of the macro body code. The user does determine the name of each variable, but no other special declarations are required.

Examples

Example 1:

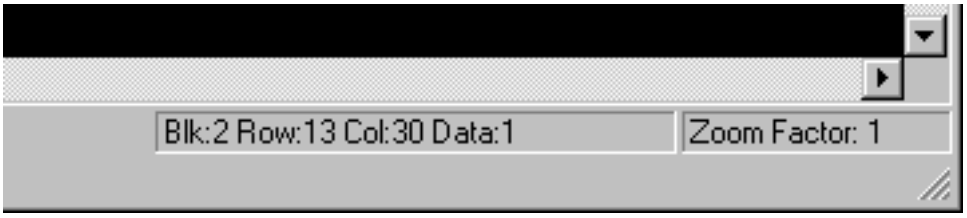
The following example alters the row value displayed in BitmapTool. The column and data values are not changed. When the mouse is left clicked in the BitmapTool zoom area the coordinates are captured (saved) in user-defined variables:

```
int save_row_coord, save_col_coord, save_databit;
TRANSLATE_BITMAP_INFO( row_coord, col_coord, databit, clicked ) {
   // Any modifications done here to the DWORDS row_coord, and/or
   // col_coord, and/or databit will be reflected in the coordinate
   // values displayed by BitmapTool
   row_coord = 10;   // BitmapTool will only ever display 10 as the
                     // row coordinate. Not very useful, but does
                     // demonstrate the use of this macro.
   // Capture mouse coordinate values here when the mouse is clicked
   // in the BitmapTool display
   if (clicked) {
      save_row_coord = row_coord;
      save_col_coord = col_coord;
      save_databit = databit;
   }
}
```

Example 2:

This example completely replaces the default row/column/data values displayed in BitmapTool's. Instead, the user code below creates a *block number* using two bits of the row

address. Then, a new display value is created to include block, row, column, and data, as shown below:



See comments in the code below.

```
// BitCount() used in TRANSLATE_BITMAP_INFO_5() example below.
// Return a count of bits set in arg1 = mask.
// Only counts low contiguous bits in mask
int BitCount(int mask) {
   int count = 0;
   for (int index =0; mask != 0; ++index, mask >>= 1)
      if (mask & 1) ++count;
   return count;
}

// As cursor moves in BitmapTool generate a new row/col/data
// display. This example displays Blk 0..3 plus row/col/data.
// High 2 bits of row value determine the block address.
// BLOCK_MASK definition assumes numx(9).
// INV_BLOCK_MASK is inverse of BLOCK_MASK
// Calls BitCount() function.

#define BLOCK_MASK 0x180// Assumes numx(9)
#define INV_BLOCK_MASK (xmax() ^ BLOCK_MASK)

TRANSLATE_BITMAP_INFO_5( row, col, data, clicked, str ) {
   // Determine block ID = 0..3.
   DWORD blk = (row & BLOCK_MASK) >> BitCount(int INV_BLOCK_MASK);

   // Remaining row bits are displayed as row address
   row = (row & INV_BLOCK_MASK);

   // Create entire display string seen in BitmapTool.
   str = vFormat("Blk:%d Row:%d Col:%d Data:%d", blk,row,col,data);
}
```

6.3.9 BitmapTool UI Variables

The following built-in UI variables are provided for programatic control of BitmapTool. These are documented in detail in UI User Variables.

Variable Name	Purpose
ui_BitmapCrossHair	Set the initial crosshair display mode.
ui_BitmapDialogDecMode	Toggle between decimal and hexadecimal values for display (default is FALSE), -1 for Ui
ui_BitmapDisplay	Invoke or update the display if already running. No BitmapTool dialog appears for programmatic invocation of BitmapTool.
ui_BitmapDisplayMode	Set the initial display mode. See BitmapTool Display Mode.
ui_BitmapDisplaySeparateZoomWindow	Control whether the full view and zoom windows are displayed together or separately. See BitmapTool Separate Window Option.
ui_BitmapDisplayTotalCount	Enable or disable collection and display of total fail count value. See Fail Count Enable Controls and BitmapTool Visible Fail Count Display.
ui_BitmapDisplayVisibleCount	Enable or disable collection and display of visible fail count value. See Fail Count Enable Controls and BitmapTool Visible Fail Count Display.
ui_BitmapdutNo	Set which DUT# to bitmap(DUT0 is bitmapped by default).
ui_BitmapMainSize	Set the size of the full view window. See BitmapTool Separate Window Option.
ui_BitmapMaxErrors	Limits the number of failures displayed. Result is approximate, based on ui_BitmapRowsChunk. BitmapTool will stop displaying errors when the max error count is reached, but errors are read from the ECR in chunks.

Variable Name	Purpose
ui_BitmapMoveTo	Moves zoom/pan box to a specified row/ column/data position in the full view BitmapTool display. Zoom window tracks the zoom/pan box.
ui_BitmapPageHScroll, ui_BitmapPageVScroll, ui_BitmapLineHScroll, ui_BitmapLineVScroll	Set the initial values for Page and Line horizonal and vertical scroll size.
ui_BitmapPan	Set the state of the zoom/pan option.
ui_BitmapRowsChunk	Maximum number of rows read from the ECR in a single chunk. Default is 512 rows. Affects resolution of ui_BitmapMaxErrors.
ui_BitmapRulers	Set the initial ruler display mode.
ui_BitmapVisibleSize	Set the size of the zoom window. See BitmapTool Separate Window Option.
ui_BitmapZoom2	Set the initial state of the power of 2 zoom option.

6.3.10 Bitmap Schemes

This section includes the following:

- Overview
- Built-in Bitmap Schemes
- Bitmap Segment Positioning
- Bitmap Scheme Functions and Data Types
- bitmap_scheme Data Type
- make_bitmap_scheme()
- add_segment()
- register_bitmap_scheme()
- dump()
- permutation Data Type
- Permutation Memory Management

- make_permutation()
- reverse()
- rotate()
- swap()
- append()
- insert()
- set()
- for_each()
- filter()
- get()
- size()
- bitmap_scheme_translate()
- bitmap_scheme_lookup()

6.3.10.1 Overview

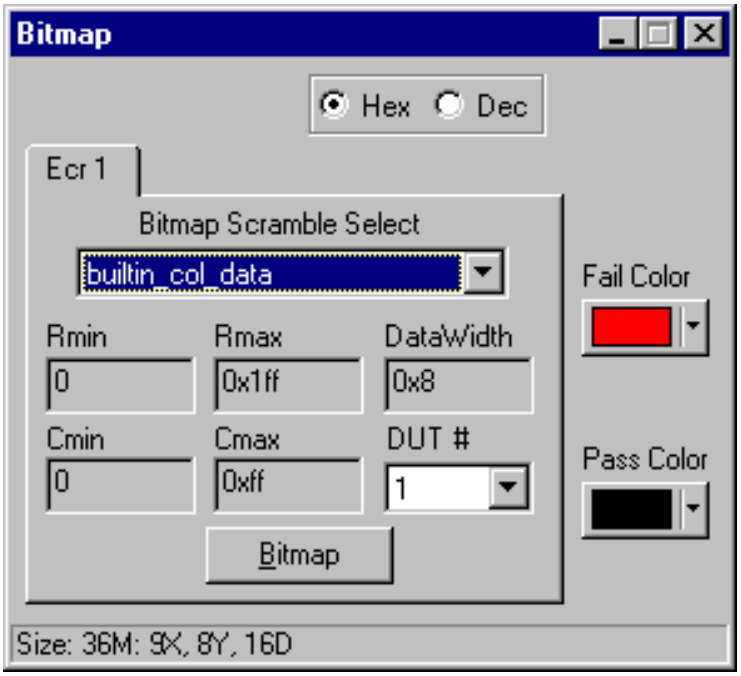
See Bitmap Schemes.

As noted earlier, by default, BitmapTool does not modify the ECR information before generating the bitmap display. The resulting display is a *logical* view of the DUT (see Logical vs. Physical, vs. Electrical Addresses).

To obtain a *physical* view, user code must define a *bitmap scheme*, which describes the desired physical topology. Creative users may find other applications for the bitmap scheme capability, to display different views of the same ECR failure information with each view defined by creating an additional bitmap scheme. It is possible to create any number of schemes.

Using BitmapTool, one bitmap scheme is selected from the **`Bitmap Scramble Select`** pull-down menu in the BitmapTool control dialog. By default, this menu displays the Built-in

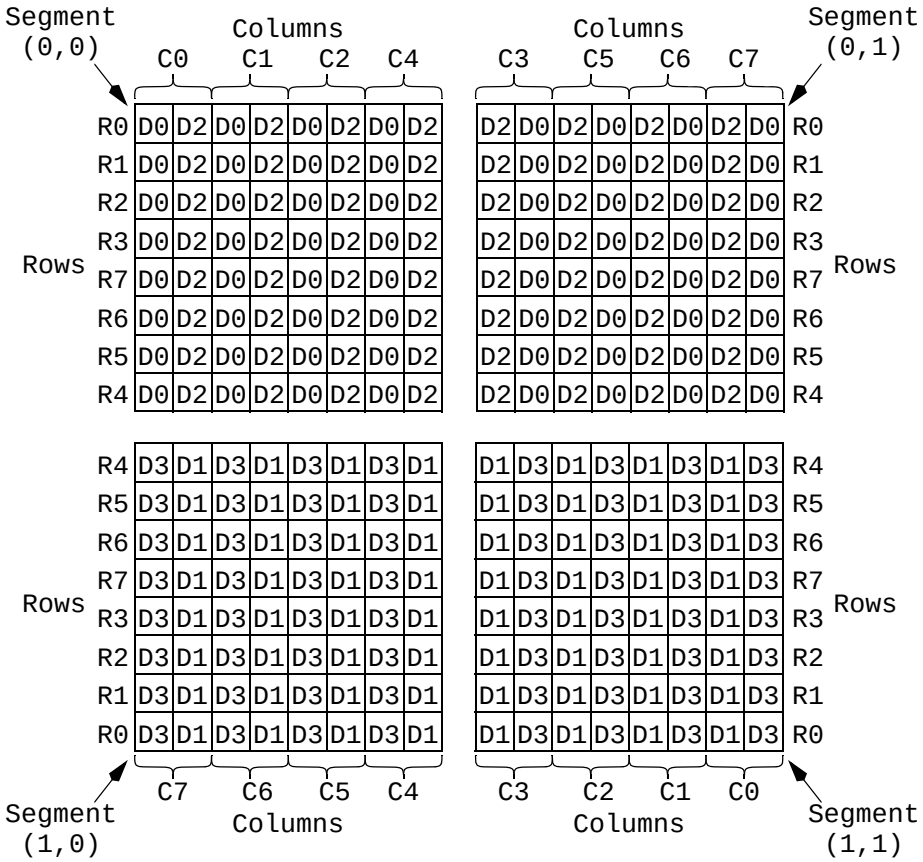
Bitmap Schemes, plus the names of any *registered* user-defined bitmap scheme. The example below shows the `builtin_col_data` scheme is selected:



A bitmap scheme describes, in software, a model used to convert the raw ECR failure data from the default *logical* view to the desired (*physical*) view. The model describes how the device is partitioned, into a number of *bitmap segments* which, in the logical-to-physical conversion application, correlates with the device's physical topology.

A bitmap segment is defined to be a contiguous rectangular area of the device (and the bitmap display) in which the *data permutation* is constant (more below).

For example, the following model shows a physical device topology containing 8 rows (R0-R7), 8 columns (C0-C7), and 4 data bits (D0-D3).



Note the following;

- The device is partitioned into four bitmap segments which, in this example, is dictated by a basic rule: the data *permutation* must be constant within a bitmap segment. It is the uniqueness of data bit organization that dictates that four segments are needed for this example.
- The data bits change in the column direction, but not in the row direction. This means data is *associated* with columns, which must be specified when creating the bitmap scheme (`make_bitmap_scheme()`).
- The order of rows is different between upper and lower halves but are otherwise identical between left/right segments. This will allow the four bitmap segments to be defined using only 2 row permutation definitions.
- The order of columns is different in all four segments thus four column permutation definitions will be needed.

It is the definitions of the row, column and data permutations which describe the order rows and columns will be displayed for each bitmap segment, which data bits appear in each segment, and in which order data bits are displayed. The code used to define the permutations for this example device is shown in Example 2:

The following steps describe the process of creating a new bitmap scheme:

- Create a new bitmap scheme object, using `make_bitmap_scheme()`.
- Evaluate the device's physical topology to determine row and column ordering and variations of that ordering. Also determine how the data bits are distributed and ordered. The result is a diagram of bitmap segments (see the example on page 1916).
- Using `make_permutation()`, and the other functions which operate on permutations documented in Bitmap Scheme Functions and Data Types, define the row, column and data permutations needed to describe each bitmap segment. In the bitmap scheme software, the term permutation is used to define a unique row configuration, ***or*** column configuration, ***or*** data bit configuration within a bitmap segment.
- Using `add_segment()`, define and add each bitmap *segment* to the bitmap *scheme*. Each bitmap segment is described with a X/Y size and with one row, column, and data permutation.
- Using `register_bitmap_scheme()`, register the bitmap scheme with UI. This makes the scheme usable in BitmapTool

Each bitmap segment is created (`add_segment()`) by specifying its X/Y position and one row, one column, and one data `permutation`. In the example on page 1916, four bitmap segments are needed: (0,0), (0,1), (1,0), (1,1). By default, Bitmap segments are numbered by their coordinate position on the screen; this is using the original automated positioning methods, called tile-mode. However, it is possible to overlay segments by specifying an absolute screen location rather than tile-mode. See Bitmap Segment Positioning.

By default, the divisions shown between segments in the example do not appear in the bitmap display. They are shown here for clarity. Using Bitmap Overlays it is possible to add visible segmentation to the display.

Each bitmap scheme description specifies whether the data permutation for that scheme is to be associated with rows or columns. This is required because the data `permutation` only describes the quantity and order of data bits, but not how they are organized within a segment i.e. whether they change in the row direction or column direction. By definition, they can't change in both directions, and the direction can't change within a bitmap scheme.

In software, each row, column, or data permutation to be created is declared as a variable of type permutation, a Nextest-defined data type. In some applications, to simplify the creation of bitmap segments, it is common to declare arrays of permutation. This supports the following software construct:

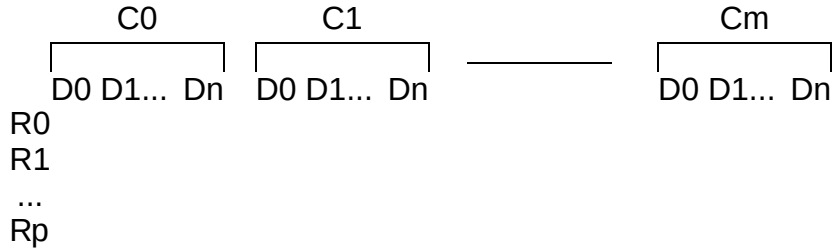
```
permutation rowperm[2];     // Array of two row permutations
permutation colperm[2][2];  // Array of four column permutations
permutation dataperm[2][2]; // Array of four data permutations

// ... Permutation definitions go here ... see Example 2:
for( int horiz_seg_pos=0; horiz_seg_pos < 2; ++horiz_seg_pos )
   for( int vert_seg_pos=0; vert_seg_pos < 2; ++vert_seg_pos )
      add_segment( scheme, horiz_seg_pos, vert_seg_pos,
                   rowperm [horiz_seg_pos],
                   colperm [horiz_seg_pos] [vert_seg_pos],
                   dataperm [horiz_seg_pos] [vert_seg_pos] );
}
```

6.3.10.2 Built-in Bitmap Schemes

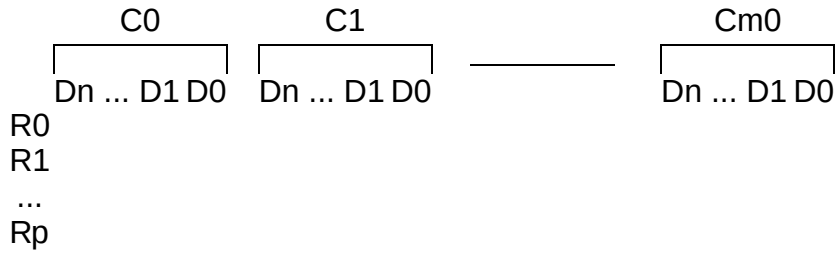
See Bitmap Schemes.

BitmapTool has 8 built-in bitmap schemes:

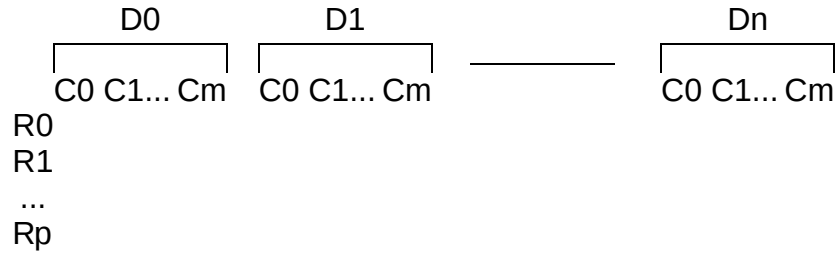
1. `builtin_col_data`: display a logical view of the ECR data. Data is associated with Columns and Data is fast i.e. *data bits are* grouped together displaying D0..Dn for column 0, followed by D0..Dn for column 1, etc. This is the default.



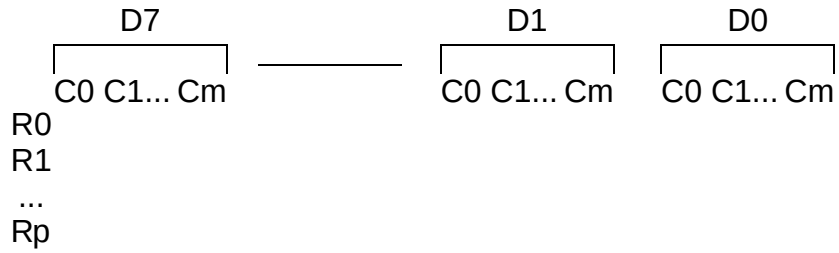
2. `builtin_col_rdata` : is the same as the previous except that Data is reversed.



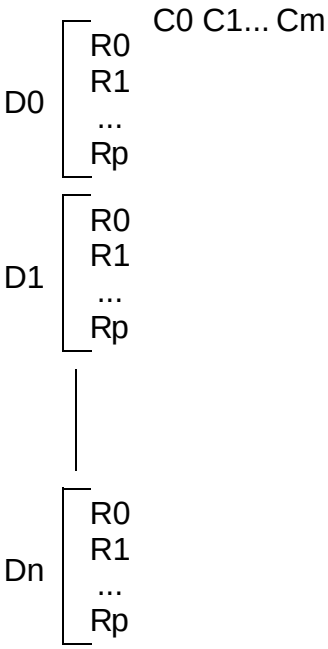
3. `builtin_data_col` : display a logical view of the ECR data. Data is associated with Columns and Columns are fast i.e. *columns* are displayed grouped together with all columns for D0, followed by all columns for D1, etc.



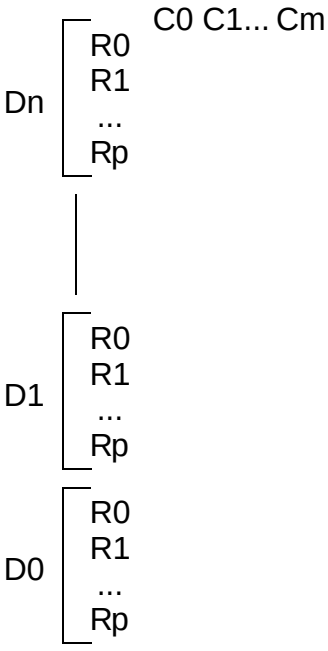
4. `builtin_rdata_col` : this is the same as the previous except that Data is reversed.



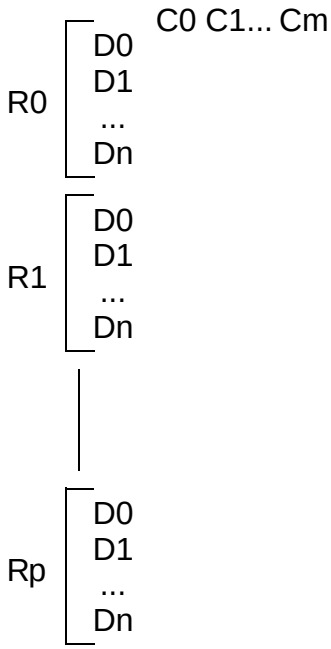
5. `builtin_data_row` : display a logical view of the ECR data. Data is associated with Rows and Rows are fast i.e. *Rows* are displayed grouped together with all Rows for D0, followed by all Rows for D1, etc.



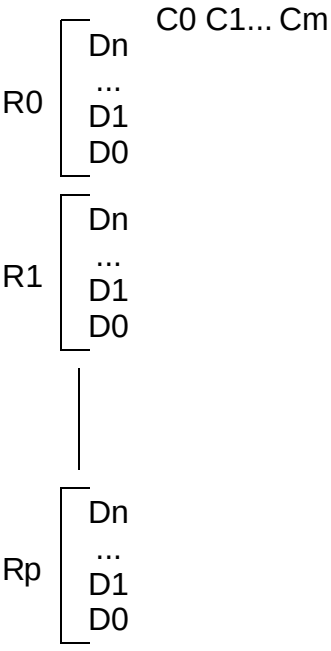
6. `builtin_rdata_row` : is the same as the previous with Data reversed.



7. `builtin_row_data` : display a logical view of the ECR data. Data is associated with Rows and Data is fast i.e. *data bits are* grouped together displaying D0..Dn for Row 0, followed by D0..Dn for Row 1, etc.



8. `builtin_row_rdata` : this is the same as the previous with Data reversed.



In the builtin schemes above:

- **m** = the number of Columns per DUT, and is determined by the smaller of `numy()` and the ECR Y-width as set using `ecr_config_set()`.
- **p** = the number of Rows per DUT, and is determined by the smaller of `numx()` and the ECR X-width as set using `ecr_config_set()`.

n = number of Data bits per DUT, and is determined by the data width as set using `ecr_config_set()` divided by the number of DUT(s) defined in the Pin Assignment Table.

6.3.10.3 Bitmap Segment Positioning

See Bitmap Schemes.

Normally, each segment added to a Bitmap Schemes was automatically positioned in the display based on the segment coordinate values specified using the `rowseg` and `colseg` arguments to `add_segment()`. This is called tile-mode. In this mode, the `rowseg` and `colseg` values identify a segment's tile position, which cannot overlap with other segments, and there can't be gaps between segments. In this mode, all the Rules noted in Bitmap Scheme Functions and Data Types apply.

An optional mode allows bitmap segments to be positioned using an absolute screen-space mode. This allows segments to overlap, there can be gaps between seqments, etc.

The new mode is enabled using the optional `auto_arrange` argument to `make_bitmap_scheme()`. In this mode, when defining bitmap segments (`add_segment()`), the values specified for `rowseg` or `colseg` now define the absolute position of the segment in screen-space terms, relative to 0,0 (upper left corner).

Using the new mode, when defining row or column permutations (see permutation Data Type) the value -1 can be used to indicate *no value* for a component of that permutation. Once all segments are defined, any place a given row/column coordinate is still defined as -1 will not have any fail data displayed i.e. a hole in the bitmap display is created.

The example below should help to clarify this.

Example

As an example, consider the following device scrambling, which has data associated with columns and data shown fast. Row information is not shown because it doesn't improve this example:

Column:	0	1	2	3	4	5	6	7
Data:	0246	1357	0246	1357	0246	1357	0246	1357

Using the default tile-mode display option, a device with this layout would require 8 segments, because the data permutation changes in every column:

Column:	0	1	2	3	4	5	6	7
Data:	0246	1357	0246	1357	0246	1357	0246	1357
	Seg (0,0)	Seg (0,1)	Seg (0,2)	Seg (0,3)	Seg (0,4)	Seg (0,5)	Seg (0,6)	Seg (0,7)

Using the new mode, the same device can be defined using 2 overlapping segments. Both will be located at (0,0). Use -1 to create holes in each segment definition:

Seg (0,0) Column:	0	-1	2	-1	4	-1	6	-1
Seg (0,0) Data:	0246		0246		0246		0246	

Seg (0,0) Column:	-1	1	-1	3	-1	5	-1	7
Seg (0,0) Data:		1357		1357		1357		1357

This example can be coded as shown below:

```
// Data associated with columns, data fast, auto_arrange = FALSE
// See make_bitmap_scheme()
scheme = make_bitmap_scheme( TRUE, TRUE, FALSE )

// Define 2 column permutations, using -1 to create overlap holes

// See make_permutation()
static int EvenCols[] = { 0, -1, 2, -1, 4, -1, 6, -1 };
permutation Ceven = make_permutation( *EvenCols, 8 );

static int OddCols[] = { -1, 1, -1, 3, -1, 5, -1, 7 };
permutation Codd = make_permutation( *OddCols, 8 );

// Define 2 data permutations

static int EvenData[] = { 0, 2, 4, 6 };
permutation Codd = make_permutation( *EvenData, 4 );
```

```
static int OddData[] = { 1, 3, 5, 7 };
permutation Codd = make_permutation( *OddData, 4 );

// Row info is distracting...
permutation Rows = make_permutation( ... );

// Locate segment in upper-left (0,0) of bitmap display
add_segment( scheme, 0, 0, Rows, EvenCols, EvenData )

// Overlay 2nd segment on top of the 1st, this fills the holes
add_segment( scheme, 0, 0, row, OddCols, OddData )

// Register the scheme to be useable
// See register_bitmap_scheme()
register_bitmap_scheme( "Physical Map", scheme );
```

The following example delivers the same results and helps to demonstrate how the screen-space coordinate system works:

```
// The even-column segment definition doesn't change from above

// Note that the odd column segment definition now starts with the
// column 1 (the previous -1 is missing). Below, to properly use
// this segment to overlay the other one, it must be offset by one
// data width (4) in screen-space.
static int OddCols[] = { 1, -1, 3, -1, 5, -1, 7 };
permutation Codd = make_permutation( *OddCols, 7 );

// Row and data permutations don't change

// Locate 1st segment in upper-left (0,0) of bitmap display
// This is the same as the previous example.
add_segment( scheme, 0, 0, Rows, EvenCols, EvenData )

// Offset the overlay segment by 4 (one data width)
add_segment( scheme, 0, 4, row, OddCols, OddData )
```

6.3.10.4 Bitmap Scheme Functions and Data Types

See Bitmap Schemes.

A set of data structures and functions are provided to create bitmap scheme(s), create bitmap segments, define the row, column, and data permutations, and register the bitmap scheme.

The functions are organized under the following data type definitions:

- bitmap_scheme Data Type
- permutation Data Type

The following rules must be followed to obtain a useful display when using a user-defined bitmap scheme:

Rules

1. An ECR is required to perform bitmapping, and the ECR must be correctly configured (`ecr_config_set()`) to match the device's logical geometry.
2. To log errors to the ECR, `funtest()` must be executed using the `fullec` argument.
3. Normally, bitmap scheme code should be executed in the Host process, ***after*** all sites have loaded the test program. The simplest method is to define the `ui_ProgLoaded` user variable and put bitmap scheme code in the body of the variable. Also see Host Waiting for Site to Load example code. When bitmap scheme code is executed in a site process, the scheme can only affect those site(s).
4. Once a user-defined bitmap scheme is created, it is necessary to *register* it with UI before BitmapTool will display that scheme for selection. See `register_bitmap_scheme()`.
5. Each bitmap scheme defines whether the data bits are *associated* with rows or columns This association is constant for all segments.
6. Every row, column, and data bit coordinate must represent a unique value read from the ECR.

6.3.10.5 bitmap_scheme Data Type

See Bitmap Schemes.

The purpose and application of a bitmap scheme is discussed in Bitmap Schemes.

The Nextest software provides the functions noted below to:

- Create a bitmap scheme
- Define the `Bitmap Segment Positioning` mode
- Add bitmap segments to the scheme
- Name and register the scheme with UI; BitmapTool can only use bitmap schemes registered with UI.

- `dump()` the definition of a bitmap scheme for review.

The functions, macros, and key data structures related to the `bitmap_scheme` data type are:

`make_bitmap_scheme()`

`add_segment()`

`register_bitmap_scheme()`

`dump()`

6.3.10.6 make_bitmap_scheme()

See Bitmap Schemes.

Description

The `make_bitmap_scheme()` creates a new `bitmap scheme` object, which must be done before any bitmap segments can be created (added).

Each bitmap scheme description specifies whether the data permutations used in the scheme are associated with rows or columns. This is required because the data permutation only describes the order of data bits, but not how they are organized within a segment i.e. whether they change in the row direction or column direction. They can't change in both directions, and the direction can't change within a bitmap scheme.

Eight Built-in Bitmap Schemes are available to switch between row/column association and whether data or row/column is displayed fast.

Usage

```
bitmap_scheme make_bitmap_scheme( BOOL colData );
bitmap_scheme make_bitmap_scheme(
            BOOL data_with_col,
            BOOL data_fast DEFAULT_VALUE( TRUE ) );
bitmap_scheme make_bitmap_scheme(
            BOOL data_with_col,
            BOOL data_fast DEFAULT_VALUE( TRUE ),
            BOOL auto_arrange DEFAULT_VALUE( TRUE ) );
```

where:

data_with_col is TRUE if the data permutations are associated with columns, and FALSE if associated with rows.

data_fast is optional, and determines whether BitmapTool displays data fast (see above). Default is TRUE.

auto_arrange is optional, and determines whether Bitmap segments are automatically positioned (tiled, the original method), or positioned explicitly by the user. Specifying TRUE enables the original tile-mode operation, FALSE enables the new method. See Bitmap Segment Positioning.

The returned value is a `bitmap_scheme` object, usable as an argument to related functions.

Example

```
#define COLDATA TRUE // Data associated with columns
bitmap_scheme scheme = make_bitmap_scheme( COLDATA );
```

6.3.10.7 add_segment()

See Bitmap Schemes.

Description

The `add_segment()` function adds a new bitmap segment to an existing Bitmap Schemes.

Arguments define the coordinate position of the bitmap segment in the overall scheme, and which row, column, and data permutations describe the segment architecture.

All bitmap segments within a given bitmap scheme must be the same shape and size.

Usage

```
void add_segment(   bitmap_scheme scheme,
                    int rowseg,
                    int colseg,
                    permutation rowperm,
                    permutation colperm,
                    permutation dataperm );
```

where:

scheme is an existing bitmap scheme created using `make_bitmap_scheme()`. This identifies to which bitmap scheme the new bitmap segment is being added.

rowseg and **colseg** identify the coordinates of the bitmap segment being added in the overall scheme, and determines the position of each segment in the BitmapTool display. The upper left bitmap segment is identified as (0,0) i.e. **rowseg** = 0, **rowseg** = 0.

rowperm identifies the `row permutation` for this bitmap segment, previously created using `make_permutation()`.

colperm identifies the column permutation for this bitmap segment, previously created using `make_permutation()`.

dataperm identifies the data permutation for this bitmap segment, previously created using `make_permutation()`.

Example

This example is a portion of the code used to describe the example device shown on page 1916. The row, column, and data permutation definitions are ***not shown*** here; see `Example 2:` for the complete solution.

```
#define COLDATA TRUE // Data associated with columns
bitmap_scheme scheme = make_bitmap_scheme( COLDATA );
// Add 4 segments to the scheme using nested loops
for( int horiz_seg_pos=0; horiz_seg_pos < 2; ++ horiz_seg_pos )
   for( int vert_seg_pos=0; vert_seg_pos < 2; ++ vert_seg_pos )
      add_segment( scheme, horiz_seg_pos, vert_seg_pos,
                   rowperm[horiz_seg_pos],
                   colperm[horiz_seg_pos][ vert_seg_pos],
                   dataperm[horiz_seg_pos][ vert_seg_pos] );
```

6.3.10.8 register_bitmap_scheme()

See Bitmap Schemes.

Description

The `register_bitmap_scheme()` function assigns a bitmap scheme a name, and registers it with UI. This makes the bitmap scheme selectable (and usable) in BitmapTool by adding the specified name to BitmapTool's *Bitmap Scramble Select* dialog menu.

Note: the `register_bitmap_scheme()` frees the memory used to store any and all `permutations` which were used *directly*, or *indirectly*, to define the `bitmap segments` which comprise the scheme being registered. This has some important side effects. See Permutation Memory Management.

Usage

```
void register_bitmap_scheme( LPCTSTR name, bitmap_scheme scheme );
```

where:

name is the name to appear in the BitmapTool controls dialog used to select a bitmap scheme.

scheme identifies bitmap scheme being registered.

Example

```
register_bitmap_scheme( "Physical Map", scheme );
```

6.3.10.9 dump()

See Bitmap Schemes.

Description

In the context of BitmapTool, the `dump()` function has two overloads used for validation and debugging bitmap schemes.

- Output the description of the specified bitmap scheme.
- Output the description of the specified permutation.

Usage

```
void dump( bitmap_scheme scheme );
```

```
permutation dump( permutation p );
```

where:

scheme is the bitmap scheme of interest.

p is a row, column, or data `permutation` of interest.

Example

```
bitmap_scheme scheme = make_bitmap_scheme( TRUE );
... other code to describe permutations and bitmap segments
dump( scheme );
```

6.3.10.10 permutation Data Type

See Bitmap Schemes.

The purpose and application of *permutations* is discussed in Bitmap Schemes.

The functions documented in this section are used to define the row, column, and data permutations used to specify a bitmap segments.

In general, a permutation has two attributes:

- Permutation *size* i.e. how many elements (rows or columns or data bits) are represented by the permutation.
- Permutation *values* i.e. an integer value identifying which row(s), column(s), or data bit(s) are included in the permutation.

For example, a permutation is created with a *permutation size* of 4, and containing *permutation values* of 0, 2, 4, 6. This could be used to define a `permutation` of 4 rows, or 4 columns, or 4 data bits. If the permutation represented *rows* it would order 4 rows as R0, R2, R4, and R6, in that order. If the permutation represented *data bits* it would order 4 data bits as D0, D2, D4, and D6, in that order.

Each `permutation` is created by declaring a variable of type `permutation`, a Nextest-defined data type. In some applications, to simplify the creation of `bitmap segments`, it is common to declare arrays of permutation. This can be seen in Example 2:.

Each of the functions which create or modify a permutation have a permutation return value. This supports nesting or cascading these functions. For example, the following two statements can be replaced by a single statement as shown;

```
permutation p = make_permutation( 8 );
permutation n = reverse ( p );
```

Both **p** and **n** permutations exist and are each available to define a bitmap segment.

```
permutation n = reverse ( make_permutation( 8 ) );
```

This example creates the same permutation **n** as the above, however, permutation **p** is not created and thus not available for use to specify a bitmap segment.

The following style is also legal:

```
permutation p = make_permutation( 8 );
p = reverse ( p );
```

Which results in the single permutation **p**.

The number and variety of functions provided which create or modify a permutation is related to software efficiency. The example on page 1916 is quite simple, and can be described quite easily (see Example 2:). However, when describing the physical topology for large devices, often the number of bitmap segments can be quite large, with a correspondingly large number of row, column and data permutations. With some creativity, the functions noted below can be combined to simplify creating these permutations. See Examples.

The functions, macros, and key data structures related to the `permutation` data type are:

- `make_permutation()`
- `append()`
- `dump()`
- `filter()`
- `for_each()`
- `get()`
- `insert()`
- `reverse()`
- `rotate()`
- `set()`
- `size()`
- `swap()`

6.3.10.11 Permutation Memory Management

See Bitmap Schemes.

Only `make_permutation()` actually allocates memory for permutation use. The other functions which return a permutation are returning the original permutation. Some functions can modify the size of an existing permutation, which requires the use of dynamic memory management methods, to allocate and free memory as needed. To keep this transparent to the user, the underlying code has a simple paradigm:

- The `make_permutation()` function allocates memory.
- For any permutations added to a segment all additional memory management is handled by system software.

This has two important side effects:

- Any permutations used to define a bitmap segment are destroyed after `register_bitmap_scheme()` returns, and thus do not exist to create additional schemes.
- Any permutation which is created but otherwise not used to define a bitmap segment will not have its associated memory freed.

6.3.10.12 make_permutation()

See Bitmap Schemes.

Description

The `make_permutation()` function has 5 overloads, any of which can be used to create a row, column or data permutation to be used when describing a bitmap segment.

Usage

```
permutation make_permutation( int size,
                              int start DEFAULT_VALUE( 0 ),
                              int inc DEFAULT_VALUE( 1 ) );
permutation make_permutation( int *data, int count );
permutation make_permutation( CArray< int, int > &array );
permutation make_permutation( permutation p );
```

```
permutation make_permutation( permutation p, int start, int end );
```

where:

`size` specifies the permutation *size*. This represents the number of rows ***or*** columns ***or*** data bits in the permutation being created. When no additional arguments are specified, the first permutation *value is* 0, and the value is incremented by 1.

In the 1st function above, **`start`** is optional, and defaults to 0. If used, **`start`** specifies the first permutation *value*.

`inc` is optional, and defaults to 1. If used, **`inc`** specifies how permutation *values* are incremented. When using this argument also requires using the optional **`start`** argument.

`*data` is a user-defined array of integers to be used as explicit permutation *values* for the permutation being created. **`count`** is the permutation *size*, and specifies the number of *values* in the array which are used. The array is accessed starting with the first element.

`&array` allows the use of an *MFC* `CArray` to store an array of integers to be used as explicit permutation *values* for the permutation being created. The number of permutation *values* is derived from the size of the `CArray`.

`p` identifies another permutation, which will be copied into the new permutation being created. This supports copy/modify methods to leverage an existing permutation.

In the last function above, **`start`** and **`end`** specify the first and last permutation *values* to be copied from the permutation specified by **`p`**. The new permutation is sized to match the number of values copied.

Note: the `make_permutation()` function allocates memory for storing the permutation being created. See Permutation Memory Management.

Examples

Example 1:

```
// First function overload with no optional arguments
permutation p = make_permutation( 16 ); // p = { 0, 1, 2, …, 15 }

// First function overload using the optional start argument
permutation p = make_permutation( 16, 1 ); // p = { 1, 2, 3, …, 16 }

// First overload using both optional arguments
permutation p = make_permutation( 16, 1, 2 ); // p = { 1,3,5,…,31 }
```

```
// Specify an explicit list of permutation values using an array
// but only use 4 of 5 values from the array.
static int data[] = { 1, 2, 4, 8, 16 };
permutation p = make_permutation( *data, 4 ); // p = { 1, 2, 4, 8 }

// Copy permutation p to create new permutation c
permutation c = make_permutation( p );  // c = { 0, 1, 2, …, 15 }

// Copy part of permutation p to create new permutation d
permutation d = make_permutation( p, 2, 3);  // c = { 2, 3 }

// Specify an explicit list of permutation values using an MFC
// CArray. Note there are many other ways to size/init a CArray.
CArray < int, int > a; // Create the array object
a.SetSize( 16 );       // Set the size of the array
for(int i = 0; i < a.GetSize(); ++i ) // Put 0..15 in the array
   a[ i ] = i;
permutation p = make_permutation( a ) // p = { 0, 1, 2, …, 15 }
```

Example 2:

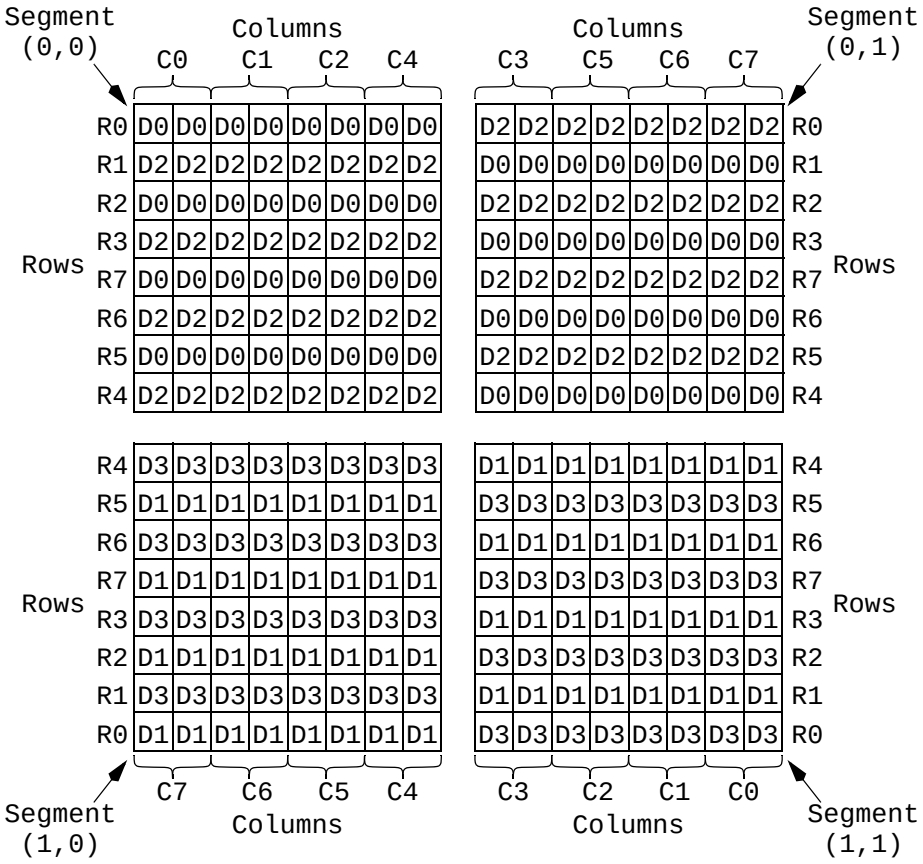
This code describes the physical topology for the example device model shown on page 1916.

```
permutation rowperm[2];// Two row permutations

// For segments (0,0) (0,1) = row order {0,1,2,3,7,6,5,4}
rowperm[0] = reverse( make_permutation(8), 4, 7);

// For segments (1,0) (1,1) = row order {4,5,6,7,3,2,1,0}
rowperm[1] = reverse( reverse( make_permutation(8)), 0, 3);

permutation colperm[2][2];// Four column permutations

// For segmentS (0,0) = column order {0,1,2,4}
colperm[0][0] = set( make_permutation(4), 3, 4 );

// For segment (0,1) = column order {3,5,6,7}
colperm[0][1] = set( make_permutation(4,4), 0, 3 );

// For segment (1,0) = column order {7,6,5,4}
colperm[1][0] = reverse( make_ permutation(4,4) );

// For segment (1,1) = column order {3,2,1,0}
colperm[1][1] = reverse( make_ permutation(4) );

permutation dataperm[2][2];// Four data permutation

// For segment (0,0) = data order {0,2}
dataperm[0][0] = make_permutation( 2, 0,  2 );
```

```
// For segment (0,1) = data order {2,0}
dataperm[0][1] = make_permutation( 2, 2, -2 );

// For segment (1,0) = data order {3,1}
dataperm[1][0] = make_permutation( 2, 3, -2 );

// For segment (1,1) = data order {1,3}
dataperm[1][1] = make_permutation( 2, 1,  2 );

#define COLDATA TRUE // Data associated with columns

bitmap_scheme scheme = make_bitmap_scheme( COLDATA );

// Add 4 segments to the scheme using nested loops

for (int horiz_seg_pos=0; horiz_seg_pos < 2; ++ horiz_seg_pos)
   for (int vert_seg_pos=0; vert_seg_pos < 2; ++ vert_seg_pos)
     add_segment( scheme, horiz_seg_pos, vert_seg_pos,
                  rowperm[horiz_seg_pos],
                  colperm[horiz_seg_pos][ vert_seg_pos],
                  dataperm[horiz_seg_pos][ vert_seg_pos] );

dump( scheme );   // For review and debugging

// Name and register the scheme with UI to enable use in BitmapTool
register_bitmap_scheme( "Physical Map", scheme );
```

Example 3:

The example below is identical to that on page 1916 except that the data changes in the row direction



The software used to create a bitmap scheme for this device is identical except for one statement:

```
// Data is associated with rows
bitmap_scheme scheme = make_bitmap_scheme( FALSE );
```

6.3.10.13 reverse()

See Bitmap Schemes.

Description

The `reverse()` function modifies an existing permutation by reversing the order of permutation *values*.

Two overloads exist:

- Reverse the values in the complete permutation.
- Reverse a range of *values* in the permutation.

Usage

```
permutation reverse( permutation p );
permutation reverse( permutation p, int start, int end );
```

where:

p identifies the `permutation` being modified.

`start` and **end** specify a range of *value* positions (inclusive) to be reversed.

This function returns the modified permutation, which is convenient when cascading (nesting) the various permutation functions to obtain complex results in terse software statements.

Examples

Example 1:

To reverse the order of an entire permutation:

```
permutation p = make_permutation( 8 );// p = { 0, 1, 2, …, 7 }
p = reverse( p );                     // p = { 7, 6, 5, …, 0 }
```

The two statements above can be combined:

```
permutation p = reverse( make_permutation(8) );
```

Example 2:

To reverse a range of *values* in the permutation:

```
permutation p = make_permutation( 8 );// p = { 0, 1, 2, …, 7 }
p = reverse( p, 0, 2 );               // p = {{ 2, 1, 0, …, 7 }
```

The statements above can be combined:

```
permutation p = reverse( make_permutation( 8 ), 0, 2);
```

6.3.10.14 rotate()

See Bitmap Schemes.

Description

The `rotate()` function modifies an existing permutation by rotating the order of permutation *values*. Both right-rotation and left-rotation is possible.

Usage

```
permutation rotate( permutation p, int amount );
```

where:

p identifies the permutation being modified.

amount specifies how much rotation is performed. A negative value causes left-rotation.

This function returns a permutation, which is convenient when cascading (nesting) the various permutation functions to obtain complex results in terse software statements.

Examples

Example 1:

To rotate the values in a permutation right by 2 positions:

```
permutation p = make_permutation( 8 );// p = { 0, 1, 2, …, 7 }
p = rotate( p, 2 );                   // p = { 6, 7, 0, 1, …, 5}
```

The statements above can be combined:

```
p = rotate( make_permutation( 8 ), 2 );
```

Example 2:

To rotate the values in a permutation *left* by 2 positions:

```
permutation p = make_permutation( 8 );// p = { 0, 1, 2, …, 7 }
p = rotate( p, -2 );                  // p = { 2, 3, 4, …, 0, 1}
```

The statements above can be combined:

```
permutation p = rotate( make_permutation( 8 ), -2 );
```

6.3.10.15 swap()

See Bitmap Schemes.

Description

The `swap()` function modifies an existing permutation by swapping a range of permutation *values*

Usage

```
permutation swap( permutation p, int start1,
                  int end1, int start2, int end2 );
```

where:

p identifies the permutation being modified.

`start1` and **`end1`** specify a range of permutation values to be swapped with the range of values specified by **`start2`** and **`end2`**.

This function returns a permutation , which is convenient when cascading (nesting) the various permutation functions to obtain complex results in terse software statements.

Example

```
permutation p = make_permutation( 16 );
p = swap( p , 1, 6, 9, 14 );
```

The statements above result in the following permutations:

```
Original Permutation p: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Specified swap() ranges: 0 [1 2 3 4 5 6] 7 8 [9 10 11 12 13 14] 15 16

Resulting Permutation p: 0 [9 10 11 12 13 14] 7 8 [1 2 3 4 5 6] 15 16
```

6.3.10.16 append()

See Bitmap Schemes.

Description

The `append()` function modifies an existing permutation by *inserting* one permutation into to another permutation. The function prototype is used to describe its operation:

```
permutation append( permutation p, int index, permutation more );
```

- The **`more`** permutation is inserted into the **`p`** `permutation` at the **`index`** position +1.
- Any previous values in the **`p`** permutation at **`index`** +1 or beyond are moved to positions after the inserted elements. No holes are created. No values are lost.
- If the value of **`index`** is < 0 it is set = 0.
- If the value of **`index`** is beyond the original end of permutation **`p`**, empty values (0?) are inserted into permutation to fill the gap.
- The modified permutation **`p`** is returned by the `append()` function.

Note: the `insert()` function operates identically except that the insertion is made at the index value, not at index +1.

Usage

```
permutation append( permutation p, int index, permutation more );
```

where:

`p` identifies the permutation being modified.

`index` identifies the location in the permutation copy after which the permutation identified by **`more`** will be inserted. The index number of the first value in a `permutation` is 0.

This function returns a permutation, which is convenient when cascading (nesting) the various permutation functions to obtain complex results in terse software statements.

Note: the `append()` function may allocate memory for storing the permutation being created. See Permutation Memory Management.

Example

The permutation **`n`** is appended to permutation **`p`**:

```
permutation p = make_permutation( 8 );  // p = {0, 1, 2, …, 7}
permutation n = make_permutation( 2, 9);// n = {9, 10}
```

```
p = append( p, 7, n );                    // p = {0, …, 7, 9, 10}
```

6.3.10.17 insert()

See Bitmap Schemes.

Description

The `insert()` function modifies an existing permutation by effectively inserting one permutation into to another permutation. The function prototype is used to describe its operation:

```
permutation insert( permutation p, int index, permutation more );
```

- The **`more`** permutation is inserted into the **`p`** permutation at the **`index`** position.
- Any previous values in the **`p`** permutation at **`index`** or beyond are moved to positions after the inserted elements. No holes are created. No values are lost.
- If the value of **`index`** is < 0 it is set = 0.
- If the value of **`index`** is beyond the original end of permutation **`p`**, empty values (0?) are inserted into permutation to fill the gap.
- The modified **`p`** permutation is returned by the `append()` function.

Note: the `append()` function operates identically except that the insertion is made at the index value +1.

Usage

```
permutation insert( permutation p, int index, permutation more );
```

where:

`p` identifies the permutation being modified.

`index` identifies the location in permutation **`p`** at which the permutation identified by **`more`** will be inserted. The index number of the first value in a permutation is 0.

This function returns a permutation, which is convenient when cascading (nesting) the various permutation functions to obtain complex results in terse software statements.

Note: the `insert()` function may allocate memory for storing the permutation being created. See Permutation Memory Management.

Example

To insert a permutation before the specified index of a permutation:

```
permutation p = make_permutation( 8 ); // p = {0, 1, 2, …, 7}
permutation n = make_permutation( 2, 9); // n = {9, 10}
p = insert( p, 2, n );                  // p = { 0, 1, 9, 10, 3, …, 7}
```

6.3.10.18 set()

See Bitmap Schemes.

Description

The `set()` function modifies an existing permutation by changing the *value* at the specified *index* position to a specified *value*. The function prototype is used to describe its operation

```
permutation set( permutation p, int index, int value );
```

- The previous value at the **`index`** position in permutation **`p`** is replaced with **`value`**.
- If the value of **`index`** is < 0 it is set = 0.
- If the value of **`index`** is beyond the original end of permutation **`p`**, empty values (0?) are inserted into permutation to fill the gap.
- The modified **`p`** permutation is returned by the `append()` function.

Usage

```
permutation set( permutation p, int index, int value );
```

where:

`p` identifies the permutation being modified.

`index` identifies which position is to be modified. The index number of the first value in a permutation is 0.

`value` is the new value.

This function returns a permutation, which is convenient when cascading (nesting) the various permutation functions to obtain complex results in terse software statements.

Example

In this example, the value in permutation **p** is modified at position 2 from 2 to 9:

```
permutation p = make_permutation( 8 );  // p = {0, 1, 2, …, 7)
p = set( p, 2, 9 );                     // p = {0, 1, 9, …, 7)
```

The statements above can be combined:

```
permutation p = set( make_permutation( 8 ), 2, 9 );
```

6.3.10.19 for_each()

See Bitmap Schemes.

Description

The `for_each()` function modifies an existing permutation by calling a user-written callback function once for each *value* in the permutation. Each execution passes an index into the user callback function to indicate which *value* is currently being processed.

Note: the `for_each()` function is similar to `filter()`. The `for_each()` callback function returns void, and any manipulations of the permutation only occur from within the callback code. The `filter()` callback function returns BOOL, and in addition to any manipulations done in the callback code, if the callback returns FALSE the `filter()` function removes the permutation *value* from the permutation.

Usage

```
permutation for_each(  permutation p,
                       void (*func)( permutation p,
                       int index ) );
```

where:

p identifies the permutation being modified.

***func** is a user-written function conforming to the following function prototype:

```
void f( permutation p, int index );
```

where:

p is the permutation copy being processed.

index is initially = 0, and is subsequently incremented by `for_each()` on each execution of the callback function i.e. the index value increments from the first value to the last value. This allows the user callback to process each value in the permutation as a function of its position in the permutation.

This function returns a permutation, which is convenient when cascading (nesting) the various permutation functions to obtain complex results in terse software statements.

Example

To apply a function to each element of a permutation:

```
// user-written callback function. Prototype is documented above.
void triple( permutation p, int index ) {
   set( p, index, get( p, index ) * 3 ); // Any user code desired
}
permutation p = make_permutation( 8 );  // p = { 0, 1, 2, …, 7 }
p = for_each( p, triple );              // p = { 0, 3, 6, …, 21 }
```

6.3.10.20 filter()

See Bitmap Schemes.

Description

The `filter()` function processes and modifies an existing permutation by calling a user-written callback function once for each *value* in the copied permutation. Each execution passes an index into the user callback function to indicate which *value* is currently being processed.

The callback function can manipulate the permutation copy as desired. In addition, the BOOL return value may cause `filter()` to remove a value from the permutation copy as follows:

- Callback returns FALSE: `filter()` deletes the permutation *value* at the index position, and other values are shifted to fill the gap.

- Callback returns TRUE: `filter()` ignores the permutation *value* at the index position.

Note: the `for_each()` function is similar to `filter()`. The `for_each()` callback function returns `void`, and any manipulations of the permutation only occur from within the callback code. The `filter()` callback function returns `BOOL`, and in addition to any manipulations done in the callback code, if the callback returns FALSE the `filter()` function removes the permutation *value* from the permutation.

Usage

```
permutation filter( permutation p,
                    BOOL (*func)( permutation p,
                    int index ) );
```

where:

p identifies the permutation being modified.

`*func` is a user-written function conforming to the following function prototype:

```
BOOL f( permutation p, int index );
```

where:

p is the permutation copy being processed.

`index` is initially = 0, and is subsequently incremented by `filter()` on each execution of the callback function i.e. the index value increments from the first value to the last value. This allows the user callback to process each value in the permutation copy as a function of its position in the permutation.

The `BOOL` return value from the callback function directs the `filter()` function to delete (FALSE) or ignore (TRUE) the permutation *value* referenced by the current **`index`** value.

The `filter()` function returns a permutation (the copy), which is convenient when cascading (nesting) the various permutation functions to obtain complex results in terse software statements.

Example

The code below will remove *even* values from the permutation:

```
// user-written callback function. Prototype is documented above.
BOOL IsOdd( permutation p, int index ) {
   return get( p, index ) % 2; // Return FALSE if value is even
}
permutation p = make_permutation( 8 );// p = { 0, 1, 2, …, 7 }
permutation p = filter( p, IsOdd);    // p = { 1, 3, 5, 7 }
```

6.3.10.21 get()

See Bitmap Schemes.

Description

The `get()` function returns the permutation *value* from the specified permutation at the specified index position.

Usage

```
int get( permutation p, int index );
```

where:

p identifies the permutation of interest.

index specifies which position in the permutation is being read. The index number of the first value in a permutation is 0.

Example

```
permutation p = make_permutation( 8, 5);// p = { 5, 6, 7, …, 12 }
int v = get( p, 2 ); // v = 7
```

6.3.10.22 size()

See Bitmap Schemes.

Description

The `size()` function returns the count of *values* in the specified permutation.

Usage

```
int size( permutation p );
```

where:

p identifies the permutation of interest.

Example

```
permutation p = make_permutation( 8 );
int s = size( p ); // s = 8
```

6.3.10.23 bitmap_scheme_translate()

See Bitmap Schemes.

Description

The `bitmap_scheme_translate()` function can be used to translate one or more bitmap rectangles defined in the terms of ECR-space (typically a logical view of the DUT) to equivalent rectangles after being processed by a specified Bitmap Schemes.

Note: `bitmap_scheme_translate()` does not return useful values if either rectangle axis is zero-size. This is because it is in effect not selecting one or more BitmapTool atoms to be translated.

Usage

```
int bitmap_scheme_translate(  CString scheme,
                              RECT ecr_rect,
                              __int64 mask,
                              RectArray *screen_rects );
int bitmap_scheme_translate(  CString scheme,
                              RectArray &ecr_rects,
                              __int64 mask,
                              RectArray *screen_rects );
```

where:

scheme is the name of an existing Bitmap Schemes used to make the translation. If a NULL string is specified the currently selected Bitmap scheme is used.

ecr_rect specifies one rectangle, using ECR-space values, which identifies which rows/columns are to be included in the translation. Note that data is not included in this rectangle.

ecr_rects specifies multiple rectangles, each using ECR-space values, which identifies which rows/columns are to be included in the translation. Again, data is not included in this rectangle.

mask is a bit mask used to identify which data bits are to be included in the translation. The **mask** specification must be consistent with the specified **scheme** to obtain useful results.

screen_rects is a pointer to an existing RectArray variable used to return one or more rectangles resulting from the translation. This is suitable as an argument to bitmap_overlay_add(), used to create Bitmap Overlays.

bitmap_scheme_translate() returns the number of rectangles being returned in **screen_rects**. Any error returns -1.

Example

See Using Overlays to Locate Information in BitmapTool and the code below.

This example implements a *click-to-highlight* feature:

```
// Placing the code below in the test program enables the
// the following: when left-mouse is clicked in the BitmapTool zoom
// display, record screen Row/Col coords (ignore data). Use
// bitmap_scheme_translate() with mask = 0xFF to create rectangles
// around low 8 databits of Row/Col. Create a Bitmap overlay using
// those rectangles and display in BitmapTool.
TRANSLATE_BITMAP_INFO_7( row, col, databit,
                         clicked, message,
                         bitmap_row, bitmap_col ) {
   if ( ! clicked ) return;
   RectArray screen;
   bitmap_scheme_translate( "",
        CRect( col, row, col + 1, row + 1 ), 0xff, &screen );
   bitmap_overlay_add( "8 Data", screen, PS_SOLID, TRUE,
                       RGB(0,255,0) );
}
```

6.3.10.24 bitmap_scheme_lookup()

See Bitmap Schemes.

Description

Given the screen-space coordinates (logical view) of a databit in BitmapTool the `bitmap_scheme_overlay()` function can be used to return the corresponding (physical) row, column, and databit after the specified Bitmap scheme is considered.

Usage

```
BOOL bitmap_scheme_lookup(  CString scheme,
                            DWORD screen_row,
                            DWORD screen_col,
                            DWORD *row,
                            DWORD *col,
                            DWORD *io );
```

where:

`scheme` is the name of the Bitmap scheme of interest. If a `NULL` string is passed the currently selected Bitmap scheme is used.

`screen_row` and **`screen_col`** specify the screen-space row/column coordinates to be looked-up. Screen coordinates begin at 0/0 in the upper left corner of the BitmapTool display.

`row`, **`col`**, and **`io`** are the addresses of existing `DWORD` variables used to return the physical row, column, and databit values of the specified **`screen_row`** and **`screen_col`**, after being processed by the specified Bitmap scheme.

If either value specified for **`screen_row`** or **`screen_col`** falls outside the physical size of the device `bitmap_scheme_lookup()` returns `FALSE`, otherwise `TRUE` is returned.

Example

See the `BMT_print_ECR_space()` function in the example for `bitmap_overlay_add()`.

6.3.11 Bitmap Overlays

This section includes the following:

6.3.11.1 Overview

See Bitmap Overlays.

A Bitmap overlay is a user-defined graphic rectangle drawn in the BitmapTool display. Each time the BitmapTool display is updated, all defined Bitmap overlays are drawn, in the order created, after the initial BitmapTool display is painted and after failures are displayed.

Bitmap overlays have the following applications:

- Outline device physical attributes: memory segmentation, boot blocks, etc.
- Indicate areas of the BitmapTool display which don't represent valid memory addresses or otherwise need to be highlighted.

- Visibly indicate the location of specific row(s), and/or column(s), and/or data bit(s). This can be useful when complex Bitmap Schemes are used and the screen location of a specific row/column/data value is difficult to locate. See Using Overlays to Locate Information in BitmapTool.

Below is an example BitmapTool image showing 16 overlays generated by the code included in Example:

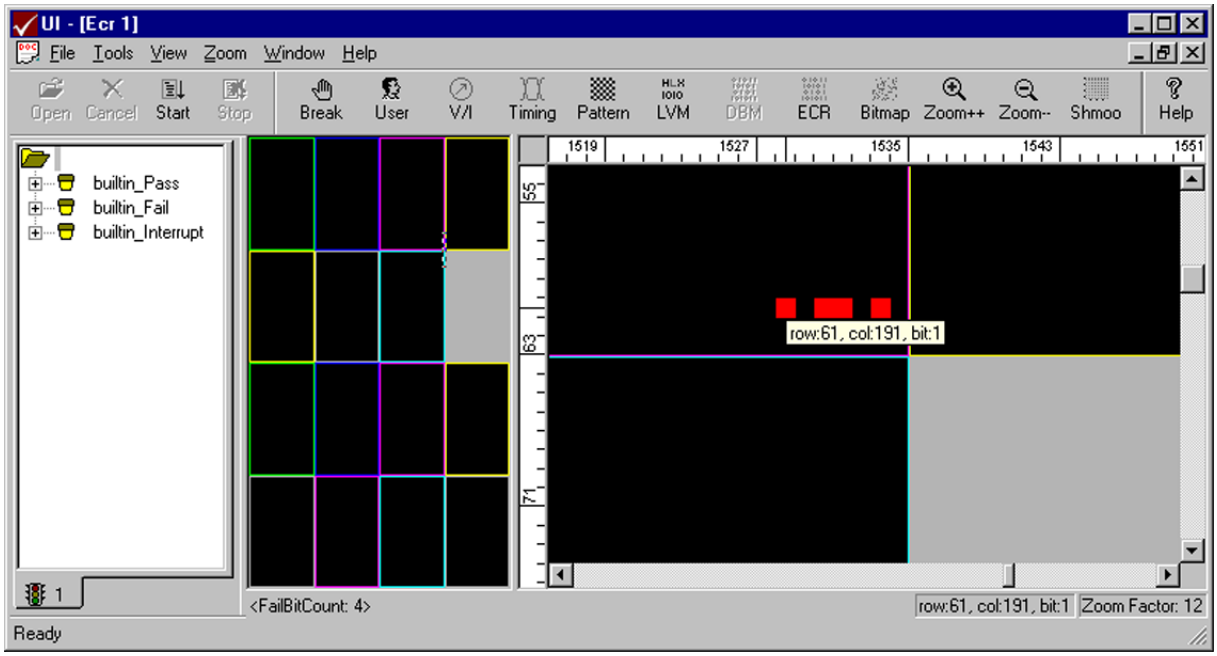


Figure-72: Example BitmapTool Display with Overlays

Note the following:

- Each overlay uses a different color.
- They all use a solid pen style (line type)
- One overlay (grey) is filled, the rest are not filled.
- The right screen is zoomed-in to enlarge a portion of the left display. Several failures are seen in the zoom window, displayed in red (default failure color).
- The optional BitmapTool rulers are displayed. See above

6.3.11.2 Creating Bitmap Overlays

See Bitmap Overlays.

Each Bitmap overlay is created using `bitmap_overlay_add()` to define one rectangle. Any number of Bitmap overlays can be defined, each with the following user specified attributes, specified as arguments to `bitmap_overlay_add()`:

- Overlay name, used to subsequently control or modify the overlay
- Rectangle size, which can also be a line or single point. More below.
- Filled or not filled (hollow). See `bitmap_overlay_add()`
- Color, specified as RGB values. See Bitmap Overlay Colors.
- Pen style (line type): SOLID, DASH, DOT, DASHDOT, DASHDOTDOT. See Bitmap Overlay Penstyles
- Enabled or disabled state i.e. visible or invisible. See bitmap_overlay_enable()

An overlay rectangle's size and location is specified using 4 position values corresponding to the *left*, *top*, *right*, and *bottom* sides of the rectangle. These are specified using screen-space values i.e. values which do not consider how the DUT row, column, and databit information is displayed in BitmapTool.

A rectangle specified with one zero-width axis creates a line. A rectangle specified with both axis as zero-width creates a point.

From the user viewpoint, the basic unit of display in BitmapTool is one databit, positioned at one row/column location. Depending on the current BitmapTool zoom factor BitmapTool draws each databit using one or many pixels. Changing the zoom factor only changes the number of pixels used to draw each databit.

Bitmap overlays are always drawn using single pixels, regardless of the current BitmapTool zoom factor. The following images show an exaggerated view of one databit and how each variation of overlay is drawn:

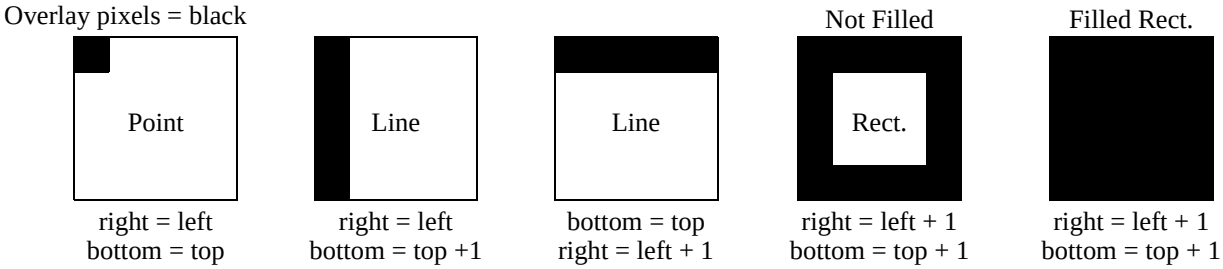


Figure-73: BitmapTool Atom vs. Overlay Rectangle Size

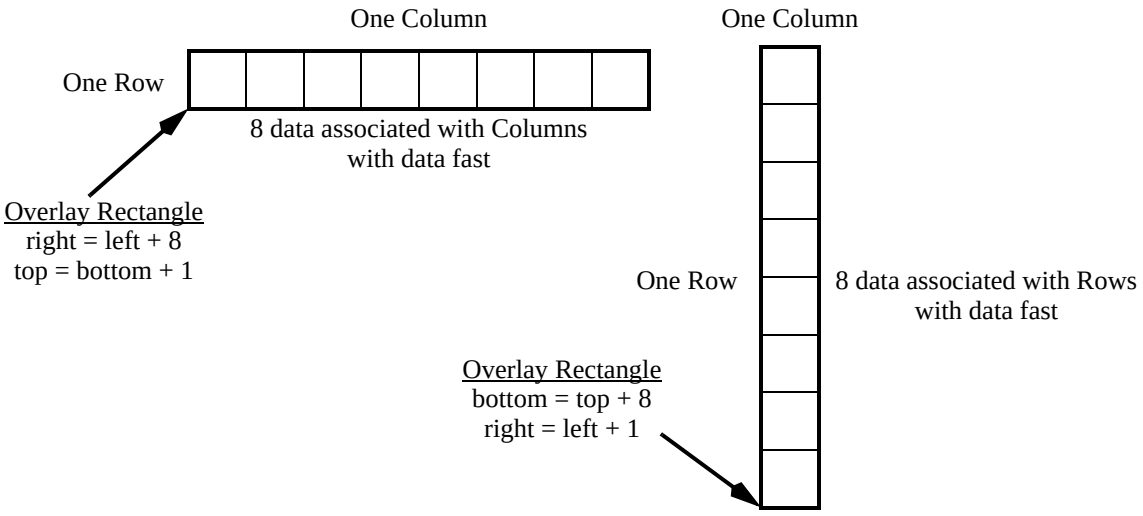
Bitmap overlay rectangles, and lines are not constrained to single databits, but if so specified would appear as above if BitmapTool could zoom-in to the extent shown.

Rectangle top and left values position the upper left corner of the overlay rectangle in the BitmapTool screen, relative to the upper left corner (0/0) of the display. Rectangle right and

bottom values determine the size of the rectangle relative to the left and top values respectively. When specifying right and bottom values two considerations exist:

- Rectangles have *width*, as noted above.
- Normally, one axis of the rectangle must be sized to account for displaying bitmap *data*.

Regarding the latter, BitmapTool typically displays information for multiple databits for each device row/column in the display. The currently selected Bitmap Schemes determines both how data is displayed, either associated with rows (data displayed vertically) or with columns (data displayed horizontally), and whether data is displayed fast or not (see Built-in Bitmap Schemes for examples clarifying these issues). Based on whether data is associated with rows or columns, and regardless of whether data is fast or not, an overlay rectangle must be sized to accommodate data in the appropriate axis. The diagram below shows both cases (both assuming data fast for simplicity). If the goal is to draw an overlay rectangle around all databits at one row/column address the required rectangle size will differ depending on whether data is associated with rows or columns:



6.3.11.3 Bitmap Overlay Colors

See Bitmap Overlays.

Note: this information only applies when the default Windows color pallet is in effect. Nextest can not support customized color pallets.

Bitmap overlay rectangles, lines, and points are drawn using a user-defined color, specified as 3 argument values (RGB) to the `bitmap_overlay_add()` function.

Overlays are drawn after the BitmapTool display is painted and after failures are added. Overlay lines are drawn by XOR'ing the specified line color with the existing color at any given point. This raises several issues:

- Overlay lines (points, etc.) occupy the same space as the pass/fail information normally displayed by BitmapTool. In particular, it is possible (likely) that a failure and an overlay line will partially or completely overlay the same screen space.
- When a failure and an overlay line do overlay the same screen space the XOR of the two colors will result in a third color, *EXCEPT* when the overlay color is the same as that used to display failures (normally red), in which case they cancel (the resulting color is black). This suggest that it is not wise to use the color red as an overlay color.
- Similarly, when portions of two overlays occupy the same space the resulting color will be different than that specified for the individual overlays (see Note on page 1955).

The following image is greatly (and artificially) enlarged, to show how colors will interact in common overlay scenarios:

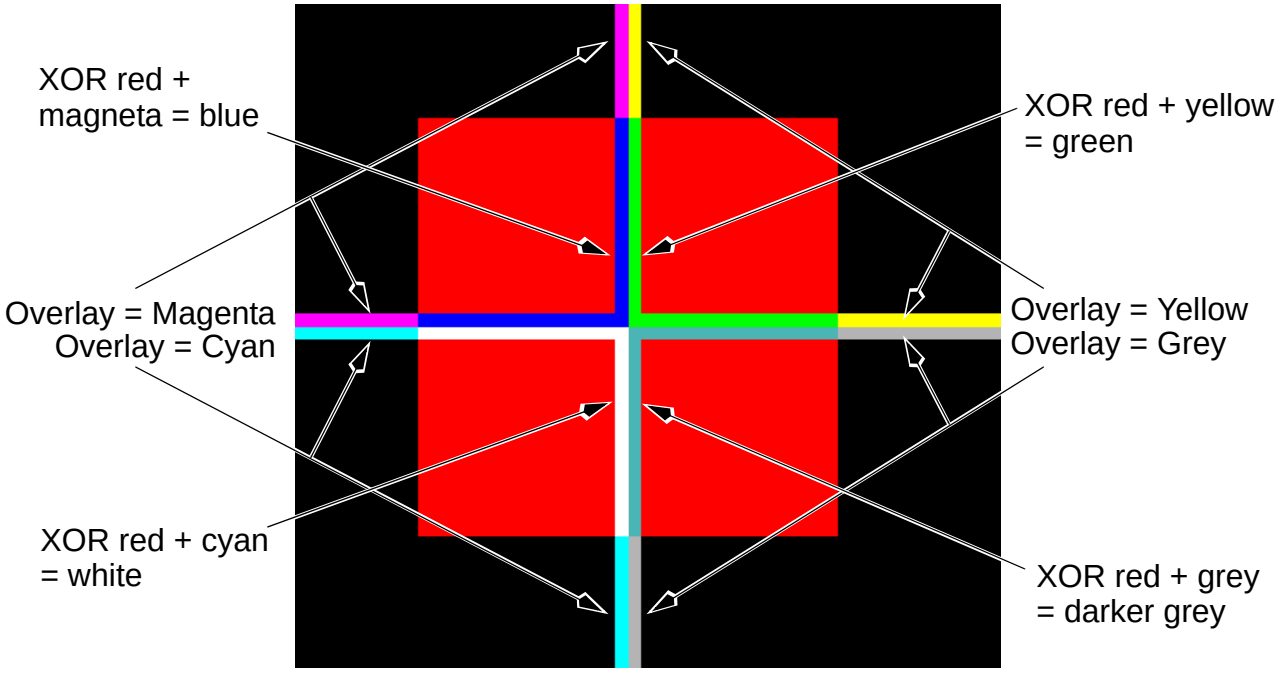


Figure-74: Bitmap Overlay Color XOR Example

Note the following:

- The colored lines entering the image at top-center, bottom-center, left-center, and right-center are portions of 4 separate Bitmap overlays, drawn with the colors specified by the overlay definitions (magenta, yellow, cyan, grey).
- The 4 red squares represent 4 discrete failing databits. Due to the extreme enlargement, each failing databit is drawn using many pixels. Overlay lines are always drawn using single pixels, which are exaggerated in this image.
- Note that portions of each failing bit occupy the same screen space as portions of the overlay lines. The resulting colors are the XOR of red (default failing bit color) and the overlay color.
- This type of interaction will also occur when overlays collide.

Note: the previous bullet is important. When overlays collide the only indication is that the expected colors do not appear. Conversely, when defining overlays, when the expected colors do not appear, it is likely the overlays are colliding.

The `bitmap_overlay_draw()` function can be used to cause a specified Bitmap overlay to flash. When an overlay is XOR'ed with itself the result is black, thus repeatedly executing `bitmap_overlay_draw()` of the same overlay will cause it to flash. See Example.

6.3.11.4 Bitmap Overlay Penstyles

See Bitmap Overlays.

Bitmap overlay rectangles, and lines are drawn with user specified pen style (line type), specified as an argument to the `bitmap_overlay_add()` function. The following options are available:

```
PS_SOLID       a solid line
PS_DASH        -------
PS_DOT         .......
PS_DASHDOT     _._._._
PS_DASHDOTDOT  _.._.._
```

6.3.11.5 Using Overlays to Locate Information in BitmapTool

For applications where overlay(s) are used to highlight a device's physical layout (segmentation, etc.) the rectangle values are easily specified using screen-space values. However, another useful application of Bitmap overlays is to visually highlight specific row, column, and data bits when the BitmapTool display has been mapped using [complex] Bitmap Schemes.

A Bitmap scheme determines the mapping of data bits in the ECR to pixels on the screen. There is always a Bitmap scheme in effect. By default it is one of the Built-in Bitmap Schemes, named `builtin_col_data`, which associates data with columns and displays data fast.

The most common application for user-defined Bitmap schemes is to cause BitmapTool to display a physical view of ECR topological data i.e. display rows, columns and databits as they are physically located in the DUT rather than as a logical view of the ECR. See Logical vs. Physical, vs. Electrical Addresses. In these situations, it is common for rows and/or columns and/or databits to be displayed such that ordering in the display is not linear i.e. the screen position of a given row/column/databit in the BitmapTool display is sometimes not easy to locate. Often, all databits of a given row/column address may not be displayed contiguously or adjacently, even when data is displayed fast. In these situations, a Bitmap overlay can be used to highlight the screen position(s) of the desired information in the BitmapTool display (see example This example implements a click-to-highlight feature:).

Note: Bitmap schemes can be used for applications other than converting a logical view into a physical view. However, to aid comprehension of these concepts, this terminology is used consistently in this section.

To use this technique, the `bitmap_scheme_translate()` function is used to translate rectangle(s) specified in ECR-space to rectangles defined in screen-space. For example:

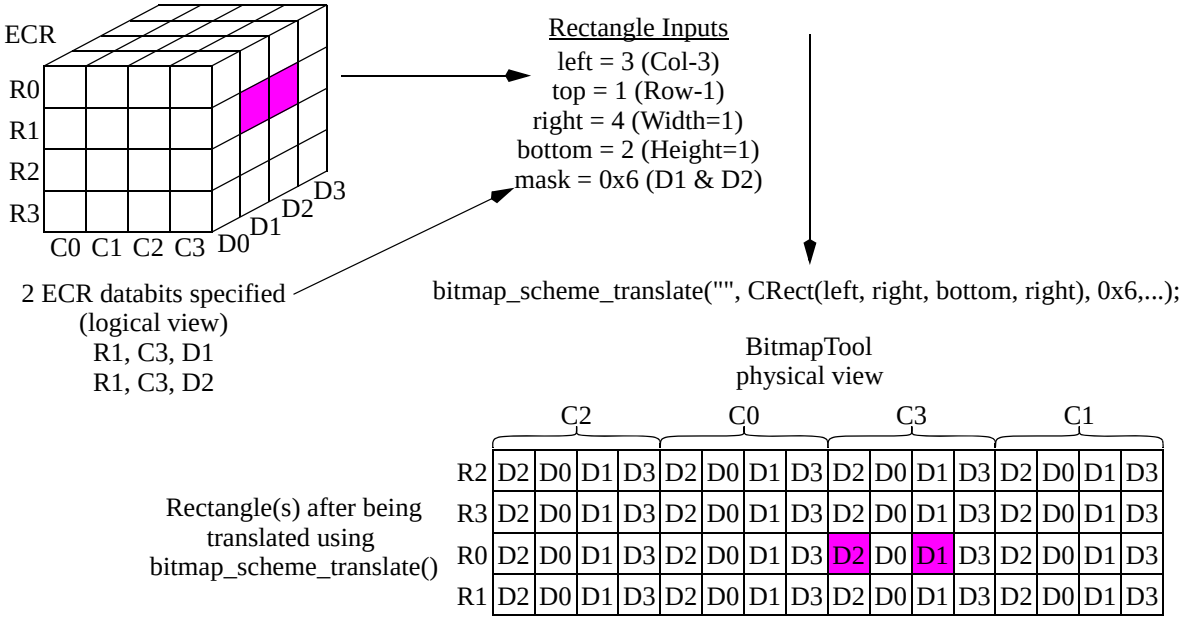


Figure-75: Overlay using bitmap_scheme_translate()

Referring to the images above, note the following:

- The ECR is a 3 dimensional array of row, column and pin (databit) information. To keep things manageable this example shows only 4 columns (C0..C3), four rows (R0..R3), and 4 databits (D0..D3).
- The ECR view highlights 2 databits in ECR-space. To select which information is processed by `bitmap_scheme_translate()`a rectangle is defined which encompasses the desired rows and columns in ECR-space. Note that in databits are selected using a bit mask argument to `bitmap_scheme_translate()`, thus the rectangle used to select the 2 databits noted above will have a width of 1, selecting column-3, and a height of 1, selecting row-1. The data mask = 0x6 to select databit-1 and databit-2.
- The currently selected Bitmap Schemes used for this example has data associated with columns, and data displayed fast, plus the row, column, and databit ordering seen in the lower BitmapTool image.

Note: `bitmap_scheme_translate()` does not return useful values if either input rectangle axis is zero-size. This is because the rectangle is, in effect, not selecting one or more databits to be translated.

Given the screen-space coordinates (logical view) of a databit in BitmapTool the `bitmap_scheme_lookup()` function can be used to return the corresponding (physical) row, column, and databit after the current Bitmap scheme is considered.

6.3.11.6 Bitmap Overlay Example Device

See Bitmap Overlays.

The following device architecture is used in this section to demonstrate various Bitmap overlay features. The code for both the Bitmap scheme used to set up this segmentation and set up these overlays is included in the Example for `bitmap_overlay_add()`:

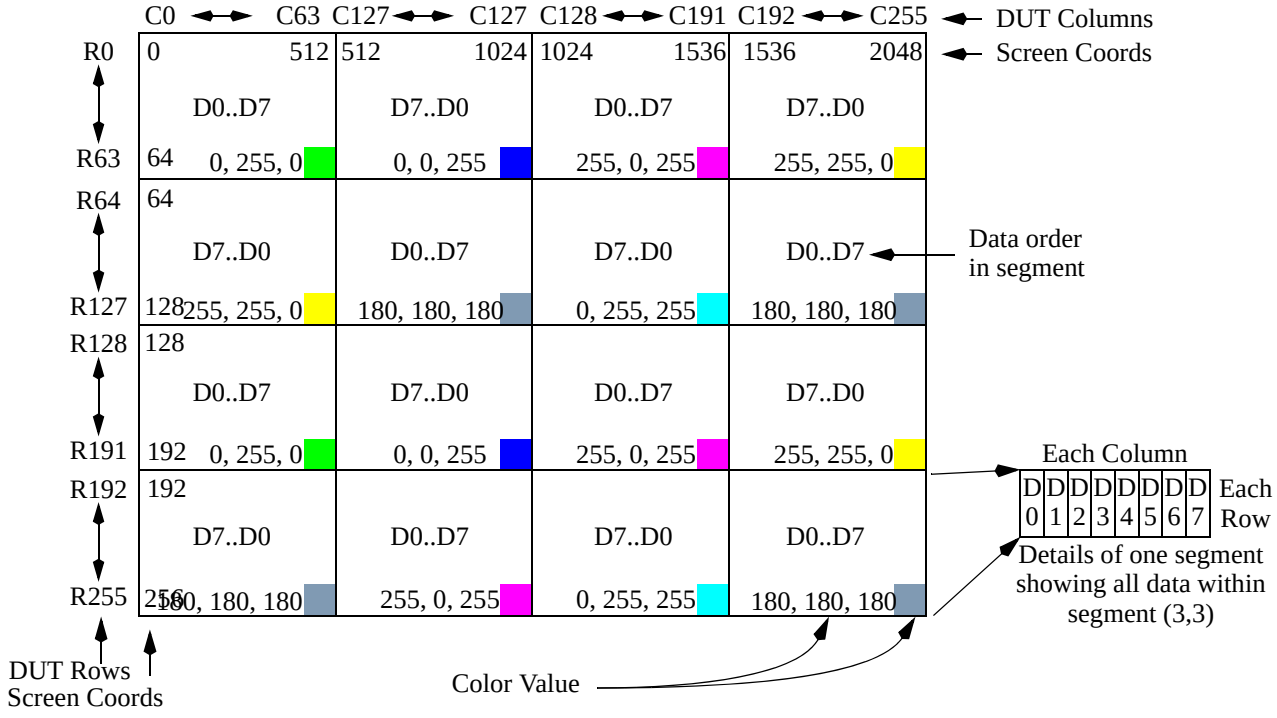


Figure-76: Bitmap Overlay Example Device Scheme

Note the following:

- This model represents both the device physical segmentation and the desired BitmapTool display.
- This example device has 16 segments. Each segment will be outlined using a Bitmap overlay, using the color values shown.
- In the example Bitmap scheme, data is associated with columns, and for bitmapping purposes data will be fast. The ordering of data in each segment is the reverse of the adjacent segments. This data ordering does not impact how Bitmap overlays are defined but would affect how rectangles are drawn if `bitmap_scheme_translate()` were used. See Using Overlays to Locate Information in BitmapTool.
- The *Screen Coords* values shown above are those used to specify the overlay rectangles, for each segment's overlay. Note that right and bottom values of each overlay are always +1, thus the right value of one overlay may also appear as the left value of the next. Similarly, the bottom value of one overlay may also appear as the top value of the next.

6.3.11.7 bitmap_overlay_names()

See Bitmap Overlays.

Description

The `bitmap_overlay_names()` function can be used to obtain a list of all Bitmap Overlays currently defined in the test program.

Usage

```
void bitmap_overlay_names( CStringArray *names );
```

where:

***names** is a pointer to an existing `CStringArray` used to return the names. This array will be zero-size if the test program does not define any Bitmap overlays.

Example

```
CStringArray names;
bitmap_overlay_names( &names );

for ( int i = 0; i < names.GetSize(); ++i )
   output("%s", names[i]);
```

6.3.11.8 bitmap_overlay_add()

See Bitmap Overlays.

Description

The `bitmap_overlay_add()` function is used to create Bitmap Overlays.

Most of the important information about using `bitmap_overlay_add()` is covered in Creating Bitmap Overlays, Bitmap Overlay Colors, and Bitmap Overlay Penstyles.

Usage

Two versions (overloads) of `bitmap_overlay_add` are available. The first is used to create a single rectangle, the second multiple rectangles all of which share the same name and attributes.

```
void bitmap_overlay_add( LPCTSTR name,
                         RECT rect,
                         int penStyle,
                         BOOL filled,
                         COLORREF color,
                         BOOL enabled DEFAULT_VALUE( TRUE ) );
void bitmap_overlay_add( LPCTSTR name,
                         RectArray &rects,
                         int penStyle,
                         BOOL filled,
                         COLORREF color,
                         BOOL enabled DEFAULT_VALUE( TRUE ) );
```

where:

`name` specifies the name of the overlay being created.

`rect` is a (Microsoft) `RECT` structure containing left, top, right, bottom elements of type `LONG`. **`rects`** is an array of `RECT` structures used to define multiple rectangles to be created using the same name and attributes. Single overlays are easily defined by initializing the **`rect`** argument using the Microsoft `CRect()` function as in:

```
RECT r = CRect ( 0, 0, 1, 1);
```

The multiple rectangle version of `bitmap_overlay_add()` was added to support `bitmap_scheme_translate()`. See Using Overlays to Locate Information in BitmapTool. Also see the usage examples in Bitmap Overlay Code:

`penStyle` is one of the following manifest values:

```
PS_SOLID
PS_DASH
PS_DOT
PS_DASHDOT
PS_DASHDOTDOT
```

See Bitmap Overlay Penstyles.

`color` is a `DWORD` value which is best initialized using the `RGB()` macro, which takes three arguments, each in the range of 0-255, to define the red (R), green (G), and blue (B) values used to draw the overlay line(s). `RGB(255,255,255)` results in white, `RGB(0,0,0)` results in black.

`enabled` controls whether the overlay is initially enabled (`TRUE`) or disabled (`FALSE`). When disabled the overlay is not displayed.

Example

This example is rather large, and was included because (a) this is an advanced topic for experiences users and (b) simple examples are easy to create and experiment with. This example includes the following:

- Bitmap Scheme Code representing the Bitmap Overlay Example Device.
- Bitmap Overlay Code which results in the overlays seen in Example BitmapTool Display with Overlays.
- Example Output using Screen-space Coordinates
- Example Output using Screen-space Coordinates

The resulting BitmapTool image is shown in Example BitmapTool Display with Overlays.

Bitmap Scheme Code

This code defines the 16 segments with the row, column, and data permutations shown in the Bitmap Overlay Example Device. It is included here to aid in understanding the relationship between bitmap scheme and bitmap overlays. This example code uses the following functions: `make_bitmap_scheme()`, `add_segment()`, `make_permutation()`, `reverse()`, `register_bitmap_scheme()`

```
void BMT_set_scheme() {
```

```
bitmap_scheme scheme = make_bitmap_scheme( TRUE, TRUE );
add_segment( scheme, 0, 0,
   make_permutation( 64, 0, 1 ), // Rows: 0..63
   make_permutation( 64, 0, 1 ), // Cols: 0..63
   make_permutation( DWID ) );
add_segment( scheme, 0, 1,
   make_permutation( 64, 0, 1 ),    // Rows: 0..63
   make_permutation( 64, 127, -1 ), // Cols: 127..64
   reverse( make_permutation( DWID )) );
add_segment( scheme, 0, 2,
   make_permutation( 64, 0, 1 ),   // Rows: 0..63
   make_permutation( 64, 128, 1 ), // Cols: 128..191
   make_permutation( DWID ) );
add_segment( scheme, 0, 3,
   make_permutation( 64, 0, 1 ),   // Rows: 0..63
   make_permutation( 64, 192, 1 ), // Cols: 192..255
   reverse( make_permutation( DWID )) );
add_segment( scheme, 1, 0,
   make_permutation( 64, 127, -1 ), // Rows: 127..64
   make_permutation( 64, 0, 1 ),    // Cols: 0..63
   reverse( make_permutation( DWID )) );
add_segment( scheme, 1, 1,
   make_permutation( 64, 127, -1 ), // Rows: 127..64
   make_permutation( 64, 127, -1 ), // Cols: 127..64
   make_permutation( DWID ) );
add_segment( scheme, 1, 2,
   make_permutation( 64, 127, -1 ), // Rows: 127..64
   make_permutation( 64, 128, 1 ),  // Cols: 128..191
   reverse( make_permutation( DWID )) );
add_segment( scheme, 1, 3,
   make_permutation( 64, 127, -1 ), // Rows: 127..64
   make_permutation( 64, 192, 1 ),  // Cols: 192..255
   make_permutation( DWID ) );
add_segment( scheme, 2, 0,
   make_permutation( 64, 191, -1 ), // Rows: 191..128
   make_permutation( 64, 0, 1 ),    // Cols: 0..63
   make_permutation( DWID ) );
```

```
   add_segment( scheme, 2, 1,
      make_permutation( 64, 191, -1 ), // Rows: 191..128
      make_permutation( 64, 127, -1 ), // Cols: 127..64
      reverse( make_permutation( DWID )) );
   add_segment( scheme, 2, 2,
      make_permutation( 64, 191, -1 ), // Rows: 191..128
      make_permutation( 64, 128, 1 ),  // Cols: 128..191
      make_permutation( DWID ) );
   add_segment( scheme, 2, 3,
      make_permutation( 64, 191, -1 ), // Rows: 191..128
      make_permutation( 64, 192, 1 ),  // Cols: 192..255
      reverse( make_permutation( DWID )) );
   add_segment( scheme, 3, 0,
      make_permutation( 64, 192, 1 ), // Rows: 192..255
      make_permutation( 64, 0, 1 ),   // Cols: 0..63
      reverse( make_permutation( DWID )) );
   add_segment( scheme, 3, 1,
      make_permutation( 64, 192, 1 ),  // Rows: 192..255
      make_permutation( 64, 127, -1 ), // Cols: 127..64
      make_permutation( DWID ) );
   add_segment( scheme, 3, 2,
      make_permutation( 64, 192, 1 ), // Rows: 192..255
      make_permutation( 64, 128, 1 ), // Cols: 128..191
      reverse( make_permutation( DWID )) );
   add_segment( scheme, 3, 3,
      make_permutation( 64, 192, 1 ), // Rows: 192..255
      make_permutation( 64, 192, 1 ), // Cols: 192..255
      make_permutation( DWID ) );
   register_bitmap_scheme( "SixteenSegs", scheme );
}
```

Bitmap Overlay Code

This code defines the 16 overlay rectangles using the attributes shown in the Bitmap Overlay Example Device. Because the associated bitmap scheme allows it, this code can define the 16 overlays using either screen coordinates or using ECR coordinate values. This is controlled by the boolean argument passed to `BMT_set_overlays()` which affects conditional code within `BMT_set_overlays()()`:

```
#include "tester.h"
#define DWID 8  // Width of data used in screen-space
struct BMT_olay { LPCSTR id; BOOL fill; int pen;
                  int R; int G; int B; };
BMT_olay BMT_olays[] = {
//   ID      Fill    Pen         R    G    B
// ------    -----   --------  ---- ---- ----
  {"Olay0",  FALSE,  PS_SOLID,    0, 255,   0  },
  {"Olay1",  FALSE,  PS_SOLID,    0,   0, 255  },
  {"Olay2",  FALSE,  PS_SOLID,  255,   0, 255  },
  {"Olay3",  FALSE,  PS_SOLID,  255, 255,   0  },
  {"Olay4",  FALSE,  PS_SOLID,  255, 255,   0  },
  {"Olay5",  FALSE,  PS_SOLID,  180, 180, 180  },
  {"Olay6",  FALSE,  PS_SOLID,    0, 255, 255  },
  {"Olay7",  TRUE ,  PS_SOLID,  180, 180, 180  },
  {"Olay8",  FALSE,  PS_SOLID,    0, 255,   0  },
  {"Olay9",  FALSE,  PS_SOLID,    0,   0, 255  },
  {"Olay10", FALSE,  PS_SOLID,  255,   0, 255  },
  {"Olay11", FALSE,  PS_SOLID,  255, 255,   0  },
  {"Olay12", FALSE,  PS_SOLID,  180, 180, 180  },
  {"Olay13", FALSE,  PS_SOLID,  255,   0, 255  },
  {"Olay14", FALSE,  PS_SOLID,    0, 255, 255  },
  {"Olay15", FALSE,  PS_SOLID,  180, 180, 180  }
};

// Create overlay using passed args. Arg-2 determines if
// bitmap_scheme_translate() will process coords before overlay is
// created.
void BMT_add_olay(  LPCTSTR id, BOOL ECR_space,
                    int left, int top, int right, int bottom,
                    int mask, int pen, BOOL fill,
                    int r, int g, int b ) {
   output("  %6s => %4d, %3d, %4d, %3d\\",    id, left, top,
                                              right, bottom );

   RectArray screen;
   int num_rects = 0;
   // Translate to screen coords if O'lay set using ECR-space rect.
   if (ECR_space)
      num_rects = bitmap_scheme_translate( "",
```

```
            CRect( left, top, right, bottom ), mask, &screen );
      else screen.Add (CRect( left, top, right, bottom ));

   for ( int i = 0; i < screen.GetSize(); ++i ) {
      if ( (i > 10) && (i < 254) ) continue;
      RECT rect = screen[ i ];
      if (i==0)
         output(": [%2d] %4d, %4d, %4d, %4d",
                  i, rect.left, rect.top,
                  rect.right, rect.bottom );
      else
         output("%31s [%2d] %4d, %4d, %4d, %4d", ":",
                  i, rect.left, rect.top,
                  rect.right, rect.bottom );
   }
   bitmap_overlay_add( id, screen, pen, fill, RGB( r, g, b ) );
}
// Print the physical R/C/D values for the passed top/left and
// bottom/right corners of each overlay. bitmap_scheme_lookup() is
// only valid when O'lays are set up using screen coords w/o
// bitmap_scheme_translate()
void BMT_print_ECR_space(   int left, int top,
                            int right, int bottom ) {
   DWORD row, col, io;
   output(" L/T:R/B = %4d/%-3d : %4d/%-3d\\",
          left, top, right, bottom );
   BOOL ok = bitmap_scheme_lookup( "", top, left, &row, &col, &io );
   output(" = ECR T/L D%d/R%3d/C%-3d\\", io, row, col );
   // Adjust bottom and right vals -1 to remain within valid ECR
   // space
   ok = bitmap_scheme_lookup( "", (bottom - 1), (right - 1), &row,
                                 &col, &io );
   output(" B/R = D%d/%R%3d/C%-3d", io, row, col );
}

// Set up overlays. Arg determines if ECR rect. or screen coords are
// used. ECR rect. uses values without factoring in where the data
// goes. In BMT_add_olay(), called here, bitmap_scheme_translate()
```

```
// is called to translates ECR rect. to the screen coords required
// by bitmap_overlay_add(), which is also executed in
// BMT_add_olay(). This code ASSUMES data with columns !!!
void BMT_set_overlays( BOOL ECR_space ) {
   output("\n  BMT_set_overlays(%s)",
          ECR_space ? "ECR" : "Screen");
   output("  (LTRB)         Inputs          : Rectangle");
   // Set new scheme in BitmapTool
   remote_set( "ui_CurrentBitmapScheme", "SixteenSegs", -1 );
   int dwidth;
   if( ! ECR_space ) dwidth = DWID;
   else dwidth = 1;
   int rows_per_seg = 64;
   int cols_per_seg = 64;
   int num_V_segs = 4;
   int num_H_segs = 4;
   int i = 0; // Array index to info for each segment
   int row_start = 0; int col_start = 0;
   for( int segrow = 0; segrow < num_V_segs; segrow++ ) {
      col_start = 0; // Reset @ start of each vertical segment
      for( int segcol = 0; segcol < num_H_segs; segcol++ ) {
         BMT_add_olay(  BMT_olays[i].id,
                        ECR_space,
                        col_start, row_start,
                        (col_start + (cols_per_seg * dwidth)),
                        (row_start + rows_per_seg),
                        0xFF,
                        BMT_olays[i].pen,
                        BMT_olays[i].fill,
                        BMT_olays[i].R,
                        BMT_olays[i].G,
                        BMT_olays[i].B );
           col_start += (cols_per_seg * dwidth);
           i++;
         }
         row_start += rows_per_seg;
   }
   // If screen coords were used, call bitmap_scheme_lookup() to
   // print the screen R/C/D values at the top/left and bottom/right
   // corners of each overlay.
```

```
   if( ! ECR_space ) {
      output("\n Overlays were set using Screen coords.\\");
      output(" Print ECR R/C/D of T/L and B/R corners");
      row_start = 0;
      for( int segrow = 0; segrow < num_V_segs; segrow++ ) {
         col_start = 0;
         for( int segcol = 0; segcol < num_H_segs; segcol++ ) {
            BMT_print_ECR_space(   col_start,
                                   row_start,
                              (col_start + (cols_per_seg * dwidth)),
                                   row_start + rows_per_seg );
            col_start += (cols_per_seg * dwidth);
         }
         row_start += rows_per_seg;
      }
   }
}
SITE_BEGIN_BLOCK( SB1 ) {
   // ... other code here
   BMT_set_scheme();
   BMT_set_overlays( TRUE ); // TRUE = use DUT coords
   // ... other code here
}
```

Example Output using Screen-space Coordinates

This output is generated by code in both `BMT_set_overlays()` and `BMT_add_olay()` above.

```
BMT_set_overlays(Screen)
  (LTRB)         Inputs          : Rectangle
   Olay0 =>    0,   0,  512,  64: [ 0]    0,    0,  512,   64
   Olay1 =>  512,   0, 1024,  64: [ 0]  512,    0, 1024,   64
   Olay2 => 1024,   0, 1536,  64: [ 0] 1024,    0, 1536,   64
   Olay3 => 1536,   0, 2048,  64: [ 0] 1536,    0, 2048,   64
   Olay4 =>    0,  64,  512, 128: [ 0]    0,   64,  512,  128
   Olay5 =>  512,  64, 1024, 128: [ 0]  512,   64, 1024,  128
   Olay6 => 1024,  64, 1536, 128: [ 0] 1024,   64, 1536,  128
   Olay7 => 1536,  64, 2048, 128: [ 0] 1536,   64, 2048,  128
   Olay8 =>    0, 128,  512, 192: [ 0]    0,  128,  512,  192
   Olay9 =>  512, 128, 1024, 192: [ 0]  512,  128, 1024,  192
```

```
  Olay10 => 1024, 128, 1536, 192: [ 0] 1024,  128, 1536,  192
  Olay11 => 1536, 128, 2048, 192: [ 0] 1536,  128, 2048,  192
  Olay12 =>    0, 192,  512, 256: [ 0]    0,  192,  512,  256
  Olay13 =>  512, 192, 1024, 256: [ 0]  512,  192, 1024,  256
  Olay14 => 1024, 192, 1536, 256: [ 0] 1024,  192, 1536,  256
  Olay15 => 1536, 192, 2048, 256: [ 0] 1536,  192, 2048,  256
```

This output is generated primarily by code in `BMT_print_DUT_coords()` above.

```
Overlays were set using Screen coords. Print ECR R/C/D of T/L and
B/R corners
 L/T:R/B =   0/0   :  512/64  = ECR T/L D0/R  0/C0   B/R = D7/R 63/C63
 L/T:R/B =  512/0   : 1024/64  = ECR T/L D7/R  0/C127 B/R = D0/R
63/C64
 L/T:R/B = 1024/0   : 1536/64  = ECR T/L D0/R  0/C128 B/R = D7/R
63/C191
 L/T:R/B = 1536/0   : 2048/64  = ECR T/L D7/R  0/C192 B/R = D0/R
63/C255
 L/T:R/B =    0/64  :  512/128 = ECR T/L D7/R127/C0   B/R = D0/R
64/C63
 L/T:R/B =  512/64  : 1024/128 = ECR T/L D0/R127/C127 B/R = D7/R
64/C64
 L/T:R/B = 1024/64  : 1536/128 = ECR T/L D7/R127/C128 B/R = D0/R
64/C191
 L/T:R/B = 1536/64  : 2048/128 = ECR T/L D0/R127/C192 B/R = D7/R
64/C255
 L/T:R/B =    0/128 :  512/192 = ECR T/L D0/R191/C0   B/R = D7/
R128/C63
 L/T:R/B =  512/128 : 1024/192 = ECR T/L D7/R191/C127 B/R = D0/
R128/C64
 L/T:R/B = 1024/128 : 1536/192 = ECR T/L D0/R191/C128 B/R = D7/
R128/C191
 L/T:R/B = 1536/128 : 2048/192 = ECR T/L D7/R191/C192 B/R = D0/
R128/C255
 L/T:R/B =    0/192 :  512/256 = ECR T/L D7/R192/C0   B/R = D0/
R255/C63
 L/T:R/B =  512/192 : 1024/256 = ECR T/L D0/R192/C127 B/R = D7/
R255/C64
 L/T:R/B = 1024/192 : 1536/256 = ECR T/L D7/R192/C128 B/R = D0/
R255/C191
 L/T:R/B = 1536/192 : 2048/256 = ECR T/L D0/R192/C192 B/R = D7/
R255/C255
```

Example Output using ECR Rectangle Coordinates

```
BMT_set_overlays(ECR)
  (LTRB)        Inputs        : Rectangle
   Olay0 =>    0,   0,   64,  64: [ 0]    0,    0,  512,   64
   Olay1 =>   64,   0,  128,  64: [ 0]  512,    0, 1024,   64
   Olay2 =>  128,   0,  192,  64: [ 0] 1024,    0, 1536,   64
   Olay3 =>  192,   0,  256,  64: [ 0] 1536,    0, 2048,   64
   Olay4 =>    0,  64,   64, 128: [ 0]    0,   64,  512,  128
   Olay5 =>   64,  64,  128, 128: [ 0]  512,   64, 1024,  128
   Olay6 =>  128,  64,  192, 128: [ 0] 1024,   64, 1536,  128
   Olay7 =>  192,  64,  256, 128: [ 0] 1536,   64, 2048,  128
   Olay8 =>    0, 128,   64, 192: [ 0]    0,  128,  512,  192
   Olay9 =>   64, 128,  128, 192: [ 0]  512,  128, 1024,  192
  Olay10 =>  128, 128,  192, 192: [ 0] 1024,  128, 1536,  192
  Olay11 =>  192, 128,  256, 192: [ 0] 1536,  128, 2048,  192
  Olay12 =>    0, 192,   64, 256: [ 0]    0,  192,  512,  256
  Olay13 =>   64, 192,  128, 256: [ 0]  512,  192, 1024,  256
  Olay14 =>  128, 192,  192, 256: [ 0] 1024,  192, 1536,  256
  Olay15 =>  192, 192,  256, 256: [ 0] 1536,  192, 2048,  256
```

6.3.11.9 bitmap_overlay_delete()

See Bitmap Overlays.

Description

The `bitmap_overlay_delete()` function is used to delete a named Bitmap overlay.

Usage

```
void bitmap_overlay_delete( CString name );
```

where **name** specified the Bitmap overlay to be deleted.

Example

```
bitmap_overlay_add("myOverlay",... ); // Create it
bitmap_overlay_delete("myOverlay");   // Delete it
```

6.3.11.10 bitmap_overlay_lookup()

Description

The `bitmap_overlay_lookup()` function can be used to look-up the following attributes of a named Bitmap overlay:

- Pen style (line type). See Bitmap Overlay Penstyles.
- Filled or not-filled. See Creating Bitmap Overlays.
- Color. See Bitmap Overlay Colors
- Enabled state. See Creating Bitmap Overlays.

Note that it is not possible to lookup the location/size attributes of an overlay because it may consist of many rectangles described in screen-space, which is not particularly useful information.

Usage

```
void bitmap_overlay_lookup( LPCTSTR name,
                            int *penStyle,
                            BOOL *filled,
                            COLORREF *color,
                            BOOL *enabled );
```

where:

`name` specifies the Bitmap overlay of interest.

`penStyle` is a pointer to an existing `integer` variable used to return the pen style attribute, which will be one of:

```
PS_SOLID
PS_DASH
PS_DOT
PS_DASHDOT
PS_DASHDOTDOT
```

See Bitmap Overlay Penstyles.

`filled` is a pointer to an existing `BOOL` variable used to return whether the overlay was drawn filled or not-filled.

color is a pointer to an existing COLORREF variable used to return the attribute of the overlay. The returned value can best be decomposed by referring to the definition of the RGB( ) macro in Visual C++ *wingdi.h* include file.

enabled is a pointer to an existing BOOL variable used to return the enabled state of the overlay.

Example

```
bitmap_overlay_add( "myOverlay",
                    CRect(0,0,1,1),
                    PS_SOLID, TRUE, RBG(255,255,255) );
   int penStyle;
   BOOL filled;
   COLORREF color;
   BOOL enabled;
   bitmap_overlay_lookup( "myOverlay", &penStyle, &filled,
                          &color, &enabled );
   output( "penStyle = %d, filled = %d, color = 0x%x, enabled = %d",
        penStyle, filled, color, enabled );
```

6.3.11.11 bitmap_overlay_setup()

See Bitmap Overlays.

Description

The bitmap_overlay_setup( ) function is used to modify the following attributes of an existing Bitmap overlay.

- Pen style (line type). See Bitmap Overlay Penstyles.
- Filled or not-filled. See Creating Bitmap Overlays.
- Color. See Bitmap Overlay Colors
- Enabled state. See Creating Bitmap Overlays.

Note that it is not possible to modify the location/size attributes of an overlay.

Usage

```
void bitmap_overlay_setup(  LPCTSTR name,
                            int penStyle,
                            BOOL filled,
                            COLORREF color,
                            BOOL enabled );
```

where:

name specifies the Bitmap overlay of interest.

penStyle is the desired pen style attribute, which must be one of:

```
PS_SOLID
PS_DASH
PS_DOT
PS_DASHDOT
PS_DASHDOTDOT
```

See Bitmap Overlay Penstyles.

filled specifies whether the overlay was drawn filled (TRUE) or not-filled (FALSE).

color is a DWORD value which is best initialized using the RGB( ) macro, which takes three arguments, each in the range of 0-255, to define the red (R), green (G), and blue (B) values used to draw the overlay line(s). RGB(255,255,255) results in white, RGB(0,0,0) results in black.

enabled controls whether the overlay is in it ally enabled (TRUE) or disabled (FALSE). When disabled the overlay is not displayed.

Example

This example first creates a Bitmap overlay named "myOverlay", then modifies the attributes noted above:

```
bitmap_overlay_add( "myOverlay",
                    CRect(0,0,1,1),
                    PS_SOLID, TRUE, RBG(255,255,255) );
bitmap_overlay_setup(  "myOverlay", PS_DOT, FALSE,
                       RBG(255,127,127), FALSE );
```

6.3.11.12 bitmap_overlay_enable()

See Bitmap Overlays.

Description

The `bitmap_overlay_enable()` function is used to enable (display) or disable (hide) an existing Bitmap overlay.

Usage

```
void bitmap_overlay_enable( LPCTSTR name, BOOL enabled );
```

where:

`name` specifies the Bitmap overlay of interest.

`enabled` controls whether the overlay is enabled (`TRUE`) or disabled (`FALSE`).

Example

```
bitmap_overlay_add( "myOverlay",
                    CRect(0,0,1,1),
                    PS_SOLID, TRUE, RBG(255,255,255) );
bitmap_overlay_enable(" myOverlay", FALSE );
```

6.3.11.13 bitmap_overlay_draw()

See Bitmap Overlays.

Description

The `bitmap_overlay_draw()` function is used to cause a named Bitmap overlay to flash, by repeatedly causing just the specified overlay to be painted. Since no attributes are changed, each time the overlay is painted the color used is XOR'ed with the previous image, and since any given color XOR'ed with itself results in black the affect is to cause the overlay to flash. See Bitmap Overlay Colors.

Usage

```
void bitmap_overlay_draw( LPCTSTR name );
```

where **name** is the Bitmap overlay to draw.

Example

The following code can be included in the test program to support flashing multiple Bitmap overlays.

```
// CList used to store O'lay names to be flashing.
// Use BMT_set_olay_flash_mode() to change the contents
// of this flasher_list to change which O'lays flash.
// Any O'lay name in this list will flash. Must be global.
CList< CString, LPCTSTR > flasher_list;

// Invoke this using remote_set() to start thread to flash O'lays
VOID_VARIABLE( BMT_olay_flasher, "" ) {
   while ( 1 ) { // No need to terminate once started.
      // For each O'lay in this list, make it flash in BMT
      for ( POSITION pos = flasher_list.GetHeadPosition(); pos; )
         bitmap_overlay_draw( flasher_list.GetNext( pos ) );
      Sleep( 500 ); // Time constant = 0.5 seconds
   }
}

// Add/remove one named O'lay from flasher_list.
// If in the list, that O'lay will flash. Also starts
// the thread which causes the flashing.
void BMT_set_olay_flash_mode ( CString overlay, BOOL enable ) {
   // Only Start the flash thread once !!! IMPORTANT
   static BOOL started = FALSE;
   if ( ! started ) {
      remote_set( BMT_olay_flasher, "", site_num() );
      started = TRUE;
   }

   // Always remove specified O'lay from the list first
   // This disables it.
   POSITION pos = flasher_list.Find( overlay );
   if ( pos ) flasher_list.RemoveAt( pos );
```

```
   // Put it back in list only if enabled
   if ( enable ) flasher_list.AddTail( overlay );
}
SITE_BEGIN_BLOCK( SB1 ) {
   // ... other code here
   BMT_set_olay_flash_mode( "myOverlay", TRUE );// Do flash
   BMT_set_olay_flash_mode( "otherOverlay", FALSE ); // No flash
   // ... other code here
}
```

6.4 Breakpoint Monitor

This section includes the following:

Note: the Run Buttons were added in software release h2.2.7/h1.2.7.

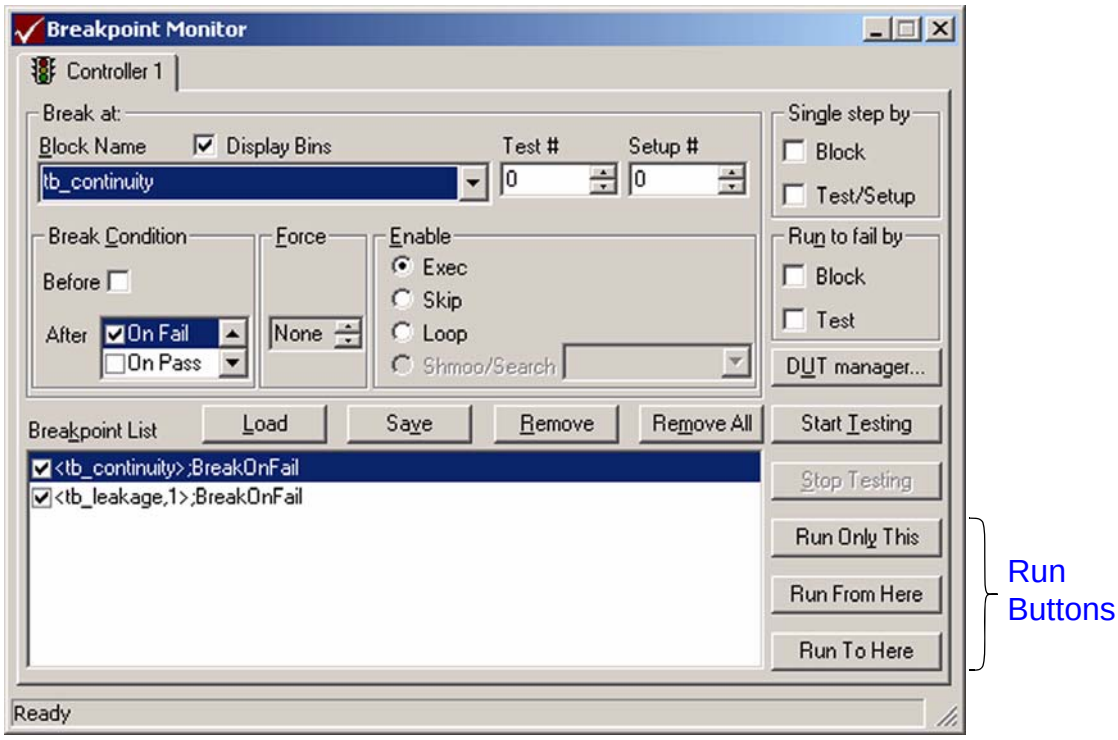
6.4.1 Overview

See Breakpoint Monitor.

The Breakpoint monitor is one of the most commonly used tools for test program debugging. Quite often, other interactive tools such as Voltage and Current Tool or TimingTool are usable only after setting a breakpoint using the Breakpoint monitor.

The Breakpoint monitor allows the user to set one or more breakpoints in the test program under different conditions, execute the Sequence & Binning Table, and perform various

actions when a breakpoint is reached:



The **`Block Name`** combo box contains a list of all the test block names in the order they appear in the sequence table. It is possible for the same name to appear more than once if the same test block is performed multiple times in the test flow.

`Test#` is an editable field used to enter a `Test Numbers` (0-999). The test number value is incremented by any of the Nextest functions which execute a test i.e. `funtest()`, `partest()`, `ac_partest()`, `test_supply()`, `ac_test_supply()`.

Note: the first test in Test Blocks has a test number of 1, regardless of whether a breakpoint is set to break `Before` or not.

`Setup#` is an editable field used to enter a `Setup Numbers` (0-999). Setup numbers are incremented by any Nextest function (as documented in this manual) except the test functions. Examples of setup functions are `vih()`, `cycle()`, `edge_strobe()`, etc.

Note: the first setup number in Test Blocks is 1, regardless of whether a breakpoint is set to break `Before` or not.

Breakpoints can be set at Test Blocks, Test Numbers, and Setup Numbers.

- To set a breakpoint at a test block, choose the appropriate test block name. Set both *Test#* and *Setup#* to 0.
- To set a breakpoint at a test in a test block, choose the appropriate test block name. Set *Test#* to a number greater than 0, and set *Setup#* to 0.
- To set a breakpoint at a setup in a test block, choose the appropriate test block name. Set *Setup#* to a number greater than 0. Each time a *Test#* is incremented the *Setup#* is reset to 0, thus it may be necessary to set the *Test#* to a value greater than one also.

The **DUT manager** button is used to invoke the DUT Manager dialog, which can be used in Magnum 1/2/2x Multi-DUT Test Programs to interactively move DUT(s) in or out of the Ignored DUTs Set (IDS). See Magnum 1, 2 & 2x Parallel Test.

6.4.2 Starting the Breakpoint Monitor

See Breakpoint Monitor.

To start the Breakpoint monitor:

- Click on the Breakpoint monitor icon from the *Ui* toolbar 
- Type keyboard shortcut **`Ctrl+K`**
- Choose **`Tools: Breakpoint`**…
- From the *Sequence View* in UI. See Starting the Breakpoint Monitor.

6.4.3 Breakpoint Attributes

See Breakpoint Monitor.

Attributes are the conditions under which the breakpoint is set. This is dependent on the scope of the breakpoint. For example, a setup breakpoint does not have a *break after on pass* or *break after on fail* context, since setup does not have any pass/fail outcome.

The Breakpoint attributes are:

`Before` the test block / *Test#* / *Setup#* depending on the scope.

`After` the test block / *Test#* **`On Pass`**.

`After` the test block / *Test#* **`On Fail`**.

`After` the *Setup#*.

None of the attributes are set by default.

The Breakpoint monitor supports breaking on an integer value test result, in addition to **`On Pass`** (1) and **`On Fail`** (0). This supports the macros TEST3, TEST4, etc. to make it possible to break only when a test block returns integer values other than (for example: 0 through 4). It is now possible to choose any combination of numbers between 0 and 63 inclusive. For example, it is possible to set *Break After* on integer return values of 4, 7, 9, and 63.

It is possible to set *Break After* and *Loop* on a *TestBin* just as you can do with a *TestBlock*. The *Break Before* option also continues to work as expected. Looping on a TestBin causes any body code of that bin to execute repeatedly.

6.4.4 Breakpoint Actions

See Breakpoint Monitor.

Actions are things the user wants the breakpoint monitor to perform once the test flow reaches the breakpoint. The actions are:

`Force` *Pass* / *Fail* if the scope is a test block /*Test#*. None is the default.

`Enable` *Skip* / *Loop*. The default is *Exec* (normal execution). The scope is test block / *Test#* / *Setup#*. For example, you can *Loop* on a test or an entire test block depending on the kind of breakpoint.

A breakpoint is added to the **`Breakpoint List`** automatically if it has a node number (invisible to the user), indicated by the test block name, *Test#* and *Setup#* combination. Breakpoint should have at least one non-default *attribute* or *action*.

Multiple breakpoints can be added or removed from the list. Once you select a breakpoint from the list, the display is updated with the proper **`Block Name`**, **`Test#`**, **`Setup#`**, **`Break Condition`**, **`Force`** and **`Enable`** actions. The attribute and action fields could be modified and the list item will be updated accordingly.

It is possible to set *Break After* and *Loop* on a *TestBin* just as you can do with a *TestBlock*. The *Break Before* option also continues to work as expected. Looping on a TestBin causes any body code of that bin to execute repeatedly.

6.4.5 Breakpoint Removal

See Breakpoint Monitor.

A single, several or all set breakpoints can be removed. There are three ways to remove breakpoints.

- Select one or more breakpoints and click on the **Remove** button.
- Select a breakpoint from the list and modify the attribute and action to be the default one.
- Click the **Remove All** button to remove all breakpoints.

A breakpoint can be ***temporarily disabled*** by un-checking the checkbox in the corresponding list entry. To enable the breakpoint, simply check it again. This avoids the necessity of recreating the breakpoint from scratch. For example, you are looping on a test unconditionally and you want to get out of the loop. Un-checking the checkbox will stop the loop. The box could be checked again if looping is necessary later.

6.4.6 Breakpoint Definition File

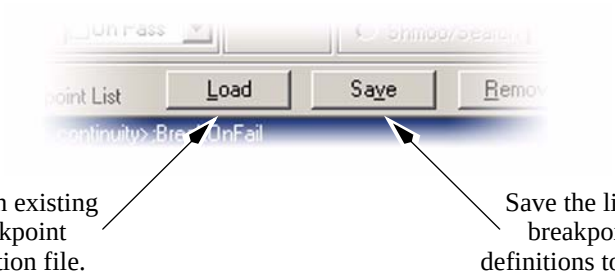
See Breakpoint Monitor.

Note: the location of these controls were changed in software release h2.2.7/h1.2.7. The images below show the later locations. The operation of the controls did not change.

The Breakpoint Monitor tool provides a method for saving breakpoint definitions to a file on disk.

Note: changes made using the DUT Manager dialog, are not saved.

A breakpoint definition file is created, or replaced, using the **Save** button in Breakpoint Monitor. A breakpoint definition file is loaded into the Breakpoint Monitor using the **Load** button:



In either case, a standard Windows file browser is displayed, and used to select the desired disk, path, and file.

The `ui_BreakPointFile` UI user variable (see UI User Variables) can be used to specify a breakpoint definition file when using a command line or batch file (see `ui_BatchFile`).

Note the following:

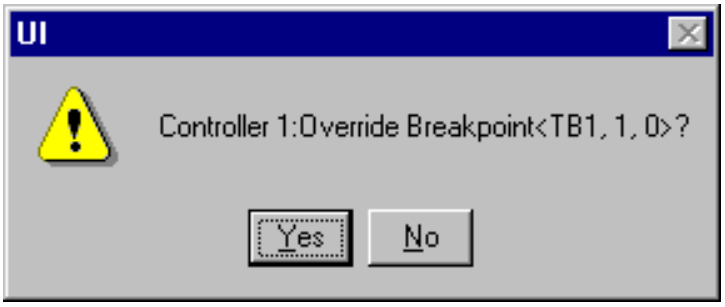
- The information is stored in ASCII files (**.txt*), which are readable using any text editor.

Note: Nextest reserves the right to modify the format of these files at any time. Manually editing these files is therefore discouraged.

- The **Save** operation saves all the definitions shown in the Breakpoint List box. It is not possible to save a subset.
- The **Load** operation loads the entire contents of the selected file.

When **Load** is invoked, the user is prompted for additional inputs under the following conditions:

- The Breakpoint List already contains a breakpoint which conflicts with one in the specified breakpoint definition file. A dialog similar to the following will be displayed:



Selecting **Yes** causes the breakpoint in the definition file to be used. Selecting **No** causes the breakpoint in the Breakpoint List be to be used.

- When the breakpoint definition file references one or more test blocks which do not exist in the test program. A dialog similar to the following will be displayed:

This is not fatal, but Breakpoint List should be reviewed and validated.

6.4.7 Single-stepping

See Breakpoint Monitor.

Using Breakpoints, single-stepping through program code and tests can be enabled at any time by checking either or both of the **Single step by** check boxes.

Block: Same as setting **Break Before** attribute for all test blocks.

Test/Setup: Same as setting **Break Before** attribute for all *Test#* and *Setup#*.

6.4.8 Run to Fail

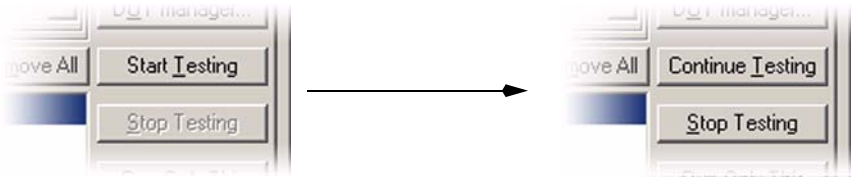
See Breakpoint Monitor.

Block: Same as setting the **Break After On fail** attribute for all test blocks.

Test: Same as setting the **Break After On fail** attribute for all *Test#*.

Start Testing button can be clicked to start the test flow. This button is identical to the **Start Testing** item in the File menu. The keyboard shortcut is also the same (**Ctrl+T**). Once the test program reaches a breakpoint, the following things will happen:

1. It will select the appropriate breakpoint from the list.
2. The sequence table view will highlight the appropriate test block.
3. The output window will display a notification message in the appropriate controller window, indicating the nature of the breakpoint the test program has reached.
4. The **Start Testing** button will change to **Continue Testing** button and the **Stop Testing** button will be enabled:



5. The status bar of *Ui* and *Breakpoint monitor* will indicate that the test program is at a breakpoint.
6. The test time stopwatch, if previously enabled from the View menu, will stop.

To continue execution from the breakpoint, click the **Continue Testing** button. This will start the test time stopwatch if Test Time option was enabled from the View menu. The Stop Testing button is the same as in the File menu.

Notes:

1. Breakpoints set in this manner have no bearing on, and are not to be confused with, the source line breakpoint set in the debugging session from *Microsoft Developer Studio*.
2. Breakpoints are remembered in the current loading of the test program. The *Breakpoint monitor* can be dismissed after the breakpoints are set. The *Breakpoint monitor* pops up automatically when a breakpoint is reached, during the execution of the test program.

3. Using controller tabs to set different breakpoints in different controllers may cause unpredictable results in the case of multiple test site controllers. It is recommended to debug one controller at a time. Other controllers can be disabled through the **Option** item of the **Tools** menu as described later.
4. The **Stop Testing** button is active if the test program state (indicated in the status bar) is *Running* or *Looping*. At present, clicking this button may cause unpredictable results, if the program is not looping.

6.4.9 Breakpoint Usage

See Breakpoint Monitor.

This section includes the following:

6.4.9.1 Breakpoints on Test Functions

See Breakpoint Usage, Breakpoint Monitor.

Breakpoints, single-stepping, and looping can be performed on all Nextest-created C functions described in this manual. Breakpoints are treated slightly differently for C functions that execute actual tests of the DUT. These functions are:

```
funtest()
partest()
ac_partest()
test_supply()
ac_test_supply()
hv_test_supply()
hv_ac_test_supply()
```

`ptu_partest()`

`ptu_partest()`

Breakpoints can be set on any of the above functions, to halt execution either just before or just after the function executes. These Breakpoints are identified by Test Block name and Test Numbers.

6.4.9.2 Breakpoints on C Functions

See Breakpoint Usage, Breakpoint Monitor.

Set-up of DUT conditions (timing, voltage, etc.) is done by calling Nextest-created C-functions, and is typically done prior to executing a test function. Breakpoints can be set on these non-test C-functions to halt execution either just before or just after the function executes.

These Breakpoints are identified by Test Block name and Setup Numbers. The setup number counter starts at zero when a test block is entered and increments each time a Nextest-created (non-test) C-function executes. User-created C-functions do not increment the set-up counter. Also, each time one of the test functions executes, the set-up counter is set to zero by the system software. Setup C-functions are identified for breakpointing based on:

- Test block name
- Test number
- Setup number

where:

`Test block name` is the user-defined name for the test block

`Test number` is an integer representing the test number for the most recently executed test function

`Setup number` is an integer for the set-up function number defined by the set-up function counter. The test number is zero until the first test function is executed in the test block.

Example

Assume the following test block is from a user test program.

```
TEST_BLOCK( speed_test ) {
   dps( 3.3 V, vcc );
   vih( 2.0 V );
   vil( 0.8 V );
   cycle( TSET2, 40 NS );
   settime( TSET2, io_pins, STROBE, 28 NS, 36 NS );
   if ( funtest( minmax, error ) == FAIL ) return FAIL;
   lport( 0, 4 );
   user_written_function();
   lbdata( 5 );
   dps( 2.7 V, vcc );
   return ( funtest( minmax, error ));
}
```

The C-functions in the example test block above are listed below along with their identifiers that are used for breakpointing.

Table 6.4.9.2-1 Example Breakpoint Test/Setup Numbers

Function	Test Block	Test #	Setup #
dps()	speed_test	0	1
vih()	speed_test	0	2
vil()	speed_test	0	3
cycle()	speed_test	0	4
settime()	speed_test	0	5
funtest()	speed_test	1	n/a (gets set to zero though)
lport()	speed_test	1	1
lbdata()	speed_test	1	2
dps()	speed_test	1	3
funtest()	speed_test	2	n/a (gets set to zero though)

Note that the `user_written_function()` does not appear in this list. It is not possible to set a breakpoint at a user-written function using Breakpoint Monitor. A more advanced (and complex) breakpoint capability allows setting breakpoints on any C-function. This is a standard part of the *Microsoft Developer Studio* capabilities. See Debugging With Developer Studio.

6.4.9.3 Breakpoint Macros

See Breakpoint Usage, Breakpoint Monitor.

Two macros, `TEST_BREAKPOINT()`, and `SETUP_BREAKPOINT()`, allow user-written C-code to interact with the Breakpoint Monitor and ShmooTool / SearchTool.

These macros are the same ones used by Nextest software. When a *test* (`funtest()`, `partest()`, etc.) increments a `test_number()` it is using the `TEST_BREAKPOINT()` macro. Similarly, when the other Nextest functions increments a `setup_number()` it is using the `SETUP_BREAKPOINT()` macro. Basically, using these macros allows user-written code to interact with the Breakpoint Monitor and ShmooTool / SearchTool the same as Nextest-written software.

These macros have the following key features

- User-written body code is *wrapped* by the macro. This body code will execute normally (unconditionally), but can also be executed under control of the Breakpoint Monitor. When used with ShmooTool / SearchTool, the body code executes for each shmoo plot point, and determines whether each point passes or fails.
- Some of the Nextest functions which access hardware will not trigger a breakpoint in simulation mode. However, user-written code in the body of these macros will execute in simulation mode.
- Like `funtest()`, `partest()`, etc. the `TEST_BREAKPOINT()` macro causes the `test_number()` to be incremented. This allows the Breakpoint Monitor to conditionally stop program execution before the macro body code executes (Break before test number **n**), or based on the value of an integer control variable set within the macro body code (Break after on pass, fail, or value **n**). This control variable is specified as an argument to the macro, which creates the variable and initializes it to 0. The last value assigned to the control variable within the macro body code is evaluated by the Breakpoint Monitor. When ShmooTool / SearchTool is used at a breakpoint controlled by `TEST_BREAKPOINT()`, it is the last value assigned to the control variable which determines whether a given point in a shmoo is displayed as fail (0) or pass (not 0);
- Like all Nextest-written functions (except test functions) the `SETUP_BREAKPOINT()` macro causes the `setup_number()` to be incremented. This allows the Breakpoint Monitor to conditionally stop program execution before or after the macro body code executes.

- Using either macro, it is possible to execute the macro body code in a loop. Options set in the Breakpoint Monitor make it possible to loop continuously (unconditionally) or, if using `TEST_BREAKPOINT()`, to loop until the control variable matches pass, fail, or a specific value.
- Unlike other Nextest-defined macros, these macros are *not* global in scope, and are only usable within Test Blocks, or functions called from test blocks.
- It is legal to nest these macros, however, a somewhat unusual, but simple, syntax is needed. See next item.

Usage

```
TEST_BREAKPOINT( control_var ) {
// Optional macro body code here
 control_var = 0; // Last value set can trigger breakpoint
// or determine shmoo pass/fail per-point
// Optional macro body code here
}
SETUP_BREAKPOINT {
// Macro body code here
}
```

Examples

Example 1:

The test block below is designed to support looping the commands. This allows using an oscilloscope to view the resulting LPORT output waveforms as they occur.

```
TEST_BLOCK( TB3 ) {
   dps ( 5 V, vcc );
   TEST_BREAKPOINT( val ){ // test number = 1
      output (" val => %d", val );
      val = 10; // Set breakpoint count to any value >0.
   }
   return ( funtest (patname, error ) );
}
```

The way the `val` control variable is used allows Breakpoint Monitor to be set to break on any number of loop iterations. Or, the loop can be executed continuously.

In normal use, the macro body code executes once. Using the Breakpoint Monitor, if a breakpoint is set on test number = 1, break on 10, Loop, when Start Test is invoked the macro body code will execute 9 times, then halt. The loop stops on entry to the 10th

iteration. If the breakpoint is set as test number = 1, Loop, Start Test will continuously loop the macro body code until the Stop Testing button is clicked.

Note that to loop on any single function the TEST_BREAKPOINT macro is not really needed; use a standard breakpoint triggered on Setup Numbers instead. Conversely, the only way to loop on just the three lport() commands requires that either TEST_BREAKPOINT or SETUP_BREAKPOINT be used.

Example 2:

In this example the SETUP_BREAKPOINT macro body code calls both a Nextest function and a user-written C function. In normal use, the macro body code executes once. However, using the Breakpoint Monitor it is possible to halt execution before or after the macro body code executes, or to loop on the macro body code. When setting the breakpoint the setup_number() of the macro will be 1 because the previous funtest() reset the setup number to 0.

```
TEST_BLOCK( speed_test ) {
   dps( 3.3 V, vcc );
   vih( 2.0 V );
   vil( 0.8 V );
   cycle( TSET2, 40 NS );
   settime( TSET2, io_pins, STROBE, 28 NS, 36 NS );
   if ( funtest( minmax, error ) == FAIL ) return FAIL;
      SETUP_BREAKPOINT {
         user_written_function(); // Optional
      }
   lbdata( 5 );
   dps( 2.7 V, vcc );
   return ( funtest( minmax, error ) );
}
```

The various C functions in the example above are shown below along with the test numbers and setup numbers which can be used to control breakpoint execution:

Table 6.4.9.3-1 Example Test/Setup Numbers

Function	Test block	Test #	Setup #
dps()	speed_test	0	1
vih()	speed_test	0	2
vil()	speed_test	0	3

Table 6.4.9.3-1 Example Test/Setup Numbers *(Continued)*

Function	Test block	Test #	Setup #
cycle()	speed_test	0	4
settime()	speed_test	0	5
funtest()	speed_test	1	Reset to zero
SETUP_BREAKPOINT	speed_test	1	1
lbdata()	speed_test	1	3
dps()	speed_test	1	4
funtest()	speed_test	2	Reset to zero
Next Test Block	block name	0	0

6.4.9.4 Looping and Single-stepping

See Breakpoint Usage, Breakpoint Monitor.

Breakpoints, single-stepping, and looping can be performed on all user level C-functions described in this manual.

This includes the test execution functions like funtest() and partest() as well as setup functions like settime(), vih(), lbdata(), etc. C-functions written by the user do not automatically have breakpoint, single-step, or looping capability. These capabilities can be added to user-written functions by adding the Breakpoint Macros to user-written functions.

Using the Breakpoint dialog, a breakpoint can be set on C-functions described in this manual. Once execution stops at the breakpoint, C-function single-step may be selected. The modes available for C function single-step are:

- Break before the C-function executes
- Break after the C-function executes
- Break both before and after the C-function executes

Each time the **Single Step** button is pushed in the Breakpoint dialog, test program execution continues from the current location to the next single-step break location, as defined by the single-step mode. This allows the user to step from one tester C function to the next at the push of a button.

While stopped at a Breakpoint, the **Loop** button can be pushed in the Breakpoint dialog to initiate continuous looping on the current function. Pushing the **Stop Looping** button causes looping to stop. The **Single Step** button can be pushed to continue single-stepping through tester C-functions from the current location. If looping is active when the **Single Step** button is pushed, looping is automatically terminated before stepping to the next single-step break location.

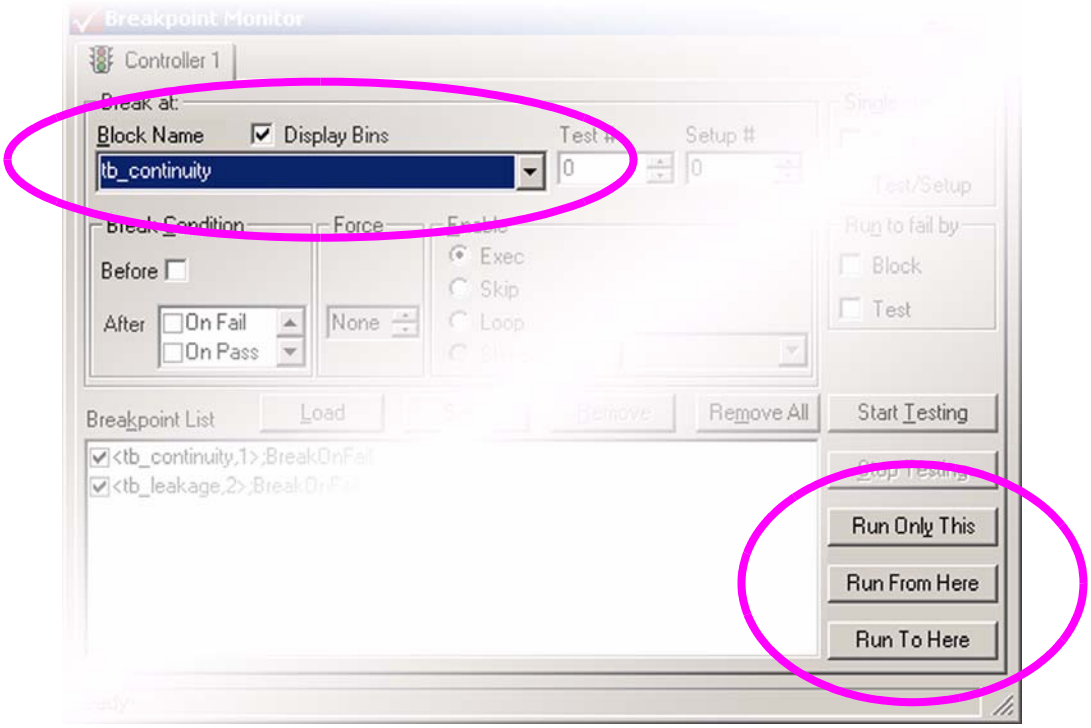
When a functional pattern is looping two signals are available on the DUT Board which are useful when using an oscilloscope to evaluate timing signals and failing strobes.

6.4.10 Run Buttons

Note: first available in software release h2.2.7/h1.2.7.

The three Run buttons each cause the execution of some portion of the Sequence & Binning Table. These are the same options available using the right-mouse context menu in the UI Sequence and Binning sub-window.

All three options begin with the test block currently selected in the **Block Name** menu:



- **Run Only This** - execute the selected test block.
- **Run From Here** - execute the Sequence & Binning Table starting with the selected test block.
- **Run To Here** - execute the Sequence & Binning Table up to and including the selected test block.

Note the following:

- All three options always execute the specified test block.
- Test block execution begins with the first line of user-written code in the block, regardless of any values selected for **Test#** or **Setup#**.
- Breakpoint operation is the same regardless of how a given test block execution is invoked.

6.5 DBMTool

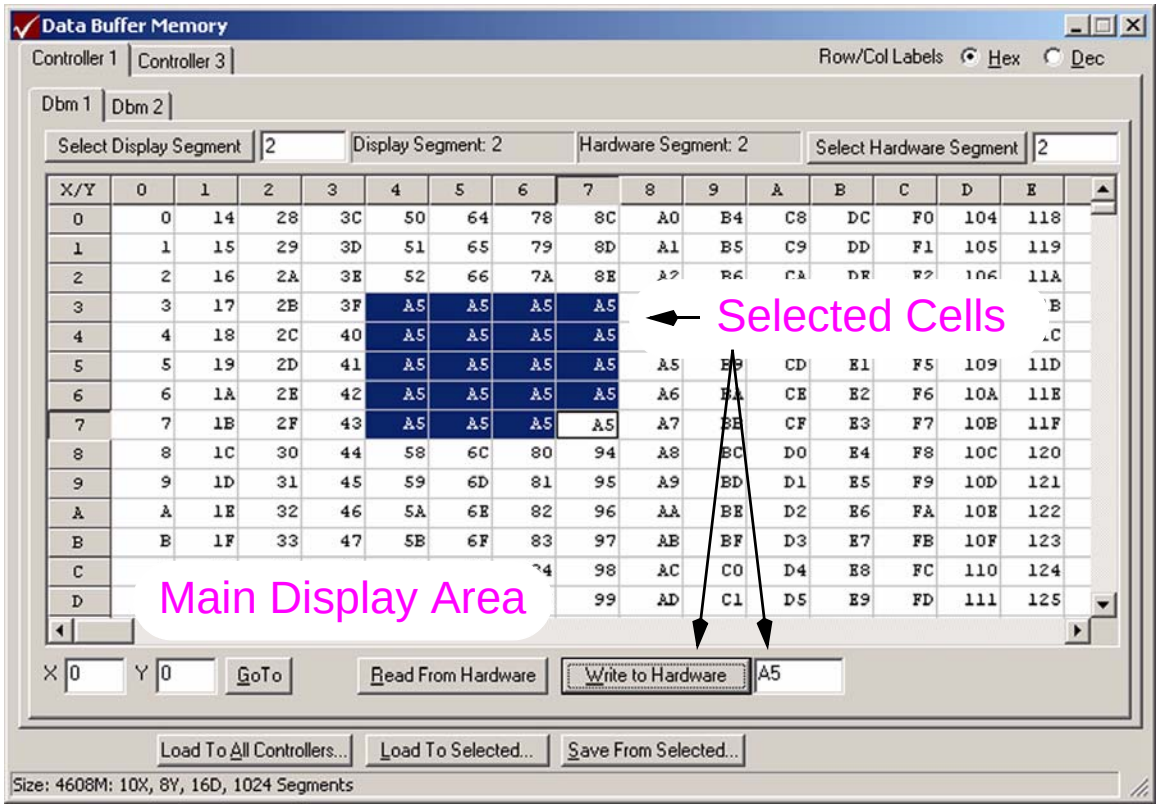
Note: the information here includes changes to DBMTool which were first available in software release h2.2.7/h1.2.7.

This section includes the following:

6.5.1 Overview

See DBMTool.

DBMTool is used to display and modify the contents of one or more Data Buffer Memory (DBM) segments. The image below shows a typical DBMTool image:



See DBMTool Controls and DBM Data Modification

Figure-77: DBMTool

In this example, DBMTool displays two controller tabs and two DBM tabs, representing a 4-site test system with Sites-per-Controller = 2; i.e. each controller represents 2 sites, each with two DBMs (one per APG). See DBM & Multiple Sites-per-controller.

The various display features and controls are described in DBMTool Controls. DBM data can be modified using DBMTool, see DBM Data Modification.

6.5.2 Starting DBMTool

See DBMTool.

Three methods are available to start DBM tool:

- Click on the DBM icon in the *Ui* toolbar 
- Type keyboard shortcut **`Ctrl+D`**
- Choose **`Tools: DBM`**...

5. Beginning in software release h2.xx.yy, DBMTool can be resized and the final size can be persistent (see UI Tool Persistence).

6.5.3 DBMTool Controls

See DBMTool.

DBMTool has the following displays and controls:

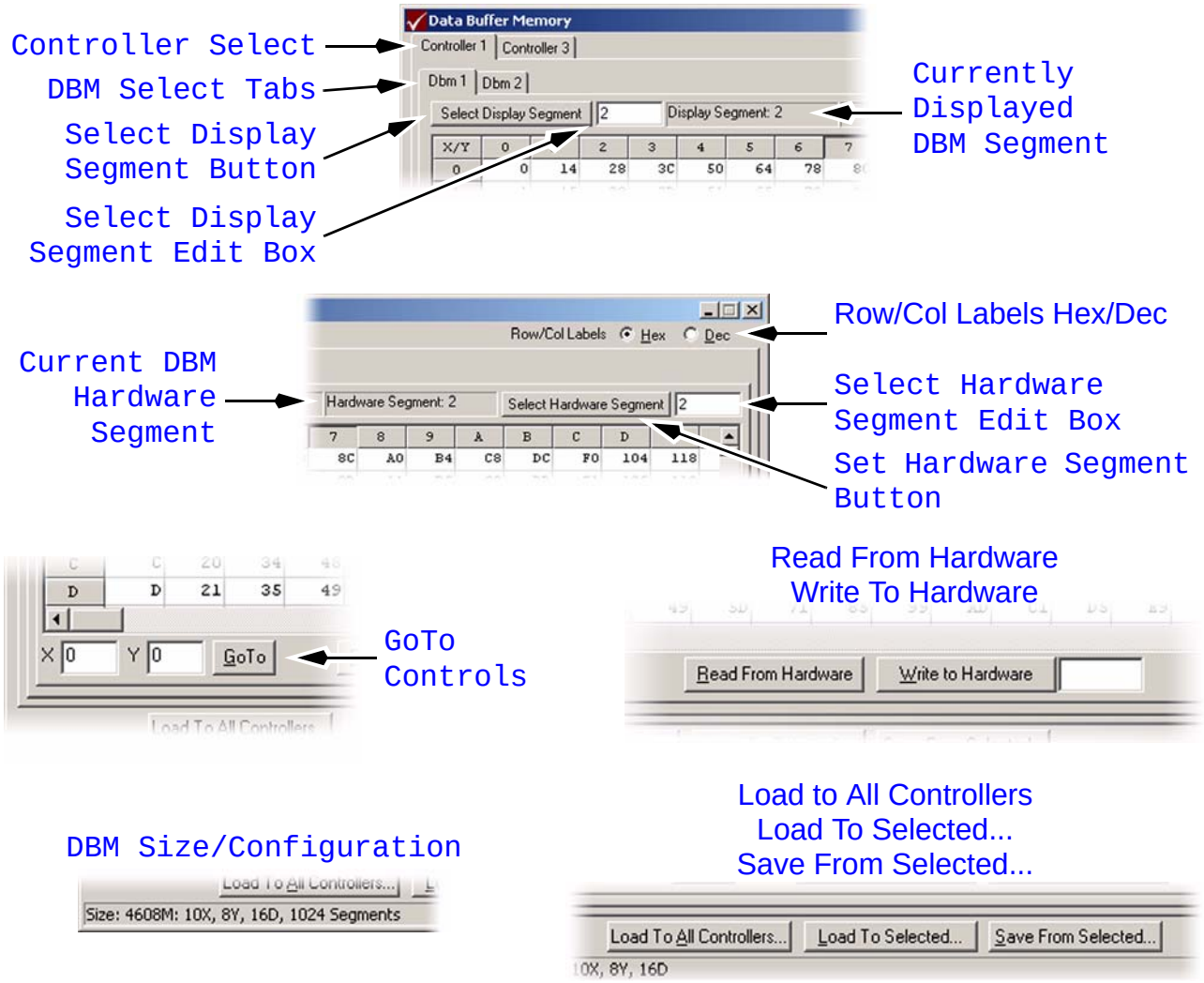


Figure-78: DBMTool Controls

The following table describes the various DBMTool controls:

Control	Purpose
Controller Select Tabs	Used to select the site controller which contains the DBM to display. When Sites-per-Controller > 1, only tabs for the master controllers are displayed.
DBM Select Tabs	Used to select which DBM is to be displayed. These tabs are only displayed when Sites-per-Controller > 1. These tabs were previously labeled **APG#** (where # = 1 to n).
Select Display Segment Button	Clicking this button causes the value entered in the Select Display Segment Edit Box to be read and used to select which DBM segment is being displayed. Takes effect immediately. The number of selectable segments is determined by the size of the installed DBM vs. the size of the DBM configuration, set using dbm_config_set(). Both Read From Hardware and Write To Hardware access this segment. Does NOT change DBM segment selection in hardware. (new in software release h2.2.7/h1.2.7)
Select Display Segment Edit Box	Edit box used to enter the segment number to be displayed. Has no effect until the Select Display Segment Button is clicked.
Currently Displayed DBM Segment	Indicates which DBM segment is currently being displayed.
Row/Col Labels Hex/Dec	Use to select the mathematical base used to display DBMTool's Row/Column header values. Also affects values entered using the GoTo Controls. Does not affect Main Display Area (cell values) which are always Hex values.

Control	Purpose
Set Hardware Segment Button	Clicking this button causes the value entered in the Select Hardware Segment Edit Box to be read and used to change the DBM segment selection in hardware. Takes effect immediately. Also sets the Select Display Segment Edit Box value. The number of selectable segments is determined by the size of the installed DBM vs. the size of the DBM configuration, set using dbm_config_set(). Updated by Read From Hardware. (new in software release h2.2.7/h1.2.7)
Select Hardware Segment Edit Box	Edit box used to enter the segment number to be selected in hardware. Has no effect until the Set Hardware Segment Button is clicked.
Current DBM Hardware Segment	Indicates which DBM segment is currently selected in hardware for the DBM being displayed.
GoTo Controls	Used to explicitly select a cell in the Main Display Area. Scrolls the display as necessary. GoTo value entry is affected by the Row/Col Labels Hex/Dec selection.
Read From Hardware	Reads hardware values from the currently selected controller, DBM and segment. Updates the Main Display Area, the value displayed for Select Hardware Segment Edit Box and the DBM Size/Configuration.
Write To Hardware	Writes the specified value to the cells currently selected in the Main Display Area, see DBM Data Modification. Writes only to the currently selected controller, DBM and segment. Note that at least one cell is always selected, thus clicking this button will always modify at least one DBM data value.
Load to All Controllers	Loads all DBMs on all controllers from a specified disk file. See DBM File Read/Write. (new in software release h2.2.7/h1.2.7)

Control	Purpose
Load To Selected...	Loads the currently selected segment in the currently selected DBM and controller from a specified disk file. See DBM File Read/Write. (new in software release h2.2.7/h1.2.7)
Save From Selected...	Saves the currently selected segment from the currently selected DBM and controller to a specified disk file. See DBM File Read/Write. (new in software release h2.2.7/h1.2.7)
DBM Size/Configuration	Displays the size of the installed DBM and the current DBM configuration as previously set using `dbm_config_set()`. Also displays the number of available DBM segments which is determined by the size of the installed DBM vs. the size of the DBM configuration. Updated by Read From Hardware.

6.5.4 DBM Data Modification

See DBMTool.

DBMTool can be used to modify the data stored in the currently selected DBM segment:

1. Depending on the configuration of the system in use, one or more controller and APG tabs are presented in DBMTool. These are used to select the DBM to be viewed and modified. See DBMTool Controls (`Controller Select Tabs`, `DBM Select Tabs`, etc.).
2. The DBM addresses to be modified are selected using various methods
   - Only adjacent addresses can be selected for editing.
   - Left-click the mouse in the Main Display Area to select one DBM address.
   - Left-click and drag the mouse in the Main Display Area to select multiple DBM addresses.
   - Use the `GoTo Controls` to select one DBM address. This will also scroll the display as necessary.

- Left-click the mouse in the Main Display Area to select the upper-left corner of a region of DBM addresses (cells), hold the shift button and left-click to select the lower-right corner.
- An entire row or column can be selected by clicking on the appropriate row or column label.
- The entire DBM can be selected by clicking on the upper-left cell labeled “X\Y”.

3. Before or after making the desired selection, the value to be written is entered in the edit-box next to the Write To Hardware button.
4. Click the Write To Hardware button to write the value to the selected DBM addresses.

Note: at least one cell is always selected in DBMTool, thus clicking the Write To Hardware button will always modify at least one DBM data value. Beware!

6.5.5 DBM File Read/Write

See DBMTool.

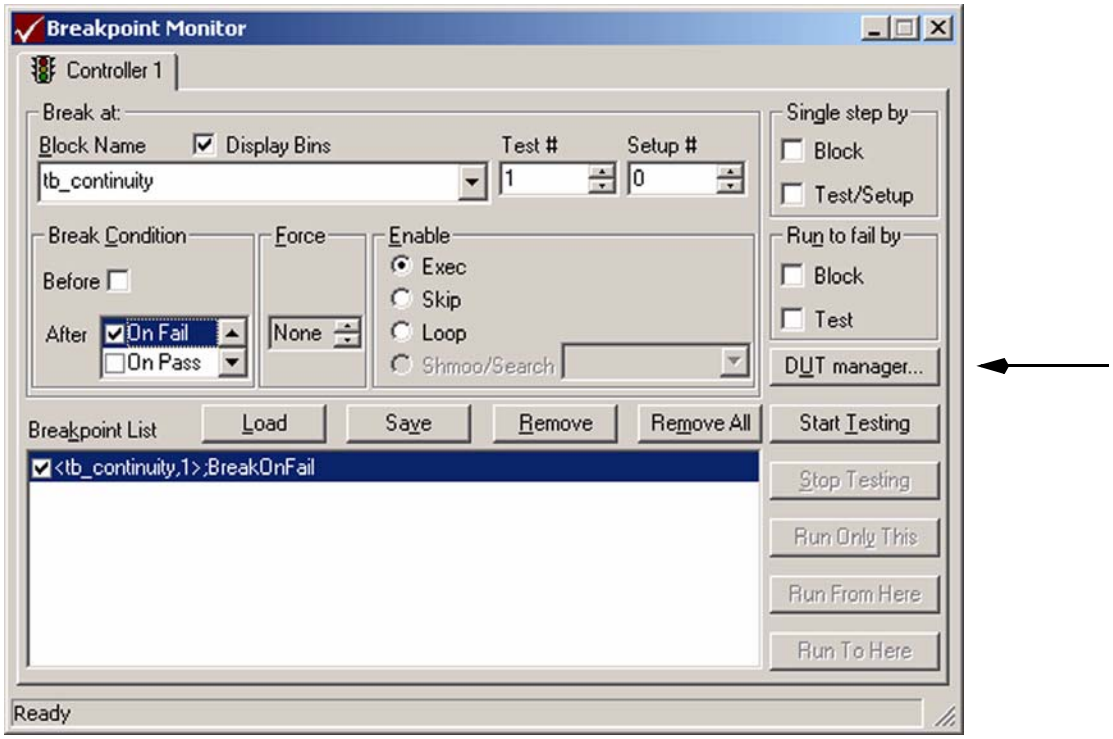
Beginning in software release h2.2.7/h1.2.7, DBMTool can be used to save DBM data to a disk file or load DBM data from a disk file. Note the following:

- The Save From Selected... button will put data from the currently selected DBM segment to a specified disk file. The entire segment is saved.
- The Load To Selected... button will load data from a specified disk file into to the currently selected DBM segment in the currently selected controller. The entire segment is loaded.
- The Load to All Controllers button will load data to the currently selected DBM segment of in all controllers from a specified disk file. The entire segment is loaded.
- A standard Windows file browser is presented to select the file to be loaded or written.
- The `dbm_file_image_write()` and `dbm_file_image_read()` functions are also available to write and read a DBM image to/from a disk file.
- The DBM file on disk is in a binary format, as described in DBM Data File Format. This format includes a DBM configuration, in the form of a header. Load To Selected... will only load the file into the DBM if the current DBM configuration matches the configuration saved in the file header.

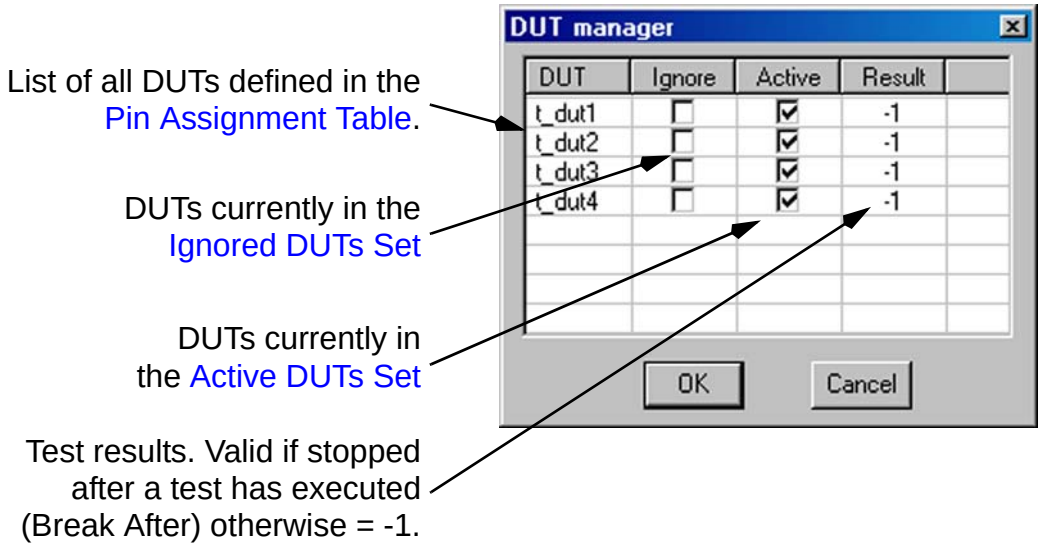
6.6 DUT Manager

The DUT Manager dialog is used in Magnum 1/2/2x Multi-DUT Test Programs to interactively move DUT(s) into or out of the Ignored DUTs Set (IDS) and/or Ignored DUTs Set (IDS). See Magnum 1, 2 & 2x Parallel Test.

The DUT Manager is started from Breakpoint Monitor, using the **DUT manager** button:



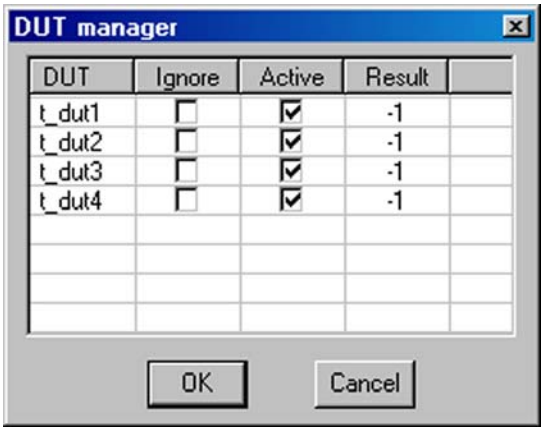
The DUT Manager has the following displays and controls:



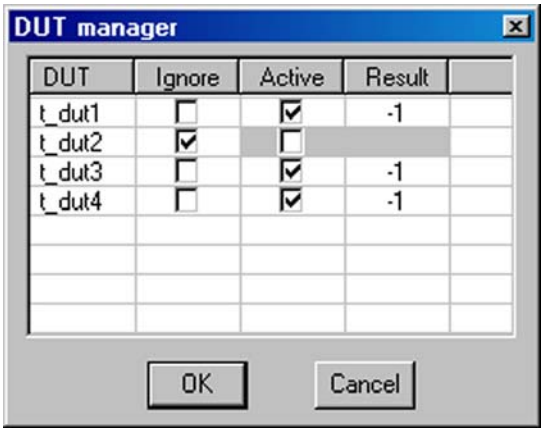
Note the following:

- The list of DUTs defined in the Pin Assignment Table is display only i.e. it can't be modified.
- The Ignored DUTs Set (IDS) can only be modified between Sequence & Binning Table executions. It is not possible to modify the IDS when stopped at a breakpoint.
- The Active DUTs Set (ADS) can only be modified when the Sequence & Binning Table is being executed while execution is stopped at a breakpoint. Breakpoints are managed using the Breakpoint Monitor.
- Changes to the Active DUTs Set (ADS) are scoped to the test block being executed at the time the ADS is modified. Detailed rules are described in Active DUTs Set (ADS).
- When a DUT is added to the IDS it is removed from the ADS. This is done immediately.
- When a DUT is removed from the IDS, it is added to the ADS, but this does not occur until the next time the Sequence & Binning Table executes.
- Changes to the ADS while looping take effect just like changes to voltages, timing, etc.
- Multi-DUT Test Results are updated while looping, but only for DUT(s) in the ADS.

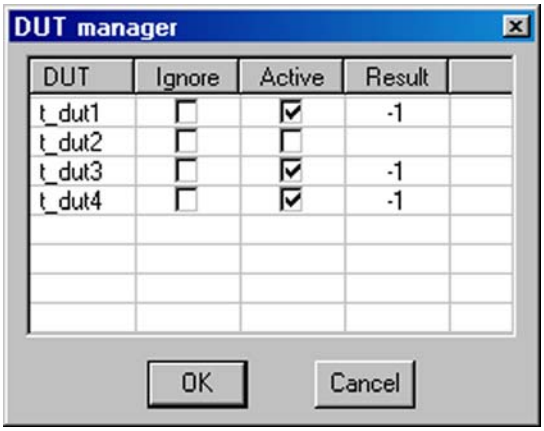
The following images were taken using a test program which defined 4 DUTs.:



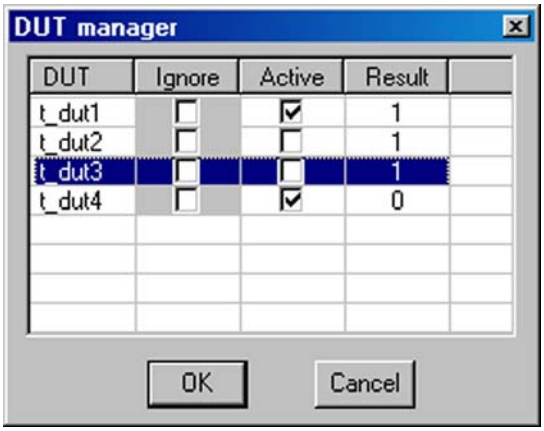
Initially, all DUTs in the test program are in the Active DUTs Set (ADS) i.e. enabled for testing. See Magnum 1, 2 & 2x Parallel Test. The Result values (-1) are invalid unless testing is stopped after a test has executed (Break After).



In this example, the user has put `t_dut2` into the Ignored DUTs Set (IDS), which removes it from the Active DUTs Set (ADS). If the Sequence & Binning Table is executed, only `t_dut1`, `t_dut3` and `t_dut4` will be tested.



In this example, the user removed `t_dut2` from the Ignored DUTs Set (IDS) but has not yet executed the Sequence & Binning Table. It is not until the Sequence & Binning Table is executed that `t_dut2` will reappear in the Active DUTs Set (ADS).



In this example, the test program is stopped at a breakpoint (Break After) and the user has removed `t_dut2` and `t_dut3` from the Active DUTs Set (ADS). Note that this does not add them to the Ignored DUTs Set (IDS) and once Sequence & Binning Table execution is resumed these DUTs may reappear in the ADS as controlled by the test program. Note the Result values are now valid since the values were read after a test had executed (Break

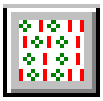
6.7 ECRTool

ECRTool is used to examine and modify the contents of an ECR.

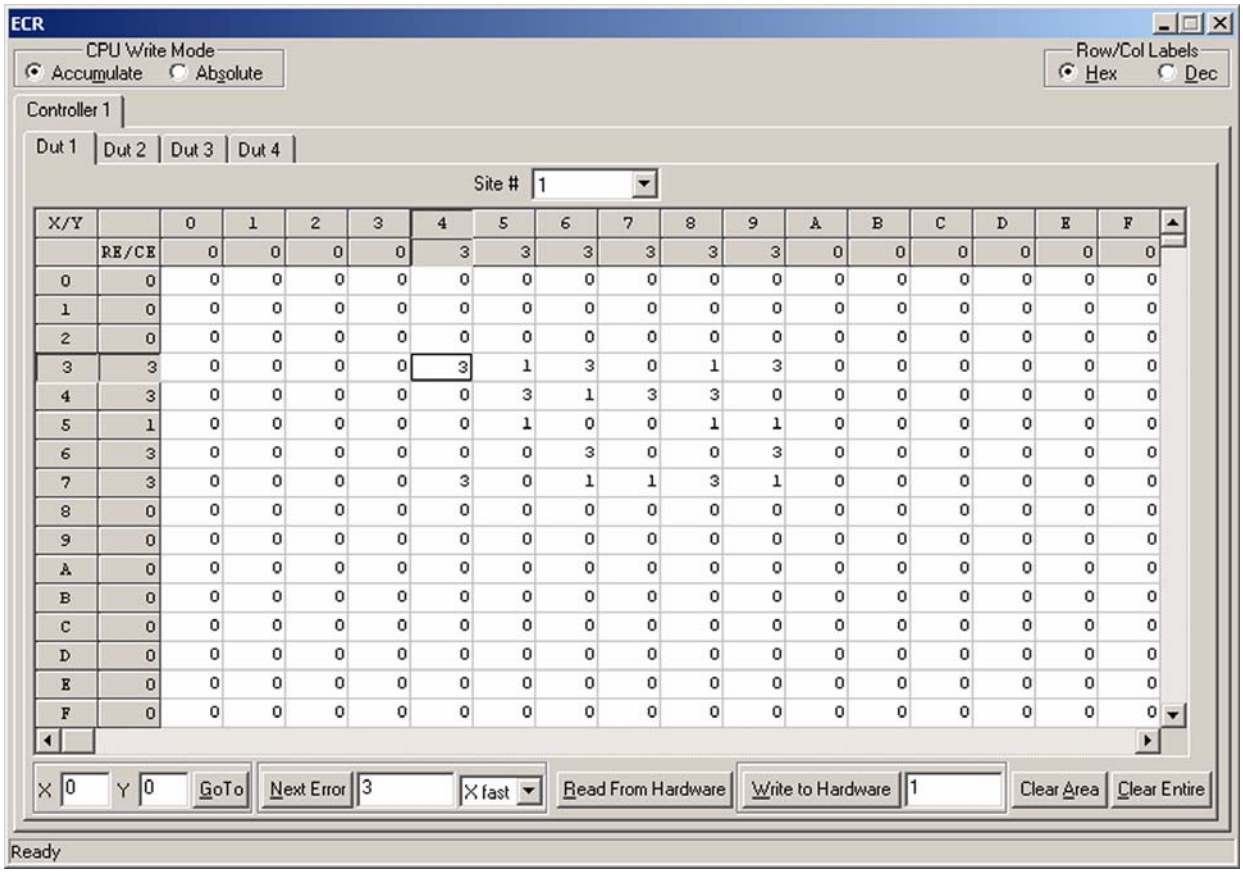
ECRTool cannot be started when:

- The ECR hardware (RAMs) are not installed
- The ECR is currently configured for Logic Error Catch use. See Logic Error Catch (LEC).

To start ECRTool:

- Click on the ECRTool icon from the *Ui* tool-bar 
- Type keyboard shortcut **Ctrl+E**
- Choose **Tools: ECR**...

The following image shows ECRTool for Magnum 1/2/2x:



X/Y		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	RE/CE	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	3	0	0	0	0	3	1	3	0	1	3	0	0	0	0	0	0
4	3	0	0	0	0	0	3	1	3	3	0	0	0	0	0	0	0
5	1	0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0
6	3	0	0	0	0	0	0	3	0	0	3	0	0	0	0	0	0
7	3	0	0	0	0	3	0	1	1	3	1	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The Magnum 1/2/2x ECRTool has the following displays and controls:

Table 6.7.0.0-1 Magnum 1/2/2x ECRTool Controls

Control	Type	Purpose
CPU Write Mode Accumulate Absolute	Radio Button	Affects the operation of the **Write to Hardware** button. Accumulate means that new errors (1's) are added to existing errors; i.e. accumulate 1's. Absolute causes the contents of the selected ECR cells to be completely replaced with the specified value.
Controller Tab	Tab	Used in conjunction with Site# to select the ECR to be read (Site# is only pertinent when sites-per-controller > 1). For example, in a system containing 8 Site Assembly Boards, with sites-per-controller = 2 there will be 4 master sites and 4 contoller tabs, labeled Controller 1, 3, 5, and 7. To read the ECR from the slave site of Controller 3 select the second Controller Tab (Controller 3) and select Site# 2. To read the ECR on the master select Site# 1.
DUT Tabs	Tab	In Multi-DUT Test Programs ECRTool will display errors for only the selected DUT. Also see `ecr_config_set()`.
Site#	Combo Box	Displayed only when sites-per-controller > 1. Used in conjunction with Controller Tab to select the ECR to be read. See Controller Tab.
Row/Col Hex Dec	Radio Buttons	Determines whether the X/Y row/column labels are displayed as decimal or hexadecimal values.
Note: only the ECR selected using the Controller Tab and, when sites-per-controller > 1, the Site# selection, is accessed. Beginning in software release h2.2.7/h1.2.7, ECRTool can be resized and the final size can be persistent (see UI Tool Persistence).		

Table 6.7.0.0-1 Magnum 1/2/2x ECRTool Controls *(Continued)*

Control	Type	Purpose
X/Y Row/Column Labels	Label	Displays the logical ECR address of each row (X) and column (Y) in the display. This equates to the logical X and Y address output from the APG when the logging information to the ECR.
RE/CE	Label	Displays values read from the Row RAM and Column RAM (CE). See Magnum ECR Block Diagram.
GoTo X Y	Button Edit Box Edit Box	Used to move the ECRTool display to a specified location. Enter the row (X) and column (Y) address values and click the **GoTo** button to scroll the main display to the specified X/Y addresses.
Next Error Next Error Search Value Next Error Search Direction	Button Edit Box Combo Box	Used to locate and display the *next* error. See ECRTool Next Error Search.
Read from Hardware	Button	Used to retrieve and display information from the ECR. This affects the entire ECR display, regardless of any cells which are currently selected.
Note: only the ECR selected using the Controller Tab and, when sites-per-controller > 1, the Site# selection, is accessed. Beginning in software release h2.2.7/h1.2.7, ECRTool can be resized and the final size can be persistent (see UI Tool Persistence).		

Table 6.7.0.0-1 Magnum 1/2/2x ECRTool Controls *(Continued)*

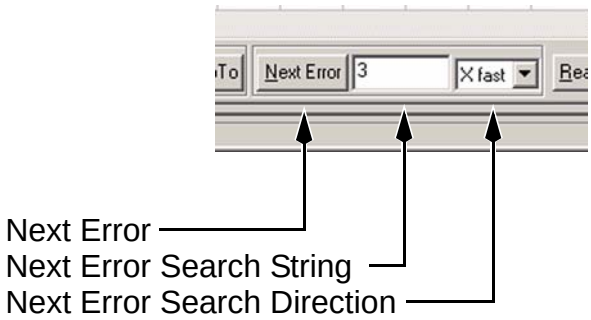
Control	Type	Purpose
Write to Hardware Write Value	Button Edit Box	The value entered in the edit box (right of the **Write to Hardware** button) is written to the cell(s) currently selected in the ECRTool display. Also updates the row RAM (RE) and column RAM (CE). Note: *IMPORTANT*: the RE/CE values are not modified and *may be wrong* after using **Write to Hardware**. See Note:.
Clear Area	Button	Clears errors from the area of the ECR selected in the main display (using the left-mouse button). Note: *IMPORTANT*: the RE/CE values are not modified and *may be wrong* after using **Clear Area**. See Note:.
Clear Entire	Button	Clears all errors from the ECR, including Row RAM (RE) and Column RAM (CE). See Magnum ECR Block Diagram.
Note: only the ECR selected using the Controller Tab and, when sites-per-controller > 1, the Site# selection, is accessed. Beginning in software release h2.2.7/h1.2.7, ECRTool can be resized and the final size can be persistent (see UI Tool Persistence).		

Note: proper Redundancy Analysis (RA) and BitmapTool operation depends upon synchronization between the contents of the main ECR RAM and values in all ECR RAMs and counters. These values are guaranteed to be valid after funtest() is executed. Conversely, these values are suspect ANY time user-code or ECRTool modifies some values in the ECR hardware. Beware! See ecr_rams_update().

6.7.1 ECRTool Next Error Search

In software release h1.1.23 The operation of ECRTool's **Next Error** button was enhanced, as follows:

- The Next Error Search String edit box was added to the right of the **Next Error** button. This allows a search string to be entered, the value of which modifies the operation of the **Next Error** button (more below).
- The Next Error Search Direction combo-box was added to select between **X Fast** and **Y Fast** search direction.



As indicated, the operation of ECRTool's **Next Error** button depends on the value entered (if any) in the Next Error Search String and the selection of the **X Fast** and **Y Fast** search direction:

- If the search string value is empty the **Next Error** operation remains unchanged from that in previous software releases; i.e. locate and display the next failing address in the ECR, regardless of which data bit(s) failed.
- Entering 0 causes the **Next Error** operation to locate the next address in the ECR which has no errors.
- Any other search string value is treated as the search target when the **Next Error** button is clicked. During the search, the search value will only be tested against the value in a single cell in ECRTool; i.e it is not possible to search for values which span more than one cell in ECRTool.
- The search direction can be set as **X Fast** (row fast = search downwards) or **Y Fast** (column fast = search right wards) using the combo-box to the right of search string edit box.

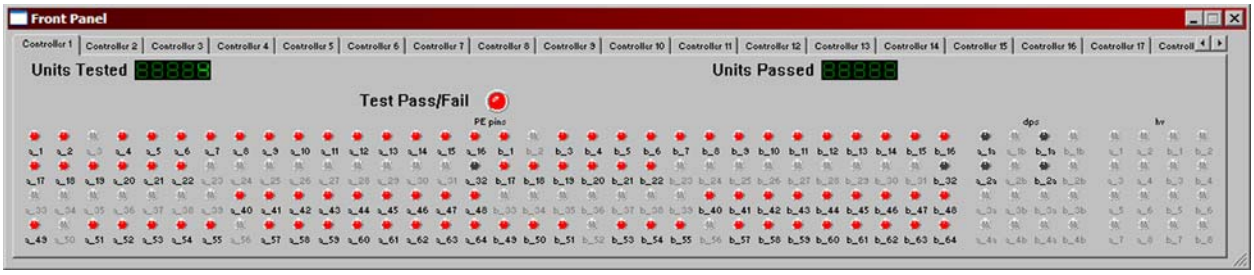
6.8 FrontPanelTool

UI - User Interface provides a built-in Front Panel tool, displaying various indicators and counts related to previously executed tests (more below).

To start FrontPanelTool:

- Type keyboard shortcut **`Alt+1`**
- Choose **`View: Front Panel`**

The image below shows FrontPanelTool displaying 17 Controller tabs (of potentially 32 possible using Magnum 1/2/2x):



In this example, note the following:

- Since there are multiple Controller tabs, the program is currently executing on a multi-site system. Additional Controller tabs would be displayed if running on a multi-site system (SV, SSV, GV. etc.).
- Since 128 pins are displayed (64 A pins and 64 B pins) the program is using Sites-per-Controller = 1. If Sites-per-Controller = 2 256 pins would be displayed, etc.
- Not all pins are being used by the test program. Those not used are greyed-out i.e. a_3, b_2, etc.
- Four DUTs have been tested, and all four failed (0 Units Passed).
- The last executed test failed (Test Pass/Fail = Red) and only signal pins failed (no DPS indicators or HV indicators are on)
- The test program only uses DPS-1 and DPS-2; the other DPS are greyed-out).
- The test program does not use any High Voltage Source/Measure Unit (HV) i.e. all are greyed-out).

In the Front Panel display:

- The pin numbers represent used tester pin numbers.

- Red on the *Test Pass/Fail* indicates an overall FAIL, green indicates an overall PASS.
- Red on the PE, HV, and DPS pins indicate FAIL; dark gray indicates PASS or UNTESTED.
- The *Units Passed* count display will only be updated if the test program defines a Test Bin Group named *units_passed* (case insensitive). For example, to add myBin1 and myBin2 to the *units_passed* group, add the following line to your test program:

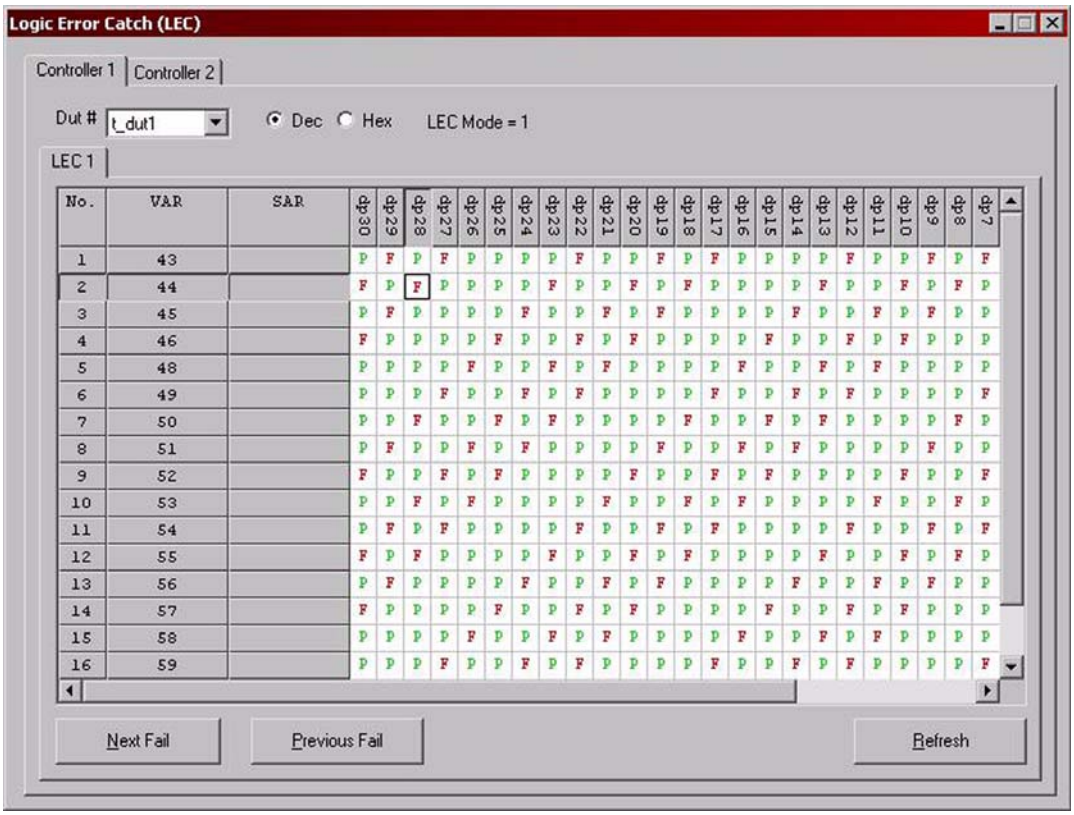
```
TEST_BIN( myBin1 ) {}
TEST_BIN( myBin2 ) {}
TEST_BIN_GROUP( units_passed ) { BINS2( Bin1, Bin2 ) }
```

FrontPanelTool is quite useful when a test is looping (see Breakpoint Monitor). The **`Test Pass/Fail`** indicator and failing PE pins/dps indicators can be used to observe stable or intermittent pass/fail results as the test loops. While a test is looping the effect on pass/fail results can be observed using FrontPanelTool while modifying voltages/currents (see Voltage and Current Tool) and timing (see TimingTool).

FrontPanelTool is for display only i.e. it is not possible to control any hardware or software features using FrontPanelTool.

6.9 LEC Tool

Logic Error Catch Tool (LECTool) can be used to examine the contents of the optional Logic Error Catch (LEC) RAM, which is the Error Catch RAM (ECR) being used in logic error catch mode. The image below is an example of the LECTool display:



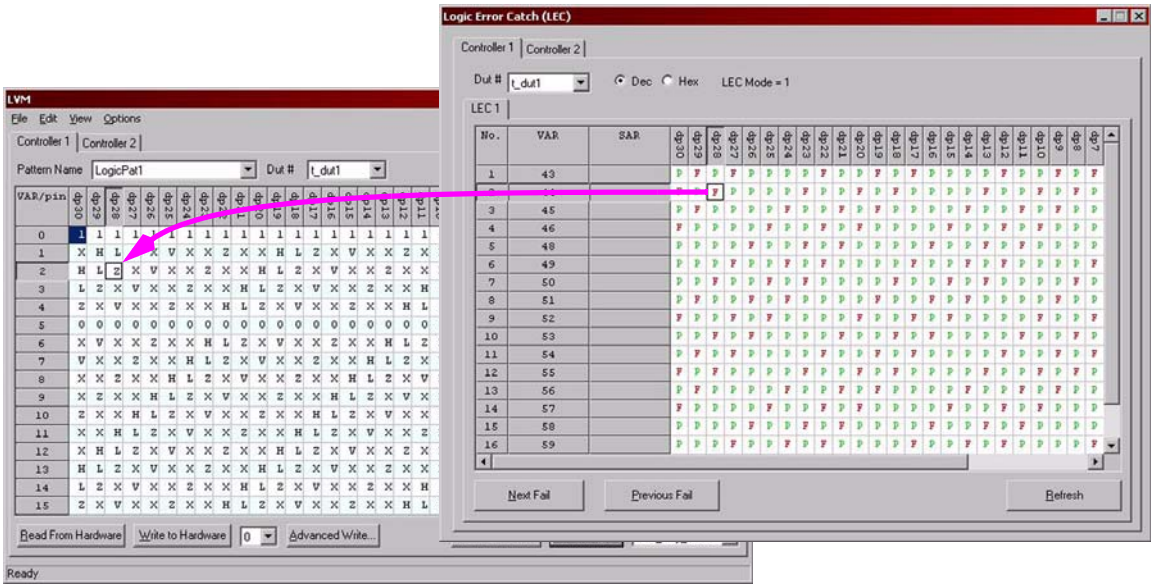
Note the following:

- It is not possible to modify Logic Error Catch (LEC) contents using LECTool.
- The **No.** column indicates the position (address) of the LEC which is displayed in each row.
- The **VAR** column displays the logic vector address (VAR) of each failing vector read from the Logic Error Catch (LEC). This is an absolute VAR (not pattern relative).
- When failing scan vectors are logged, the The **SAR** column displays the scan vector address (SAR) of each failing vector read from the Logic Error Catch (LEC). See VAR/SAR Description. This is an absolute SAR (not pattern relative).

- Failing pins are indicated by a red **F**. Passing pins with a green **P**.
- Pins are displayed in the order of the pin list specified when `lec_config_set()` was last executed. See below.
- The example above was captured using:

  ```
  funtest( LogicPat1, LEC_only_errors ); // funtest()
  ```

  The `LEC_only_errors` option captures the first 2Meg (2^{21}-6) failing vectors.
- Clicking the **NextFail** button or **PreviousFail** button moves the selection box (shown on the first pin of the first vector above) to the next or previous failing pin. The LECTool display will scroll up or down as necessary. Clicking these buttons sends information to LVMTool, causing it to move its selection box to the same pin/vector coordinate. This can be seen in the images below:



- Using the mouse to manually select a failing pin (F) also causes LVMTool to move its selection box to the same pin/vector coordinate.

LECTool can be used in two ways:

- Invoked from LVMTool using the **Run with LEC** button.
- Invoked from UI's Tool menu.

When LECTool is invoked from LVMTool, the system software does the following to capture errors to the Logic Error Catch (LEC):

- Executes `lec_config_set()` to specify which pin(s) are to be captured in the LEC. The pin list selected in LVMTool is used. Note that this also changes the operating mode of the ECR.

- Executes `funtest()` using the pattern stop option selected in LVMTool. The default is `LEC_only_errors`.

Note: when LECTool is invoked from LVMTool using the **`Run with LEC`** button the system software does ***NOT*** restore the state set by prior executions of `lec_config_set()` from user code.

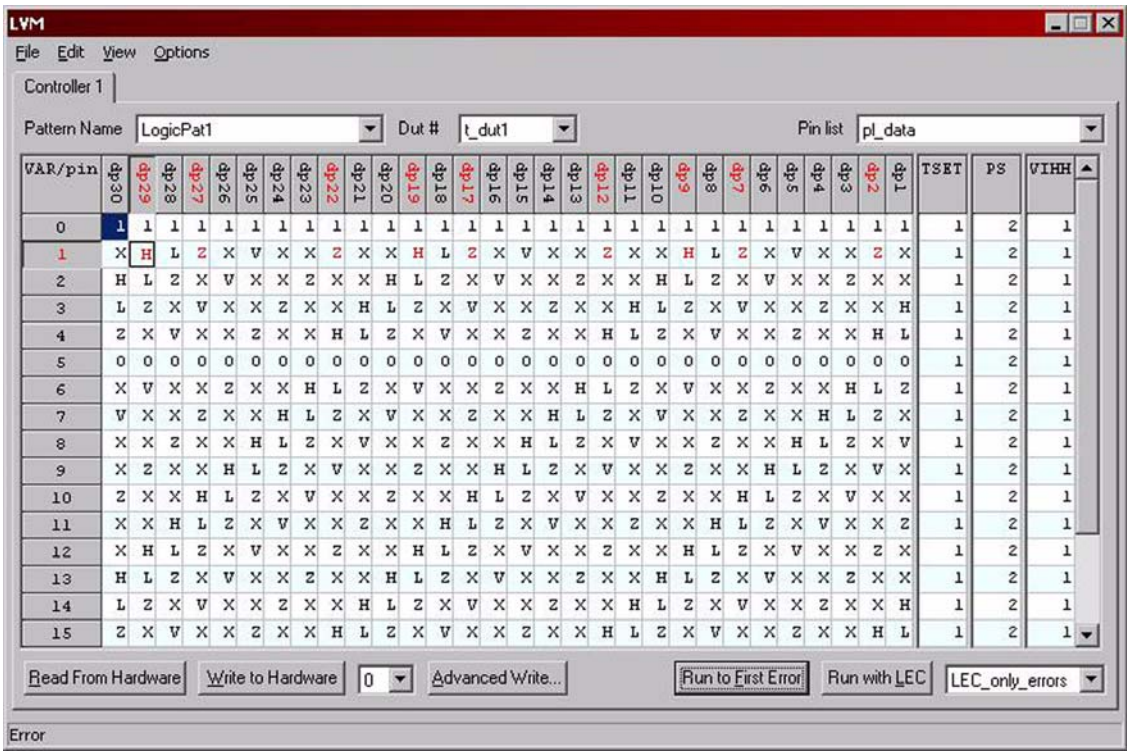
When LECTool is to be invoked from UI's Tool menu, the user's test program must do the following to obtain useful results:

- Execute `lec_config_set()` to specify which pin(s) are to be captured in the LEC.
- Execute `funtest()` using one of the LEC pattern stop options. i.e. `LEC_first_vectors`, `LEC_last_vectors`, `LEC_before_error`, `LEC_after_error`, `LEC_only_errors`, `LEC_center_error`.

If any of these steps is skipped, LECTool will start but no pin/vector information will be displayed.

6.10 LVMTool

LVMTool is used to display and modify the contents of the optional Logic Vector Memory (LVM) and Scan Vector Memory (SVM). LVMTool will also display failing pins and failing vector address (VAR) or failing scan vector address (SAR) from the first failing cycle:



This section covers the following topics:

- Starting LVMTool
- LVMTool Tool-bar
- LVMTool Use
- PINFUNC Field Display & Edit
- Copy/Paste LVM Pattern Data
- DDR LVMTool
- LVMTool in Simulation Mode
- LVMTool Limitations

6.10.1 Starting LVMTool

See LVMTool.

LVMTool can be started several ways:

- Click on the LVMTool button in Ui's tool-bar: 

- In UI, choose **Tools: LVM...**
- Type the keyboard shortcut **Ctrl+L**

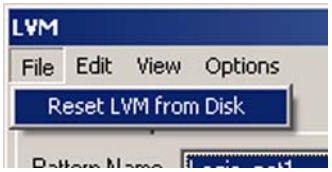
These controls are disabled if the currently loaded test program did not load one or more Logic Test Patterns.

Using Magnum, LVMTool may be used in simulation mode by setting the environment variable `SIMULATED_LVM`. See LVMTool in Simulation Mode.

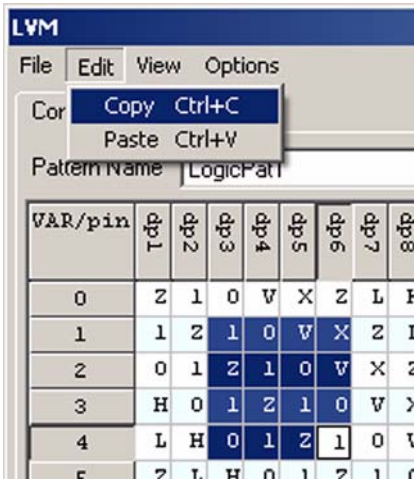
6.10.2 LVMTool Tool-bar

See LVMTool.

LVMTool has the following tool-bar menu options:

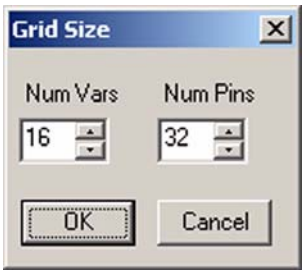


File->Reset LVM From Disk. This reloads **ALL** logic and scan patterns from disk into the LVM. It also updates the pattern currently displayed in LVMTool:



Edit->Copy and **Edit->Paste**. Used to copy and paste values from like areas in LVMTool. See Copy/Paste LVM Pattern Data. The **Paste** option is not enabled until a **Copy** operation has been performed.

View->TSET, **View->PS**, **VIEW->VIHH**. Used to hide or display the specified columns in LVMTool.



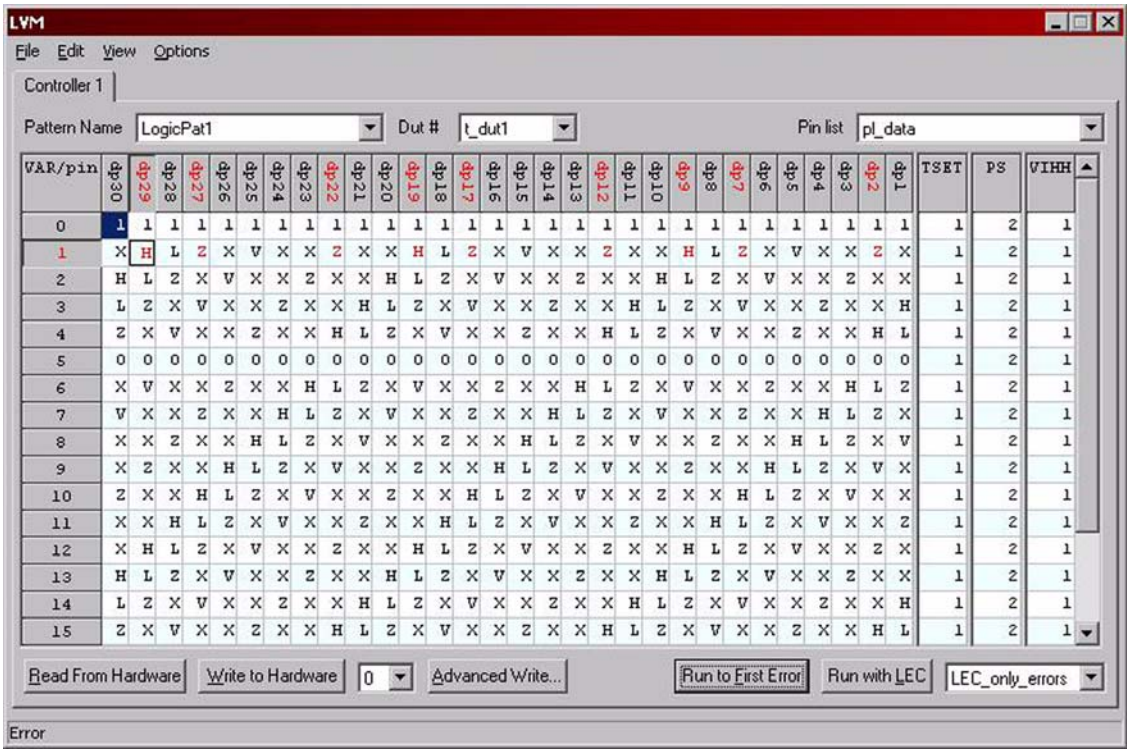
Options->Grid Size.... Invokes the Grid Size dialog, which can be used to modify the size of the LVMTool display, in the terms of rows (Num Vars) and columns (Num Pins).

6.10.3 LVMTool Use

See LVMTool.

LVMTool displays the contents of logic vector memory (LVM) i.e. the per-pin per-cycle pattern information. The same Logic Vector Bit Codes used in the pattern source file are displayed. See LVMTool Limitations.

The image below has LVMTool displaying the Logic Test Pattern named *LogicPat1*. It is also showing failing pins from the first failing cycle for the first DUT (`t_dut1`):



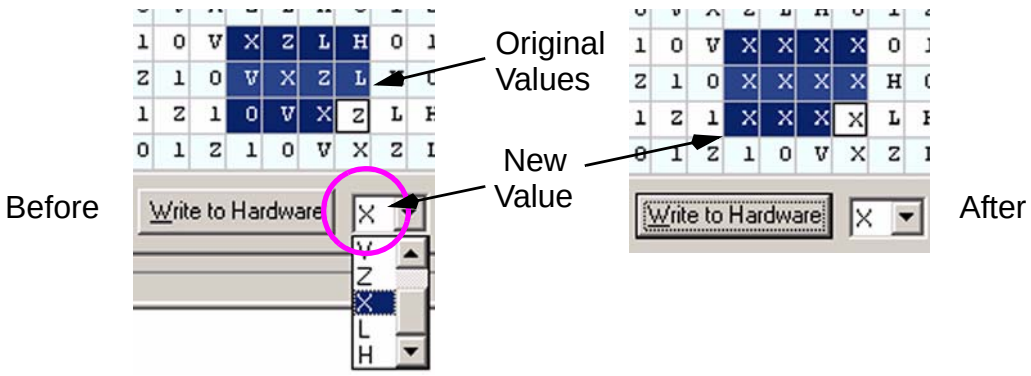
Note the following:

- Beginning in software release h2.2.7/h1.2.7, LVMTool can be resized and the final size can be persistent (see UI Tool Persistence).
- The main area of the tool display shows the contents of LVM for the selected pin list of the selected pattern. Initially, the display will show the first vector (VAR 0) of the selected pattern in the top row. The characters displayed are the same Logic Vector Bit Codes used in the pattern source file.
- When first invoked, LVMTool displays the first Logic Test Patterns loaded. The **`Pattern Name`** menu allows selection of a different Logic Test Pattern for display. The list of patterns shown in this menu will *not* include Memory Test Patterns. However, since Scan Test Patterns are normally combined with Logic Test Patterns they will be displayed.

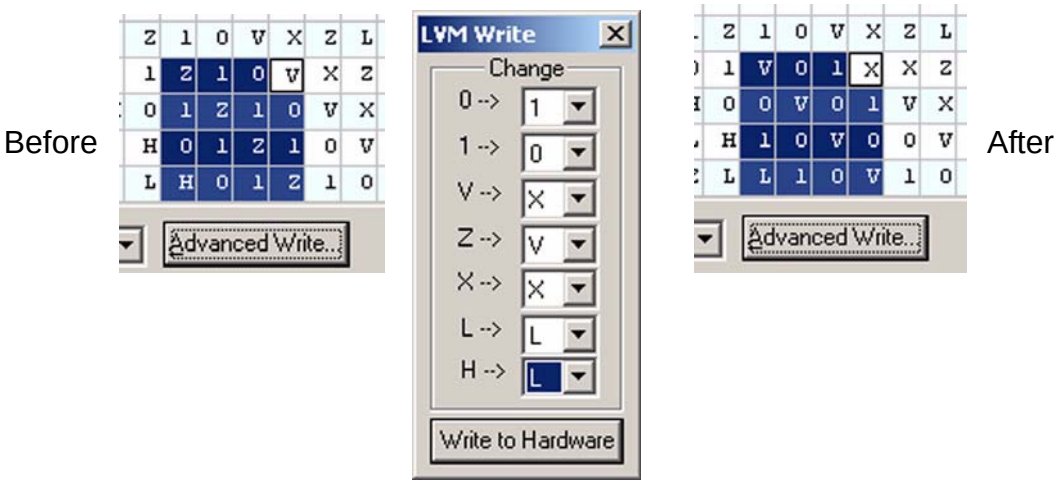
- Selecting a new pattern using the **Pattern Name** menu immediately reads the pattern from the Logic Vector Memory (LVM) and displays it in LVMTool. The pattern is not executed, and any previously displayed PASS/FAIL information is discarded.
- The **Dut #** menu is used, in Multi-DUT Test Programs, to select a DUT. This only affects the display of failing pin information generated using the **Run to First Error** button. More below.
- The **Pinlist** menu allows selection of the pin list used to determine the columns of pattern information to be displayed. The selected pin list controls how many columns (pins) are displayed and the order in which pins are displayed. The first pin in the pin list is the left-most column, etc. Both user-defined pin lists and the built-in pin lists are selectable using this menu, however only pin lists which contain pins which are mapped to `t_lvm` in at least one Pin Scramble Map are displayed for selection. Selecting a new pin list immediately reads the appropriate pattern information from LVM and displays it in LVMTool. The pattern is not executed, and any previously displayed PASS/FAIL information is discarded. Only pattern information from pins of the currently selected Site are displayed (see **Controller** tab below).
- Each row of displayed pattern information represents one LVM address. See LVMTool Limitations for information about RPT, STARTLOOP, GOSUB/RETURN, etc. In DDR mode each row displays either the A-cycle or B-cycle LVM content. See DDR LVMTool.
- The LVMTool display size is not user adjustable. Scroll bars will appear if the amount of information to be displayed exceeds the size of the display. It is possible to resize the VAR number column and the pin name row using the left mouse.
- The **Reset LVM from Disk** button causes ***ALL*** logic patterns to be reloaded from the disk into LVM. And, LVMTool is updated to display the reloaded information from LVM. Use this button to un-do manual edits to LVM contents made using LVMTool (see below).
- The **Read from Hardware** button re-reads LVM into the LVMTool display. This may be desirable after executing C-code which modifies LVM.

LVM content can be edited using two basic techniques, described below. The process begins using the left mouse button in the main LVMTool display to select one or more cells of the pattern is to be modified. Single cells, whole rows (vectors) or columns (pins), or an arbitrary rectangular region can be selected. Then, either:

- Use the **Write to Hardware** menu and button, to select a single new value to replace the value in all selected cells. For example, the following changes the selected cells from their original value to X:



- Select the **Advanced Write** button to display the LVM Write dialog:



Using this dialog, after selecting the desired value changes click the **Write to Hardware** button to change LVM contents for the selected cells. This dialog allows multiple changes to be made, but they are only applied to the cells previously selected in the LVMTool display. In the example above the following changes were made: 0→1, 1→0, V→X, Z→V, H→L.

The **Run to First Error** button executes the test pattern currently selected in LVMTool. If the pattern fails, the display is updated to show the first failing vector with failing pins displayed using RED characters. In Multi-DUT Test Programs, only failures for the DUT selected in the **DUT #** menu are displayed. Note that LVMTool has no control over the state of the tester's timing, PE voltages/currents, or DPS status, thus meaningful PASS/FAIL operation will normally require setting a breakpoint, using the Breakpoint Monitor, and

executing the Sequence & Binning Table to properly configure the tester before executing the pattern using LVMTool.

Clicking the **Run with LEC** button causes LEC Tool to be used to potentially display mulitple failing cycles/pins. Operation is as follows:

- The Logic Error Catch (LEC) is configured using the pins in the pin list selected in LVMTool. For reference see `lec_config_set()`.
- If the pattern selected in LVMTool is a non-DDR pattern the LEC configuration uses `t_single`, if the pattern is a DDR pattern the LEC configuration uses `t_double`.
- The pattern selected in LVMTool is then executed with the LEC option selected via the menu to the right of the **Run with LEC** button. These options are documented in LEC Capture Options.
- LEC Tool is then displayed and directed to read errors from the Logic Error Catch (LEC). Any errors displayed in LEC Tool are also indicated, using red tokens, in LVMTool.

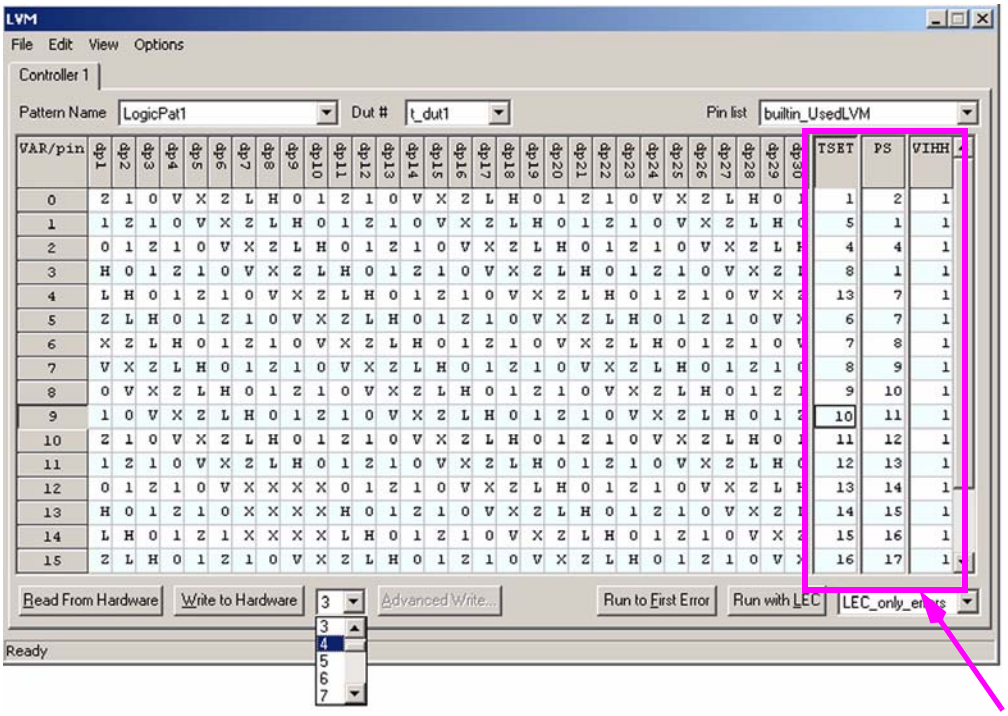
The **Controller** tabs seen in the upper portion of the LVMTool display are used to select which Site is currently being displayed. Selecting a new controller tab causes LVMTool to read and display fresh information from the appropriate LVM hardware, similar to what occurs when LVMTool is first started. This means that the display will show VAR 0, with no PASS/FAIL information i.e. any scrolling or failing pin information seen when the previous tab was selected is discarded.

6.10.4 PINFUNC Field Display & Edit

See LVMTool, PINFUNC Instruction.

The PINFUNC fields in LVMTool supports the following features:

- Display and edit the per-vector Time-set(TSET), Pin Scramble Map(PS), and VIHH Map(VIHH) selections. These are seen in the right columns of the LVMTool display below:



The menu located to the right of the **Write to Hardware** button is used to select the desired value which is written to the selected cells by clicking the **Write to Hardware** button.

6.10.5 Copy/Paste LVM Pattern Data

See LVMTool.

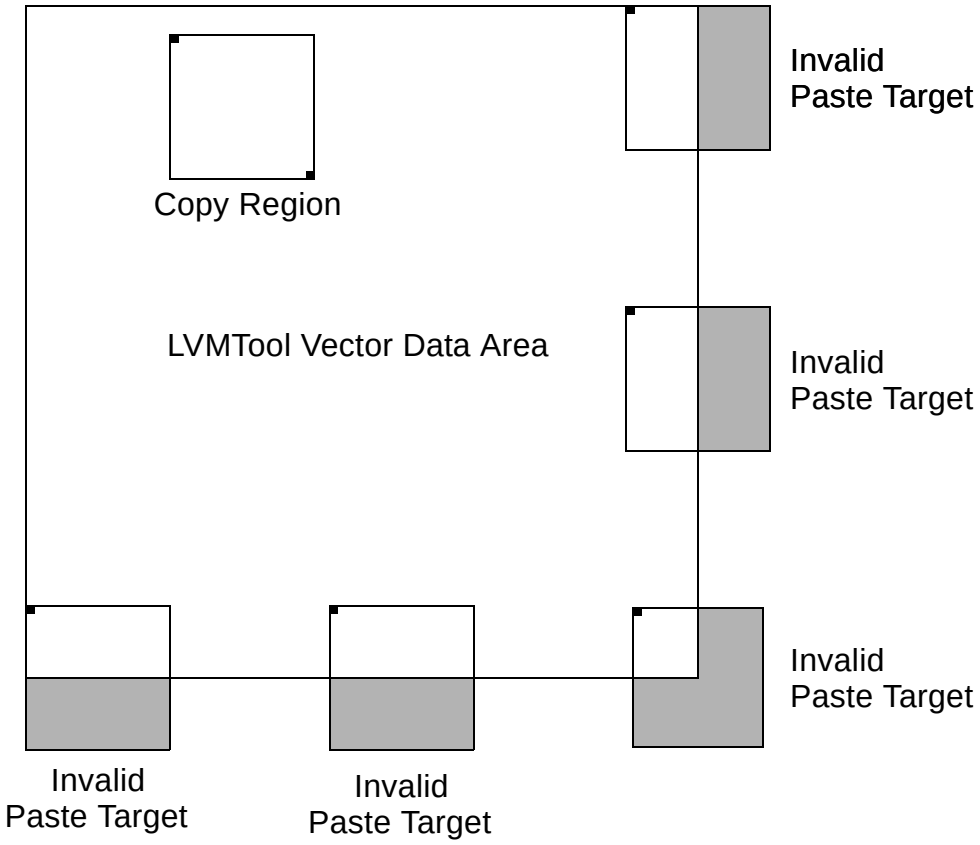
LVMTool allows modification of Logic Vector Memory (LVM) contents using the **Edit->Copy** and **Edit->Paste** operations.

Note the following:

- The mouse is used to select a region of pattern data to copy. Several options are available:

  - Select one or more entire columns (pins) using the pin name field.
  - Select one or more entire rows (vectors) using the Var/pin field.
  - Select one cell (1 pin @ 1 vector).
  - Select a rectangle of cells spanning multiple pins and/or multiple vectors. This is done by locating the upper left and bottom right corners of the desired region. The left-mouse button can be clicked and dragged or two cells can be selected while holding the shift key down.
- The copy operation is invoked using **Edit->Copy** or typing **Ctrl+C**.
- Selection of the paste destination is similar to selecting the copy region. By design, the paste region is constrained:
  - When using the first 3 selection options noted above the dimensions of the paste region must exactly match the copy region. For example, if two entire columns are selected and copied, it is necessary to select two entire columns before pasting.
  - When using the last selection option above, the destination can be chosen by selecting one cell, to locate the upper-left corner of the paste range, or an range identical to that copied can be selected. More below.
  - The warning dialog shown in the diagram below is displayed any time a paste operation is attempted with an incorrectly selected destination range.
  - The paste operation is invoked using **Edit->Paste** or typing **Ctrl+V**.

- As noted above, when selecting the paste destination the user must ensure that the entire region copied fits. In the example below, the paste targets labeled *Invalid* are not legal since some of the copied pattern will not fit. When this is attempted the dialog shown below is displayed:

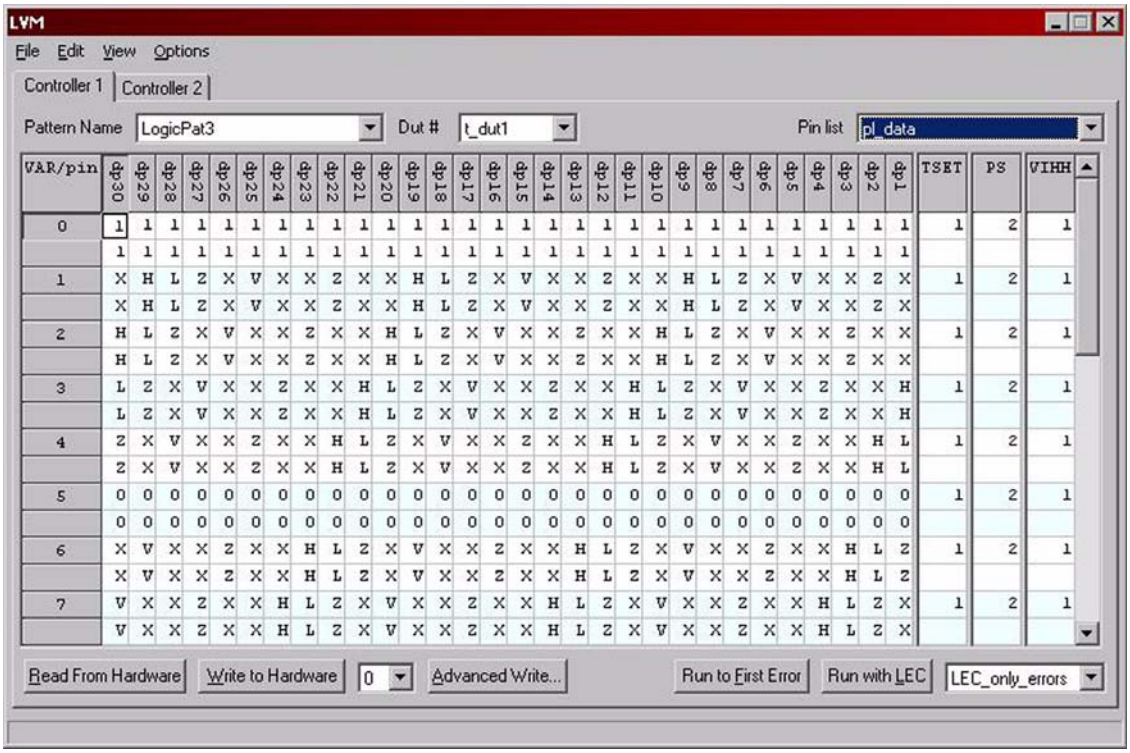


- Copy/Paste copies logic pattern information in the context of Logic Vector Bit Codes (H, L, 0, 1, X, etc.) or PINFUNC information as an integer value. It is the user's responsibility to ensure that the resulting operation is appropriate for the DUT being tested. The Copy/Paste software cannot prevent a tester pin from driving (1,0) a DUT output which is also driving, or to prevent strobing (H/L) a DUT input pin, etc.

6.10.6 DDR LVMTool

See LVMTool, PINFUNC Instruction.

LVMTool supports DDR logic patterns. The image below is used to describe how a DDR logic pattern is displayed:



Note the following:

- In DDR mode, the first row is the first DDR A-cycle, the second row is the first DDR B-cycle, etc. This is consistent with the typical pattern source file.
- Failures are only displayed for the first failing DDR cycle i.e. if, in a given tester cycle, both the DDR A-cycle and B-cycle contain failures, only those in the A-cycle are displayed (in RED).
- The PINFUNC area displays information aligned with the A-cycle, which is somewhat different than seen in the pattern source file (which requires PINFUNC information after the B-cycle vector tokens).
- Editing pattern data remains unchanged. See Copy/Paste LVM Pattern Data.

6.10.7 LVMTool in Simulation Mode

See LVMTool.

To use LVMTool without hardware requires setting the environment variable SIMULATED_LVM = 1, see Setting Environment Variables. Then, computer RAM is used to emulate the hardware LVM, allowing the user to interact with LVMTool. Note that the interface with LEC Tool is not operational in simulation mode.

6.10.8 LVMTool Limitations

See LVMTool.

LVMTool has the following limitations:

1. LVMTool displays only LVM contents, thus:
   - The execution order of logic vectors is not visible in LVMTool.
   - RPT, STARTLOOP, ENDLOOP, GOSUB, or RETURN display or editing is not supported. A RPT vector displays as a single vector, with no indication that the vector is a RPT. The other instructions are totally invisible.
2. Logic Error Catch (LEC) use is supported via LEC Tool. LEC Tool can display multiple failing vectors and, when a given failure is selected (clicked), the LVMTool display will be updated to indicate the location in LVM of the logic instruction and pin which failed.
3. LVMTool does not consider the order pins were defined in the test pattern source file using the VECDEF Compiler Directive. By default, pins are initially displayed in the order pins were added, by the system software, to the built-in pin list named builtin_UsedLVM. A user defined pin list may then be selected to determine which pins are displayed and the order pins are displayed in LVMTool.

6.11 WaveformTool (MSWT)

6.11.1 Overview

This section documents the following MSWT features:

- MSWT Usage Model outlines the target MSWT usage.
- MSWT Look & Feel defines *current waveform*, and describes how MSWT's waveform windows operate.
- MSWT Toolbar File Menu describes the following menu items:
  - `File->Generate`
  - `File->Compare Waveforms`
  - `File->Open`
  - `File->Close`
  - `File->Save`, `File->Save As`
  - `File->Print`
- MSWT Toolbar View Menu describes the following controls:
  - `View->Calculator`
  - `View->Compare Controls`
  - `View->Cursor Controls`
  - `View->Graph Controls`
  - `View->Toolbar`
  - `View->Properties`
  - `View->Always On Top`
  - `View->Angles as Degrees`
  - `View->AutoSynchronize`
- MSWT Toolbar Tester Menu describes the following controls:
  - `Tester->Read Waveform`
  - `Tester->Read ACI`
  - `Tester->Read ADC`
  - `Tester->Read AWI`
  - `Tester->Read DCI`
  - `Tester->Read Modulator`
  - `Tester->Set Waveform`
  - `Tester->Synchronize`

- MSWT Toolbar Window Menu describes the standard windows management options.
- Waveform Synchronization describes how MSWT's waveform display(s) may be refreshed as waveform attributes change in the tester hardware and/or test program `Waveform*` variables.
- Waveform Calculator documents the features of MSWT's waveform calculator.
- Response to UI User Variable Signals describes how MSWT reacts to selected signals from UI.
- MSWT Programming Functions describes a set of functions which can be used to control various MSWT features.

6.11.2 MSWT Usage Model

Note: the information documented here was added to the Magnum 1 software beginning in software release h2.2.7/h1.2.7. These features were developed for the Lightning Test System, which contains waveform generation and capture hardware not available using Magnum 1/2/2x, thus, Magnum 1/2/2x applications must synthesize waveform contents using other methods. References to Lightning hardware were removed from this section, to reduce confusion. Refer to the *Lightning Programmer's Manual* for more information.

MSWT may be used to create and display waveforms commonly used by most mixed signal tests. Waveforms can be saved to disk, loaded from disk, read or written to test program waveform variables, etc.

MSWT has the ability to create waveforms, using the same waveform generation functions available from test program code. MSWT also has waveform editing features. A waveform generated in MSWT can be loaded to a test program waveform variable (`Waveform*`) which is used in a test block being debugged.

This section describes the Mixed Signal Wavetool (MSWT) used by test and product engineers to interact with waveforms. See Waveform Overview.

6.11.3 MSWT Look & Feel

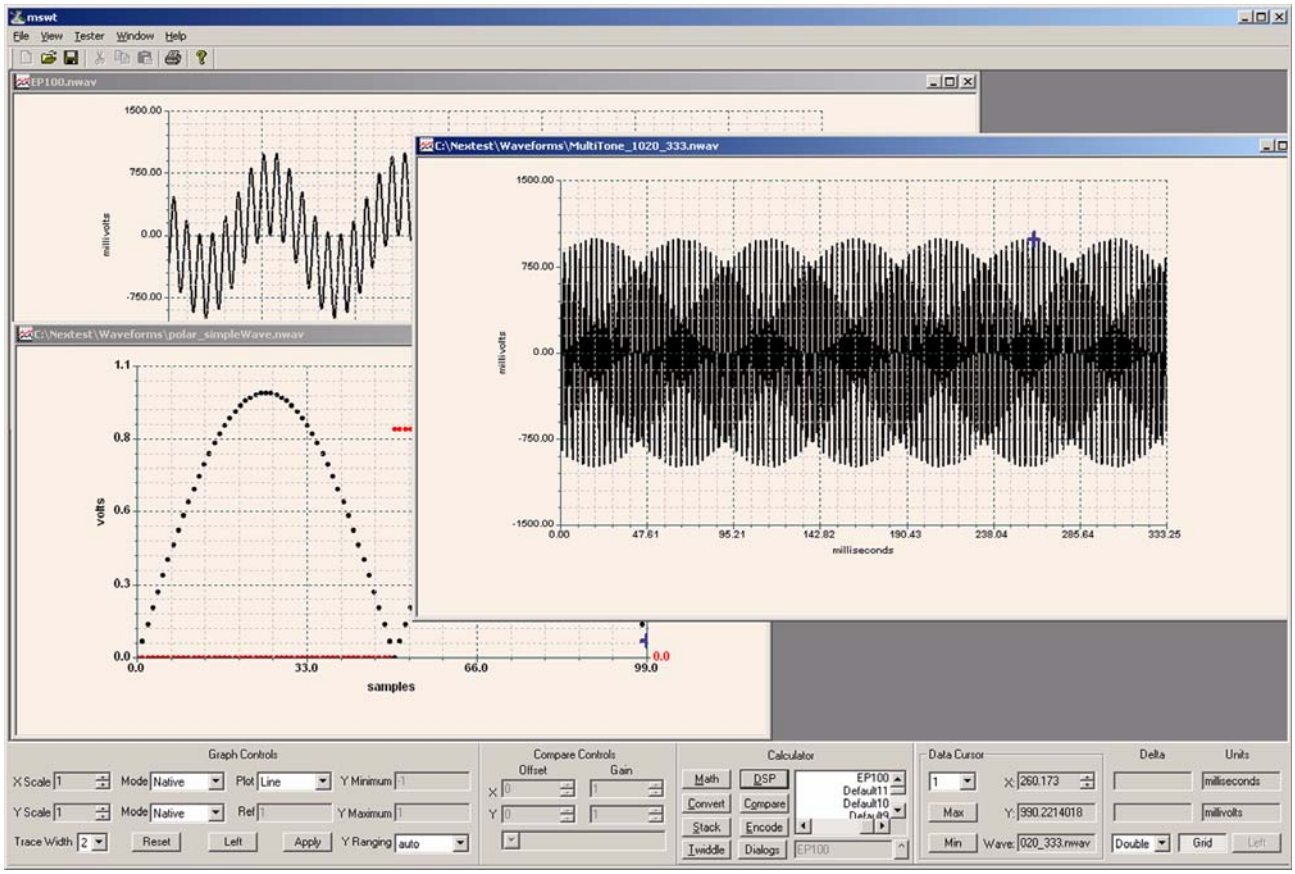


Figure-79: MSWT Main Display

MSWT follows the Microsoft MDI application model where one or more waveform *views* can be displayed, but only one waveform view may be selected at one time. The selected waveform view is considered the *current waveform*; most MSWT operations apply to the current waveform:

- The current waveform can display a waveform read from or stored to an existing waveform variable (`Waveform*`) in the test program. See `Tester->Read Waveform` and `Tester->Set Waveform`.
- Using `File->Save`, and `File->Save As` the current waveform can be stored to a file on disk.

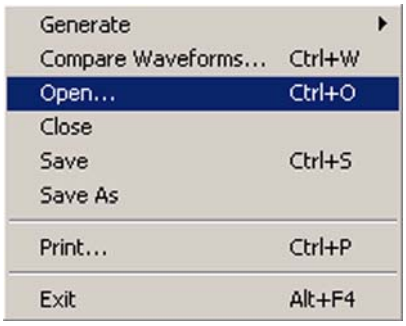
- Using `File->Open`, the current waveform can be read from a file on disk. See Waveform File Formats.
- The current waveform can be manipulated and analyzed using features accessed using the MSWT's Waveform Calculator. Many of the associated operations are based on various Waveform Functions.
- A waveform created using `File->Generate` will be displayed in a newly created current view.

Each waveform window retains its association with it's resource i.e. where the waveform was originally obtained for display. Each displayed waveform has a time stamp, which MSWT uses to know if the displayed version of the waveform is current. For waveforms which are associated with test program variables, Waveform Synchronization can be used to update the waveform display in MSWT.

6.11.4 MSWT Toolbar File Menu

See WaveformTool (MSWT).

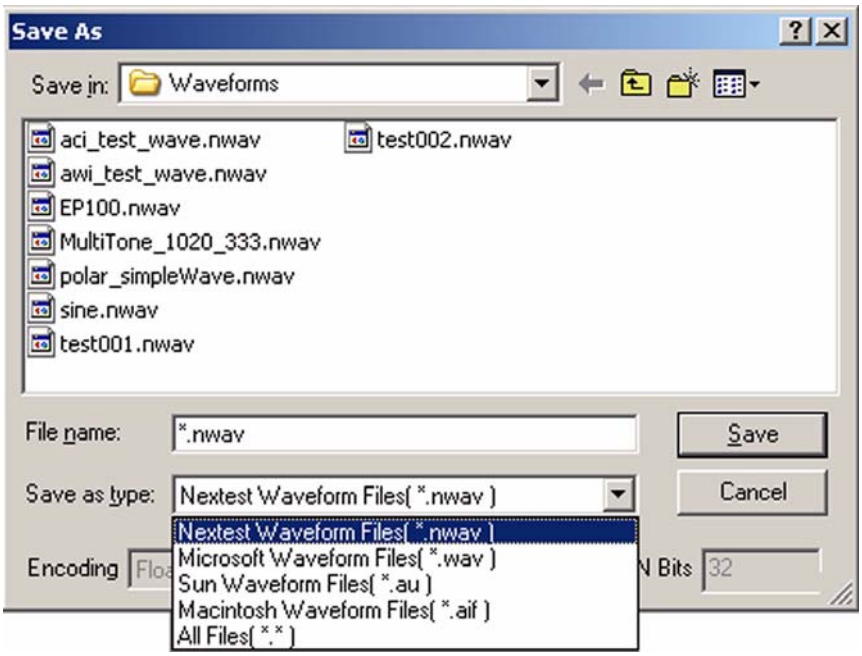
The File menu provides standard Windows file controls plus the items noted below:



`File->Generate`	Opens the File->Generate Menu, used to create a new waveform in MSWT.
`File->Compare Waveforms`	Displays the waveform File->Compare Waveforms Dialog. Also see `View->Compare Controls`.
`File->Open`	Read and display a waveform read from a file on disk. The standard file browser is used. MSWT supports reading several waveform formats, more below. Also see Waveform File Formats.

File->Close	Closes the current waveform without saving or updating original waveform source.
File->Save **File->Save As**	MSWT will save the waveform in the current view to a file on disk, in the format selected. A standard file browser is used, with the desired format selected in the browser. More below. Also see Waveform File Formats.
File->Print	Prints the current waveform using the standard print mechanisms.
File->Exit	Terminates MSWT.

When saving a waveform to a file on disk, the desired file format is selected using the **Save as type** menu in the file browser (see Waveform File Formats):



6.11.4.1 File->Generate Menu

See MSWT Toolbar File Menu.

The **File->Generate** dialog is used to create a new waveform within MSWT.

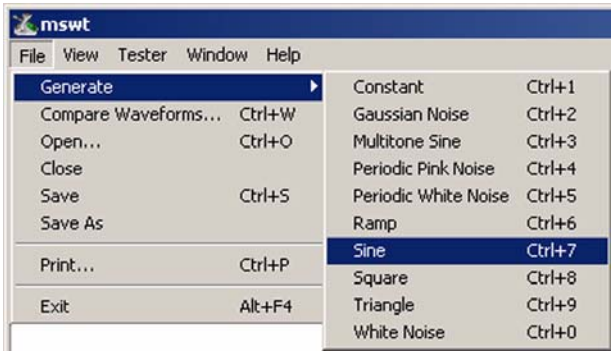


Figure-80: File->Generate Menu

Selecting any of the **File->Generate** menu options will display a dialog which is used to specify the various waveform attributes particular to the type of waveform selected. If **OK** is then clicked the new waveform will be created and displayed as the current waveform in MSWT:

Note that the generated waveform initially only exists within MSWT i.e. additional actions must be taken to save the waveform (File->Save, File->Save As), or make it available to the test program (Tester->Set Waveform).

The following waveform types can be created using the **File->Generate** menu. Note that each has a corresponding Waveform Generate Functions, which documents the parameters used to properly define that waveform type:

Type	Reference
Constant	See waveform_constant_fill(), waveform_generate_DC() and File Generate Constant Dialog.
Gaussian Noise	See waveform_generate_gaussian_noise() and File Generate Gaussian Noise Dialog.
Multitone Sine	See Generating Multi-tone Waveforms.
Periodic Pink Noise	See waveform_generate_periodic_pink_noise() and File Generate Pink/White Noise Dialog.
Periodic White Noise	See waveform_generate_periodic_white_noise() and File Generate Pink/White Noise Dialog.
Ramp	See waveform_generate_ramp() and File Generate Ramp/Triangle Dialog.

Type	Reference
Sine	See `waveform_generate_sine_wave()` and File Generate Sine Waveform Dialog.
Square	See `waveform_generate_square_wave()` and File Generate Square Waveform Dialog.
Triangle	See `waveform_generate_triangle_wave()` and File Generate Ramp/Triangle Dialog.
White Noise	See `waveform_generate_white_noise()` and File Generate Pink/White Noise Dialog.

6.11.4.2 File Generate Constant Dialog

See File->Generate Menu, MSWT Toolbar File Menu.

This dialog is used to complete the definition of a constant value waveform, see `waveform_generate_DC()`:

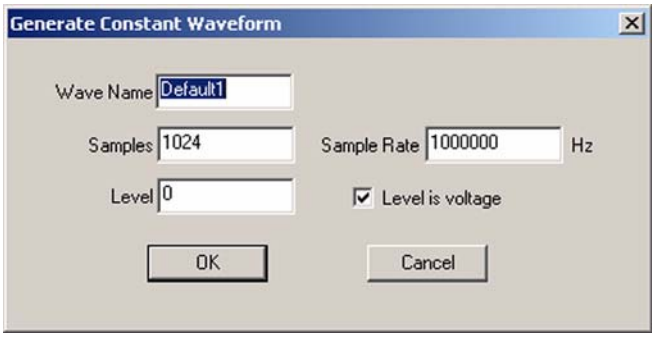


Figure-81: Generate Constant Waveform Dialog

Wave Name	Used to specify the name of the waveform being created.
Samples	Used to specify the number of sample values in the waveform being created.
Sample Rate	Used to specify the sample rate of the waveform being created.

Level	Used to specify the DC level of the constant waveform being created.
Level is Voltage	When enabled, uses `waveform_generate_DC()` to generate the waveform, otherwise uses `waveform_constant_fill()` to generate the waveform.
OK, CANCEL	Standard control to close the dialog. **OK** causes MSWT to accept and apply the values in the dialog. **Cancel** discards any changes made using the dialog.

6.11.4.3 File Generate Gaussian Noise Dialog

See File->Generate Menu, MSWT Toolbar File Menu.

This dialog is used to complete the definition of a gaussian noise waveform, see `waveform_generate_gaussian_noise()`:

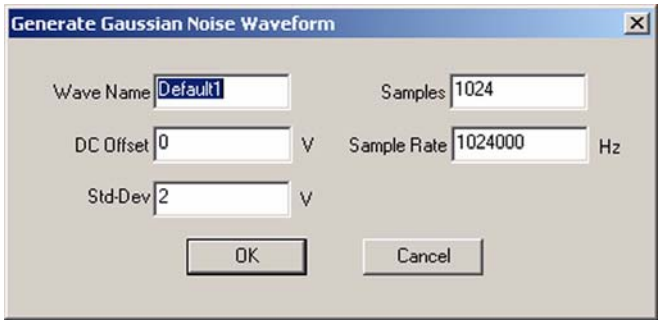


Figure-82: Generate Gaussian Noise Waveform Dialog

Wave Name	Used to specify the name of the waveform being created.
Samples	Used to specify the number of sample values in the waveform being created.
DC Offset	Used to specify the DC offset of the waveform being created.
Sample Rate	Used to specify the sample rate of the waveform being created.
StdDev	See `waveform_generate_gaussian_noise()`.
OK, CANCEL	Standard control to close the dialog. **OK** causes MSWT to accept and apply the values in the dialog. **Cancel** discards any changes made using the dialog.

6.11.4.4 Generating Multi-tone Waveforms

See File->Generate Menu, MSWT Toolbar File Menu.

Some mixed signal tests require a waveform consisting of two or more test frequencies i.e. a multi-tone waveform. MSWT supports generating multi-tone waveforms using the `File->Generate->Multitone Sine` dialog.

Two algorithms are available to generate multitone waveforms:

- Accuracy method, which optimizes the accuracy of the test tones, i.e. minimal difference between programmed vs. actual values.
- Speed method which optimizes UTP = Unit Test Period, i.e. the smallest increment of time in which the signal is coherent.

Some of the controls change based on which mode is selected:

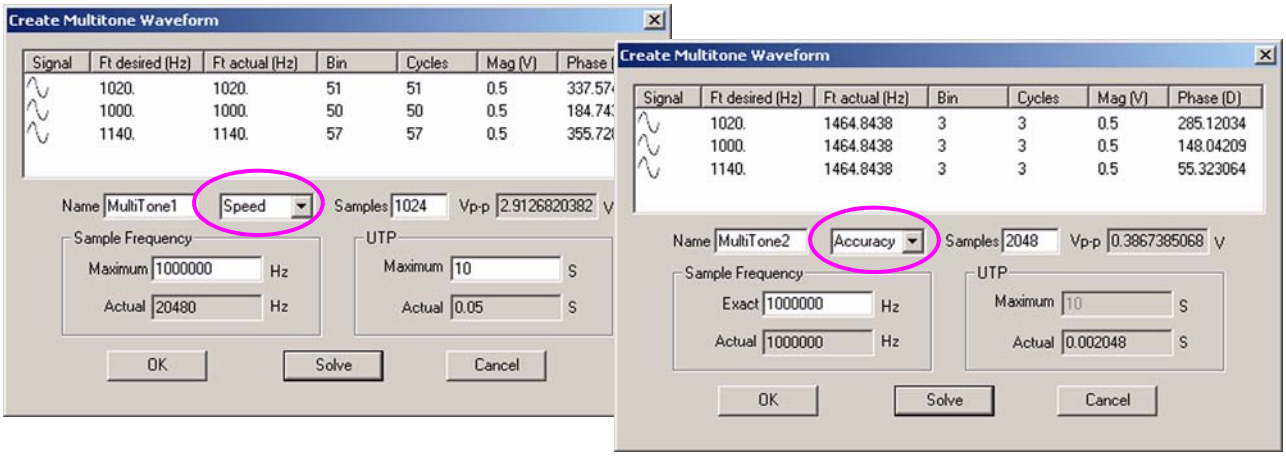


Figure-83: Create MultiTone Waveform Dialog

Dialog Control	Usage	Purpose
Signal	Computed	Arbitrary reference name for the sub waveform
Ft desired (Hz)	User Input	Desired test frequency, in Hz
Ft actual (Hz)	Computed	Actual test frequency, in Hz

Dialog Control	Usage	Purpose
Bin	Computed	FFT bin is the bin number of the signal in the frequency domain. It can occur in a bin other than bin-1 if the signal repeats in the UTP = Unit Test Period.
Cycles	Computed	Number of cycles in UTP = Unit Test Period.
Mag	User Input	Desired signal amplitude (magnitude) in Volts.
Phase	Computed	Relative signal phase, in degrees.
Name	User Input	The waveform name to be displayed in the MSWT.
Speed/Accuracy Selection	User Input	Two algorithms are available to generate multitone waveforms: - Accuracy method, which optimizes the accuracy of the test tones, i.e. minimal difference between programmed vs. actual values. - Speed method, which optimizes UTP = Unit Test Period, i.e. the smallest increment of time in which the signal is coherent.
Sample Frequency Maximum	User Input	Maximum desired sample frequency. Used in Speed mode (see Speed/Accuracy Selection and Sample Frequency Exact)
Sample Frequency Exact	User Input	Maximum desired sample frequency. Used in Accuracy mode (see Speed/Accuracy Selection and Sample Frequency Maximum)
Sample Frequency Actual	Computed	Actual sample frequency
UTP Maximum	User Input	Longest desirable UTP = Unit Test Period.
UTP Actual	Computed	Actual UTP = Unit Test Period.
Samples	User Input	Desired number of sample values to be generated

Dialog Control	Usage	Purpose
Vp-p	Computed	Actual peak-to-peak voltage amplitude
Solve	User Control	Executes the waveform generation algorithm and updates the computed values. Does not generate the waveform (use **OK**).
OK **Cancel**	User Control	Standard control to close the dialog. **OK** causes MSWT to generate and display the waveform. **Cancel** discards any changes made using the dialog.

Note: the created waveform only exists within MSWT. To use it outside MSWT, it can be saved to disk (`File->Save`, `File->Save`) or loaded to a `Waveform*` (`Tester->Set Waveform`), etc.

A multi-tone waveform is created by specifying parameters using the dialog.

The user fills in some the fields associated with each signal by clicking on the field and typing in an appropriate value. Values that apply to all signals are specified using the other controls.

Once the desired number of tones are present, and the required values have been specified, clicking the **Solve** button triggers the waveform generation algorithm and updates the computed values (adjusting the Ft Actual values, if necessary). Once the desired results are obtained, clicking the OK button will generate the composite waveform and update the display. The waveform can be viewed, saved to disk, saved to a waveform variable, and/or loaded into the tester hardware.

6.11.4.5 File Generate Pink/White Noise Dialog

See File->Generate Menu, MSWT Toolbar File Menu.

This dialog is used to complete the definition of a periodic pink noise waveform (see `waveform_generate_periodic_pink_noise()`), a periodic white noise waveform (see `waveform_generate_periodic_white_noise()`), or a white noise waveform

(see `waveform_generate_white_noise()`). Only the periodic pink noise dialog is shown:

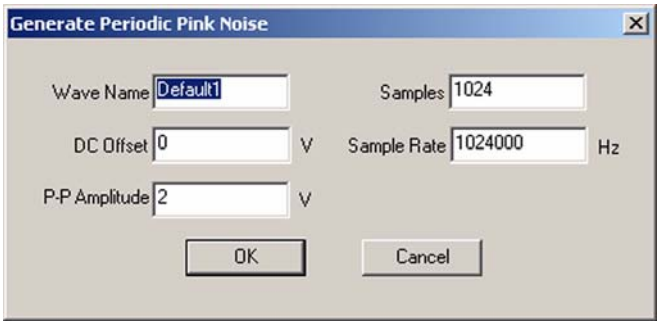


Figure-84: Generate Pink/White Noise Waveform Dialog

`Wave Name`	Used to specify the name of the waveform being created.
`Samples`	Used to specify the number of sample values in the waveform being created.
`DC Offset`	Used to specify the DC offset of the waveform being created.
`Sample Rate`	Used to specify the sample rate of the waveform being created.
`P-P Amplitude`	Used to specify the peak-to-peak amplitude of the waveform being created.
`OK, CANCEL`	Standard control to close the dialog. **`OK`** causes MSWT to generate and display the waveform. **`Cancel`** discards any changes made using the dialog.

6.11.4.6 File Generate Ramp/Triangle Dialog

See File->Generate Menu, MSWT Toolbar File Menu.

This dialog is used to complete the definition of a ramp waveform (see `waveform_generate_ramp()`) or a triangle waveform (see `waveform_generate_triangle_wave()`). Only the ramp waveform dialog is shown:

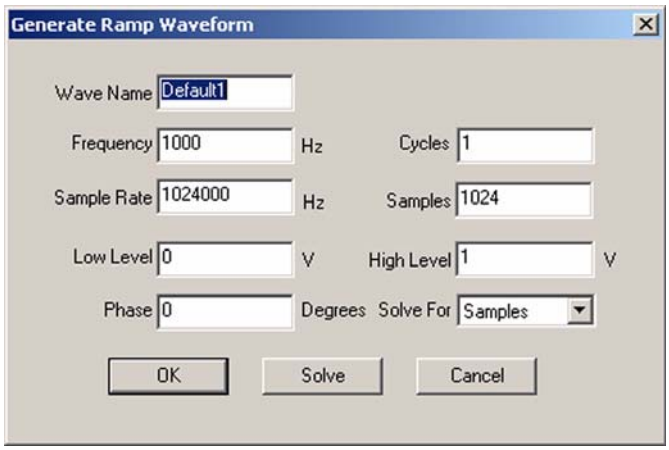


Figure-85: Generate Ramp/Triangle Waveform Dialog

`Wave Name`	Used to specify the name of the waveform being created.
`Frequency`	Used to specify the frequency of the waveform being created.
`Cycles`	Used to specify the number of ramp cycles in the waveform being created.
`Sample Rate`	Used to specify the sample rate of the waveform being created.
`Samples`	Used to specify the number of sample values in the waveform being created.
`Low Level`	Used to specify the minimum voltage of the waveform being created.
`High Level`	Used to specify the maximum voltage of the waveform being created.
`Phase`	Used to specify the phase of the waveform being created, in degrees.

Solve For	Given the specified inputs can rarely all be perfectly obtained the **Solve For** selection sets the priority: **Sample** - Given the frequency, number of cycles and sample rate, calculate the number of samples values in the waveform. **Frequency** - Given the number of samples, number of cycles and sample rate, calculate the frequency of the waveform. **Cycles** - Given the number of samples, frequency and sample rate, calculate the number of cycles in the waveform. **Sample Rate** - Given the frequency, number of cycles and number of samples, calculate the sample rate of the waveform.
Solve	Executes the waveform generation algorithm and updates the computed values. Does not generate the waveform (use **OK**).
OK, CANCEL	Standard control to close the dialog. **OK** causes MSWT to generate and display the waveform. **Cancel** discards any changes made using the dialog.

6.11.4.7 File Generate Sine Waveform Dialog

See File->Generate Menu, MSWT Toolbar File Menu.

This dialog is used to complete the definition of a sine waveform, see `waveform_generate_sine_wave()`:

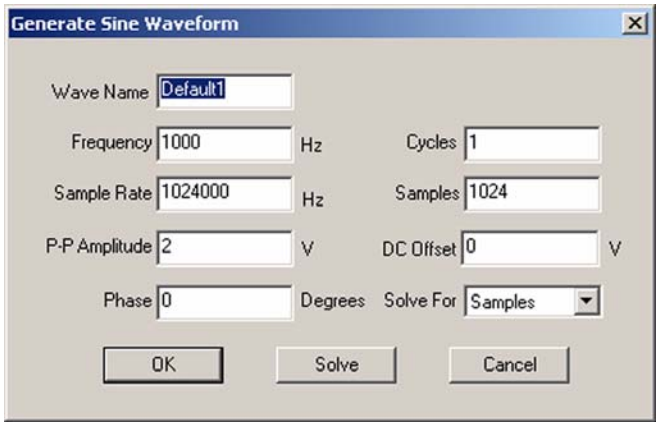


Figure-86: Generate Sine Waveform Dialog

Wave Name	Used to specify the name of the waveform being created.
Frequency	Used to specify the frequency of the waveform being created.
Cycles	Used to specify the number of ramp cycles in the waveform being created.
Sample Rate	Used to specify the sample rate of the waveform being created.
Samples	Used to specify the number of sample values in the waveform being created.
P-P Amplitude	Used to specify the peak-to-peak amplitude of the waveform being created.
DC Offset	Used to specify the DC offset of the waveform being created.
Phase	Used to specify the phase of the waveform being created, in degrees.

Solve For	Given the specified inputs can rarely all be perfectly obtained the **Solve For** selection sets the priority: **Sample** - Given the frequency, number of cycles and sample rate, calculate the number of samples values in the waveform. **Frequency** - Given the number of samples, number of cycles and sample rate, calculate the frequency of the waveform. **Cycles** - Given the number of samples, frequency and sample rate, calculate the number of cycles in the waveform. **Sample Rate** - Given the frequency, number of cycles and number of samples, calculate the sample rate of the waveform.
Solve	Executes the waveform generation algorithm and updates the computed values. Does not generate the waveform (use **OK**).
OK, CANCEL	Standard control to close the dialog. **OK** causes MSWT to generate and display the waveform. **Cancel** discards any changes made using the dialog.

6.11.4.8 File Generate Square Waveform Dialog

See File->Generate Menu, MSWT Toolbar File Menu.

This dialog is used to complete the definition of a square wave waveform, see `waveform_generate_square_wave()`:

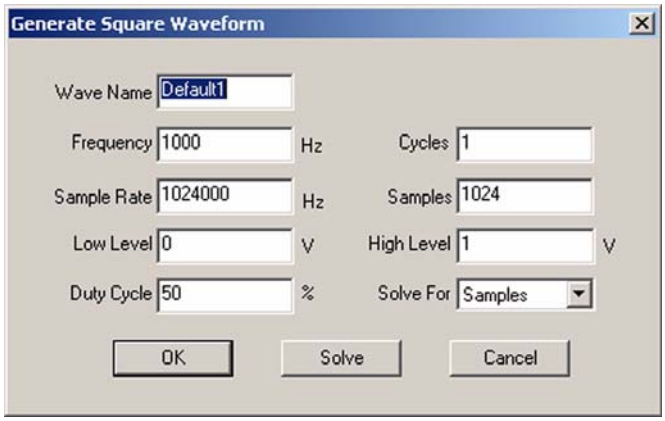


Figure-87: Generate Square Waveform Dialog

Wave Name	Used to specify the name of the waveform being created.
Frequency	Used to specify the frequency of the waveform being created.
Cycles	Used to specify the number of ramp cycles in the waveform being created.
Sample Rate	Used to specify the sample rate of the waveform being created.
Samples	Used to specify the number of sample values in the waveform being created.
Low Level	Used to specify the minimum voltage of the waveform being created.
High Level	Used to specify the maximum voltage of the waveform being created.
Duty Cycle	Used to specify the duty cycle of the waveform being created.

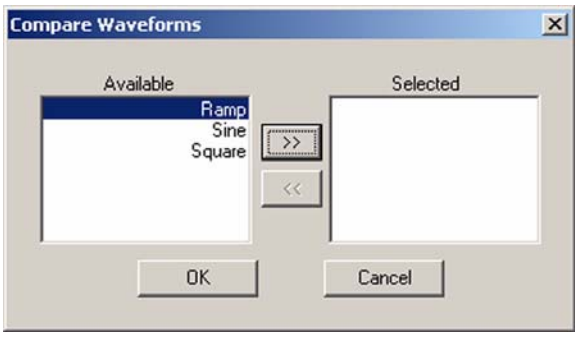
Solve For	Given the specified inputs can rarely all be perfectly obtained the **Solve For** selection sets the priority: **Sample** - Given the frequency, number of cycles and sample rate, calculate the number of samples values in the waveform. **Frequency** - Given the number of samples, number of cycles and sample rate, calculate the frequency of the waveform. **Cycles** - Given the number of samples, frequency and sample rate, calculate the number of cycles in the waveform. **Sample Rate** - Given the frequency, number of cycles and number of samples, calculate the sample rate of the waveform.
Solve	Executes the waveform generation algorithm and updates the computed values. Does not generate the waveform (use **OK**).
OK, CANCEL	Standard control to close the dialog. **OK** causes MSWT to generate and display the waveform. **Cancel** discards any changes made using the dialog.

6.11.4.9 File->Compare Waveforms Dialog

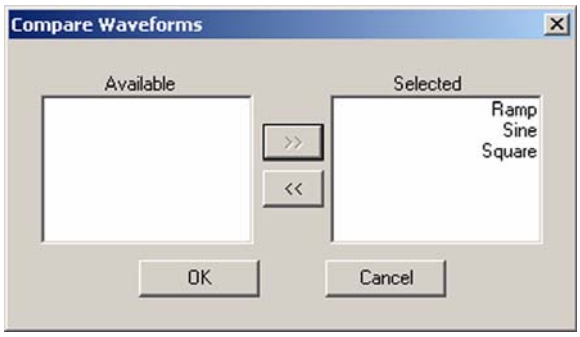
See `File->Compare Waveforms`, MSWT Toolbar File Menu.

It is often useful to view two or more waveforms in the same window, at the same time, using the same relative X and Y axis scales. MSWT's Compare Waveforms window is used for this purpose, which provides a primarily visual comparison capability.

When `File->Compare Waveforms` is selected, the dialog shown below is displayed:



Before Selection



After Selection

Figure-88: File->Compare Dialog

The user selects one or more waveforms by moving the desired waveform name(s) from the **`Available`** list to the **`Selected`** list, using the dialog's arrow controls. Up to 4 waveforms can be added to the **`Selected`** list. When **`OK`** is clicked the **`Selected`** waveforms will be displayed in MSWT's Compare Waveforms window.

The controls displayed by invoking View->Compare Controls can be used to individually manipulate each waveform displayed in the Compare Waveforms window.

Using the example above results in the following:

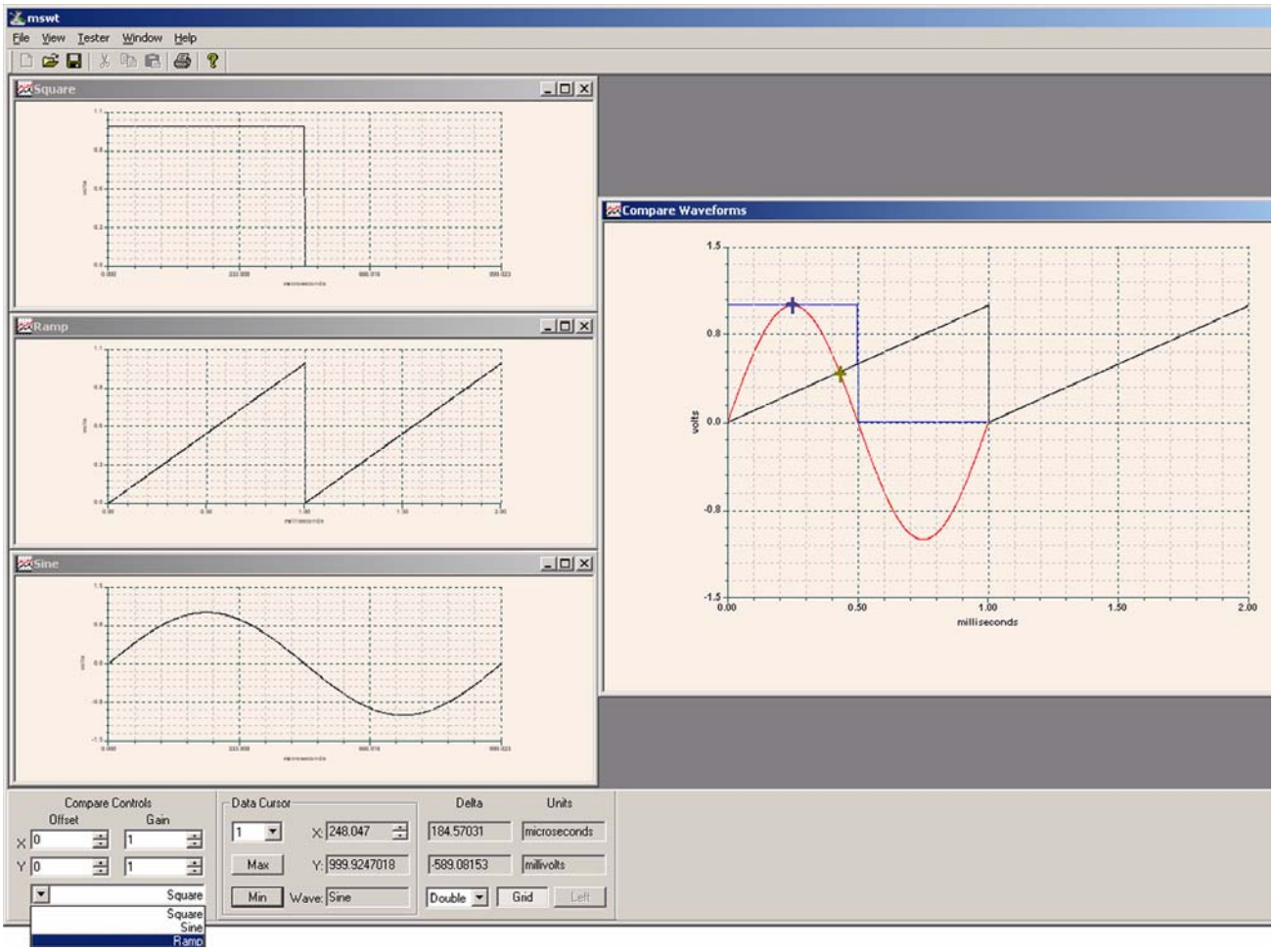


Figure-89: Compare Waveform Result

In the example above the waveforms in the Compare Waveforms window are displayed with default settings but can be independently moved in the X-axis and/or Y-axis using the Compare Controls, which are displayed by selecting `View->Compare Controls`.

The two cursors seen in the Compare Waveforms window are added using the mouse:

- Left-mouse click inserts cursor 1.
- Shift-left-mouse click inserts cursor 2.

The Cursor Controls are displayed by selecting `View->Cursor Controls`. When two cursors are displayed the delta X/Y values indicate the offset between the cursors and the controls can be used to move a cursor to a specific value, etc.

6.11.5 MSWT Toolbar View Menu

See WaveformTool (MSWT).

These options are used to hide or display selected controls or to enable operating modes:

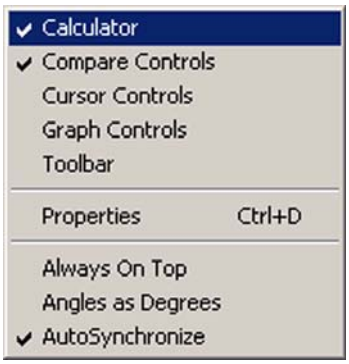


Figure-90: File->View Menu

The following menu items are available in the MSWT Toolbar View Menu:

View->Calculator	Displays Waveform Calculator controls.
View->Compare Controls	Displays controls which are used to manipulate the waveforms displayed in MSWT's Compare Waveforms window. See View->Compare Controls
View->Cursor Controls	Displays controls which are used to manipulate a user inserted cursor displayed in the current waveform window. See View->Cursor Controls
View->Graph Controls	Displays controls which are used to control display options for the current waveform window. See View->Graph Controls
View->Toolbar	Controls display of the MSWT's main tool bar.
View->Properties	Displays the Waveform Properties dialog, which describes the current waveform. See View->Properties Dialog.

View->Always On Top	Controls whether other graphic tools may be placed on top of MSWT
View->Angles as Degrees	Controls whether [phase] angle values are displayed in degrees or radians. Does not affect waveforms already displayed.
View->AutoSynchronize	Used to enable/disable auto synchronization. See Waveform Synchronization.

6.11.5.1 View->Compare Controls

See MSWT Toolbar View Menu.

Displayed using `View->Compare Controls`. Used to manipulate the display of waveforms in MSWT's Compare Waveform window, which must be selected as the active window. Also see `File->Compare Waveforms`, and Calculator Compare Menu:

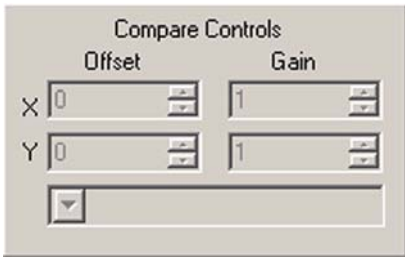


Figure-91: View->Compare Controls

X Offset **Y Offset**	Moves the selected waveform in the X/Y axis. Values can be incremented/decremented using the arrow controls or a literal value can be entered.

X Gain	Changes the frequency of the selected waveform. Value can be incremented/decremented using the arrow controls or a literal value can be entered.
Y Gain	Changes the amplitude of the selected waveform. Value can be incremented/decremented using the arrow controls or a literal value can be entered.
Waveform Selection Menu	Used to select one of the waveforms currently displayed in MSWT's Compare Waveform window. Changing any of the gain or offset values (above) modifies the display of the currently selected waveform.

6.11.5.2 View->Cursor Controls

See MSWT Toolbar View Menu.

Displayed using `View->Cursor Controls`. Used to interact with cursors interactively inserted into waveform displays using the mouse :

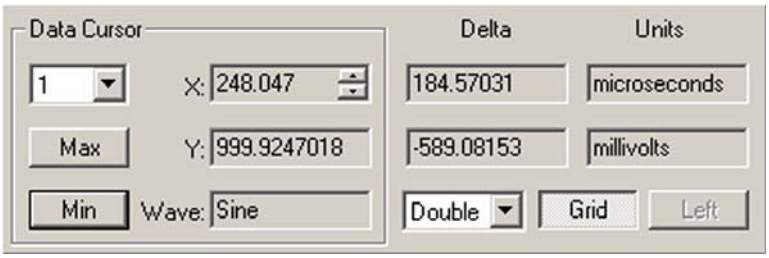


Figure-92: View->Cursor Controls

`Data Cursor`	Used to select the cursor to manipulate. A 2nd cursor is added to a waveform using shift-left-mouse.
`X Value` **`Y Value`**	X/Y axis value at selected cursor's position. Can be used to move the cursor to a specified value or to increment/decrement the cursor position by one resolution point
`Max` **`Min`**	Position the selected cursor at the maximum/minimum value in the waveform. If that position is out of the window, the display will scroll to display the cursor position in the waveform.
`Wave`	Displays the name of the current waveform.
`X Delta` **`X Units`**	Displays the difference, in **`X Units`**, between two cursor positions in the X axis. Disabled until a 2nd cursor is added using shift-left-mouse.
`Y Delta` **`Y Units`**	Displays the difference, in **`Y Units`**, between two cursor positions in the Y axis. Disabled until a 2nd cursor is added using shift-left-mouse.

Value Base	Selects the numerical base being used to display the X/Y cursor position values. Options include: **Double**, **Integer**, **Hex**. Note that this affects the values displayed in the Cursor Controls dialog only.
Grid	This is an On/Off control, used to show/hide the grid in the current waveform.
Left	When a waveform window contains two sets of data (for example, polar notation), controls whether the cursor is attached to data values displayed on the left side or right side of the display.

6.11.5.3 View->Graph Controls

See MSWT Toolbar View Menu.

Displayed using View->Graph Controls. Used to manipulate the display of waveforms in MSWT :

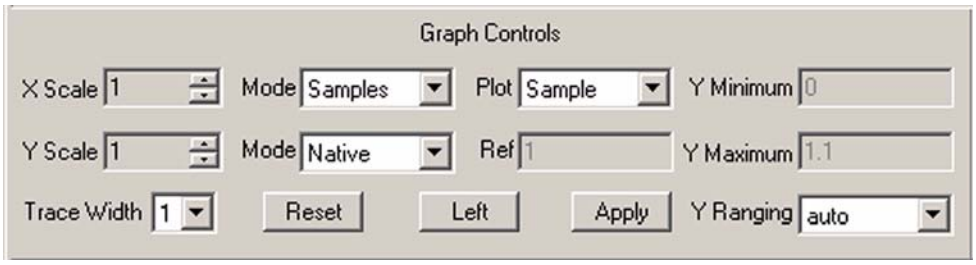


Figure-93: View->Graph Controls

X Scale	Used to change the displayed X Scale value, effectively zooming in/out on X axis of the current waveform. Scroll-bars will appear if appropriate.
Y Scale	Used to change the displayed Y Scale value, effectively zooming in/out on Y axis of the current waveform. Scroll-bars will appear if appropriate.
X Mode	Used to select the mode in which X axis values are displayed. Options are: Native and Samples, see below.
Y Mode	Used to select the mode in which Y axis values are displayed. Options are: Native, Magnitude, dbAuto and dbAbsolute. See below.
Plot	Used to select the line type used to plot the waveform sample data. Options include: **Line**, **LineMark**, **Sample**, **Staircase**, **Bar**.

Ref	When Y Mode = dbAuto, MSWT sets this field after calculating the DB reference (locating the bin with the highest magnitude). When Y Mode = dbAbsolute mode, the user enters the desired DB reference.
Y Minimum	See **Y Ranging** below.
Y Maximum	See **Y Ranging** below.
Y Ranging	Normally the plotting software *auto ranges*. Sometimes when waveforms are being compared, it is useful to fix the axis ranges, which is done using the **Y Ranging** selections: **Auto** - auto ranging **Fixed Left** - the left axis range is controlled by **Y Minimum** and **Y Maximum**. **Fixed Right** - the right axis range is controlled by **Y Minimum** and **Y Maximum**.
Trace Width	Used to select the line width used to plot the waveform sample data. Options include: 1, 2 and 4.
Reset	Resets all **View->Graph Controls** to default values.
Left	Toggles display mode between **Left**, **Right**, and **Both**. Useful when a given window displays multiple plots, to control which plot(s) are displayed.
Apply	Used when Y Mode = dbAbsolute to apply a user specified dB reference value.

Graph mode options determine the mode in which X and Y axis values are displayed. The **X Mode** options are:

Native	The X axis label is set from the waveform's X_units value. See Waveform Units.
Samples	The X axis label is sample count (SCALE_SAMPLES), regardless of the waveforms X_units value.

The **Y Mode** options are:

Native	The Y axis label is set from the waveform's Y_units value. See Waveform Units.
Magnitude	The Y axis label and plotted values are determined by processing the waveform through the `waveform_magnitudes()` function. This is useful to display the magnitudes of an FFT.
dbAuto	Automatically calculates the waveform's Y axis label and displayed values by finding the highest bin magnitude, then calculating dB from that value.
dbAbsolute	The user provides the reference used to establish the Y axis label and displayed values i.e. dB relative to a user supplied reference value.

6.11.5.4 View->Properties Dialog

See MSWT Toolbar View Menu.

The `View->Properties` dialog is used to display key information about the current waveform. This dialog can be invoked two ways:

- Using the right mouse button on the current waveform
- Using `View->Properties` from the MSWT Toolbar View Menu:

Either method will display the WaveProperties dialog:

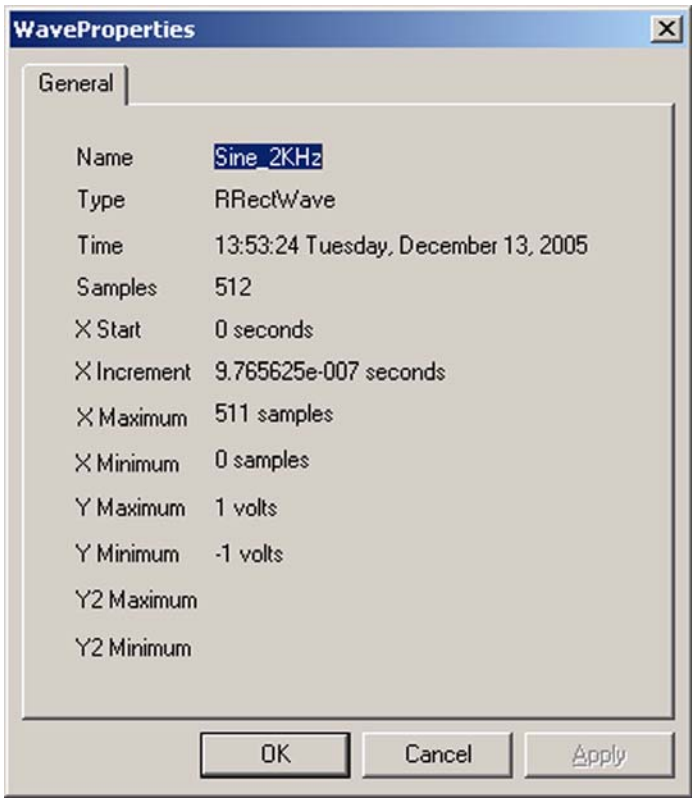


Figure-94: View->Properties Dialog

If `View->Properties` is invoked when MSWT's Compare Waveform window is selected the Scope Properties dialog is displayed:

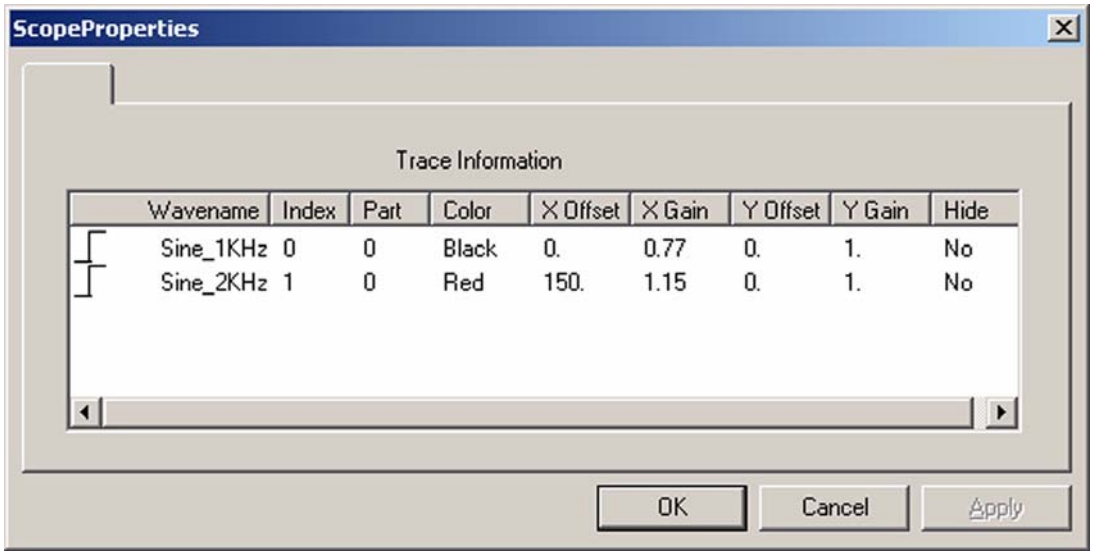


Figure-95: Scope Properties Dialog

This allows the user to view information about the waveforms displayed in the Compare Window. The `Hide` control is a toggle which can be used to hide/show the waveform.

6.11.6 MSWT Toolbar Tester Menu

See WaveformTool (MSWT).

The Tester menu is used to:

- Select waveforms to be displayed in MSWT, either from test program variables or from hardware which stores waveforms.
- Send the current waveform to an existing waveform variable in the test program, or tester hardware which stores waveforms.
- Update (synchronize) the waveforms
- Update any waveforms displayed in MSWT which were read from test program variables from hardware.
- Execute a selected test block.

The Tester menu contains the following options:

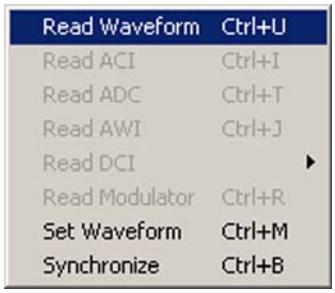


Figure-96: Tester Menu

`Tester->Read Waveform`	Displays the Tester->Read Waveform Dialog which is used to select a waveform (`Waveform*`) from the test program to be displayed.
`Tester->Set Waveform`	Displays the Tester->Set Waveform Dialog which is used to select a waveform (`Waveform*`) to be updated from the current waveform. This CAN over-write existing waveform parameters. This option is not shown unless MSWT has at least one waveform displayed.
`Tester->Synchronize`	Refreshes the display of waveforms which were read from the test program (using `Tester->Read Waveform`) . No additional dialogs are used. See Waveform Synchronization.
Note: the disabled controls in the dialog are only enabled when using Lightning hardware.	

6.11.6.1 Tester->Read Waveform Dialog

See MSWT Toolbar Tester Menu.

Displayed using `Tester->Read Waveform`. Displays all currently defined waveforms (`Waveform*`) in the loaded test program. Selecting a waveform and clicking **`OK`** causes

that waveform to be read from the test program and displayed in MSWT as the current waveform:

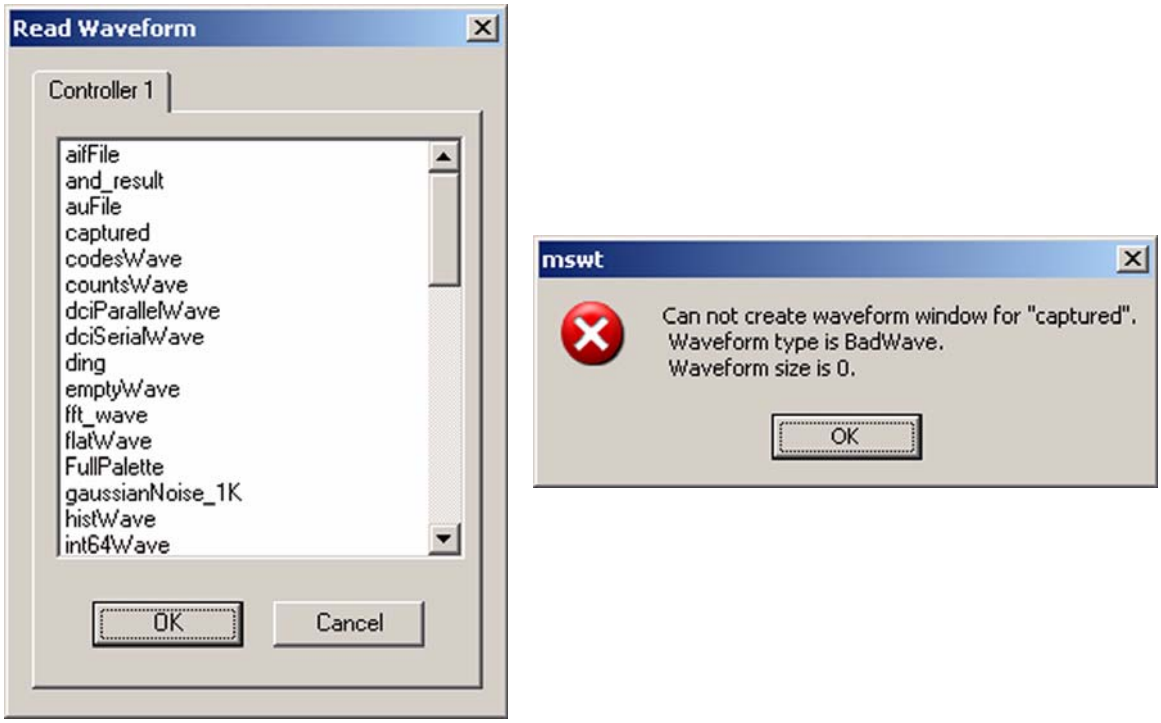


Figure-97: Tester->Read Waveform Dialogs

If the selected waveform has not been initialized or contains bad data the error dialog shown above is displayed and no waveform will be displayed in MSWT.

6.11.6.2 Tester->Set Waveform Dialog

See MSWT Toolbar Tester Menu.

Displayed using `Tester->Set Waveform`. Used to set (write) a waveform in the test program from the current waveform displayed in MSWT. Note the following:

- Displays all waveforms (`Waveform*`) currently defined in the loaded test program (whether initialized/valid or not).
- Selecting a waveform and clicking **OK** causes that waveform to be updated from MSWT's current waveform.
- If the selected waveform already contains sample values the user will be prompted to confirm over-writing of the selected waveform. Clicking **OK** over-writes any existing waveform parameters of the selected waveform.
- The `Tester->Set Waveform` is not displayed until MSWT displays at least one waveform:

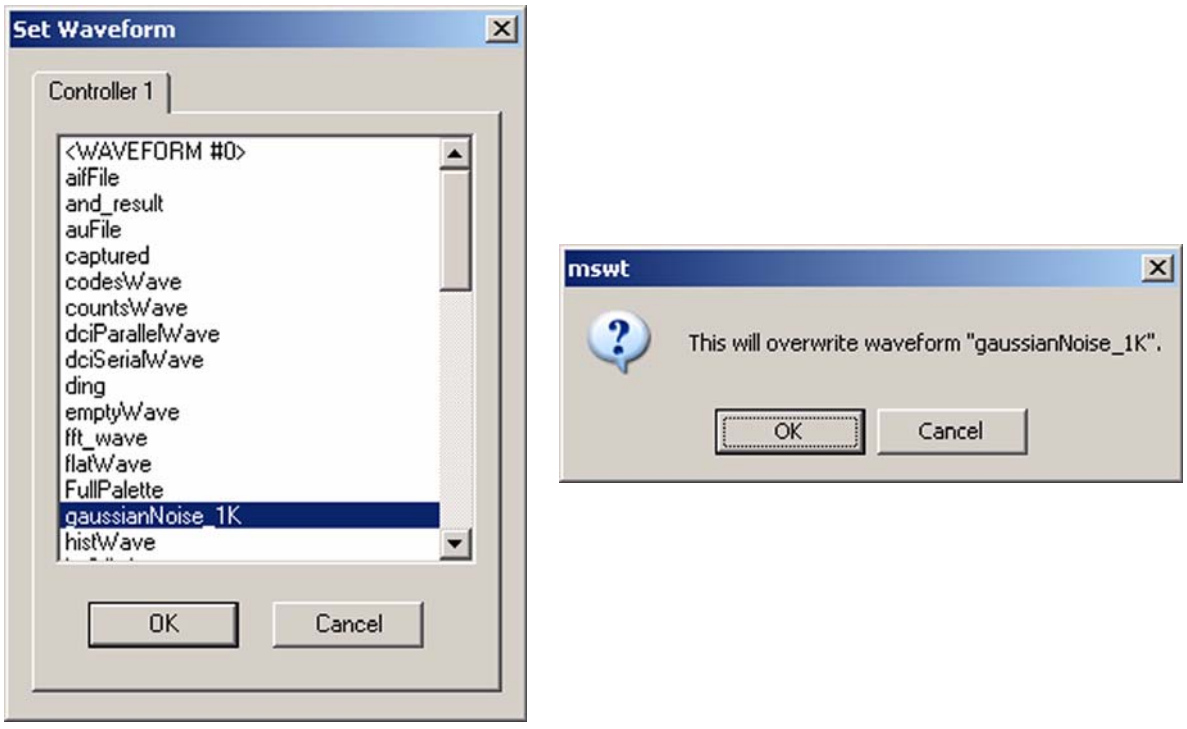


Figure-98: Tester->Set Waveform Dialog & Confirmer

6.11.7 MSWT Toolbar Window Menu

See WaveformTool (MSWT).

The MSWT Windows menu provides standard Windows options used to tile, cascade, arrange icons, etc. This menu is not displayed until a MSWT displays at least one waveform:

Figure-99: Window Menu

6.11.8 Waveform Synchronization

See WaveformTool (MSWT).

Some waveforms displayed in MSWT will represent volatile data. Software waveform definitions are also subject to change, from executing test program code.

When using MSWT, if the test program or hardware state changes, it may be necessary to update (synchronize) the appropriate MSWT waveform display windows. Synchronization will occur when:

- In MSWT, `Tester->Synchronize` is selected.
- UI sends the `ui_TestDone` signal (see Response to UI User Variable Signals) and auto-synchronization is enabled (default, or using `View->AutoSynchronize`).

Synchronization causes MSWT to update any waveforms which were displayed by reading:

- A waveform variable (`Waveform*`) using `Tester->Read Waveform`.

For each of these, the original waveform source is read to update its corresponding waveform window. This may or may not result in a change in the waveform being displayed, depending on whether the waveform source has been modified since the last synchronization occurred.

The following waveforms will not be updated:

- Waveforms displayed by reading a waveform file using `File->Open`.
- Waveforms created using `File->Generate`.
- Waveforms generated by the Waveform Calculator.

6.11.9 Waveform Calculator

See MSWT Toolbar View Menu.

MSWT's waveform calculator can be used to perform various mathematical and DSP operations, conversions, user defined operations, etc. on waveforms displayed in MSWT.

This section is divided into the following subsections:

6.11.9.1 Overview

See Waveform Calculator.

In general, using MSWT's waveform calculator, the sample values of a specified waveform are processed by a selected operation. The available operations include many of the same Waveform Functions usable in test program code.

For example, below, the upper waveform is a 3-tone sine wave (1.02KHz, 1.04KHz, 1.0KHz, 1024 samples), generated using Generating Multi-tone Waveforms. Below it is the result of performing a Real FFT, using the Calculator DSP Menu RealFFT option:

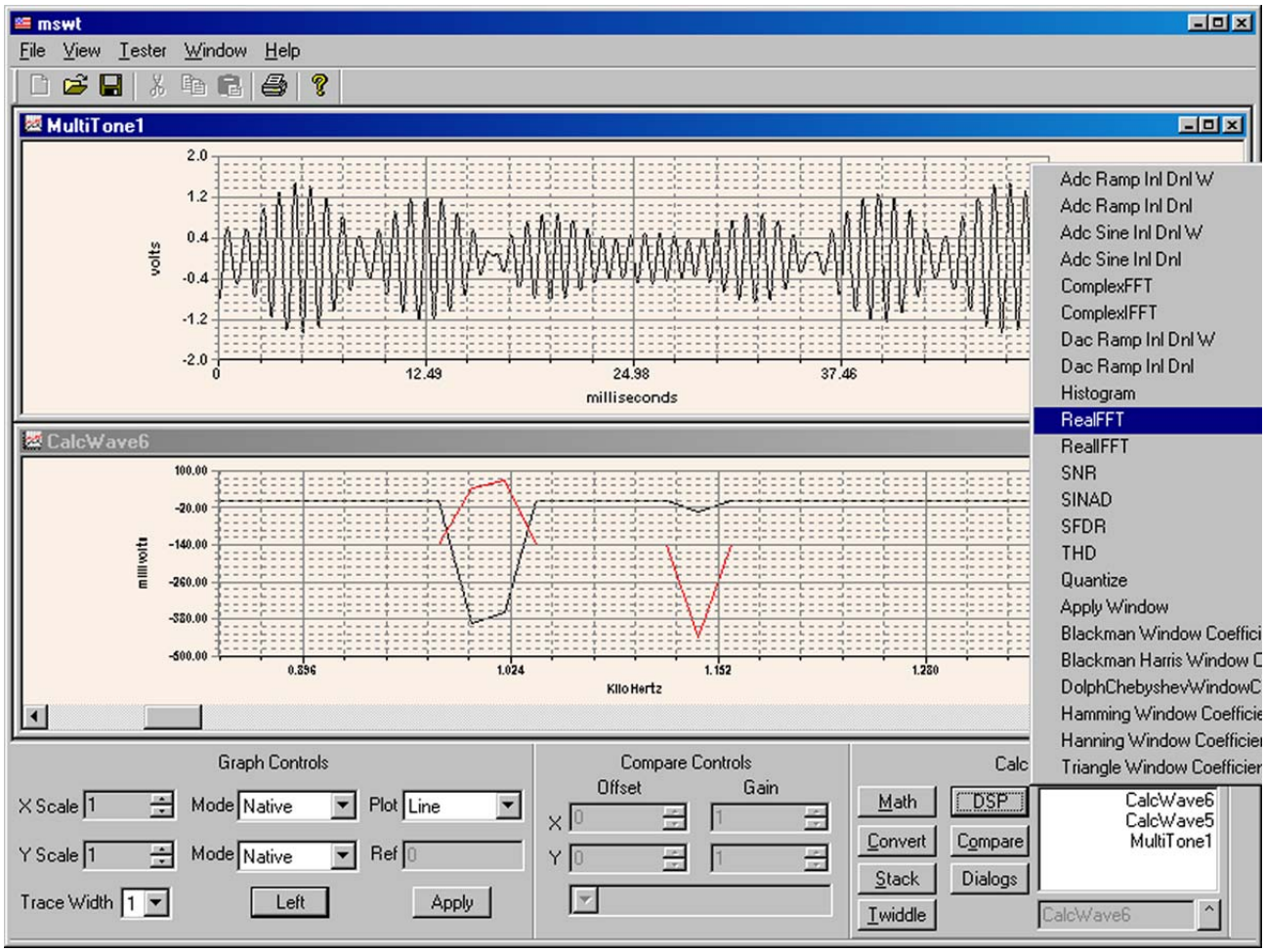


Figure-100: Waveform Calculator Example

Using a stack paradigm, MSWT's waveform calculator can use RPN or standard operations. Stack operations are displayed in a panel, to show waveforms and/or values to be used by an operation selected using the adjacent buttons.

The basic operational model is as follows:

1. The user pushes one or more items onto the stack and clicks an operation button.
2. MSWT performs the operation and pushes the result onto the stack.

3. If a calculator operation result is a Waveform, MSWT opens a new waveform window to display the new waveform. Not all operations result in a waveform (THD calculations for example). The calculator will provide an output window for displaying numerical results.

The available controls and operations are described in the following sections.

6.11.9.2 Calculator Controls

See Waveform Calculator.

Each of the buttons available in the Waveform Calculator panel invoke a menu of additional calculator options:

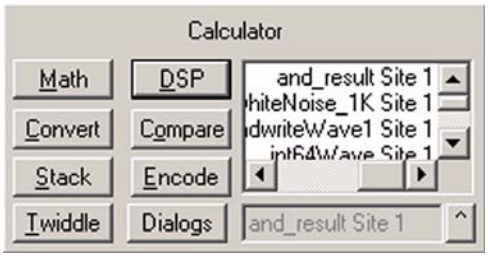


Figure-101: Calculator Controls

Math	Displays the Calculator Math Menu, which is used to select mathematical operations to be performed on the value/waveform at the top of the stack.
DSP	Displays the Calculator DSP Menu, which is used to select DSP operations to be performed on the value/waveform at the top of the stack.
Convert	Displays the Calculator Convert Menu, which is used to select conversion operations to be performed on the value/waveform at the top of the stack.
Compare	Displays the Calculator Compare Menu, which is used to manipulate the X/Y Offset and Gain parameters affecting waveforms displayed in the Compare Waveforms window.
Stack	Displays the Calculator Stack Menu, which is used to directly interact with the calculator stack.

Encode	Displays the Calculator Encode Menu, which is used to compress (encode) on the value/waveform at the top of the stack.
Twiddle	Displays the Calculator Twiddle Menu, which is used to select boolean and other operations to be performed on the value/waveform at the top of the stack.
Dialogs	Displays the Calculator Dialogs/RPN Option, which is used to select MSWT's calculator operating mode.

6.11.9.3 Calculator Math Menu

See Waveform Calculator, Calculator Controls.

These operations each generate and display a new waveform, using sample values taken from a specified waveform (default = current waveform) and modified based on the selected operation, which is based on the Waveform Functions usable in test program code:

Figure-102: Calculator Math Menu

Absolute Value	See `waveform_absolute_value()`
Add **AddScalar**	See `waveform_add()`
Arithmetic Mean	See `waveform_arithmetic_mean()`
Average	See `waveform_average()`
Clamp	See `waveform_clamp()`.

Clip Lower **Clip Upper**	See waveform_clip_upper(), waveform_clip_lower()
Differencing	See waveform_differencing()
Divide **DivideScalar**	See waveform_divide()
Exp **Exp10**	See waveform_exp(), waveform_exp10()
Geometric Mean	See waveform_geometric_mean()
Linear Regression	See waveform_linear_regression()
Log **Log10**	See waveform_log(), waveform_log10()
Max **Min**	See waveform_min_max()
Median	See waveform_median()
Multiply **MultiplyScalar**	See waveform_multiply()
Negate	See waveform_negate()
Power (double) **Power (integer)**	See waveform_power()
Reciprocal	See waveform_reciprocal()
Rescale	See waveform_rescale()
RMS	See waveform_rms()
Sort	See waveform_sort()
Standard Deviation	See waveform_standard_deviation()
Subtract **SubtractScalar**	See waveform_subtract()
Sum	See waveform_sum()

Sum Of Squares	See waveform_sum_of_squares()
Summing	See waveform_summing()
Variance	See waveform_variance()

6.11.9.4 Calculator DSP Menu

See Waveform Calculator, Calculator Controls.

These operations each generate and display a new waveform, using sample values taken from a specified waveform (default = current waveform) and processed based on the

selected operation, which is based on the Waveform Functions usable in test program code:

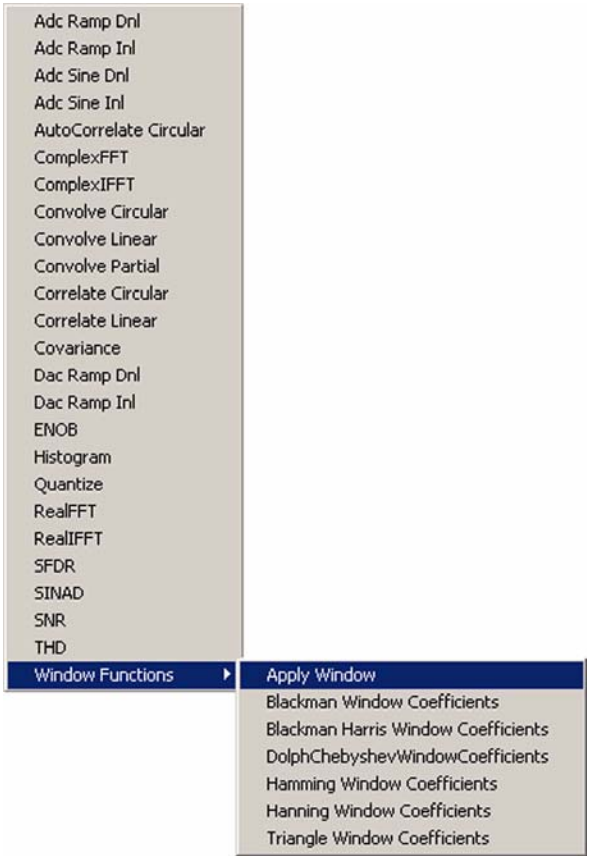


Figure-103: Calculator DSP Menu

Adc Ramp Dnl **Adc Ramp Inl**	See `waveform_adc_ramp_inl_dnl()`
Adc Sine Dnl **Adc Sine Inl**	See `waveform_adc_sine_inl_dnl()`
AutoCorrelate Circular	See `waveform_autocorrelate_circular()`
ComplexFFT	See `waveform_complex_fft()`
ComplexIFFT	See `waveform_complex_ifft()`
Convolve Circular	See `waveform_convolve_circular()`
Convolve Linear	See `waveform_convolve_linear()`

Convolve Partial	See waveform_convolve_partial()
Correlate Circular	See waveform_correlate_circular()
Correlate Linear	See waveform_correlate_linear()
Covariance	See waveform_covariance()
Dac Ramp Dnl **Dac Ramp Inl**	See waveform_dac_ramp_inl_dnl()
ENOB	See waveform_enob()
Histogram	See waveform_histogram().
Quantize	See waveform_quantize()
RealFFT	See waveform_real_fft().
RealIFFT	See waveform_real_ifft().
SFDR	See waveform_sfdr().
SINAD	See waveform_sinad().
SNR	See waveform_snr().
THD	See waveform_thd().
Apply Window	See waveform_apply_window()
Blackman Window Coeffients	See waveform_blackman_window_coefficients() and Waveform Window Functions
Blackman Harris Window Coeffients	See waveform_blackman_harris_window_coefficients() and Waveform Window Functions
DolphChebyshev Window Coeffients	See waveform_dolph_chebyshev_window_coefficients() and Waveform Window Functions
Hamming Window Coeffients	See waveform_hamming_window_coefficients() and Waveform Window Functions
Hanning Window Coeffients	See waveform_hanning_window_coefficients() and Waveform Window Functions
Triangle Window Coeffients	See waveform_triangle_window_coefficients() and Waveform Window Functions

6.11.9.5 Calculator Convert Menu

See Waveform Calculator, Calculator Controls.

These operations each generate and display a new waveform, using sample values taken from a specified waveform (default = current waveform) and modified based on the selected operation, which is based on the Waveform Functions usable in test program code:

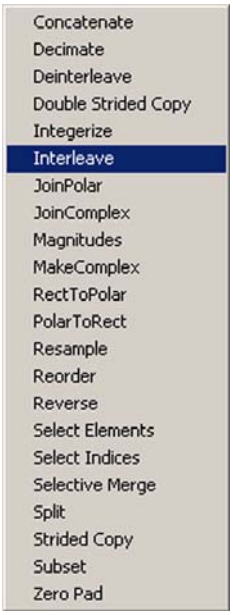


Figure-104: Calculator Convert Menu

Concatenate	See `waveform_concat()`.
Decimate	See `waveform_decimate()`
Deinterleave	See `waveform_deinterleave()`
Double Strided Copy	See `waveform_double_strided_copy()`
Integerize	See `waveform_integerize()`
Interleave	See `waveform_interleave()`
JoinPolar	See `waveform_join_polar()`.
JoinComplex	See `waveform_join_complex()`.
Magnitudes	See `waveform_magnitudes()`.
MakeComplex	See `waveform_make_complex()`.

RectToPolar	See waveform_rectangular_to_polar()
PolarToRect	See waveform_polar_to_rectangular().
Resample	See waveform_resample()
Reorder	See waveform_reorder()
Reverse	See waveform_reverse()
Select Elements	See waveform_select_elements()
Select Indices	See waveform_select_indices()
Selective Merge	See waveform_selective_merge()
Split	See waveform_split().
Strided Copy	See waveform_strided_copy()
Subset	See waveform_subset().
Zero Pad	See waveform_zero_pad()

6.11.9.6 Calculator Compare Menu

See Waveform Calculator, Calculator Controls.

These operations are used to perform various value or waveform comparisons. Each option presents a dialog used to enter value(s) or select waveform(s) consistent with the waveform function noted, which is based on the Waveform Functions usable in test program code:

EQ Scalar **EQ**	Equality test. See `waveform_eq()`
GE Scalar **GE**	Greater-than or equal-to test. See `waveform_ge()`
GT Scalar **GT**	Greater-than test. See `waveform_gt()`
LE Scalar **LE**	Less-than or equal-to test. See `waveform_le()`
LT Scalar **LT**	Less-than test. See `waveform_lt()`
Within Bounds SS **Within Bounds WS** **Within Bounds SW** **Within Bounds WW**	Within-bounds test. See `waveform_within_bounds()`. S = Scalar W = Waveform i.e. WS = Waveform to Scalar evaluation

6.11.9.7 Calculator Stack Menu

See Waveform Calculator, Calculator Controls.

These operations directly interact with the Waveform Calculator stack:

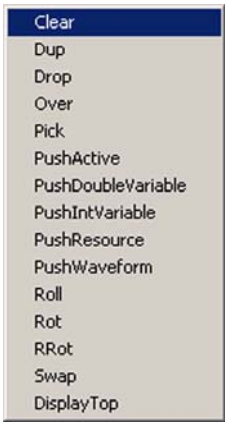


Figure-105: Calculator Stack Menu

Clear	Clears all values from the stack.
Dup	Pushes the previous stack value onto the stack (duplicate)
Drop	Removes only the last value on the stack (pop).
Over	Pushes the 2nd value on the stack onto the top of the stack.
Pick	Pushes value at the indexth position on the stack onto the top of the stack. See Calculator Stack Pick Dialog.
PushActive	Pushes the current waveform onto the stack.
PushDoubleVariable	See Calculator Stack PushDoubleVariable Dialog. Presents a dialog displaying all DOUBLE_VARIABLE user variables. A test program must be loaded. When **OK** is clicked, the value of the selected variable is pushed onto the stack.
PushIntVariable	See Calculator Stack PushIntVariable Dialog. Presents a dialog displaying all INT_VARIABLE user variables. A test program must be loaded. When **OK** is clicked, the value of the selected variable is pushed onto the stack.

PushResource	See Calculator Stack PushResource Dialog. Presents a dialog displaying all `Waveform*` variables. A test program must be loaded. When **OK** is clicked, the selected waveform is pushed onto the stack
PushWaveform	See Calculator Stack PushWaveform Dialog. Presents a dialog displaying all the waveforms currently displayed in MSWT. When **OK** is clicked, the selected waveform is pushed onto the stack and made the current window.
Roll	Removes the value at the indexth position in the stack and pushes it onto the top of the stack. See Calculator Stack Roll Dialog.
Rot	Rotates the values currently on the stack, which must contain at least 3 values. The last value on the stack becomes the first value. No dialog is presented for this option.
RRot	Reverse rotates the values currently on the stack, which must contain at least 3 values. The first value on the stack becomes the last value. No dialog is presented for this option.
Swap	Reverses the last two entries on the stack. No dialog is presented for this option.
Display Top	Causes the first waveform on the stack to become the current waveform. No dialog is presented for this option.

6.11.9.8 Calculator Stack Pick Dialog

See Calculator Stack Menu.

Pushes value at the indexth position on the stack onto the top of the stack:

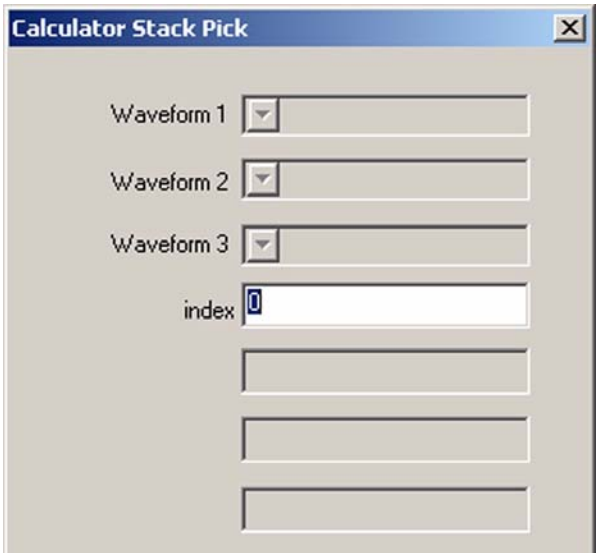


Figure-106: Calculator Stack Pick Dialog

6.11.9.9 Calculator Stack PushDoubleVariable Dialog

See Calculator Stack Menu

This dialog displays a list of double User Variable(s) defined in the currently loaded test program. When **OK** is clicked the value from the selected variable is pushed onto the stack

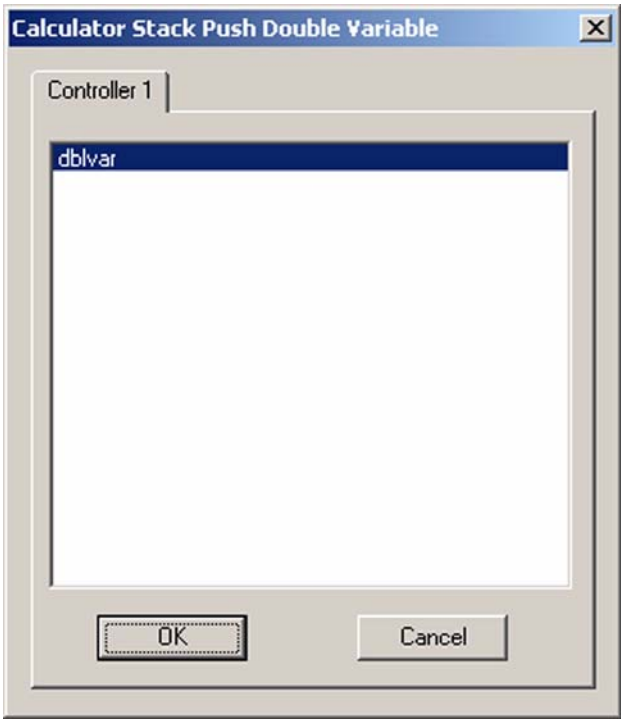


Figure-107: Calculator Stack Push Double Variable Dialog

6.11.9.10 Calculator Stack PushIntVariable Dialog

See Calculator Stack Menu

This dialog displays a list of integer User Variable(s) defined in the currently loaded test program. When **OK** is clicked the value from the selected variable is pushed onto the stack:

Figure-108: Calculator Stack Push Integer Variable Dialog

6.11.9.11 Calculator Stack PushResource Dialog

See Calculator Stack Menu

This dialog displays a list of waveform resources (`Waveform*`) in the currently loaded test program. When **OK** is clicked the selected waveform resource is displayed in MSWT, made the current waveform, and is pushed onto the stack:

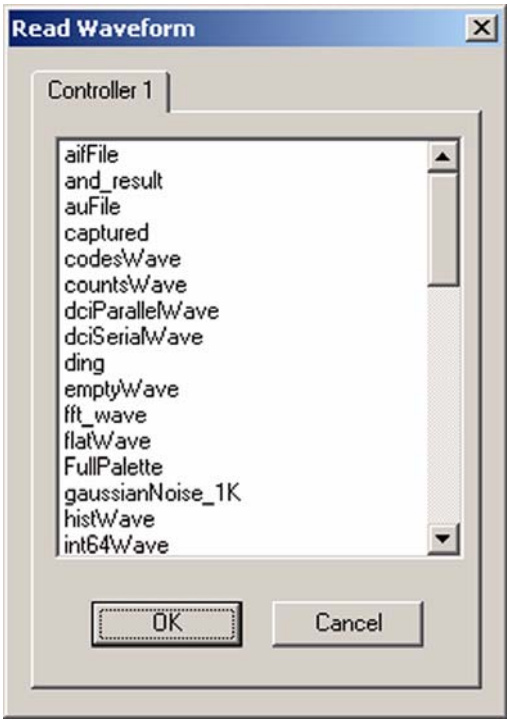


Figure-109: Calculator Stack Push Resource Variable Dialog

6.11.9.12 Calculator Stack PushWaveform Dialog

See Calculator Stack Menu

This dialog displays a list of waveforms currently displayed in MSWT. When **OK** is clicked the selected waveform is pushed onto the stack:

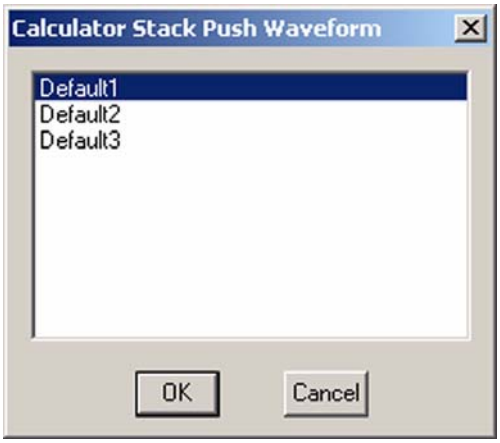


Figure-110: Calculator Stack Push Waveform Dialog

6.11.9.13 Calculator Stack Roll Dialog

See Calculator Stack Menu.

Removes the value at the indexth position in the stack and pushes it onto the top of the stack:

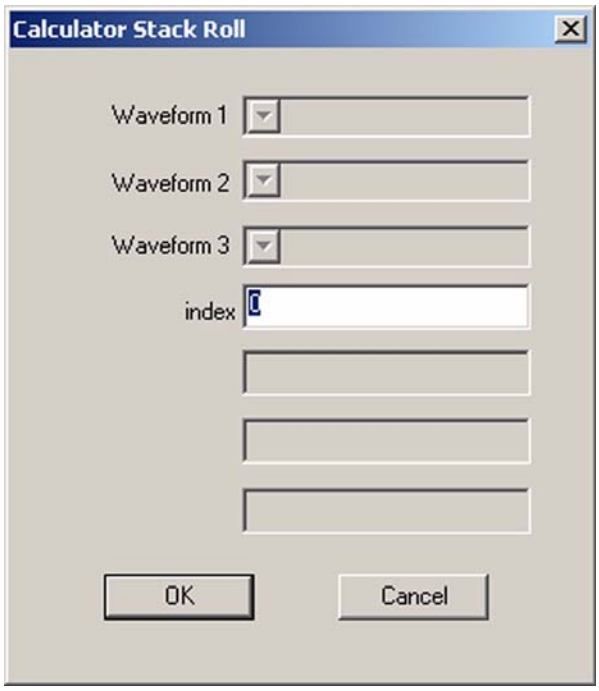


Figure-111: Calculator Stack Roll Dialog

6.11.9.14 Calculator Encode Menu

See Waveform Calculator, Calculator Controls.

These operations are used to compress (encode) a waveform, see Waveform Compression Functions. Each option below refers to the associated waveform function used to perform the same operation in test program code:

Figure-112: Calculator Encode Menu

A Law Decode	See `waveform_a_law_decode()`
A Law Encode	See `waveform_a_law_encode()`
BCD To Binary	See `waveform_bcd_to_binary()`
Binary To Gray Code	See `waveform_binary_to_gray_code()`
Binary To BCD	See `waveform_binary_to_bcd()`
Binary To Offset Binary	See `waveform_binary_to_offset_binary()`
Binary To Ones Complement	See `waveform_binary_to_ones_complement()`
Binary To Sign and Magnitude	See `waveform_binary_to_sign_and_magnitude()`
Binary To Twos Complement	See `waveform_binary_to_twos_complement()`

Gray Code To Binary	See waveform_gray_code_to_binary()
Lookup	See waveform_lookup()
Mu Law Decode	See waveform_mu_law_decode()
Mu Law Encode	See waveform_a_law_encode()
Offset Binary To Binary	See waveform_offset_binary_to_binary()
Ones Complement to Binary	See waveform_ones_complement_to_binary()
Sign and Magnitude To Binary	See waveform_sign_and_magnitude_to_binary()
Twos Complement to Binary	See waveform_twos_complement_to_binary()

6.11.9.15 Calculator Twiddle Menu

See Waveform Calculator, Calculator Controls.

These operations each generate and display a new waveform, using sample values taken from a specified waveform (default = current waveform) and modified based on the selected

function. Each option below refers to the associated waveform function used to perform the same operation in test program code:

Figure-113: Calculator Twiddle Menu

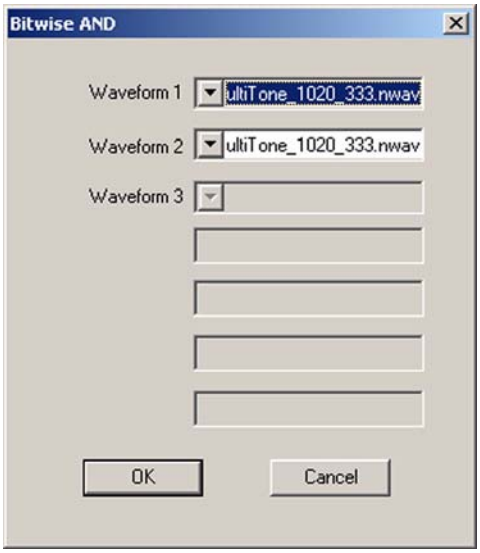
Bitwise AND **Bitwise AND Scalar**	See `waveform_bitwise_and()` and Calculator Twiddle Bitwise/Logical Dialogs
Bitwise OR **Bitwise OR Scalar**	See `waveform_bitwise_or()` and Calculator Twiddle Bitwise/Logical Dialogs
Bitwise Reorder	See `waveform_bitwise_reorder()` and Calculator Twiddle Rotate/Shift Dialogs
Bitwise Reverse	See `waveform_bitwise_reverse()` and Calculator Twiddle Rotate/Shift Dialogs
Bitwise RTL **Bitwise RTL Scalar**	See `waveform_bitwise_rotate_left()` and Calculator Twiddle Rotate/Shift Dialogs
Bitwise RTR **Bitwise RTR Scalar**	See `waveform_bitwise_rotate_right()` and Calculator Twiddle Rotate/Shift Dialogs
Bitwise SHL **Bitwise SHL Scalar**	See `waveform_bitwise_shift_left()` and Calculator Twiddle Rotate/Shift Dialogs

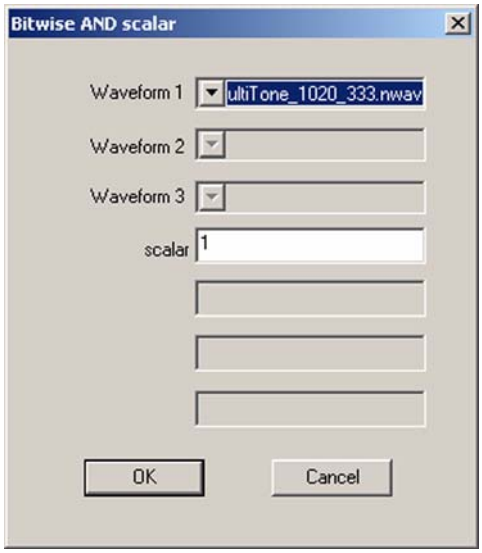
Bitwise SHR **Bitwise SHR Scalar**	See `waveform_bitwise_shift_right()` and Calculator Twiddle Rotate/Shift Dialogs
Bitwise XOR **Bitwise XOR Scalar**	See `waveform_bitwise_xor()` and Calculator Twiddle Bitwise/Logical Dialogs
Logical AND	See `waveform_logical_and()` and Calculator Twiddle Bitwise/Logical Dialogs
Logical OR	See `waveform_logical_or()` and Calculator Twiddle Bitwise/Logical Dialogs
Logical NOT	See `waveform_logical_not()` and Calculator Twiddle Bitwise/Logical Dialogs
Logical XOR	See `waveform_logical_xor()` and Calculator Twiddle Bitwise/Logical Dialogs
RTL **RTR**	See `waveform_rotate_left()`, `waveform_rotate_right()` and Calculator Twiddle Bitwise/Logical Dialogs

6.11.9.16 Calculator Twiddle Bitwise/Logical Dialogs

See Calculator Twiddle Menu.

Several Calculator Twiddle Menu options use dialogs very similar to those displayed below; i.e. other than the dialog title (Bitwise AND, etc.), the other dialog fields and controls are identical, and are described once below:





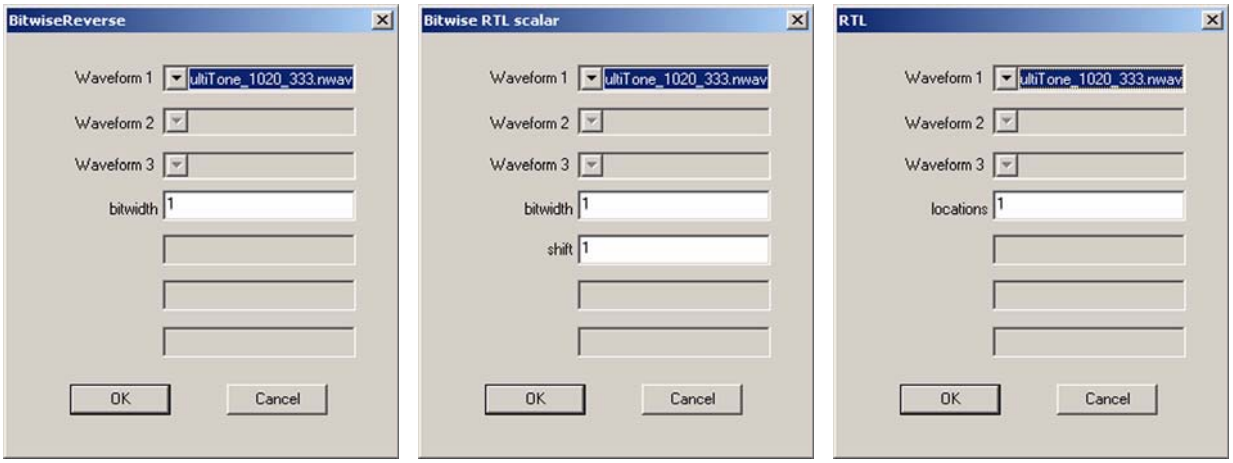
- Bitwise AND
- Bitwise OR
- Bitwise Reorder
- Bitwise XOR
- Logical AND
- Logical OR
- Logical NOT
- Logical XOR

- Bitwise AND Scalar
- Bitwise OR Scalar
- Bitwise XOR Scalar

6.11.9.17 Calculator Twiddle Rotate/Shift Dialogs

See Calculator Twiddle Menu.

Several Calculator Twiddle Menu options use dialogs similar to those displayed below i.e. other than the dialog title (Bitwise Reverse, etc.), the other dialog fields and controls are identical, and are described once below:



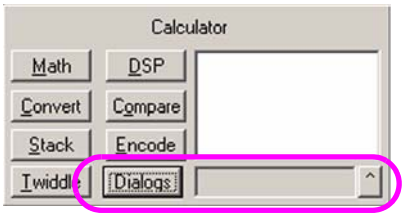
- Bitwise Reverse
- Bitwise RTL
- Bitwise RTR
- Bitwise SHL
- Bitwise SHR
- Bitwise RTL Scalar
- Bitwise RTR Scalar
- Bitwise SHL Scalar
- Bitwise SHR Scalar
- RTL
- RTR

Figure-114: Calculator Twiddle Rotate/Shift Dialogs

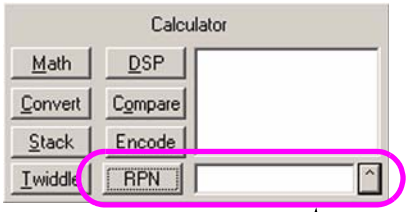
6.11.9.18 Calculator Dialogs/RPN Option

See Waveform Calculator, Calculator Controls.

MSWT's Waveform Calculator has two operating modes: Reverse Polish Notation (RPN) and standard, here called Dialogs mode. The **`RPN/Dialogs`** button is used to toggle between modes.



Dialogs Mode



RPN Mode
Data Entry Window

Note the button name changes, and the Data Entry Window is enabled in RPN mode. This window is used to manually enter data to be processed.

6.11.10 Response to UI User Variable Signals

See WaveformTool (MSWT).

UI defines a number of user variables which can trigger actions in tool processes. MSWT responds to (only) the following signals, as noted. Additional details of each UI user variable can be found in the *Maverick-I/-II Programmer's Manual*.

- `ui_ProgLoaded`: updates `ui_SiteMask` and `ui_Engineering` variables in MSWT.
- `ui_ProgUnloaded`: MSWT closes all waveform windows associated with the test program being unloaded.
- `ui_TestDone`: updates each waveform window by reading the current hardware state or waveform variable.
- `ui_Minimized` - MSWT display is minimized
- `ui_Restored` - MSWT display is restored
- `ui_ToolLoaded`: ignored
- `ui_ToolUnloaded`: ignored

- `ui_TestStarted`: ignored

6.11.11 MSWT Programming Functions

See WaveformTool (MSWT).

The following functions can be used to interact with WaveformTool (MSWT) from test program code. These are listed in approximate usage order:

- Types, Enums, etc.
- `mswt_present()`
- `mswt_start()`
- `mswt_minimize()`
- `mswt_restore()`
- `mswt_always_on_top()`
- `mswt_close_windows()`
- `mswt_display_file()`
- `mswt_display_waveform()`
- `mswt_synchronize()`
- `mswt_auto_synchronize()`
- `mswt_set_timeout()`
- `mswt_view_graph_controls()`
- `mswt_view_calculator_controls()`
- `mswt_view_compare_controls()`
- `mswt_view_cursor_controls()`
- `mswt_reset_graph_controls()`
- `mswt_angles_as_degrees()`
- `mswt_set_x_axis_mode(), mswt_set_y_axis_mode()`
- `mswt_set_y_axis_reference()`
- `mswt_set_plot_mode()`
- `mswt_set_trace_width()`
- `mswt_display_grid()`
- `mswt_set_axis_units()`

- mswt_set_y_range

6.11.11.1 Types, Enums, etc.

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The following enumerated types are used in support of various WaveformTool (MSWT) functions:

Usage

The MSWTPlotMode enumerated type is used to specify the plot mode for the current waveform displayed in WaveformTool (MSWT). See mswt_set_plot_mode():

```
enum MSWTPlotMode { mswt_line, mswt_line_mark, mswt_sample,
                    mswt_staircase, mswt_bar };
```

The MSWTXAxisMode and MSWTYAxisMode enumerated types are used to specify the X and Y display mode of the current waveform displayed in WaveformTool (MSWT). See mswt_set_x_axis_mode(), mswt_set_y_axis_mode():

```
enum MSWTXAxisMode { mswt_x_native, mswt_samples };
enum MSWTYAxisMode {mswt_y_native, mswt_magnitude, mswt_db_auto,
                    mswt_db_absolute };
```

The MSWTAxisUnits enumerated type is used to specify the X axis units used to display the current waveform in WaveformTool (MSWT). See mswt_set_axis_units():

```
enum MSWTAxisUnits { mswt_double, mswt_integer, mswt_hex };
```

The MSWTYRangeMode enumerated type is used to select the Y Ranging value using mswt_set_y_range:

```
MSWTYRangeMode
enum MSWTYRangeMode {  mswt_y_auto,
                       mswt_fixed_left,
                       mswt_fixed_right };
```

6.11.11.2 mswt_present()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_present()` function is used to confirm that WaveformTool (MSWT) has been started.

Usage

```
BOOL mswt_present();
```

where:

`mswt_present()` returns TRUE if WaveformTool (MSWT) is started, otherwise FALSE is returned.

Example

```
if( ! mswt_present() ) mswt_start();
```

6.11.11.3 mswt_start()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_start()` function is used to invoke WaveformTool (MSWT).

Usage

```
BOOL mswt_start();
```

where:

`mswt_start()` returns TRUE if WaveformTool (MSWT) starts without error, otherwise FALSE is returned.

Example

```
if( ! mswt_present() ) mswt_start();
```

6.11.11.4 mswt_minimize()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_minimize()` function is used to minimize the WaveformTool (MSWT) display. Note the following:

- The WaveformTool (MSWT) display can be restored using `mswt_restore()`.
- `mswt_minimize()` and `mswt_restore()` operate normally even when `mswt_always_on_top()` is TRUE.
- `mswt_minimize()` will return FALSE if executed when WaveformTool (MSWT) is not running.

Usage

```
BOOL mswt_minimize();
```

where:

`mswt_minimize()` returns FALSE if executed when WaveformTool (MSWT) is not running, otherwise TRUE is returned.

Example

```
if( ! mswt_minimize() )
   output(" mswt_minimize() executed when MSWT not started");
```

6.11.11.5 mswt_restore()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_restore()` function is used to cause the WaveformTool (MSWT) display to be un-minimized. Note the following:

- The WaveformTool (MSWT) display can be minimized using `mswt_minimize()`.

- `mswt_restore()` and `mswt_minimize()` operate normally even when `mswt_always_on_top()` is TRUE.
- `mswt_restore()` will return FALSE if executed when WaveformTool (MSWT) is not running.

Usage

```
BOOL mswt_restore();
```

where:

`mswt_restore()` returns FALSE if executed when WaveformTool (MSWT) is not running, otherwise TRUE is returned.

Example

```
if( ! mswt_restore() )
   output(" mswt_restore() executed when MSWT not started");
```

6.11.11.6 mswt_always_on_top()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_always_on_top()` function is used to control whether the WaveformTool (MSWT) display is always displayed on top of other displays. Note the following:

- `mswt_minimize()` and `mswt_restore()` operate normally even when `mswt_always_on_top()` is TRUE.
- `mswt_always_on_top()` will return FALSE if executed when WaveformTool (MSWT) is not running.

Usage

```
BOOL mswt_always_on_top( BOOL always );
```

where:

`always` specifies whether WaveformTool (MSWT) display is always displayed on top of other displays (TRUE) or not (FALSE).

`mswt_always_on_top()` returns FALSE if executed when WaveformTool (MSWT) is not running, otherwise TRUE is returned.

Example

```
if( ! mswt_always_on_top( TRUE ) )
   output(" mswt_always_on_top() executed when MSWT not started");
```

6.11.11.7 mswt_close_windows()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_close_windows()` function is used to terminate WaveformTool (MSWT) and close all associated windows.

`mswt_close_windows()` will return FALSE if executed when WaveformTool (MSWT) is not running.

Usage

```
BOOL mswt_close_windows();
```

where:

`mswt_close_windows()` returns FALSE if executed when WaveformTool (MSWT) is not running, otherwise TRUE is returned.

Example

```
if( ! mswt_close_windows() )
   output(" mswt_close_windows() executed when MSWT not started");
```

6.11.11.8 mswt_display_file()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_display_file()` function is used to read a specified waveform file and display the waveform in a new WaveformTool (MSWT) window. Several waveform formats are supported, see Waveform File Formats.

Usage

```
BOOL mswt_display_file(  CString filepath,
                         int channel DEFAULT_VALUE( -1 ) );
```

where:

`filepath` specifies the waveform file to be read. If a full path to the file is not specified the file must be relative to the test program executable location i.e. *disk:\path_to_program\Debug\program.exe*.

`channel` is optional, and if used specifies which channel of a multi-channel waveform (for example, stereo) is to be read. Default = -1 = all channels.

`mswt_display_file()` returns TRUE if successful, otherwise FALSE is returned.

Example

```
CString f = "d:\\my_path\\all_waveforms\\this_waveform.nwav";
if( ! mswt_display_file( f ) )
   output(" ERROR reading waveform file %s", f );
```

6.11.11.9 mswt_display_waveform()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_display_waveform()` function is used to display a specified waveform in WaveformTool (MSWT). The waveform is read from the test program.

Usage

```
BOOL mswt_display_waveform( Waveform* pWaveform );
```

where:

`pWaveform` identifies the waveform to be displayed.

`mswt_display_waveform()` returns TRUE if **`pWaveform`** points to a valid waveform, otherwise FALSE is returned.

Example

```
if( ! mswt_display_waveform( sine_wf ) )
   output(" ERROR: mswt_display_waveform() returned FALSE");
```

6.11.11.10 mswt_synchronize()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_synchronize()` function is used to synchronize (update) selected waveform windows in WaveformTool (MSWT). See Waveform Synchronization.

Usage

```
BOOL mswt_synchronize();
```

where:

`mswt_synchronize()` returns FALSE if executed when WaveformTool (MSWT) is not running, otherwise TRUE is returned.

Example

```
if( ! mswt_synchronize() )
  output(" ERROR: mswt_synchronize() executed when MSWT not started");
```

6.11.11.11 mswt_auto_synchronize()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_auto_synchronize()` function is used to enable or disable the WaveformTool (MSWT) auto-synchronize mode. See Waveform Synchronization.

Usage

```
BOOL mswt_auto_synchronize( BOOL auto_synchronize );
```

where:

`auto_synchronize` specifies whether auto-synchronize mode is enabled (TRUE) or disabled (FALSE).

`mswt_auto_synchronize()` returns FALSE if executed when WaveformTool (MSWT) is not running, otherwise TRUE is returned.

Example

```
if( ! mswt_auto_synchronize( TRUE ) )
  output(" ERROR: mswt_auto_synchronize() executed when MSWT not started");
```

6.11.11.12 mswt_set_timeout()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_set_timeout()` function is used to specify a time-out value for WaveformTool (MSWT).

Usage

```
DWORD mswt_set_timeout( DWORD timeout );
```

where:

`timeout` specifies the desired time-out value, in uS.

`mswt_set_timeout()` returns the currently programmed timeout value.

Example

```
DWORD t = mswt_set_timeout( 1000000 );
```

6.11.11.13 mswt_view_graph_controls()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_view_graph_controls()` function is used to enable or disable the display of WaveformTool (MSWT)'s graph controls. See View->Graph Controls in MSWT Toolbar View Menu.

Usage

```
BOOL mswt_view_graph_controls( BOOL visible );
```

where:

`visible` specifies whether WaveformTool (MSWT)'s graph controls are displayed (TRUE) or not (FALSE).

`mswt_view_graph_controls()` returns FALSE if executed when WaveformTool (MSWT) is not running, otherwise TRUE is returned.

Example

```
if( ! mswt_view_graph_controls( TRUE ))
  output(" ERROR: mswt_view_graph_controls() executed when MSWT not started");
```

6.11.11.14 mswt_view_calculator_controls()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_view_calculator_controls()` function is used to enable or disable the display of WaveformTool (MSWT)'s Calculator Controls.

Usage

```
BOOL mswt_view_calculator_controls( BOOL visible );
```

where:

visible specifies whether MSWT's Calculator Controls are displayed (TRUE) or not (FALSE).

`mswt_view_calculator_controls()` returns FALSE if executed when WaveformTool (MSWT) is not running, otherwise TRUE is returned.

Example

```
if( ! mswt_view_calculator_controls( TRUE ) )
 output(" ERROR: mswt_view_calculator_controls() executed when MSWT not started");
```

6.11.11.15 mswt_view_compare_controls()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_view_compare_controls()` function is used to enable or disable the display of WaveformTool (MSWT)'s compare controls. See View->Compare Controls in MSWT Toolbar View Menu.

Usage

```
BOOL mswt_view_compare_controls( BOOL visible );
```

where:

visible specifies whether WaveformTool (MSWT)'s compare controls are displayed (TRUE) or not (FALSE).

`mswt_view_compare_controls()` returns FALSE if executed when WaveformTool (MSWT) is not running, otherwise TRUE is returned.

Example

```
if( ! mswt_view_compare_controls( TRUE ) )
  output(" ERROR: mswt_view_compare_controls() executed when MSWT not started");
```

6.11.11.16 mswt_view_cursor_controls()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_view_cursor_controls()` function is used to enable or disable the display of WaveformTool (MSWT)'s cursor controls. See View->Cursor Controls in MSWT Toolbar View Menu.

Usage

```
BOOL mswt_view_cursor_controls( BOOL visible );
```

where:

`visible` specifies whether WaveformTool (MSWT)'s cursor controls are displayed (TRUE) or not (FALSE).

`mswt_view_cursor_controls()` returns FALSE if executed when WaveformTool (MSWT) is not running, otherwise TRUE is returned.

Example

```
if( ! mswt_view_cursor_controls( TRUE ) )
  output(" ERROR: mswt_view_cursor_controls() executed when MSWT not started");
```

6.11.11.17 mswt_reset_graph_controls()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_reset_graph_controls()` function is used to reset MSWT's View->Graph Controls to their default values. This is the same as clicking the `Reset` button.

Usage

```
BOOL mswt_reset_graph_controls();
```

where:

`mswt_reset_graph_controls()` returns FALSE if executed when WaveformTool (MSWT) is not running, otherwise TRUE is returned.

Example

```
if( ! mswt_reset_graph_controls( TRUE ) )
  output(" ERROR: mswt_reset_graph_controls() executed when MSWT not started");
```

6.11.11.18 mswt_angles_as_degrees()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_angles_as_degrees()` function is used to specify whether angle values in the current waveform displayed in WaveformTool (MSWT) are displayed as degrees or radians. This is the same control displayed using WaveformTool (MSWT)'s `View->Angles as Degrees` option.

Usage

```
BOOL mswt_angles_as_degrees( BOOL degrees );
```

where:

`degrees` specifies whether WaveformTool (MSWT)'s angle values are displayed as degrees (TRUE) or radians (FALSE). Only affects the current waveform being displayed in WaveformTool (MSWT).

`mswt_angles_as_degrees()` returns FALSE if executed when WaveformTool (MSWT) is not running, otherwise TRUE is returned.

Example

```
if( ! mswt_angles_as_degrees( TRUE ) )
  output(" ERROR: mswt_angles_as_degrees() executed when MSWT not started");
```

6.11.11.19 mswt_set_x_axis_mode(), mswt_set_y_axis_mode()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_set_x_axis_mode()` and `mswt_set_y_axis_mode()` functions are used to specify the X and Y display mode of the current waveform displayed in WaveformTool (MSWT). These are the same controls displayed in View->Graph Controls dialog.

Usage

```
BOOL mswt_set_x_axis_mode( MSWTXAxisMode x_axis_mode );
BOOL mswt_set_y_axis_mode( MSWTYAxisMode y_axis_mode );
```

where:

`x_axis_mode` and **`y_axis_mode`** specify the desired X/Y axis display mode. Legal values are of the `MSWTXAxisMode` enumerated type. Only affects the current waveform being displayed in WaveformTool (MSWT).

`mswt_set_x_axis_mode()` and `mswt_set_y_axis_mode()` return FALSE if executed when WaveformTool (MSWT) is not running, otherwise TRUE is returned.

Example

```
if( ! mswt_set_x_axis_mode( mswt_x_native ) )
  output(" ERROR: mswt_set_x_axis_mode() executed when MSWT not started");
if( ! mswt_set_y_axis_mode( mswt_db_absolute ) )
  output(" ERROR: mswt_set_y_axis_mode() executed when MSWT not started");
```

6.11.11.20 mswt_set_y_axis_reference()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_set_y_axis_reference()` function is used to specify the Y axis reference for the current waveform displayed in WaveformTool (MSWT). This is the same value displayed in the `Ref` value.

Usage

```
BOOL mswt_set_y_axis_reference( double y_reference );
```

where:

y_reference specifies the desired reference value. See description for Ref. Only affects the current waveform being displayed in WaveformTool (MSWT).

mswt_set_y_axis_reference() returns FALSE if executed when WaveformTool (MSWT) is not running, otherwise TRUE is returned.

Example

```
if( ! mswt_set_y_axis_reference( 1 ) )
  output(" ERROR: mswt_set_y_axis_reference() executed when MSWT not started");
```

6.11.11.21 mswt_set_plot_mode()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The mswt_set_plot_mode() function is used to specify the plot mode for the current waveform displayed in WaveformTool (MSWT). This is the same value displayed in View->Graph Controls for the Plot value.

Usage

```
BOOL mswt_set_plot_mode( MSWTPlotMode plot_mode );
```

where:

plot_mode specifies the desired plot mode. Legal values are of the MSWTPlotMode enumerated type. See Plot for description. Only affects the current waveform being displayed in WaveformTool (MSWT).

mswt_set_plot_mode() returns FALSE if executed when WaveformTool (MSWT) is not running, otherwise TRUE is returned.

Example

```
if( ! mswt_set_plot_mode( mswt_line ) )
  output(" ERROR: mswt_set_plot_mode() executed when MSWT not started");
```

6.11.11.22 mswt_set_trace_width()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_set_trace_width()` function is used to specify the trace width for the current waveform displayed in WaveformTool (MSWT). This is the same value displayed in the View->Graph Controls as the `Trace Width` value.

Usage

```
BOOL mswt_set_trace_width( int trace_width );
```

where:

`trace_width` specifies the desired trace width. Legal values are 1, 2 or 4. Only affects the current waveform being displayed in WaveformTool (MSWT).

`mswt_set_trace_width()` returns FALSE if executed when WaveformTool (MSWT) is not running, otherwise TRUE is returned.

Example

```
if( mswt_set_trace_width( 2) == FALSE )
   output(" ERROR: mswt_set_trace_width() returned FALSE");
```

6.11.11.23 mswt_display_grid()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_display_grid()` function is used to specify whether the grid is displayed in the current waveform.

Usage

```
BOOL mswt_display_grid( BOOL visible );
```

where:

visible specifies whether the grid is displayed (TRUE) or not (FALSE). Only affects the current waveform being displayed in WaveformTool (MSWT).

mswt_display_grid() returns FALSE if executed when WaveformTool (MSWT) is not running, otherwise TRUE is returned.

Example

```
if( ! mswt_display_grid( FALSE ) )
  output(" ERROR: mswt_display_grid() executed when MSWT not started");
```

6.11.11.24 mswt_set_axis_units()

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The mswt_set_axis_units() function is used to specify the X axis units used to display the current waveform in WaveformTool (MSWT).

Usage

```
BOOL mswt_set_axis_units( MSWTAxisUnits axis_units );
```

where:

axis_units specifies the desired X axis units option. Legal values are of the MSWTAxisUnits enumerated type. Only affects the current waveform being displayed in WaveformTool (MSWT).

mswt_set_axis_units() returns FALSE if executed when WaveformTool (MSWT) is not running, otherwise TRUE is returned.

Example

```
if( ! mswt_set_axis_units( mswt_integer ) )
  output(" ERROR: mswt_set_axis_units() executed when MSWT not started");
```

6.11.11.25 mswt_set_y_range

See WaveformTool (MSWT), MSWT Programming Functions.

Description

The `mswt_set_y_range()` function is used to specify the Y Ranging and Y minimum values used to display the current waveform. These are the same as displayed in the View->Graph Controls.

Usage

```
BOOL mswt_set_y_range( MSWTYRangeMode range_mode,
                       double y_minimum,
                       double y_maximum );
```

where:

`range_mode` specifies the desired ranging mode. Legal values are of the `MSWTYRangeMode` enumerated type. See Y Ranging for description. Only affects the current waveform being displayed in WaveformTool (MSWT).

`y_minimum` and **`y_maximum`** are used when **`range_mode`** = `mswt_fixed_left` or `mswt_fixed_right`. When to **`range_mode`** = `mswt_fixed_left` the left axis range is controlled by **`Y Minimum`** and **`Y Maximum`**. When **`range_mode`** = `mswt_fixed_right` the right axis range is controlled by **`Y Minimum`** and **`Y Maximum`**.

`mswt_set_y_range()` returns FALSE if executed when WaveformTool (MSWT) is not running, otherwise TRUE is returned.

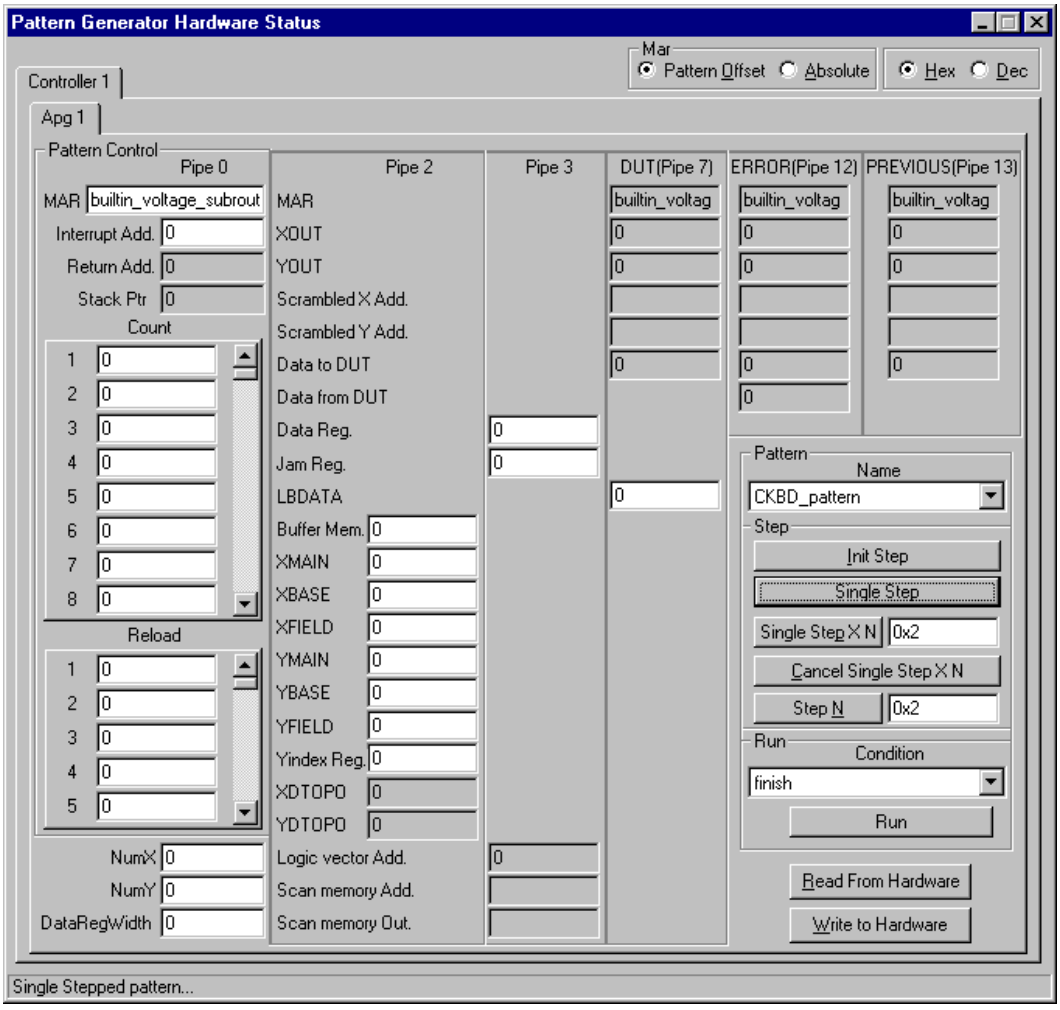
Example

```
if( ! mswt_set_y_range( mswt_y_auto, 0, 10 ) )
  output(" ERROR: mswt_set_y_range() executed when MSWT not started");
```

6.12 PatternDebugTool

To start `PatternDebugTool`:

- Click on the PatternDebugTool icon from the *Ui* toolbar 
- Type keyboard shortcut **`Ctrl+P`**
- Choose **`Tools: Pattern`**



6.13 Resource Manager

Under Development.

6.14 ScanTool

Note: not yet documented for Magnum 1/2/2x.

6.15 ShmooTool / SearchTool

This section contains the following:

6.15.1 Overview

See ShmooTool / SearchTool.

ShmooTool and *SearchTool* are the same tool, used to specify or modify the various parameters needed to define a shmoo or binary/linear search. Only the term ShmooTool will be used in the remainder of this chapter.

ShmooTool is used to create and maintain named shmoo and search definitions. Once defined, shmoo and search execution can be invoked several ways:

- Interactively, using the Breakpoint Monitor: See Executing Shmoos and Searches Interactively.
- Programmatically, in user-written C code. See Executing Shmoos and Searches Programmatically.

Controls in ShmooTool are used to:

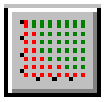
- Select whether a shmoo or binary/linear search definition is being viewed.
- Specify a name for each shmoo/search definition. These names are used:

  - In ShmooTool, to select a specific shmoo/search definition to view or modify.
  - In the Breakpoint Monitor, to control interactive shmoo/search execution.
- Specify shmoo/search parameters, start/stop values, resolution, tracking parameters, axis labels, a title, etc.
- **Save** or **Load** shmoo/search definitions to/from disk (see Shmoo Definition File).

6.15.2 Starting ShmooTool

See ShmooTool / SearchTool.

ShmooTool is normally started from UI, after a test program is loaded. The UI **Advanced**, and **Engineering Mode** options must both have been selected. The following methods can be used to start ShmooTool:

- Click on the ShmooTool icon from the *Ui* tool bar 
- Type keyboard shortcut **Ctrl+H**
- Choose **Tools: Shmoo/Search**...

6.15.3 Search Output

See ShmooTool / SearchTool.

Search output results are displayed in the site output window(s) of UI. Below is an example linear search output. The parameter being searched was a user variable named: `x_axis_float`:

```
x_axis_float : 1.734245
```

Search results can also be captured in a disk file, using `ui_ShmooOutputFile` (see built-in UI User Variables and Executing Shmoos and Searches Programmatically for an example).

6.15.4 Shmoo Output

See ShmooTool / SearchTool.

A graphical shmoo output is displayed in a separate window in UI. This window opens automatically when a shmoo is generated. Below is an example graphical shmoo, showing some of the key features:

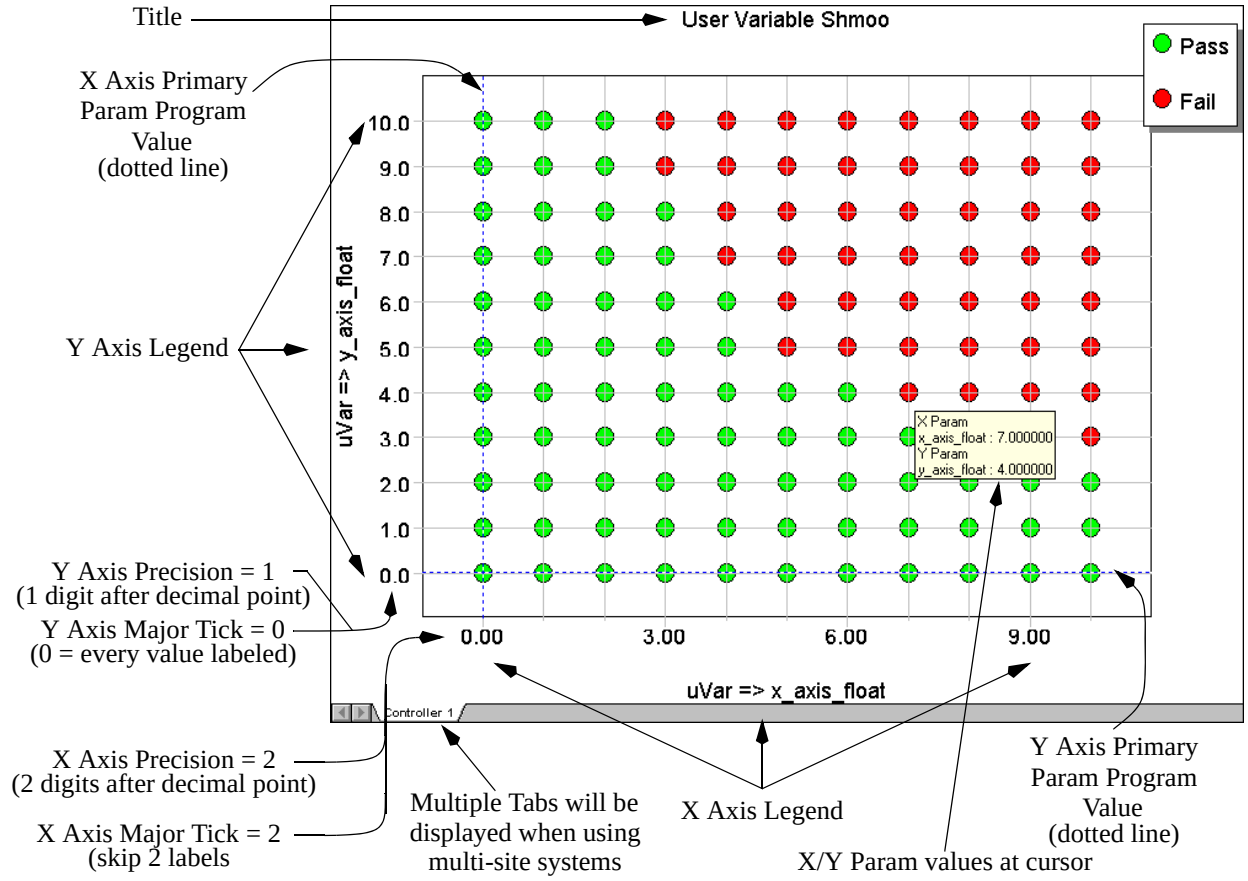


Figure-115: Example Shmoo Output

The shmoo display is limited to PASS and FAIL values. A PASS value is any non-zero value returned by the executed test or test block.

When using a multi-site system each site controller will have a separate tab in this display, allowing the user to view the shmoo from each site.

Using the right-mouse button in the shmoo display will cause the following menu to appear:

Figure-116: Shmoo Output Window Controls

These controls can be used to:

- **`Hide`** the shmoo window
- **`Clear`** the shmoo window
- **`Update the Title and Legend`**: after changing Title and/or Legend properties using ShmooTool, select this option to update the shmoo display without actually generating a new shmoo.
- **`Print`** the shmoo window
- **`Save`** the shmoo window to a disk file, which saves an ASCII shmoo (see below)
- **`Float`** the shmoo window in the main display

A shmoo can also be captured in a disk file, using the `ui_ShmooOutputFile` (see UI User Variables and Executing Shmoos and Searches Programmatically for an example). The shmoo captured in a disk file is an ASCII equivalent of the graphical shmoo. Below is the ASCII shmoo equivalent of the grahical shmoo shown above:

```
version:1
   title:User Variable Shmoo
   type:0
   order:0
   xlegend:uVar => x_axis_float
   xmajor:2
   xdigits:2
   xpoints:11
   xcount:1
```

```
   ylegend:uVar => y_axis_float
   ymajor:0
   ydigits:1
   ypoints:11
   ycount:1
   xparam:x_axis_float
   min:0
   max:10
   yparam:y_axis_float
   min:0
   max:10
ShmooInput:Shmoo1
***........
***........
****.......
****.......
*****......
*****......
*******....
*********..
**********
**********
**********
```

6.15.5 ShmooTool Help

See ShmooTool / SearchTool.

A form of context sensitive on-line help is available which is accessible in the various graphic tools:

- Using the mouse, left-click an item of interest in the graphic tool
- Press the **F1** key

Below is an example. Using the mouse, the user had clicked the 

selector then pressed the **F1** key. Note the result below:

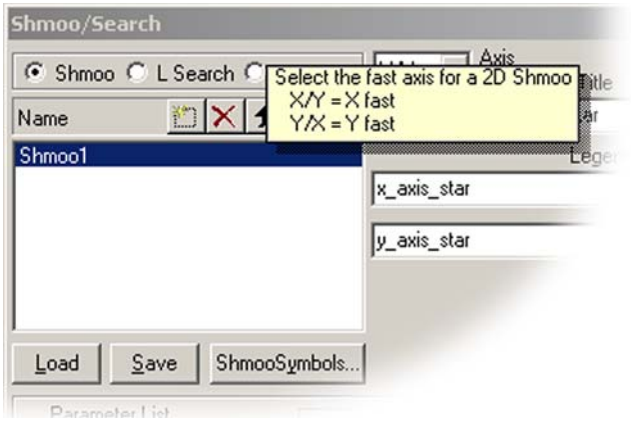


Figure-117: Shmoo/Search Help

6.15.6 Defining Shmoos & Searches

See ShmooTool / SearchTool.

The initial ShmooTool dialog is shown below, before a shmoo or search definition has begun:

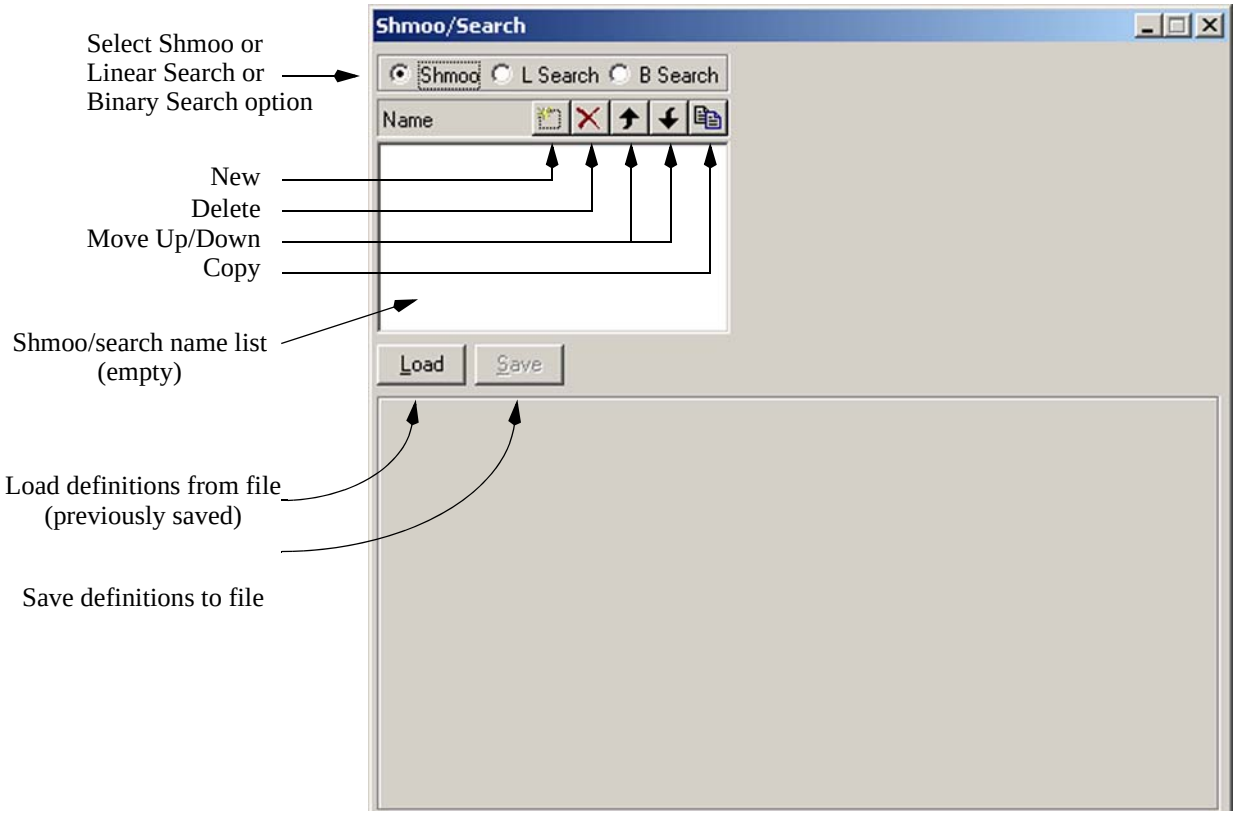


Figure-118: Initial Shmoo/Search Dialog

The initial display allows the following:

- Selection of Shmoo, Linear Search, or Binary Search option. This specifies the type of definition being created, viewed, or modified.
- Add a **`New`** name to the list of shmoo/search names. This begins the process of creating a named shmoo/search definition. Default names depend on whether `Shmoo`, `LSearch` or `BSearch` is selected i.e. *Shmoo1*, *LSearch1*, *BSearch1*. The user may also edit these name as desired. Once the name is correct type **`<Enter>`** to proceed.
- **`Load`** an existing shmoo/search definition file from disk. These would have been previously saved using **`Save`**. See Shmoo Definition File.

Once a name is added to the list, or an existing name is selected from the list, the ShmooTool display will change, to enable and display additional controls suitable for

defining the shmoo or a search. Note that if an existing name was selected from the list some/all parameters for that shmoo/search would likely be filled in.

Note: the selection of Shmoo, LSearch, or BSearch is an attribute of the definition being viewed. If, at any time, this selection is changed, the type of definition being viewed is also changed. This allows, for example, changing a Shmoo to a binary search without having to create a completely new definition. Or, an existing definition can be copied (using the Copy button) and the type of the copy can be changed easily using the Shmoo, LSearch, and BSearch radio buttons.

6.15.6.1 ShmooTool: Search Controls

See ShmooTool / SearchTool, Defining Shmoos & Searches.

The example below shows the display seen when LSearch or BSearch is selected, before any of the parameters have been defined:

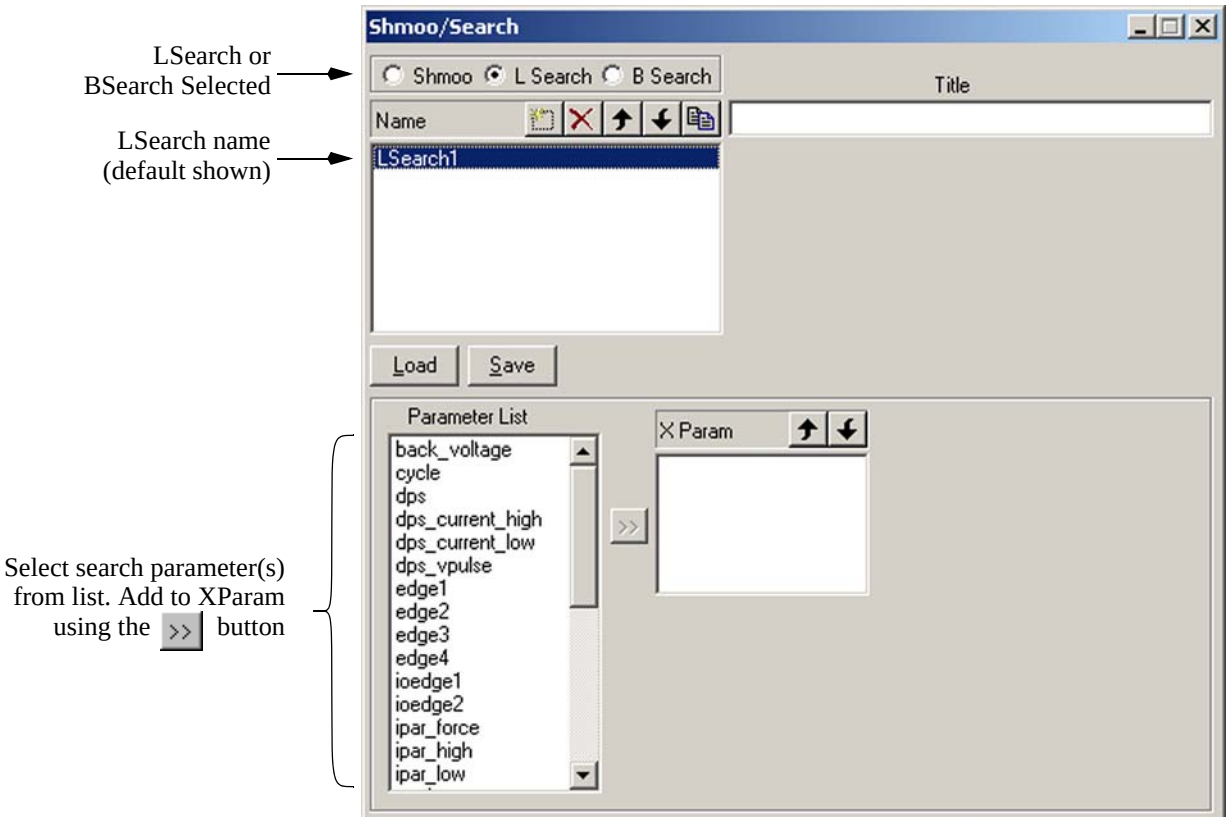


Figure-119: Search Controls

Note that this is a subset of controls used to define shmoos (see ShmooTool: Shmoo Controls). For this reason, only the shmoo controls will be documented further.

6.15.6.2 ShmooTool: Shmoo Controls

See ShmooTool / SearchTool, Defining Shmoos & Searches.

The example below shows the shmoo display before any of the parameters have been defined:

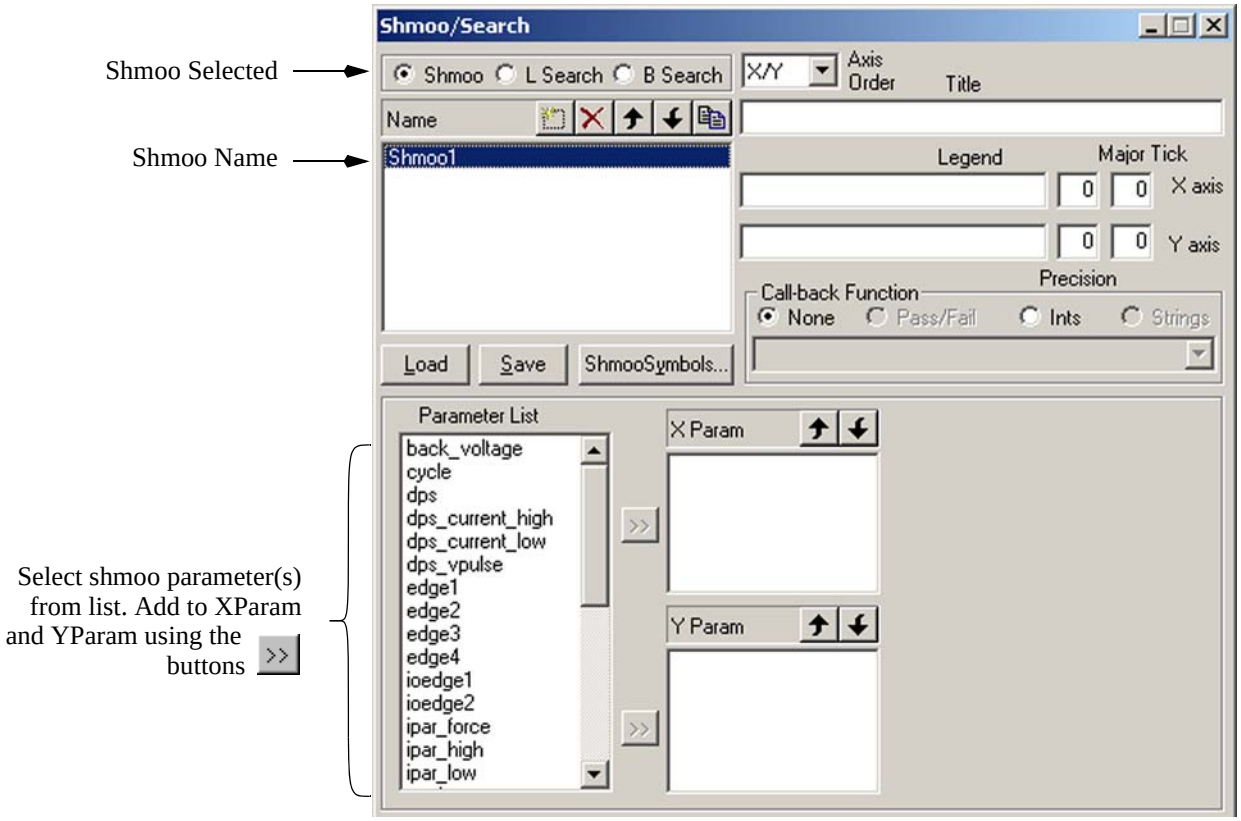


Figure-120: Shmoo Controls

The `Title` window and `Legend` window will display either default labels, or the user can manually enter preferred labels. Once a `Title` or `Legend` has been manually entered, the default mechanism is disabled. The default labels are based on the *primary* XParam selection and the primary YParam parameter selection for each axis. See below.

In the following example, the primary XParam is `dps`, the *secondary* XParam is `vih`, and the primary YParam is `vol`.The `dps` parameter is selected thus the rest of the paramter options apply to it. Note the default Title and Legend labels:

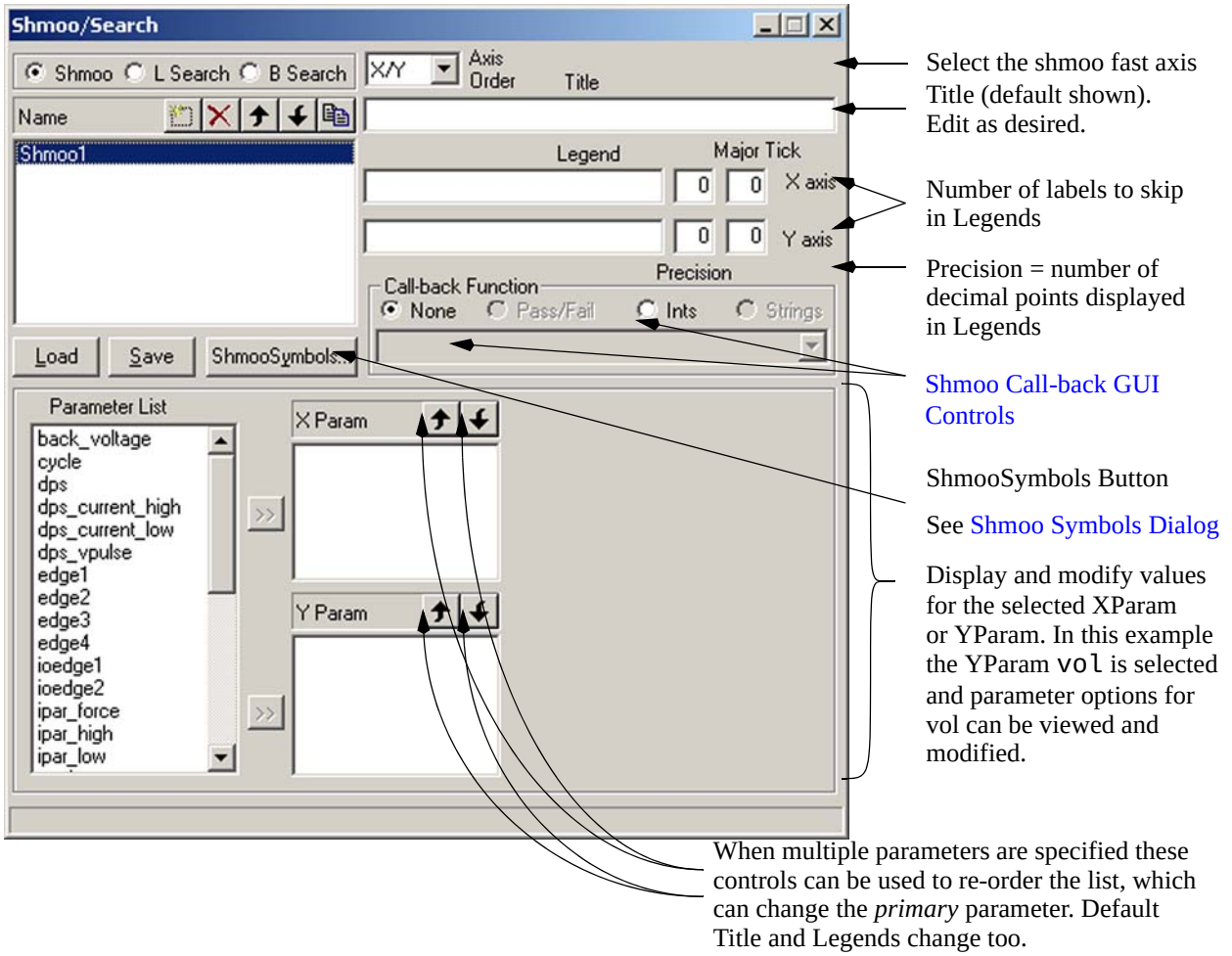


Figure-121: Shmoo Controls

As shown, it is possible to specify more than one parameter for the XParam and/or YParam fields. This creates a tracking parameter scenario, where the first parameter specified on each axis is the *primary* parameter and all others on that axis are *secondary* parameters. Only the *primary* parameter directly considers the `Calculate` option, which determines how the resolution of the shmoo/search is determined. The resolution of all secondary parameters is based on the `#Points`, calculated or specified i.e. (max-min/#Points).

It is also possible to create a single axis shmoo. This is done by leaving either the X Param or Y Param list empty.

The complete Parameter List is shown in Shmoo/Search Optional Parameters.

Different parameter option fields are displayed in the right portion of the parameter area, and depend on which parameter is selected. Below are three examples:

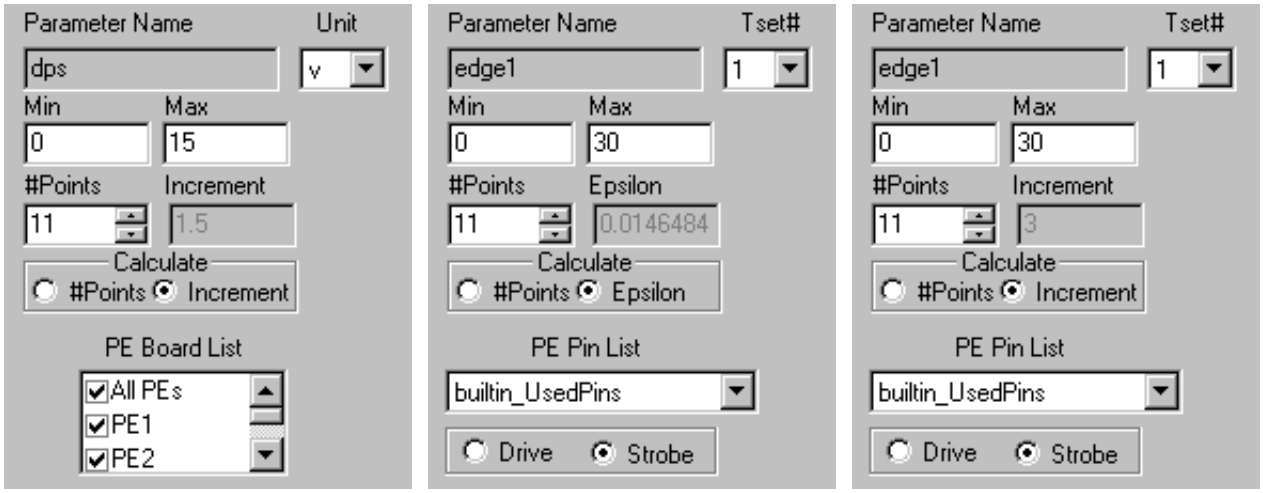


Figure-122: Example Shmoo Parameter Options

The following option fields are displayed regardless of the type of parameter selected:

- **`Parameter Name`**: based on the parameter currently selected. Not directly editable. All other fields are applied to the selected parameter. The first parameter of an axis sets the default Legend and part of the default Title.
- **`Min`** / **`Max`**: the shmoo/search limit, inclusive.

Note: no range or limit checks are made to the min/max values entered. Use caution (and the Product Spec.)

- **`Calculate`**: radio buttons used to specify whether the **`#Points`** or **`Increment (or Epsilon)`** value is calculated. The other value must be manually entered by the user. Default is **`Increment`** (or **`Epsilon`**). Only the *primary* parameter on each axis is directly affected i.e. the step-size (resolution) of all other parameters on the same axis is calculated using **`#Points`** (whether user programmed or calculated).
- **`#Points`**: the resolution of the shmoo/search on that axis, as a count. When **`#Points`** is user specified the **`Increment`** (or **`Epsilon`**) is calculated.
- **`Increment`**: the resolution of the shmoo or linear search on that axis, as a value. When **`Increment`** is user specified the **`#Points`** is calculated.

- **Epsilon**: the resolution of binary search on that axis, as a value. When **Epsilon** is user specified the **#Points** is calculated. **Epsilon** is limited in software to a maximum of 31.

The other parameter option fields are:

- **Unit**: selects how Legend values are scaled and which *units* are displayed. Applies to voltage and current parameters only. Timing values are always specified in Nanoseconds (nS).
- **TSet#**: select which time-set is to be modified when shmoo/search cycle period or edge time values.
- **PE Pin List**: specify which pins are to be modified. Applies only to parameters which are programmable per-pin. Default selection is the `builtin_UsedPins` pin list.
- **PE Board List**: specify which PE Boards are to be modified. Applies to parameters which are programmable per-board. Default selection is all PE boards. Note that this option does not appear when using Magnum 1/2/2x.
- **Drive/Strobe**: radio button used to select whether drive or strobe formats are to be modified. Applies to edge time values only.

The following table shows which options apply to each parameter type:

Table 6.15.6.2-1 Shmoo/Search Optional Parameters

Shmoo/Search Parameter	Unit	TSet#	PE Pin List	DPS Pin List	Drive/ Strobe
back_voltage	Yes	No	Yes	No	No
cycle	No	Yes	No	No	No
dps	Yes	No	No	Yes	No
dps_current_high	Yes	No	No	Yes	No
dps_current_low	Yes	No	No	Yes	No
dps_vpulse	Yes	No	No	Yes	No
edge1 edge2 edge3 edge4	No	Yes	Yes	No	Yes

Table 6.15.6.2-1 Shmoo/Search Optional Parameters *(Continued)*

Shmoo/Search Parameter	Unit	TSet#	PE Pin List	DPS Pin List	Drive/ Strobe
ioedge1 ioedge2	No	Yes	Yes	No	Yes
ipar_force	Yes	No	Yes	No	No
ipar_high	Yes	No	Yes	No	No
ipar_low	Yes	No	Yes	No	No
partime	Yes	No	No	No	No
positive_clamp	Yes	No	Yes	No	No
negative_clamp	Yes	No	Yes	No	No
vih	Yes	No	Yes	No	No
vil	Yes	No	Yes	No	No
vihh	Yes	No	Yes	No	No
voh	Yes	No	Yes	No	No
vol	Yes	No	Yes	No	No
vpar_force	Yes	No	Yes	No	No
vpar_high	Yes	No	Yes	No	No
vpar_low	Yes	No	Yes	No	No
vtt	Yes	No	Yes	Yes	No
vz	Yes	No	Yes	Yes	No
INT_VARIABLE[1]	No	No	No	No	No
DWORD_VARIABLE[1]	No	No	No	No	No
FLOAT_VARIABLE[1]	No	No	No	No	No
DOUBLE_VARIABLE[1]	No	No	No	No	No
INT64_VARIABLE[1]	No	No	No	No	No

Notes:
1) User variables are optional, and will appear in the Parameter List using the name of the user variable.
See Shmoos and Searches using User Variables

6.15.7 Shmoo Functions

See ShmooTool / SearchTool.

The following functions may be used to access various shmoo/search options:

- Types, Enums, etc.
- `shmoo_title_get()`
- `shmoo_type_get()`
- `shmoo_direction_get()`
- `shmoo_axis_params_get()`
- `shmoo_param_get()`
- `shmoo_param_pointval_get()`
- `shmoo_duts_subtitle_set()`, `shmoo_duts_subtitle_get()`
- `search_results_get()`

Note: the functions above were first available in software release h1.1.23.

6.15.7.1 Types, Enums, etc.

See ShmooTool / SearchTool, Shmoo Functions.

Description

The following enumerated types are used in support of various Shmoo Functions:

Usage

The `ShmooAxis` enumerated type is used select a specific shmoo axis, typically when retrieving information about shmoo parameters for that axis. See `shmoo_axis_params_get()`, `shmoo_param_get()`, `shmoo_param_pointval_get()`:

```
enum ShmooAxis{ t_xaxis, t_yaxis, t_axis_na };
```

The `ShmooType` enumerated type is used as return values when retrieving the type of a names shmoo. See `shmoo_type_get()`:

```
enum ShmooType{t_shmoo,
               t_linearsearch,
               t_binarysearch,
               t_shmoo_search_na };
```

The ShmooAxisOrder enumerated type is returned by shmoo_direction_get():

```
enum ShmooAxisOrder { t_shmoo_XY, t_shmoo_YX };
```

The SearchResultStruct structure is used to return search results for one DUT. See search_results_get():

```
typedef struct SearchResultStruct {
   BOOL search_valid;
   double last_pass;
   double last_fail;
} search_result;
```

The SearchResultArray structure is used to return search results for multiple DUTs. See search_results_get():

```
typedef CArray< search_result, search_result& > SearchResultArray;
```

6.15.7.2 shmoo_title_get()

See ShmooTool / SearchTool, Shmoo Functions.

Note: first available in software release h1.1.23.

Description

The shmoo_title_get() function may be used to retrieve the title defined for a named shmoo. This is the title displayed at the top-center of the shmoo output window and set in the **Title** edit box (see Shmoo Controls).

Usage

```
BOOL shmoo_title_get( LPCTSTR shmoo_name, CString* title);
```

where:

shmoo_name identifies the target shmoo.

title is a pointer to an existing CString variable used to return the title.

shmoo_title_get() returns TRUE if the specified **shmoo_name** is valid, otherwise FALSE is returned.

Example

```
CString title;
BOOL ok = shmoo_title_get( "myShmoo", &title );
if( ! ok ) output(" ERROR: bad shmoo name specified");
else output(" Shmoo Title => %s", title );
```

6.15.7.3 shmoo_type_get()

See ShmooTool / SearchTool, Shmoo Functions.

Note: first available in software release h1.1.23.

Description

The shmoo_type_get() function may be used to retrieve the shmoo/search type of a named shmoo. This is set using the **Shmoo**, **LSearch** and **BSearch** radio buttons in ShmooTool (see Shmoo Controls).

Usage

```
BOOL shmoo_type_get( LPCTSTR shmoo_name, ShmooType* shmoo_type );
```

where:

shmoo_name identifies the target shmoo.

shmoo_type is a pointer to an existing ShmooType variable used to return the shmoo type.

shmoo_type_get() returns TRUE if the specified **shmoo_name** is valid, otherwise FALSE is returned.

Example

```
ShmooType shmoo_type;
BOOL ok = shmoo_type_get( "myShmoo", &shmoo_type );
if( ! ok ) output(" ERROR: bad shmoo name specified");
```

6.15.7.4 shmoo_direction_get()

See ShmooTool / SearchTool, Shmoo Functions.

Note: first available in software release h1.1.23.

Description

The `shmoo_direction_get()` function may be used to retrieve the Axis Order (i.e. fast-axis, shmoo direction, etc.) attribute of a named shmoo. This is set using the **`Axis Order`** selection menu in ShmooTool (see Shmoo Controls).

Usage

```
BOOL shmoo_direction_get(LPCTSTR shmoo_name,
                         ShmooAxisOrder* direction );
```

where:

`shmoo_name` identifies the target shmoo.

`direction` is a pointer to an existing `ShmooAxisOrder` variable used to return the Axis Order value.

`shmoo_direction_get()` returns TRUE if the specified **`shmoo_name`** is valid, otherwise FALSE is returned.

Example

```
ShmooAxisOrder direction;
BOOL ok = shmoo_direction_get( "myShmoo", &direction );
if( ! ok ) output(" ERROR: bad shmoo name specified");
```

6.15.7.5 shmoo_axis_params_get()

See ShmooTool / SearchTool, Shmoo Functions.

Note: first available in software release h1.1.23.

Description

The `shmoo_axis_params_get()` function may be used to retrieve the parameter information for a specified axis of a named shmoo. These are the parameters set using various Shmoo Controls in ShmooTool.

Usage

```
BOOL shmoo_axis_params_get( LPCTSTR shmoo_name,
                            ShmooAxis axis,
                            CString* legend,
                            int* major_tick,
                            int* precision,
                            int* points,
                            CStringArray* param_name_array );
```

where:

`shmoo_name` identifies the target shmoo.

`axis` identifies the target axis. Legal values are of the `ShmooAxis` enumerated type (but `t_axis_na` should not be used).

`legend` is a pointer to an existing `CString` variable used to return the Legend associated with the specified **`axis`**.

`major_tick` is a pointer to an existing `int` variable used to return the Major Tick value associated with the specified **`axis`**.

`precision` is a pointer to an existing `int` variable used to return the Precision value associated with the specified **`axis`**.

`points` is a pointer to an existing `int` variable used to return the #Points value associated with the specified **`axis`**.

`param_name_array` is a pointer to an existing `CStringArray` variable used to return one or more Parameter Names associated with the specified **`axis`**. The array will be resized by

the system software and any prior contents are lost. The first element in the returned array is the primary parameter name. Each subsequent element is a secondary parameter. Elements are returned in the order shown in ShmooTool's XParam or YParam list (as determined by the **axis** value).

`shmoo_axis_params_get()` returns TRUE if the specified **shmoo_name** is valid, otherwise FALSE is returned.

Example

```
CString legend;
int major_tick, precision, points;
CStringArray* params;
BOOL ok = shmoo_axis_params_get( "myShmoo", t_xaxis,
            &legend, &major_tick, &precision, &points, &params );
if( ! ok ) output(" ERROR: bad shmoo name specified");
```

6.15.7.6 shmoo_param_get()

See ShmooTool / SearchTool, Shmoo Functions.

Note: first available in software release h1.1.23.

Description

The `shmoo_param_get()` function may be used to retrieve the minimum and maximum values specified for a specified Parameter Name associated with a given axis in ShmooTool (see Example Shmoo Parameter Options).

Usage

```
BOOL shmoo_param_get(  LPCTSTR shmoo_name,
                       LPCTSTR param_name,
                       ShmooAxis axis,
                       double* min,
                       double* max );
```

where:

shmoo_name identifies the target shmoo.

param_name identifies the target parameter, which must be valid for the specified **axis**. These are case sensitive and identify one of the values shown in Shmoo/Search Optional Parameters.

axis identifies the target axis. Legal values are of the ShmooAxis enumerated type (but t_axis_na should not be used).

min and **max** are pointers to existing int variables used to return the Min and Max values specified for the specified **param_name**.

shmoo_param_get() returns TRUE if the specified **shmoo_name** is valid and **param_name** is valid for the specified **axis**, otherwise FALSE is returned.

Example

```
int min, max;
BOOL ok = shmoo_param_get( "myShmoo", "dps", t_xaxis, &min, &max );
if( ! ok ) output(" ERROR: bad shmoo name specified");
```

6.15.7.7 shmoo_param_pointval_get()

See ShmooTool / SearchTool, Shmoo Functions.

Note: first available in software release h1.1.23.

Description

The shmoo_param_pointval_get() function may be used to retrieve the calculated value set for a specified point of a specified parameter on a specified axis of a named shmoo. This will be the value set for the specified parameter during the execution of the test which determines the result for the specified point.

Usage

```
BOOL shmoo_param_pointval_get(LPCTSTR shmoo_name,
                              LPCTSTR param_name,
                              ShmooAxis axis,
                              int point,
                              double* val );
```

where:

shmoo_name identifies the target shmoo.

param_name identifies the target parameter, which must be valid for the specified **axis**. These are case sensitive and identify one of the values shown in Shmoo/Search Optional Parameters.

axis identifies the target axis. Legal values are of the ShmooAxis enumerated type (but t_axis_na should not be used).

point identifies the point-of-interest. Legal values will be 0 to #Points -1, as specified or calculated.

val is a pointer to an existing int variable used to return the calculated value.

shmoo_param_pointval_get() returns TRUE if the specified **shmoo_name** is valid and **param_name** is valid for the specified **axis**, otherwise FALSE is returned.

Example

```
CString legend;
int major_tick, precision, points;
double value;
CStringArray params;

// Get #Points
BOOL ok = shmoo_axis_params_get( "myShmoo", t_xaxist,
            &legend, &major_tick, &precision, &points, &params );
if( ! ok ) output(" ERROR: bad shmoo name specified");

else {
   int c = params.GetSize(); // Num param on this axis
   for( int p = 0; p < c; ++p ){ // For each param on this axis...
      output(" Param => %s", params.GetAt(p) );
      for( int i = 0; i < points; ++i ){ // For each point on axis
         BOOL ok = shmoo_param_pointval_get("myShmoo",
                                            params.GetAt(p),
                                            t_xaxis,
                                            i,
                                            &value);
      if( ! ok ) output(" ERROR: bad shmoo name specified");
      output(" Value at point-%d => 0.3f", i, value );
      }
   }
}
```

6.15.7.8 shmoo_duts_subtitle_set(), shmoo_duts_subtitle_get()

See ShmooTool / SearchTool, Shmoo Functions.

Note: first available in software release h1.1.23.

Description

The `shmoo_duts_subtitle_set()` function is used to define a secondary title (subtitle) to be displayed in all subsequent shmoos. The subtitle will display immediately below the Title (see Example Shmoo Output).

The `shmoo_duts_subtitle_get()` function is used to get the currently defined subtitle(s).

Note the following:

- Each subtitle is first defined as a `CString`.
- Then, to support specification of a different subtitle for each DUT (see Multi-DUT Shmoos) each `CString` (each subtitle) is added to a `CStringArray`, which is then passed as an argument to `shmoo_duts_subtitle_set()`. When testing a single DUT, only the first element in the `CStringArray` will be used.
- `shmoo_duts_subtitle_set()` is the only method for defining a subtitle; i.e. it is not possible using ShmooTool / SearchTool.

Usage

```
BOOL shmoo_duts_subtitle_set( CStringArray& subtitles );
int shmoo_duts_subtitle_get( CStringArray* subtitles );
```

where:

`subtitles` is used in two contexts:

- In the set function, used to specify the subtitle text to be displayed.
- In the get function, used to return the currently defined subtitle(s).

`shmoo_duts_subtitle_set()` returns TRUE if one or more shmoos are currently defined, otherwise FALSE is returned.

`shmoo_duts_subtitle_get()` returns the number of elements in **`subtitles`**.

Example

The following example was designed for use in a Multi-DUT Test Program which tests 4 DUTs in parallel. It adds a unique subtitle (“DUT-n”) to each Shmoo output window, each window corresponding to one DUT. Each time the Sequence/Binning table is executed the subtitle changes for each Shmoo window:

```
BEFORE_TESTING_BLOCK( BTB ){
   CStringArray stitles;
   DutNumArray duts;
   static int dut = 1;
   for( int i = 0; i < active_duts_get( &duts ); ++i ){
      CString s = vFormat("DUT-%d", dut++);
      stitles.Add( s );
   }
   BOOL ok = shmoo_duts_subtitle_set( stitles );
   if( ! ok ) output(" ERROR: NO shmoos currently defined");
   // Get and output the current subtitles
   CStringArray curstitles;
   int c = shmoo_duts_subtitle_get( &curstitles );
   for( i = 0; i < c; ++i )
      output(" %s", curstitles.GetAt( i ) );
}
```

6.15.7.9 search_results_get()

See ShmooTool / SearchTool, Shmoo Functions.

Note: first available in software release h1.1.23.

Description

The `search_results_get()` function can be used to retrieve the result(s) of the most recently executed search. Note the following:

- This function is useful only after executing a search defined using ShmooTool / SearchTool and which is triggered by a breakpoint set using BreakpointTool (see Shmoo/Search Execution). This is targeted for use in conjunction with `ui_ShmooDone`.

- `search_results_get()` supports multi-DUT searches (see Multi-DUT Shmoos). Values are returned in an array, named `result_array`. When testing a single DUT the returned array will contain a single element.
- Each return value (each array element) consists of several components, stored in a `SearchResultStruct` structure:

```
typedef struct SearchResultStruct {
   BOOL search_valid;
   double last_pass;
   double last_fail;
} search_result;
```

  - `search_valid` = TRUE if the search algorithm converged on a PASS/FAIL boundary, otherwise FALSE is returned. If `search_valid` returns FALSE the other structure parameters are invalid.
  - `last_pass` = the value of the search parameter the last time the search algorithm executed with a passing test result. In most applications, this will be the desired search result. However, using linear searches, it is sometimes desirable to use the last failing value.
  - `last_fail` = the value of the search parameter the last time the search algorithm executed with a failing test result.

Usage

```
int search_results_get( SearchResultArray* result_array );
```

where:

`result_array` is a pointer to an existing `SearchResultArray` variable used to return the search results. See Description.

`search_results_get()` returns the number of elements in the returned **`result_array`**.

Example

```
CSTRING_VARIABLE( ui_ShmooDone, "", "" ){
   SearchResultArray result;
   int c = search_results_get( &results ); // search_results_get();
}
```

6.15.8 Multi-DUT Shmoos

See ShmooTool / SearchTool.

Description

The Multi-DUT shmoo capability is targeted for use in test programs which test multiple DUTs in parallel.

Multi-DUT shmoos are always enabled in Multi-DUT Test Programs and a shmoo output window will be created for each DUT in the test program. However, during shmoo execution, no shmoo is generated for DUT(s) which are not in the Active DUTs Set (ADS); i.e. the output window for disabled DUT(s) will be empty.

Shmoo Call-back Function

Each shmoo may or may not have a user-written call-back function registered. If a call-back is registered it will be executed for each point of the shmoo, with the call-back code determining the result displayed for each point for each active DUT.

Multiple call-back functions are supported since the code needed to evaluate functional test results differs from the code needed for PMU test results, DPS current test results, etc. One side effect of the call-back method is that user code is solely responsible for determining the value displayed for each point in each DUT's shmoo and thus can use any criteria to do so.

Regarding the shmoo call-back functions, note the following:

- A shmoo call-back is optional, but if used, completely determines the shmoo results displayed for each active DUT.
- if one or more shmoo call-back functions are registered (next bullet) ShmooTool displays the Shmoo Call-back GUI Controls. These are used to associate a specific call-back function with each shmoo.
- Three variations of shmoo call-back functions are supported:
  - `shmoo_dutsPF_callback`: *Pass/Fail* call-back which returns a `boolean` value for each DUT. In the shmoo, these values are displayed using **Red**/**Green** (or */. in ASCII shmoo output). Registered using `shmoo_duts_PF_callback_set()`.
  - `shmoo_duts_int_callback`: *Integer* call-back which returns an `integer` value for each DUT. User-code defines a color to be displayed for each integer value using the dialog displayed by the ShmooSymbols Button. Registered using `shmoo_duts_int_callback_set()`.

  - `shmoo_duts_string_callback`: *string* call-back which returns an `LPCTSTR` value for each DUT. The string value is displayed as-is. Registered using `shmoo_duts_string_callback_set()`.
- Registering a call-back function allows a user-defined name to displayed for selection in ShmooTool's Shmoo Call-back GUI Controls combo box.
- The call-back registration functions (`shmoo_duts_PF_callback_set()`, `shmoo_duts_int_callback_set()` and `shmoo_duts_string_callback_set()`) must be executed from Site code. They are ignored if executed from the Host or a User Tool code.
- During shmoo execution, the call-back function is executed only on the Site. The return values are sent to UI for display in the shmoo(s).

Shmoo Call-back GUI Controls

The controls shown below are displayed if one or more Shmoo Call-back Function(s) are registered. These controls are used to associate a Shmoo Call-back Function with the shmoo being defined:

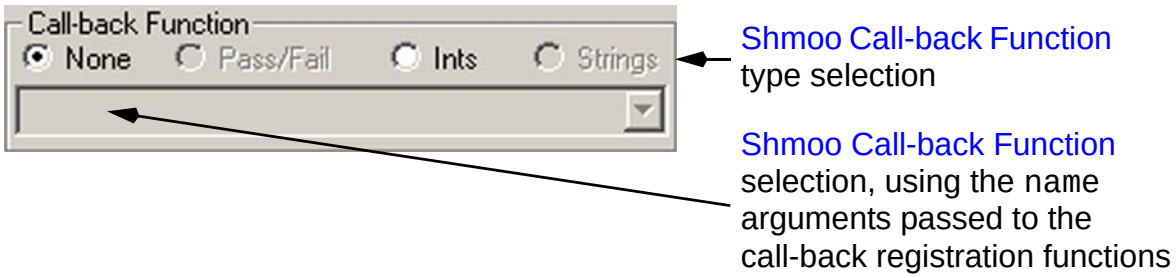


Figure-123: Shmoo Call-back Controls

The radio buttons are used to select the type of call-back function to be selected:

- **`None`** - default. Select this to specify that no Shmoo Call-back Function is to be executed by the shmoo being defined.
- **`Pass/Fail`** - allows display and selection of Shmoo Call-back Functions which were registered using `shmoo_duts_PF_callback_set()`. These functions return a boolean value for each DUT.
- **`Ints`** - allows display and selection of Shmoo Call-back Functions which were registered using `shmoo_duts_int_callback_set()`. These functions return an integer value for each DUT.
- **`Strings`** - allows display and selection of Shmoo Call-back Functions which were registered using `shmoo_duts_string_callback_set()`. These functions return a string (`LPCTSTR`) value for each DUT.

A given radio button will not be enabled for use if no call-back functions of the related type have been registered.

As indicated, the comb-box will display the names of the Shmoo Call-back Functions of the selected type. The actual names displayed are not the function names, but rather the `name` argument passed to the call-back registration functions.

Shmoo Symbols Dialog

Note: first available in software release h1.1.23.

When one or more shmoos has a Shmoo Call-back Function the ShmooSymbols Button will be displayed. Clicking this button will display the Shmoo Symbols dialog, which is used to determine which color (or symbol in ASCII shmoos) is displayed for each value returned by a `shmoo_duts_int_callback` function.

A `shmoo_duts_int_callback` function returns an integer value for each DUT being tested. Using this call-back, ShmooTool does not display the integer value, but rather the color mapped to the integer using the Shmoo Symbols dialog. In the example below, a

unique color is mapped to the 8 integer values (in the Val column) expected to be returned by the call-back function:

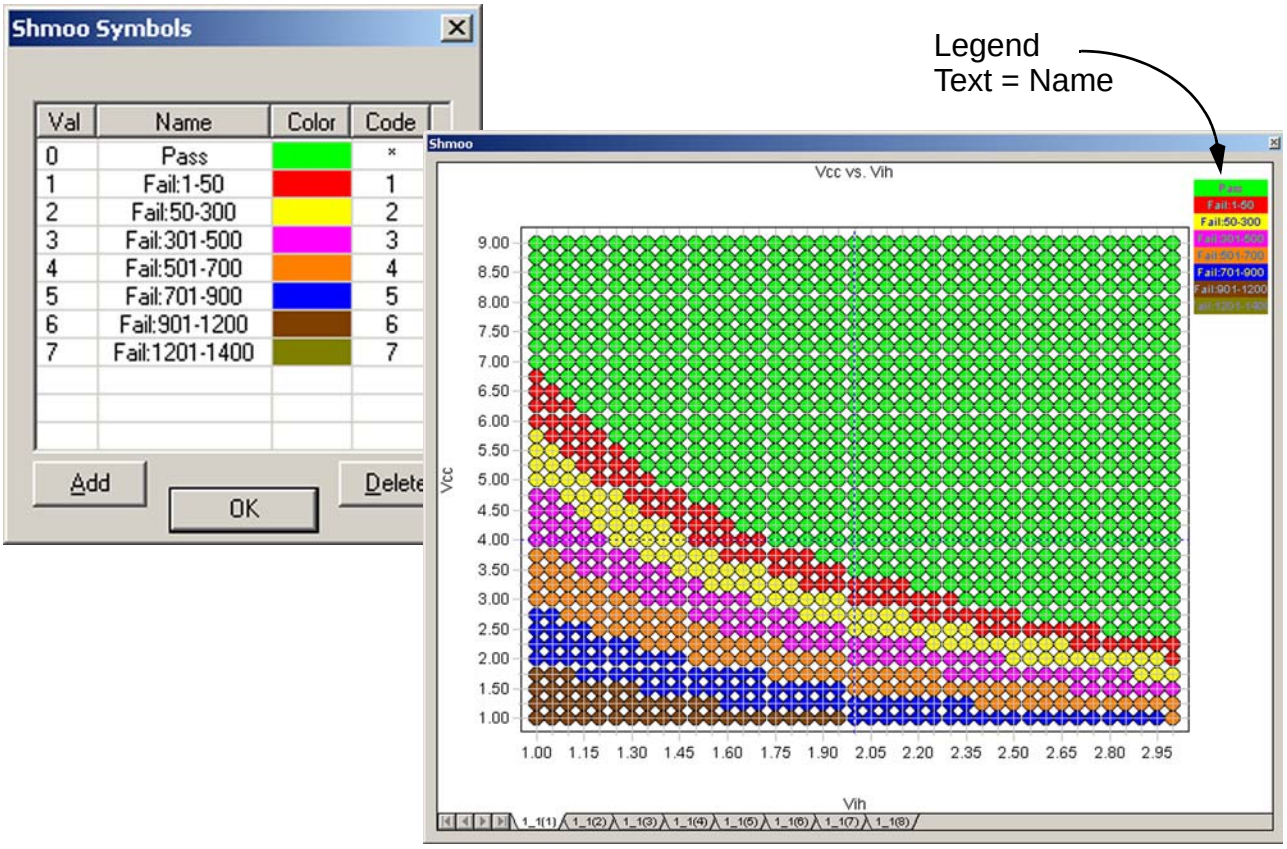


Figure-124: Example Shmoo Integer Output

Note the following:

- The dialog initially has two values: 0 and 1, representing Fail/Red and Pass/Green.
- When ShmooTool receives an integer result which is not mapped to a color the resulting shmoo will contain an empty (white) point.
- Rows (Vals) are added using the **Add** button and deleted using the **Delete** button.
- The *Name* column specifies the label displayed in the legend for each *Val*. Name values are edited by double-clicking in the appropriate *Name* cell.
- The *Color* column specifies the color displayed in the main ShmooTool display and in the legend for each *Val*. *Color* values are edited by double-clicking in the appropriate *Color* cell and selecting the desired color from the dialog presented.

- The *Code* column specifies the character(s) displayed in ASCII shmoos for each *Val*. *Code* values are edited by double-clicking in the appropriate *Code* cell.

Usage

The following functions are used to register a user-written call-back function:

```
void shmoo_duts_PF_callback_set(
                        LPCTSTR name,
                        shmoo_dutsPF_callback func );
```

The following functions were first available in software release h1.1.23.

```
void shmoo_duts_int_callback_set(
                        LPCTSTR name,
                        shmoo_duts_int_callback func );
void shmoo_duts_string_callback_set(
                        LPCTSTR name,
                        shmoo_duts_string_callback func );
```

where:

name is the value displayed in ShmooTool's **DUT Pass/Fail Callback** combo box, for use in selecting the function identified by **func**.

func identifies the user-written call-back function and is used in several contexts:

- The first form is used to return a `boolean` (Pass/Fail) result for each DUT. The user-written `shmoo_dutsPF_callback` function must conform to the following prototype:

  ```
  DWORD (*shmoo_dutsPF_callback)( DWORD dut_mask );
  ```

- The second form is used to return an integer value for each DUT. The user-written `shmoo_duts_int_callback` function must conform to the following prototype:

  ```
  void (*shmoo_duts_int_callback)( DWORD dut_mask,
                                   IntArray* results );
  ```

- The third form is used to return an string (`LPCTSTR`) value for each DUT. The user-written `shmoo_duts_string_callback` function must conform to the following prototype:

  ```
  void (*shmoo_duts_string_callback)( DWORD dut_mask,
                                      CStringArray* results );
  ```

where:

dut_mask is a bit-wise mask identifying which DUT(s) are being tested. The LSB represents DUT-1, LSB+1 = DUT-2, etc. **dut_mask** is derived from the current Active DUTs Set (ADS).

results is a pointer to an existing IntArray or CStringArray used to return an integer or string value for each DUT which is active in **dut_mask**. The user's call-back function code must add a value to the array for each DUT which is active in **dut_mask** before returning.

The shmoo_dutsPF_callback() must return a bit mask in which each bit represents the PASS/FAIL results for one DUT. A logic-1 represents PASS and a logic-0 represents FAIL. The LSB equates to DUT-1, LSB+1 = DUT2, etc. Only the bit positions which match those set in the input **dut_mask** argument are used.

Example

The following example is pseudo-code used to display different colors based on the number of failures retrieved from the ECR for each DUT. Note that additional configuration is required to specify the color to be displayed for each value returned by my_ints_callback. This is done using the Shmoo Symbols Dialog. The example shown above was generated using code similar to the following example:

```
int dut_failed_bits( int dut ){ // Support function
   DutNumArray ads;
   active_duts_get( &ads );              // Save for ADS restore
   active_duts_enable( dut, FALSE );     // Enable one dut
   int bit_cnt = ecr_main_ram_scan(...); // Args for your app
   if( bit_cnt == 0 ) return 0;     // Pass = Green
   if( bit_cnt > 51 ) return 1;     // Fail: 1-50 = Red
   if( bit_cnt > 301) return 2;     // Fail: 51-300 = Yellow
   if( bit_cnt > 501 ) return 3;    // Fail: 301-500 = Magenta
   if( bit_cnt > 701 ) return 4;    // Fail: 501-700 = Orange
   if( bit_cnt > 901 ) return 5;    // Fail: 701-900 = Blue
   if( bit_cnt > 1201 ) return 6;   // Fail: 901-1200 = Brown
   if( bit_cnt > 1401 ) return 7;   // Fail: 1201-1400 = Grey
   active_duts_enable( ads, FALSE );     // Restore ADS
   return bit_cnt;
}

// Users Ints call-back
void my_ints_callback( DWORD dut_mask, IntArray* results ){
   for( int dut = 0; dut_mask != 0; ++dut, dut_mask >>= 1 )
```

```
      if ( dut_mask & 1 )
         results->Add( dut_failed_bits( dut ) );
}
SITE_BEGIN_BLOCK( SB1 ){
   // ... other code as desired
   shmoo_duts_int_callback_set( // Register call-back
        "ECRFailBitCounts",
        my_ints_callback );
   // ... other code as desired
}
```

6.15.9 Shmoo/Search Execution

See ShmooTool / SearchTool.

Shmoos and searches can be invoked using two methods, documented separately:

- Executing Shmoos and Searches Interactively
- Executing Shmoos and Searches Programmatically

6.15.9.1 Executing Shmoos and Searches Interactively

See ShmooTool / SearchTool, Shmoo/Search Execution

The paradigm for defining and interactively executing shmoos and searches utilizes two tools:

- ShmooTool: used to create, maintain, and save named shmoo and search definitions
- Breakpoint Monitor: used to create, maintain, and save breakpoints used to execute shmoos and searches

Note: programmatic shmoo/search execution uses these same tools for the same purposes, but additional methods are available to select and enable breakpoints in user-written C code. See Executing Shmoos and Searches Programmatically.

This paradigm has the following benefits:

1. Shmoo/search definitions are managed using an interactive tool. Methodology is identical regardless of how the resulting shmoo/search is executed.
2. Shmoo/search definitions are program independent, and thus are mostly reusable. The one exception to this is when the shmoo/search uses User Variables, which are inherently program specific.
3. The Breakpoint Monitor provides versatility in executing shmoos/searches, independent of how they are created. A given shmoo, or search, can be applied to more than one test, or test block, or a mix of tests and test blocks. Using Breakpoint Monitor, it is possible to apply a shmoo/search to an arbitrary block of user-written code.

The following example is used to show the location of key Breakpoint Monitor controls:

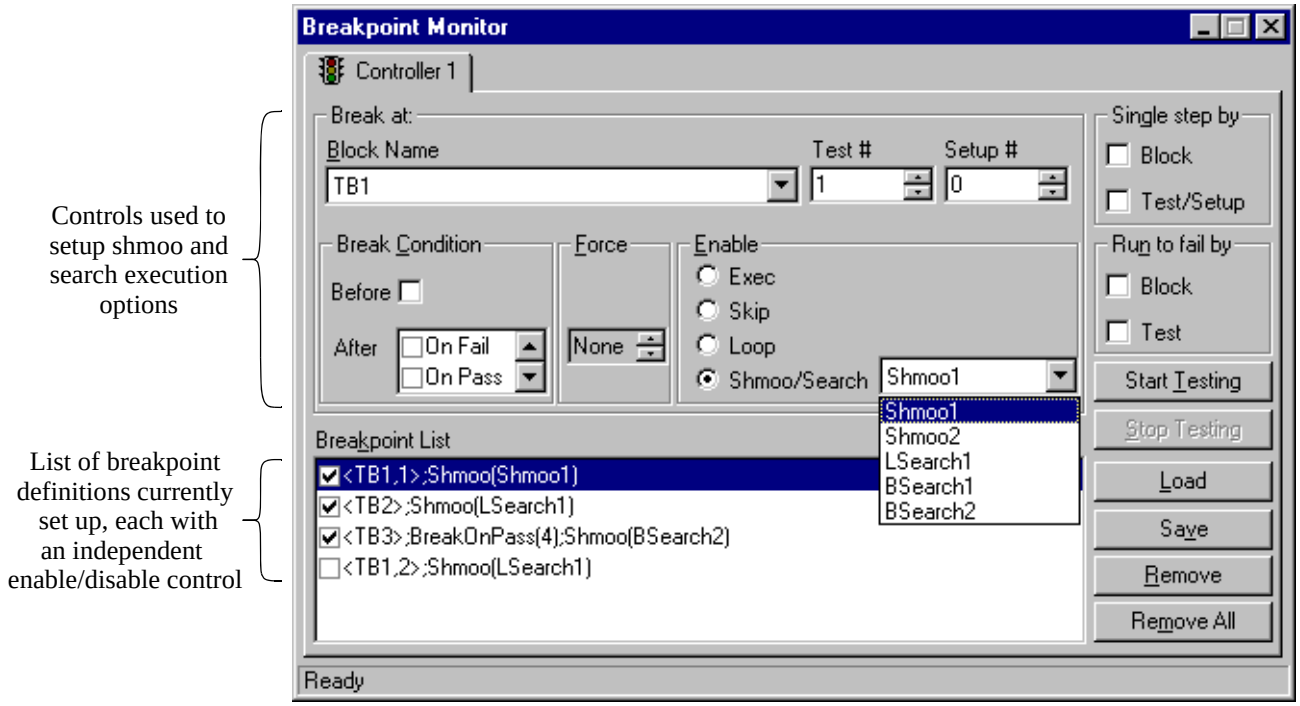


Figure-125: Shmoo Breakpoint Monitor Controls

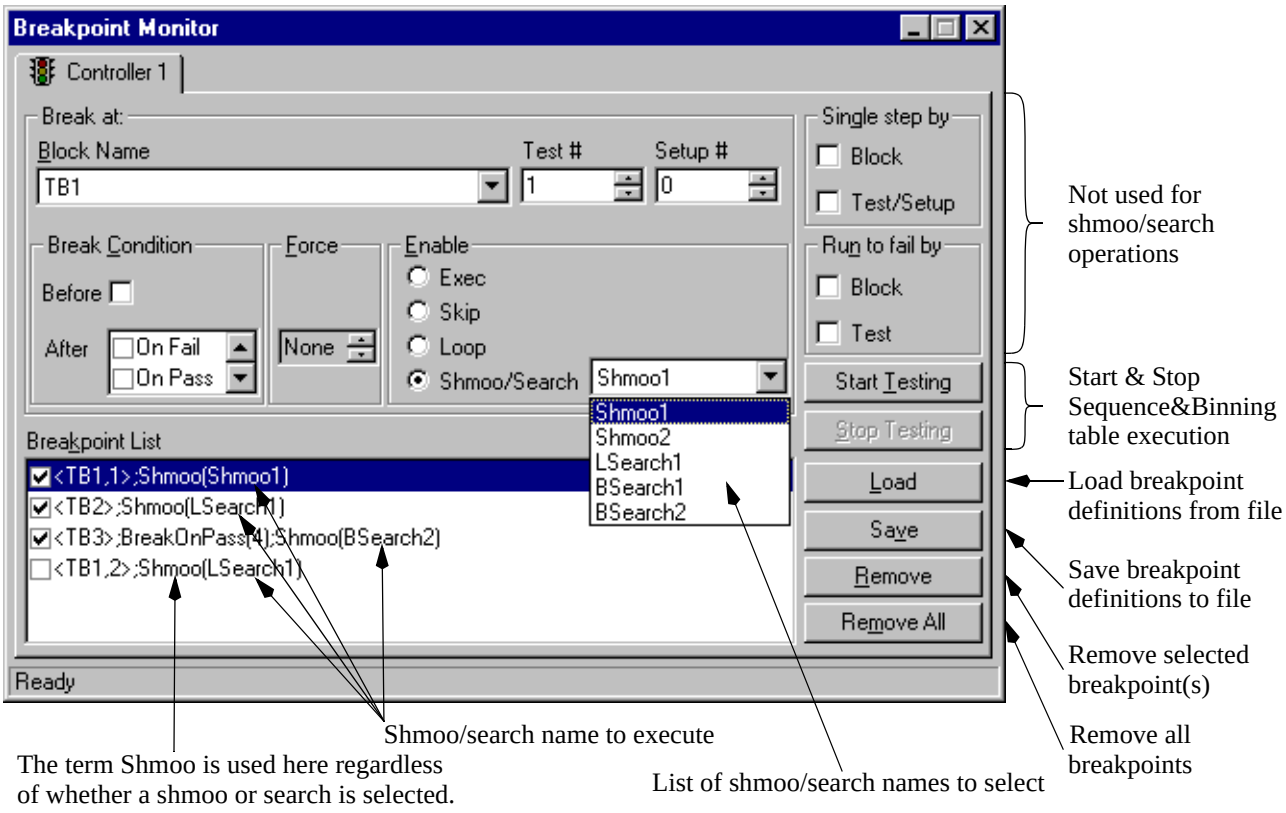


Figure-126: Shmoo Breakpoint Monitor Controls

There is a proper sequence to using these controls. When improperly used, confusion will occur. The order is simple:

1. Select the desired **`Test Block Name`**, from the pull-down list. A test block must be selected.
2. Select the desired **`Test#`**. Enter **`0`** when the entire test block is to execute at each shmoo/search data point.
3. Shmoos/searches only apply to test blocks and test numbers. Selecting a **`Setup#`** other than 0 will disable the Shmoo/Search option.
4. Shmoo/searches *do **not*** require setting a **`Break Condition`**. It is OK to set a Break Condition, but shmoo/search execution will be interrupted each time the condition occurs.

Note: a known problem exists when combining a shmoo breakpoint with an ***After* Breakpoint Condition**. Until this is corrected, it is recommended that this combination not be used. If used, it may be necessary to terminate the test program and UI using the Windows Task Manager.

5. Clicking any Break Condition or the Shmoo/Search Enable radio button causes a breakpoint to be entered into the **Breakpoint List**. This list is not directly editable, but can be modified by selecting the desired breakpoint and using the previous controls.
6. When all else fails, **Remove** any questionable breakpoints and start over.
7. Once one or more shmoo/search operations are successful, **Save** the breakpoint definitions to a file. All definitions are saved (it is not possible to save a subset). A standard windows file browser will be presented, to be used to specify disk, folder, and file name.

Note: use caution when saving into the test program's *debug* directory. Many users routinely delete this folder to recover disk space.

The example Breakpoint List below is used to describe several variations of Breakpoint Monitor shmoo/search definitions. Note that breakpoints are only operational during Sequence & Binning Table execution:

Four breakpoints are defined.
The first 3 are enabled (note the check box)
Each is documented further below

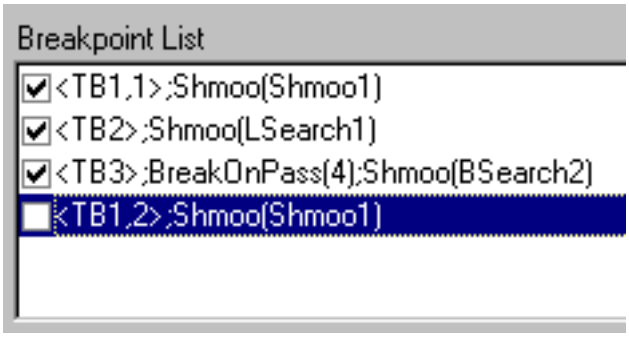
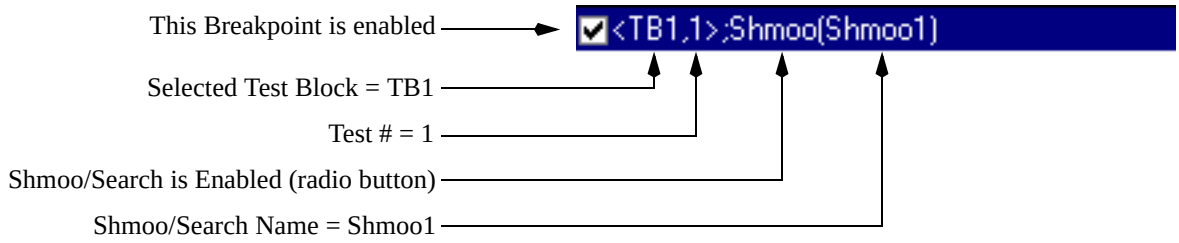


Figure-127: Shmoo Breakpoint List

First entry in the Breakpoint List:

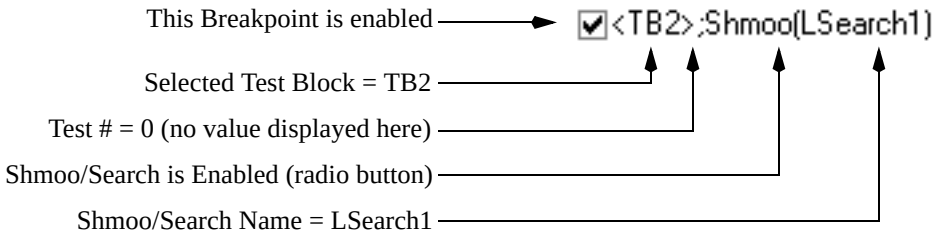


Using this example, the shmoo (or search) named Shmoo1 will execute when:

- The Sequence/Binning table is executed, and...
- Test block TB1 executes, and...
- Test number 1 executes

The test represented by Test# 1 will execute at each data point, and its pass/fail result will determine the results of the shmoo/search.

Second entry in the Breakpoint List

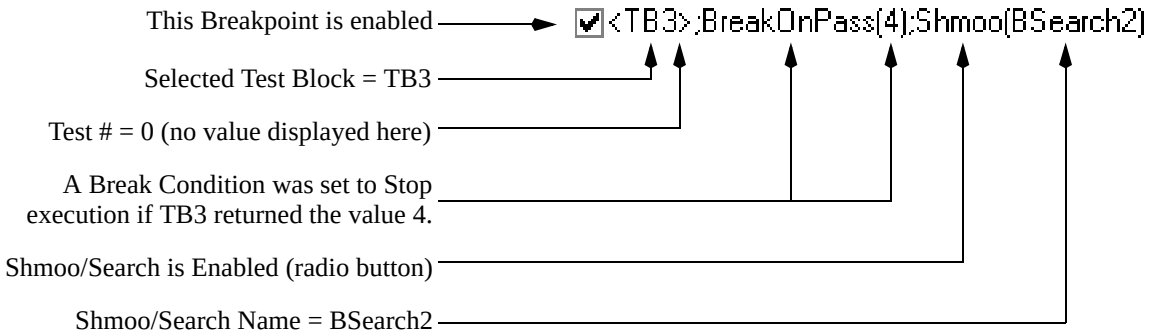


Using this example, the search (or shmoo) named LSearch1 will execute when:

- The Sequence/Binning table is executed, and...
- Test block TB2 executes

Since the specified Test# = 0, the entire test block will execute at each data point, and the value returned by the test block will determine the results of the shmoo/search.

Third entry in the Breakpoint List

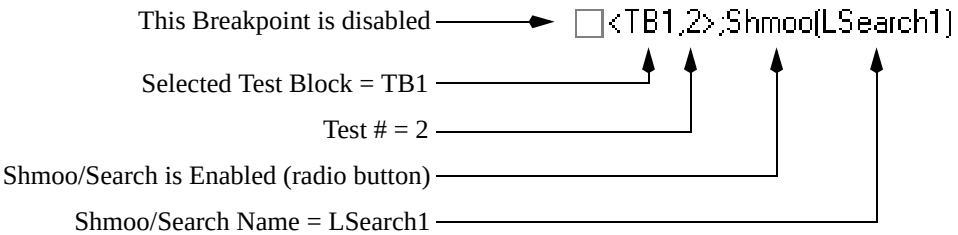


Using this example, the search (or shmoo) named BSearch2 will execute when:

- The Sequence/Binning table is executed, and...
- Test block TB3 executes

Since the specified Test# = 0, the entire test block will execute at each data point, and the value returned by the test block will determine the results of the shmoo/search. Any non-zero value returned by the test block is considered a PASS by the shmoo/search. Since a **`Break Condition`** is also set, shmoo execution will be interrupted if the test block returns the value 4. When this occurs, the Breakpoint Monitor **`Continue Execution`** button can be used to resume execution.

Fourth entry in the Breakpoint List



Using this example, the shmoo or search named LSearch2 will not execute because the breakpoint is disabled. Clicking in the box will enable the breakpoint.

6.15.9.2 Executing Shmoos and Searches Programmatically

See ShmooTool / SearchTool, Shmoo/Search Execution

As used here, the term *executing programmatically* means that shmoos or searches are invoked using C code in the test program i.e. not interactively.

The same methods documented in Executing Shmoos and Searches Interactively are used to create shmoo/search definitions and breakpoint definitions. To be used programmatically these definitions must be saved to disk files. See Shmoo Definition File and Breakpoint Definition File.

Then, using built-in UI User Variables, user-written C code can:

- Load a Shmoo Definition File
- Load a Breakpoint Definition File
- Specify an output file used to capture shmoo/search results (in ASCII).

Once these files are loaded, any subsequent execution of the Sequence & Binning Table will invoke any shmoo/search definitions triggered by any enabled breakpoints, andstore the output in the specified file.

Note: breakpoints function the same as when interactively using the Breakpoint Monitor. Thus if any **`Break Conditions`** are set program execution will react accordingly i.e. stop execution. However, since the interactive Breakpoint Monitor is not displayed, it may not be obvious to the user why program execution halted. Starting the Breakpoint Monitor allows the test program execution to be resumed by clicking the **`Continue Execution`** button.

Usage

The following code examples show how to use three built-in user variables to enable shmoos and searches from user-written C code.

```
remote_set( "ui_ShmooInput","c:/path/shmoo_defs.txt", -1 );
remote_set( "ui_ShmooOutputFile","d:/path/outfile.txt", -1 );
remote_set( "ui_BreakPointFile","d:/path/break_defs.txt", -1);
```

See UI User Variables. These functions must be called from `SITE_BEGIN_BLOCK()` or `SITE_CONFIGURATION()` block (see Limitations).

Note: all built-in UI User Variables are scoped to the UI process. This means that unloading the test program does NOT change the state of the variable in UI, and any subsequently loaded test programs may be affected. Terminating UI will reset these conditions.

Limitations

The Shmoo Definition File and Breakpoint Definition File are loaded once, when the program loads. For this reason, code which specifies `ui_ShmooInput` and/or `ui_BreakPointFile` must execute from the `SITE_BEGIN_BLOCK()` or `SITE_CONFIGURATION()` block.

6.15.10 Shmoos and Searches using User Variables

See ShmooTool / SearchTool,

The ability to shmoo/search User Variables is very powerful. It allows arbitrary user-written C code (the body code of the user variable) to be executed for each step of the shmoo/search.

This can be used to modify PE voltage/current values, DPS voltage/current values, timing, parametric parameters, etc. It can also be used to implement nonlinear operations, set limits, enforce user rules, etc.

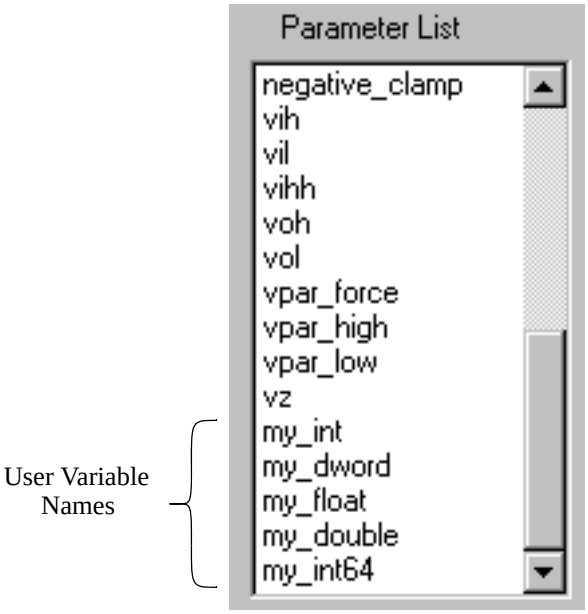
The following user variable types can be used as shmoo/search parameters:

- `INT_VARIABLE`
- `DOUBLE_VARIABLE`
- `DWORD_VARIABLE`
- `INT64_VARIABLE`
- `FLOAT_VARIABLE`

In the ShmooTool *Parameter List* list box, user variables appear with the name of the variable. For example, the following user variables will appear as shown below:

```
INT_VARIABLE( my_int, 1, "x") {}
DWORD_VARIABLE( my_dword, 1, "x") {}
FLOAT_VARIABLE( my_float, 0.0, "x") {}
DOUBLE_VARIABLE( my_double, 1, "x") {}
INT64_VARIABLE( my_int64, 1, "x") {}
```

```
// The "x" argument enables ShmooTool display, any value works
```



User variables appear at the bottom of the Parameter List. They are listed by type first (`INT`, `DWORD`, `FLOAT`, `DOUBLE`, `INT64`), then alphabetical by name. To be listed, the user variable must have a non-NULL label specified as the 3rd argument (the label itself is not displayed). For example, the following user variable will not appear in the list:

```
INT_VARIABLE( some_name, 1, "") {} // Note 3rd argument is NULL
```

Examples

Example 1:

The following example uses 2 user variables, without any body code, to generate a shmoo which looks like a *star* (not very useful, except as an example). An entire test block is executed to determine pass/fail at each point in the shmoo. The values of the user variables are modified within the test block, and determine whether any given test block execution passes or fails, which in turn causes the shmoo to display pass or fail. The example has four parts:

- C-Code:
- ShmooTool Setup
- BreakPoint Monitor Setup
- Shmoo Output

C-Code:

This example assumes the shmoo min/max parameters are 0->10 for both axis.

```
INT_VARIABLE( x_axis_star, 0, "x") {}
INT_VARIABLE( y_axis_star, 0, "y") {}
TEST_BLOCK( TB_star ) {
  if ( x_axis_star == y_axis_star )          return FALSE;
  if ( x_axis_star == 5 )                    return FALSE;
  if ( y_axis_star == (10 - x_axis_star) ) return FALSE;
  if ( y_axis_star == 5 )                    return FALSE;
  return TRUE;
}
```

ShmooTool Setup

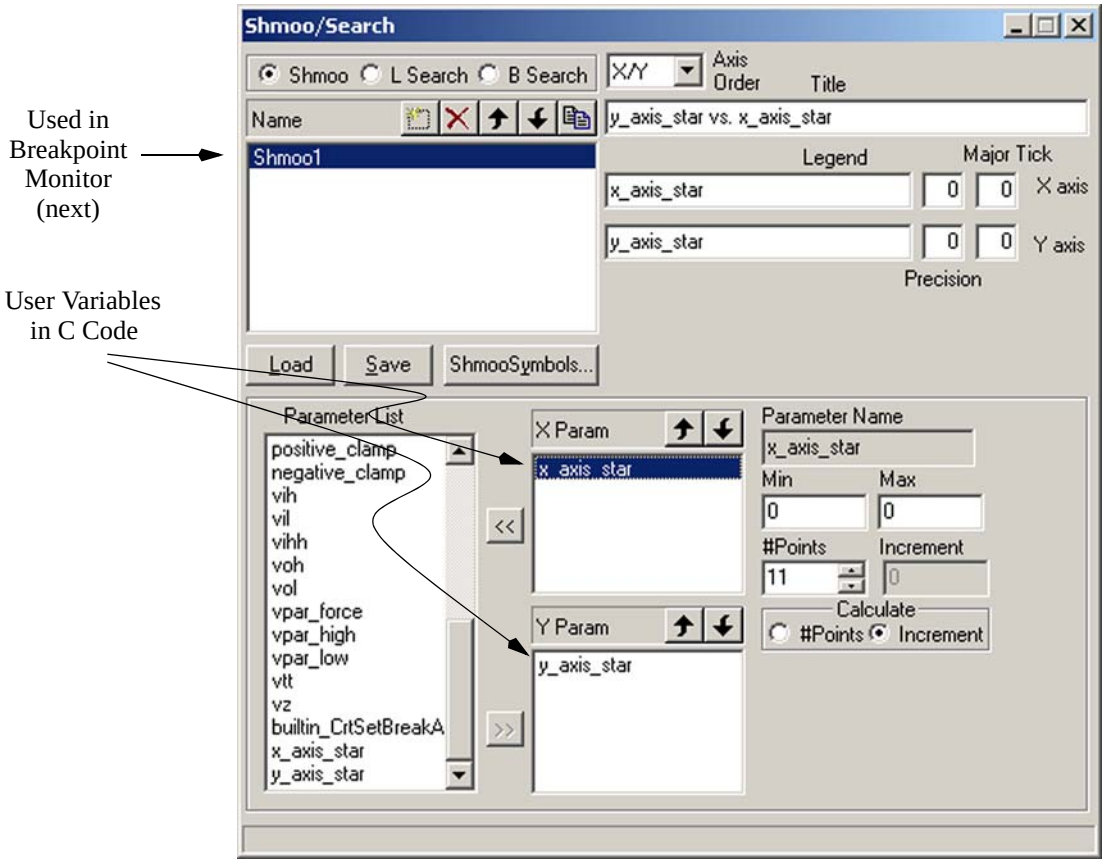


Figure-128: Shmoo User Variable Setup

BreakPoint Monitor Setup

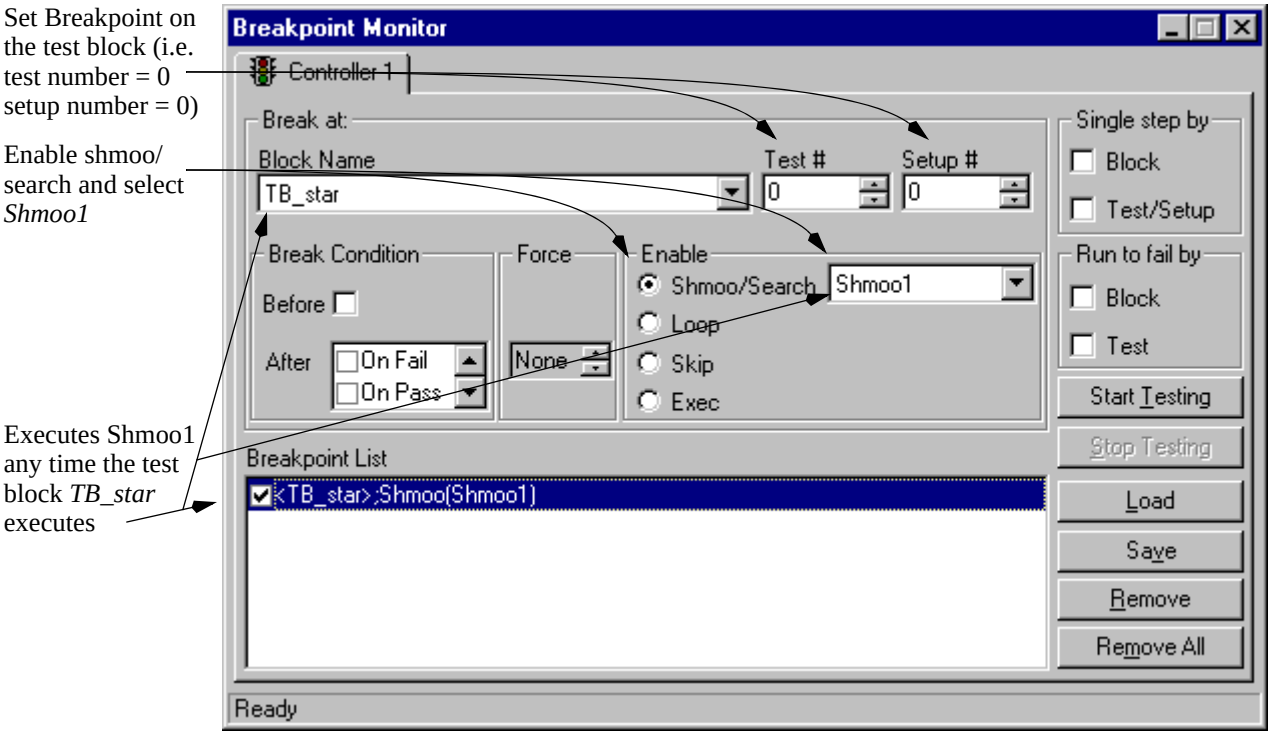
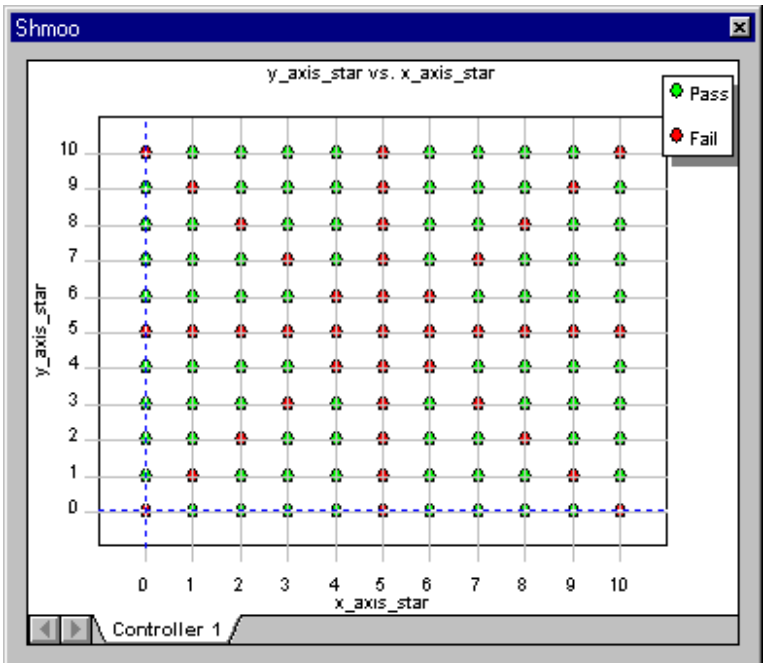


Figure-129: Shmoo Breakpoint Setup

Shmoo Output



(there is a star there, honest)

Figure-130: Shmoo Output

6.15.11 Shmoo Definition File

See ShmooTool / SearchTool

ShmooTool provides a method for saving shmoo/search definitions files on disk.

These files are created or replaced using the **Save** button in ShmooTool, and read into ShmooTool using the **Load** button:

Figure-131: Shmoo Definition File Controls

In either case, as standard Windows file browser is displayed, and used to select the desired disk, path, and file.

Note the following:

- The information is stored in ASCII files (*.txt), which are readable using any text editor.

Note: Nextest reserves the right to modify the format of these files at any time. Manually editing these files is therefore discouraged.

- The **Save** operation saves all the definitions shown in the Name list box. It is not possible to save a subset.
- The **Load** operation loads the entire contents of the selected file. Existing definitions are not removed. If a definition name in the load file matches an existing name in the Name list box, the user will be advised, and allowed to confirm or cancel.
- Using command line arguments, it is possible to specify a shmoo/search definition file to load as UI is started. See Starting UI from a Command Line, and `ui_ShmooInput`.

6.16 SummaryTool

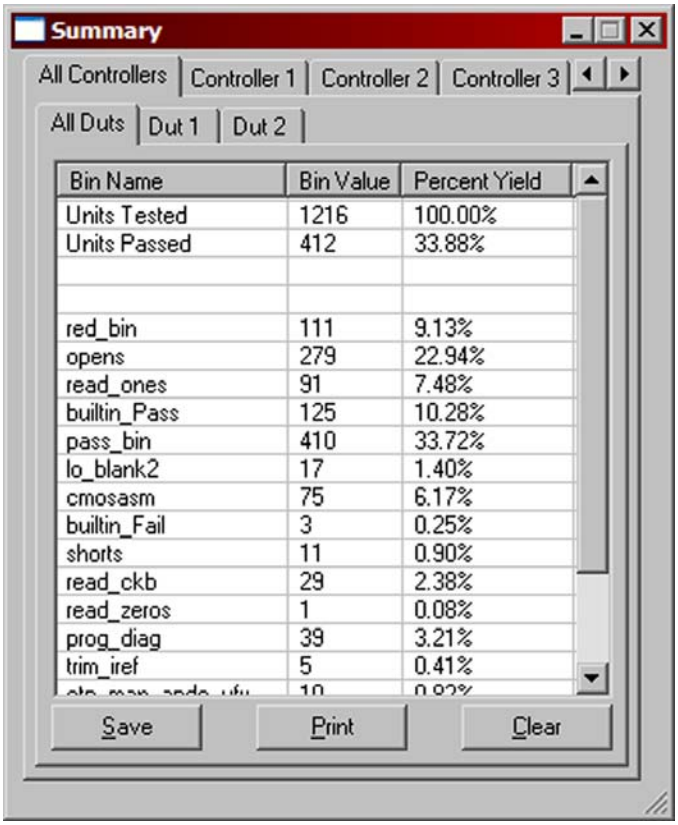
Description

UI - User Interface provides a built-in summary display which can be used to view the current values of Test Bins and Test Bin Groups.

To start SummaryTool:

- Type keyboard shortcut **`Alt+F3`**
- Choose **`View: Default Summary`**

This causes the Summary display to appear:



Note the following:

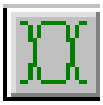
- Only nonzero values are displayed. If either `builtin_Pass` and/or `builtin_Fail` contains a non-zero count value they will also be displayed.

- The sum of all bins is also displayed (first). Note that the name for this value is *Units Tested*, which may be inappropriate since the value displayed is actually the sum of all bins. In some applications more than one bin may be incremented for each device tested.
- The *Save* option is used to write the summary values to a file chosen interactively from the file dialog to follow.
- The *Print* option is used to to print the summary values to a printer, installed on the host computer (running *Ui)* or over a network.
- The *Clear* option, available only in engineering mode, is used to clear the summary on the selected test site (controller). This also clears the related Test Bins and Test Bin Groups
- The *AllDUTs* and individual DUT tabs (*DUT1*, etc.) are used to select which summary data will be displayed. These are useful in Multi-DUT Test Programs.

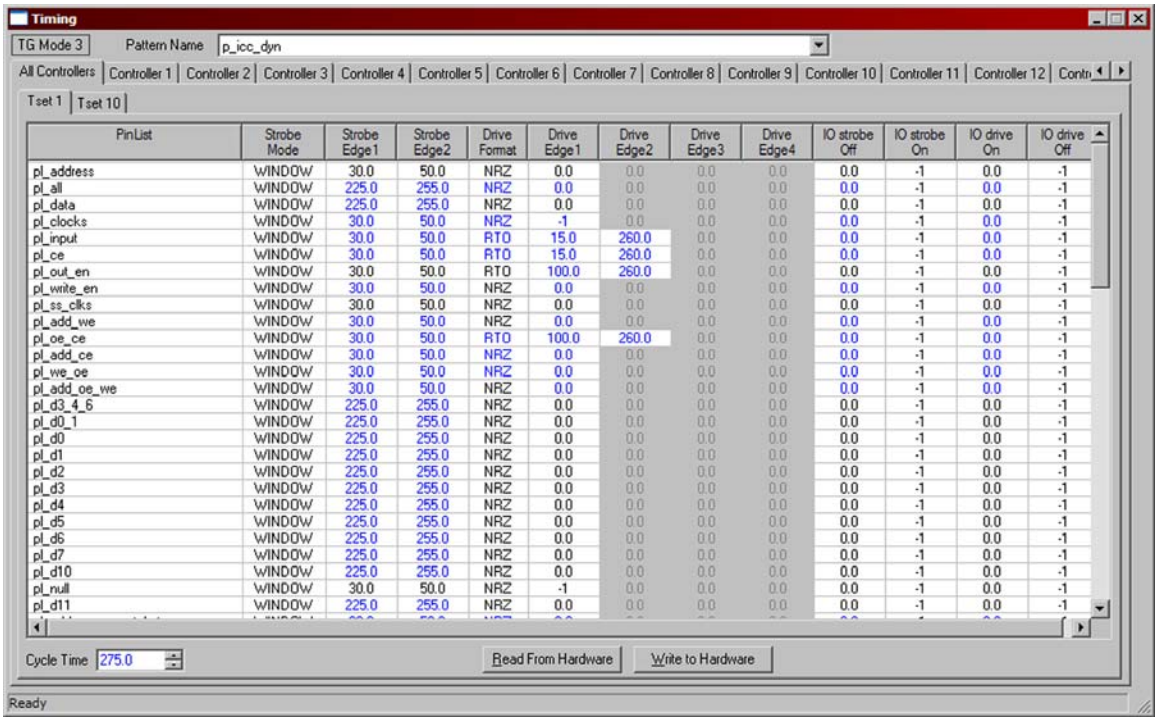
This dialog is intended to provide runtime visibility of bin values. Since the format of values displayed is fixed, it is not likely to satisfy the needs of all users. To obtain a customized summary, the functions available in Test Bin Functions and Test Bin Group Functions can be used, to generate text output, display values in User Dialogs or User Tools, save output to disk files, etc.

6.17 TimingTool

To start Timing Tool:

- Click on the TimingTool icon from the *Ui* toolbar 
- Type keyboard shortcut **`Ctrl+I`**
- Choose **`Tools: Timing`**

The image below shows the Magnum 1/2/2x TimingTool:



Note the following:

- The number of Controller tabs displayed is based on the system in use.
- The current `tgmode()` state of the hardware is shown in the upper left area of the TimingTool display. This is a display-only value, which is only updated when TimingTool is first started e.g. it is necessary to terminate and restart TimingTool to update this value, which will be necessary, for example, when execution stops at different breakpoints.

- The Pattern Name list box allows the user to select a test pattern, which then causes the TimingTool display to display only the time-sets actually used in the test pattern. To see all time-set selection tabs, select the pattern named <none>.
- The cycle period value of the currently selected time-set is displayed in the lower left corner of the tool.

6.18 User Variables Tool

Note: this tool was completely re-designed in software release h2.2.xx/h1.2.xx. The information below only applies to the new design.

This section includes the following:

6.18.1 Overview

See User Variables Tool.

User Variables Tool may be used to display and optionally modify User Variables. The image below displays 28 user variables which existed in an example test program. This example includes several instances of each type of user variable type:

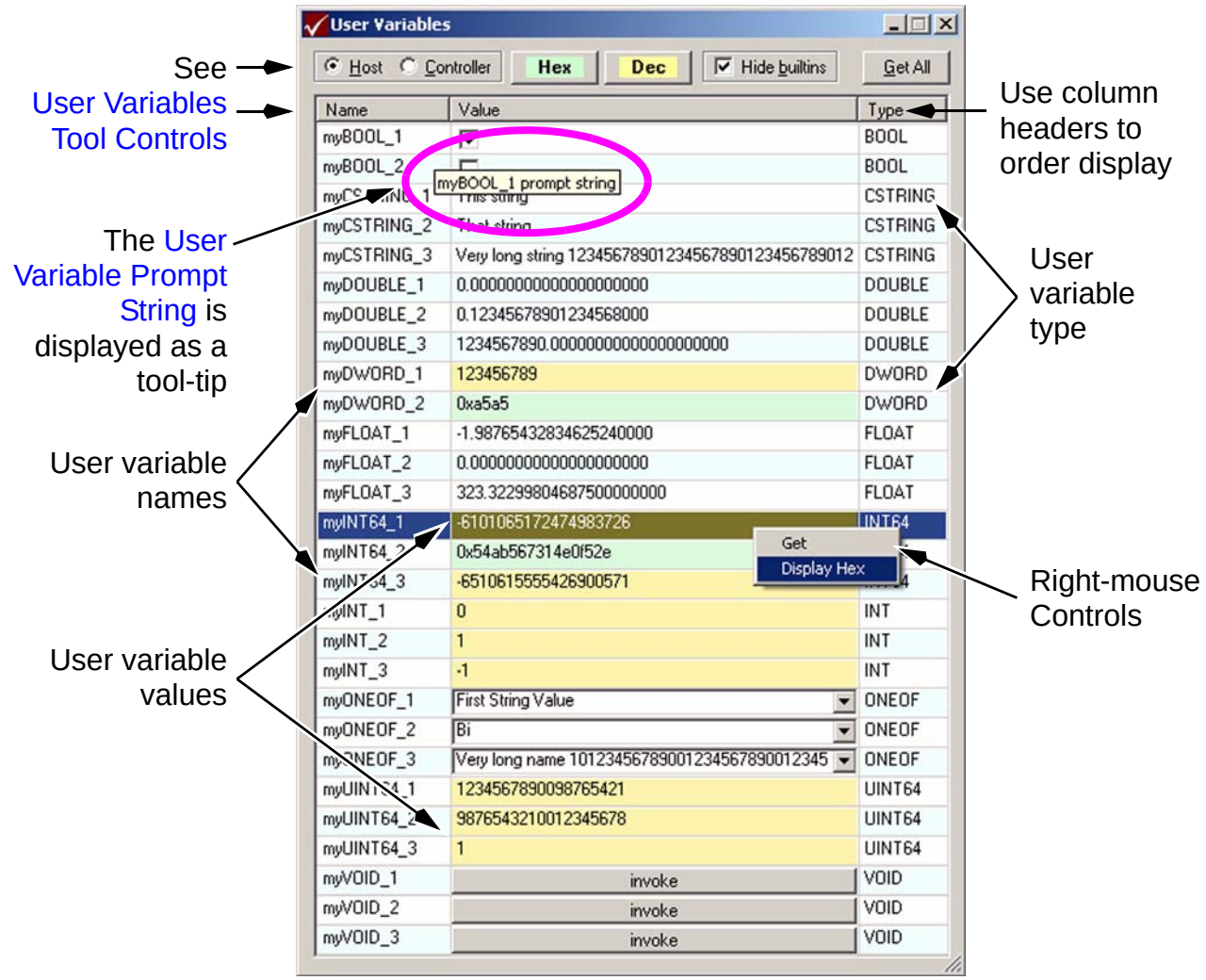


Figure-132: User Variables Tool Display

6.18.2 Starting User Variables Tool

See User Variables Tool.

Three methods are available to start User Variables Tool:

- Click on the User Variables Tool icon in the *UI* tool-bar 
- Type the keyboard shortcut **`Ctrl+U`**
- Choose **`Tools: User Variables...`**

6.18.3 User Variable Prompt String

See User Variables Tool, Overview.

Only user variables which have a prompt string are displayed in User Variables Tool. For example:

```
BOOL_VARIABLE( myBOOL_1, TRUE, "myBOOL_1"){}
BOOL_VARIABLE( myBOOL_3, TRUE, ""){}
```

`myBOOL_3` will not be displayed in User Variables Tool because it has a NULL prompt string.

As the cursor is moved onto a given user variable the standard Windows tool-tips are used to display each user variable's prompt string (see example in User Variables Tool Display).

6.18.4 User Variables Tool Controls

See User Variables Tool.

User Variables Tool has three sets of controls:

- Display Option Controls
- Display Sort Controls
- User Variable Display/Modification Controls

Display Option Controls

The following image shows the display option controls. The table below describes these controls:

Figure-133: User Variables Tool Display Option Controls

The following table describes the various User Variables Tool display option controls:

Control	Purpose
Host	When selected, any changes to user variable values are made in the Host only and the user variable's body-code executes in the Host. Clicking Get All will read values from the Host.
Controller	When selected, any changes to user variable values are made in all Sites and the user variable's body-code executes in each Site. Clicking Get All will read values from the first Site.
Hex	When selected, all integer numerical values displayed in User Variables Tool are displayed as hexadecimal values and the background color of those values is light-green. Note that the numerical base of a single integer value can also be changed by first selecting the target user variable and then selecting the desired base via the right-mouse button.
Dec	When selected, all integer numerical values displayed in User Variables Tool are displayed as decimal values and the background color of those values is light-yellow. Note that the numerical base of a single integer value can also be changed by first selecting the target user variable and then selecting the desired base via the right-mouse button.
Hide builtins	When selected, the built-in user variables defined by the system software are not displayed in User Variables Tool. See Built-in User Variables.
Get All	Updates the value displayed for each user variable displayed in User Variables Tool. This should be used any time user-code executes which may have changed one or more user variable values. Reads the values from the Host or first Site, depending on which is currently selected (see **Host**/**Controller** above).

Figure-134: User Variables Tool Display Option Controls

Display Sort Controls

The following image shows the display sort controls. The controls are described below:

Figure-135: User Variables Tool Display Sort Controls

The width of each column can be resized using the mouse.

Clicking on **Name** will sort the list of user variables alphabetically, by name.

Clicking on **Value** has no effect.

Clicking on **Type** will sort the list of user variables alphabetically, by type.

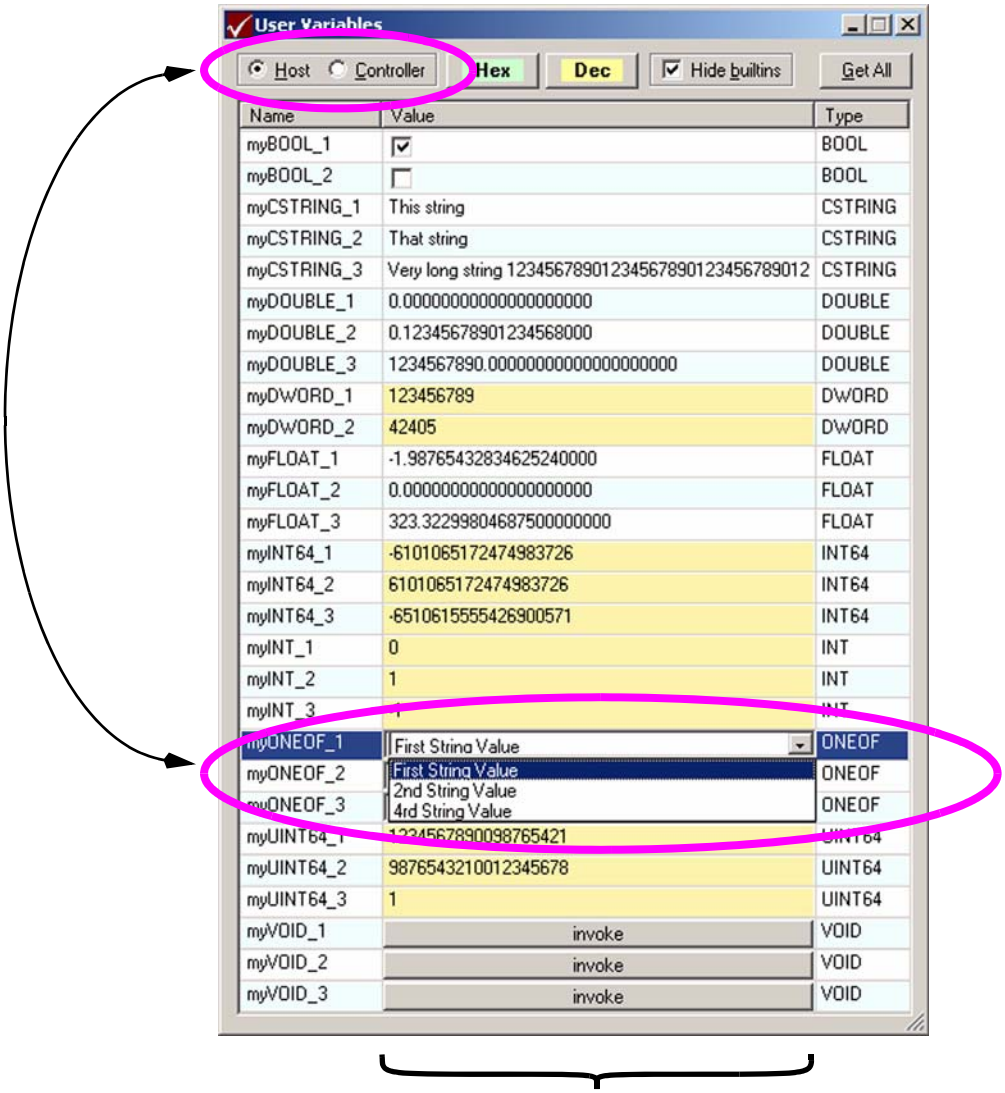
The control key may be used to modify the sort algorithm. This is typically only useful as follows:

- Click on **Type** to sort the list alphabetically, by type.
- Press and hold the control key and click on **Name** to sort the list of user variables alphabetically, by name.

The result: the user variables remain sorted by **Type** but will also be sorted by **Name** within each **Type**.

User Variable Display/Modification Controls

The following image shows the main User Variables Tool display area, which is used to view each user variable's name, value and type and optionally to modify a user variable's value:



Modify values in the **`Value`** column. The value is changed in the Host or all Sites depending on the **`Host`**/**`Controller`** selection.

Figure-136: User Variable Display/Modification Controls

A user variable's name is that assigned by the user in test program code. A user variable's name cannot be changed using User Variables Tool.

A user variable's value is the value read when User Variables Tool is invoked or when the Get All button is clicked. Values are read from the Host or first Site, depending on the **Host**/**Controller** selection.

A user variable's type is determined by the macro used in the test program to define each user variable. A user variable's type cannot be changed using User Variables Tool.

The method used to change a value depends on the *type* of the user variable:

- BOOL_VARIABLE: left-click the check-box to toggle the value. This takes effect immediately.
- Value-based variables: left-click the current value to select the current value then type the new value. This applies to DWORD_VARIABLE, INT64_VARIABLE, CSTRING_VARIABLE, FLOAT_VARIABLE, UINT64_VARIABLE, DOUBLE_VARIABLE and INT_VARIABLE. When value-based variables are modified typing **Enter** or changing the focus to a different variable value causes the change to take effect immediately. The **Esc** key can be used to cancel an edit.
- Each ONEOF_VARIABLE has a pull-down menu which may be used to select a new value from the values defined for that variable. Above, the value selection for myONEOF_1 has been selected. Selecting a new value from the pull-down menu causes the change to take effect immediately. The **Esc** key can be used to cancel an edit.
- A VOID_VARIABLE does not have not value but can be invoked by clicking the **Invoke** button in the **Value** column.

Also note:

- When a user variable is modified in User Variables Tool its body-code IS executed in the Host or all Sites depending on the **Host**/**Controller** selection.

Note: the user variable's body-code WILL be executed any time that variable's value has been selected for editing, even though the value may not actually be changed. The **Esc** key can be used to cancel an edit, which will prevent the body-code from executing.

6.18.5 Built-in User Variables

See User Variables Tool, Display Option Controls.

The system software defines a number of built-in user variables, each named with the prefix `builtin_`. These variables allow the system software to use underlying support for user variables, just like user code. For the most part, these variables are used for internal needs; the exceptions are noted in the table below (and some of these have no value to the user).

The **`Hide builtins`** control in User Variables Tool can be used to display or hide these user variables. See Display Option Controls.

The table below lists these built-in user variables. Only the variables which have a Purpose description below should be manipulated by the user:

Variable Name	Purpose
`builtin_CrtSetBreakAlloc`	See `_CrtSetBreakAlloc` in MSDN documentation. Useful for memory leak diagnosis.
`builtin_batch_file`	Passes the specified path/file name to `set_values_from_file()` which is executed in the Host or all Sites based on the **`Host`**/**`Controller`** selection. Rules for `set_values_from_file()` apply.
`builtin_dump_globals`	Specify the path/file name which will be generated and contain global variable declarations suitable for `#include` in a DLL source file. See comments in the generated file. Default path is test program \debug\ folder.
`builtin_dynload`	Loads a specified DLL and initializes resources from that DLL. Loads in the DLL in the Host all or Sites depending on the **`Host`**/**`Controller`** selection. See Loading DLLs.
`builtin_fatal`	Generates an output message containing the specified string in the Host or all Sites depending on the **`Host`**/**`Controller`** selection and unloads the test program.
`builtin_getenv`	Not useful.
`builtin_invoke_dialog`	Can be used to invoked the specified user dialog which must be defined in the currently loaded test program.

Variable Name	Purpose
builtin_invoke_sequencetable	Can be used to execute the specified Sequence & Binning Table. Only useful when **Controller** is selected.
builtin_invoke_testbin	Can be used to execute the specified Test Bin. Only useful when **Controller** is selected.
builtin_invoke_testblock	Can be used to execute the specified Test Block. Only useful when **Controller** is selected.
builtin_loadlibrary	See LoadLibrary in MSDN documentation. Instead, use builtin_dynload, which performs additional house-keeping functions.
builtin_message_box	Displays a message box displaying the specified string. Below, the value test this was entered. Note that if OK is not selected within a few seconds a time-out warning dialog will be displayed. 
builtin_output	Generates an output message containing the specified string in the Host or all Sites depending on the **Host**/**Controller** selection.
builtin_putenv	May be used to set the value of an environment variable. The variable will be defined only within the scope of the program executing in the Host or all Sites depending on the **Host**/**Controller** selection.

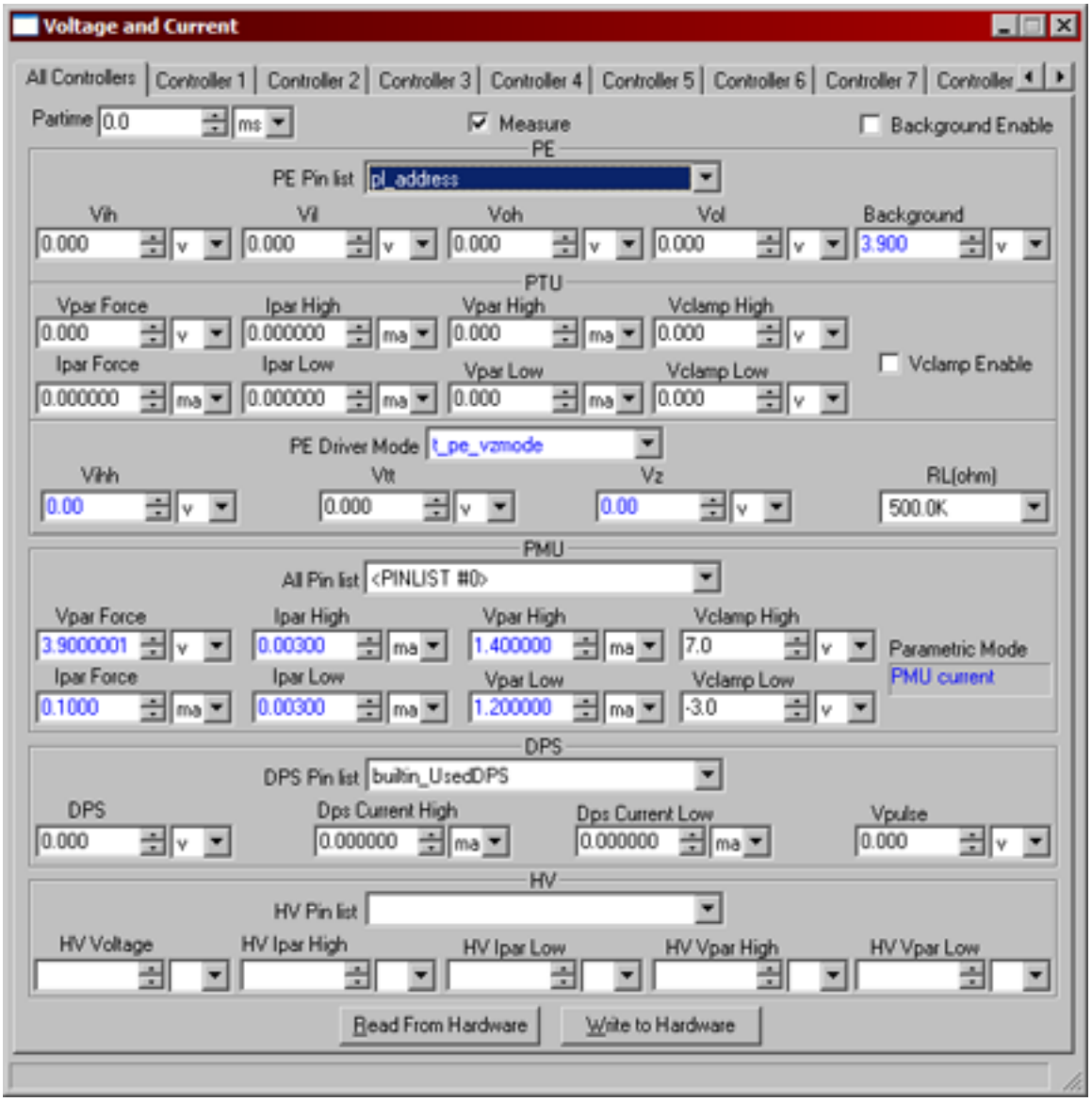
Variable Name	Purpose
builtin_remote_signal	Can be used to send a signal to a pending remote_wait(). The value specified must match the signal name specified by remote_wait(). The signal is sent to the Host or all Sites depending on the **Host**/**Controller** selection
builtin_resource_deallocate	See Resource Control Functions.
builtin_resource_initialize	See Resource Control Functions.
builtin_unload	The complement of builtin_dynload. Unloads the specified DLL and deallocates the associated resources.
builtin_update_control	Can be used to update the specified control in a user dialog which is currently being displayed. The control is identified by its associated user variable name.
builtin_warning	Generates a warning message containing the string entered in the Host or all Sites depending on the **Host**/**Controller** selection.
builtin_what_exe	Displays path and file name of the currently loaded test program. This is read-only.

6.19 Voltage and Current Tool

To start VoltageTool:

- Click on the Voltage Tool icon from the *Ui* toolbar 
- Type keyboard shortcut **`Ctrl+R`**
- Choose **`Tools: Voltage and current…`**

The image below shows the Magnum 1/2/2x VoltageTool interface:



6.20 WaveTool

Note: this section is a work in progress, is not complete and may contain errors. It is included anyway, because the information which is complete may be of some value. Note that the design of the various dialogs is evolving and changes are likely as the implementation proceeds. Thank you for your patience.

This section contains the following topics:

6.20.1 Example Display

See WaveTool.

The following image shows a typical WaveTool display:

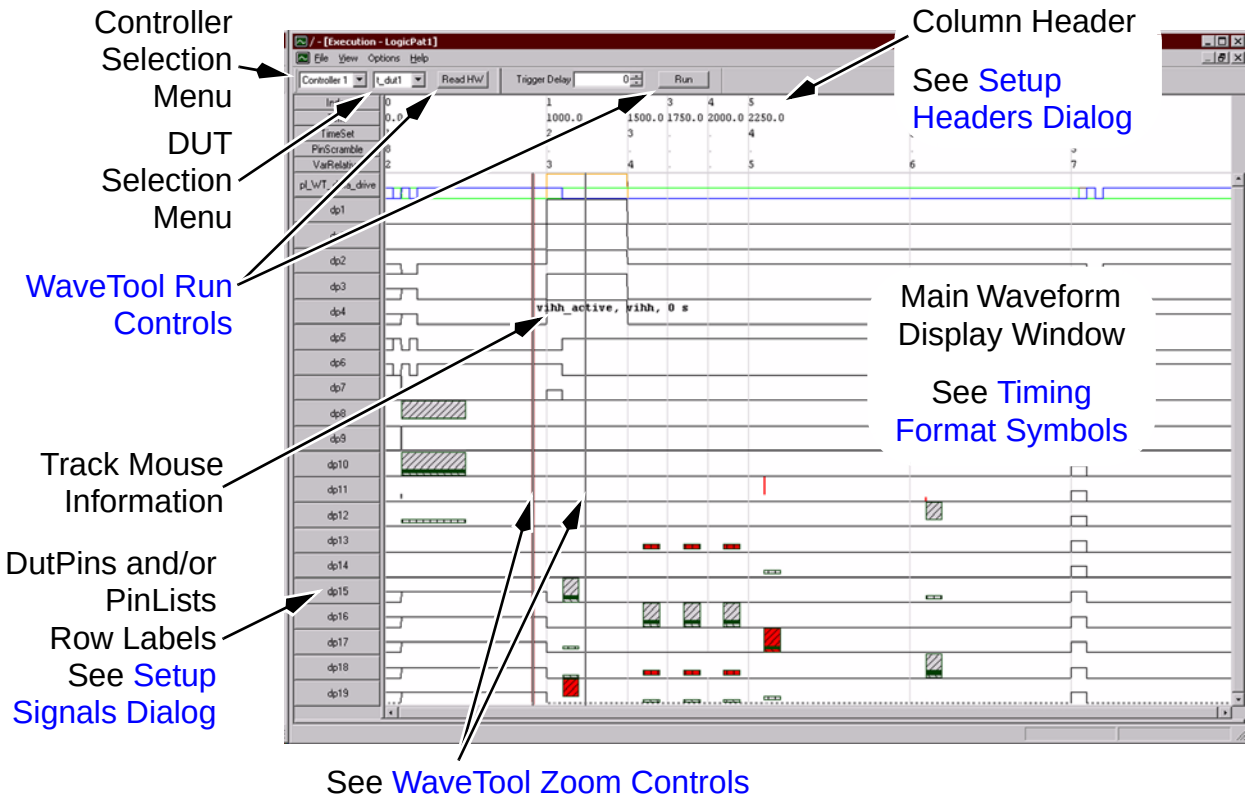


Figure-137: WaveTool Display

The image above shows WaveTool with a well configured display. When WaveTool is initially started, the system software inserts one pin into the Display Signals List and updates the Main Waveform Display Window to show one cycle of the most recently executed test pattern. Subsequently, the user must use the various WaveTool Setup Controls to change the various display options, as reflected in the image above.

6.20.2 Overview

See WaveTool.

WaveTool has the following capabilities:

- Graphically display format and timing information (waveforms) for one or more selected pin(s) or pin list(s) of specified cycles of a selected test pattern source file. This can include time-set selection, pin scramble selection, etc. See **Source->FileOrder** in Setup Acquire Input Controls.
- Display the sequence of format and timing generated on one or more pins/pin lists vs. the per-cycle test pattern execution sequence, as acquired in the hardware History RAM. See **Source->History** in Setup Acquire Input Controls. This can include time-set selection, pin scramble selection, etc. The Magnum 1 History RAM always contains the last 512 cycles executed.
- Display the sequence of format and timing from the instruction execution sequence of a Logic Test Pattern, as captured using the Logic Error Catch (LEC). See **Source->Lec** in Setup Acquire Input Controls.

In addition:

- WaveTool has several user selectable color schemes, see WaveTool Color Schemes.
- WaveTool supports tool setup files, allowing configurations to be saved and reused.
- WaveTool a *.ini* file which records tool size, location, etc.
- WaveTool can display waveform amplitudes using symbolic levels or actual values read from the hardware. See **Default Levels**.

6.20.3 Starting WaveTool

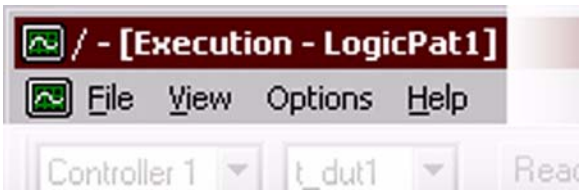
See WaveTool.

WaveTool is started from UI's Tool menu by selecting **Tool->Wavetool** or type `Control+Alt+V`.

6.20.4 WaveTool Tool-bar Controls

See WaveTool.

WaveTool's tool-bar includes several pull-down menus used to select between options or perform various actions:



See:

- WaveTool Tool-bar File Control Options
- WaveTool Tool-bar View Control Options
- WaveTool Tool-bar Options Control Options
- WaveTool Tool-bar Help Control Options

The WaveTool tool-bar's **File** options are:

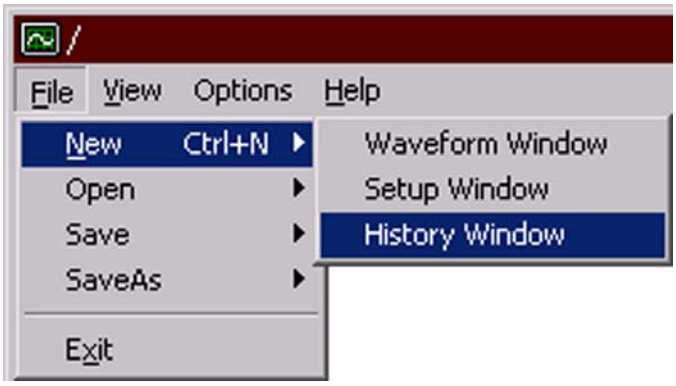


Table 6.20.4.0-1 WaveTool Tool-bar File Control Options

Control	Description
New->Waveform Window	Invoke a new waveform display window. Since only one window is supported this control is only useful if the window has been terminated.
New->Setup Window	Invoke a new WaveTool Setup Dialogs window. See WaveTool Setup Controls. Only one window is supported. If it exists (has not been terminated) it will be terminated and restarted with all values set = default values.
New->History Window	Invoke a new History RAM Display window, to display the contents of the History RAM.
Open->Setup File	Select and load a WaveTool Setup File from disk. Presents a standard file browser, initially pointing to the currently loaded test program's *\Debug* folder. WaveTool Setup Files have the *.suf* file name suffix.
Save->Setup File	Save the current WaveTool setup (as defined using the WaveTool Setup Dialogs) to the current WaveTool Setup File .
SaveAs->Setup File	Save the current WaveTool setup (as defined using the WaveTool Setup Dialogs) to a specified WaveTool Setup File on disk. If Presents a standard file browser, initially pointing to the currently loaded test program's *\Debug* folder. WaveTool Setup Files have the *.suf* file name suffix.
Exit	Terminate WaveTool.

The WaveTool tool-bar's **View** options are:

Table 6.20.4.0-2 WaveTool Tool-bar View Control Options

Control	Description
Setup Window	Display or hide WaveTool Setup Dialogs window. See WaveTool Setup Controls.
Run Controls	Show or hide WaveTool Run Controls.

The WaveTool tool-bar's **Options** options are:

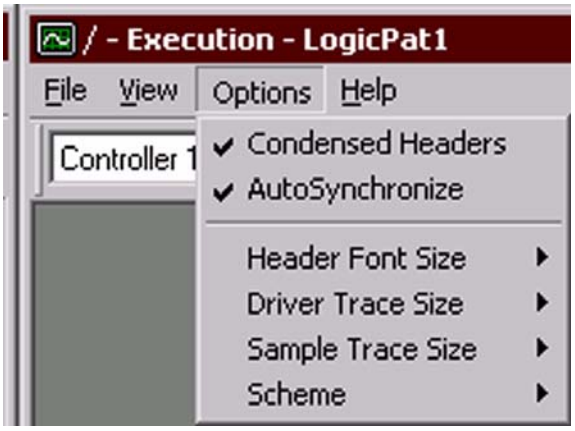


Table 6.20.4.0-3 WaveTool Tool-bar Options Control Options

Control	Description
Condensed Headers	By default, WaveTool displays some header information only when it changes from the previous cycle. This makes it easy to see when, for example, the `TimeSet` or `PinScramble` selection changes from one cycle to the next. If **Condensed Headers** is disabled, this information is displayed in every cycle.
AutoSynchronize	By default, the WaveTool display will be updated when the test pattern currently selected in the Setup Acquire Execute Controls is executed using mechanisms external to WaveTool. If **AutoSynchronize** is disabled WaveTool will only be updated if the WaveTool Run Controls are used.

Table 6.20.4.0-3 WaveTool Tool-bar Options Control Options *(Continued)*

Control	Description
Header Font Size	Use to change the size of text displayed in WaveTool's Column Headers. Default = `medium`. Changing this value has no effect until one of the WaveTool Run Controls is selected.
Driver Trace Size	Use to change the size of lines used to draw driver waveforms in the WaveTool Display. Default = `thin`. The other options are `none`, `medium` and `thick`. Changing this value has no effect until one of the WaveTool Run Controls is selected.
Scheme	Use to change the color scheme used in the WaveTool Display, see WaveTool Color Schemes. Default = `white`. The other options are `green` and `black`. The selection affects both the window background color and the color of the lines used to draw waveforms. Changing this value has no effect until one of the WaveTool Run Controls is selected.

The WaveTool tool-bar's **`Help`** options are:

Table 6.20.4.0-4 WaveTool Tool-bar Help Control Options

Control	Description
Help Topics	Displays the Nextest Documentation Index.
Creating Trace Files	See Creating WaveTool Trace Files.
About WaveTool	Displays the WaveTool version number.

6.20.5 WaveTool Setup Files

See WaveTool Tool-bar Controls.

The various configuration options set using WaveTool Setup Controls (i.e. the options set in the Setup Signals Dialog, Setup Headers Dialog) may be saved and reused later to quickly configure WaveTool. This is done, by the user, by saving any number of configurations, each to a separate WaveTool configuration file (more below) and subsequently loading the desired file.

Note the following:

- A WaveTool configuration file is created using **`File->SaveAs->Setup File`**.
- Changes made to the currently loaded configuration file may be saved using **`File->Save->Setup File`**.
- A configuration may be loaded (read) using **`File->Open->Setup File`**. This immediately updates the WaveTool Setup Controls however the WaveTool Display is not updated until one of the WaveTool Run Controls (`ReadHW` or `Run`) is used.
- When a file browser is displayed, by default it points to the currently loaded test program's *\Debug* folder.
- WaveTool configuration files use the *.suf* file name suffix.

6.20.6 WaveTool Setup Controls

See WaveTool Tool-bar Controls.

The image shown in WaveTool Display shows WaveTool with a well configured display. When WaveTool is first started, the system software inserts one pin into the Display Signals List and updates the Main Waveform Display Window to show one cycle of the most recently executed test pattern. Subsequently, the user may use the WaveTool Setup Dialogs shown below or a WaveTool Setup File may be loaded to change the various WaveTool display

options to display the desired information:

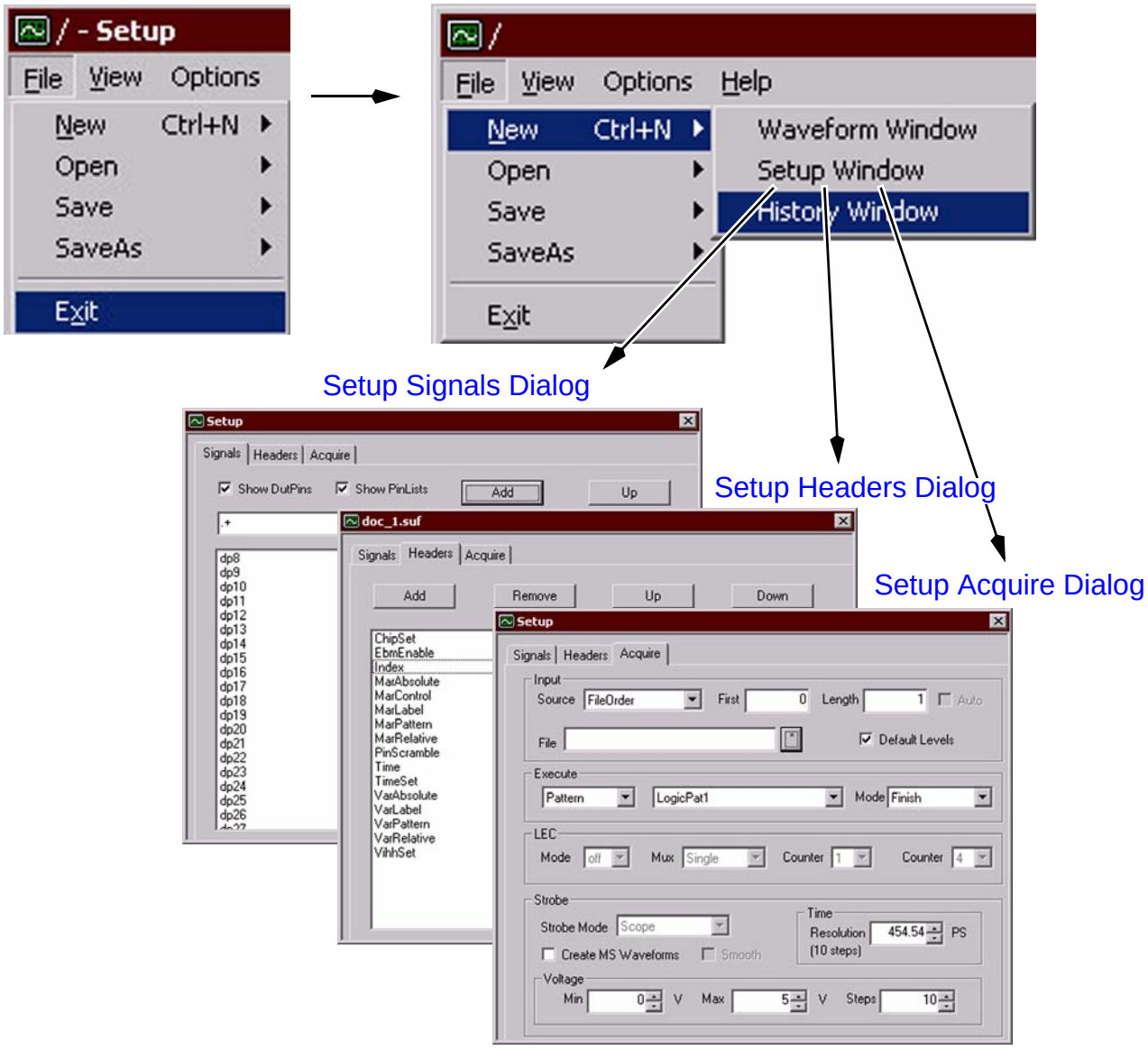


Figure-138: WaveTool Setup Dialogs

6.20.6.1 Setup Signals Dialog

See WaveTool.

This dialog is invoked by selecting the **Signals** tab in the WaveTool Setup Controls .

The setup **Signals** dialog is used to select which pin(s) will be displayed in WaveTool.

This dialog is invoked by selecting the **Signals** tab after using either **File->New->Setup Window** or by loading a previously saved setup using **File->Open->Setup File**.

Only the DutPins and/or PinLists shown in the Display Signals List will be displayed in WaveTool. DutPins and/or PinLists are added to the Display Signals List by selecting items from the Signal Selection List and clicking the **Add** button. Below, DutPins `dp1` through `dp19` have been added to the Display Signals List:

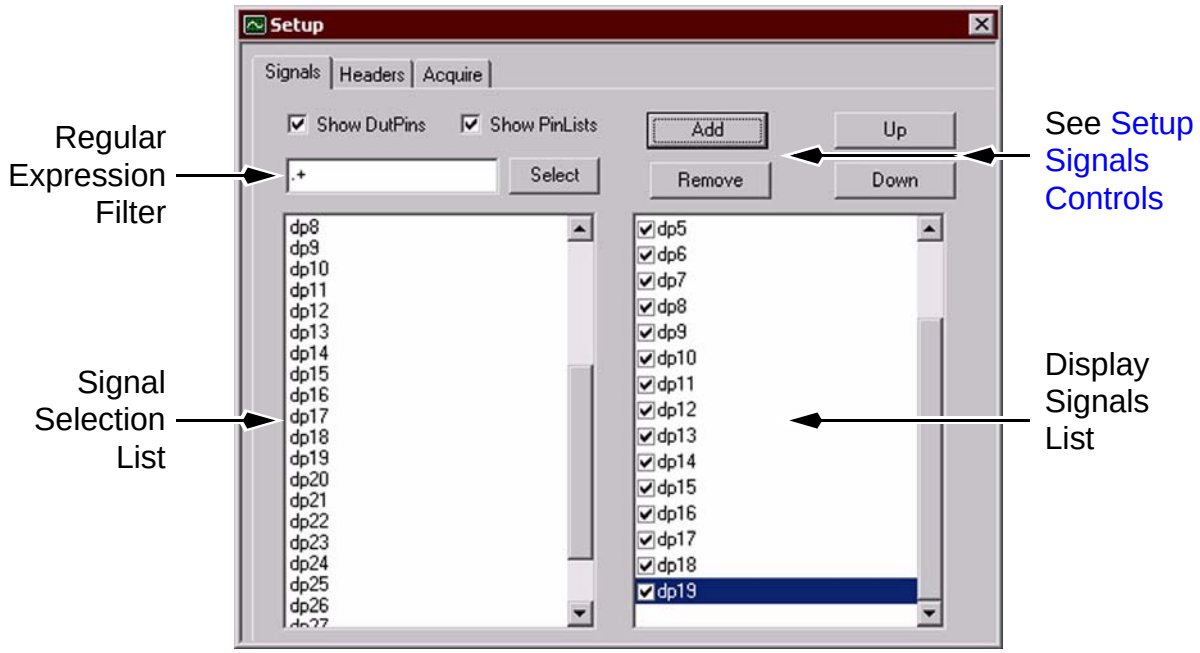


Figure-139: Setup Signals Dialog

The following table describes the remaining controls in this dialog:

Table 6.20.6.1-1 Setup Signals Controls

Control	Description
	Causes DutPins to be displayed for selection in the Signal Selection List.
	Causes PinLists to be displayed for selection in the Signal Selection List. When a PinList is displayed in WaveTool the format symbols drawn in each cycle are the composite of all formats used on all pins in the PinList (per cycle). This can be confusing to interpret, see WaveTool Timing Format Symbols.
	Used in conjunction with the Regular Expression Filter. This will filter the list of DutPins/PinLists displayed in the Signal Selection List based on the specified regular expression. The default expression (**.+**) causes all DutPins/PinLists to be displayed: **.** = any single character **+** = zero or more characters
	Causes the DutPins/PinLists selected in the Signal Selection List to be added to the Display Signals List.
	Removes any DutPins/PinLists selected in the Display Signals List from the list.
 	Moves any DutPins/PinLists selected in the Display Signals List up or down in the list. Used to reorder how pins are displayed in Main Waveform Display Window.
	Only DutPins/PinLists which are selected will be moved Up/Down or Removed when these controls are used.

6.20.6.2 Setup Headers Dialog

See WaveTool.

This dialog is invoked by selecting the **Headers** tab in the WaveTool Setup Controls .

The **`Setup->Header`** dialog is used to determine the information displayed in the Column Header area of each column in the WaveTool Display.

When WaveTool is first started, the system software inserts one value: `index` into the Headers List. Subsequently, the user must use the Setup Header dialog to change the information displayed in Column Headers.

Header options are added to the Header Options Display List by selecting Column Header Options from the Header Option Selection List and clicking the **Add** button. In the image below the Index, Time, TimeSet, PinScramble and VarRelative options have been added to the Header Options Display List:

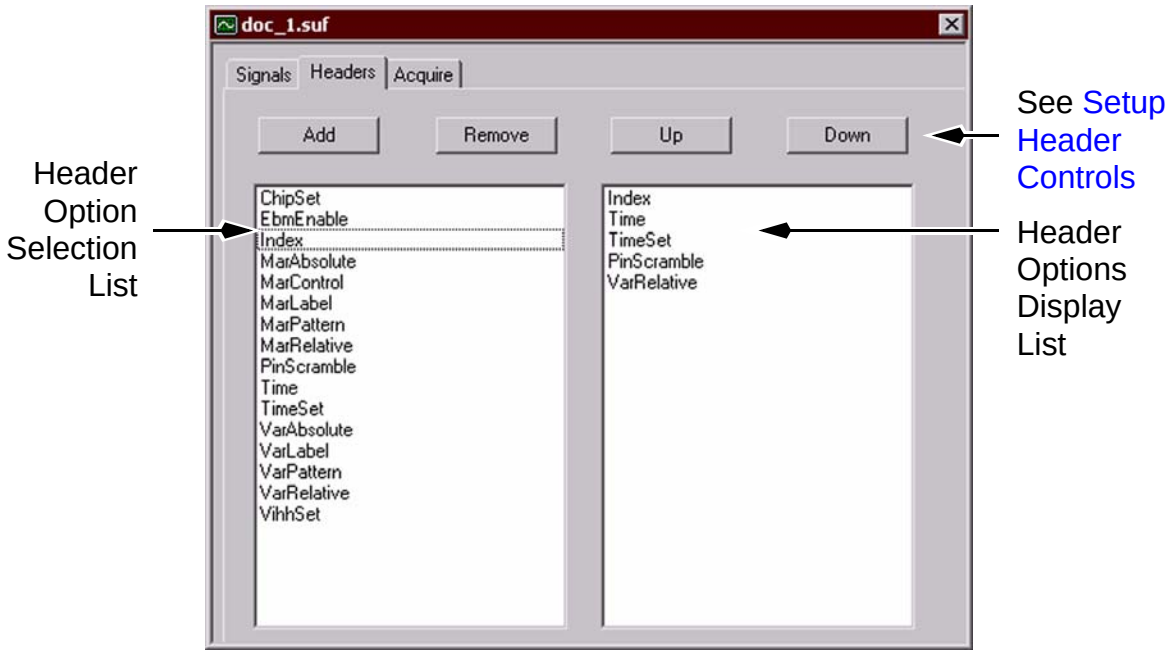


Figure-140: Setup Headers Dialog

The following table describes the remaining controls in this dialog. Below this is the table of Column Header Options:

Table 6.20.6.2-1 Setup Header Controls

Control	Description
Add	Causes the selections in the Header Option Selection List to be added to the Header Options Display List.
Remove	Removes selected options from the Header Options Display List.
Up	Moves any options selected in the Header Options Display List up or down in the list. This changes the order they are displayed in WaveTool.
Down	

The options which can be displayed in each column header are:

Table 6.20.6.2-2 Column Header Options

Option	Description
ChipSet	Not supported on Magnum 1.
EbmEnable	Not supported on Magnum 1.
Index	The cycle number relative to the trigger event. This is useful when using zoom controls, to determine which cycles are being displayed.
MarAbsolute	Display the absolute MAR address of each cycle. Zero (0) is displayed when the selected test pattern (see Setup Acquire Execute Controls) is a pure logic pattern.
MarControl	Displays the MAR execution control used in each cycle. No value is displayed when the selected test pattern (see Setup Acquire Execute Controls) is a pure logic pattern.
MarLabel	Display the Pattern Label from the MAR pattern (if any) for each cycle. No value is displayed when the selected test pattern (see Setup Acquire Execute Controls) is a pure logic pattern.
MarPattern	Display name of the memory pattern controlling each cycle. No value is displayed when the selected test pattern (see Setup Acquire Execute Controls) is a pure logic pattern.

Table 6.20.6.2-2 Column Header Options *(Continued)*

Option	Description
MarRelative	Display the pattern-relative MAR address of each cycle (zero-based). -1 is displayed when the selected test pattern (see Setup Acquire Execute Controls) is a pure logic pattern.
PinScramble	Display the Pin Scramble (PS#) selection for each cycle. A value is displayed only if it changed from the prior cycle.
Time	The time relative to the trigger event.
TimeSet	Display the Time-set (TS#) selection for each cycle. A value is only displayed if it changed from the prior cycle.
VarAbsolute	Display the absolute VAR address of each cycle. Zero (0) is displayed when the selected test pattern (see Setup Acquire Execute Controls) is a pure memory pattern.
VarLabel	Display the Pattern Label from the VAR pattern (if any) for each cycle. No value is displayed when the selected test pattern (see Setup Acquire Execute Controls) is a pure memory pattern.
VarPattern	Display name of the logic pattern controlling each cycle. No value is displayed when the selected test pattern (see Setup Acquire Execute Controls) is a pure memory pattern.
VarRelative	Display the relative VAR address of each cycle (zero-based). -1 is displayed when the selected test pattern (see Setup Acquire Execute Controls) is a pure memory pattern.
VihhSet	Display the VIHH selection in each cycle.

6.20.6.3 Setup Acquire Dialog

See WaveTool.

This dialog is invoked by selecting the **Acquire** tab in the WaveTool Setup Controls .

The setup **Acquire** dialog is used to configure several WaveTool options:

- Select the source of information used to generate the display and choose the first and number of cycles to be displayed. See Setup Acquire Input Controls.

- Select the test pattern to be used to generate the display when the ReadHW button is clicked. Also determines the pattern executed and the pattern execution stop condition used when the Run button is clicked. See Setup Acquire Execute Controls and WaveTool Run Controls.
- Select various Logic Error Catch (LEC) options to be used when **Input->Source->Lec** is selected (see Setup Acquire Input Controls). Also see Setup Acquire LEC Controls.

This dialog is displayed by selecting the **Acquire** tab in the WaveTool Setup Controls dialog.

When WaveTool is first started, the system software configures the Setup Acquire dialog as shown below (LogicPat1 is a user test pattern):

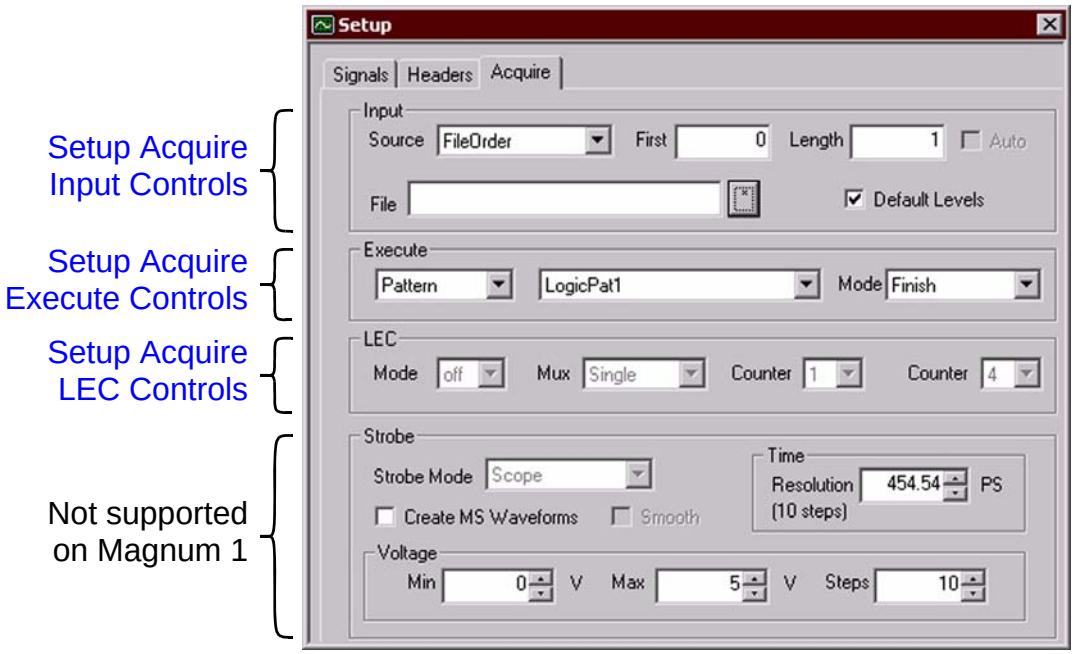


Figure-141: WaveTool Setup Acquire Dialog

Options are described further below.

6.20.6.4 Setup Acquire Input Controls

See Setup Acquire Dialog, WaveTool.

These controls are used to select the source of information used to generate the display and choose the first and number of cycles to be displayed:

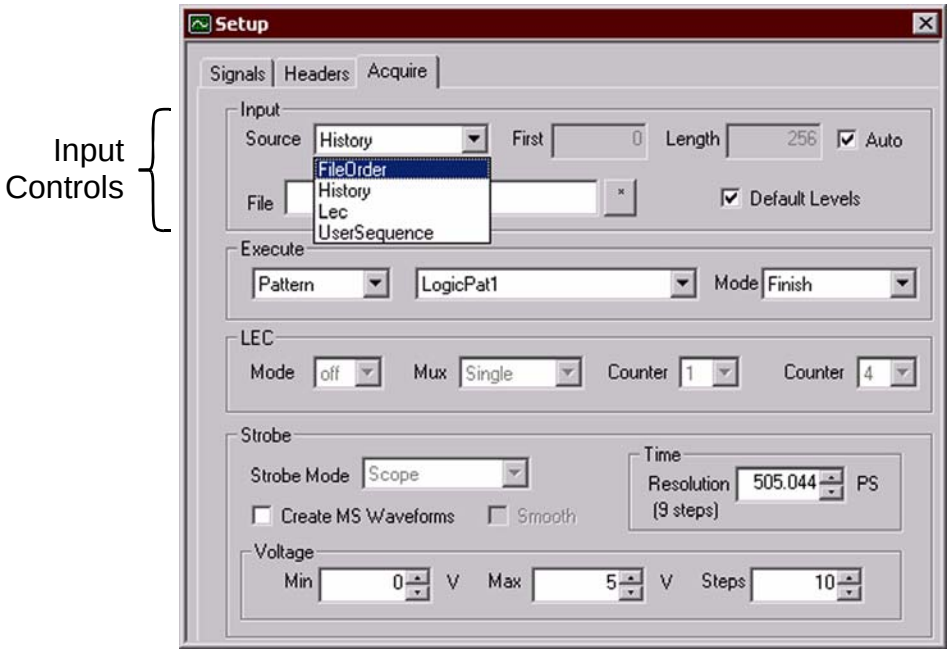


Figure-142: WaveTool Setup->Acquire Input Controls

The following table describes the controls in this portion of the dialog:

Table 6.20.6.4-1 Setup Acquire Input Controls

Control	Description
Source->FileOrder	The displayed sequence of cycles and related test pattern information is retrieved from the loaded test pattern selected using the `Pattern` option in the Setup Acquire Execute Controls. After the desired pattern is selected the `ReadHW` button must be clicked (the `Run` button has no effect) .

Table 6.20.6.4-1 Setup Acquire Input Controls *(Continued)*

Control	Description
Source->History	The displayed sequence of cycles and related test pattern information is retrieved from the History RAM. The History RAM contains up to 512 cycles of per-cycle information acquired during the most recent execution of a test pattern, including the actual sequence of cycles executed by conditional branch operations, etc.
Source->Lec	The displayed sequence of cycles and related test pattern information is retrieved from the Logic Error Catch (LEC). When this option is selected, the LEC configuration set via test program code, if any, is not used. Instead, when the Run button is clicked, before the selected test pattern (see Setup Acquire Execute Controls) is executed the LEC is configured by the system software, using `lec_config_set()` and `lec_mode_set()` with the values specified using the Setup Acquire LEC Controls. Note: any prior LEC or ECR configuration is over-written and not restored.
Source-UserSequence	The displayed sequence of cycles and related test pattern information is retrieved from a selected WaveTool execution trace file (**.etf*) stored on disk. See Creating WaveTool Trace Files. A standard file browser is presented initially pointing to the currently loaded test program's *\Debug* folder.
First	Specify the first cycle to be displayed. This is a zero-based value relative to the first cycle of the test pattern selected using the Setup Acquire Execute Controls. Disabled when **`Source->History`** is selected and **`Auto`** is enabled

Table 6.20.6.4-1 Setup Acquire Input Controls *(Continued)*

Control	Description
Length	Specifies the number of cycles to be displayed. The maximum value may be affected by the selected **Input Source** option. For example, if **Source->Lec** is currently selected and the Logic Error Catch (LEC) is configured to capture only failing cycles (see **Mode**) only 2 cycles will be displayed cycles when only 2 cycles fail. Disabled when **Source->History** is selected and **Auto** is enabled
Auto	History RAM. This control is only enabled when **Source->History** is selected. When **Auto** is enabled, the and
File	Enabled (usable) only when **Source-UserSequence** is selected, to allow the user to select an *execution trace file* (**.etf*) to be displayed. See Creating WaveTool Trace Files.
Default Levels	If enabled, the amplitude of the displayed WaveTool Timing Format Symbols is symbolic; i.e. not based on actual or user-programmed levels. If disabled, symbols are drawn using levels as currently configured in the hardware when the **ReadHW** or **Run** button is clicked.

6.20.6.5 Setup Acquire Execute Controls

See Setup Acquire Dialog, WaveTool.

These controls are used to select test pattern-related options used by WaveTool to configure the display the when the **ReadHW** or **Run** button is clicked.

Note: using these controls, when the pattern, sequence or test block selection is changed the contents of the History RAM and Logic Error Catch (LEC) are not updated until the **Run** button is clicked, which executes the selected pattern, sequence or test block. If the **ReadHW** button is used before the **Run** button is clicked WaveTool will display old/stale information. This applies when **Source->History** and **Source->Lec** are selected.

Note: using the **Run** button, useful test pattern operation (PASS/FAIL results, branch-on-error operations, etc.) depends on the current configuration of numerous DC and AC parameters which are programmed independent of the test pattern and independent of WaveTool. In general, when **`Source->History`** and **`Source->Lec`** will be useful only when the test pattern is stopped at a breakpoint and these other parameters are properly configured to execute the selected test pattern.

The following image shows the Acquire Pattern Controls in the Setup Acquire Dialog:

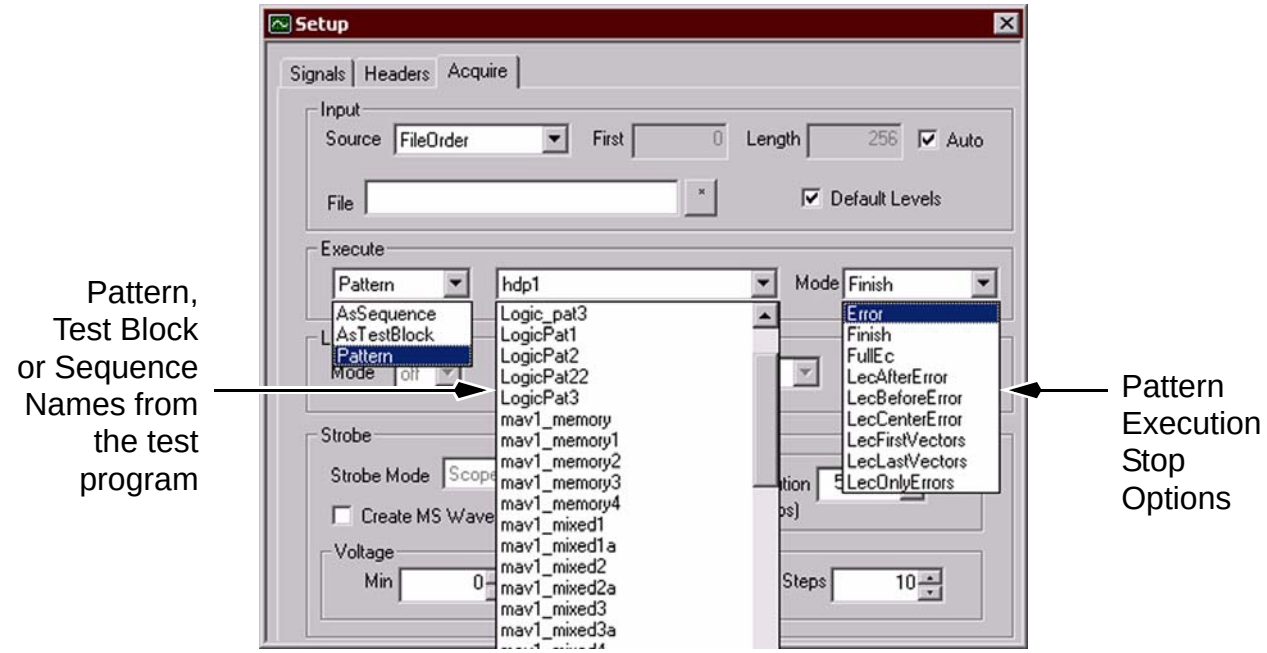


Figure-143: WaveTool Setup->Acquire Execute Controls

The following table describes the controls in this portion of the dialog:

Table 6.20.6.5-1 Setup Acquire Execute Controls

Control	Description
Pattern	The test pattern selected in the next menu is used when the **ReadHW** button is clicked or executed when the **Run** button is clicked. The **Mode** selection determines the pattern execution stop option used when the pattern is executed using the **Run** button. See `funtest()`. Ignored when **Source** = **Source->UserSequence**.
AsSequence	The sequence selected in the next menu is executed when the Run button is clicked. The **Mode** control is disabled. The display then depends on the selected **Source** option (see Note:): • **Source->FileOrder**: **AsSequence** is ignored. • **Source->History**: the contents of the History RAM at the end of sequence execution are used. • **Source->Lec**: the contents of the Logic Error Catch (LEC) at the end of sequence execution are used. • **Source->UserSequence**: **AsSequence** is ignored.
AsTestBlock	The test block selected in the next menu is executed when the Run button is clicked. The **Mode** control is disabled. The display then depends on the selected **Source** option (see Note:): • **Source->FileOrder**: **AsTestBlock** is ignored. • **Source->History**: the contents of the History RAM at the end of sequence execution are used. • **Source->Lec**: the contents of the Logic Error Catch (LEC) at the end of sequence execution are used. • **Source->UserSequence**: **AsTestBlock** is ignored.
Mode	Selects the pattern execution stop option to be used when the Run button is clicked. This is equivalent to the `Condition` argument to `funtest()`. Enabled only when **Pattern** is selected and **Source->History** or **Source->Lec** are used.

6.20.6.6 Setup Acquire LEC Controls

See See Setup Acquire Dialog, WaveTool.

These controls are enabled only when **`Source->Lec`** is selected, to control how the Logic Error Catch (LEC) is configured when the **Run** button is clicked.

When selected, the current LEC configuration, if any, is not used. Instead, when the **Run** button is clicked, before the test pattern selected using Setup Acquire Execute Controls is executed the LEC is configured by the system software, using `lec_config_set()` and `lec_mode_set()` with the values specified using the LEC controls in this dialog.

Note: any prior LEC or ECR configuration is over-written and not restored.

LEC Configuration and Mode Options

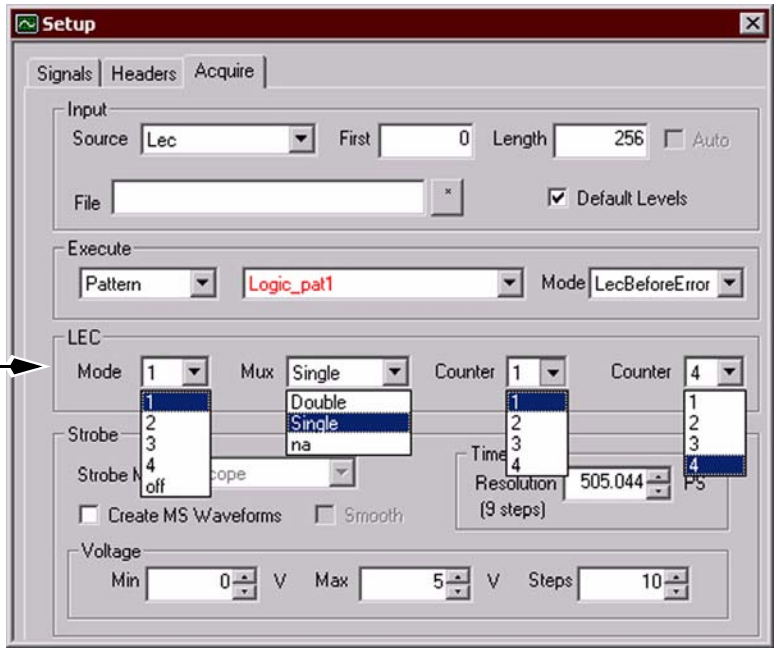


Figure-144: WaveTool Setup->Acquire LEC Controls

The following table describes the controls in this portion of the dialog:

Table 6.20.6.6-1 Setup Acquire LEC Controls

Control	Description
Mode	This determines the Logic Error Catch (LEC) mode. See Setup Acquire LEC Mode Options below and `lec_mode_set()` for more details.
MUX	This determines how the Pin Scramble MUX is configured. Select `single` when executing a non-DDR test pattern, use `single` when executing a Double Data Rate (DDR) Mode test pattern. See `fail_signal_mux()` for more details.
Counter	Select the first counter displayed when **Mode** = 2 or 3 or the only counter displayed when **Mode** = 4. See `lec_mode_set()` for more details.
Counter	Select the second counter displayed when **Mode** = 2 or 3. See `lec_mode_set()` for more details.

The following table describes the legal LEC modes:

Table 6.20.6.6-2 Setup Acquire LEC Mode Options

Mode	Equivalent lec_mode_set() Argument	LEC Capture
1	`t_lec_mode_1`	64 pin P/F states (per ECR), 32 bit VAR, 32 bit SAR. Default.
2	`t_lec_mode_2`	64 pin P/F states (per ECR), 28 bit VAR or SAR, 1 bit VAR/SAR flag, one 32 bit counter value, one 18 bit counter value.

Table 6.20.6.6-2 Setup Acquire LEC Mode Options *(Continued)*

Mode	Equivalent lec_mode_set() Argument	LEC Capture
3	t_lec_mode_3	64 pin P/F states (per ECR), 28 bit VAR or SAR; 1 bit VAR/SAR flag, four 12 bit counter values.
4	t_lec_mode_4	64 pin P/F states (per ECR), 28 bit VAR, 28 bit SAR, one 20 bit counter value.
VAR = Vector Address. SAR = Scan Vector Address.		

6.20.7 WaveTool Run Controls

See WaveTool.

WaveTool's Run Controls consist of two buttons labeled **Run** and **ReadHW**. These are used

to retrieve and update the information displayed in WaveTool, as follows:

Table 6.20.7.0-1 WaveTool Run & ReadHW Button Description

Button	Description
Run	Operation depends on the **Setup->Acquire->Input->Source** option currently selected in the Setup Acquire Input Controls of the Setup Acquire Dialog: • If **Source->History** or **Source->Lec** is selected, `funtest()` is executed with the test pattern currently selected in the Setup Acquire Execute Controls. The pattern execution stop option used is that selected in the **Mode** menu of the Setup Acquire Execute Controls. The first cycle and number of cycles displayed is determined by the First and Length values in the Setup Acquire Input Controls (disabled when **Source->History** is selected and **Auto** is enabled). If **Source->Lec** is selected, failing strobes are shown in RED (see WaveTool Timing Format Symbols). • If **Source->FileOrder** or **Source-UserSequence** is selected the **Run** button operates as above but information displayed doesn't change because the test results are not used in these two modes. Note: useful test pattern operation, which can affect which cycles are executed and the per-pin PASS/FAIL results, depends on the current hardware configuration at the time the **Run** button is clicked. This includes DPS voltages, pin levels, timing, etc. Thus, the **Run** button will typically be useful only when the test program is paused a breakpoint, at which these parameters are properly configured.

Table 6.20.7.0-1 WaveTool Run & ReadHW Button Description *(Continued)*

Button	Description
ReadHW	Operation depends on the **Setup->Acquire->Input->Source** option currently selected using the Setup Acquire Input Controls: • If **Source->History** is selected the **ReadHW** button reads and displays information from the History RAM, which contains information acquired during the most recent pattern execution. See Note: • If **Source->Lec** is selected the **ReadHW** button reads and displays the contents of the Logic Error Catch (LEC), which contains information acquired during the most recent pattern execution. This option is supported only when a pure logic pattern (i.e. not Memory or Mixed) is executed. See Note: • If **Source->FileOrder** is selected, the cycles displayed in WaveTool are determined by reading (from pattern memory) the test pattern selected using the Setup Acquire Execute Controls. The first cycle and number of cycles displayed is determined by the **First** and **Length** controls in the Setup Acquire Input Controls. The information displayed does not represent actual cycles executed but, rather, cycles from a test pattern source-view perspective. • If **Source-UserSequence** is selected, the cycles displayed in WaveTool are determined by reading a previously saved WaveTool trace file from disk. See Creating WaveTool Trace Files. The file is selected using the **File** selector in the Setup Acquire Input Controls. The first cycle and number of cycles displayed is determined by the **First** and **Length** controls in the Setup Acquire Input Controls.

Note: when **Source->History** or **Source->Lec** are enabled the contents of the History RAM and Logic Error Catch (LEC) are not automatically updated when the selected test pattern, sequence or test block is changed (see Setup Acquire Execute Controls). The user must use the **Run** button to update the hardware any time this selection is changed. Failure to do so will cause old/stale information to be read from the Logic Error Catch (LEC) or History RAM and displayed in WaveTool.

6.20.8 WaveTool Timing Format Symbols

See WaveTool.

The following table shows the various symbols used to display timing edges and formats for *single pins* (see WaveTool Display). When a PinList is displayed in WaveTool (see Setup Signals Dialog) the format symbols drawn in each cycle are the composite of all formats used on all pins in the PinList. This can be confusing to interpret, more below. By default, the amplitude of these symbols is based on voltages defined by Nextest, but the currently programmed hardware levels may also be used, see Default Levels control:

Table 6.20.8.0-1 Timing Format Symbols

Symbol	Format	Comments
	NRZ Data = 1	Format: Non-return to Zero The image shown here shows a transition from an unknown prior state or tri-state to pattern data = 1. See WaveTool Drive Waveform Images.
	NRZ Data = 0	Format: Non-return to Zero The image shown here shows a transition from an unknown prior state or tri-state to pattern data = 0. See WaveTool Drive Waveform Images.
	RTZ Data = 1	Format: Return to Zero The image shown here shows a transition from an unknown prior state or tri-state to pattern data = 1. See WaveTool Drive Waveform Images.

Table 6.20.8.0-1 Timing Format Symbols *(Continued)*

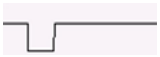
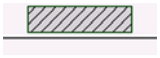
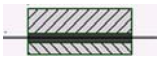
Symbol	Format	Comments
	RTO Data = 0	Format: Return to One The image shown here shows a transition from an unknown prior state or tri-state to pattern data = 0. See WaveTool Drive Waveform Images..
	RTC Data = 1	Format: Return to Complement The image shown here shows a transition from a prior state = 0 to pattern data = 1. See WaveTool Drive Waveform Images.
	RTC Data = 0	Format: Return to Complement The image shown here shows a transition from a prior state = 1 to pattern data = 0. See WaveTool Drive Waveform Images.
	Double Clock Pos Data = 1	Format: Positive Double Clock The image shown here shows a transition from a prior state = 0 to pattern data = 1. See WaveTool Double Clock Drive Waveform Images.
	Double Clock Neg Data = 0	Format: Negative Double Clock The image shown here shows a transition from a prior state = 1 to pattern data = 0. See WaveTool Double Clock Drive Waveform Images.

Table 6.20.8.0-1 Timing Format Symbols *(Continued)*

Symbol	Format	Comments
	Window Strobe Data = 1 (H)	The window strobe graphic symbol has width equivalent to the time between the two strobe edges. An edge strobe is a vertical line, positioned at the time of the strobe edge. Strobe-1 (H) shows the strobe graphic symbol above the reference line, which represents VOH. Strobe-0 (L) shows the symbol below the reference line, which represents VOL. Strobe-V (valid) shows the symbol both above and below the reference, representing both above VOH and below VOL, with a darker region in the center which represents the invalid (fail) range between VOH and VOL. Strobe-Z (tri-state) uses a small symbol, very near the reference line, which represents the region between VOH/VOL; i.e. tri-state.
	Window Strobe Data = 0 (L)	
	Edge Strobe Data = 1 (H)	
	Edge Strobe Data = 0 (L)	
	Window Strobe Data = n/a (V)	
	Edge Strobe Data = n/a (V)	
	Window Strobe Data = n/a (Z) (tri-state)	
	Edge Strobe Data = n/a (Z) (tri-state)	

When displaying drive waveforms, WaveTool will show an additional (possibly 4th) logic level in those cycles in which a non-default VIHH Map is selected, on pins which are included in the selected VIHH Map. The logic levels are:

- VIL, when the drive waveform is driving low
- VIH, when the drive waveform is driving high
- VZ or VTT, when the pin is tri-stated
- VIHH, when the pin is included in the non-default VIHH Map selection

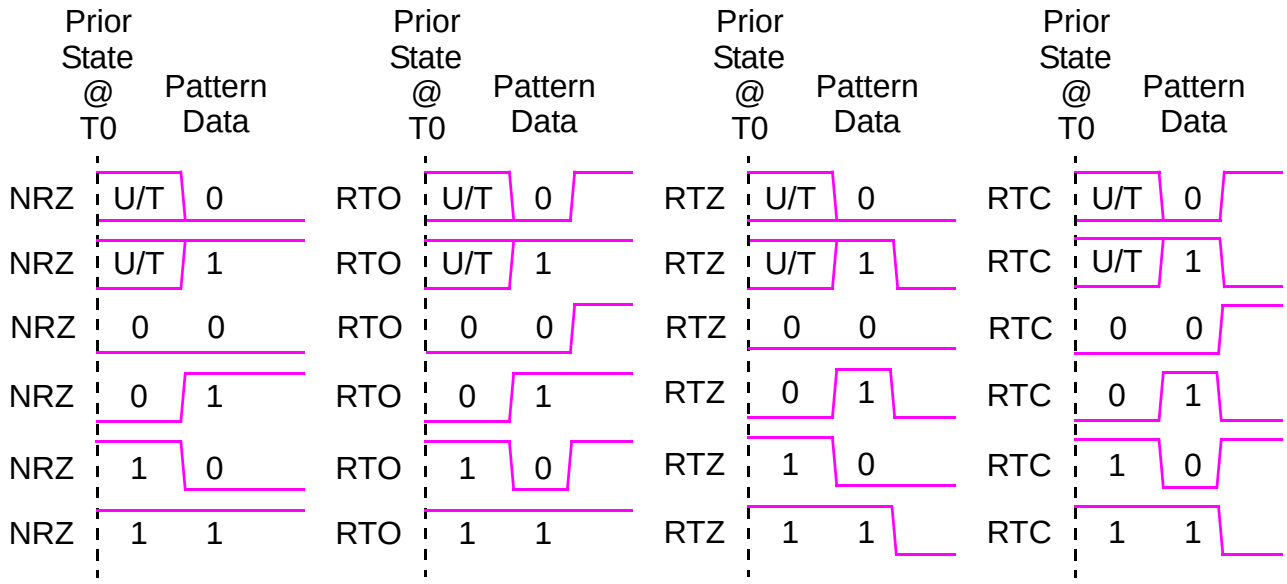
For example:

Third logic level = VIHH Level

This waveform can easily be confused

When WaveTool draws a drive waveform format, the actual image seen will depend on both the format used in a given cycle (as determined by the time-set selection in each cycle) and

the drive state from the previous cycle, if any. Four prior states are possible: unknown, tri-state, drive-1 and drive-0. The following diagrams shows the various waveforms drawn in one cycle, for each combination of drive format vs. each prior state vs. pattern data:



U/T = unknown prior state (U) or tri-state (T). 0/1 = pattern data.

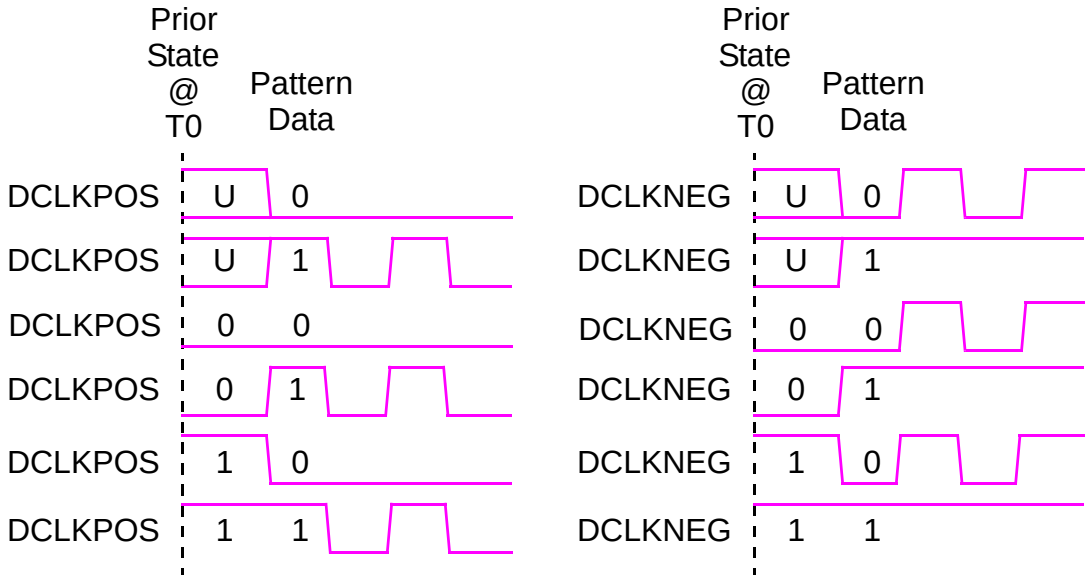
I/O drive edges are shown at T0 (0nS) which is the default, see Note:.

Actual format edge timing is determined by the user's program.

Only the lines shown in here in magenta are actually displayed in WaveTool; i.e. the `U`, `0` and `1` characters are not displayed.

Figure-145: WaveTool Drive Waveform Images

Double clock formats are shown in the table below:



U = unknown prior state (U). 0/1 = pattern drive data.

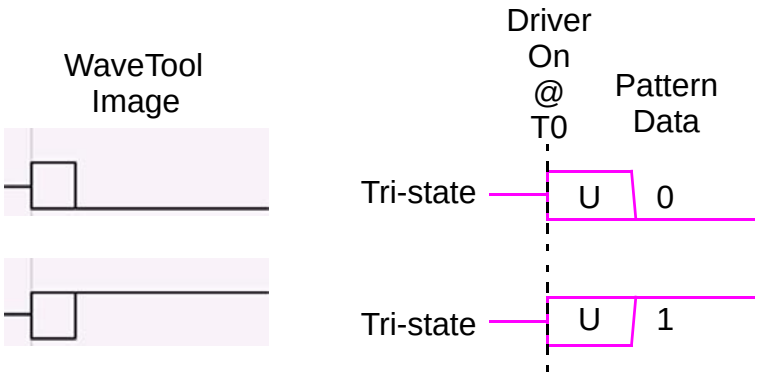
Double clock waveforms are special in that any pins using these formats cannot be tri-stated, strobed, or drive other formats. Thus, the unknown prior state region (U) only applies to the first cycle of the pattern, when the prior state is not known.

Actual format edge timing is determined by the user's program.

Only the lines shown in here in magenta are actually displayed in WaveTool; i.e. the `U`, `0` and `1` characters are not displayed.

Figure-146: WaveTool Double Clock Drive Waveform Images

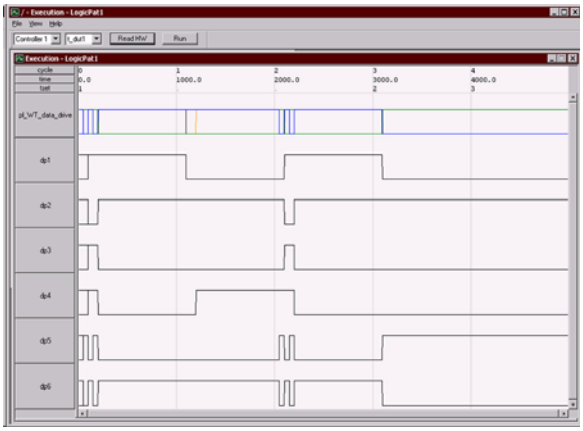
Note: in cycles containing a transition from tri-state to drive (1 or 0), if the first drive edge occurs later than the I/O drive edge, WaveTool will draw an unknown prior state (U below) during that time. In the example below, the I/O drive edge is shown at T0 (0nS) which is the default I/O timing:



This is significant because it differs from actual hardware operation: at the time of the tri-state-to-drive (I/O) transition, the hardware will drive the logic state from the last cycle in which the driver was not tri-stated. WaveTool does not know this prior drive state and thus can't draw it correctly (and the software needed to resolve this is not practical).

As indicated above, when a PinList is displayed in WaveTool (see Setup Signals Dialog) the format symbols drawn in each cycle are the composite of all formats used on all pins in the PinList (in the cycle). This can be confusing to interpret. For example:

This image displays 6 individual pins (`dp1` through `dp6`) which have only drive formats assigned in the cycles shown. At the top, the PinList named `pl_WT_data_drive` shows the composite of those pins/formats.



This image displays 6 different individual pins (`dp7` through `dp12`) which have only strobe formats assigned in the cycles shown. At the top, the PinList named `pl_WT_data_strobe` shows the composite of those pins/formats.

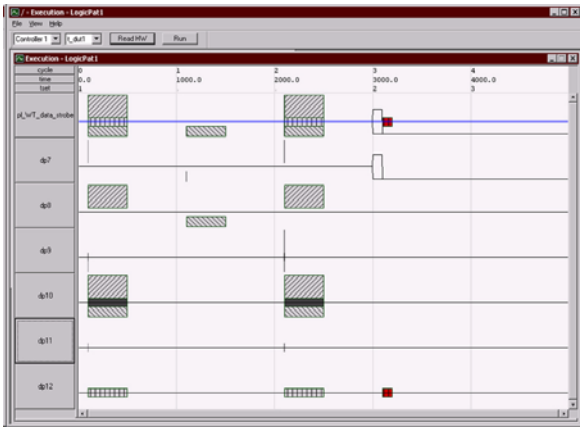


Figure-147: WaveTool PinList Composite Symbol Examples

6.20.9 WaveTool Color Schemes

See WaveTool Tool-bar Options Control Options.

WaveTool has 3 color schemes which affect the window background color and the color of the lines used to draw waveforms in the WaveTool Display. The scheme may be changed by selecting **`Options->Scheme`** using WaveTool Tool-bar Options Control Options.

The images below show the available color schemes:

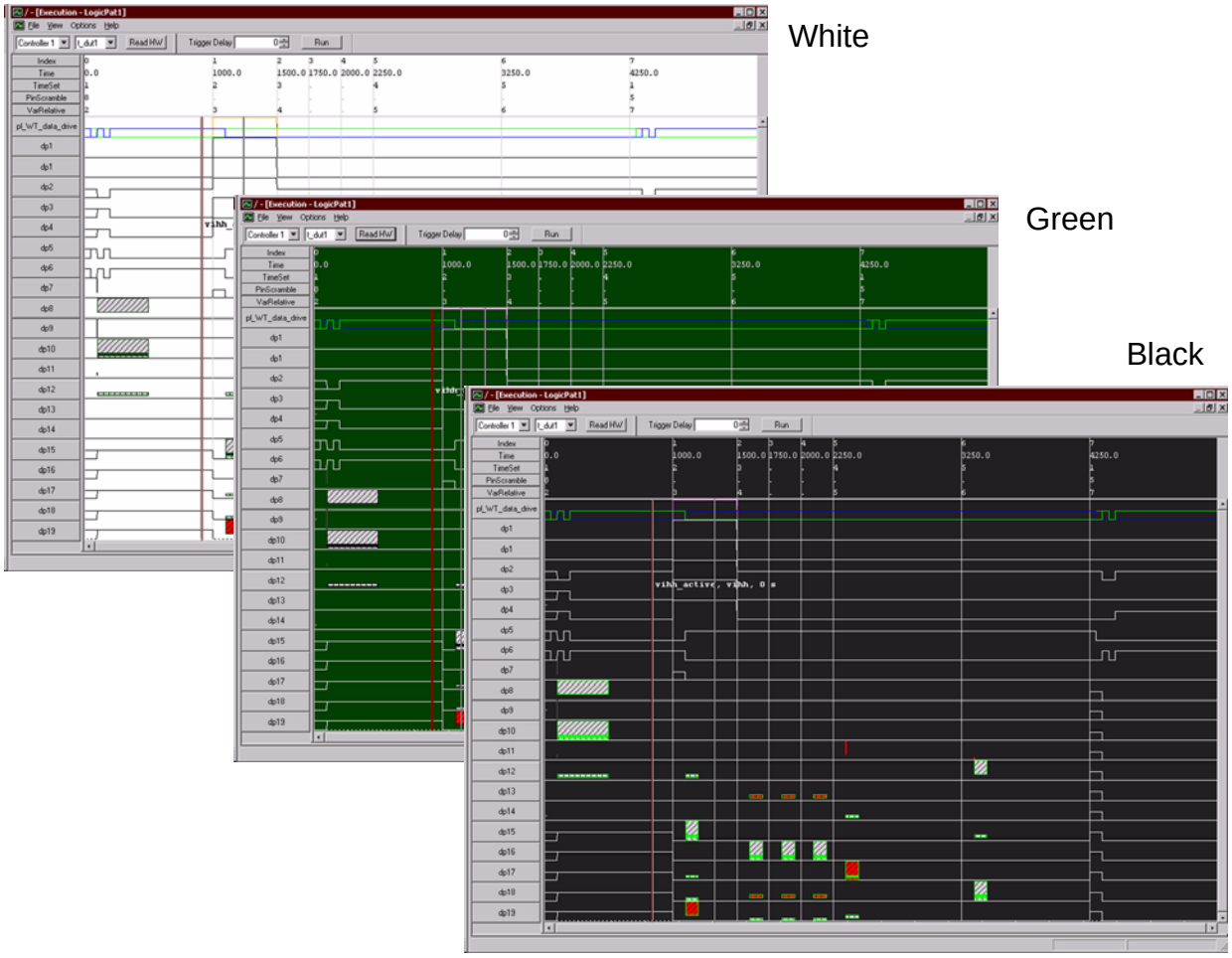


Figure-148: WaveTool Color Scheme Examples

Note the following:

- By default, the `white` color scheme is used.
- Changing the scheme selection value has no effect until one of the WaveTool Run Controls is invoked.

6.20.10 WaveTool Zoom Controls

See WaveTool.

It is possible to zoom-in on a region of the Main Waveform Display Window. This is done using a left-mouse click-hold-drag operation. The location of the initial left-mouse click draws a vertical line in the Main Waveform Display Window. This identifies the location of the left border of the zoomed display. The location of the right border is determined by holding the left-mouse button and dragging the cursor to the right. As this is done, a second vertical line is drawn, which determines the right side of the display if/when the mouse button is released:

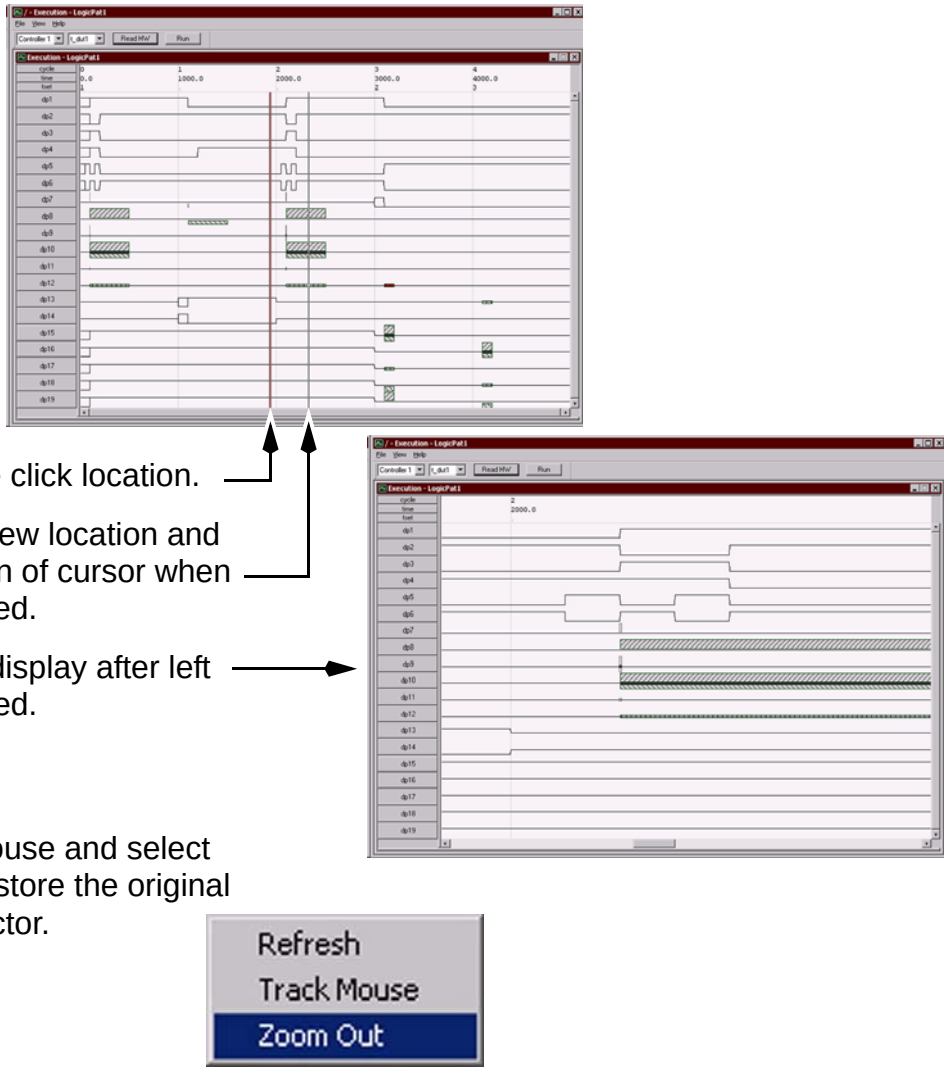


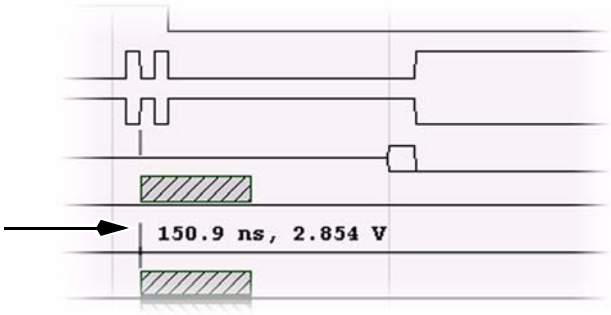
Figure-149: WaveTool Zoom Controls

6.20.11 WaveTool Mouse Track Controls

See WaveTool.

Enabling the Track Mouse option (as shown below) causes additional information to be displayed as the mouse is moved about the Main Waveform Display Window. Clicking the right-mouse button in the Main Waveform Display Window invokes the following menu:

With Track Mouse enabled, the initial value displayed = cursor position in cycle: time vs. voltage .



After a short delay, the displayed value changes to show the format, edge type and edge time of nearest transition,

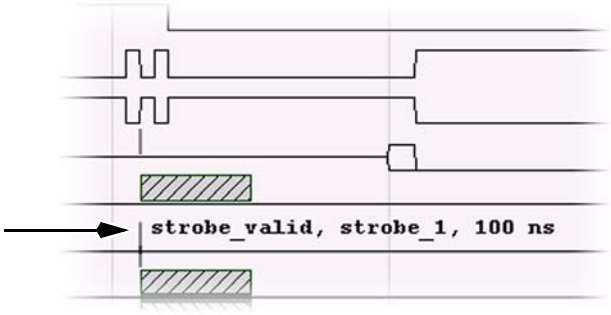


Figure-150: WaveTool Track Mouse Controls

The timing information seen in the upper waveform display reflects the location of the cursor within the current test cycle. However, if the mouse remains motionless for approximately 2 seconds, the display changes, as shown in the lower waveform display, to indicate the format, edge type and edge time of the transition closest to the cursor.

6.20.12 Creating WaveTool Trace Files

See WaveTool Tool-bar Controls.

Below is an example of the information displayed when **Help->Creating Trace Files**

is selected. Note that the specific information displayed may change as features are added or refined; i.e. use the image below as an example but use the **Help->Creating Trace Files** button in WaveTool to get the latest version:

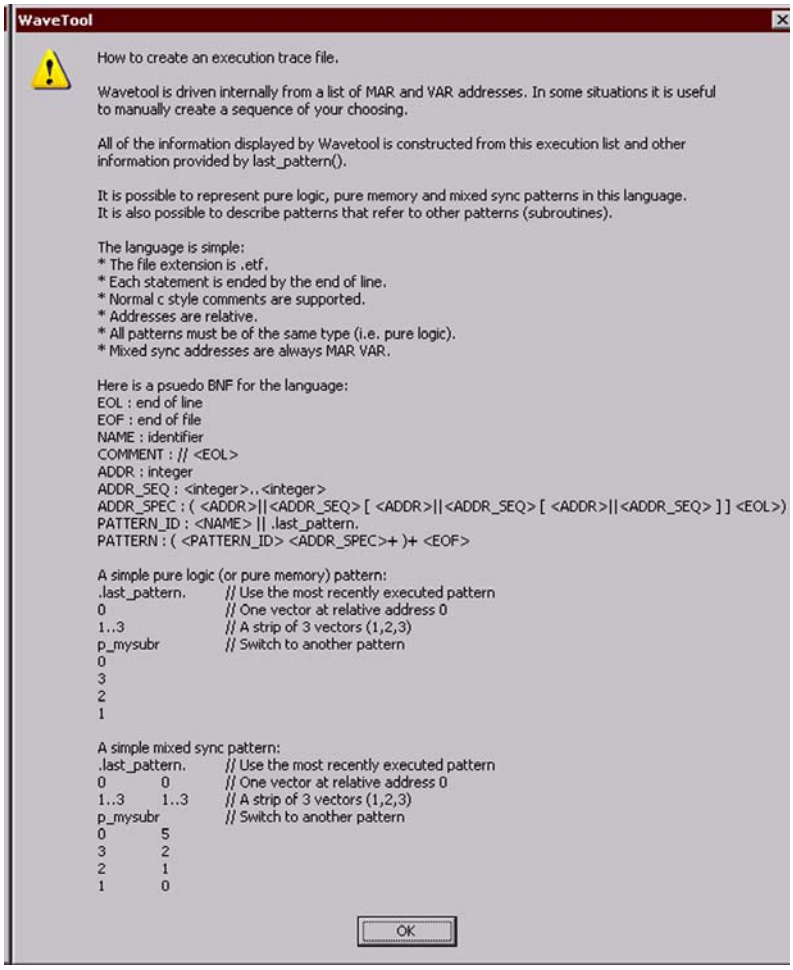


Figure-151: WaveTool Trace File Creation Information

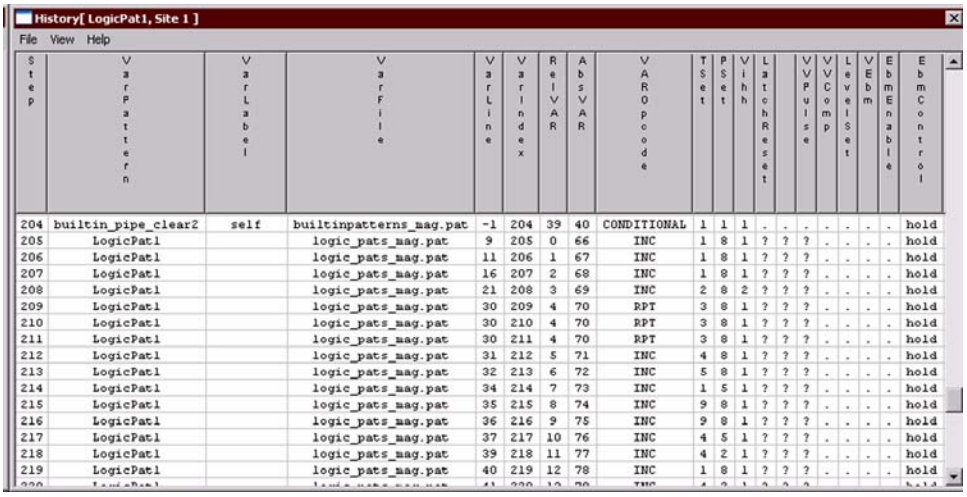
6.20.13 History RAM

See Overview, WaveTool.

The History RAM is specialized hardware which records up to 512 cycles of key information about the sequence of executed test pattern instructions. WaveTool reads and displays information from the History RAM when **Source->History** is selected using the Setup

Acquire Input Controls. Additional information can be displayed using WaveTool's History RAM Display, invoked using **File->New->History Window** (see WaveTool Tool-bar File Control Options).

The image below is a typical History RAM Display for a pure logic pattern. Note that the first cycles displayed are from built-in test pattern instructions (required to setup the hardware) executed prior to the first cycle of the user's test pattern:



Step	VarPattern	VarLabel	VarFile	VarLine	VarIndex	RelVAR	AbsVAR	VAROpcode	TSet	PSet	VIhh	LatchReset		VPulse	VComp	LevelSet	VEbm	EbmEnable	EbmControl
204	builtin_pipe_clear2	self	builtinpatterns_mag.pat	-1	204	39	40	CONDITIONAL	1	1	1	.	.	.	.	.	.	.	hold
205	LogicPat1		logic_pats_mag.pat	9	205	0	66	INC	1	8	1	?	?	?	.	.	.	.	hold
206	LogicPat1		logic_pats_mag.pat	11	206	1	67	INC	1	8	1	?	?	?	.	.	.	.	hold
207	LogicPat1		logic_pats_mag.pat	16	207	2	68	INC	1	8	1	?	?	?	.	.	.	.	hold
208	LogicPat1		logic_pats_mag.pat	21	208	3	69	INC	2	8	2	?	?	?	.	.	.	.	hold
209	LogicPat1		logic_pats_mag.pat	30	209	4	70	RPT	3	8	1	?	?	?	.	.	.	.	hold
210	LogicPat1		logic_pats_mag.pat	30	210	4	70	RPT	3	8	1	?	?	?	.	.	.	.	hold
211	LogicPat1		logic_pats_mag.pat	30	211	4	70	RPT	3	8	1	?	?	?	.	.	.	.	hold
212	LogicPat1		logic_pats_mag.pat	31	212	5	71	INC	4	8	1	?	?	?	.	.	.	.	hold
213	LogicPat1		logic_pats_mag.pat	32	213	6	72	INC	5	8	1	?	?	?	.	.	.	.	hold
214	LogicPat1		logic_pats_mag.pat	34	214	7	73	INC	1	5	1	?	?	?	.	.	.	.	hold
215	LogicPat1		logic_pats_mag.pat	35	215	8	74	INC	9	8	1	?	?	?	.	.	.	.	hold
216	LogicPat1		logic_pats_mag.pat	36	216	9	75	INC	9	8	1	?	?	?	.	.	.	.	hold
217	LogicPat1		logic_pats_mag.pat	37	217	10	76	INC	4	5	1	?	?	?	.	.	.	.	hold
218	LogicPat1		logic_pats_mag.pat	39	218	11	77	INC	4	2	1	?	?	?	.	.	.	.	hold
219	LogicPat1		logic_pats_mag.pat	40	219	12	78	INC	1	8	1	?	?	?	.	.	.	.	hold

Figure-152: History RAM Display

Also note:

- The contents of the History RAM are updated any time a functional test pattern is executed.
- The Magnum 1 History RAM always contains the last 512 cycles executed.

6.21 WafermapTool

6.21.1 Overview

WaferMapTool provides the basic features needed to create, display and interact with a wafer map.

WaferMapTool support includes both manual controls and software used to:

- Configure WaferMapTool (see WaferMapTool Configuration). This can be done several ways:
  - Manually, using WaferMapTool Configuration.
  - Programmatically using `wmap_set()`.
  - By loading a Configuration File, either manually or using `wmap_set()`.
- Send per-die test result data to WaferMapTool for display, using `wmap_die_set()`. This causes WaferMapTool to display, for a specified die location, one of:
  - A specified bin color
  - A specified bin name
  - An arbitrary text string
  - A specified bitmap (image)
- Clear WaferMapTool's Main Map and Bin Counts Table, either manually or using `wmap_set()`.
- Execute user code (via call-back function) when a die is selected in the Main Map i.e. clicked using the left-mouse button. The X/Y die coordinates of the selected die are passed to the call-back function. The `wmap_onclick_set()` function is used to register/un-register the user's call-back function.
- Save the current WaferMapTool configuration to a disk file, either manually or using `wmap_set()`.
- Save the WaferMapTool contents to a disk file i.e. the Main Map, Bin Counts Table and configuration, either manually or using `wmap_set()`.
- Display a wafer map previously saved to a disk file, either manually or using `wmap_set()`.

- Create a visual outline (die field) around multiple die. This is useful when testing multiple die in parallel, to indicate which die are being tested at any given time. See Die Field Display.

Key topics covered in this section include:

- WafermapTool Communication Architecture
- Starting WaferMapTool
- WafermapTool Persistence
- WaferMapTool Configuration
  - Configuration File
- User Interface & Controls
- Die Attributes
  - Die Display Options
  - Marked Die
- WaferMapTool Software
  - Types, Enums, etc.
  - `wmap_set(), wmap_get()`
  - WafermapTool File Access Rules
  - `wmap_die_set(), wmap_die_get()`
  - `wmap_cmd_start(), wmap_cmd_end()`
  - `wmap_die_cmd_start(), wmap_die_cmd_end()`
  - `wmap_onclick_set()`
- WafermapTool Die-Bitmap Support
  - Dynamically Defined Monochromatic Images
  - Dynamically Defined Color Images
  - Statically Defined Images
  - UI BitmapTool Images
- Die Field Display

6.21.2 WafermapTool Communication Architecture

See Overview.

The overall architecture is as follows:

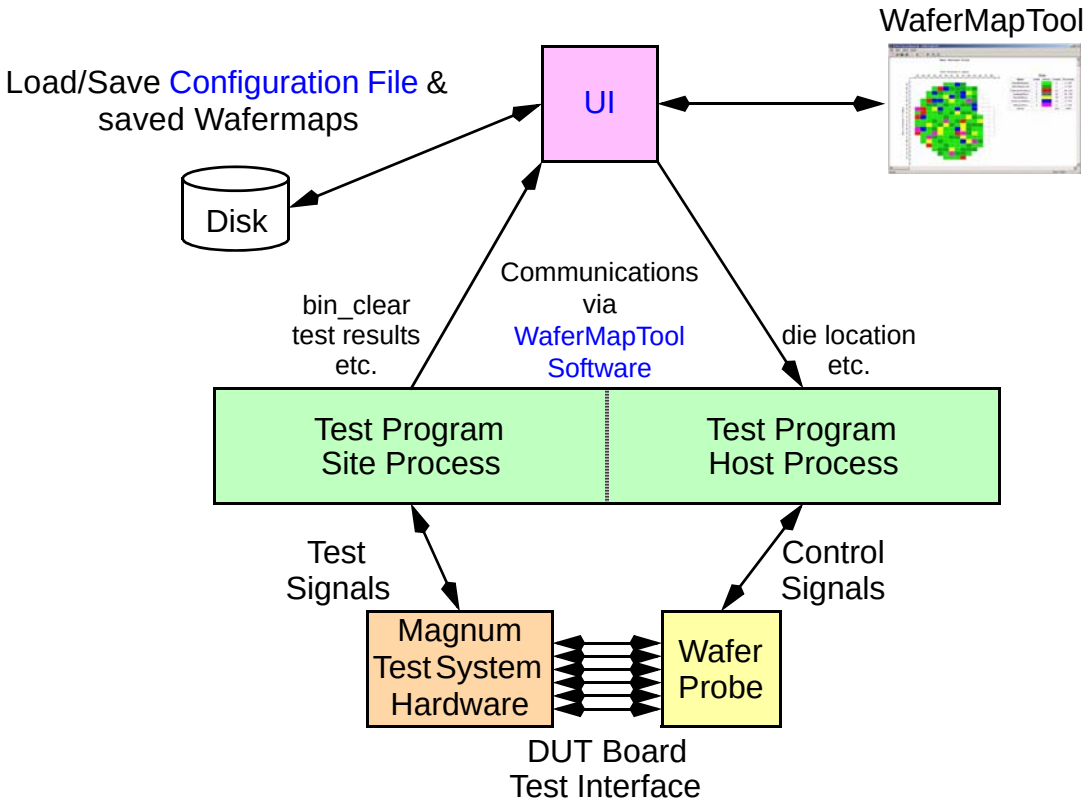


Figure-153: WaferMapTool Communication Architecture

Note the following:

- WaferMapTool can be started any time after UI is loaded. See Starting WaferMapTool.

- Before WaferMapTool can display useful information it must be configured. See WaferMapTool Configuration:

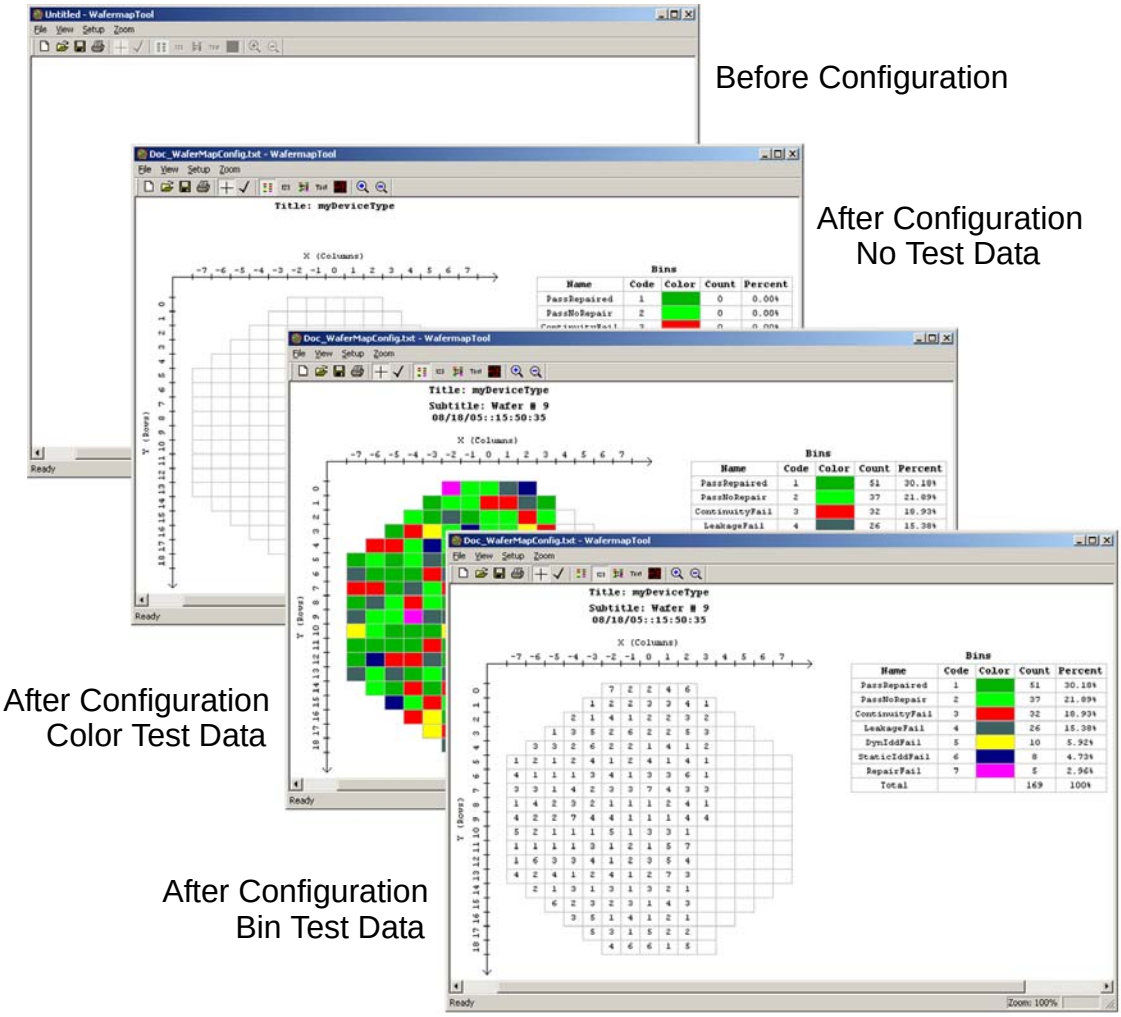


Figure-154: WaferMapTool Display

The WaferMapTool Configuration defines, for example, the number and location of each die, the size of each die, which WaferMapTool display axis is treated as the X or Y axis, which color and numerical code is mapped to each test bin, etc. Note that loading a previously saved wafer map also loads the configuration saved with that wafer map.

- Once WaferMapTool is configured, it will display information sent by the test program using the appropriate WaferMapTool Software functions. In general, the process is as follows:

  - At the start of each wafer, `wmap_set()` sends `wmap_bin_clear` to clear WaferMapTool's Main Map and Bin Counts Table.
  - For each execution of the Sequence & Binning Table, `wmap_die_set()` sends `wmap_die_bin` for the die tested, This is typically done in an After-testing Block. The WaferMapTool Configuration determines how color and bin codes are mapped to each test bin, by name.
- If the user clicks a die in the Main Map, that die is marked (see Marked Die). If a call-back function is registered (see `wmap_onclick_set()`) that function is executed in the Host process (only), receiving the coordinates of selected die. This is targeted at directing the prober to move to the selected die; user code is required.
- Various interactive controls are available, to manipulate and save the current configuration, print the wafer map, save/load a configuration or wafer map, etc. See User Interface & Controls.

6.21.3 Starting WaferMapTool

See Overview.

WaferMapTool is started from UI's Tool menu (**`Tool->Wafermap...`**) or by clicking on the WaferMapTool icon in UI's Tool Bar: 

WafermapTool can also be started by test program code by loading a Configuration File using `wmap_set()` with the `wmap_config_load` option.

WaferMapTool does not display the Main Map or Bin Counts Table until a WaferMapTool Configuration is done.

6.21.4 WafermapTool Persistence

Note: first available in software release h1.1.23.

The following WafermapTool attributes are recorded when UI terminates (normally). Later, when WafermapTool is started again, it is automatically configured with these recorded values. This is commonly known as attribute persistence:

- WafermapTool's window size and location
- The currently selected view option
- The state of the cursor view selection
- The current zoom factor
- The state of the marked die enable
- Initial tool visibility (more below)

Like the other UI tools, WafermapTool's display paradigm is somewhat different than other Windows applications. UI's tools will always be in one of the following states:

- Not started: the tool process is not running
- Started: the tool process is running. The tool is in one of the following states:
  - Tool is visible
  - Tool is visible but minimized; it is seen in the taskbar
  - Tool is hidden; i.e. not displayed and not seen in the taskbar

Prior to software release h1.1.23, the latter state could only be entered by first starting WafermapTool and then clicking the cancel button (the **X** in the upper-right corner of the tool). As indicated, this does not terminate the WafermapTool process. Instead, it hides the display; the tool does not appear in the task bar nor on the display. Clicking the WafermapTool button makes it visible again. This operation allows the tool to be hidden without losing any information.

When UI is terminated, the persistence facility records whether WafermapTool is hidden. This does not affect WafermapTool's operation when it is subsequently started using UI's Tool menu or the WafermapTool button, but it does affect whether WafermapTool is visible when subsequently started by test program code. Normally, any time WafermapTool is started by executing one of the `wmap_xxx()` functions (`wmap_set()`, `wmap_die_set()`, etc.), it will be visible (and not minimized). Now, WafermapTool's persistent states determines whether it is visible or hidden when started by these functions. In other words, when the persistent state is hidden, if WafermapTool is started by executing one of the `wmap_xxx()` functions the tool will be started but remain hidden.

Note: WafermapTool's visibility persistence is only recorded if the WafermapTool process is running when UI is terminated.

6.21.5 WaferMapTool Configuration

See Overview, See Configuration File.

Before WaferMapTool can display test results it must be configured, to identify die size, locations, axis orientation and labels, title, etc.

Several methods are available:

- Load an existing Configuration File, either manually using **File->Open** and selecting the file, by name, or...
- Load an existing Configuration File via user code using `wmap_set()` with the `wmap_config_load` option.
- Use controls in WafermapTool to interactively define or modify (and save) the configuration. This can be somewhat slow, but can be useful when creating an initial configuration for a given wafer topology.

If the test program attempts to send test result data to WaferMapTool before a configuration is loaded (or to an invalid die location) an error similar to the following will be displayed:

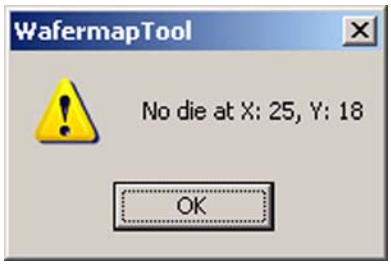


Figure-155: WaferMapTool Error: Sending Data Before Configuration

6.21.5.1 Configuration File

See WaferMapTool Configuration.

This section describes the format of the configuration file. Note the following:

- The configuration file is ASCII, and can be manually edited using any text editor.
- UI requires the file name to include the *.txt* extension i.e. *myConfig.txt*.
- Each line in the configuration file must consist of one of the following:

- Comment. Only C++ style comments are supported (i.e. // comment). Note that any comments manually added using a text editor WILL BE LOST if the configuration is saved, using WaferMapTool's **File->Save** or **File->Save As...** controls.
- Blank line (ignored)
- `keyword : value[, value][, value][, value]` (more below)

As indicated the colon delimits the keyword from one or more comma separated values

- The only required keyword is `version`, plus a corresponding version number. This must be the first line in the configuration file which is not blank or a comment. When editing an existing configuration file, the version value must not be modified. The version is automatically added to a configuration file when saving the file to disk, using WaferMapTool's **File->Save** or **File->Save As...** controls.
- All keywords are case-insensitive

The following table describes each supported keyword (listed alphabetically) with related usage information. Additional information about some keywords is covered after the table:

Table 6.21.5.1-1 WaferMapTool Configuration File Keywords

Keyword	Legal Values	Purpose
axis_horizontal	X Y	Specifies whether the Main Map horizontal display axis represents the wafer/die X-axis or Y-axis. Only one of axis_horizontal or axis_vertical are needed (last one wins). This selection affects how values specified using die_by_x, die_by_y, die_by_xy, die_x_size, die_y_size, legend_x, and legend_y are actually applied.
axis_horizontal_inc	RightToLeft LeftToRight	Specify the direction that die test results are updated (added) in the horizontal axis.
axis_vertical	X Y	Specifies whether the Main Map vertical display axis represents the wafer/die X-axis or Y-axis. Only one of axis_horizontal or axis_vertical are needed (last one wins). This selection affects how values specified using die_by_x, die_by_y, die_by_xy, die_x_size, die_y_size, legend_x, and legend_y are actually applied.
axis_vertical_inc	TopToBottom BottomToTop	Specifies the direction that die test results are updated (added) in the vertical axis.
bin_code	Two values More below	Specifies a string (typically a bin number) mapped to one bin name. This is text displayed when the specified bin is mapped to a given die. using wmap_die_set(). See Die Display Options.

Table 6.21.5.1-1 WaferMapTool Configuration File Keywords *(Continued)*

Keyword	Legal Values	Purpose
bin_color	Four values More below	Specifies an RGB color mapped to one bin name. This is the color displayed when the specified bin is mapped to a given die. using wmap_die_set(). See Die Display Options.
die_by_x	Three values More below	Adds one or more die location(s) to the Main Map, in the X-axis. Affected by axis_horizontal and axis_vertical.
die_by_y	Three values More below	Adds one or more die location(s) to the Main Map, in the Y-axis. Affected by axis_horizontal and axis_vertical.
die_by_xy	Two values More below	Adds one die location to the Main Map. Affected by axis_horizontal and axis_vertical.
die_size_unit	Micron Mil	Specifies the unit of value for both die_x_size and die_y_size. Mil = .001”
die_x_size	+Integer	Specifies die size in the X-axis, in die_size_units. Affected by axis_horizontal and axis_vertical.
die_y_size	+Integer	Specifies die size in the Y-axis, in die_size_units. Affected by axis_horizontal and axis_vertical.

Table 6.21.5.1-1 WaferMapTool Configuration File Keywords *(Continued)*

Keyword	Legal Values	Purpose
legend_x	String	Specifies the user-defined X-axis title (see WaferMapTool with Test Data). All characters after colon to end-of-line are displayed. Quotes are not required and are displayed if used. Display location is affected by axis_horizontal and axis_vertical.
legend_y	String	Specifies the user-defined Y-axis title (see WaferMapTool with Test Data). All characters after colon to end-of-line are displayed. Do not quote string. Display location is affected by axis_horizontal and axis_vertical.
title	String	Specifies the user-defined title (see WaferMapTool with Test Data). All characters after colon to end-of-line are displayed.
version	Auto-generated. Do not edit	Required by system software. Automatically added to the Configuration File file by **File->Save** and **File->Save As...**. Do not edit.

The following keywords require additional usage details:

bin_code specifies the code mapped to one bin name. Multiple bin_code statements are typically used, one for each bin name. bin_code requires 2 values:

```
bin_code : bin_name_string, code_string
```

where:

bin_name_string identifies the name of the bin. This must match one of the bin names being sent to WafermapTool using wmap_die_set() in the test program.

code_string identifies the code mapped to the **bin_name_string**. This will typically represent the hardware bin number corresponding to the bin name, but can be any string value.

bin_color specifies the RGB color mapped to one bin name. Multiple bin_color statements are typically used, one for each bin name. bin_color requires 4 values:

```
bin_color : bin_name_string, R, G, B
```

where:

bin_name_string identifies the name of the bin. This must match one of the bin names being sent to WafermapTool using wmap_die_set().

R, **G**, and **B** are each a value of 0-255, representing the amount of each color to be applied. Also see WaferMapTool Bin Colors Dialog.

die_by_x and die_by_y add one or more die location(s) to the Main Map, in the X-axis or Y-axis. Multiple die_by_x or die_by_y statements may be used. Duplicate dies are silently ignored. These keywords require 3 values:

```
die_by_x : Xlocation, Ystart, Ystop
die_by_y : Ylocation, Xstart, Xstop
```

where:

Xlocation and **Ylocation** identify the reference X or Y location in the Main Map. One or more die will be added in the X-axis or Y-axis starting at this location. Negative die location values are OK. Whether the X or Y axis is horizontal or vertical depends upon axis_horizontal or axis_horizontal.

Ystart and **Xstart** identifies the starting Y or X location in the Main Map i.e. the location of the first die to be added. Negative die location values are OK. Whether the X or Y axis is horizontal or vertical depends upon axis_horizontal or axis_horizontal.

Ystop and **Xstop** identify the location of the last die to be added in the specified axis. Negative die location values are OK.

die_by_xy adds one die location to the Main Map. Multiple die_by_xy statements may be used. Duplicate dies are silently ignored. die_by_xy requires 2 values:

```
die_by_xy : Xlocation, Ylocation
```

where:

Xlocation identifies the location of the die being added in the X-axis. Whether the X or Y axis is horizontal or vertical depends upon axis_horizontal or axis_horizontal.

Ylocation identifies the location of the die being added in the Y-axis. Whether the X or Y axis is horizontal or vertical depends upon axis_horizontal or axis_horizontal. Negative die location values are OK.

Example

The following example consists of two parts:

- Example Configuration File
- Example Test Program Code

Example Configuration File

Note: some auto-generated comments were deleted below to improve readability.

```
// WaferMapTool Configuration File
// ------------------------------

// All key words are case-insensitive
// only C++ style comments are supported.

version : 1// Must be the first line which is not blank or a comment

////////////////// Display Headings ////////////////////////
// Each value an arbitrary user-defined string
// Quotes not required, displayed if used.
title : "User defined Title"
legend_x : User Defined X Label
legend_y : User Defined Y Label

/////////////// Main Map Axis Orientation /////////////////
// One of the next two lines is optional, last one wins
// Winner affects die_by_x, die_by_y, die_by_xy, legend_y,
// legend_x, die_x_size, die_y_size
axis_horizontal : X
axis_vertical : Y

/////////////// Main Map Axis Motion /////////////////
// Specifies direction dies are updated
axis_horizontal_inc : LeftToRight
axis_vertical_inc : BottomToTop

////////////////// Die Size ////////////////////////
die_size_unit : Micron// Options: Micron or Mil (.001")
die_x_size : 7812
die_y_size : 5625

////////////////// Die Locations ////////////////////
// Options:
//   die_by_x : Xlocation, Ystart, Ystop
//   die_by_y : Ylocation, Xstart, Xstop
```

```
//   die_by_xy : Xlocation, Ylocation
// All are affected by axis_horizontal/axis_vertical
die_by_x : 25, 18, 24
die_by_x : 26, 16, 26
die_by_x : 27, 15, 28
die_by_x : 28, 14, 29
die_by_x : 29, 13, 29
die_by_x : 30, 12, 30
die_by_x : 31, 12, 30
die_by_x : 32, 12, 30
die_by_x : 33, 12, 30
die_by_x : 34, 13, 29
die_by_x : 35, 14, 29
die_by_x : 36, 15, 28
die_by_x : 37, 16, 26
die_by_x : 38, 18, 24

////////////////// Test Bin Map ////////////////////
// Maps bin_string sent via wmap_die_set() to code value
//   bin_code : bin_name_string, code_string
bin_code : PassNoRepair,   1
bin_code : PassRepaired,   2
bin_code : ContinuityFail, 3
bin_code : LeakageFail,    4
bin_code : DynIddFail,     5
bin_code : StaticIddFail,  6
bin_code : RepairFail,     7

// Maps bin_string sent via wmap_die_set() to RGB color
//   bin_color : bin_name_string, R, G, B
bin_color : PassNoRepair,     0, 255,   0
bin_color : PassRepaired,     0, 180,   0
bin_color : ContinuityFail, 255,   0,   0
bin_color : LeakageFail,     64, 128, 128
bin_color : DynIddFail,     255, 255,   0
bin_color : StaticIddFail,    0,   0, 128
bin_color : RepairFail,     255,   0, 255
```

Example Test Program Code

The following test program code matches the Example Configuration File. In particular, the Bin names must match in both locations.

```
CString binID;
AFTER_TESTING_BLOCK( ATB ) { // See AFTER_TESTING_BLOCK()
   BOOL ok = wmap_die_set( wmap_die_bin, x, y, bin_string );
   if(! ok )
      output("  ERROR: wmap_die_set:wmap_die_bin = %s @ %d/%d",
         bin_string, die_x, die_y);
}
TEST_BIN( ContinuityFail ) { binID = resource_name( current_test_bin );
} // See TEST_BIN()
TEST_BIN( LeakageFail    ) { binID = resource_name( current_test_bin ); }
TEST_BIN( DynIddFail     ) { binID = resource_name( current_test_bin ); }
TEST_BIN( StaticIddFail  ) { binID = resource_name( current_test_bin ); }
TEST_BIN( PassNoRepair   ) { binID = resource_name( current_test_bin ); }
TEST_BIN( PassRepaired   ) { binID = resource_name( current_test_bin ); }
TEST_BIN( RepairFail     ) { binID = resource_name( current_test_bin ); }
```

Note that the body code of each `TEST_BIN()` macro can be modified as needed, provided `binID` is correctly set to the name of the bin.

6.21.6 User Interface & Controls

See Overview.

WaferMapTool's controls provide the following general capabilities:

- Load or save a WaferMapTool Configuration
- Interactively create or modify the current configuration
- Save a wafer map or load a previously saved wafer map.
- Clear test results displayed in the Main Map and Bin Counts Table.

The following image shows the WaferMapTool after being configured and after displaying test data for most of a wafer:

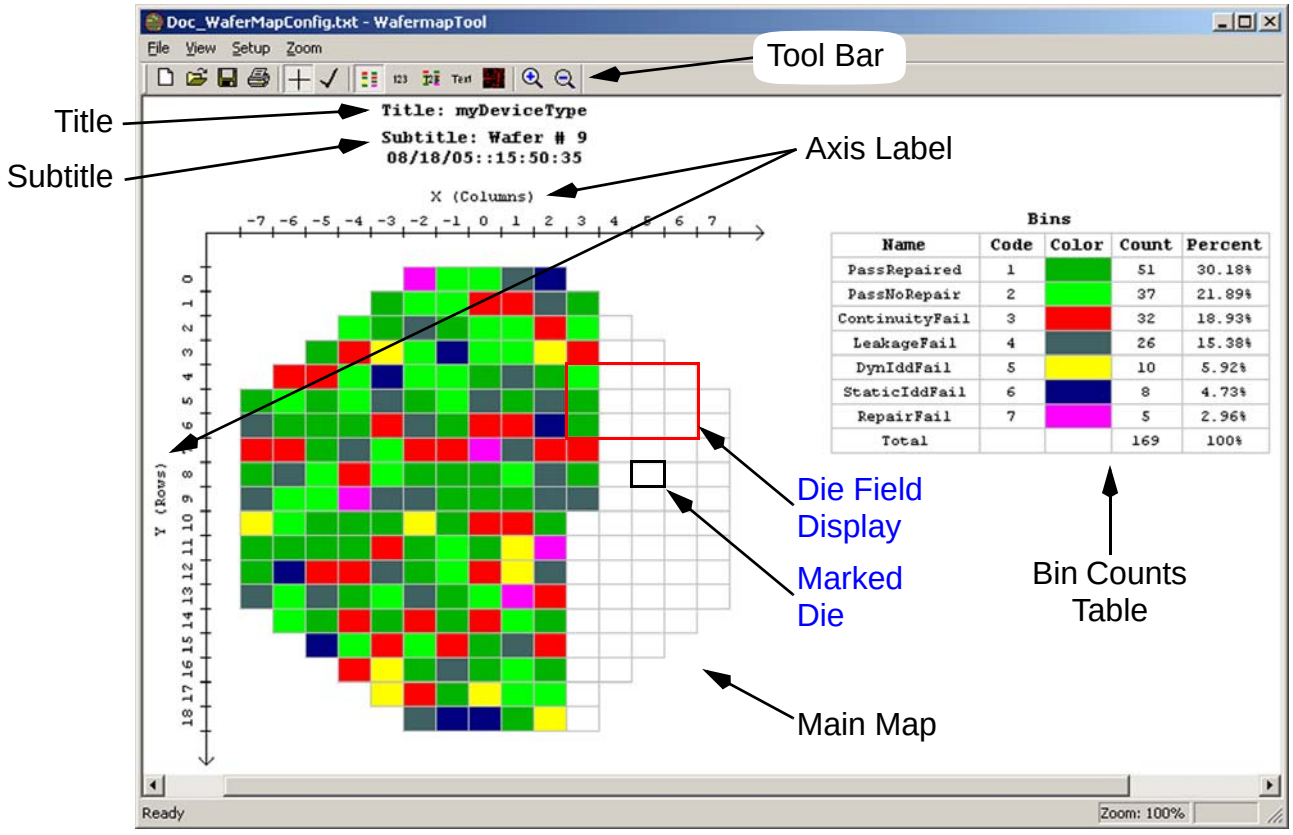


Figure-156: WaferMapTool with Test Data

The following images expand WaferMapTool's main menu to show the available sub-menus and dialogs. These are described on later pages:

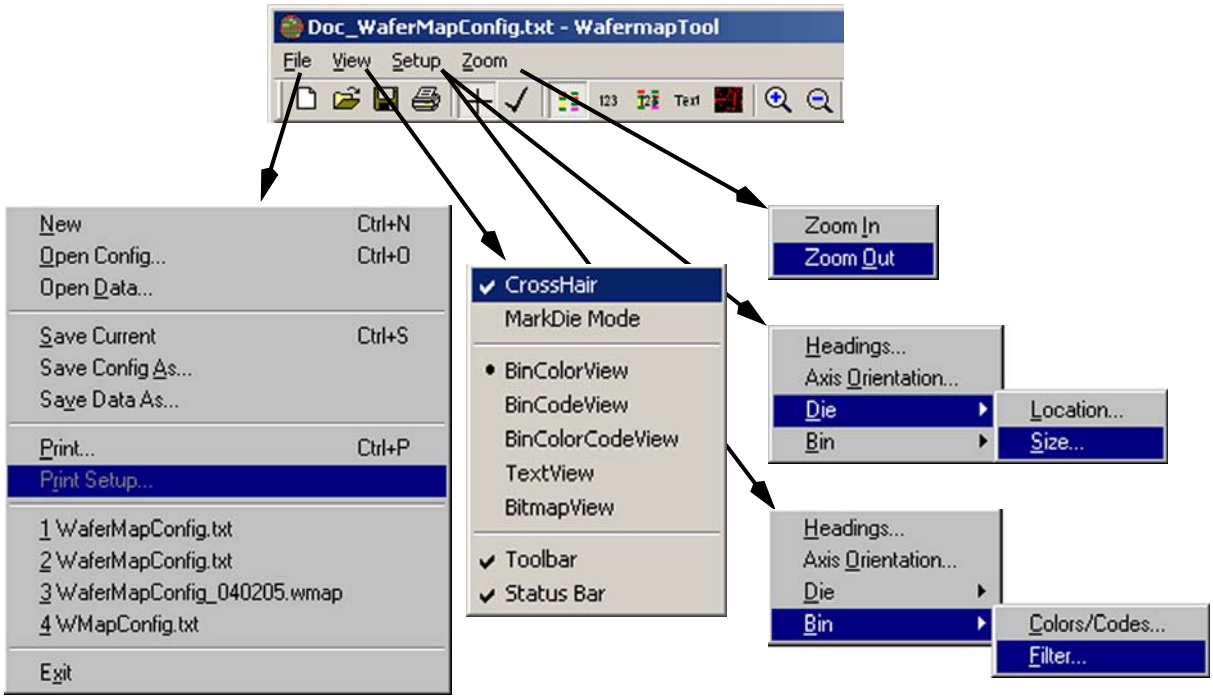


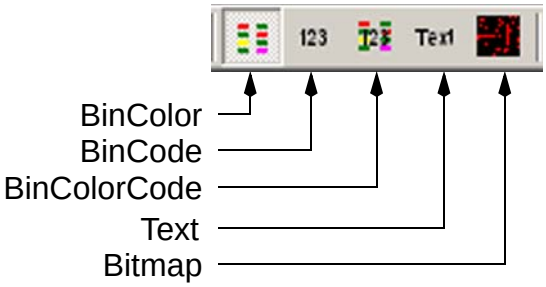
Figure-157: WaferMapTool Main Menu Options

WaferMapTool's **File** menu contains the following controls:

Control	Purpose
New	Create a new configuration (see WaferMapTool Configuration). Clears existing configuration, Main Map and Bin Counts Table.
Open Config	Open an existing WaferMapTool Configuration file. Clears existing configuration, Main Map and Bin Counts Table. The standard file browser is presented. See WafermapTool File Access Rules.
Open Data	Open and display a wafer map previously saved using **Save Data As**. Replaces the existing WaferMapTool Configuration, Main Map and Bin Counts Table. The standard file browser is presented. See WafermapTool File Access Rules.

Control	Purpose
Save Current	Save **EITHER** the current configuration or the current wafer map into the file last opened using one of: - **File->Open Config** - **File->Open Data** - **File->Save Config As...** - **File->Save Data As...** - wmap_set() with wmap_config_load - wmap_set() with wmap_config_save - wmap_set() with wmap_data_load - wmap_set() with wmap_data_save The file name displayed in WafermapTool's title bar indicates both the file being written and whether a configuration is being saved (**.txt* file) or a wafer map is being saved (**.wmp* file). See WafermapTool File Access Rules.
Save Config As...	Save the current configuration into a specified WaferMapTool Configuration file. The standard file browser is presented. See WafermapTool File Access Rules.
Save Data As...	Save the current wafer map into a specified file. This saves the current WaferMapTool Configuration, Main Map and Bin Counts Table into a file which can subsequently be displayed in WafermapTool using **Open Data**. The standard file browser is presented. See WafermapTool File Access Rules.
Print	Print the current Main Map and Bin Counts Table.
1 *file name* **2** *file name* **3** *file name* **4** *file name*	Short-cut selection of a file to load. Both Configuration Files (**.txt*) and wafer map data files (**.wmp* file) may be displayed. The selection determines the type of file loaded. Note that loading either type replaces the current WaferMapTool Configuration, if any.
Exit	Close WafermapTool.

WaferMapTool's **View** menu contains the following controls:

Control	Purpose
CrossHair	Enables/disables the crosshair display. This is also selectable using the crosshair button: 
MarkDie Mode	Enables/disables the crosshair display. This is also selectable using the crosshair button: 
BinColorView **BinCodeView** **BinColorCodeView** **TextView** **BitmapView**	Selects which display mode is used in the Main Map. This selection is also possible using the buttons shown here: 
ToolBar	Enables/disables WMapTool's tool bar.
StatusBar	Enables/disables WMapTool's status bar.

The **Setup->Heading** dialog can be used to interactively define or modify 3 user-defined labels seen in the WaferMapTool display:

Corresponding Configuration File entries:

- `title`
- `legend_x`
- `legend_y`

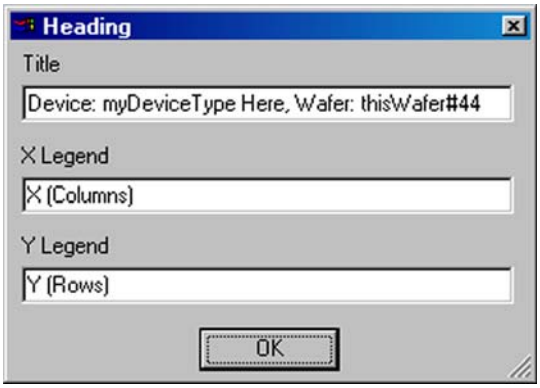


Figure-158: WaferMapTool Setup Headings Dialog

These heading can also be accessed from the test program using `wmap_set()`.

The **Setup->Axis Orientation...** dialog can be used to interactively define or modify WaferMapTool axis increment and axis orientation parameters:

Corresponding Configuration File entries:

- axis_horizontal_inc
- axis_vertical_inc
- axis_horizontal
- axis_vertical

The latter 2 options define the same parameter (last one wins).

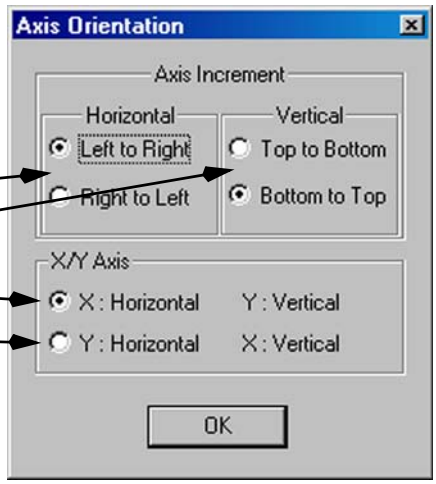


Figure-159: WaferMapTool Setup Axis Orientation Dialog

These parameters can also be accessed from the test program using wmap_set().

The **Setup->Die->Location...** dialog can be used to interactively define or modify die locations:

Corresponding Configuration File entries:

- die_by_x
- die_by_y
- die_by_xy

Values entered are affected by axis_horizontal and axis_vertical.

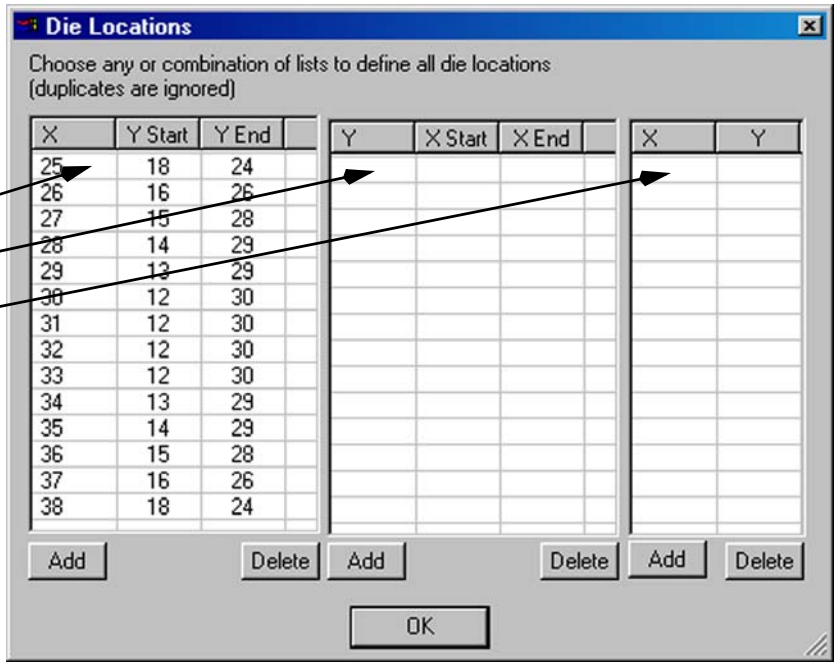


Figure-160: WaferMapTool Setup Die Locations Dialog

These parameters can also be accessed from the test program using `wmap_set()`.

The **`Setup->Die->Size...`** dialog can be used to interactively define or modify die size:

Corresponding Configuration File entries:

- `die_size_unit`
- `die_x_size`
- `die_y_size`

The bottom 2 values are affected by `axis_horizontal` and `axis_vertical`.

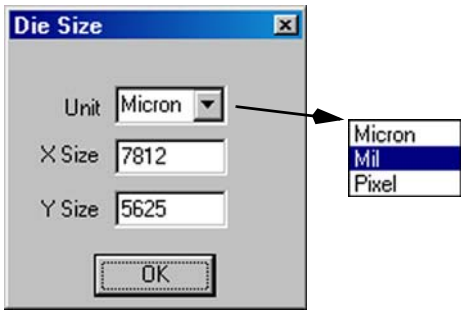


Figure-161: WaferMapTool Setup Die Size Dialog

These parameters can also be accessed from the test program using `wmap_set()`.

The **`Setup->Bin->Mapping...`** dialog can be used to interactively define or modify which color and/or code is mapped to each bin name. Bin names can be added or deleted:

Corresponding Configuration File entries:

- Bin Names
- `bin_code`
- `bin_color`

To edit individual table cells, double-left-click in the target cell.

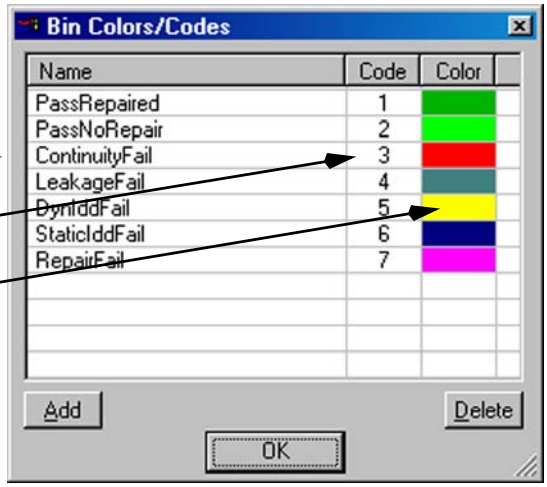


Figure-162: WaferMapTool Setup Bin Colors/Codes Dialog

These parameters can also be accessed from the test program using `wmap_set()`.

When editing a color value the following dialog is displayed and used to choose the desired color:

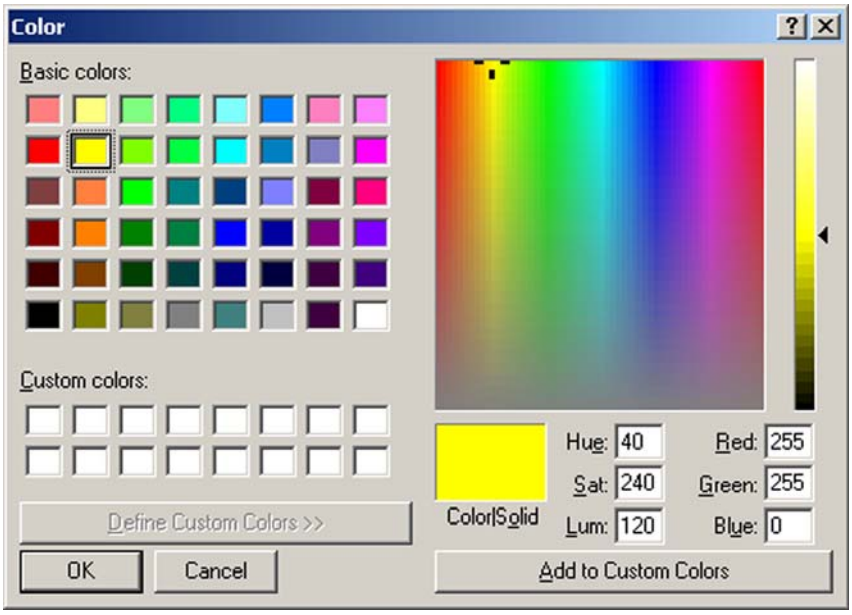


Figure-163: WaferMapTool Bin Colors Dialog

The **Setup->Bin->Filter...** dialog can be used to interactively select which bin(s) are displayed in the Main Map. This makes it easy, for example, to see only pass bins, etc.

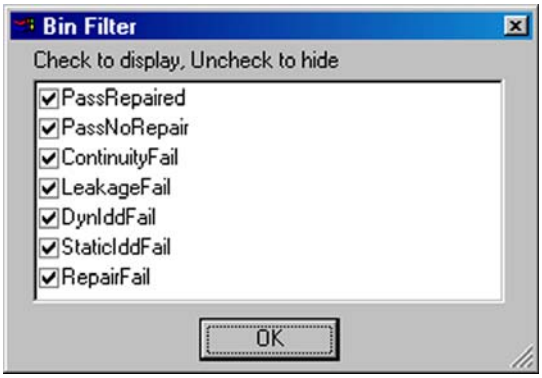


Figure-164: WaferMapTool Bin Filter Dialog

6.21.7 Die Attributes

See Overview.

After WafermapTool has been started and configured (see WaferMapTool Configuration) the Main Map displays a grid, with each cell representing one die.

Initially (i.e. before the test program sends any test results to WafermapTool) all die will be blank. During program execution, test program code will use `wmap_die_set()` to send test result data for one or more die. Which data is actually displayed is controlled, for the whole wafer using Die Display Options.

Using the left-mouse button, the user may select a die in the Main Map. This can cause the die to be marked or unmarked (see Marked Die) and can also invoke a user-written call-back function, if previously registered using `wmap_onclick_set()`.

6.21.7.1 Die Display Options

See Die Attributes, Overview.

During program execution, test program code will use `wmap_die_set()` to send test result data to each die in WafermapTool's Main Map. This determines, for example, whether a

given die turns green or red. However, color is only one of several die display options, which include the following:

Option	wmap_die_set() Type Argument	Displayed in Mode	Operation
Bin	wmap_die_bin	Bin Code View	The wmap_die_bin value sent via wmap_die_set() is displayed in the specified die. See Note below.
Color	wmap_die_bin	Bin Color View	If the wmap_die_bin value sent via wmap_die_set() exactly matches a bin name specified in the WaferMapTool Configuration the color mapped to that bin is displayed, otherwise the die remains empty (white). See Note below.
Color & Bin	wmap_die_bin	Bin Color-Code View	Combination of Bin Color View and Bin Code View.
Text	wmap_die_text	Text View	The wmap_die_text value sent via wmap_die_set() is displayed in the specified die. It may be necessary to zoom IN to see the entire string displayed for a given die.
Bitmap	wmap_die_bitmap	Bitmap View	The wmap_die_bitmap sent via wmap_die_set() is displayed in the specified die. See WafermapTool Die-Bitmap Support.
Note: the die attributes displayed for a given die in the Bin Code View, Bin Color View and Bin Color-Code View depend upon the same wmap_die_bin value sent via wmap_die_set(). However, the WaferMapTool Configuration determines the color displayed for a given wmap_die_bin value.			

The following image shows the bin color view:

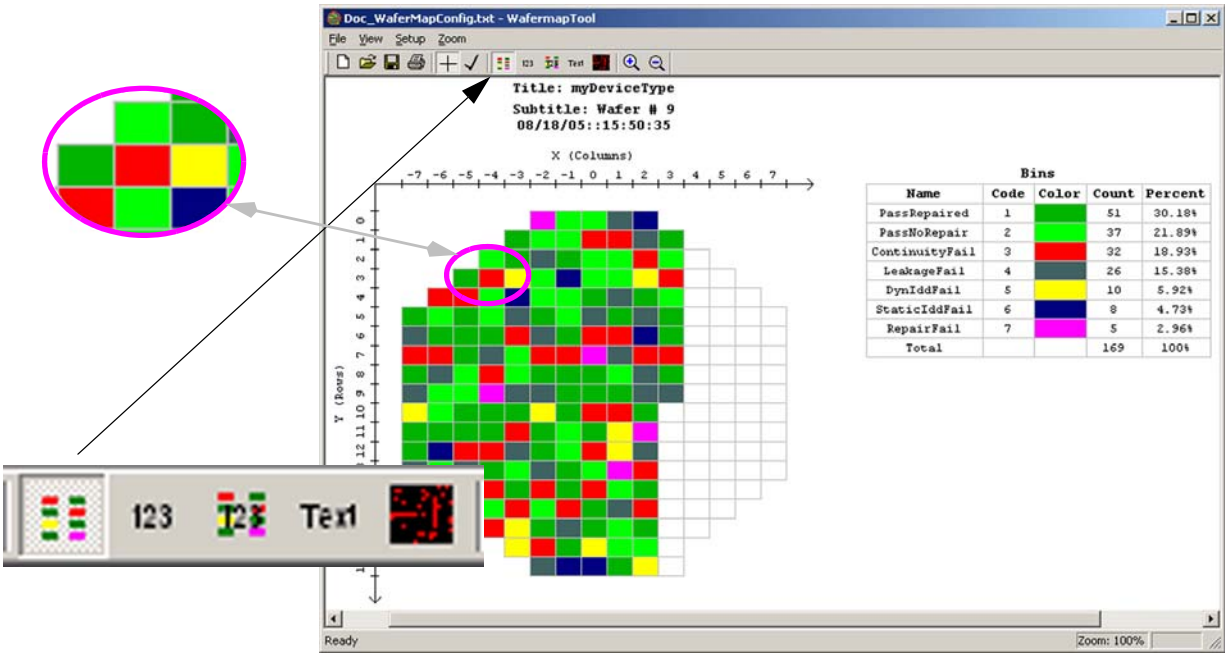


Figure-165: Bin Color View

The bin color to be displayed is defined in the WaferMapTool Configuration, by associating a set of RGB color values with each bin name which will be sent from the test program. The bin name is sent from the test program to WafermapTool using `wmap_die_set()`. For example:

```
TEST_BIN( PassBin ){}
wmap_die_set( wmap_die_bin, 25, 33, "PassBin" );
```

In this example, a test bin named `PassBin` is defined using the `TEST_BIN` macro. Later, when `wmap_die_set()` sends the name of this bin (`PassBin`) to WafermapTool, if Bin Color View is selected, the RGB color mapped to `PassBin` in the WaferMapTool Configuration will be displayed for the die at location 25,33.

The following image shows the bin code view:

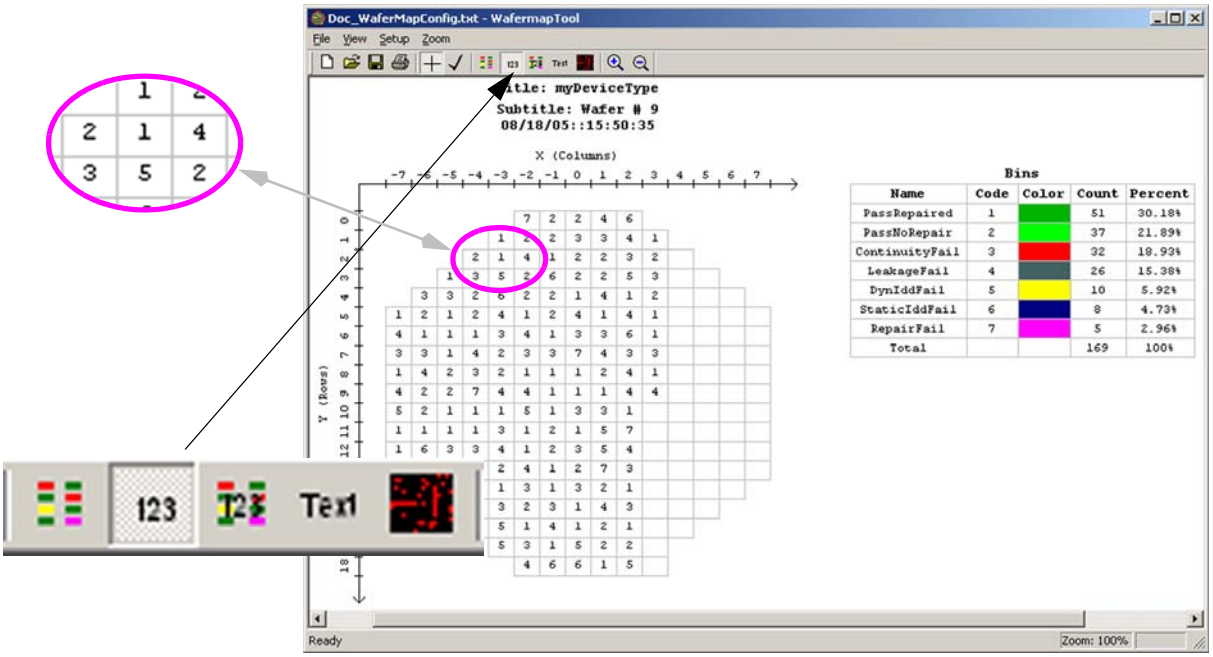


Figure-166: Bin Code View

The bin code to be displayed is defined in the WaferMapTool Configuration, by associating an integer bin number (code) with each bin name which will be sent from the test program. The bin name is sent from the test program to WafermapTool using `wmap_die_set()`. For example:

```
TEST_BIN( FailBin ){}
wmap_die_set( wmap_die_bin, 25, 33, "FailBin" );
```

In this example, a test bin named `FailBin` is defined using the `TEST_BIN` macro. Later, when `wmap_die_set()` sends the name of this bin (`FailBin`) to WafermapTool, if Bin Code View is selected, the bin code mapped to `FailBin` in the WaferMapTool Configuration will be displayed for the die at location 25, 33.

The following image shows the bin color-code view:

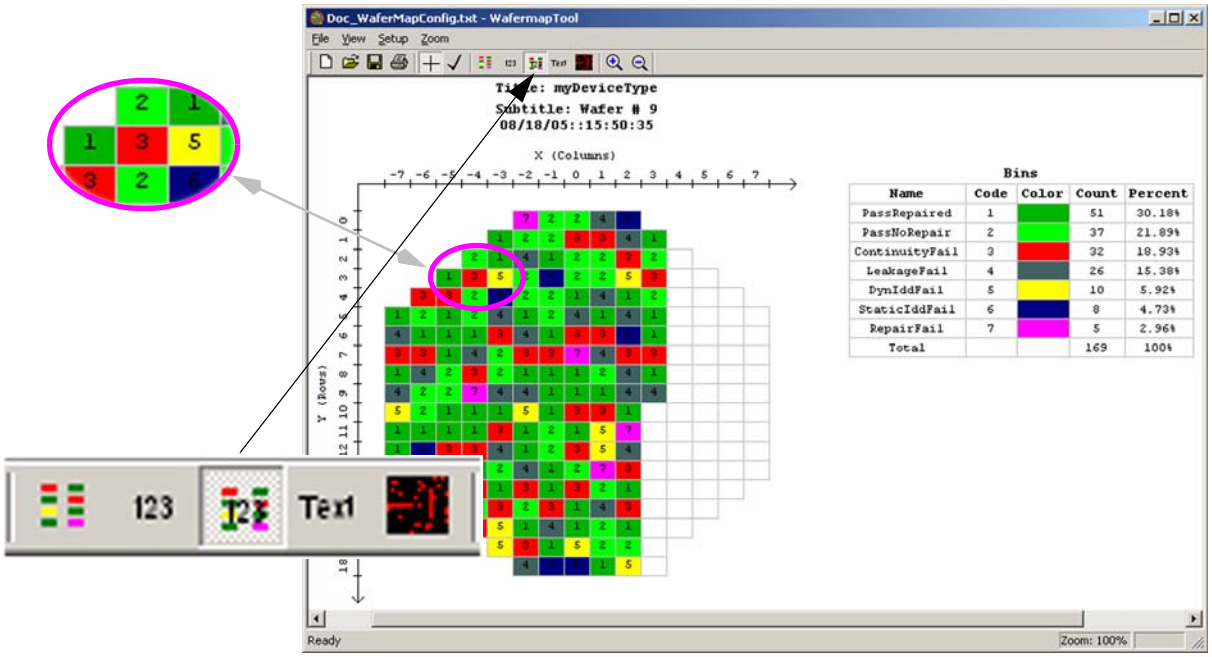


Figure-167: Bin Color-Code View

This view combines the Bin Color View and the Bin Code View.

The following image shows the text view:

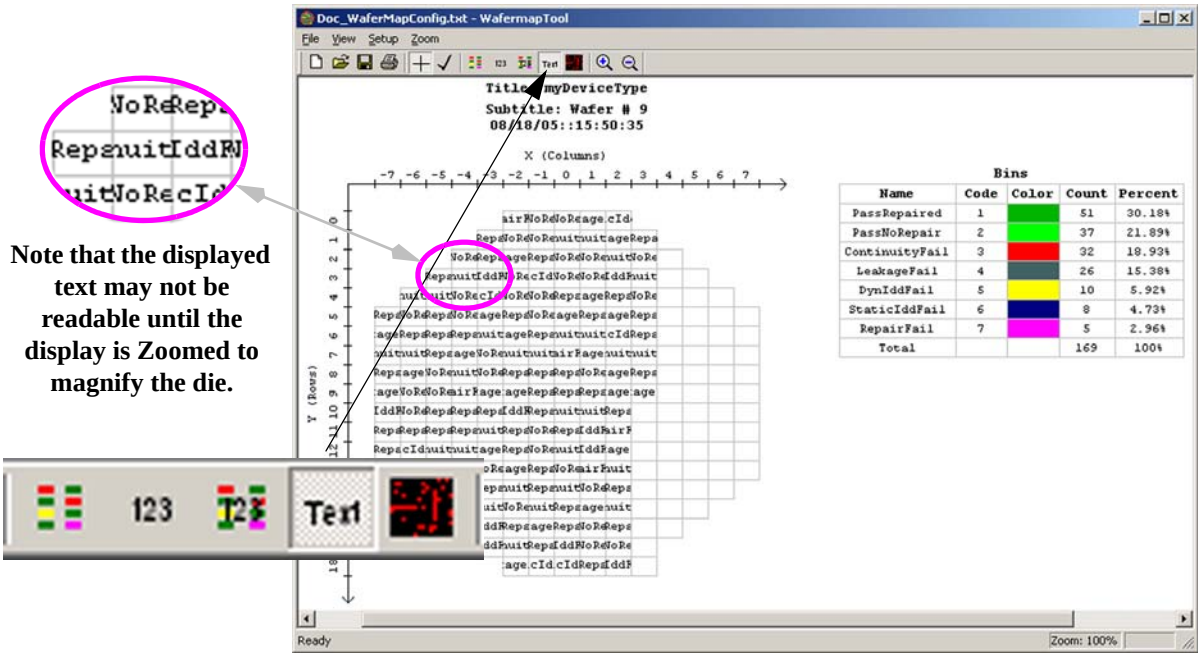


Figure-168: Text View

The text to be displayed is sent as the `value` argument to `wmap_die_set()`. For example:

```
wmap_die_set( wmap_die_text, 34, 29, "myText" );
```

When `wmap_die_set()` sends "`myText`" to WafermapTool, if Text View is selected, the text will be displayed for the die at location 34, 29.

The following example shows a variety of color bitmaps displayed for selected die:

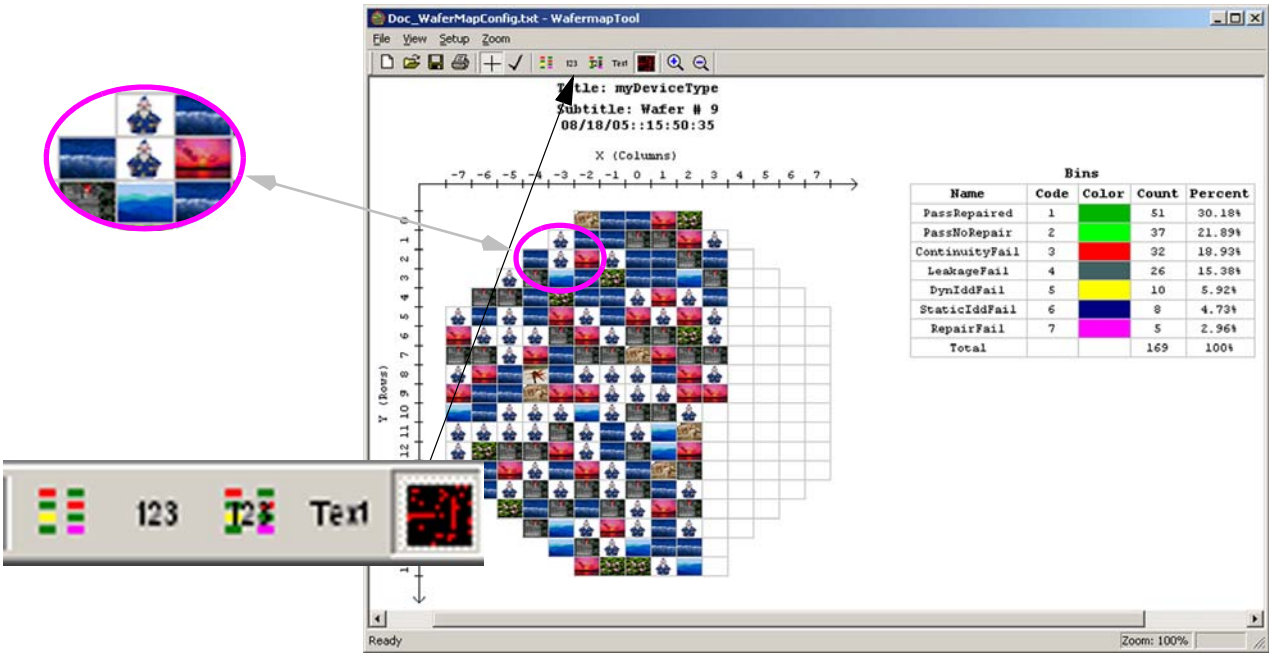


Figure-169: Bitmap View

The bitmap to be displayed is sent as the `value` argument to `wmap_die_set()`. For example:

```
wmap_die_set( wmap_die_bitmap, 34, 29, IDB_BITMAP2 );
```

When `wmap_die_set()` sends `IDB_BITMAP2` to WafermapTool, if Bitmap View is selected, the bitmap image will be displayed for the die at location 34, 29.

Several methods are available for creating the bitmap images to be displayed:

- Dynamically Defined Monochromatic Images
- Dynamically Defined Color Images
- Statically Defined Images
- UI BitmapTool Images i.e. a BitmapTool-like image.

See WafermapTool Die-Bitmap Support.

Note: WafermapTool will automatically scale any bitmap being displayed, to the size of the target die in WafermapTool (see WaferMapTool Configuration `die_x_size`, `die_y_size`). However, the details in complex bitmap images may not be easy to see without zooming to increase the visible size of the die image in WafermapTool.

6.21.7.2 Marked Die

See Die Attributes, Overview.

WafermapTool allows one or more die to be marked:

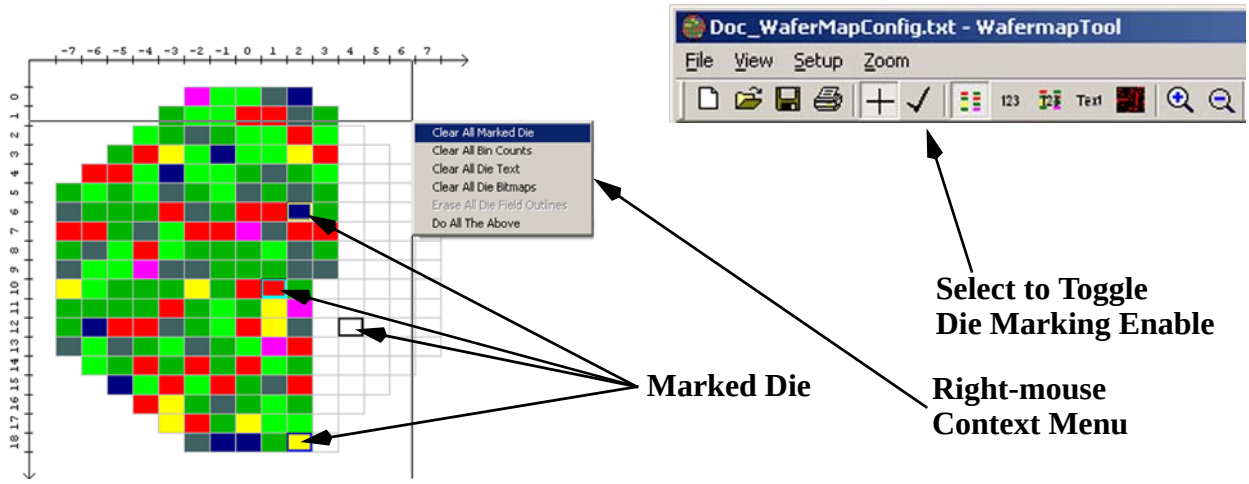


Figure-170: WaferMapTool Marked Die and Clear Dialog

Note the following:

- By default, die marking is disabled. Click the check icon  in WafermapTool's tool bar to enable/disable die marking. The cursor will change to a check mark when die marking is enabled.
- If a call-back function is registered (see `wmap_onclick_set()`) that function will execute, in the Host process only, and receive the coordinates of the die just marked. The call-back code may, for example, direct the wafer prober to move to the marked die.
- A die can also be marked from user code using `wmap_die_set()`.

- Any number of die can be marked. This can be useful when testing multiple die in parallel to indicate which die are being tested at any given moment. User code uses `wmap_die_set()` to mark the desired die.
- Marked die can be un-marked (cleared) several ways:
  - Using the left-mouse button, click the die again.
  - Use the right-mouse button to display the popup menu shown above and select "`Clear all marked die`"
  - From the test program execute `wmap_set()` with `wmap_mark_clear`.

6.21.8 WaferMapTool Software

See Overview.

Rather than define a large set of functions, each targeted at a specific WafermapTool attribute or control, a smaller set of functions are provided. These functions use a keyword to identify the target attribute or control to be accessed. Zero or more additional arguments may be specified to complete the task. The following functions are provided:

- `wmap_set(), wmap_get()` - used to access all WafermapTool attributes except per-die features.
- `wmap_die_set(), wmap_die_get()` - used to access per-die attributes.
- `wmap_cmd_start(), wmap_cmd_end()` - used to improve performance, by defining a collection of WafermapTool commands to be sent to WafermapTool with a single transaction.
- `wmap_die_cmd_start(), wmap_die_cmd_end()` - used to improve performance, by defining a collection of per-die commands to be sent to WafermapTool with a single transaction.
- `wmap_onclick_set()` - used to register a user-written call-back function executed when a die is clicked in WafermapTool's Main Map.
- WafermapTool Die-Bitmap Support includes additional functions used to create or define bitmap images to be displayed for a die.

6.21.8.1 Types, Enums, etc.

See WaferMapTool Software.

Description

The following enumerated types are used in support of various WafermapTool functions:

Description

The `wmap_type` enumerated type is used to identify a WafermapTool attribute or control, to set or get using `wmap_set()` and `wmap_get()`:

```
enum wmap_type {wmap_bin_clear, wmap_mark_clear, wmap_text_clear,
                wmap_bitmap_clear, wmap_all_clear,
                wmap_config_load, wmap_config_save,
                wmap_data_load,wmap_data_save, wmap_title,
                wmap_subtitle, wmap_die_field_clear };
```

The `wmap_die_type` enumerated type is used to identify a WafermapTool die attribute or control, to set or get using `wmap_die_set()` and `wmap_die_get()`. `wmap_die_field` was added in software release h1.1.23:

```
enum wmap_die_type {wmap_die_bin, wmap_die_marked, wmap_die_text,
                    wmap_die_bitmap, wmap_die_field};
};
```

6.21.8.2 wmap_set(), wmap_get()

See WafermapTool, WaferMapTool Software.

Description

The `wmap_set()` function is used to send commands and information to WafermapTool. See `wmap_die_set()` function to perform similar operations for a specific die.

The `wmap_get()` function is used to get the current value of some WafermapTool attributes. See `wmap_die_get()` function to perform similar operations for a specific die.

Note: in general, it is desirable to have a *get* function for each *set* function. However, the initial options supported by `wmap_set()` do not include parameters which would typically need to be retrieved. `wmap_get()` is defined here as a place-holder for potential future enhancements.

Both `wmap_set()` and `wmap_get()`are used for multiple purposes. The `type` argument identifies the target attribute or control being accessed. Zero or more additional arguments are used to specify value(s). The following table describes the various options:

Table 6.21.8.2-1 wmap_set() Type/Value Descriptions

Type	Func	Value Type	Operation
`wmap_all_clear`	Set	`void`	Clears WafermapTool's Main Map in all views (Bin Code View, Bin Color View, Bin Color-Code View, Text View, and Bitmap View), clears the Bin Counts Table, and un-marks all marked die in the Main Map.
`wmap_bin_clear`	Set	`void`	Clears WafermapTool's Main Map for the Bin Code View, Bin Color View and Bin Color-Code View and clears the Bin Counts Table.
`wmap_bitmap_clear`	Set	`void`	Clears WafermapTool's Main Map for only the Bitmap View only and clears the Bin Counts Table.
`wmap_config_load`	Set	`CString`	Loads a WaferMapTool Configuration from the specified file. This completely replaces the existing configuration, if any, and clears the Main Map, Bin Counts Table, and all marked die, if any. See WafermapTool File Access Rules. This will also start WafermapTool if it is not already running.
`wmap_config_save`	Set	`CString`	Saves the current WaferMapTool Configuration to the specified file. See WafermapTool File Access Rules. See Note:.

Table 6.21.8.2-1 wmap_set() Type/Value Descriptions

Type	Func	Value Type	Operation
wmap_data_load	Set	CString	Load a previously saved wafer map from the specified file. This defines the WaferMapTool Configuration (replacing the previous configuration, if any) and loads the Main Map and Bin Counts Table, replacing the content, if any. See WafermapTool File Access Rules.
wmap_data_save	Set	CString	Saves the current wafer map into the specified file. Includes the current WaferMapTool Configuration, the Main Map and Bin Counts Table. See WafermapTool File Access Rules.
wmap_mark_clear	Set	void	Un-marks all marked die in the Main Map.
wmap_subtitle Added in software release h1.1.23	Set	CString	Defines the Subtitle displayed at the top of the WafermapTool display, just below the Title. The Subtitle can only be set from the test program using wmap_set(); i.e. it cannot be set from the WaferMapTool Configuration.
wmap_text_clear	Set	void	Clears WafermapTool's Main Map for only the Text View only and clears the Bin Counts Table.
wmap_title	Set	CString	Defines the main Title displayed at the top of the WafermapTool display. Can also be set via the WaferMapTool Configuration.
	Get	*CString	Returns the current main Title displayed at the top of the WafermapTool display.

Performance can be improved when executing a series of `wmap_set()` functions by using `wmap_cmd_start(), wmap_cmd_end()`.

Usage

The following function is used to send a signal/command to WafermapTool. This version has no additional value arguments:

```
BOOL wmap_set( wmap_type type );
```

The following functions are used to set or get a `CString` value to/from WafermapTool:

```
BOOL wmap_set( wmap_type type, CString value );
BOOL wmap_get( wmap_type type, CString *value );
```

Note: see `wmap_cmd_start(), wmap_cmd_end()` for overloads of `wmap_set()` and `wmap_get()` which include the `wmap_cmd` argument. These are used to improve performance by aggregating multiple commands.

---

where:

`type` identifies the specific WafermapTool attribute being accessed. See wmap_set() Type/ Value Descriptions table above.

`value` is used in two contexts:

- Using the set function, **`value`** specifies the value being set.
- Using the get function, **`value`** is a pointer to an existing variable of the appropriate type used to return the target value.

`wmap_set()` and `wmap_get()` return TRUE when no errors occur, otherwise FALSE is returned.

Example

The following code clears WafermapTool's Main Map for the Bin Code View, Bin Color View and Bin Color-Code View and clears the Bin Counts Table:

```
wmap_set( wmap_bin_clear );
```

The following code saves the wafermap bin data into the specified file:

```
CString fname = "D:/myPath/myWmapSaveFile.wmp";
wmap_set( wmap_data_save, fname );
```

The following code loads the specified wafermap configuration file. This will also start WafermapTool if it is not already running:

```
wmap_set( wmap_config_load, "D:/myPath/myConfigFile.txt" );
```

The following code defines the Subtitle, with an incrementing wafer number and time stamp. This example should be executed each time a new wafer is loaded:

```
static int wafer_no = 0;
CString st;
st.Format( "Wafer => %d\n", ++wafer_no );
st += CTime::GetCurrentTime( ).Format( "%m/%d/%y::%H:%M:%S" );
wmap_set( wmap_subtitle, st );
```

6.21.8.3 WafermapTool File Access Rules

See `wmap_set()`, User Interface & Controls.

The following rules apply to operations which save-to or load-from a file:

- Configuration Files require the *.txt* extension. This facilitates opening/editing these files with a text editor and prevents accidentally saving a wafer map data file into a configuration file, and vice-versa.
- Wafer map data files require the *.wmp* extension. This prevents accidentally saving a wafer map data file into a configuration file, and vice-versa.
- Loading a Configuration File, regardless of the method used, replaces the existing configuration, and clears the Main Map and Bin Counts Table content. This will also start WafermapTool if it is not already running.
- Loading a wafer map data file, regardless of the method used, also replaces the existing configuration, and replaces the contents of the Main Map and Bin Counts Table.
- Loading or saving either file type sets the behavior of subsequent WafermapTool's **File->Save Current** selections. See **Save Current**.

Note: saving a Configuration File to an existing configuration file completely over-writes the original file, including any comments which were manually entered into the target file (using a text editor).

6.21.8.4 wmap_die_set(), wmap_die_get()

See WafermapTool, WaferMapTool Software.

Description

The `wmap_die_set()` function is used to send commands and attributes to a specific die in WafermapTool. As appropriate, this updates the Main Map and Bin Counts Table. See `wmap_set()` function to perform similar operations for general WafermapTool attributes.

The `wmap_die_get()` function is used to retrieve (get) information about a specific die from WafermapTool. See `wmap_get()` function to perform similar operations for general WafermapTool attributes.

`wmap_die_set()` and `wmap_die_get()` are used for multiple purposes. The `type` argument identifies the target attribute or control with additional arguments used to specify the target die and to specify value(s) or to return a value. The following table describes the various options:

Table 6.21.8.4-1 wmap_die_set() Type/Value Descriptions

Type	Func	Value Type	Operation
`wmap_die_bin`	Set	`CString`	Send the test bin name to the specified die in WafermapTool. See Die Display Options. The `CString` value must exactly match one of the bin names specified for `bin_code` and `bin_color` in the WaferMapTool Configuration. Affects Bin Color View, Bin Code View and Bin Color-Code View, see Die Display Options.
	Get	`*CString`	Gets the test bin name for the specified die from WafermapTool.
`wmap_die_bitmap`	Set	`*bitmap`	Send a bitmap image to the specified die in WafermapTool. No get function. Affects Bitmap View only, see Die Display Options.

Table 6.21.8.4-1 wmap_die_set() Type/Value Descriptions *(Continued)*

Type	Func	Value Type	Operation
wmap_die_marked	Set	int	Used to mark or unmark the specified die, similar to left-clicking that die in the Main Map (see Marked Die). Sending the value 0 (FALSE) un-marks the die, sending the value 1 (TRUE) marks the die. Does NOT trigger the call-back function, even if one is registered using wmap_onclick_set().
	Get	*int	Gets the current marked state for the specified die; 0 = unmarked, 1 = marked.
wmap_die_text	Set	CString	Send the text string to the specified die in WafermapTool. Affects Text View only, see Die Display Options.
	Get	*CString	Gets the text string for the specified die from WafermapTool.
wmap_die_field Added in software release h1.1.23	Set	Multiple Values, see Die Field Display	Allows user code to generate a color outline surrounding one or more die. Useful when testing multiple die in parallel, to indicate which die are being tested at any given time. See Die Field Display.

Performance can be improved when executing a series of wmap_die_set() functions by using wmap_die_cmd_start(), wmap_die_cmd_end().

Usage

The following functions are used to set or get a CString value to/from a specific die in WafermapTool. This is used with both wmap_die_bin and wmap_die_text:

```
BOOL wmap_die_set(wmap_die_type type,
                  int x, int y,
                  CString value );
BOOL wmap_die_get(wmap_die_type type,
                  int x, int y,
                  CString *value );
```

The following functions are used to set or get an integer (`int`) value to/from a specific die in WafermapTool. This is used with `wmap_die_marked`:

```
BOOL wmap_die_set( wmap_die_type type, int x, int y, int value );
BOOL wmap_die_get( wmap_die_type type, int x, int y, int *value );
```

The following functions are used to set a monochromatic or color bitmap to a specific die in WafermapTool. This is used with `wmap_die_bitmap`. See WafermapTool Die-Bitmap Support:

```
BOOL wmap_die_set(wmap_die_type type,
                  int x,
                  int y,
                  mono_bitmap *pBitmap );
BOOL wmap_die_set(wmap_die_type type,
                  int x,
                  int y,
                  color_bitmap *pBitmap );
```

The following function uses UI's BitmapTool software to generate a bitmap image and display it for the specified die. This is only used with `wmap_die_bitmap`. Unlike the other versions of `wmap_die_set()` this overload does not have a value parameter. Instead, this option invokes the BitmapTool code which reads errors from the ECR and generates the bitmap image, and sends it to the target die in WafermapTool. See WafermapTool Die-Bitmap Support and UI BitmapTool Images:

```
BOOL wmap_die_set( wmap_die_type type, int x, int y );
```

Note: see `wmap_die_cmd_start(), wmap_die_cmd_end()` for overloads of `wmap_die_set()` and `wmap_die_get()` containing the `wmap_die_cmd` argument. These are used to improve performance by aggregating multiple commands.

Note: see Die Field Display for overloads of `wmap_die_set()` using the `wmap_die_field` argument.

---

where:

`type` identifies the specific die attribute being accessed. See wmap_die_set() Type/Value Descriptions table above.

x and **y** identify the location of the target die. Proper operation requires that these values exactly match one of the die locations defined in the WaferMapTool Configuration. Sending an invalid die coordinate causes an error dialog similar to Figure-155: .

value identifies an additional parameter being sent along with the **type** command specified. See wmap_die_set() Type/Value Descriptions.

wmap_die_set() and wmap_die_get() return TRUE when no errors occur, otherwise FALSE is returned.

Example

The following example sends the bin code OpensFail to the die at coordinates X=5, Y=17:

```
BOOL ok = wmap_die_set( wmap_die_bin, 5, 17, "OpensFail" );
if( ! ok )
   output(" ERROR: wmap_die_set( wmap_die_bin ) at location 5/17");
```

The following example marks the die at coordinates X = 10, Y = 11:

```
BOOL ok = wmap_die_set( wmap_die_marked, 10, 11, 1 );
if( ! ok )
   output(" ERROR: wmap_die_set( wmap_die_marked ) at 10/11");
```

6.21.8.5 wmap_cmd_start(), wmap_cmd_end()

See WafermapTool, WaferMapTool Software.

Description

The wmap_cmd_start() and wmap_cmd_end() functions are used to improve performance when sending multiple wmap_set(), and to a lesser extent wmap_get(), commands from the test program to WafermapTool.

In simple terms, wmap_cmd_start() identifies a collection of commands which will be executed all at once when wmap_cmd_end() is executed. This results in a single program-to-tool transaction for the set of collected commands rather than a series of individual transactions. However...

The previous description is only correct when using wmap_set() (with the wmap_cmd argument). When using wmap_get() (with the wmap_cmd argument), each function is executed immediately, to allow the returned value to be used immediately. However,

performance *may* be improved, depending on whether other associated *get* commands are used with the same wmap_cmd argument, because information related to the specified get parameter is automatically retrieved, in anticipation that it may be requested next.

Also note the following

- Do not nest wmap_cmd_start() and wmap_cmd_end() functions.
- These functions have no effect on wmap_onclick_set() or any registered call-back function, if any.

Usage

```
wmap_cmd* wmap_cmd_start();
void wmap_set( wmap_cmd *cmd, wmap_type type );
void wmap_set( wmap_cmd *cmd, wmap_type type, CString value );
BOOL wmap_get( wmap_cmd *cmd, wmap_type type, CString *value );
void wmap_cmd_end( wmap_cmd *cmd );
```

where:

cmd is the pointer returned by wmap_cmd_start().

type identifies the specific WafermapTool attribute being accessed. See wmap_set(), wmap_get().

wmap_cmd_start() returns a pointer which must be passed to each function to be controlled by wmap_cmd_start()/wmap_cmd_end() and to wmap_cmd_end() to complete the transaction.

Example

The following example executes two wmap_set() functions with a single transaction from the site to WafermapTool:

```
wmap_cmd* c = wmap_cmd_start();
wmap_set( c, wmap_config_load, "D:myPath/MyWMapConfig.txt" );
wmap_set( c, wmap_title, "myWafermapTitle" );
wmap_cmd_end( c );
```

6.21.8.6 wmap_die_cmd_start(), wmap_die_cmd_end()

See WafermapTool, WaferMapTool Software.

Description

The `wmap_die_cmd_start()` and `wmap_die_cmd_end()` functions are used to improve performance when sending multiple `wmap_die_set()`and, to a lesser extent, `wmap_die_get()` commands from the test program to WafermapTool.

In simple terms, `wmap_die_cmd_start()` identifies a collection of die commands which will be executed all at once when `wmap_die_cmd_end()` is executed. This results in a single program-to-tool transaction for the set of collected commands rather than a series of individual transactions. However...

The previous description is only correct when using `wmap_die_set()` (with the `wmap_die_cmd` argument). When using `wmap_die_get()` (with the `wmap_die_cmd` argument), each function is executed immediately, to allow the returned value to be used immediately. However, performance *may* be improved, depending on whether other associated *get* commands are used with the same `wmap_die_cmd` argument, because information related to the specified die is automatically retrieved, in anticipation that it may be requested next. For example, a given die has the following attributes (more may be added in the future):

- `wmap_die_bin`
- `wmap_die_marked`

If `wmap_die_get()` is used with the `wmap_die_cmd` argument to, for example, get `wmap_die_bin`, the `wmap_die_marked` attribute is also automatically retrieved. Then, if `wmap_die_get()` is subsequently used with the same `wmap_die_cmd` argument, to get `wmap_die_marked`, the value previously retrieved is used, saving one site-to-tool transaction.

Also note the following

- Do not nest `wmap_die_cmd_start()` and `wmap_die_cmd_end()` functions.
- These functions have no effect on `wmap_onclick_set()` or any registered call-back function, if any.

Usage

```
wmap_die_cmd* wmap_die_cmd_start();
```

```
void wmap_die_set(wmap_die_cmd *cmd,
                  wmap_die_type type,
                  int x, int y,
                  CString value );
void wmap_die_set(wmap_die_cmd *cmd,
                  wmap_die_type type,
                  int x, int y,
                  int value );
BOOL wmap_die_get(wmap_die_cmd *cmd,
                  wmap_die_type type,
                  int x, int y,
                  CString *val );
BOOL wmap_die_get(wmap_die_cmd *cmd,
                  wmap_die_type type,
                  int x, int y,
                  int *val );
```

The following overload was added in software release h1.1.23.

```
void wmap_die_set(wmap_die_cmd *cmd,
                  wmap_die_type type,
                  int x,
                  int y,
                  int numx,
                  int numy,
                  int penStyle,
                  COLORREF color );
void wmap_die_cmd_end( wmap_die_cmd *cmd );
```

where:

cmd is the pointer returned by `wmap_cmd_start()`.

type identifies the specific WafermapTool die attribute being accessed. See `wmap_die_set(), wmap_die_get()`.

`wmap_die_cmd_start()` returns a pointer which must be passed to each function to be controlled by `wmap_die_cmd_start()/wmap_die_cmd_end()` and to `wmap_die_cmd_end()` to complete the transaction.

Example

The following example executes two `wmap_set()` functions with a single transaction from the site to WafermapTool:

```
wmap_die_cmd* dc = wmap_cmd_start();
wmap_die_set( dc, 0, 10, "OpensFail" );
wmap_die_set( dc, 0, 11, "PassNoRepair" );
wmap_die_cmd_end( dc );
```

6.21.8.7 wmap_onclick_set()

See WafermapTool, WaferMapTool Software.

Description

The `wmap_onclick_set()` function is used to register a call-back function which will be executed any time the user clicks the left-mouse button on a die in WafermapTool's Main Map. The call-back receives the X/Y die coordinates for the die selected.

Note the following:

- The call-back function executes only in the Host process. This is consistent with the execution scope of wafer prober control software.
- The call-back can be registered at any time, but useful operation also requires that WafermapTool be configured. See WaferMapTool Configuration.
- The call-back function name is user-defined, but the prototype is defined by Nextest. See Usage.
- The call-back function can be unregistered by passing a NULL pointer.

Usage

The following function is used to send a keyword and values for a specified die location to WafermapTool:

```
wmap_onclick_set( void* func );
```

where:

`func` is a pointer to a user-defined function with the following prototype:

```
void (*wmap_callback_function)( int x, int y );
```

where:

x and **y** identify the coordinates of the die which was clicked in the WafermapTool display.

Example

The following code registers the user call-back function which follows:

```
wmap_onclick_set( myCallbackFunc );
```

If registered as shown, the following call-back example will execute in the Host process any time a die is clicked in WafermapTool's Main Map. This example will output the die's X/Y location:

```
void myCallbackFunc( int x, int y ) {
   output( "Selected Die at X=> %d, Y=> %d", x, y );
}
```

6.21.9 WafermapTool Die-Bitmap Support

WafermapTool allows bitmap images to be displayed for a die. The following methods are available to define/create the image(s) to be displayed:

- Dynamically Defined Monochromatic Images
- Dynamically Defined Color Images
- Statically Defined Images
- UI BitmapTool Images

6.21.9.1 Dynamically Defined Monochromatic Images

See WafermapTool, WaferMapTool Software, WafermapTool Die-Bitmap Support.

Description

The functions documented here allow test program code to dynamically create, define and destroy a monochromatic bitmap image. In a typical application, the test program will also send the bitmap to a die in WafermapTool (more below). See Dynamically Defined Color Images for methods used with bitmaps containing arbitrary colors or images.

Note the following:

- The `wmap_bitmap_mono_create()` function is used to create a new, empty, monochromatic bitmap image of a specified size.

Note: WafermapTool will automatically scale any bitmap being displayed, to the size of the target die in WafermapTool (see WaferMapTool Configuration `die_x_size`, `die_y_size`). However, the details in complex bitmap images may not be easy to see without zooming to increase the visible size of the die image in WafermapTool.

- The `wmap_bitmap_mono_set()` function is used to set a specified bit in a specified monochromatic bitmap. Bit(s) which are set are displayed using the current BitmapFailColor (see `ui_BitmapFailColor, ui_BitmapPassColor`). The default color is red.
- The `wmap_bitmap_mono_clear()` function is used to clear a specified bit in a specified monochromatic bitmap. Bit(s) which are cleared are displayed using the current BitmapPassColor (see `ui_BitmapFailColor, ui_BitmapPassColor`). The default color is black.
- The `wmap_bitmap_mono_get()` function to get the current value of a specified bit in a specified monochromatic bitmap.
- The `wmap_bitmap_mono_clear_all()` function is used to clear all bits in a specified monochromatic bitmap.
- The `wmap_bitmap_mono_delete()` function is used to destroy a specified monochromatic bitmap.
- `mono_bitmap` is an opaque data type used by these functions as a handle to a monochromatic bitmap.

A `mono_bitmap` can be displayed in WafermapTool, for a specified die, using the `wmap_die_bitmap` option to `wmap_die_set()`.

Usage

```
mono_bitmap* wmap_bitmap_mono_create( int width, int height );

void wmap_bitmap_mono_set(  mono_bitmap *pBitmap,
                            int row,
                            int col );
```

```
BOOL wmap_bitmap_mono_get(  mono_bitmap *pBitmap,
                            int row,
                            int col );
void wmap_bitmap_mono_clear(mono_bitmap *pBitmap,
                            int row,
                            int col );
void wmap_bitmap_mono_clear_all( mono_bitmap *pBitmap );
void wmap_bitmap_mono_delete( mono_bitmap *pBitmap );
```

where:

width and **height** specify the size of the bitmap being created, in pixels.

pBitmap identifies the bitmap of interest, previously created using `wmap_bitmap_mono_create()`.

row and **col** specify the coordinates of a bit to be set, cleared or retrieved. Invalid coordinate values are silently ignored.

`wmap_bitmap_mono_create()` returns the bitmap created.

`wmap_bitmap_mono_get()` returns TRUE if the specified bit is currently set and FALSE if the bit is cleared.

Example

The following example creates a 64x64 pixel monochromatic bitmap identified as `myBitmap`. A diagonal of bits are set then the bitmap is sent to WafermapTool for display in the die location specified by `die_x`/`die_y`. Last, the bitmap is destroyed:

```
mono_bitmap* myBitmap = wmap_bitmap_mono_create( 64, 64 );
for( int i = 0; i < 64; ++i )
   wmap_bitmap_mono_set( myBitmap, i, i );
wmap_die_set( wmap_die_bitmap, die_x, die_y, myBitmap );
wmap_bitmap_mono_delete( myBitmap );
```

6.21.9.2 Dynamically Defined Color Images

See WafermapTool, WaferMapTool Software, WafermapTool Die-Bitmap Support.

Description

The functions documented here allow test program code to dynamically create, define and destroy a color bitmap image. In a typical application, the test program will also send the bitmap to a die in WafermapTool (more below). See Dynamically Defined Monochromatic Images for methods used with bitmaps containing only two colors.

Note the following:

- The `wmap_bitmap_color_create()` function is used to create a new, empty, color bitmap of a specified size.

Note: WafermapTool will automatically scale any bitmap being displayed, to the size of the target die in WafermapTool (see WaferMapTool Configuration `die_x_size`, `die_y_size`). However, the details in complex bitmap images may not be easy to see without zooming to increase the visible size of the die image in WafermapTool.

- The `wmap_bitmap_color_setcolor()` function is used to set a specified bit in a specified color bitmap to an RGB color.
- The `wmap_bitmap_color_getcolor()` function is used to get the current RGB color value of a specified bit in a specified color bitmap.
- The `wmap_bitmap_color_delete()` function is used to destroy a specified color bitmap.
- `color_bitmap` is an opaque data type used by these functions as a handle to a color bitmap.

A `color_bitmap` can be displayed in WafermapTool, for a specified die, using the `wmap_die_bitmap` option to `wmap_die_set()`.

Also see Dynamically Defined Monochromatic Images.

Usage

```
color_bitmap* wmap_bitmap_color_create(
                int width,
                int height,
                COLORREF bg_color DEFAULT_VALUE( RGB( 0, 0, 0 ) ) );
void wmap_bitmap_color_setcolor( color_bitmap *pBitmap,
                                 int row,
                                 int col,
                                 COLORREF rgb );
```

```
COLORREF wmap_bitmap_color_getcolor(color_bitmap *pBitmap,
                                    int row,
                                    int col );
void wmap_bitmap_color_delete( color_bitmap *pBitmap );
```

where:

`width` and **`height`** specify the size of the bitmap being created, in pixels.

`bg_color` is optional and, if used, specifies the RGB background color for the bitmap being created. Default = 0,0,0 i.e. black.

`pBitmap` identifies the bitmap of interest, previously created using `wmap_bitmap_color_create()`.

`row` and **`col`** specify the coordinates of a bit for which the RBG color is to be set or retrieved. Invalid coordinate values are silently ignored.

`wmap_bitmap_color_create()` returns the bitmap created.

`wmap_bitmap_color_getcolor()` returns the RGB value as a `COLORREF`, a Microsoft defined data type.

Example

The following example creates a 64x64 pixel color bitmap identified as `myCBmap` with a grey background color. Then a diagonal of pixels are set to 4 different colors. The bitmap is sent to WafermapTool for display in the die location specified by `die_x`/`die_y`. The RGB color of the pixel at 8/8 is retrieves and the individual color values are printed. Last, the bitmap is destroyed:

```
color_bitmap* myCBmap;
myCBmap = wmap_bitmap_color_create( 64, 64, RGB( 128, 128, 128 ));
for( int i = 0; i < 16; ++i )
   wmap_bitmap_color_setcolor( myCBmap, i, i, RGB( 255, 255, 0 ) );
for( i = 16; i < 32; ++i )
   wmap_bitmap_color_setcolor( myCBmap, i, i, RGB( 255, 0, 255 ) );
for( i = 32; i < 48; ++i )
   wmap_bitmap_color_setcolor( myCBmap, i, i, RGB( 0, 255, 255 ) );
for( i = 48; i < 64; ++i )
   wmap_bitmap_color_setcolor( myCBmap, i, i, RGB(255, 255, 255 ));
wmap_die_set( wmap_die_bitmap, die_x, die_y, myCBmap );
```

```
COLORREF c = wmap_bitmap_color_getcolor( myCBmap, 8, 8 );
BYTE v = GetRValue( c ); output("  Red   => %d", v );  // SB 255
     v = GetGValue( c ); output("  Green => %d", v );  // SB 255
     v = GetBValue( c ); output("  Blue  => %d", v );  // SB 0
wmap_bitmap_color_delete( myCBmap );
```

6.21.9.3 Statically Defined Images

See WafermapTool, WaferMapTool Software, WafermapTool Die-Bitmap Support.

Description

In WafermapTool, It is possible to display statically defined bitmap image for a die (see Bitmap View). Both methods below use Developer Studio to incorporate the desired bitmap(s) into the test program:

- Import a bitmap image from disk into the test program, as a Bitmap resource.
- Use Developer Studio's built-in bitmap editor to manually create or modify a Bitmap resource.

Using either method, the resulting bitmap image will typically have an identity similar to `IDB_BITMAP1`, which is then used as the `value` argument to `wmap_die_set()`. For example:

```
wmap_die_set( wmap_die_bitmap, die_x, die_y, IDB_BITMAP1 );
```

Note: WafermapTool will automatically scale any bitmap being displayed, to the size of the target die in WafermapTool (see WaferMapTool Configuration `die_x_size`, `die_y_size`). However, the details in complex bitmap images may not be easy to see without zooming to increase the visible size of the die image in WafermapTool.

6.21.9.4 UI BitmapTool Images

See WafermapTool, WaferMapTool Software, WafermapTool Die-Bitmap Support.

Description

WafermapTool allows Error Catch RAM (ECR)-based bitmap images to be displayed for a die. This is the same image, typically much reduced, displayed using UI's BitmapTool.

Note the following:

- One version (overload) of the `wmap_die_set()` function generates the bitmap image and sends it to a specific die in WafermapTool:

  ```
  BOOL wmap_die_set( wmap_die_type type, int x, int y );
  ```

  e.g.

  ```
  wmap_die_set( wmap_die_bin, 9, 11 );
  ```

  Note that unlike the other versions of `wmap_die_set()` this overload does not have a value parameter. This version is only used to invoke the BitmapTool code which reads errors from the ECR and generates the bitmap image, and sends it to a die in WafermapTool.
- All preparations for generating a bitmap image using BitmapTool are also required to use this feature. This includes configuring the ECR, executing a test which executes a Memory Test Pattern which captures errors to the ECR, etc.
- Even though BitmapTool is not visibly used to obtain the image to be displayed same underlying code is executed, much as though BitmapTool was being used interactively. Thus, all of BitmapTool's various options can/will affect the resulting image, whether set manually using BitmapTool's menus/controls and/or set using the various UI User Variables specific to BitmapTool (`ui_BitmapDisplayMode`, `ui_BitmapFailColor, ui_BitmapPassColor`, etc.). In other words, the bitmap image displayed is configured by the current BitmapTool configuration settings, whether default values or as configured by the user or user code.
- If BitmapTool is in use (visible), its display will change to show the image being collected for WafermapTool. Any existing image will be discarded.

6.21.10 Die Field Display

See WafermapTool, WaferMapTool Software.

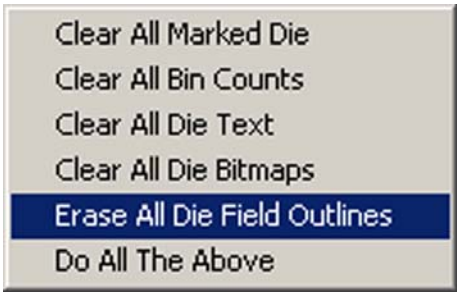
Note: first available in software release h1.1.23.

Description

The `wmap_die_field` option to the `wmap_die_set()` function allows user code to generate a visual color outline surrounding one or more die. This is useful when testing multiple die in parallel, to indicate which die are being or can be tested as a group.

Note the following:

- A die field outline is specified by identifying an origin die, using the `x`, `y` arguments, and specifying the size of the outline as a count of die in the X and Y axis, using `numx` and `numy` arguments. See Usage.
- Multiple die field outlines may be drawn and coexist. When defining multiple outlines at one time it is faster to queue multiple `wmap_die_set()` commands using `wmap_die_cmd_start()` and `wmap_die_cmd_start()`. See Examples.
- An existing die field outline can be erased (cleared) several ways:
  - Execute `wmap_die_set()` with the `numx`, and `numx` arguments both set to 0, AND the `x` and `y` values set to the origin die of the previously generated outline to be erased.
  - In WafermapTool, use the right-mouse button to display the context menu shown below and select **`Erase All Die Field Outlines`**. As indicated, this erases all die field outlines:



  - Use the right-mouse button to display the context menu shown above and select **`Do All The Above`**. This clears all non-configuration data from the Wafermap.
- A die field outline may cover portions of the wafermap which don't contain die, with the following restrictions:
  - The `x` and `y` argument values to `wmap_die_set()` must be the coordinates of a valid die.
  - The die field outline is clipped by the wafermap configuration's `xmin`, `xmax`, `ymin`, and `ymax` values. See WaferMapTool Configuration. In simple terms, the outline will not extend past any border of the wafer.

Usage

```
BOOL wmap_die_set(wmap_die_type type,
                  int x,
                  int y,
                  int numx,
                  int numy,
                  int penStyle,
                  COLORREF color );
```

Note: see `wmap_die_cmd_start(), wmap_die_cmd_end()` for overloads of `wmap_die_set()` containing the `wmap_die_cmd` argument. These are used to improve performance by aggregating multiple commands.

---

where:

x and **y** specify the reference (origin) die location, which must be a valid die location. The die field outline then extends **numx** and **numy** die. The direction that the outline actually extends is determined by the wafermap's axis orientation and direction configuration. See WaferMapTool Configuration (`axis_horizontal`, `axis_horizontal_inc`, etc.).

`penStyle` specifies the type of line used to drawn the outline. The following declarations are available for use as `penStyle` argument values. These are defined in a Microsoft library file:

```
#define PS_SOLID       0 // Solid line
#define PS_DASH        1 // -------
#define PS_DOT         2 // .......
#define PS_DASHDOT     3 // _._._._
#define PS_DASHDOTDOT  4 // _.._.._
```

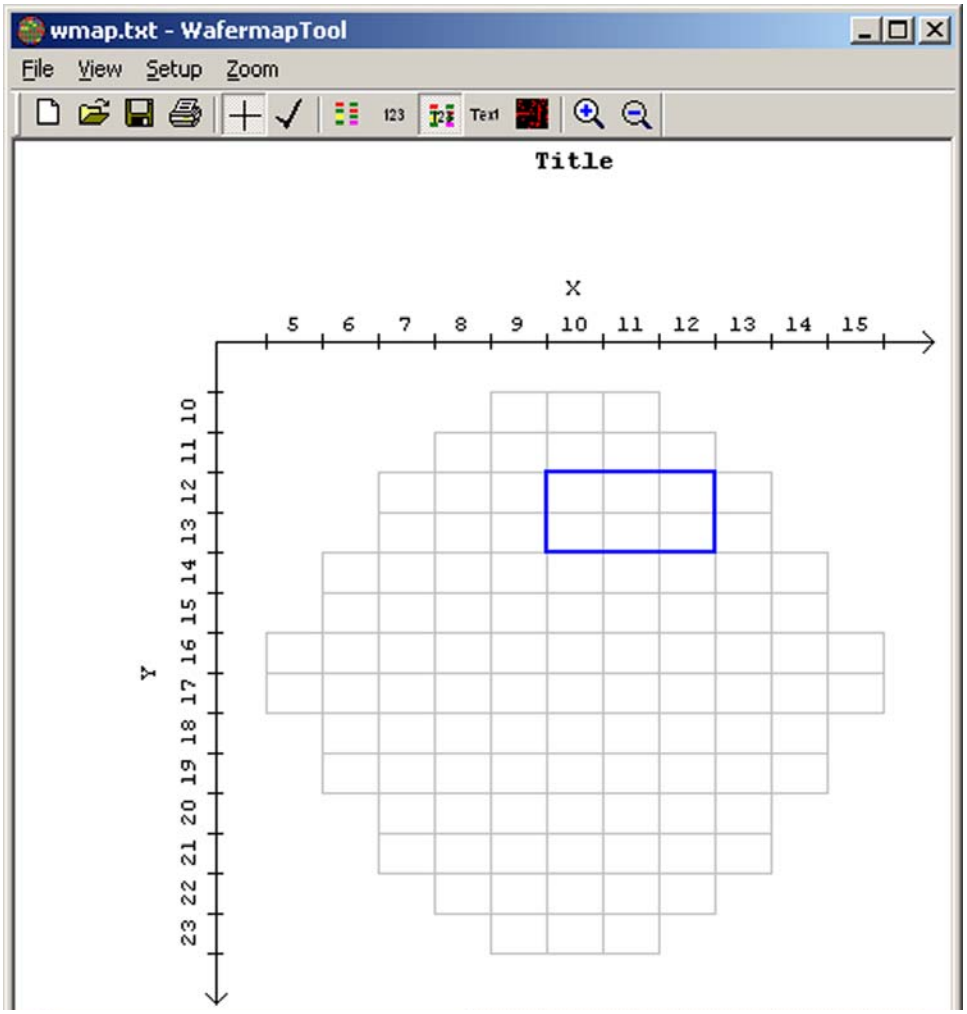
`color` specifies the desired RGB color for the outline being created. This is best defined using the `RGB` macro; i.e. `RGB(0,0,0)` = black, `RGB(255,0,0)` = red, etc. See Examples.

`wmap_die_set()` returns TRUE when no errors occur, otherwise FALSE is returned.

Examples

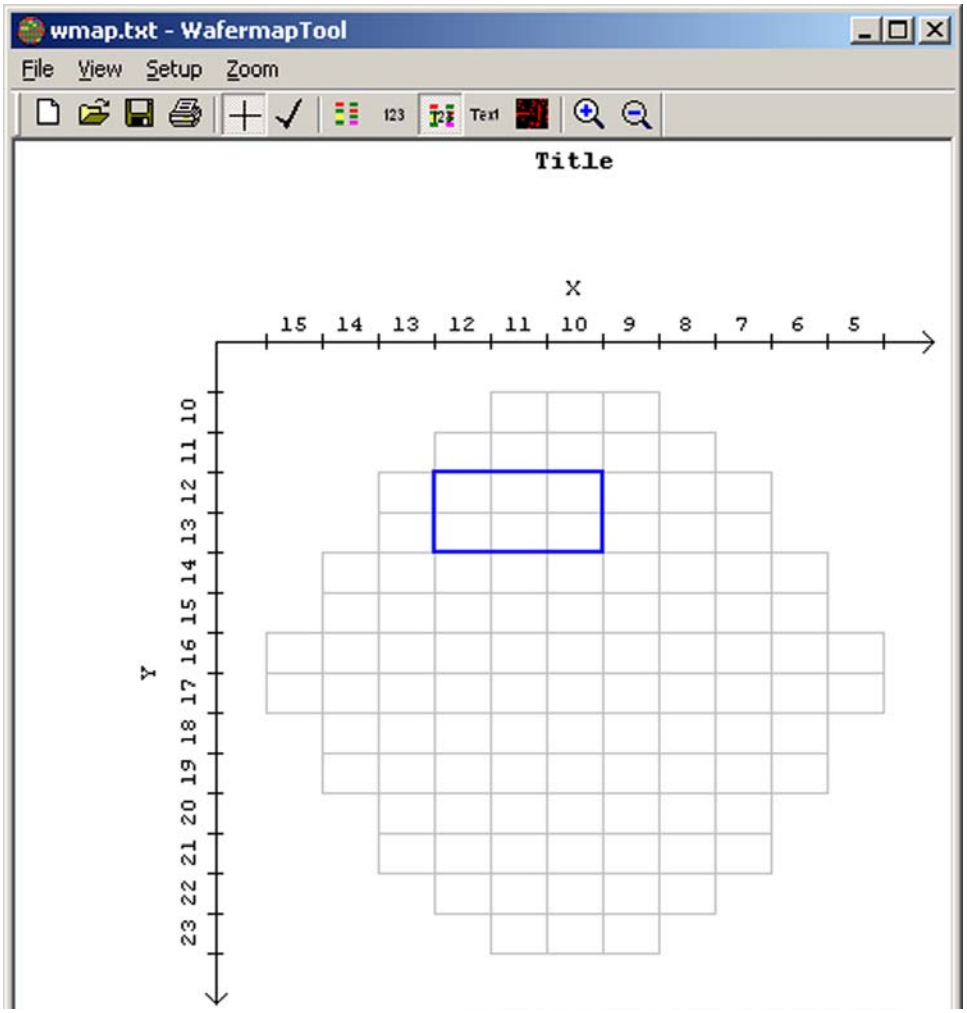
The following three examples use different die field specifications but produce the same result, shown in the image below:

```
wmap_die_set( wmap_die_field, 12,12, -3,2, PS_SOLID, RGB(0,0,255));
wmap_die_set( wmap_die_field, 10,13, 3,-2, PS_SOLID, RGB(0,0,255));
wmap_die_set( wmap_die_field, 12,13, -3,-2, PS_SOLID, RGB(0,0,255));
```



The following example shows the effect when the X axis direction is reversed:

```
wmap_die_set( wmap_die_field, 10,12, 3,2, PS_SOLID, RGB(0,0,255));
```



The following example queue's several commands to draw multiple die field outlines more quickly:

```
wmap_die_cmd *c = wmap_die_cmd_start();
wmap_die_set( c, wmap_die_field, 9, 12,3,2, PS_DASH, RGB(0,0,0));
wmap_die_set( c, wmap_die_field, 9,14, 3,2, PS_DASH, RGB(0,0,0));
wmap_die_cmd_end( c );
```

Chapter 7 Advanced Topics

This section includes the following topics

7.1 User Variables

7.1.1 Overview

Description

User variables are special global variable objects that are used to:

- Provide variables of various types whose values can be viewed and modified using UI's User Variables Tool.
- Connect the graphic components of User Dialogs with user-written C-code. In this context, user variables provide for both value storage and command execution functionality. The Test Program Wizards generates the *dialog.cpp* file which contains many basic examples of using user variables with dialogs.
- Pass both values and/or commands between the test program processes, i.e. between Host and Site and User Tools. Read the Overview in Binning for information about the different processes.

User-defined User Variables can be created to perform specialized functions in test programs, User Dialogs, and User Tools.

There are a large number of built-in UI User Variables that support commonly desired functions.

User variables share the following common attributes:

- Each user variable has a name, which can be treated as any variable is used, for value storage, logical operations, etc.
- Each user variable has an initial value (except for `VOID_VARIABLE` which has no value).
- Each user variable has an optional *prompt string* which, if defined, is displayed in User Variables Tool to enable display and modification of the variable at runtime.
- Each user variable has optional body code, which can execute standard C/C++ code and can call the Nextest functions. When the user variable is invoked (see Invoking User Variable Body Code), the body code is executed (in a separate thread).
- When a user variable is tied to a component of User Dialogs, modification of its value within its body code automatically causes the dialog value to be updated when the body code execution terminates. Separate functions also exist to explicitly update the dialog or the user variable: see Transferring Values to/from Dialog Resources.
- Within the scope of user variable body code, two built-in variables named **`sender`** and **`oldval`** are automatically accessible. `sender` identifies the `site_num()` which invoked the user variable body code. `oldval` contains the previous value of the user variable.
- User variables are global to the process in which they are defined. This means that user variables defined in test program code automatically exist in both the Host and all Site processes. Conversely, a user variable defined in User Tools do not automatically exist in the test program, and vice versa.
- The `remote_send()`, `remote_fetch()`, `remote_set()`, and `remote_get()` functions can be used to synchronize the values of a user variable between different processes. Some can also be used to invoke the body code (see Invoking User Variable Body Code).
- A collection (set, snapshot) of user variables can be created using `SNAPSHOT()`. A snapshot can then be used with `remote_send()`, `remote_fetch()`, `remote_set()`, and `remote_get()` to perform the same operation on a number of user variables with a single function call.

- It is possible for one site to intercept remote transactions to specified User-defined User Variables sent to a different site. See Intercepting User Variables.

7.1.2 Usage

This section shows an overview of how user variables can be used. One `INT` user variable, named `myIntVar`, is defined here and used in all of the examples below:

```
INT_VARIABLE( myIntVar, 0, "myIntVar") {
   output(" myIntVar = %d, previously = %d", myIntVar, oldval );
}
```

`myIntVar` is used as a simple variable. None of these execute `myIntVar`'s body code:

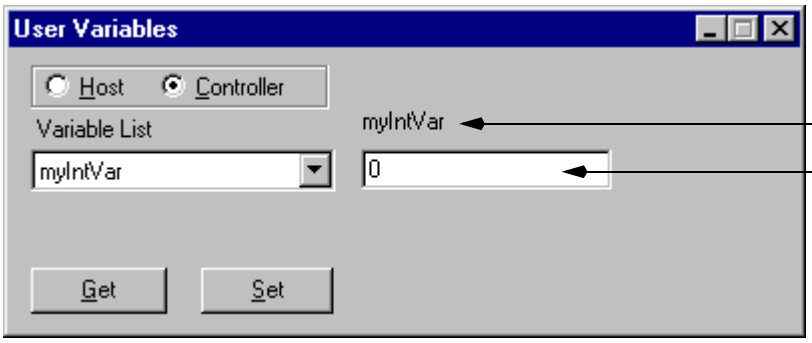
```
myIntVar = 10;
myIntVar++;
if( myIntVar < 10) {...}
for (myIntVar = 10; myIntVar >0; myIntVar--) {...}
etc...
```

Methods for executing the body code of `myIntVar`. For more information see Invoking User Variable Body Code:

```
// Execute body code in the local process
invoke ( myIntVar );

// Update value in Site-1, do not execute body code
remote_send( myIntVar, 1, FALSE, INFINITE );
remote_set( "myIntVar", value, 1, FALSE ); // remote_set()

// Update value in Site-1, do execute body code on site-1
remote_send( myIntVar, 1, TRUE, INFINITE );
remote_set( "myIntVar", value, 1, TRUE );

// Get value from Host (Site 0), do not execute body code
remote_fetch( myIntVar, 0, FALSE );
remote_get( "myIntVar", 0, FALSE ); // remote_get()

// Get value from Host (Site 0), body code executes in local process
remote_fetch( myIntVar, 0, TRUE );
remote_get( "myIntVar", 0, TRUE );
```

Note that what is shown above uses just one version (one overload) of `remote_send()`, `remote_fetch()`, `remote_set()`, and `remote_get()`; there are others.

UI's User Variables Tool can be used to view and modify `myIntVar`'s value:



The code below is typical of that used to support User Dialogs. None of these execute the body code of `myIntVar`:

```
// Connect myIntVar to dialog resource IDC_widget1
CONTROL( IDC_widget1, myIntVar )

// Get the current value from the dialog resource into myIntVar
update_variable( myIntVar );

// Put the current value of myIntVar into the dialog resource
update_control( myIntVar );
```

Assuming additional user variables named `myBoolVar`, and `myCStringVar` exist, the following creates a snapshot containing all, three variables. This can be used with `remote_send()`, `remote_fetch()`, `remote_set()`, and `remote_get()`:

```
SNAPSHOT( name ) { // See SNAPSHOT()
   VARIABLE( myIntVar )
   VARIABLE( myBoolVar )
   VARIABLE( myCStringVar )
}
```

7.1.3 User-defined User Variables

When the built-in UI User Variables do not provide a desired functionality, user-defined user variables may be used, in test program code and User Tools code.

A set of Test System Macros exists for this purpose. A corresponding set of macros exists to make external user variable declarations.

Usage

The following macros are used to define user variables:

```
BOOL_VARIABLE( var_name, initial_value, prompt ) { body code }
CSTRING_VARIABLE( var_name, initial_value, prompt ) { body code }
DOUBLE_VARIABLE( var_name, initial_value, prompt ) { body code }
DWORD_VARIABLE( var_name, initial_value, prompt ) { body code }
FLOAT_VARIABLE( var_name, initial_value, prompt ) { body code }
INT_VARIABLE( var_name, initial_value, prompt ) { body code }
INT64_VARIABLE( var_name, initial_value, prompt ) { body code }
UINT64_VARIABLE( var_name, initial_value, prompt ) { body code }
ONEOF_VARIABLE( var_name, value_list, prompt ) { body code }
VOID_VARIABLE( var_name, prompt ) { body code }
```

The following Test System Macros are used to declare a user variable as *external:*

```
EXTERN_BOOL_VARIABLE( var_name )
EXTERN_CSTRING_VARIABLE( var_name )
EXTERN_DOUBLE_VARIABLE( var_name )
EXTERN_DWORD_VARIABLE( var_name )
EXTERN_FLOAT_VARIABLE( var_name )
EXTERN_INT_VARIABLE( var_name )
EXTERN_INT64_VARIABLE( var_name )
EXTERN_UINT64_VARIABLE( var_name )
EXTERN_ONEOF_VARIABLE( var_name )
EXTERN_VOID_VARIABLE( var_name )
```

where:

The prefix in the macro name (**`VOID`**, **`INT`**, **`BOOL`**, etc.) defines the type of user variable being created (`void`, `int`, `BOOL`, etc.). The **`ONEOF_VARIABLE`** is essentially the same as **`CSTRING_VARIABLE`**, except that the value list is a comma delimited string of discrete values. This is useful for filling in the values of combo boxes in User Dialogs.

var_name is the name of the variable being defined. The name must follow the rules for C identifiers for variables of the same generic type as specified by the prefix in the macro name (**VOID, INT, BOOL,** etc.).

The **EXTERN** macros are used to make external, or forward, declarations. They can be thought of as equivalent to the normal C *extern* declaration.

initial_value is the initial value of the user variable. The initial value is set at program initialization and can be a value or a function that returns a value of the correct type.

prompt is a string enclosed in double quotes. This string will appear above the edit box in UI's User Variables Tool. A NULL prompt string ("") prevents the user variable from appearing in User Variables Tool.

body code is optional, and can include any C++ code. This body code will be executed when the user variable is invoked (see Invoking User Variable Body Code).

Examples

Example 1:

```
INT_VARIABLE( bad_cell_count, 0, "" ) {}
```

In this example, an INT user variable named bad_cell_count is defined. The initial value of bad_cell_count is 0, and there is no prompt string thus this user variable will not be accessible in User Variables Tool. There is no C code to execute in the body section of the variable definition.

Example 2:

In this example, code in a TEST_BLOCK (which executes in a Site process) causes an AfxMessageBox to be displayed in the Host process, and waits for it to be dismissed before continuing.

```
// The body code of okDialog will be invoked using remote_send()
// from DoConfirm(). Intended to execute in the Host process
CSTRING_VARIABLE( okDialog, "", "" ) {
   AfxMessageBox(vFormat("%s\n(from site:%d)", okDialog, sender));
}
// DoConfirm() is intended for Site execution. It will cause the
// Host to display an AfxMessageBox
void DoConfirm( LPCTSTR question ) {
   // Display question in an AfxMessageBox on the Host. Wait for the
   // user response.
```

```
   okDialog = question; // Set uVar value = question
   remote_send( okDialog, 0, TRUE, INFINITE ); // 0 = Host
}
TEST_BLOCK( abc ) {
   // ...
   DoConfirm( "About to set vih" );
   // ...
}
```

7.1.4 Invoking User Variable Body Code

Description

The body code of user variables can be caused to execute using the following methods:

- Directly, using the `invoke()` function. The body code executes in the process which calls `invoke()`.
- Indirectly, using `remote_send()`, `remote_fetch()`, `remote_set()`, `remote_get()`. The body code executes when the TRUE argument is specified, and executes in the process which is the *destination* of the variable value being set, send, get, or fetched. This requires that the calling code know the site number (see `site_num()`) of the destination process, which is simple for Host/Site processes but more complex for User Tools. See Intercepting User Variables. It is also possible for one site to intercept remote transactions to specified User-defined User Variables sent to a different site. See Intercepting User Variables.
- By setting a new value in the user variable using User Variables Tool. The body code executes in the process specified using the radio buttons in User Variables Tool.
- When `set_values_from_file()` is called to perform User Variable Text File Initialization of user variables.

User variable body code executes in its own thread. If multiple Sites `remote_send()`, `remote_fetch()`, `remote_set()`, or `remote_get()` to/from the Host at the same time, the events are serialized.

7.1.5 User Variable Command Line Initialization

Description

Test programs can be invoked from a Windows shell command line. When this is done, it is possible to also start and use UI, or not.

The latter case is targeted at software testing, is quite limited, not very useful in normal device testing operations, and thus not documented here. For the former case, see Starting UI from a Command Line.

It is also possible to specify other command line options, modes, etc. most of which are controlled using UI User Variables. Command line usage and syntax is covered in the detailed documentation of each UI user variable.

When UI is started from a command line it is possible to specify initial values for User-defined User Variables. It is also possible to set different values for the same user variable in Host vs. Site processes. In both cases, these values over-ride the value specified in the source code.

If the test program loaded does not contain the specified user variable (exact spelling, case sensitive) an error message similar to the following will be displayed in either the Host output window, or the Site output window(s) of all enabled sites.

```
Warning: Attempt to set undefined user-variable "myVar" from site
-1 ignored.
```

Note that site -1 refers to UI, which forwards the command line value to the Host or Site processes.

When a test program is unloaded (Closed) it is possible, without terminating UI, to load another (or the same) test program. The *reload option*, enabled using the added **R** token as shown below, is provided. If used, this causes any user variable initialization specified via the command line or batch file (see User Variable Batch File Initialization) to be performed on each test programs loaded using the current instance of UI. When the reload option is not specified, the initial values of all user variables in the 2nd..nth test programs loaded will be the value specified in the source code. Once UI is terminated, all reload information is discarded.

Usage

```
/S:uVarName=<value>
/SR:uVarName=<value>
```

```
/H:uVarName=<value>
/HR:uVarName=<value>
```

where:

uVarName is the user variable to be initialized.

<value> is the initialization value, and must be of the correct type for the user variable being initialized (`int`, `BOOL`, etc.).

/S is used to initialize the specified user variable's value in all enabled Site processes.

/SR is the same as **/S** except that the Reload option is enabled, see above.

/H is used to initialize the specified user variable's value in the Host process.

/HR is the same as **/H** except that the Reload option is enabled, see above.

Example

The following example command line starts UI and initializes two user variables. One, named `test_var` is initialized to 10 in the Host process. The other, named `my_var`, is initialized to -1, in all enabled Site process(es). In this example, proper operation requires that these user variables both be `INT_VARIABLE`, `DWORD_VARIABLE`, `INT64_VARIABLE`, `FLOAT_VARIABLE`, or `DOUBLE_VARIABLE`:

```
ui /H:test_var=10 /S:my_var=-1
```

7.1.6 User Variable Batch File Initialization

Description

Test programs can be invoked from an ASCII batch file, using ui_BatchFile. The batch file can also specify numerous options, modes, etc. most of which are controlled using UI User Variables. Batch file usage and syntax is covered in the detailed documentation of each UI user variable.

When a batch file is used, it is possible to specify initial values for user variables. It is also possible to set different values for the same user variable in Host vs. Site processes. In both cases, these values over-ride the value specified in the source code.

To do this in a batch file combines the use of the ui_HostModeCommandLine and/or ui_SiteModeCommandLine UI user variables, plus the syntax documented in User Variable Command Line Initialization.

If the test program loaded does not contain a specified user variable (exact spelling, case sensitive) an error message similar to the following will be displayed in either the Host output window, or the Site output window(s) of all enabled sites.

```
Warning: Attempt to set undefined user-variable "myVar" from site
-1 ignored.
```

Note that site -1 refers to UI, which forwards the command line value to the Host or Site processes.

Usage

```
ui_SiteModeCommandLine=/S:uVarName=<value>
ui_SiteModeCommandLine=/SR:uVarName=<value>
ui_HostModeCommandLine=/H:uVarName=<value>
ui_HostModeCommandLine=/HR:uVarName=<value>
```

Note that the values assigned to ui_HostModeCommandLine and ui_SiteModeCommandLine use the same syntax documented in User Variable Command Line Initialization.

Example

The following is an example of a batch file which inhibits the initial UI splash screen, enables engineering mode, initializes three user variables, and loads the test program *D:/MinTestProg/Debug/MinTestProg*. Note the quoted string value assigned to CSvar which, for proper operation, must be a CSTRING_VARIABLE:

```
ui_NoLogo=1
ui_EngineeringMode=1
ui_SiteModeCommandLine=/S:Var1=101 /S:CSvar="str val" /S:Var2=99
ui_TestProgName=D:/MinTestProg/Debug/MinTestProg.exe
```

If these statements are in the file *C:\test9_batch.txt* the batch file can be executed from the command line using

C:\ui /BATCH=C:\test9_batch.txt

7.1.7 User Variable Text File Initialization

See Built-in User Variables.

Definition

The `set_values_from_file()` function allows user C-code to set the value of one or more user defined User Variables from an external text file.

The Built-in User Variables `builtin_batch_file` can be used for the same purpose.

When `set_values_from_file()` is executed, it opens the specified file and reads the names and values of user variables. For each user variable in the file, it updates the value, and then executes the body code of that user variable. Anything in the file that is not a user variable/value pair is ignored. Lines starting with # are treated as comments.

Executing `set_values_from_file()` within the Host process initializes user variables only in that process. Similarly, executing `set_values_from_file()` within a Site process initializes user variables only in that process. This operation can be modified, see Intercepting User Variables.

If the initialization file refers to a user variable which does not exist in the loaded test program (exact spelling, case sensitive) an error message similar to the following will be displayed in either the Host output window, or the Site output window(s).

```
Warning: Attempt to set undefined user-variable "myVar" from site
1 ignored.
```

`builtin_batch_file` serves the same purpose. It is one of the Built-in User Variables, and has predefined body code which calls `set_values_from_file()`.

It is invoked using `remote_set()` where the site argument determines where `set_values_from_file()` will be executed, and the value argument specifies the file. If the specified file is not valid a warning message is displayed in the output window of the specified site.

Usage

```
BOOL set_values_from_file( LPCTSTR filename );
```

where:

`filename` is the name of the text file to be read. The file must be accessible in a file system accessible to the test program executable program. If an absolute path is not specified,

`set_values_from_file()` will search for the file using the `PATH` environment variable. See Environmental Variables.

`set_values_from_file()` returns `TRUE` if the specified file exists otherwise `FALSE` is returned.

`builtin_batch_file` is used with `remote_set()`. When sent to a Site (`remote_set()` with site = 1, 2, etc.) the user variables will be initialized on the Site. When multiple Site are in use `remote_set()` must be called once for each site. When sent to the Host (`remote_set()` with site = 0) the user variables will be initialized on the Host. Correct operation requires that the *execute body code* argument to `remote_set()` be `TRUE`.

Example

This example contains the following parts. Each part has comments about important features:

- Program Code
- Host Values File
- Site Values File
- Host Output Window
- Site Output Window

Program Code

This code defines two user variables of each type (except `VOID`), each with an initial value and similar body code. When executed, the body code outputs the current value of the user variable. Below the user variables, a Host Begin Block and Site Begin Block both call `set_values_from_file()`, each specifying a different file name. Executing `set_values_from_file()` reads the specified file, sets the specified user variables to the value in the file, and executes the body code of each user variable.

Also included below is a `CONFIGURATION()` block which shows the how `remote_set()` is used to invoke `builtin_batch_file`. In this example the same two files are read and set the same variables to the same values. Using both methods in this way is redundant but does show proper syntax, etc.:

```
BOOL_VARIABLE(B1, TRUE, "") {
   output("B1 =[%s]", B1?"TRUE":"FALSE");
}
```

```
BOOL_VARIABLE(B2, TRUE, "") {
   output("B2 =[%s]", B2?"TRUE":"FALSE");
}
CSTRING_VARIABLE(S1, "?", "") {output("S1 =[%s]", S1);}
CSTRING_VARIABLE(S2, "?", "") {output("S2 =[%s]", S2);}
DOUBLE_VARIABLE(D1, -1, "") {output("D1 =[%5f]", D1);}
DOUBLE_VARIABLE(D2, -1, "") {output("D2 =[%5f]", D2);}
DWORD_VARIABLE(DW1, -1, "") {output("DW1 =[%d]", DW1);}
DWORD_VARIABLE(DW2, -1, "") {output("DW2 =[%d]", DW2);}
FLOAT_VARIABLE(F1, -1.0, "") {output("F1 =[%0.3f]", F1);}
FLOAT_VARIABLE(F2, -1.0, "") {output("F2 =[%0.3f]", F2);}
INT_VARIABLE(I1, -1, "") {output("I1 =[%d]", I1);}
INT_VARIABLE(I2, -1, "") {output("I2 =[%d]", I2);}
INT64_VARIABLE(I641, -1, "") {output("I641 =[%I64x]", I641);}
INT64_VARIABLE(I642, -1, "") {output("I642 =[%I64x]", I642);}
ONEOF_VARIABLE(O1, "A,B,C", "") {output("O1 =[%s]", O1);}
ONEOF_VARIABLE(O2, "D,E,F", "") {output("O2 =[%s]", O2);}
HOST_BEGIN_BLOCK( HBB1 ) { // See HOST_BEGIN_BLOCK()
   set_values_from_file ("c:/uVar_Host_values.txt");
}
SITE_BEGIN_BLOCK ( SBB1 ) { // See SITE_BEGIN_BLOCK()
   set_values_from_file ("c:/uVar_Site_values.txt");
}
CONFIGURATION( CB1 ) { // See CONFIGURATION()
   remote_set("builtin_batch_file","c:/uVar_Host_values.txt", 0);
   remote_set("builtin_batch_file","c:/uVar_Host_values.txt", 1);
}
```

Host Values File

This file, used to initialize the user variables in the Host process, is named *c:/uVar_Host_values.txt*. Note that each user variable name has a corresponding value, which is different than the value specified in the Program Code above.

```
B1: TRUE
B2: FALSE
S1: two words
S2: three word string
D1: 1234
D2: 5678
DW1: 9876
DW2: 5432
F1: 99.999
F2: -88.999
Int1: 13
Int2: -13
I64_1: 0xFFFFFFFFFFFFFFFF
I64_2: 0xEFFEEFFEEFFEEFFE
One1: B
One2: F
```

Site Values File

This file, used to initialize the user variables in each enabled Site process, is named *c:/uVar_Site_values.txt*. Note that each user variable name has a corresponding value, which is different than the value specified in the Program Code above.

```
B1: FALSE
B2: TRUE
S1: four word string here
S2: three words here
D1: 12345678
D2: 56789012
DW1: 98765432
DW2: 54321098
F1: 77.777
F2: -66.666
Int1: 19
Int2: -19
I64_1: 0xA55AA55AA55AA55A
I64_2: 0xABCDEF0FEDCBA0FE
One1: C
One2: E
```

Host Output Window

This is the output generated by executing the user variable body code in the Host process:

```
B1 =[TRUE]
B2 =[FALSE]
S1 =[two word]
S2 =[three word string]
D1 =[1234.000000]
D2 =[5678.000000]
DW1 =[9876]
DW2 =[5432]
F1 =[99.999]
F2 =[-88.999]
I1 =[13]
I2 =[-13]
I641 =[ffffffffffffffff]
I642 =[effeeffeeffeeffe]
O1 =[B]
O2 =[F]
```

Site Output Window

This is the output generated by executing the user variable body code in a Site process:

```
B1 =[FALSE]
B2 =[TRUE]
S1 =[four word string here]
S2 =[three words here]
D1 =[12345678.000000]
D2 =[56789012.000000]
DW1 =[98765432]
DW2 =[54321098]
F1 =[77.777]
F2 =[-66.666]
I1 =[19]
I2 =[-19]
I641 =[a55aa55aa55aa55a]
I642 =[abcdef0fedcba0fe]
O1 =[C]
O2 =[E]
```

7.1.8 Modifying ONEOF Variables

Definition

The function `set_choices()` allows user C-code to programmatically modify the comma separated list of values associated with a `ONEOF_VARIABLE`.

This is desirable to modify the list of items displayed in User Dialogs Combo boxes or List boxes, which are linked to a `ONEOF_VARIABLE`. Both the `ONEOF` user variable and the dialog resource are updated by `set_choices()`.

The modification can replace the entire previous list of comma separated values or, optionally, append to the existing list.

Definition

```
BOOL set_choices( VariableProxy Oneof_Variable,
                  CString choices,
                  BOOL append DEFAULT_VALUE( FALSE ) );
BOOL set_choices( VariableProxy Oneof_Variable,
                  CStringArray &choices,
                  BOOL append DEFAULT_VALUE( FALSE ) );
```

where:

`Oneof_Variable` identifies the `ONEOF_VARIABLE` user variable to be modified.

`choices` is a `CString` value, which can be a single value or a comma separated list of values.

`append` determines whether the existing list is replaced (`FALSE`) or appended to (`TRUE`).

Example

The following code creates a `ONEOF_VARIABLE` with an initial list of three color values. The User Dialog code associates this `ONEOF` variable with a dialog resource:

```
ONEOF_VARIABLE(SM_listbox, "Orange, Yellow, Green", "Colors" ) {}
DIALOG( simple_dialog ) {
   CONTROL( IDC_SM_listbox, SM_listbox ) // Tied to dialog List Box
}
```

The original list displayed in the dialog will show *Orange, Yellow, Green*. The code below modifies both the list displayed in the dialog, and the list of choices associated with the underlying ONEOF_VARIABLE:

```
set_choices( SM_listbox, "Red, White, Blue");
```

7.1.9 Intercepting User Variables

See User Variables.

Description

User Variables allow one site (process) to transfer data to/from another site and optionally to cause user code to be executed on that site, by invoking the User Variable's body code. This is explicitly done using the remote_send(), remote_fetch(), remote_set() and remote_get() functions.

The intercept() function causes most operations on a specified User Variable on a specific site to be automatically forwarded to the site that calls intercept(). This includes transactions caused by the remote_xxx() functions noted above and transactions invoked less directly (more below).

The remote_send(), remote_fetch(), remote_set() and remote_get() functions each require an explicit site number argument, to identify the remote site. There are four categories of sites:

- UI (site number = -1)
- Host process (site number = 0)
- Site processes (site number = 1 through 40)
- User Tools (site number = > 1024, randomly assigned)

When the intercept() function is executed by a given site it identifies, using the site argument, which site is being intercepted. For example, given the following User Variable declaration:

```
CSTRING_VARIABLE( myVar, "abc", "") {}
```

The site which executes the following code will receive all remote transactions to myVar sent to the Host (site number = 0):

```
intercept( myVar, 0 );
```

The site which executes the following code will receive a copy of remote transactions to `myVar` sent to the Host (site number = 0). The Host process will also receive the transaction:

```
intercept( myVar, 0, TRUE ); // Note the tee argument = TRUE
```

And, the site which executes the following code will receive all remote transactions to `myVar` sent to UI (site number = -1):

```
intercept( myVar, -1 );
```

This latter example reflects a mainstream application, more below.

The `fumble()` function is used restore normal remote command operation for a specified User Variable.

Note that the site numbers of Host, Site and UI are constant, making the use of the `remote_xxx()` functions very straight forward. However, the site number for User Tools is randomly assigned, which normally would require that each Host/Site process which needed to communicate with a given User Tool somehow get the site number for that tool.

Alternatively, the User Tool can use `intercept()` to register key User Variables with UI. Subsequently, any site (process) which needs to communicate with that User Tool will still use the `remote_xxx()` functions but will specify UI as the target site (-1). When UI receives the transaction, it will forward it to the process which *intercepted* the User Variable; i.e. the User Tool.

For example, the following User Tool code intercepts the `myVar` User Variable when related remote transactions are sent to UI (site number = -1):

```
TOOL_BEGIN_BLOCK( TB1 ) {
   intercept( myVar, -1 );
}
```

Subsequently, any site (including a User Tool) may execute the following code to communicate with the User Tool:

```
remote_set( myVar, "abc", -1 );
```

Note that the remote command specifies UI (-1) as the target site, but since `myVar` has been intercepted the transaction will be forwarded to the User Tool.

Also note:

- Any Host, Site or User Tool (process) can use `intercept()`.
- The `site` argument identifies the site on which the specified User Variable will be intercepted; i.e. the site which will no longer receive the remote transaction but will instead forward it to the site which executed the `intercept()` function.

- The optional `tee` argument causes the transaction to be tee'ed to the process which executed `intercept()`. When `tee` is FALSE only the process which executed `intercept()` receives the remote transactions to the specified User Variable. When `tee` is TRUE, both the process which executed `intercept()` and the specified `site` receive the transaction; i.e. `site` still gets notified.
- As indicated above, `intercept()` changes the site/process which receives transactions related to the specified User Variable. This mechanism affects the `remote_xxx()` functions noted above, but also affects the following, less obvious, transactions to intercepted User Variables:
  - A simple value assignment to an intercepted User Variable only affects the local variable; i.e. `intercept()` has no effect.
  - When a User Variable is associated with a dialog using `IMMEDIATE_CONTROL()` and that dialog control is invoked the transaction will be sent to the intercepted User Variable.
  - Using `invoke()` to explicitly invoke a User Variable's body. See Invoking User Variable Body Code.
  - `set_values_from_file()` and `builtin_batch_file` will affect intercepted User Variable.
  - Note `intercept()` will affect User Variables associated with User Dialogs, however, it is not expected that `intercept()` will target these User Variables. If the variable in question is intercepted, then `update_variable()`, `update_variables()` will change the remote value, and `update_control()` and `update_controls()` will apply the remote value to the control(s).

Note: different versions of `intercept()` and `fumble()` are used to redirect output messages. See Redirecting Output Messages.

Usage

```
BOOL intercept(VariableProxy v,
               int site DEFAULT_VALUE( -1 ),
               BOOL tee DEFAULT_VALUE( FALSE ) );
BOOL intercept(Snapshot *ss,
               int site DEFAULT_VALUE( -1 ),
               BOOL tee DEFAULT_VALUE( FALSE ) );
BOOL fumble( VariableProxy v, int site DEFAULT_VALUE( -1 ) );
```

where:

v identifies the target User Variable.

site is optional and, if used, specifies the site one which the target User Variable is being intercepted. Default = -1 = UI.

tee is optional and, if used, specifies whether the specified **site** also receives remote transactions (TRUE) or whether only the site which intercepted the variable receives the transaction (FALSE, default).

ss identifies a snapshot, which identifies one or more user variable(s). See SNAPSHOT().

intercept() return TRUE if the operation is successful, otherwise FALSE is returned.

fumble() return TRUE if the specified site was intercepting the specified user variable, otherwise FALSE is returned. This operation was first available in software release h2.0.xx.

Example

```
if( ! intercept( myUvar ) )
   output("ERROR: intercept( myUvar, -1 ) returned FALSE");
```

7.1.10 Built-in User Variables

See User Variables.

Description

The following three built-in User Variables are automatically part of every test program:

- builtin_what_exe : identify the software release in use or the current test program name and location.
- builtin_dynload : load a Dynamic Link Library (DLL). Documented in Loading DLLs.
- builtin_batch_file : initialize User Variables from a text file. This can also be done using set_values_from_file(). Both are documented in User Variable Text File Initialization.

7.1.10.1 builtin_what_exe

See Built-in User Variables.

Description

Using `builtin_what_exe` plus `remote_get()` it is possible to get, as a `CString`, the following:

- The complete *disk:\path\filename* of the currently executing UI, which normally indicates the software release in use. For example:

  `C:\Nextest\v2.10.15\Bin\Ui.exe`

- The current *disk:\path\filename* of the test program.

Either can be used for informational purposes, or to enable code to conditionally execute functions only available after a specific software release. See Example.

Also see Retrieving the Nextest Software Version.

Usage

`builtin_what_exe` is used with `remote_get()`. When `builtin_what_exe` is retrieved from the UI process (`remote_get()` with site = -1) the path to UI is returned. When `builtin_what_exe` is retrieved from the Host or any Site process (`remote_get()` with site = 0, 1, 2, etc.) the path to the test program is returned.

Example

This is an example using `builtin_what_exe` to display the path to UI and the path to the executing test program. A conditional statement is also shown which checks to see if the current software release is v2.10.15. Note that correct operation depends upon a typical software installation where the release number is included in the path to UI.

```
// Get builtin_what_exe from UI (-1) to locate release
CString release = remote_get("builtin_what_exe", -1 );
output(" Release location => %s", release );

// Determine if the release in use is v2.10.15
if (release.Find("v2.10.15") != -1)
   output("  Release used is v2.10.15");
else
   output("  Release used is NOT v2.10.15");

// Get builtin_what_exe from Host/Site to locate test program
CString program = remote_get("builtin_what_exe", 0 ); // 0= Host
output(" Program location => %s", program );
```

7.1.10.2 Loading DLLs

See Built-in User Variables.

Description

Using `builtin_dynload` plus `remote_set()` the test program and/or User Tools can cause the a Dynamic Link Library (DLL) to be loaded. This allows program independent User Tools to, for example, cause the test program to load a DLL containing User-defined User Variables which are required by the tool but are not normally included in the test program. Or, a test program can load a standard library of functions developed and maintained independent of the program.

Using `builtin_dynload` requires that the location of the DLL be specified. Two options exist:

- Relative path: put the DLL in any location listed in the `PATH` environment variable. See Environmental Variables. In this scenario, no path is specified in the value assigned to `builtin_dynload`. The first location checked is always relative to the test program executable (*.exe*). Other locations can vary depending on which Nextest software release is being used and user modifications to the `PATH` environment variable.
- Absolute path: put the DLL in any location and specify the complete path to it in the value assigned to `builtin_dynload`.

When using multi-site systems, the `PATH` environment variable on the Site computers does not exactly match that on the Host, and environment variables set on the Host are not automatically configured on the Sites. This can be addressed using the RBoot Client File.

Usage

`builtin_dynload` is used with `remote_set()`. When sent to a Site (`remote_set()` with site = 1, 2, etc.) the site process will load the DLL. When multiple sites are in use `remote_set()` must be called once for each site. When sent to the Host (`remote_set()` with site = 0) the Host process will load the DLL. Correct operation requires that the *execute body code* argument to `remote_set()` be TRUE.

Example

An example of a User Tools using `builtin_dynload` to cause the test program to load a DLL is located at Example 3:.

7.1.10.3 RBoot Client File

See Loading DLLs.

Description

As indicated in Loading DLLs, when using multi-site systems, the PATH environment variable on the Site computers does not exactly match that on the Host. For example, the Site computers see the Host computer's `C:\` partition as `Y:\`. And, any user defined environment variables set on the Host are not automatically configured on the Sites.

To address this, a file in the Host computer's *C:\rboot\server* folder, named *client* (with no file name extension) can be modified to set environment variables and/or execute commands on the site controllers (only). This happens each time the Site computer reboots. The user manually edits the *client* file to add commands using the syntax noted below.

Note the following:

- First available in rbootHD version 1.13. The version is seen when *Startserver* is executed during system initialization.
- One environment variable or command per line.
- # symbol is used for comments.
- The syntax noted below also outlined in the default *C:\rboot\server\client* file.
- The user is responsible for ensuring environment variables match (when necessary) between the Host and Site(s).

Usage

To define an environment variable

```
ENV=variable:value
```

To define a command:

```
EXEC=command
```

where:

`variable` is the environment variable name.

`value` is the value to assign to variable.

`command` is the statement to execute via the Site's command shell. Quote the command string as necessary for proper command shell execution.

Example

The following example sets the environment variable named `MYDLL` to the location of the target DLL:

```
ENV=MYDLL:D:/myLibs/mySpecialDLL
```

7.1.11 UI User Variables

Overview

A number of built-in User Variables are available to provide predefined functionality which is accessible via the user variable. In general, UI user variables can be used to:

- Set selected values, modes, or attributes when UI - User Interface is started from a Windows command line or batch file
- Allow user-written C-code to communicate with UI to set, or get, selected values, modes, or attributes, or to command UI to perform some function. The C-code can be in a test program and/or User Tools.
- Allow UI to provide notification of key events, which will execute user C-code in the test program and/or User Tools. This is done using Callback UI User Variables.

UI User Variable Scope is discussed separately, and is key to understanding operation when, within a single invocation of UI, multiple test programs are loaded and unloaded.

UI User Variables Categories describes how UI user variables share attributes and similarities.

UI User Variables, Alphabetical Listing provides a brief overview of each variable, its type, category, etc.

UI Call-back User Variables is a separate table documenting which processes are notified by UI.

7.1.11.1 UI User Variable Scope

One important characteristic of all UI user variables which have a value or state is that the scope of the variable is the UI process. This is true even when UI subsequently uses the variable value to influence a test program.

For many UI user variables, test program and User Tools C-code can change its value or state. When these changes are sent to UI (the normal practice) the act of unloading a test program does not further change the state of the variable in UI. Thus, during the same invocation of UI, the next program loaded will see the modified variable value.

Some UI user variables have Reload support, which can be used to un-do changes made by test program or User Tools code, but only when the variable was originally set from a Windows command line. See Reload.

7.1.11.2 UI User Variables Categories

UI user variables fall into several categories, which are based on the following:

- The purpose of each variable. There are three general applications:
  - Set and/or Get a value, state, or attribute
  - Send a command to UI
  - Callback UI User Variable
- How the variable may be used:
  - Command line
  - Batch file
  - user-written C-code

  Some UI user variables are only useful in some of these contexts, while others can be useful in all.
- The type of the variable: `void`, `int`, `BOOL`, `CString`, etc. This is important to set and/or get an attribute value, and to send a command to UI. Some Callback UI User Variables also deliver information of a particular type when invoked.
- Whether the variable value is persistent, via the *Ui.INI* file. This attribute is noted in the detailed documentation of each variable.

- Whether the variable has Reload support, which can be important when using command line methods. This attribute is noted in the detailed documentation of each variable.

Table 7.1.11.2-1 UI User Variables, Alphabetical Listing

Variable Type	VariablerName	Variable Purpose	Remarks
CString	ui_BatchFile	Set attribute	Used to specify a batch file on a command-line or within another batch file.
BOOL	ui_BitmapCrossHair	Set or get attribute	Set the initial crosshair display mode.
BOOL	ui_BitmapDialogDecMode	Set or get attribute	Sets initial BitmapTool coordinate display mode to Decimal. Or, gets the current mode.
int	ui_BitmapDisplay	UI command	Invokes BitmapTool, or if already running, updates the display.
CString	ui_BitmapDisplayMode	Set or get attribute	Set the initial display mode to update the whole display or only the visible display.
BOOL	ui_BitmapDisplaySeparateZoomWindow	Set or get attribute	Enable or disable whether the full view and zoom windows are displayed separately. See BitmapTool Separate Window Option.
BOOL	ui_BitmapDisplayTotalCount	Set or get attribute	Enable or disable whether the total fail count is displayed. See Fail Count Enable Controls.

Table 7.1.11.2-1 UI User Variables, Alphabetical Listing *(Continued)*

Variable Type	VariablerName	Variable Purpose	Remarks
BOOL	ui_BitmapDisplayVisibleCount	Set or get attribute	Enable or disable whether the visible fail count is displayed. See Fail Count Enable Controls.
int	ui_BitmapdutNo	Set or get attribute	In parallel test applications, sets BitmapTool to display failures from one specified DUT.
int	ui_BitmapFailColor, ui_BitmapPassColor	Set or get attribute	Programmatically set the two colors used to represent Pass/ Fail in BitmapTool.
_int64	ui_BitmapMainSize	Set or get attribute	Programmatically set the size of the BitmapTool full view window.
int	ui_BitmapMaxErrors	Set or get attribute	Limits the number of failures displayed by BitmapTool.
_int64	ui_BitmapMoveTo	UI command	Moves zoom selection box to a specified row/column/data position in the main BitmapTool display.
int	ui_BitmapPageHScroll, ui_BitmapPageVScroll, ui_BitmapLineHScroll, ui_BitmapLineVScroll	Set or get attribute	Set the initial values for Page and Line horizonal and vertical scroll size.
BOOL	ui_BitmapPan	Set or get attribute	Set the state of the zoom/pan option.
int	ui_BitmapRowsChunk	Set or get attribute	Maximum number of rows read from the ECR in a single chunk.

Table 7.1.11.2-1 UI User Variables, Alphabetical Listing *(Continued)*

Variable Type	VariablerName	Variable Purpose	Remarks
BOOL	ui_BitmapRulers	Set or get attribute	Set the initial ruler display mode.
_int64	ui_BitmapTotalFailBitCount	Callback UI User Variable	If defined, executes instead of the normal ECR scan routine used to obtain the total fail bit count value.
CString	ui_BitmapTotalVisibleFailBitString	Callback UI User Variable	If defined, executes any time the BitmapTool Total/ Visible fail values are updated, but only if BitmapTool main and zoom displays are not separate. Used to modify the string displayed.
CString	ui_BitmapTotalFailBitString	Callback UI User Variable	If defined, executes any time the BitmapTool Total fail values are updated, but only if BitmapTool's main and zoom displays are separate. Used to modify the string displayed.
CString	ui_BitmapVisibleFailBitString	Callback UI User Variable	If defined, executes any time the BitmapTool Visible fail values are updated, but only if BitmapTool's main and zoom displays are separate. Used to modify the string displayed.

Table 7.1.11.2-1 UI User Variables, Alphabetical Listing *(Continued)*

Variable Type	VariablerName	Variable Purpose	Remarks
_int64	ui_BitmapVisibleSize	Set or get attribute	Programmatically set the size of the BitmapTool zoom window.
BOOL	ui_BitmapZoom2	Set or get attribute	Set the initial state of the power of 2 zoom option.
CString	ui_BreakPointFile	Set or get attribute	Specify the path/file name of a Breakpoint Definition File to load during test program load.
void	ui_BreakPointRemoveAll	UI command	Clears all breakpoints currently set via the Breakpoint Monitor.
BOOL	ui_ClearAtProgramLoad	Set or get attribute	If set, UI output windows are cleared before test program load starts.
BOOL	ui_ClearAtTestStart	Set or get attribute	If set, UI output window is cleared every time start test is issued
void	ui_Close	UI command	Same as clicking on *Close* in the UI *File* Menu
BOOL	ui_CloseAfterRun	Set or get attribute	If set, test program is unloaded after executing the Sequence & Binning Table ui_RunTestProgram times
CString	ui_Controller	Set attribute	Modify UI Controller list from command line or batch file.

Table 7.1.11.2-1 UI User Variables, Alphabetical Listing *(Continued)*

Variable Type	VariablerName	Variable Purpose	Remarks
CString	ui_CurrentBitmapScheme	Callback UI User Variable	Invoked by UI any time a new Bitmap scheme is selected in BitmapTool.
BOOL	ui_DbmDialogDecMode	Set or get attribute	Used to set the Hex/Decimal display mode for DBMTool.
BOOL	ui_DutBoardStatusCheckDisable	Set attribute	Used to disable automatic DUT Board status checking.
BOOL	ui_ECRDialogDecMode	Set or get attribute	Used to set the Hex/Decimal display mode for ECRTool.
BOOL	ui_EngineeringMode	Set or get attribute	Enable/disable Engineering Mode.
int	ui_ExcelAppEvent	Callback UI User Variable	Invoked by UI when it receives an event from Excel
void	ui_Exit	UI command	Same as clicking on *Exit* in the UI *File* Menu
BOOL	ui_ExitAfterRun	Set or get attribute	If set, terminate UI after running the test program ui_RunTestProgram times
ONEOF_VARIABLE	ui_HideTool	UI command	Used to hide many of UI's Interactive Tools from user C-code.
BOOL	ui_HostDebug	Set or get attribute	Enable/disable running host debug mode.
CString	ui_HostModeCommandLine	Set or get attribute	Specify command line options which will affect Host process only.

Table 7.1.11.2-1 UI User Variables, Alphabetical Listing *(Continued)*

Variable Type	VariablerName	Variable Purpose	Remarks
int	ui_HostTimeOut	Set or get attribute	Specify Host timeout value from command line.
int	ui_LoadTimeOut	Set or get attribute	Specify program load timeout value from command line.
DWORD	ui_LoadedMask	Get UI attribute	Identify which sites have a loaded test program.
int	ui_MonitorPort	Set or get attribute	Set the monitor port value from a command line.
int	ui_MonitorTimeOut	Set or get attribute	Set the monitor timeout value from command line.
BOOL	ui_NoLogo	Set or get attribute	Enable/disable UI logo during UI start from a command line.
CString	ui_Open	UI command	Specify a test program to load from a User Tools.
BOOL	ui_OutputAutoOpen	Set UI attribute	Control whether a closed output window will open when a new message is sent.
CString	ui_OutputFile	Set or get attribute	Specify a path/ filename to log UI output messages.
CString	ui_OutputFormat	Set or get attribute	Control a prefix to the default output() format
BOOL	ui_OutputOpen	Set UI attribute	Open or close UI's output window.
void	ui_ProgLoaded	Callback UI User Variable	Host and User Tools are notified when all enabled sites have loaded a test program

Table 7.1.11.2-1 UI User Variables, Alphabetical Listing *(Continued)*

Variable Type	VariablerName	Variable Purpose	Remarks
void	ui_ProgUnloaded	Callback UI User Variable	User Tools are notified when a test program is closed.
CString	ui_ResourceInitialized	Callback UI User Variable	Host and User Tools are notified as each resource is initialized on each enabled site.
int	ui_RunTestProgram	UI command	Send Start Test to invoke the Sequence & Binning Table in all enabled sites
CString	ui_ShmooDone	Callback UI User Variable	Sites are notified when a search/shmoo execution has completed. Useful with search_results_get().
CString	ui_ShmooInput	Set or get attribute	Specify the path/file name of a Shmoo/ Search definition file to load. See Shmoo Definition File
CString	ui_ShmooOutputFile	Set or get attribute	Specify the path/file name of a file used to store Shmoo/Search output.
int	ui_ShowOutputTab	UI command	Selects which tab is displayed in UI's output window.
BOOL	ui_Show	Callback UI User Variable	Added to user tools managed by ToolLauncher to enable ui_HideTool and ui_ShowTool to control the tool display state.

Table 7.1.11.2-1 UI User Variables, Alphabetical Listing *(Continued)*

Variable Type	VariablerName	Variable Purpose	Remarks
ONEOF_VARIABLE	ui_ShowTool	UI command	Used to invoke many of UI's Interactive Tools from user C-code.
void	ui_ShutDown	UI command	Same as clicking on *Exit* command in the UI *File* Menu
DWORD	ui_SiteDebug	Set or get attribute	Site mask for starting sites in Developer Studio debug mode.
CString	ui_SiteDone	Callback UI User Variable	Host and User Tools are notified when Sequence & Binning Table execution has completed on a given site.
int	ui_SiteLoaded	Callback UI User Variable	Host and User Tools are notified when each site completes loading a test program
int	ui_SiteMask	Set or get attribute	Used to set or get the enabled site mask.
CString	ui_SiteModeCommandLine	Set or get attribute	Specify command line options which will affect Site processes only.
int	ui_SiteUnloaded	Callback UI User Variable	Host and User Tools are notified when each enabled site unloads a test program.
void	ui_StartTest	UI command	Same as clicking on *Start Testing* command in the UI *File* Menu.

Table 7.1.11.2-1 UI User Variables, Alphabetical Listing *(Continued)*

Variable Type	VariablerName	Variable Purpose	Remarks
CString	ui_StartTool	UI command	Used to invoke User Tools from test program C-code (or from another tool)
void	ui_StopTest	UI command	Same as clicking Stop Testing in the UI *File* Menu
void or CString	ui_TestDone	Callback UI User Variable	Host and User Tools are notified when Sequence & Binning Table execution has completed on all enabled sites.
CString	ui_TestProgConfiguration	Set or get attribute	Used to specify the name of the CONFIGURATION() block to be used in the next program loaded.
CString	ui_TestProgDirPath	Set or get attribute	Use to set the initial folder location seen when File Open is invoked in UI.
CString	ui_TestProgName	Set or get attribute	Used from a command line or batch file to specify the path/name of the test program to load
void	ui_TestStarted	Callback UI User Variable	Host and User Tools are notified when UI sends the Start Test signal to the sites.
CString	ui_TimingToolPinLists	Set or get attribute	Set, modify, or get which pin lists appear in TimingTool.
void	ui_ToolLoaded	Callback UI User Variable	User Tools are notified when another tool is started.

Table 7.1.11.2-1 UI User Variables, Alphabetical Listing *(Continued)*

Variable Type	VariablerName	Variable Purpose	Remarks
CString	ui_ToolModeCommandLine	Set or get attribute	Specify command line options which will affect Tool processes only.
void	ui_ToolUnloaded	Callback UI User Variable	User Tools are notified when another tool is terminated
BOOL	ui_UserVarSiteMode	Set or get attribute	Set the initial state of the Host/ Controller radio buttons in User Variables Tool.
int	ui_UserVariableTimeout	Set or get attribute	Used to modify the default timeout value applied when executing user variable body code

7.1.11.3 Callback UI User Variable

The UI user variables shown in the table below provide call-back functionality at key times during test program operation.

In general, if any of these user variables are defined in user-written C-code UI will cause the body code to be executed based on a particular event. As noted in the table, some variables

are invoked only in User Tools processe(s) while others are invoked in both Host and user tool processe(s).

Table 7.1.11.3-1 UI Call-back User Variables

Variable	Executed in Process...			Notes
	Host	Site	User Tool	
ui_ProgLoaded	X		X	Notification that all sites have reported program load is complete.
ui_ProgUnloaded			X	Notification that the test program is unloaded.
ui_ResourceInitialized	X			Notification of each resource loaded from all enabled sites
ui_ShmooDone		X		Notification that a search/ shmoo execution has completed.
ui_SiteDone	X		X	Notification that a given site has completed execution of the Sequence & Binning Table.
ui_SiteLoaded	X		X	Notification that a site has reported program load is complete.
ui_SiteUnloaded	X		X	Notification that a site has unloaded the test program.
ui_TestDone	X		X	Notification that all enabled sites have reported test execution is completed.

Table 7.1.11.3-1 UI Call-back User Variables *(Continued)*

Variable	Executed in Process... Host	Site	User Tool	Notes
ui_TestStarted	X		X	Notification that UI has sent the Start Test signals to the sites.
ui_ToolLoaded			X	Notification that a User Tools was started. Used to signal other User Tools.
ui_ToolUnloaded			X	Notification that a User Tools was terminated. Used to signal other User Tools.

7.1.11.4 Reload

As noted in the UI User Variable Scope, some UI user variables are targeted for use from a Windows shell command line. In these situations, the value assigned to a given UI user variable is typically important to desired program operation, to set key values, modes, etc.

In most cases, these same UI user variables can be modified from the test program C-code, or from code in separate User Tools. And, as also noted in the UI User Variable Scope, the scope of all UI user variables is UI, not the loaded test program. Thus, the *Reload* option.

The *Reload* mechanism becomes useful when, within the same invocation of UI, a test program is unloaded (closed) and another test program is loaded. If the first program modifies a UI user variable value which was set from a command line with the Reload option, the original value is restored before the 2nd program loads.

The following sequence shows the effect of using and not using the Reload option:

- A command line or batch file is used to start UI and set the following UI user variables:
  - `ui_ClearAtTestStart=1` and the Reload option is not set
  - `ui_OutputFile=c:\tmp\filename.txt`, and the Reload option is set

- Test program #1 is loaded (how doesn't matter), and modifies the values of both UI user variables noted above. Note that User Tools can also modify these variables.
- Test program #1 is unloaded, and test program #2 is loaded (or #1 is loaded again, it doesn't matter).
- At this point, the value of `ui_ClearAtTestStart` is the modified value, whereas the value of `ui_OutputFile` is the original value, as set from the command line.

Also note the following:

- Reload is enabled by adding the '**R**' character to the long form of the command syntax. For example, the following line does NOT enable Reload:
  `ui /U:ui_ClearAtTestStart=1`
  The following line adds the **R** and does enable Reload:
  `ui /UR:ui_ClearAtTestStart=1`
- Not all UI user variables have Reload support. This capability is noted in the detailed documentation of each UI user variable.
- Reload state is independent for each supported UI user variable i.e. it is possible for some UI user variables to be specified with Reload enabled while others have Reload disabled.
- Reload is only enabled when the UI user variable is set from a command line, not from a batch file or from user C-code.
- The Reload value is persistent only to the current UI invocation i.e. terminating UI also terminates the Reload value.

7.1.11.5 Paths, File Names, Default Extensions, etc.

UI User Variables used to specify a file name use the following conventions:

- When a full path is not specified the environmental `PATH` will be searched for the specified file. Shmoo Definition Files (specified using `ui_ShmooInput`) do not follow this rule (see below).
- File Name Default Extensions
  - Test program name **`.exe`**
  - Batch file **`.txt`** (*not* .bat)
  - Shmoo input file name **`.txt`**
- Using `ui_ShmooInput`, if a full path is not specified only the Program Working Directory is checked i.e. the `PATH` is ***NOT*** checked.

7.1.11.6 ui_BatchFile

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BatchFile`.

Description

This UI user variable is used to specify a batch file to be executed when starting UI.

`ui_BatchFile` is only usable from a Windows command line, or from within a batch file.

The value assigned to this UI user variable is the file name of an ASCII text file with an optional path to the file.

The batch file has the following rules:

- It can only contain UI variable assignments.
- Each entry must be on a separate line.
- The # character can be used to add a comment to the end of a line. There are no multi-line comment facilities. Using two # will add a single # to the line; i.e. ## is not a comment.
- Only the *long form* of UI variable syntax is legal. This form is shown in the separate documentation of appropriate UI user variables.
- A batch file can invoke another batch file.
- If the path is not specified an attempt is made to locate the file using the `PATH` environment variable. See Environmental Variables. If the specified file is not found a warning is displayed, and command line execution continues.
- When `ui_BatchFile` is used within a batch file, if the file is not found a warning is displayed, and the batch file execution continues.

The value is NOT persistent after UI is terminated (via the UI.ini file).

Reload does not apply to this variable.

Usage

```
CSTRING_VARIABLE( ui_BatchFile, "", "" )
```

From command line:

```
Long form:   /U:ui_BatchFile=<filename>
Short form:  /BATCH=<filename>
```

From within a batch file:

```
ui_BatchFile=<filename>
```

where **<filename>** specifies the path/file-name of the batch file.

Example

These two examples operate identically when executed at a Windows command line:

```
C:\> ui /U:ui_BatchFile=c:\test_batch.txt
C:\> ui /BATCH=c:\test_batch.txt
```

Example batch file contents:

```
ui_NoLogo=1
ui_EngineeringMode=1
ui_ClearAtTestStart=1
```

7.1.11.7 ui_BitmapCrossHair

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapCrossHair`.

Description

This UI user variable is used to set the initial state of BitmapTool's cross hair option which is also controllable using the BitmapTool Control Dialog.

`ui_BitmapCrossHair` is only effective if sent to UI before BitmapTool has been started. It was created to allow User Dialogs or User Tools to modify the default state of the BitmapTool cross hair option. The default mode is FALSE (disabled).

Usage

Intended usage is via `remote_set()` and `remote_get()` from within user-written C-code. The variable is only valid when sent to UI (site = -1). Setting the value to TRUE enables the cross hairs when BitmapTool is started. FALSE disables the cross hairs.

Example

The following example will enable the cross hairs when BitmapTool is next invoked:

```
remote_set( "ui_BitmapCrossHair", TRUE, -1 ) // remote_set()
```

The following example retrieves and outputs the current default state:

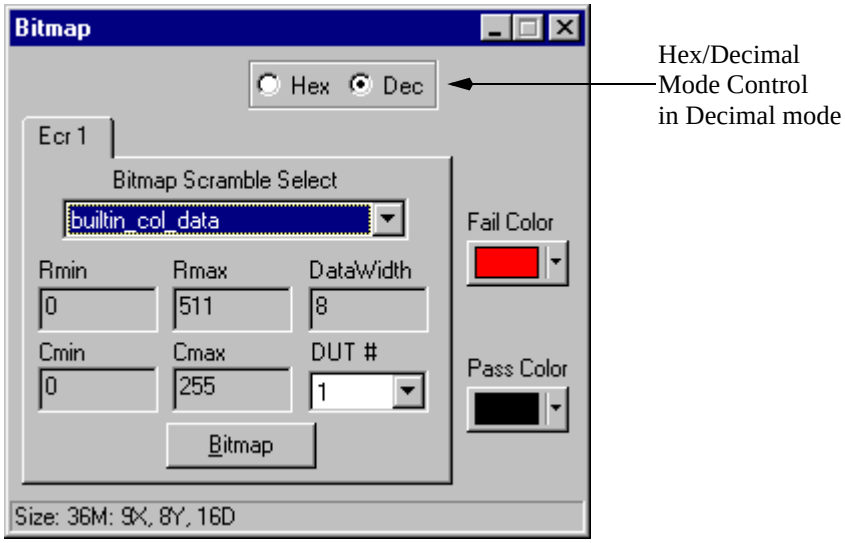
```
CString val = remote_get(  "ui_BitmapCrossHair", -1 );
output("  State => %s", val );
```

7.1.11.8 ui_BitmapDialogDecMode

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapDialogDecMode`.

Description

This UI user variable is used to set the initial state of BitmapTool's hex/decimal coordinate display mode control, which is also controllable using the BitmapTool Control Dialog::



`ui_BitmapDialogDecMode` is only effective if sent to UI before BitmapTool has been started. It was created to allow User Dialogs or User Tools to modify the default state of the BitmapTool hex/decimal mode attribute. The default mode is hexadecimal.

Usage

Intended usage is via `remote_set()` and `remote_get()` from within user-written C-code. The variable is only valid when sent to UI (site = -1). Setting the value to TRUE sets the initial

state of BitmapTool's Hex/Decimal coordinate display mode control to decimal. FALSE sets the initial state to hex.

Example

The following example will cause the Hex/Decimal mode to be decimal when BitmapTool is next invoked:

```
remote_set( "ui_BitmapDialogDecMode", TRUE, -1 ) // remote_set()
```

The following example retrieves and outputs the current default state:

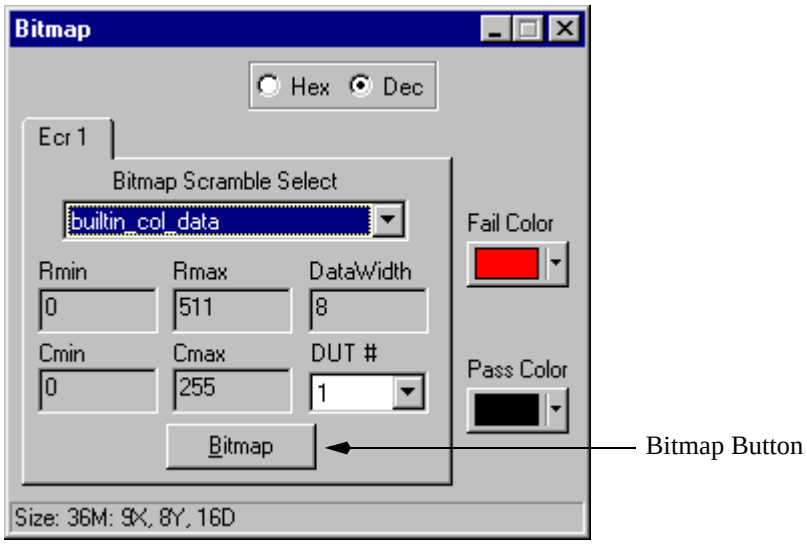
```
CString val = remote_get(  "ui_BitmapDialogDecMode", -1 );
output("  State => %s", val );
```

7.1.11.9 ui_BitmapDisplay

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapDisplay`.

Description

This UI user variable is used to invoke BitmapTool or, if already running, to update the display by re-reading the ECR. This has the same effect as clicking the Bitmap button in the BitmapTool control panel:



This UI user variable was created to allow User Dialogs or User Tools to invoke BitmapTool or update its display.

Usage

Intended usage is via `remote_set()` from within user-written C-code. The variable is only valid when sent to UI (site = -1). The value set to this variable specifies which site is to have BitmapTool invoked/updated. Using a single site system (PT) this value will be 1.

Example

The following command will cause BitmapTool to be invoked, or updated, using data from the ECR in site-1:

```
remote_set( "ui_BitmapDisplay", 1, -1 ); // remote_set(), -1 = UI
```

7.1.11.10 ui_BitmapDisplayMode

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapDisplayMode`.

Description

This UI user variable is used to set the default state of BitmapTool's display mode.

To simplify future enhancements `ui_BitmapDisplayMode` is a `CSTRING_VARIABLE`. The legal values are (case insensitive):

```
"entire"
"entire-xl"
"visible"
```

See BitmapTool Display Mode.

`ui_BitmapDisplayMode` is only effective if sent to UI before BitmapTool has been started. It was created to allow User Dialogs or User Tools to modify the default state of the BitmapTool update mode. The default mode is `"entire"`.

Usage

Intended usage is via `remote_set()` and `remote_get()` from within user-written C-code. The variable is only valid when sent to UI (site = -1).

Example

The following example will set the default update mode when BitmapTool is next invoked:

```
remote_set( "ui_BitmapDisplayMode", "visible", -1 )
```

The following example retrieves and outputs the current default state:

```
CString val = remote_get(  "ui_BitmapDisplayMode", -1 );
output("  State => %s", val );
```

7.1.11.11 ui_BitmapDisplaySeparateZoomWindow

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapDisplaySeparateZoomWindow`.

Description

This UI user variable is used to set the default state of BitmapTool's seperate zoom display option. See BitmapTool Separate Window Option

Setting the value to TRUE causes BItmapTool to use separate displays. Setting the value to FALSE causes BItmapTool to use the original joined displays.

`ui_BitmapDisplaySeparateZoomWindow` is only effective if sent to UI before BitmapTool has been started. It was created to allow User Dialogs or User Tools to modify the default state of the BitmapTool seperate zoom display option. The default mode is FALSE.

Usage

Intended usage is via `remote_set()` and `remote_get()` from within user-written C-code. The variable is only valid when sent to UI (site = -1).

Example

The following example will cause separate displays when BitmapTool is next invoked:

```
remote_set(  "ui_BitmapDisplaySeparateZoomWindow",
             TRUE,
             -1,
             TRUE,
             INFINITE );
```

The following example retrieves and outputs the current state of this variable:

```
BOOL val = remote_get( "ui_BitmapDisplaySeparateZoomWindow", -1 );
output("  State => %s", val ? "TRUE" : "FALSE" );
```

7.1.11.12 ui_BitmapDisplayTotalCount

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapDisplayTotalCount`.

Description

This UI user variable is used to set the default state of BitmapTool's display total fail count option. See Fail Count Enable Controls. Note that displaying total fail count can impact BitmapTool update performance.

Setting the value to TRUE causes BitmapTool to collect and display the total fail count value in the full view window border. Setting the value to FALSE disables BitmapTool from collecting and displaying the total fail count value.

`ui_BitmapDisplayTotalCount` is only effective if sent to UI before BitmapTool has been started. It was created to allow User Dialogs or User Tools to set the default state of the BitmapTool display total fail count mode. The default mode is TRUE.

Usage

Intended usage is via `remote_set()` and `remote_get()` from within user-written C-code. The variable is only valid when sent to UI (site = -1).

Example

The following example will retrieve and display total fail count when BitmapTool is next invoked:

```
remote_set(  "ui_BitmapDisplayTotalCount",
             TRUE,
             -1,
             TRUE,
             INFINITE );
```

The following example retrieves and outputs the current state of this variable:

```
BOOL val = remote_get( "ui_BitmapDisplayTotalCount", -1 );
output("  State => %s", val ? "TRUE" : "FALSE" );
```

7.1.11.13 ui_BitmapDisplayVisibleCount

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapDisplayVisibleCount`.

Description

This UI user variable is used to set the default state of BitmapTool's display visible fail count option. See Fail Count Enable Controls.

Setting the value to TRUE causes BItmapTool to collect and display the visible fail count value in the zoom view window border. Setting the value to FALSE disables BItmapTool from collecting and displaying the visible fail count value.

`ui_BitmapDisplayVisibleCount` is only effective if sent to UI before BitmapTool has been started. It was created to allow User Dialogs or User Tools to set the default state of the BitmapTool display visible fail count mode. The default mode is TRUE.

Usage

Intended usage is via `remote_set()` and `remote_get()` from within user-written C-code. The variable is only valid when sent to UI (site = -1).

Example

The following example will retrieve and display visible fail count when cause BitmapTool is next invoked:

```
remote_set(  "ui_BitmapDisplayVisibleCount",
             TRUE,
             -1,
             TRUE,
             INFINITE );
```

The following example retrieves and outputs the current state of this variable:

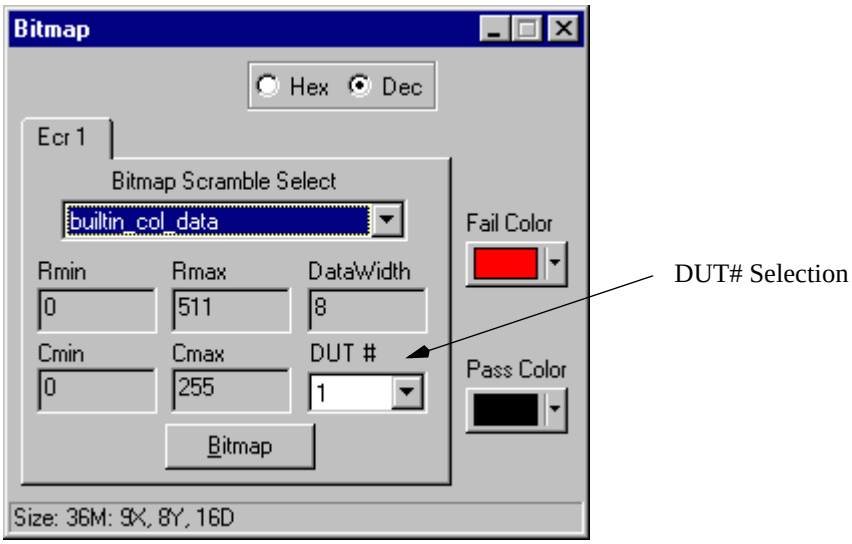
```
BOOL val = remote_get( "ui_BitmapDisplayVisibleCount", -1 );
output("  State => %s", val ? "TRUE" : "FALSE" );
```

7.1.11.14 ui_BitmapdutNo

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapdutNo`.

Description

This UI user variable can be used in user-written C-code to modify the DUT# normally controlled interactively using the BitmapTool control panel.



In the BitmapTool control panel this value is one's based i.e. the first DUT is 1. However, the system software treats this as zero based, thus the values used with `ui_BitmapdutNo` must be zero based i.e. the first DUT is 0.

This UI user variable is only effective if sent to UI before BitmapTool has been started. It was created to allow User Dialogs or User Tools to set the default state of DUT#. The default mode is DUT# 1.

Usage

Intended usage is via `remote_set()` and `remote_get()` from within user-written C-code. The variable is only valid when sent to UI (site = -1).

Example

The following example will set the value of DUT# to 2:

```
int dut = 2;
remote_set( "ui_BitmapdutNo", (dut - 1), -1 );
```

The following example read back and output the current value of DUT#:

```
CString val = remote_get( "ui_BitmapdutNo", -1 );
output("  Get ui_BitmapdutNo => %d", (val + 1));
```

7.1.11.15 ui_BitmapFailColor, ui_BitmapPassColor

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapFailColor, ui_BitmapPassColor`.

Description

`ui_BitmapFailColor` and `ui_BitmapPassColor` allow User Dialogs or User Tools to set the colors used to display Pass/Fail in BitmapTool. By default, the Pass color is black and the Fail color is red.

`ui_BitmapFailColor` and `ui_BitmapPassColor` are only effective if sent to UI before BitmapTool has been started

Usage

Intended usage is via `remote_set()` and `remote_get()` from within user-written C-code. The variable is only valid when sent to UI (site = -1).

Example

The following example will set the pass and fail color when BitmapTool is next invoked:

```
remote_set(  "ui_BitmapFailColor",
              RGB( 0, 0, 255 ),
             -1,
             TRUE,
             INFINITE );
remote_set(  "ui_BitmapPassColor",
              RGB( 0, 255, 0),
             -1,
             TRUE,
             INFINITE );
```

The following example retrieves and outputs the current state of this variable:

```
int c = atoi ( remote_get( "ui_BitmapPassColor", -1 ));
BYTE v = GetRValue( c ); output("  Red   => %d", v );
     v = GetGValue( c ); output("  Green => %d", v );
     v = GetBValue( c ); output("  Blue  => %d", v );
```

7.1.11.16 ui_BitmapMainSize

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapMainSize`.

Description

This UI user variable is used to set the size of BitmapTool's full view window. This control only works when the full view window is displayed separate from the zoom window. See BitmapTool Separate Window Option.

A single `__int64` value is used to specify both the X and Y size of the window, in pixels. The low 32 bits specify the X size, the high 32 bits specify the Y size.

`ui_BitmapMainSize` is only effective if sent to UI before BitmapTool has been started. It was created to allow User Dialogs or User Tools to set the default size of the BitmapTool full view window.

Usage

Intended usage is via `remote_set()` and `remote_get()` from within user-written C-code. The variable is only valid when sent to UI (site = -1).

Example

The following example will set the size of the full view window to 256 by 256 pixels when BitmapTool is next invoked:

```
CSize size;
size.cx = 256;
size.cy = 256;
__int64 main_window_size = __int64( size.cy ) << 32;
main_window_size |= size.cx;
remote_set(  "ui_BitmapMainSize",
             main_window_size,
```

```
            -1,
            TRUE,
            INFINITE );
```

The following example retrieves and outputs the current state of this variable:

```
__int64 val = remote_get( "ui_BitmapMainSize", -1 );
output("  Y size => %d", int (val >> 32) );
output("  X size => %d", int (val & 0xFFFF) );
```

7.1.11.17 ui_BitmapMaxErrors

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapMaxErrors`.

Description

Displaying a large number of failures for a large memory device can be quite slow. This UI user variable is used to limit the number of failures displayed.

The effect is approximate, based on ui_BitmapRowsChunk. BitmapTool will stop displaying errors when the maximum error count is reached, but errors are read from the ECR in *chunks*.

Usage

Intended usage is via `remote_set()` from within user-written C-code. The variable is only valid when sent to UI (site = -1). The value set to this variable specifies the maximum number of failures to be displayed, and should be > 0.

Example

The following example sets the maximum number of failures to 1024. As noted above, the effect will be approximate.

```
remote_set( "ui_BitmapMaxErrors", 1024, -1 );
```

The following example will read back and output the current value of `ui_BitmapMaxErrors:`

```
CString val = remote_get( "ui_BitmapMaxErrors", -1 );
output(" Get ui_BitmapMaxErrors => %s", val );
```

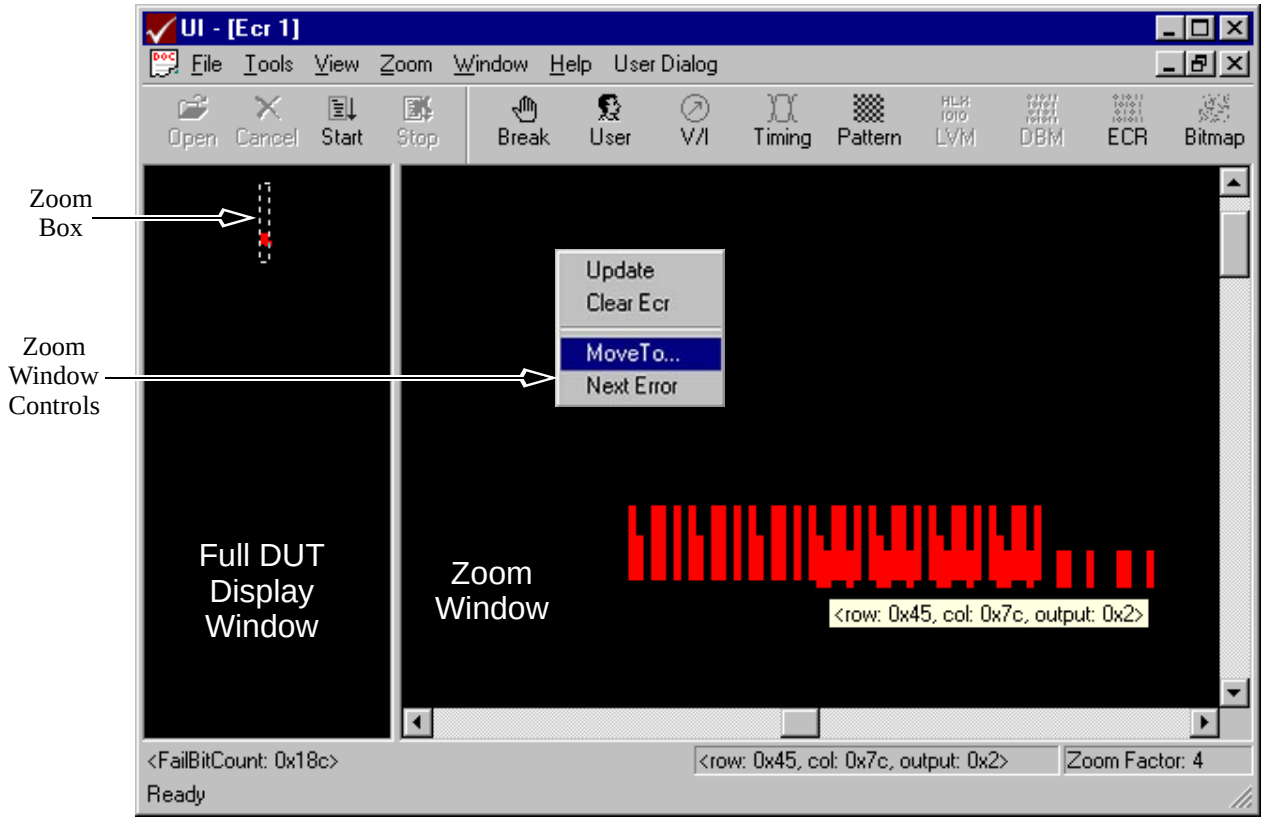
7.1.11.18 ui_BitmapMoveTo

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapMoveTo`.

Description

This UI user variable allows user-written C-code to move the *zoom box* to a specified row/column/data position within the left portion of the BitmapTool display.

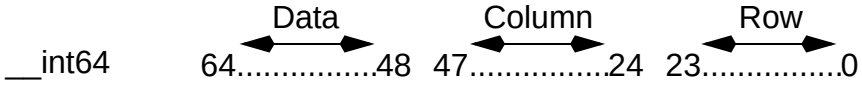
The *zoom box* is the dotted-line box that represents what portion of the entire DUT (left window) is displayed in the zoom window (right window). Zooming-in or -out causes the size of the box to change. The zoom box is moved interactively using the left mouse to drag the box to the desired location. It can also be moved using the Zoom Window Controls invoked using the right mouse button in the zoom window.



When moving the zoom box using C-code, the specified row/column/data coordinates specify the new position of the upper left corner of the zoom box. The final location does

consider the currently selected Bitmap Scramble Selection. However, movement stops when the right edge of the box reaches the right edge of the window, and/or the bottom edge of the box reaches the bottom of the window.

Using the `ui_BitmapMoveTo` variable presents a complexity which is addressed using the `BITMAP_MOVETO()` macro. To use the `ui_BitmapMoveTo` variable requires packing the row/column/data coordinate values into a single `__int64` value. Example 1: shows one method for doing this. The packed fields are:



To simplify things, use the `BITMAP_MOVETO()` macro (Example 2:) which allows row, column, and data values to be specified as separate integer arguments.

Usage

Intended usage of `ui_BitmapMoveTo` is via `remote_set()` from within user-written C-code. The variable is only valid when sent to UI (site = -1).

The `BITMAP_MOVETO()` Test System Macro can be called as desired, from Host, Site, or Tool code. It takes three arguments:

```
BITMAP_MOVETO( int row, int col, int data )
```

Example

Example 1:

This example demonstrates the explicit use of `ui_BitmapMoveTo`, with the row/column/data coordinates correctly packed into the single value passed to `remote_set()`.

```
__int64 row = 10, col = 20, data = 0x0;
remote_set( "ui_BitmapMoveTo",(data<<48|col<<24|row<<0), -1 );
```

Example 2:

This example demonstrates the use of `BITMAP_MOVETO()` macro, with separate row, column, and data coordinate values.

```
int row = 10, col = 20, data = 0x0;
BITMAP_MOVETO( row, col, data )
```

7.1.11.19 ui_BitmapPageHScroll, ui_BitmapPageVScroll, ui_BitmapLineHScroll, ui_BitmapLineVScroll

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapPageHScroll, ui_BitmapPageVScroll, ui_BitmapLineHScroll, ui_BitmapLineVScroll`.

Description

These UI user variables are used to set the initial values for BitmapTool's scrolling resolution. These are also controllable using the BitmapTool Control Dialog.

These UI user variables are only effective if sent to UI before BitmapTool has been started. They were created to allow User Dialogs or User Tools to set the initial values for BitmapTool's scrolling resolution. The default values are:

```
ui_BitmapPageHScroll   64
ui_BitmapPageVScroll   64
ui_BitmapLineHScroll   4
ui_BitmapLineVScroll   4
```

`ui_BitmapPageHScroll` and `ui_BitmapPageVScroll` are used to specify the amount the zoom window will scroll when the left mouse is clicked in the *page area* (see BitmapTool Display) of a scroll bar. The value is in databits, which are displayed using a different number of pixels depending on the current zoom factor.

`ui_BitmapLineHScroll` and `ui_BitmapLineVScroll` are used to specify the amount the zoom window will scroll when the left mouse is clicked on either *line scroll target* (see BitmapTool Display) of a scroll bar. The value is in databits, which are displayed using a different number of pixels depending on the current zoom factor.

Usage

Intended usage is via `remote_set()` and `remote_get()` from within user-written C-code. These variables are only valid when sent to UI (site = -1). It is recommended that the user experiment changing these values using the BitmapTool Control Dialog before setting values in software. Legal values are positive integers.

Example

The following example will set both the horizontal and vertical page scroll resolution to 100 databits when BitmapTool is next invoked:

```
remote_set( "ui_BitmapPageHScroll", 100, -1 )
remote_set( "ui_BitmapPageVScroll", 100, -1 )
```

The following example retrieves and outputs the current default state:

```
CString val = remote_get(  "ui_BitmapPageHScroll", -1 );
output("  State => %s", val );
```

7.1.11.20 ui_BitmapPan

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapPan`.

Description

This UI user variable is used to set the initial state of BitmapTool's zoom/pan box, which is also controllable using the BitmapTool Control Dialog.

`ui_BitmapPan` is only effective if sent to UI before BitmapTool has been started. It was created to allow User Dialogs or User Tools to modify the default mode of the zoom/pan box. The default mode is FALSE (zoom).

Usage

Intended usage is via `remote_set()` and `remote_get()` from within user-written C-code. The variable is only valid when sent to UI (site = -1). Setting the value to TRUE sets the zoom/pan box to pan mode when BitmapTool is started. FALSE sets the mode to zoom.

Example

The following example will set the mode to pan when BitmapTool is next invoked:

```
remote_set( "ui_BitmapPan", TRUE, -1 )
```

The following example retrieves and outputs the current default state:

```
CString val = remote_get(  "ui_BitmapPan", -1 );
output("  State => %s", val );
```

7.1.11.21 ui_BitmapRowsChunk

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapRowsChunk`.

Description

To update its display BitmapTool reads failure information from the ECR. For performance reasons, failure information is read in *chunks* consisting of all failing columns in a specific number of rows. In general, the larger the chunk the faster BitmapTool update will occur. The default is 512 rows (to improve the display update performance of BitmapTool the default was changed to 1024 in software release h1.0.25 and h2.0.10).

When `ui_BitmapMaxErrors` is used to limit the number of failures displayed in BitmapTool the system software can only halt when a new *chunk* is about to be read. The `ui_BitmapRowsChunk` user variable can be used to set the maximum number of rows read in a single chunk.

However, there is a tradeoff. Making `ui_BitmapRowsChunk` smaller reduces the overall update performance of BitmapTool but improves the granularity with which updating will halt when `ui_BitmapMaxErrors` is reached. Making `ui_BitmapRowsChunk` larger improves the overall update performance of BitmapTool but reduces the granularity with which updating will halt when `ui_BitmapMaxErrors` is reached.

This UI user variable can be used from a Windows command line, from within a batch file (see `ui_BatchFile`), or from user-written C-code.

Usage

From command line:

```
Long form:   /U:ui_BitmapRowsChunk=<value>
Short form:  /BRC=<value>
```

From within a batch file:

```
ui_BitmapRowsChunk=<value>
```

where **`<value>`** specifies the maximum number of rows to be read in a chunk.

From C-code, usage is via `remote_set()` and `remote_get()`. The variable is only valid when sent to UI (site = -1).

Example

These two examples perform identically when executed at a Windows command line:

```
C:\> ui /U:ui_BitmapRowsChunk=64
C:\> ui /BRC=64
```

The following example sets the maximum number of failing rows read from the ECR in one chunk to 64.

```
remote_set( "ui_BitmapRowsChunk", 64, -1 );
```

This example reads back and outputs the current chunk size:

```
CString val = remote_get( "ui_BitmapRowsChunk", -1 );
output(" Get ui_BitmapRowsChunk=> %s", val );
```

7.1.11.22 ui_BitmapRulers

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapRulers`.

Description

This UI user variable is used to set the initial state of BitmapTool's rulers (see Example BitmapTool Display with Overlays) i.e. whether rulers are displayed or not. Rulers are also controllable using the BitmapTool Control Dialog.

`ui_BitmapRulers` is only effective if sent to UI before BitmapTool has been started. It was created to allow User Dialogs or User Tools to modify the default state of rulers. The default mode is FALSE (not displayed).

Usage

Intended usage is via `remote_set()` and `remote_get()` from within user-written C-code. The variable is only valid when sent to UI (site = -1). Setting the value to TRUE causes rulers to be displayed when BitmapTool is started. FALSE hides the rulers.

Example

The following example cause rulers to be displayed when BitmapTool is next invoked:

```
remote_set( "ui_BitmapRulers", TRUE, -1 )
```

The following example retrieves and outputs the current default state:

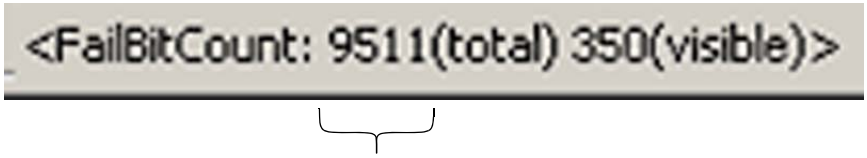
```
CString val = remote_get(  "ui_BitmapRulers", -1 );
output("  State => %s", val );
```

7.1.11.23 ui_BitmapTotalFailBitCount

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapTotalFailBitCount`.

Description

This Callback UI User Variable allows user-written code to determine the total fail bit count value displayed, instead of scanning the ECR for that value:



Total Fail Bit Count Value

This is the value displayed in the bottom edge of the main BitmapTool display. The Visible fail value does not change using `ui_BitmapTotalFailBitCount`.

If defined, `ui_BitmapTotalFailBitCount`'s body code is invoked by UI under the following conditions:

- The ECR scan routine used to obtain the total fail bit count value is about to be executed.

  And...
- BitmapTool's `Total Count` is option selected.

  And...
- The selected Bitmap Mode is either `Entire-XL` or `Visible`.

If these conditions are met, the normal ECR scan routine used to obtain the total failed bit count value is skipped, and the body code of `ui_CurrentBitmapScheme` is executed instead. This has the following effects:

- The execution time normally consumed to scan the ECR for total fail bit count is eliminated.

- The total failed bit count value displayed in BitmapTool is the value assigned to `ui_BitmapTotalFailBitCount`.
- Only the window which matches the currently selected ECR/DUT/Bitmap Scheme is updated.

BitmapTool has controls allowing the user to select the following options:

- Which DUT is being displayed (see `ui_BitmapdutNo`)
- Which Bitmap Scheme is in effect (see `ui_CurrentBitmapScheme`)
- Which Site is being displayed

When using `ui_BitmapTotalFailBitCount`, user code is totally responsible for comprehending these settings and correctly determine the total fail bit count value.

Usage

```
INT64_VARIABLE( ui_BitmapTotalFailBitCount, 0, ""){ [body ] }
```

where:

`ui_BitmapTotalFailBitCount` is a Callback UI User Variable.

body is the C-code the user adds to the body of `ui_BitmapTotalFailBitCount` used to change the value of `ui_BitmapTotalFailBitCount`.

Example

The following example executes a user-written function named `my_count_func()` to obtain the total fail bit count value to be displayed in place of the original value:

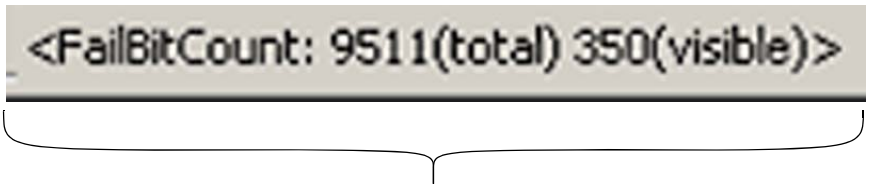
```
INT64_VARIABLE( ui_BitmapTotalFailBitCount, 0, "" ) {
   output("Original count => %I64d", ui_BitmapTotalFailBitCount );
   __int64 count = my_count_func();
   output("New count => %I64d", count );
   ui_BitmapTotalFailBitCount = count; // Display new count
}
```

7.1.11.24 ui_BitmapTotalVisibleFailBitString

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapTotalVisibleFailBitString`.

Description

This Callback UI User Variable allows user-written code to determine the complete string displayed for BitmapTool's total/visible fail bit count values:



Total/Visible Fail Bit Count String

If defined, this Callback UI User Variable is invoked by UI under the following conditions:

- BitmapTool's display is ***not*** split i.e. the full view and zoom view windows are not separate (see `ui_BitmapDisplaySeparateZoomWindow`)
- BitmapTool is about to update the total/visible fail bit count value (more below).

When `ui_BitmapTotalVisibleFailBitString` is executed, its value is the string which would be displayed if the call-back was not defined. This string includes the actual value for both total fails and visible fails. See above.

User code determines the string to display by assigning a new value to `ui_BitmapTotalVisibleFailBitString`; the last value assigned in the body code will be displayed in BitmapTool when the call-back exits.

As noted, this call-back will execute when BitmapTool is about to update the total fail bit count value. This will occur when the user clicks the Bitmap button, changes the size of the Zoom/Pan box, scrolls the zoom window, changes the Hex/Dec mode, etc.

Only the window which matches the currently selected ECR/DUT/Bitmap Scheme is updated.

Usage

```
CSTRING_VARIABLE(ui_BitmapTotalVisibleFailBitString, 0, ""){[body ]}
```

where:

`ui_BitmapTotalVisibleFailBitString` is a Callback UI User Variable .

body is the C-code the user adds to the body of `ui_BitmapTotalVisibleFailBitString` used to change the string displayed in BitmapTool.

Example

The following example changes the string to use all upper case letters:

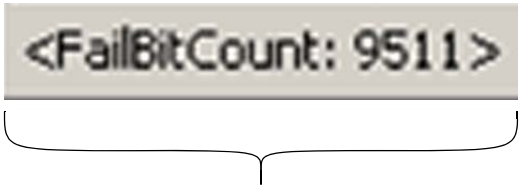
```
CSTRING_VARIABLE( ui_BitmapTotalVisibleFailBitString, "", "" ) {
   ui_BitmapTotalVisibleFailBitString.MakeUpper();
}
```

7.1.11.25 ui_BitmapTotalFailBitString

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapTotalFailBitString`.

Description

This Callback UI User Variable allows user-written code to determine the complete string displayed for BitmapTool's total fail bit count values:



Total Fail Bit Count String
This is the string displayed in Bitmap's Full View window

If defined, this Callback UI User Variable is invoked by UI under the following conditions:

- BitmapTool's display ***IS*** split i.e. the full view and zoom view windows are separate (see `ui_BitmapDisplaySeparateZoomWindow`)
- BitmapTool is about to update the total fail bit count value (more below).

When `ui_BitmapTotalFailBitString` is executed, its value is the string which would be displayed if the call-back was not defined. This string includes the actual total fails value. See above.

User code determines the string to display by assigning a new value to `ui_BitmapTotalFailBitString`; the last value assigned in the body code will be displayed in BitmapTool when the call-back exits.

As noted, this call-back will execute when BitmapTool is about to update the total fail bit count value, but only if the full and visible displays are separate. This will occur when the user clicks the Bitmap button, changes the Hex/Dec mode, etc.

Only the window which matches the currently selected ECR/DUT/Bitmap Scheme is updated.

Usage

```
CSTRING_VARIABLE(ui_BitmapTotalFailBitString, 0, ""){[body ]}
```

where:

`ui_BitmapTotalFailBitString` is a Callback UI User Variable.

body is the C-code the user adds to the body of `ui_BitmapTotalFailBitString` used to change the string displayed in BitmapTool.

Example

The following example changes the string to use all upper case letters:

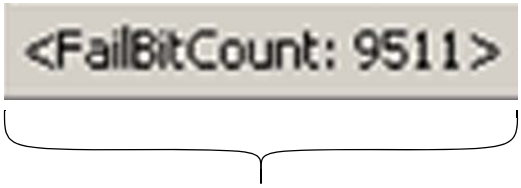
```
CSTRING_VARIABLE( ui_BitmapTotalFailBitString, "", "" ) {
   ui_BitmapTotalFailBitString.MakeUpper();
}
```

7.1.11.26 ui_BitmapVisibleFailBitString

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapVisibleFailBitString`.

Description

This Callback UI User Variable allows user-written code to determine the complete string displayed for BitmapTool's visible fail bit count values:



Visible Fail Bit Count String
This is the string displayed in Bitmap's Zoom View window

If defined, this Callback UI User Variable is invoked by UI under the following conditions:

- BitmapTool's display ***IS*** split i.e. the full view and zoom view windows are separate (see `ui_BitmapDisplaySeparateZoomWindow`)
- BitmapTool is about to update the visible fail bit count value (more below).

When `ui_BitmapVisibleFailBitString` is executed, its value is the string which would be displayed if the call-back was not defined. This string includes the actual visible fails value. See above.

User code determines the string to display by assigning a new value to `ui_BitmapVisibleFailBitString`; the last value assigned in the body code will be displayed in BitmapTool when the call-back exits.

As noted, this call-back will execute when BitmapTool is about to update the visible fail bit count value, but only if the full and visible displays are separate. This will occur when the user clicks the Bitmap button, changes the size of the Zoom/Pan box, scrolls the zoom window, changes the Hex/Dec mode, etc.

Only the window which matches the currently selected ECR/DUT/Bitmap Scheme is updated.

Usage

```
CSTRING_VARIABLE(ui_BitmapVisibleFailBitString, 0, ""){[body ]}
```

where:

`ui_BitmapVisibleFailBitString` is a Callback UI User Variable.

body is the C-code the user adds to the body of `ui_BitmapVisibleFailBitString` used to change the string displayed in BitmapTool.

Example

The following example changes the string to use all upper case letters:

```
CSTRING_VARIABLE( ui_BitmapVisibleFailBitString, "", "" ) {
   ui_BitmapVisibleFailBitString.MakeUpper();
}
```

7.1.11.27 ui_BitmapVisibleSize

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapVisibleSize`.

Description

This UI user variable is used to set the size of BitmapTool's zoom view window.

When it is displayed separate from the full view windo (see BitmapTool Separate Window Option) this has no effect on the full view window. When the two windows are connected, setting the vertical size of the zoom window can affect the full view window.

A single `__int64` value is used to specify both the X and Y size of the window, in bits (not pixels). The low 32 bits specify the X size, the high 32 bits specify the Y size.

The `set_bitmap_visible_size()` function does the same thing.

Both methods were created to allow User Dialogs or User Tools to set the default size of the BitmapTool full view window.

Using `set_bitmap_visible_size()`, separate arguments are provided for X vs. Y size. See Usage.

Using either method, size is specified in databits.

Either method is only effective if invoked before BitmapTool has been started.

Usage

The intended usage of `ui_BitmapVisibleSize` is using `remote_set()` and `remote_get()` from within user-written C-code. The variable is only valid when sent to UI (site = -1).

```
void set_bitmap_visible_size( int width, int height );
```

where:

`width` and **`height`** set the corresponding sizes of the BitmapTool zoom window.

Example

The following example will set the size of the zoom view window to 400 by 200 pixels when BitmapTool is next invoked:

```
CSize size;
size.cx = 400;
size.cy = 200;
__int64 zoom_window_size = __int64( size.cy ) << 32;
zoom_window_size |= size.cx;
remote_set(  "ui_BitmapVisibleSize",
             zoom_window_size,
             -1,
             TRUE,
             INFINITE );
```

The following example retrieves and outputs the current state of this variable:

```
__int64 val = remote_get( "ui_BitmapVisibleSize", -1 );
output("  Y size => %d", int (val >> 32) );
output("  X size => %d", int (val & 0xFFFF) );
```

7.1.11.28 ui_BitmapZoom2

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BitmapZoom2`.

Description

This UI user variable is used to set the initial state of BitmapTool's *power of 2* zoom mode, which is also controllable using the BitmapTool Control Dialog.

`ui_BitmapZoom2` is only effective if sent to UI before BitmapTool has been started. It was created to allow User Dialogs or User Tools to modify the default state of the *power of 2* zoom mode. The default mode is FALSE (disabled).

Usage

Intended usage is via `remote_set()` and `remote_get()` from within user-written C-code. The variable is only valid when sent to UI (site = -1). Setting the value to TRUE enables *power of 2* zoom mode when BitmapTool is started. FALSE disables *power of 2* zoom mode.

Example

The following example enables *power of 2* zoom mode when BitmapTool is next invoked:

```
remote_set( "ui_BitmapZoom2", TRUE, -1 )
```

The following example retrieves and outputs the current default state:

```
CString val = remote_get(  "ui_BitmapZoom2", -1 );
output("  State => %s", val );
```

7.1.11.29 ui_BreakPointFile

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BreakPointFile`.

Description

This UI user variable is used to specify a Breakpoint Definition File to be loaded as a test program is loaded.

`ui_BreakPointFile` can be used from a Windows command line, and from within a batch file (see `ui_BatchFile`).

The value is NOT persistent after UI is terminated (via the UI.ini file).

Usage

```
CSTRING_VARIABLE( ui_BreakPointFile, "", "" )
```

From command line:

```
Long form:   /U:ui_BreakPointFile=<value>
Short form:  /BRK=<value>
```

From within a batch file:

```
ui_BreakPointFile=<value>
```

where **`<value>`** is the name of a Breakpoint Definition File. If a full path/file name is not specified the current `PATH` is searched.

From C-code, usage is via `remote_set()` and `remote_get()`. The variable is only valid when sent to UI (site = -1).

Example

These two examples perform identically when executed at a Windows command line:

```
C:\> ui /U:ui_BreakPointFile=c:\breakfile.txt
```

```
C:\> ui /BRK=c:\breakfile.txt
```

This example sets the mode from user-written C-code:

```
remote_set("ui_BreakPointFile", "c:\breakfile.txt", -1 );
```

The following example will read back and output the state of this variable from user C-code:

```
CString val = remote_get("ui_BreakPointFile", -1 );
output(" Get ui_BreakPointFile => %s", val );
```

7.1.11.30 ui_BreakPointRemoveAll

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_BreakPointRemoveAll`.

Note: first available in software release h2.2.7/h1.2.7.

Description

This UI user variable is used to clear all breakpoints currently set via the Breakpoint Monitor. This has the same effect as clicking the Remove All button in the Breakpoint Monitor.

`ui_BreakPointRemoveAll` can only be used from within user-written C-code. See Usage below.

Usage

From C-code, intended usage is via `remote_set()` only. The variable is only valid when sent to UI (site = -1). The value assigned to this user variable is not used.

Example

```
remote_set("ui_BreakPointRemoveAll", 0, -1 );
```

7.1.11.31 ui_ClearAtProgramLoad

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_ClearAtProgramLoad`.

Description

This UI user variable is used to set a mode which causes the UI Site output window(s) to be cleared when a [new] test program is loaded. The default is TRUE.

`ui_ClearAtProgramLoad` can be used from a Windows command line, from within a batch file (see `ui_BatchFile`), or from within user-written C-code. See Usage below.

The value IS persistent after UI is terminated (via the UI.ini file).

Note: this variable has limited value to most users. It is targeted at applications where a command line or batch file starts UI and automatically loads and executes multiple test programs where the output windows must not be cleared between programs.

Usage

From command line:

```
Long form:   /U:ui_ClearAtProgramLoad=<value>
Short form:  /CP=<value>
Short form: /CP
```

The latter short form assumes value = 1.

From within a batch file:

```
ui_ClearAtProgramLoad=<value>
```

where **`<value>`** is 0 (FALSE) or 1 (TRUE).

From C-code, intended usage is via `remote_set()` and `remote_get()`. The variable is only valid when sent to UI (site = -1). Legal values assigned to this user variable must be 0 (FALSE) or 1 (TRUE).

Example

These three examples perform identically when executed at a Windows command line:

```
C:\> ui /U:ui_ClearAtProgramLoad=1
C:\> ui /CP=1
C:\> ui /CP
```

This example sets the mode from user-written C-code:

```
remote_set("ui_ClearAtProgramLoad", 0, -1 );
```

The following example will read back and output the state of this variable from user C-code:

```
CString val = remote_get("ui_ClearAtProgramLoad", -1 );
output(" Get ui_ClearAtProgramLoad => %s", val );
```

7.1.11.32 ui_ClearAtTestStart

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_ClearAtTestStart`.

Description

This UI user variable is used to set a mode which causes the UI Site output window(s) to be cleared each time the start test signal is invoked i.e. each time the Sequence & Binning Table is executed.

`ui_ClearAtTestStart` can be used from a Windows command line, from within a batch file (see `ui_BatchFile`), or from within user-written C-code. See Usage below.

The value IS persistent after UI is terminated (via the UI.ini file).

Usage

From command line:

```
Long form:   /U:ui_ClearAtTestStart=<value>
Short form:  /CS=<value>
Short form:  /CS
```

The latter short form assumes value = 1.

From batch file:

```
ui_ClearAtTestStart=<value>
```

where **`<value>`** is 0 (FALSE) or 1 (TRUE). The default is FALSE.

From C-code, intended usage is via `remote_set()` and `remote_get()`. The variable is only valid when sent to UI (site = -1). Legal values assigned to this user variable must be 0 (FALSE) or 1 (TRUE).

Example

These three examples perform identically when executed at a Windows command line:

```
C:\> ui /U:ui_ClearAtTestStart=1
C:\> ui /CS=1
C:\> ui /CS
```

This example sets the mode from user-written C-code:

```
remote_set("ui_ClearAtTestStart", 1, -1 );
```

The following example will read back and output the state of this variable from user C-code:

```
CString val = remote_get("ui_ClearAtTestStart", -1 );
output(" Get ui_ClearAtTestStart => %s", val );
```

7.1.11.33 ui_Close

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_Close`.

Description

This UI user variable is used to close a test program from user-written C-code, as if selecting the **`File Close`** option in the UI *File* Menu. This does not terminate UI.

Usage

Intended usage is via `remote_set()` from within user-written C-code. The variable is only valid when sent to UI (site = -1). The value assigned to the variable is ignored. See example:

Example

This example will immediately close (terminate) the test program.

```
remote_set( "ui_Close", -999, -1 ); // Value (-999) is ignored
```

7.1.11.34 ui_CloseAfterRun

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_CloseAfterRun`.

Description

This UI user variable is used to set a mode which causes a test program to be closed after executing the Sequence & Binning Table ui_RunTestProgram times (default = 1). The program is closed as if the **`File Close`** option was selected in the UI *File* Menu. This does not terminate UI.

This UI user variable can be used from a Windows command line, from within a batch file (see `ui_BatchFile`), or from within user-written C-code. See Usage below.

The value is NOT persistent after UI is terminated (via the UI.ini file).

Note: this variable has limited value to most users. It is targeted at applications where a command line or batch file starts UI, automatically loads and executes multiple test programs, and each program must close after one execution of its Sequence & Binning Table.

Usage

From command line:

```
Long form:   /U:ui_CloseAfterRun=<value>
Short form:  /CR=<value>
Short form:  /CR
```

The latter short form assumes value = 1.

From batch file:

```
ui_CloseAfterRun=<value>
```

where **`<value>`** is 0 (FALSE) or 1 (TRUE). The default is FALSE.

From C-code, intended usage is via `remote_set()` and `remote_get()`. The variable is only valid when sent to UI (site = -1). Legal values assigned to this user variable must be 0 (FALSE) or 1 (TRUE).

Example

These three examples perform identically when executed at a Windows command line:

```
C:\> ui /U:ui_CloseAfterRun=1
C:\> ui /CR=1
C:\> ui /CR
```

This example sets the mode from user-written C-code:

```
remote_set("ui_CloseAfterRun", 1, -1 );
```

The following example will read back and output the state of this variable from user C-code:

```
CString val = remote_get("ui_CloseAfterRun", -1 );
output(" Get ui_CloseAfterRun => %s", val );
```

7.1.11.35 ui_Controller

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_Controller`.

Description

This UI user variable is used when starting UI from a Windows command line or a batch file, to specify which site(s) will be enabled for use. Several usage options are possible as noted below.

Note: this variable has limited value to most users. It is targeted at applications where a command line or batch file starts UI, selects which of the target test system's site controller(s) will load the test program and loads a test program.

Note the following:

- In the original implementation, values are assigned to `ui_Controller` to specify the IP address of the site(s) which are to be used. See Usage. Beginning in software release h1.1.23 additional options are available, more below.

Note: prior to software release h1.1.23, the enabled site(s) were persistent after UI was terminated, via the *UI.ini* file. This was true regardless of the method used to select the enabled sites: UI, using `ui_Controller`, or both.
Beginning in software release h1.1.23, the enabled site(s) are persistent ONLY when set using UI's graphical interface. Conversely, when `ui_Controller` is used to specify the enabled site(s) the persistent values read from the *UI.ini* file are not considered.

- This UI user variable is only usable from a Windows command line, or from a batch file (see `ui_BatchFile`).
- Values set using `ui_Controller` are displayed in UI's Advanced splash screen. Changes made in the splash screen will overwrite those set using `ui_Controller`. Since, normally, `ui_Controller` is used to start UI and load a test program without requiring the user to interact with the UI, it is recommended that `ui_NoLogo` also be used. This allows the batch file or command line sequence to start UI and immediately load the test program, bypassing the splash screen.

Beginning in software release h1.1.23, `ui_Controller` can be used to specify which sites are enabled using a syntax which specifies a chassis number plus a site number (slot number, HSB number, etc.). The syntax is:

```
chassis_site
```

I.e.

```
ui_Controller = 1_1
ui_Controller = 1_2
```

Sites which are not explicitly enabled are not loaded. For example, the following batch file loads a test program only on the first 2 sites (HSBs) of the first four chassis:

```
ui_NoLogo = 1
ui_Controller = clear
ui_Controller = 1_1
ui_Controller = 1_2
ui_Controller = 2_1
ui_Controller = 2_2
ui_Controller = 3_1
ui_Controller = 3_2
ui_Controller = 4_1
ui_Controller = 4_2
ui_Open = D:\myProg\Debug\myProgram.exe
```

This same results are obtained using the following command line (shown on multiple lines for clarity):

```
ui /nologo /c=clear /c=1_1 /c=1_2 /c=2_1 /c=2_2 /c=3_1 /c=3_2
/c=4_1 /c=4_2 /TP="D:\myProg\Debug\myProgram.exe"
```

By default, the sites which load a test program are sequentially numbered. Thus, in the examples above, from the test program's perspective (i.e. the value returned by `site_num()`), sites 1 through 8 have loaded the test program. However, the physical site numbering can be retained by explicitly skipping those sites which are not loading the test program, using `ui_Controller = skip`, as follows:

```
ui_NoLogo = 1
ui_Controller = clear
ui_Controller = 1_1
ui_Controller = 1_2
ui_Controller = skip # Skip 1_3
ui_Controller = skip # Skip 1_4
ui_Controller = skip # Skip 1_5
ui_Controller = 2_1
ui_Controller = 2_2
ui_Controller = skip # Skip 2_3
ui_Controller = skip # Skip 2_4
ui_Controller = skip # Skip 2_5
ui_Controller = 3_1
ui_Controller = 3_2
ui_Controller = skip # Skip 3_3
ui_Controller = skip # Skip 3_4
ui_Controller = skip # Skip 3_5
ui_Controller = 4_1
ui_Controller = 4_2
ui_Open = D:\myProg\Debug\myProgram.exe
```

Or, from the command line (a shorter-hand syntax follows):

```
ui /nologo /c=clear /c=1_1 /c=1_2 /c=skip /c=skip /c=skip /c=2_1
/c=2_2 /c=skip /c=skip /c=skip /c=3_1 /c=3_2 /c=skip /c=skip
/c=skip /c=4_1 /c=4_2 /c=skip /c=skip /c=skip
/TP="D:\myProg\Debug\myProgram.exe"
```

In these examples, the test program is loaded on physical sites 1, 2, 6, 7, 11, 12, 16, and 17 and these will be the values returned by `site_num()` executing on these sites.

It is even possible (although not very useful) to control which `site_num()` a given chassis/site will return. For example, the following command line enables chassis-1/site-1 and chassis-1/site-2 (out of order), but `site_num()` executing in chassis-1/site-1 will return 3 and `site_num()` executing in chassis-1/site-2 will return 1:

```
ui /c=1_2 /c=skip /c=1_1 /nologo
```

A shorter-hand version of the command line syntax allows multiple, comma-delimited, values to be assigned to `/c`. For example, the following results in the same operation as the longer version above. Note the quotations:

```
ui /nologo /c="clear, 1_1, 1_2, skip, skip, skip, 2_1, 2_2, skip,
skip, skip, 3_1, 3_2, skip, skip, skip, 4_1, 4_2, skip, skip, skip"
/TP="D:\myProg\Debug\myProgram.exe"
```

When multiple `ui_Controller` (batch file) or the `/c` (command line) options are specified, i.e. a comma separated list of values, an automatic `ui_Controller=clear` is executed prior to setting the first `ui_Controller` value. Thus, the following is even shorter:

```
ui /nologo /c="1_1, 1_2", skip, skip, skip, 2_1, 2_2, skip, skip,
skip, 3_1, 3_2, skip, skip, skip, 4_1, 4_2, skip, skip, skip /
TP="D:\myProg\Debug\myProgram.exe"
```

To inhibit the auto-clear operation use `ui_Controller=keep`, i.e.:

```
ui /nologo /c=keep /c="1_1, 1_2" /c=2_1 /TP="D:\myProg\Debug\myProgram.exe"
```

Usage

```
CSTRING_VARIABLE( ui_Controller, "", "" )
```

From command line:

```
Long form:   /U:ui_Controller=<value>
Short form:  /C=<value>
```

From within a batch file:

```
ui_Controller=<value>
```

where **`<value>`** is one of:

- `localhost`
- An IP address i.e. 188.192.0.0
- `clear`
- `skip`
- `keep`

- Host name i.e. the computer name obtained by executing **`ipconfig /all`** in a Windows shell.

Example

These two examples perform identically when executed at a Windows command line:

```
C:\> ui /U:ui_Controller=clear /U:ui_Controller=188.192.0.0
C:\> ui /C=clear /C=188.192.0.0
```

Example batch file:

```
ui_Controller=clear
ui_Controller=localhost
```

7.1.11.36 ui_CurrentBitmapScheme

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_CurrentBitmapScheme`.

Description

This Callback UI User Variable is invoked by UI any time a new Bitmap scheme is selected in BitmapTool.

If `ui_CurrentBitmapScheme` is defined in the test program and/or User Tools its body code will be executed any time a new Bitmap scheme is selected in BitmapTool.

Usage

```
CSTRING_VARIABLE( ui_CurrentBitmapScheme, "", "" ) { [body ] }
```

where:

`ui_CurrentBitmapScheme` is a Callback UI User Variable.

`body` is the C-code the user adds to the body of `ui_CurrentBitmapScheme` in their test program or User Tools.

Examples

This example sets the size of the BitmapTool window when a new BitmapTool is selected. If the main and zoom window are displayed separately this sizes the zoom window only:

```
CSTRING_VARIABLE( ui_CurrentBitmapScheme, "", "Bitmap scheme" ){
   CSize size;
   size.cx = 256; // X size
   size.cy = 256; // Y size
   __int64 wsize = __int64( size.cy ) << 32;
   wsize |= size.cx;
   remote_set( "ui_BitmapVisibleSize", wsize, -1, TRUE, INFINITE);
}
```

7.1.11.37 ui_DbmDialogDecMode

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_DbmDialogDecMode`.

Note: first available in software release h1.1.23.

Description

This UI user variable is used to set the Hex/Decimal display mode for DBMTool. This is the control displayed in the upper-right corner of DBMTool. Note the following:

- Setting the value to TRUE causes DBMTool to display row/column header values in decimal, FALSE in hexadecimal.
- This setting does not affect the values displayed in DBMTool's main grid area, which are always displayed in hexadecimal.
- `ui_DbmDialogDecMode` is only effective if sent to UI before DBMTool has been started. It was created to allow User Dialogs or User Tools to set the default state of the DBMTool display mode. The default mode is TRUE.

Usage

Intended usage is via `remote_set()` and `remote_get()` from within user-written C-code. The variable is only valid when sent to UI (site = -1).

Example

The following example will set DBMTool's row/column header display mode to hexadecimal when DBMTool is next invoked:

```
remote_set(  "ui_DbmDialogDecMode",
             FALSE,
             -1,
             TRUE,
             INFINITE );
```

The following example retrieves and outputs the current state of this variable:

```
BOOL val = remote_get( "ui_DbmDialogDecMode", -1 );
output("  State => %s", val ? "TRUE" : "FALSE" );
```

7.1.11.38 ui_DutBoardStatusCheckDisable

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_DutBoardStatusCheckDisable`.

Note: first available in software release h1.1.23.

Description

This UI user variable is used to disable the DUT Board Status Check.

By default, the DUT board status check is performed when any test program is loaded on a Magnum 1/2/2x Test System. Note the following:

- `ui_DutBoardStatusCheckDisable` is a `BOOL_VARIABLE`, which can be set to TRUE or FALSE (default). There is no benefit in setting `ui_DutBoardStatusCheckDisable` = FALSE.
- `ui_DutBoardStatusCheckDisable` is set using `remote_set()`, and must be sent to UI (site = -1). See Usage.
- Proper operation requires that `ui_DutBoardStatusCheckDisable` be set from a `HOST_CONFIGURATION()` block. See Usage.
- The current value of `ui_DutBoardStatusCheckDisable` may be retrieved from UI using `remote_get()`:

  ```
  BOOL b = remote_get( ui_DutBoardStatusCheckDisable, -1 );
  ```

- Once the program load process has exited the `HOST_CONFIGURATION()` block, setting `ui_DutBoardStatusCheckDisable` has no effect.

- When a test program unloads, `ui_DutBoardStatusCheckDisable` is set = FALSE. This ensures the next program loaded has independent control over this option.

Usage

```
HOST_CONFIGURATION( HB1 ){ // See HOST_CONFIGURATION()
   remote_set( "ui_DutBoardStatusCheckDisable", TRUE, -1 );
}
```

Example

See Usage.

7.1.11.39 ui_ECRDialogDecMode

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_ECRDialogDecMode`.

Note: first available in software release h1.1.23.

Description

This UI user variable is used to set the Hex/Decimal display mode for ECRTool. This is the control displayed in the upper-right corner of ECRTool. Note the following:

- Setting the value to TRUE causes ECRTool to display row/column header values in decimal, FALSE in hexadecimal.
- This setting does not affect the values displayed in ECRTool's main grid area, which are always displayed in hexadecimal.
- `ui_ECRDialogDecMode` is only effective if sent to UI before ECRTool has been started. It was created to allow User Dialogs or User Tools to set the default state of the ECRTool display mode. The default mode is TRUE.

Usage

Intended usage is via `remote_set()` and `remote_get()` from within user-written C-code. The variable is only valid when sent to UI (site = -1).

Example

The following example will set ECRTool's row/column header display mode to hexadecimal when ECRTool is next invoked:

```
remote_set(  "ui_ECRDialogDecMode",
             FALSE,
             -1,
             TRUE,
             INFINITE );
```

The following example retrieves and outputs the current state of this variable:

```
BOOL val = remote_get( "ui_ECRDialogDecMode", -1 );
output("  State => %s", val ? "TRUE" : "FALSE" );
```

7.1.11.40 ui_EngineeringMode

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_EngineeringMode`.

Description

Engineering mode must be set (`TRUE`) to enable the graphical tools in UI.

This mode is normally set using the UI graphic interface (above) but can also be done using `ui_EngineeringMode`.

This UI user variable can be used from a Windows command line, from within a batch file (see ui_BatchFile), or from within user-written C-code. From C-code it is possible to enable/disable engineering mode even though UI is running and/or a test program is loaded.

The value is NOT persistent after UI is terminated (via the UI.ini file).

Usage

From command line:

```
Long form:   /U:ui_EngineeringMode=<value>
Short form:  /E=<value>
Short form:  /E
```

The latter short form assumes value = 1 (TRUE).

From batch file:

```
ui_EngineeringMode=<value>
```

where **<value>** is 0 (FALSE) or 1 (TRUE). The default is FALSE.

From C-code, intended usage is via remote_set() and remote_get(). The variable is only valid when sent to UI (site = -1). Legal values assigned to this user variable must be 0 (FALSE) or 1 (TRUE).

Example

These three examples perform identically when executed at a Windows command line:

```
C:\> ui /U:ui_EngineeringMode=1
C:\> ui /E=1
C:\> ui /E
```

This example sets the mode from user-written C-code:

```
remote_set("ui_EngineeringMode", 1, -1 );
```

The following example will read back and output the state of this variable from user C-code:

```
CString val = remote_get("ui_EngineeringMode", -1 );
output(" Get ui_EngineeringMode => %s", val );
```

7.1.11.41 ui_ExcelAppEvent

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_ExcelAppEvent`.

Description

This Callback UI User Variable is invoked by UI when it receives an event from Excel.

`ui_ExcelAppEvent` must be enabled using the `EnableExcelAppEvents()` function.

If `ui_ExcelAppEvent` enabled and defined in the test program and/or User Tools, its body code will be executed by UI when it receives an event from Excel, which occurs when the user clicks in an Excel spreadsheet.

When it is executed, the value of `ui_ExcelAppEvent` will be one of the following `ExcelAppEventsType` values, indicating the nature of the event:

```
enum ExcelAppEventsType {
   WindowResize,
   WindowActivate,
   WindowDeactivate,
   SheetSelectionChange,
   SheetBeforeDoubleClick,
   SheetBeforeRightClick,
   SheetActivate,
   SheetDeactivate,
   SheetCalculate,
   SheetChange,
   NewWorkbook,
   WorkbookOpen,
   WorkbookActivate,
   WorkbookDeactivate,
   WorkbookBeforeClose,
   WorkbookBeforeSave,
   WorkbookBeforePrint,
   WorkbookNewSheet,
   WorkbookAddinInstall,
   WorkbookAddinUninstall
};
```

Usage

```
INT_VARIABLE( ui_ExcelAppEvent, 0, "" ) { [body code] }
```

where:

`ui_ExcelAppEvent` is a Callback UI User Variable.

body is the C-code the user adds to the body of `ui_ExcelAppEvent` in their test program or User Tools.

Example

This example executes any time the user clicks in Excel. If the cursor is clicked in a worksheet cell, the event type will be `SheetSelectionChange`, and the code below will then read the coordinates of the cell:

```
INT_VARIABLE( ui_ExcelAppEvent, 0, "" ) {
   if ( ui_ExcelAppEvent == SheetSelectionChange ) {
      int row, col;
      GetActiveCell( &row, &col );
      output( " Row => %d : Col => %d", row, col );
   }
}
```

7.1.11.42 ui_Exit

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_Exit`.

Description

This UI user variable is used to terminate UI from user-written C-code, as if selecting the **`File Exit`** option in the UI *File* Menu. This will also terminate any loaded test programs.

Usage

Intended usage is via `remote_set()` from within user-written C-code. The variable is only valid when sent to UI (site = -1). The value assigned to the variable is ignored. See example:

Example

This example will immediately terminate UI and close any loaded test programs.

```
remote_set( "ui_Exit", -999, -1 ); // Value (-999) is ignored
```

7.1.11.43 ui_ExitAfterRun

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_ExitAfterRun`.

Description

This UI user variable can be used to set a mode which causes UI to be terminated after executing the Sequence & Binning Table of the currently loaded test program `ui_RunTestProgram` times (default = 1).

The value is NOT persistent after UI is terminated (via the UI.ini file).

Usage

From command line:

```
Long form:   /U:ui_ExitAfterRun=<value>
Short form:  /ER=<value>
Short form:  /ER
```

The latter short form assumes value = 1 (TRUE).

From batch file:

```
ui_ExitAfterRun=<value>
```

where **<value>** is 0 (FALSE) or 1 (TRUE). The default is FALSE.

From C-code, intended usage is via `remote_set()` and `remote_get()`. The variable is only valid when sent to UI (site = -1). Legal values assigned to this user variable must be 0 (FALSE) or 1 (TRUE).

Example

These three examples perform identically when executed at a Windows command line:

```
C:\> ui /U:ui_ExitAfterRun=1
C:\> ui /ER=1
C:\> ui /ER
```

This example sets the mode from user-written C-code:

```
remote_set("ui_ExitAfterRun", 1, -1 );
```

The following example will read back and output the state of this variable from user C-code:

```
CString val = remote_get("ui_ExitAfterRun", -1 );
output(" Get ui_ExitAfterRun => %s", val );
```

7.1.11.44 ui_HideTool

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_HideTool`.

Note: first available in software release h1.1.23.

This UI user variable can be used to hide most of UI's Interactive Tools from user C-code. In this context, hide means to cause the tool display to be hidden.

`ui_HideTool` is the complement of `ui_ShowTool`. Details for both are documented in `ui_ShowTool`.

7.1.11.45 ui_HostDebug

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_HostDebug`.

Description

Host Debug mode must be set (TRUE) to enable source code debugging using Developer Studio

This mode is normally set using the UI graphic interface (above) but can also be done using `ui_HostDebug`.

This UI user variable can be used from a Windows command line, or from within a batch file (see `ui_BatchFile`).

It is possible to use `ui_HostDebug` from C-code but this has limited value because it has no effect until the currently loaded test program is closed, and another test program is loaded, and only during the existing invocation of UI.

The value is NOT persistent after UI is terminated (via the UI.ini file).

A similar UI user variable is available to enable site debug: see `ui_SiteDebug`.

Usage

From command line:

```
Long form:   /U:ui_HostDebug=<value>
Short form:  /HD=<value>
Short form:  /HD
```

The latter short form assumes value = 1 (TRUE).

From batch file:

```
ui_HostDebug=<value>
```

where **<value>** is 0 (FALSE) or 1 (TRUE). The default is FALSE.

From C-code, intended usage is via `remote_set()` and `remote_get()`. The variable is only valid when sent to UI (site = -1).

Example

These three examples perform identically when executed at a Windows command line:

```
C:\> ui /U:ui_HostDebug=1
C:\> ui /HD=1
C:\> ui /HD
```

This example sets the mode from user-written C-code:

```
remote_set("ui_HostDebug", 1, -1 );
```

The following example will read back and output the state of this variable from user C-code:

```
CString val = remote_get("ui_HostDebug", -1 );
output(" Get ui_HostDebug => %s", val );
```

7.1.11.46 ui_HostModeCommandLine

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_HostModeCommandLine`.

Description

This UI user variable is used to specify command line arguments which will be executed in the Host process.

This is targeted at setting the value of one or more user variables in the Host process, using command line or batch file methods (see `ui_BatchFile`). Similar capabilities exist for the Site using `ui_SiteModeCommandLine`.

A single string value is assigned to `ui_HostModeCommandLine`, within which the `/h:` and `/H:` delimiters are used to:

- Delimit each user variable value pair

- Designate whether the specified user variable initialization is to occur before (`/h:`) the `CONFIGURATION()` block executes (i.e. before any program initialization has begun) or after (`/H:`) the `INITIALIZATION_HOOK()` executes (i.e. after all program initialization has completed).

Prior to being initialized by `ui_HostModeCommandLine` the user variable value is determined by the value set in the program source code.

Only one `ui_HostModeCommandLine` exists, the last one specified replaces any that were previously defined.

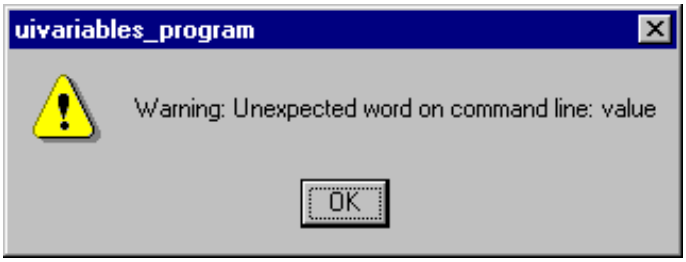
The body code of the user variable(s) being initialized executes in the Host process. If multiple values are assigned to a user variable, which can be done using separate instances of `/H:` or `/h:`, the body code will execute once for each value. It is possible to set both `/h:` and `/H:` for the same user variable giving it one value before program initialization and a different value afterwards.

Normally, the command string only executes once, while loading the first test program after UI starts. Any other test programs loaded during the same UI invocation will not be affected. Using the Reload option the initialization will occur for each test program loaded using the current UI invocation.

The value is NOT persistent after UI is terminated (via the UI.ini file).

Two error types are reported, which occur during program loading:

- An invalid user variable name results in a warning message in the Host output window. When this occurs, the test program continues to load/execute. In the example warning message below the invalid user variable name is: *Host_XCS2: Warning: Attempt to set undefined user-variable "Host_XCS2" from site 0 ignored. Warning: This message will not be repeated for user-variable "Host_XCS2".*
- An invalid command string results in the following error popup (the program name will be different). After the OK button is clicked the test program continues to load/execute:



Usage

```
CSTRING_VARIABLE( ui_HostModeCommandLine, "", "" )
```

From command line:

```
Long form:   /U:ui_HostModeCommandLine="command string"
Short form:  /HC="command string"
```

From batch file:

```
ui_HostModeCommandLine="command string"
```

where **command string** utilizes two delimiters to separate user variable value pairs:

- /h:
- /H:

Using lowercase **/h:** causes the specified user variable to be initialized before the CONFIGURATION() block executes (i.e. before any program initialization has begun). Using the uppercase **/H:** causes the specified user variable to be initialized after the INITIALIZATION_HOOK() executes (i.e. after all program initialization has completed).

A timeout value can optionally be specified for a user variable by appending {time} to the statement. The value -1 sets INFINITY. For example, to set the timeout value for the user variable names uVAR1 to 1000mS:

```
C:\> ui /HC="/H:uVAR1=xxx{1000}
```

Example

These two examples perform identically when executed at a Windows command line. Two user variables are initialized, one before program initialization starts (uVAR1) and one after initialization had completed (uVAR2):

```
C:\> ui /U:ui_HostModeCommandLine="/H:uVAR1=xxx /h:uVAR2=yyy"
C:\> ui /HC="/H:uVAR1=xxx /h:uVAR2=yyy"
```

Given the following user variable definitions exist in the test program, the output noted below will occur when either of the examples above is used:

```
CSTRING_VARIABLE( uVAR1, "?", "") {output("uVAR1 => %s", uVAR1);}
CSTRING_VARIABLE( uVAR2, "?", "") {output("uVAR2 => %s", uVAR2);}
```

Output messages in Host window:

```
uVAR2 => yyy
```

```
The test program is loaded
uVAR1 => xxx
```

7.1.11.47 ui_HostTimeOut

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_HostTimeOut`.

Description

This UI user variable can be used to specify a Host timeout value.

Setting `ui_HostTimeOut` has the same effect as setting the value in UI:

`ui_HostTimeOut` can be used from a Windows command line, or from within a batch file (see `ui_BatchFile`). While it is possible to use `ui_HostTimeOut` from user-written C-code this has limited value. Setting a new value from C-code has no effect until the next test program is loaded.

The value specified is in `mS`. The default value is 10000 (10 seconds). The value INFINITE can be set using -1.

The value IS persistent after UI is terminated (via the UI.ini file).

From command line:

```
Long form:   /U:ui_HostTimeOut=<value>
```

```
Short form:  /HT=<value>
Short form:  /HT
```

The latter short form assumes value = -1 i.e. INFINITY.

From batch file:

```
ui_HostTimeOut=<value>
```

where **<value>** is in milliseconds, or -1 (INFINITY).

From C-code, intended usage is via `remote_set()` and `remote_get()`. The variable is only valid when sent to UI (site = -1). As noted above, this has limited value.

Example

These two examples perform identically when executed at a Windows command line. Both examples set the Host timeout value to the original default value:

```
C:\> ui /U:ui_HostTimeOut=10000
C:\> ui /HT=10000
```

This example sets the mode from user-written C-code:

```
remote_set("ui_HostTimeOut", 10000, -1 );
```

The following example will read back and output the state of this variable from user C-code:

```
CString val = remote_get("ui_HostTimeOut", -1 );
output(" Get ui_HostTimeOut => %s", val );
```

7.1.11.48 ui_LoadTimeOut

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_LoadTimeOut`.

Description

This UI user variable can be used to specify a program load timeout value.

Setting `ui_LoadTimeOut` has the same effect as setting the value in UI:

`ui_LoadTimeOut` can be used from a Windows command line, or from within a batch file (see `ui_BatchFile`). While it is possible to use `ui_LoadTimeOut` from user-written C-code this has limited value. Setting a new value from C-code has no effect until the next test program is loaded.

The value specified is in `mS`. The default value is -1 (INFINITE).

The value IS persistent after UI is terminated (via the UI.ini file).

If the program load timeout value is too small, some or all of the following error popups will be displayed

From command line:

```
Long form:   /U:ui_LoadTimeOut=<value>
Short form:  /LT=<value>
Short form:  /LT
```

The latter short form assumes value = -1 i.e. INFINITY.

From batch file:

```
ui_LoadTimeOut=<value>
```

where **<value>** is in milliseconds, or -1 (INFINITY).

From C-code, intended usage is via `remote_set()` and `remote_get()`. The variable is only valid when sent to UI (site = -1). As noted above, this has limited value.

Example

These two examples perform identically when executed at a Windows command line. Both examples set the program load timeout value to the original default value:

```
C:\> ui /U:ui_LoadTimeOut=-1
C:\> ui /LT=-1
```

This example sets the mode from user-written C-code:

```
remote_set("ui_LoadTimeOut", -1, -1 );
```

The following example will read back and output the state of this variable from user C-code:

```
CString val = remote_get("ui_LoadTimeOut", -1 );
output(" Get ui_LoadTimeOut => %s", val );
```

7.1.11.49 ui_LoadedMask

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_LoadedMask`.

Description

This UI user variable can be used to determine the Site(s) on which a test program is loaded.

`ui_LoadedMask` is only usable from user-written C-code. This is a read-only value from UI.

The value is NOT persistent after UI is terminated (via the UI.ini file).

Usage

From C-code, intended usage is via `remote_get()` only. The variable is only valid when sent to UI (site = -1). A bit mask is returned with a logic-1 indicating the test program is loaded on a given Site. The LSB is Site-1.

Example

The following example will read back and output the state of this variable from user C-code. Note that the value returned by `remote_get()` is a `CString` representation of the `DWORD` bit mask value of `ui_LoadedMask`:

```
CString val = remote_get("ui_LoadedMask", -1 );
output(" Get ui_LoadedMask => %s", val );
```

7.1.11.50 ui_MonitorPort

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_MonitorPort`.

Description

This UI user variable can be used to specify a Monitor Port value.

Setting `ui_MonitorPort` has the same effect as setting the value in UI:

`ui_MonitorPort` can be used from a Windows command line, or from within a batch file (see `ui_BatchFile`). While it is possible to use `ui_MonitorPort` from user-written C-code this has limited value. Setting a new value from C-code has no effect until the next test program is loaded.

The default monitor port for Site-1 is 2000.

The value is NOT persistent after UI is terminated (via the UI.ini file).

Usage

From command line:

```
Long form:   /U:ui_MonitorPort=<value>
Short form:  /MP=<value>
```

From batch file:

```
ui_MonitorPort=<value>
```

where **`<value>`** is the desired monitor port number.

From C-code, intended usage is via `remote_set()` and `remote_get()`. The variable is only valid when sent to UI (site = -1). As noted above, this has limited value.

Example

These two examples perform identically when executed at a Windows command line. Both examples set the Monitor Port value to the original default value:

```
C:\> ui /U:ui_MonitorPort=2000
C:\> ui /MP=2000
```

This example sets the monitor port from user-written C-code:

```
remote_set("ui_MonitorPort", 2000, -1 );
```

The following example will read back and output the state of this variable from user C-code:

```
CString val = remote_get("ui_MonitorPort", -1 );
output(" Get ui_MonitorPort => %s", val );
```

7.1.11.51 ui_MonitorTimeOut

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_MonitorTimeOut`.

Description

This UI user variable can be used to specify a Monitor Port value.

Setting `ui_MonitorTimeOut` has the same effect as setting the value in UI:

`ui_MonitorTimeOut` can be used from a Windows command line, or from within a batch file (see `ui_BatchFile`). While it is possible to use `ui_MonitorTimeOut` from user-written C-code this has limited value. Setting a new value from C-code has no effect until the next test program is loaded.

The default monitor timeout value is 5000. The value -1 can be used to set INFINITY.

The value IS persistent after UI is terminated (via the UI.ini file).

Usage

From command line:

```
Long form:   /U:ui_MonitorTimeOut=<value>
Short form:  /MT=<value>
Short form:  /MT
```

The latter short form assumes value = -1 i.e. INFINITY.

From batch file:

```
ui_MonitorTimeOut=<value>
```

where **<value>** is in mS, or -1 to set INFINITY.

From C-code, intended usage is via `remote_set()` and `remote_get()`. The variable is only valid when sent to UI (site = -1). As noted above, this has limited value.

Example

These two examples perform identically when executed at a Windows command line. Both examples set the monitor timeout value to the original default value:

```
C:\> ui /U:ui_MonitorTimeOut=5000
C:\> ui /MT=5000
```

This example sets the monitor timeout value from user-written C-code:

```
remote_set("ui_MonitorTimeOut", 5000, -1 );
```

The following example will read back and output the state of this variable from user C-code:

```
CString val = remote_get("ui_MonitorTimeOut", -1 );
output(" Get ui_MonitorTimeOut => %s", val );
```

7.1.11.52 ui_NoLogo

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_NoLogo`.

Description

This UI user variable is used to disable (skip) the initial UI logo screen.

`ui_nologo` is targeted at applications where UI is to be invoked from a command line or batch file and no user interface is desired.

The value is NOT persistent after UI is terminated (via the UI.ini file).

Usage

From command line:

```
Long form:   /U:ui_NoLogo=<value>
Short form:  /NOLOGO
```

From batch file:

```
ui_NoLogo=<value>
```

where **`<value>`** is 1 (TRUE) to disable the initial UI display.

Example

These two examples perform identically when executed at a Windows command line. Both examples inhibit the initial UI display:

```
C:\> ui /U:ui_NoLogo=1
C:\> ui /NOLOGO
```

7.1.11.53 ui_Open

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_Open`.

Description

This UI user variable allows User Tools to specify and load a test program.

This has the same effect as selecting **`File Open Test Program`** in UI, entering the desired test program name, and selecting **`Open`**. Note that UI must already be running to start User Tools.

If the specified test program is not found two or more (one Host and one for each Site) error popups, similar to that below, are displayed:

The value is NOT persistent after UI is terminated (via the UI.ini file).

Usage

Intended usage is via `remote_set()` and `remote_get()` from C-code within User Tools. The variable is only valid when sent to UI (site = -1). The value assigned is the file name of the test program executable file, with an optional path to the file. The default file name extension is `.exe`. If a complete path is not specified the environment PATH is searched for the test program.

Example

The following example will cause UI to load the test program named `my_prog.exe` located at *`d:\path\my_prog\Debug\`*.

```
remote_set( "ui_Open", "d:\path\my_prog\Debug\my_prog", -1 );
```

The following example retrieves and outputs the value of this user variable:

```
CString val = remote_get(  "ui_Open", -1 );
output("  Test program => %s", val );
```

7.1.11.54 ui_OutputAutoOpen

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_OutputAutoOpen`.

Description

This UI user variable can be used, via `remote_set()`, to control whether UI's output window will open (if closed) when a new message is displayed. This allows UI's output window to stay closed even though new messages have been displayed.

Note the following:

- The default state is set at the time UI is started. It is not otherwise changed by system software, including when a given program is loaded, unloaded, or a different program is loaded.
- The default state causes UI's output window to be automatically opened (displayed) any time an `output()`, `warning()` and `fatal()` is executed from user code and when any messages are sent from UI.
- `ui_OutputAutoOpen` does not open or close UI's output window. If the window is currently open, the auto-open state has no effect when a new message is displayed.
- UI knows whether `ui_OutputAutoOpen` is set from a Host, Site, or User Tools process, allowing UI to react differently to messages from each sender This allows, for example, Host messages to open the window while Site messages do not. To ensure that UI's output window remains closed in all situations requires setting `ui_OutputAutoOpen` from each Site, Host, and any User Tools.
- Setting `ui_OutputAutoOpen` from the Host process also determines whether UI's output window will open when a message is sent from UI. UI most commonly sends load/time and run-time error messages and test results at the end of each Sequence & Binning Table execution.
- When auto-open is FALSE, the `intercept()` function combined with `ui_OutputOpen` can be used to selectively over-ride this state for messages generated using `output()`, `warning()` and `fatal()`. See Example. Note that messages generated directly by UI are NOT displayed using `output()`, `warning()` or `fatal()`, thus the `intercept()` call-back function is not invoked for these messages.

Usage

```
BOOL_VARIABLE( ui_OutputAutoOpen, TRUE, "" )
```

`ui_OutputAutoOpen` can be set from user C-code using `remote_set()`. `remote_get()` can be used to read back the current value. The variable is only valid when sent to UI (site = -1):

```
remote_set ("ui_OutputAutoOpen", FALSE, -1);
```

Example

The following example will inhibit UI from opening its output window, when messages are sent from any site which executes this code:

```
remote_set ("ui_OutputAutoOpen", FALSE, -1);
```

7.1.11.55 ui_OutputFile

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_OutputFile`.

Note: `ui_OutputFile` supports using the %S token in place of the %d token. This was done for compatability with `ui_ShmooOutputFile`. For backwards compatibility the %d token continues to operate as previously documented, however the information below now only refers to the %S token.

Description

This UI user variable can be used to specify the path/file-name of a text file used to log the messages displayed in UI's Host and/or Site output windows.

`ui_OutputFile` can be used from a Windows command line, from within a batch file (see `ui_BatchFile`), or from user-written C-code.

Rules:

- ***BEWARE***: no checks are made to prevent over-writing (clobbering) an existing file of the same name. Example 5: shows how to create a file name containing a date-time stamp which prevents accidentally clobbering an existing file.
- Folders will be created i.e. if the specified path contains folders which do not exist they will be created. If a file exists with the same name as a folder a run-time error message will be displayed.
- It is NOT possible for more than one site (Host, Sites 1..n, or User Tools) to log to a given file. The **%S** token can be used in the path/file-name string assigned to `ui_OutputFile` to represent the `site_num()` of the process which executes the `remote_set()` to define `ui_OutputFile`. This will create a unique file, or folder, for Host vs. Site vs. User Tools messages. Using this method, `remote_set()` is called multiple times, each executing independently in each process. See Example 2: and Example 4:.

- Using remote_get() to read back the value of ui_OutputFile will return only the last value sent to UI, even though UI may have received several instances of ui_OutputFile to set up different log files for Host vs. Site vs. Tool.
- Setting a new value to ui_OutputFile will cause a currently open file to be closed.
- The actual output may be spooled in system RAM until the current Sequence & Binning Table execution is complete, at which time it is flushed to the output file.
- To terminate logging to a file either terminate UI, or from C-code use remote_set() to set ui_OutputFile to a NULL path/file-name value i.e.

  ```
  remote_set("ui_OutputFile", "", -1);.
  ```

- The value is NOT persistent after UI is terminated (via the UI.ini file).
- The scope of ui_OutputFile is the test program process. This means that closing a test program closes any open files. And, if a subsequent test program defines an identical log file path/file-name, the original file will be clobbered (overwritten).
- Regardless of the value of ui_OutputFile, output messages continue to be displayed in the UI Host and Site output windows.

An alternate, and more versatile, methodology is available using the intercept() and fumble() functions.

Usage

```
CSTRING_VARIABLE( ui_OutputFile, "", "" )
```

ui_OutputFile can be set from user-written C-code using remote_set(). remote_get() can be used to read back the current value. The variable is only valid when sent to UI (site = -1). UI knows whether the sender is a Host, Site, or Tool process and will process output messages correctly for each process.

```
remote_set ("ui_OutputFile", "path", -1);
```

It is possible to set ui_OutputFile from a command line or batch file using ui_HostModeCommandLine and/or ui_SiteModeCommandLine.

From command line, to define a Host output file:

```
Long/long form: /ui_HostModeCommandLine="/U:ui_OutputFile=<path>"
Short/long form:/HC=/U:ui_OutputFile=<path>
Short form:     /HOF=<path>
```

From command line, to define a Site output file:

```
Long/long form: /ui_SiteModeCommandLine="/U:ui_OutputFile=<path>"
Long/short form: /SC=/U:ui_OutputFile=<path>
Short form:  /SOF=<path>
```

From batch file:

```
ui_HostModeCommandLine="/U:ui_OutputFile=<path>"
ui_SiteModeCommandLine="/U:ui_OutputFile=<path>"
```

where **<path>** is the desired path/file-name of the log file.

Example

Example 1:

These three examples perform identically when executed at a Windows command line. These all use a form of ui_HostModeCommandLine to define an output log file for the Host process:

```
ui /HOF=D:\logfile.txt
ui /HC=/U:ui_OutputFile=D:\testlog.txt
ui /U:ui_HostModeCommandLine="/U:ui_OutputFile=D:\testlog.txt"
```

Example 2:

This example executes from a batch file (see ui_BatchFile). It uses a form of ui_SiteModeCommandLine to define a unique output log file for each Site process. Note the use of **%S** in the file name:

```
ui_SiteModeCommandLine="/U:ui_OutputFile=D:\testlog_%S.txt"
```

Example 3:

This example executes from user C-code. If executed from Host code this will cause all Host messages to be logged to the file. Similarly, if executed from Site-1, Site-2, etc. or User Tools code the output messages from those sites will be logged to the file.

```
remote_set ("ui_OutputFile", "C:/logfile.txt", -1);
```

However, because the file name is constant, this example will only function correctly when executed from one site (Host, or Site 1..n, or User Tools). Any attempt to execute this twice will result in a run-time error. See next example:

Example 4:

This example creates a file name containing the site_num() of the process which executes the code; note the **%S** token in the file name. Thus, this code can be executed, without change, in Host, and/or Site, and/or User Tools code with confidence that each process will specify a unique file name.

```
remote_set ("ui_OutputFile", "D:/logfile_%S", -1);
```

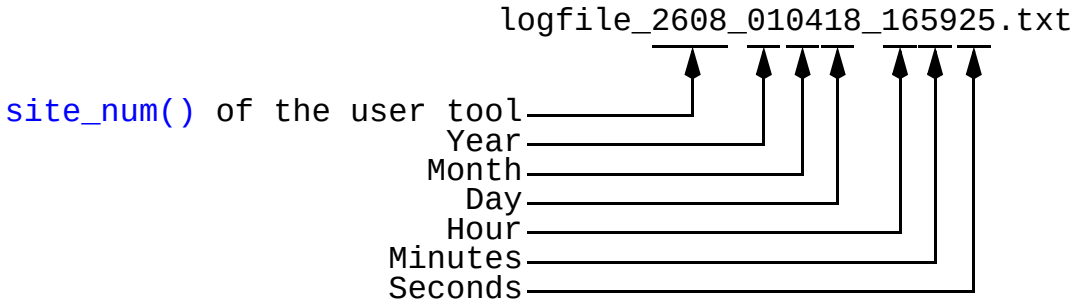
Example 5:

As noted above, no file clobber checks are done. To prevent over-writing an existing file the following example adds a date/time stamp to the file name.

```
CString s = "D:/logfile_%S_";
s += CTime::GetCurrentTime().Format("%y%m%d_%H%M%S.txt");
remote_set ("ui_OutputFile", s, -1);
```

Use the F1 key in Developer Studio to research CTime, and the format tokens seen above. Note that the **%d** used in the CTime format represents an integer day value (01..31) not the site_num() noted earlier.

Executing the code above from User Tools creates the following file name:



Note that the site_num() of User Tools is dynamic and > 1024.

7.1.11.56 ui_OutputFormat

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See ui_OutputFormat.

Note: ui_OutputFormat supports using the %S token in place of the %d token. This was done for compatability with ui_ShmooOutputFile. For backwards compatibility the %d token continues to operate as previously documented, however the information below now only refers to the %S token.

Description

This UI user variable can be used to add a prefix to messages output in UI's Host and/or Site windows.

Messages from Host and Site do not have a default prefix. Messages from User Tools have a default message prefix which can be modified using ui_OutputFormat. Adding a prefix using ui_OutputFormat affects all messages including those generated by system software.

The message prefix can include combinations of:

- User-defined text
- The site_num() of the process generating the message, using the **%%S** token
- Date and/or time information, using CTime tokens

The examples below show all three options.

The value is NOT persistent after UI is terminated (via the UI.ini file).

The scope of ui_OutputFile is the UI process. This means that closing a test program and loading a different program does not, in itself, change the state of ui_OutputFormat.

An alternate, and more versatile, methodology is available using the intercept() and fumble() functions.

Usage

```
CSTRING_VARIABLE( ui_OutputFormat, "", "" )
```

ui_OutputFormat can be set from user-written C-code using remote_set(). remote_get() can be used to read back the current value. The variable is only valid when sent to UI (site = -1). UI knows whether the sender is a Host, Site, or Tool process and will process output messages correctly for each process.

```
remote_set ("ui_OutputFormat", "prefix", -1);
```

It is possible to set ui_OutputFormat from a command line or batch file using ui_HostModeCommandLine and/or ui_SiteModeCommandLine.

From command line, to define a Host output file:

```
Long form:/U:ui_HostModeCommandLine="/U:ui_OutputFormat=<prefix>"
Short form:/HC="/U:ui_OutputFormat=<prefix>"
```

From command line, to define a Site output file:

```
Long form:/U:ui_SiteModeCommandLine="/U:ui_OutputFormat=<prefix>"
Short form:/SC="/U:ui_OutputFormat=<prefix>"
```

From batch file:

```
ui_HostModeCommandLine="/U:ui_OutputFormat=<prefix>"
ui_SiteModeCommandLine="/U:ui_OutputFormat=<prefix>"
```

where **<prefix>** is the desired prefix.

Example

Example 1:

The following example adds a fixed prefix (`Msg =>`) to any messages generated from any process which executes this code (Host, Site, or User Tools processes):

```
remote_set ("ui_OutputFormat", "Msg =>", -1);
```

Example 2:

This example uses the **%%S** token to add the `site_num()` of the process which executes this code (Host, Site, or User Tools process) to the message prefix:

```
remote_set ("ui_OutputFormat", "Site_%%S => ", -1);
```

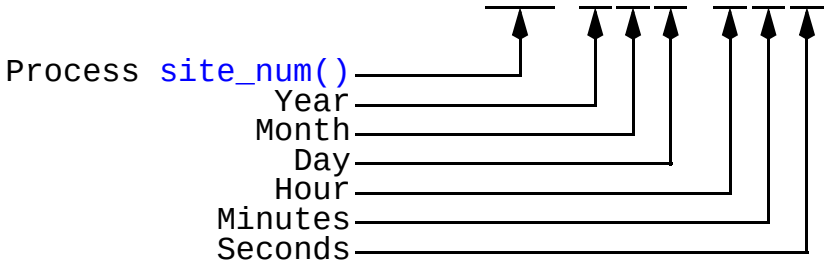
Example 3:

This example uses both the **%%S** token and tokens used by `CTime` (**%y**, **%m**, **%d**, **%H**, **%M**, **%S**) to create a prefix containing the `site_num()` of the process generating the message and a date/time stamp for each message generated:

```
remote_set ("ui_OutputFormat", "Site_%%S %y%m%d_%H%M%S: ", -1)
```

If this is executed from Host, Site and Tool code the following prefixs are typical. The **%%S** portion of the prefix will depend on which process generated the message:

```
Host process: Site_0 010102_093247: rest of message
Site process: Site_1 010102_093249: rest of message
Tool process: Site_2401 010102_093256: rest of message
```



Using this example note that the date/time information shown will change as each message is printed.

7.1.11.57 ui_OutputOpen

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_OutputOpen`.

Description

This UI user variable can be used, via `remote_set()`, to open or close UI's output window. Note the following:

- The current state of `ui_OutputAutoOpen` is ignored i.e. UI's output window will open or close independent of `ui_OutputAutoOpen`.
- The `intercept()` function combined with `ui_OutputOpen` can selectively over-ride the effect of `ui_OutputAutoOpen` for messages generated using `output()`, `warning()` and `fatal()`. See Example below.

Usage

```
BOOL_VARIABLE( ui_OutputOpen, TRUE, "" )
```

`ui_OutputOpen` can be set from user C-code using `remote_set()`. The variable is only valid when sent to UI (site = -1):

```
remote_set ("ui_OutputOpen", FALSE, -1); // Close the window
```

```
remote_set ("ui_OutputOpen", TRUE, -1); // Open the window
```

Example

The following example, selectively over-rides the operation of ui_OutputAutoOpen, to cause warning() and fatal() but not output() messages to be displayed when sent from the Host process (the Host Begin Block mechanism is used for convenience here):

```
HOST_BEGIN_BLOCK( HBB1) { // See HOST_BEGIN_BLOCK()
   // Register callback function in the Host processes
   intercept( myCallback );
   // Inhibit opening UI output window when messages from Host are
   // displayed
   remote_set( "ui_OutputAutoOpen", FALSE, -1 );
}
BOOL myCallback( char type, CString string ) {
   // Open UI's output window for warning() and fatal() but not
   // output()
   if( ! type == 'o')
      remote_set( "ui_OutputOpen", TRUE, -1 );
}
```

Note the following about this example:

- Since this example did not set ui_OutputAutoOpen from Site code, it must be assumed that messages from Site code will open UI's output window normally. However, if ui_OutputAutoOpen = FALSE was also sent to UI from Site code, this example would not have any effect when messages are sent from Site code, because the call-back is registered from the Host only.
- Since ui_OutputAutoOpen was sent from Host code, if UI's output window is closed, it will NOT be opened when messages generated by UI are displayed. Messages generated by UI are not displayed using output(), warning() or fatal() and the intercept() call-back function is not invoked to open the window.

7.1.11.58 ui_ProgLoaded

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See ui_ProgLoaded.

Description

This Callback UI User Variable is invoked by UI when a test program load has completed on all enabled sites.

If `ui_ProgLoaded` is defined in the test program and/or User Tools its body code will be executed once UI determines that test program load has completed on all enabled sites. This is typically used to synchronize code in the `HOST_BEGIN_BLOCK()` and/or `TOOL_BEGIN_BLOCK()` with site operation, see Host Waiting for Site to Load.

If any test site fails to load the test program, the `ui_ProgLoaded` body code will not be executed and the Host Process will be terminated. Similarly, `ui_ProgLoaded` body code will not be executed in User Tools but the tool will not be terminated.

A similar UI user variable reports when each site has completed loading the test program. See `ui_SiteLoaded`.

Usage

```
VOID_VARIABLE( ui_ProgLoaded, "" ) { [body code] }
```

where:

`ui_ProgLoaded` is a Callback UI User Variable.

`body` is the C-code the user adds to the body of `ui_ProgLoaded` in their test program or User Tools.

Examples

Example 1:

This simple example outputs a message when `ui_ProgLoaded` is invoked by UI. This code will execute in the Host process if `ui_ProgLoaded` is defined in the test program. This code will execute in the Tool process if `ui_ProgLoaded` is defined in User Tools.

```
VOID_VARIABLE( ui_ProgLoaded, "") {
   output( "Received ui_ProgLoaded notification from UI" );
}
```

Example 2:

This example is typical of a synchronization implementation, where code in the `HOST_BEGIN_BLOCK()` needs to wait for the test program to completely load on all used sites. The body code of `ui_ProgLoaded` executes when UI sends the `ui_ProgLoaded` notification. The `HOST_BEGIN_BLOCK()` code is waiting to receive a signal named

my_ProgLoaded_signal which is generated when the body code of ui_ProgLoaded executes:

```
VOID_VARIABLE( ui_ProgLoaded, "" ) {
   // Send a signal to corresponding remote_wait() in
   // HOST_BEGIN_BLOCK(), which is waiting for signal before
   // continuing.
   remote_signal( "my_ProgLoaded_signal", 0 );
}
HOST_BEGIN+BLOCK( HBB1 ) { // See HOST_BEGIN_BLOCK()
   // ... other code here ...
   // Wait here for the signal indicating all used sites have
   // completed program loading.
   remote_wait( "my_ProgLoaded_signal", INFINITE );
   output( "Test program is loaded. Continue execution..." );
   // ... other code here ...
}
```

Example 3:

This example is quite complex, and demonstrates several advanced features:

- ui_LoadedMask
- ui_ProgLoaded
- ui_SiteMask
- A User Tools, which appears only as a menu in UI
- Using builtin_dynload the tool code causes the test program to load a DLL required by the tool
- How to create a tool and its required DLL in a single Project Workspace

This example is divided into several sections

- User Tool code
- DLL code
- Creating the Project Workspace for both the tool and the DLL

User Tool code

```
#include "TestProgApp/public.h"
#include <sys/types.h>// For _stat
#include <sys/stat.h>// For _stat
```

```
#define MAXSITES 60

// Code below causes the site(s) to load the required DLL specified
// by myDLL. File exists is tested in code below.

#define myDLL "D:/Binning_tool_dll/Debug/Binning_tool_dll.dll"

INT64_VARIABLE( ui_SiteMask, 0, "") {} // To be remote_fetched

// Using remote_set(), DLL code sets the value of the bin
// specified using get_bin_val into one_bin_value
INT_VARIABLE( one_bin_value, 0, "" ) {}

//=============================================================

// Output (in Host window) all TestBin values from each used site
void OutputBinValues( void ) {
   remote_fetch( ui_SiteMask, -1, FALSE );

   for (int site = 0; site < MAXSITES; site++) {
      // Get list of all TestBins but only from used sites
      if ((ui_SiteMask & ((UINT64) 1 << site)) )
         CStringArray bin_names;

         int bin_count =
            resource_all_names( S_TestBin, (site + 1), &bin_names );

         for ( int i = 0; i < bin_count; ++i ) {

            // Request site update uVar named one_bin_value
            // uVar one_bin_value was put into test program via DLL
            remote_set(  "get_bin_val",
                         bin_names[i],
                         (site + 1),
                         TRUE,
                         INFINITE );

            output( "  Site=> %d : TestBin => %s = %d",
                         (site + 1),
                         bin_names[i],
                         one_bin_value );

         }
      }
   }
}
```

```
//=============================================================
// Cause used sites to load the required DLL
void LoadDLLonSites( void ) {
   remote_fetch(  "ui_SiteMask",
                  -1,
                  FALSE ); // Bitmask of used sites
   for (int site = 0; site < MAXSITES; site++) {
      if ((ui_SiteMask & ((UINT64) 1 << site)) )
         remote_set(  "builtin_dynload",
                      myDLL,
                      (site + 1),
                      TRUE,
                      INFINITE );
   }
}
//=============================================================
// If the tool was started before a test program is loaded
// ui_ProgLoaded will cause the used sites to load the required
// DLL. Otherwise, code in the TOOL_BEGIN_BLOCK will do it.
VOID_VARIABLE( ui_ProgLoaded, "" ) {
   LoadDLLonSites();
   OutputBinValues();
}
//=============================================================
// Check if the required DLL is found on disk and is a file
BOOL CheckDllFound( char fname[] ) {
   struct _stat info;
   if (  (! _stat( fname, &info) == 0) ||       // Exists on disk
         (! (info.st_mode & _S_IFMT & _S_IFREG))// Is a file
      ) {
      return (FALSE);
   }
   return (TRUE);
}
//=============================================================
// Body code executes when menu item is selected
```

```
CSTRING_VARIABLE( DoBinSummary,"","DoBinSummary") {
   OutputBinValues();
}
// TOOL_BEGIN_BLOCK executes when tool first starts. Confirms DLL
// is found. If test program is already loaded causes it to load
// the required DLL. Inserts a menu in UI used to invoke the bin
// summary
TOOL_BEGIN_BLOCK( TB1 ) {
   // If the required DLL is not found on disk tell the user and
   // terminate the tool.
   if (! CheckDllFound( myDLL ) ) {
      CString err = "ERROR: required DLL not found\nFile => ";
             err += vFormat("%s\nAborting", myDLL);
      AfxMessageBox( err );
      fatal( err ); // Message in Host output window too.
   }
   // If the test program was loaded before this tool was started
   // tell the sites to load the required DLL. Otherwise,
   // ui_ProgLoaded body code (above) does it.
   CString val = remote_get("ui_LoadedMask", -1 );
   if(! val.IsEmpty() ) LoadDLLonSites();
   // Add menu to UI.
   menu_add( "User Menu/DoBinSummary", DoBinSummary );
}
```

DLL code

This DLL is inserted into the test program by tool code calling `remote_set()` of `builtin_dynload`. If the DLL doesn't load correctly all bins will always display a count = 0:

```
#include "TestProgApp/public.h"
// Binning_tool DLL code. The get_bin_val uVar is inserted
// into the test program by code in the tool. Used to send the value
// of one bin, specified by get_bin_val (set by the sender), back to
// the sender
CSTRING_VARIABLE( get_bin_val, "", "" ) { // SITE execution
   TestBin *bin = TestBin_find( get_bin_val );
   int value = get( bin );
```

```
   // Return to sender. No need to invoke body code
   remote_set( "one_bin_value", value, sender, FALSE );
}
// Outputs message in each Site window when DLL is loaded
INITIALIZATION_HOOK()( IH1 ) {
   output( "  The required Binning_tool DLL was loaded" );
}
// Next line forces Developer Studio to generate a .lib file. This
// ensures Rebuild All will build both the tool and the DLL.
__declspec( dllexport ) int force_lib_generation = 0;
```

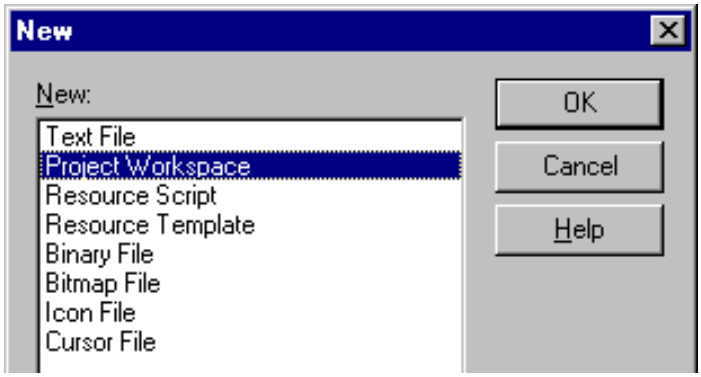
Creating the Project Workspace

This demonstrates the methods used to enable several advanced techniques not normally used when writing test programs, including:

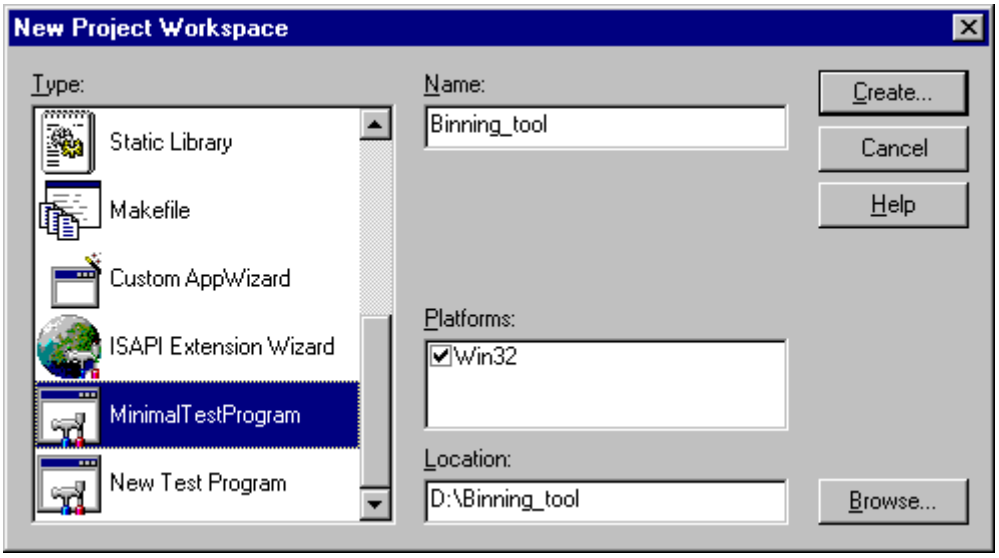
- User Tools, contained in single source file
- A DLL, also in a single source file, which contains code required by the tool which must execute on the site(s) as part of the test program, but which is not included in the test program. In User Tool code above, tool code causes the test program to load the DLL.
- Both the user tool and the DLL are part of a single Project Workspace. The DLL is set up as a subproject of the tool. This causes both the tool and the required DLL to be built when **Build Rebuild All** is invoked.

The process begins by creating a new minimal test program, adding the tool code, configuring the project, and compiling the tool. After this is successful, the DLL will be added to the project and compiled.

1. In Developer Studio invoke **File New**, select Project Workspace, and click **OK**:

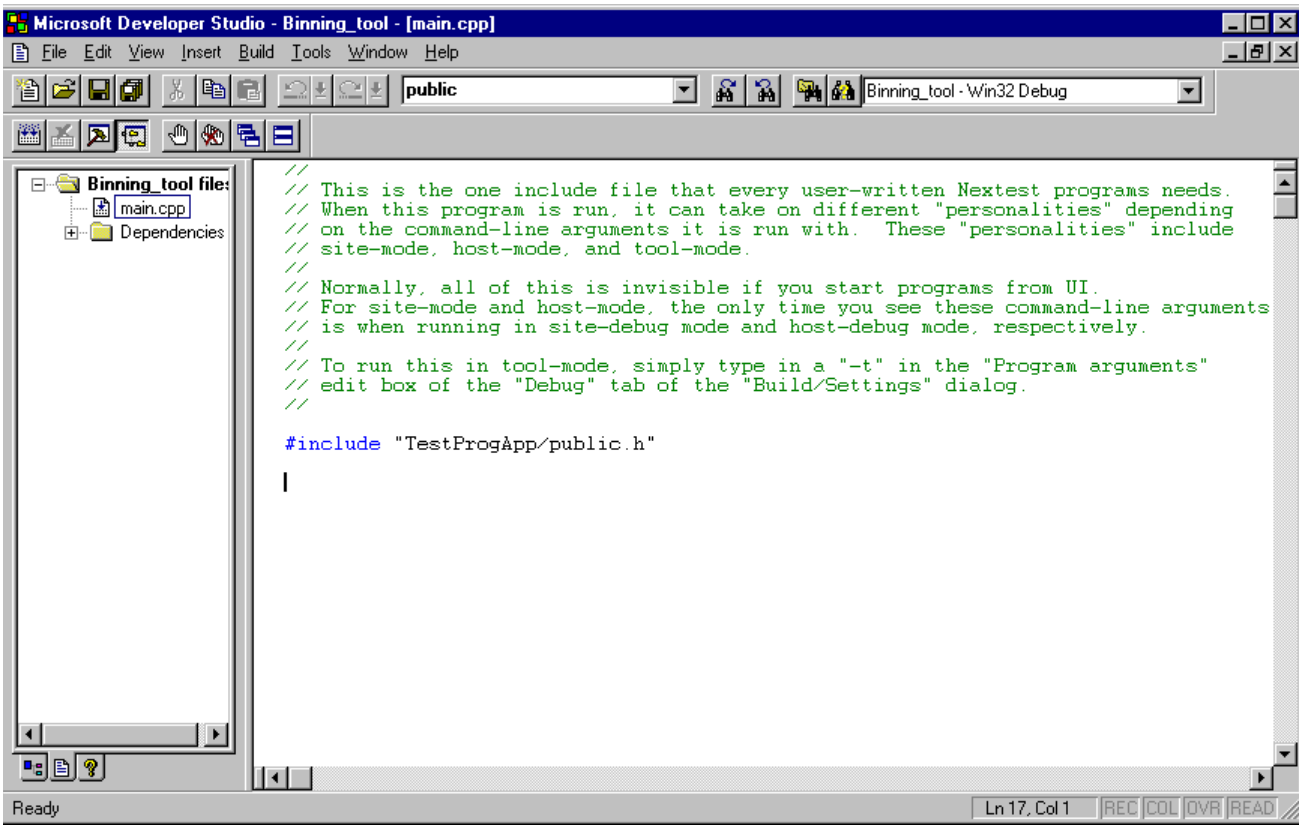


2. In the Project Workspace dialog, select *Minimal Test Program*, enter the name of the tool (`Binning_tool`), and specify the location for the project (`D:\Binning_tool`). Click **`Create`**. You can choose other names and locations, but this document uses these:



3. Read the *New Project Information* messages and click **`OK`** when done.

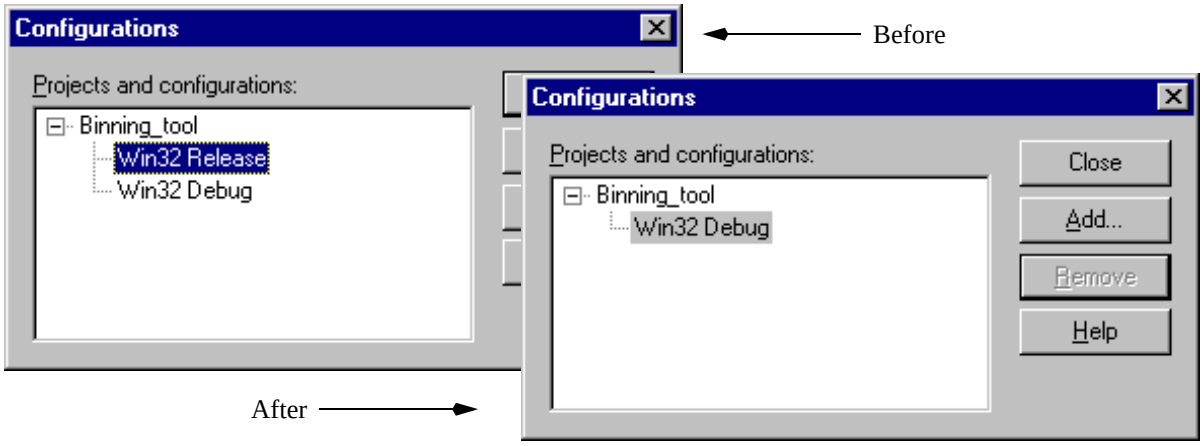
4. The project created should appear as shown below. Note that the main.cpp file contains only comments and one line of code. This file will be removed later after the User Tool code is added to the project.



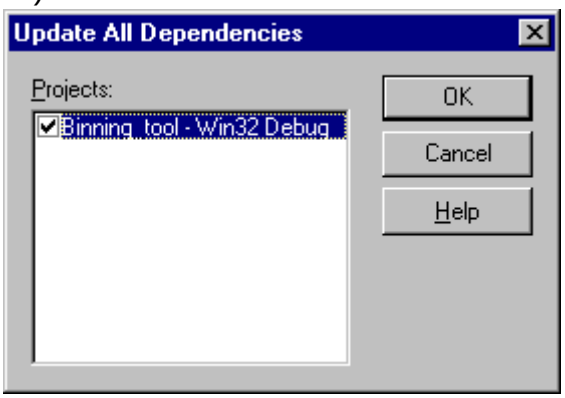
5. In Developer Studio, add the User Tool code to the project:
   - open a new text file: **File New**, select Text File.
   - Copy the User Tool code above into this file.
   - Save the file to the new Project Workspace folder: **File Save As...** The rest of these instructions assume the file is named *Binning_tool.cpp*
6. Add the *Binning_tool.cpp* file to the project:
   - **Insert Files Into Project...** Select the *Binning_tool.cpp* file and click **Add**.
   - Note that the *Binning_tool.cpp* file appears in the Project Workspace window, just above *main.cpp*.

7. In the Project Workspace window, select the *main.cpp* file and delete it from the list. This should leave only *Binning_tool.cpp* in the list as shown:

8. Remove the Win32 Release configuration (to reduce confusion):
   - Select **Build Configurations...**
   - In the Configurations dialog, select Win32 Release and click **Remove**.
   - Click **Yes** when asked to confirm.



9. Select **Build Update All Dependencies...** Click **OK**. This updates the list of dependency files (.h files):



10. Edit the Binning_tool.cpp file and change the path the DLL in your version of this tool

```
#define myDLL "D:/Binning_tool/Binning_tool_dll/Debug/Binning_tool_dll.dll"
```

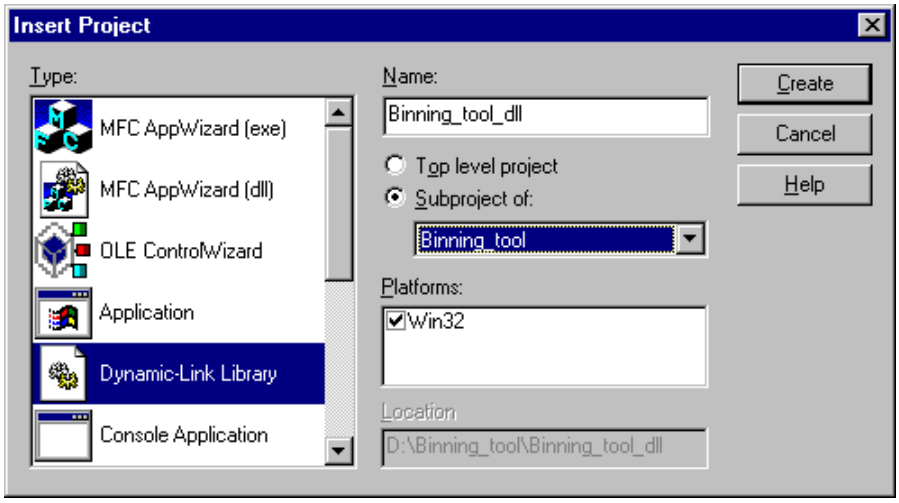
11. Compile the tool:

- **Build Rebuild All**
- No errors should occur.

At this point, the tool has been created, and compiled but it cannot be used until the required DLL is also created. The next step is to add the DLL code to the project, as a subproject of the tool.

- Insert the DLL as new project:
- **Insert Project...**
- Select Dynamic Link Library
- Add the name of the DLL (*Binning_tool_dll*)

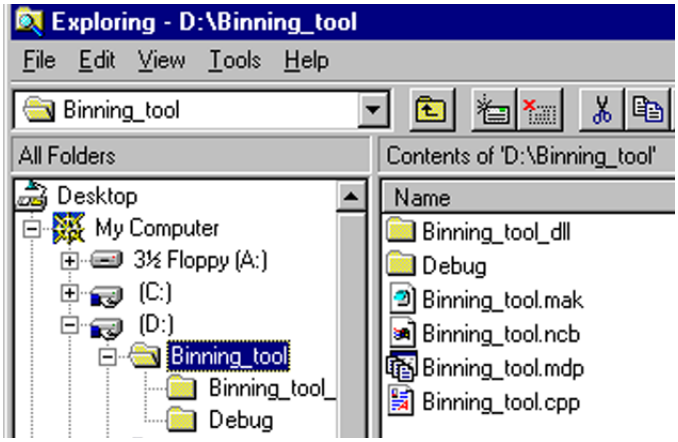
- Select the **Subproject of** radio button and note that *Binning_tool* appears in the selection box. The *Insert Project* dialog should appears as shown:



- Click **Create**

At this point the Project Workspace will appear as follows:

And, a look at the disk will show:

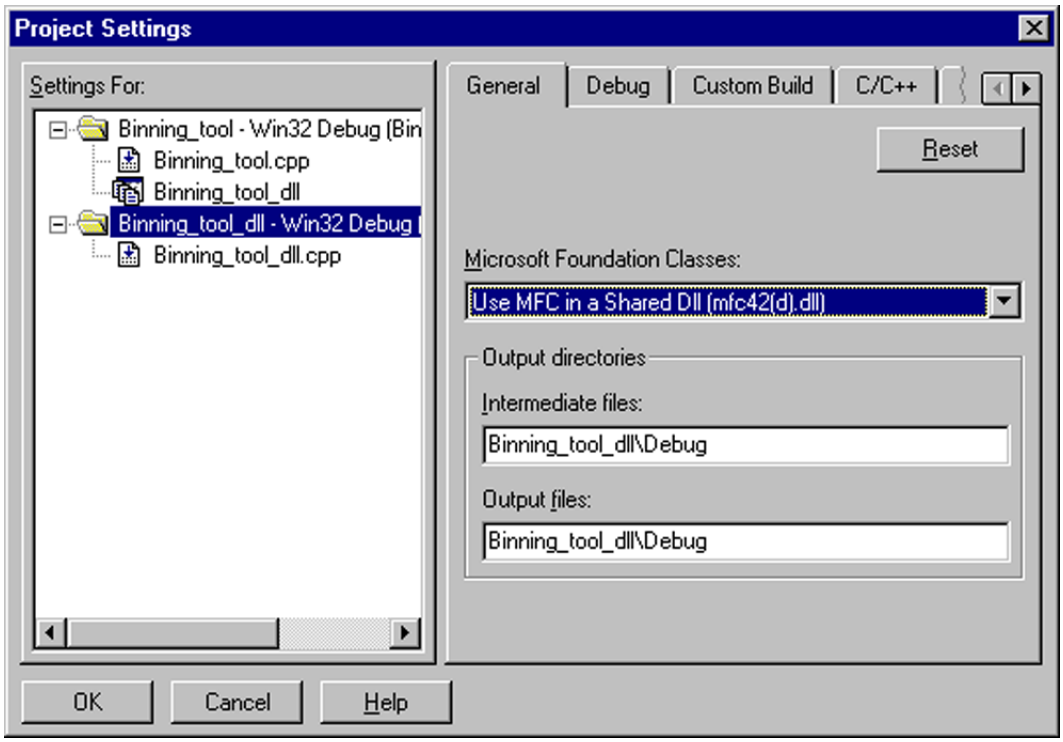


12. Add the DLL source code to the DLL project:
    - In the Developer Studio open a new text file: **File New**, select *Text File.*
    - Copy the DLL code into this file.
    - Save the file: **File Save As...**. In this example, the file is named *Binning_tool_dll.cpp.*
13. Insert this file into the subproject:
    - In the Project Workspace window select the *Binning_tool_dll* subproject
    - Invoke **Insert Files into Project...**, select *Binning_tool_dll.cpp*, and click **Add**.
    - At this point the Project Workspace will appear as follows:

14. Similar to step-8 above, remove the *Win32 Release* configuration from the *Binning_tool_dll* subproject (reduces confusion).

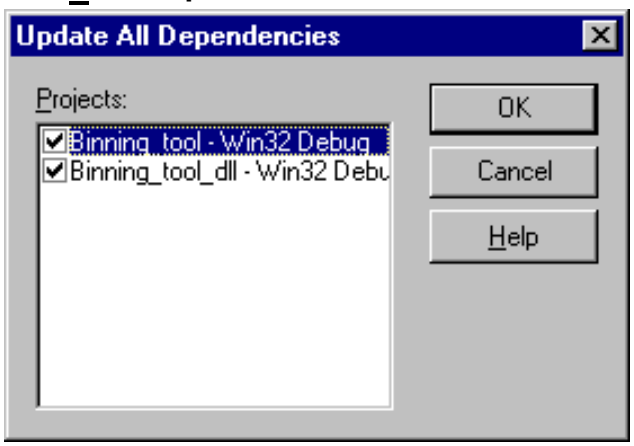
15. For this subproject change the Microsoft Foundation Class to Shared DLL:

    - Invoke **Build Settings...**
    - Click the General tab
    - In the `Settings For` window select *Binning_tool_dll - Wind32 Debug*
    - In the `Microsoft Foundation Classes` list select *Use MFC in a Shared Dll*
    - The display will appear as follows:



    - Click **OK**

16. Invoke **Build Update All Dependencies...** and select both boxes:

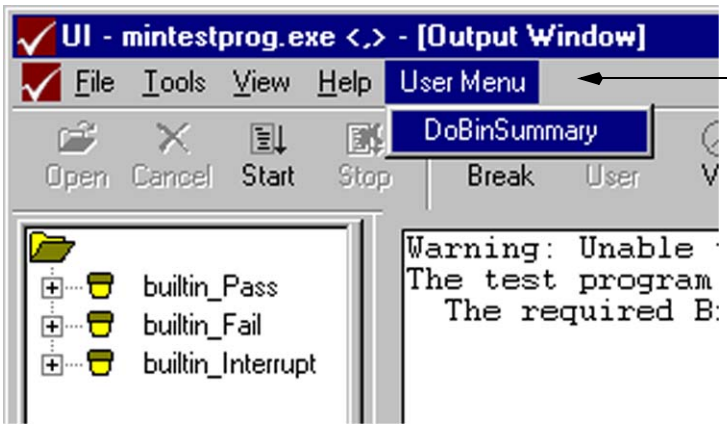


Click **OK**

17. Invoke **Build Rebuild All** to compile both the DLL and the tool. There should be no errors.

The tool is now usable:

- Start UI, load any test program
- Start the tool (see Starting/Terminating User Tools). This will insert `User Menu` into UI as shown:



Menu added by tool code.

- Invoke **File Start Testing** a few times then invoke the *User Menu / DoBinSummary* option. The output is in UI's Host output window:

```
[Tool 2186]:   Site=> 1 : TestBin => builtin_Pass = 8
[Tool 2186]:   Site=> 1 : TestBin => builtin_Fail = 0
[Tool 2186]:   Site=> 1 : TestBin => builtin_Interrupt = 0
```

This output is typical when using a single site test system. The number of actual bins displayed will depend on the test program. The 3 bins shown above are the built-in bins, automatically included in every test program by the system software.

Note that the prefix [Tool 2186]: is automatically added to any output generated by User Tools. The number is the site number of the tool, which is dynamically assigned. This prefix can be modified using `ui_OutputFormat`.

7.1.11.59 ui_ProgUnloaded

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_ProgUnloaded`.

Description

This Callback UI User Variable is invoked by UI when a test program has been unload on all enabled sites.

If `ui_ProgUnloaded` is defined in the test program and/or User Tools its body code will be executed once UI determines that a loaded test program has been unloaded on all enabled sites. This can be used to notify code in User Tools when a test program has been unloaded.

A similar UI user variable reports when a given site has reported the test program is unloaded. See ui_SiteUnloaded.

A related UI user variable reports when all sites have reported the test program is loaded. See ui_ProgLoaded.

Usage

```
VOID_VARIABLE( ui_ProgUnloaded, "" ) { [body_code] }
```

where:

`ui_ProgUnloaded` is a Callback UI User Variable.

`body_code` is user-written C-code added to the body of **`ui_ProgUnloaded`**.

Example

This example can be included in User Tools code to execute when UI notifies the tool that the test program has been unloaded:

```
VOID_VARIABLE( ui_ProgUnloaded, "") {
```

```
    // ... code here to execute when a test program is unloaded.
}
```

7.1.11.60 ui_ResourceInitialized

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_ResourceInitialized`.

Description

This Callback UI User Variable is invoked in the host process as each resource is loaded on each enabled test site. This can be used to track program load progress, but is quite verbose. An Example Output is included below, from a test program executing on a single site system (PT).

If `ui_ResourceInitialized` is defined in the test program its body code will be executed once, in the host process, for each resource loaded on a site. Each time `ui_ResourceInitialized` is invoked its value will reflect the resource just loaded. The `sender` parameter can be used within the body code to determine which site caused `ui_ResourceInitialized` to be invoked (see example).

Usage

```
CSTRING_VARIABLE( ui_ResourceInitialized, "", "" ) { [body code] }
```

where:

`ui_ResourceInitialized` is a Callback UI User Variable.

body is the C-code the user adds to the body of `ui_ResourceInitialized` in their test program.

Example

This simple example outputs a message in the Host output window as each resource is loaded on each enabled test site. Below the code is an Example Output:

```
CSTRING_VARIABLE( ui_ResourceInitialized, "", "x" ) {
  output( "ui_ResourceInitialized on Site[%d] => %s",
          sender,
          ui_ResourceInitialized );
```

```
}
```

Example Output

```
ui_ResourceInitialized on Site[1] => Variable_int
ui_ResourceInitialized on Site[1] => Variable_BOOL
ui_ResourceInitialized on Site[1] => Variable_DWORD
ui_ResourceInitialized on Site[1] => Variable_float
ui_ResourceInitialized on Site[1] => Variable_double
ui_ResourceInitialized on Site[1] => Variable_int64
ui_ResourceInitialized on Site[1] => Variable_CString
ui_ResourceInitialized on Site[1] => Variable_OneOf
ui_ResourceInitialized on Site[1] => Variable_void
ui_ResourceInitialized on Site[1] => TesterHW
ui_ResourceInitialized on Site[1] => ATCBoardList
ui_ResourceInitialized on Site[1] => AVSPinList
ui_ResourceInitialized on Site[1] => Variable_int
ui_ResourceInitialized on Site[1] => Variable_BOOL
ui_ResourceInitialized on Site[1] => Variable_DWORD
ui_ResourceInitialized on Site[1] => Variable_float
ui_ResourceInitialized on Site[1] => Variable_double
ui_ResourceInitialized on Site[1] => Variable_int64
ui_ResourceInitialized on Site[1] => Variable_CString
ui_ResourceInitialized on Site[1] => Variable_OneOf
ui_ResourceInitialized on Site[1] => Variable_void
ui_ResourceInitialized on Site[1] => Dialog
ui_ResourceInitialized on Site[1] => Snapshot
ui_ResourceInitialized on Site[1] => Configuration
ui_ResourceInitialized on Site[1] => SiteConfiguration
ui_ResourceInitialized on Site[1] => PinAssignments
ui_ResourceInitialized on Site[1] => PinScramble
ui_ResourceInitialized on Site[1] => PEBoardList
```

```
ui_ResourceInitialized on Site[1] => CurrentShare
ui_ResourceInitialized on Site[1] => PinList
ui_ResourceInitialized on Site[1] => VihhMap
ui_ResourceInitialized on Site[1] => TestBin
ui_ResourceInitialized on Site[1] => TestBinGroup
ui_ResourceInitialized on Site[1] => TestBlock
ui_ResourceInitialized on Site[1] => BeforeTestingBlock
ui_ResourceInitialized on Site[1] => AfterTestingBlock
ui_ResourceInitialized on Site[1] => SequenceTable
ui_ResourceInitialized on Site[1] => ScanPattern
ui_ResourceInitialized on Site[1] => LogicVector
ui_ResourceInitialized on Site[1] => SiteBeginBlock
ui_ResourceInitialized on Site[1] => Pattern
ui_ResourceInitialized on Site[1] => PatternSet
ui_ResourceInitialized on Site[1] => SiteEndBlock
ui_ResourceInitialized on Site[1] => InitializationHook
ui_ResourceInitialized on Site[1] => Resource
```

7.1.11.61 ui_RunTestProgram

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_RunTestProgram`.

Description

This UI user variable can be used to specify the number of times the Sequence & Binning Table is to execute.

This is targeted at applications where UI and the test program are invoked from a command line or a batch file (see `ui_BatchFile`). The value can be set or read back using `remote_set()` and `remote_get()` from test program code or from User Tools code.

This can be combined with `ui_CloseAfterRun` to cause the program to unload after executing, or `ui_ExitAfterRun` to terminate UI after executing.

The value is NOT persistent after UI is terminated (via the UI.ini file).

Usage

From command line:

```
Long form:   /U:ui_RunTestProgram=<value>
Short form:  /R=<value>
Short form:  /R
```

The latter short form assumes value = 1.

From batch file:

```
ui_RunTestProgram=<value>
```

where **<value>** is the number of times the Sequence & Binning Table is to execute.

From C-code, intended usage is via `remote_set()` and `remote_get()`. The variable is only valid when sent to UI (site = -1).

Examples

Example 1:

These three examples perform identically when executed at a Windows command line. In all three cases UI will start normally. After the user loads a test program and invokes **File Start Testing** the Sequence & Binning Table will execute once and the program will unload:

```
C:\> ui /U:ui_CloseAfterRun=1 /U:ui_RunTestProgram=1
C:\> ui /CR=1 /R=1
C:\> ui /CR /R
```

This example sets `ui_RunTestProgram` from user-written C-code:

```
remote_set("ui_RunTestProgram", 1, -1 );
```

The following example will read back and output the state of this variable from user C-code:

```
CString val = remote_get("ui_RunTestProgram", -1 );
output(" Get ui_RunTestProgram => %s", val );
```

Example 2:

The following is an example of a batch file which will start UI, load the test program *D:/MinTestProg/Debug/MinTestProg*, execute the Sequence & Binning Table 3 times, and then unload the test program:

```
ui_NoLogo=1
ui_CloseAfterRun=1
ui_RunTestProgram=3
ui_TestProgName=D:/MinTestProg/Debug/MinTestProg
```

If these statements are in the file *C:\test3_batch.txt* the batch file can be executed from the command line using

```
C:\ui /BATCH=C:\test3_batch.txt
```

7.1.11.62 ui_ShmooDone

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See ui_ShmooDone.

Note: first available in software release h1.1.23.

Description

This UI user variable is used by UI to notify Site processes when a search/shmoo execution has completed, typically after executing a shmoo/search defined using ShmooTool / SearchTool and which is triggered by a breakpoint set using BreakpointTool (see Shmoo/ Search Execution). This is targeted for use in conjunction with search_results_get().

If this user variable is defined in the test program its body code will be executed, on sites only, any time UI determines that a shmoo or search execution has completed.

Usage

```
CSTRING_VARIABLE( ui_ShmooDone, "", "" )
```

Example

```
CSTRING_VARIABLE( ui_ShmooDone, "", "" ){
   SearchResultArray result;
   int c = search_results_get( &results ); // search_results_get();
}
```

7.1.11.63 ui_ShmooInput

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_ShmooInput`.

Description

This UI user variable can be used to specify the path/file-name of a shmoo/search definition file to load when the next test program is loaded. For file format and other rules see Shmoo Definition File.

This is targeted at applications where UI and the test program are invoked from a command line or a batch file (see `ui_BatchFile`). `ui_ShmooInput` can be accessed from C-code both to set its value and/or get the current value however this has limited value because setting a new value does not take effect until the current test program is unloaded and another program loaded, within the same instance of UI.

The value is NOT persistent after UI is terminated (via the UI.ini file).

Usage

```
CSTRING_VARIABLE( ui_ShmooInput, "", "" )
```

From command line:

```
Long form:   /U:ui_ShmooInput=<path>
Short form:  /SI=<path>
```

From batch file:

```
ui_ShmooInput=<path>
```

where **<path>** is the path/file-name to a previously created Shmoo Definition File. When an absolute path is not specified the environmental `PATH` is searched.

From C-code, intended usage is via `remote_set()` and `remote_get()`. The variable is only valid when sent to UI (site = -1).

Examples

Example 1:

These two examples perform identically when executed at a Windows command line:

```
C:\> ui /U:ui_ShmooInput=C:/myprogram/shmoodefs1.txt
C:\> ui /SI=C:/myprogram/shmoodefs1.txt
```

This example sets `ui_RunTestProgram` from user-written C-code:

```
remote_set("ui_ShmooInput", "C:/myprogram/shmoodefs1.txt", -1 );
```

The following example will read back and output the state of this variable from user C-code:

```
CString val = remote_get("ui_ShmooInput", -1 );
output(" Get ui_ShmooInput => %s", val );
```

Example 2:

The following is an example of a batch file which will start UI, enable Engineering Mode, load the test program *D:/MinTestProg/Debug/MinTestProg*, and load the Shmoo Definition File at *`D:/UIVar_tool/Shmoo1.txt`*:

```
ui_NoLogo=1
ui_EngineeringMode=1
ui_ShmooInput=D:/UIVar_tool/Shmoo1.txt
ui_TestProgName=D:/MinTestProg/Debug/MinTestProg
```

If these statements are in the file *`C:\test4_batch.txt`* the batch file can be executed from the command line using

```
C:\ui /BATCH=C:\test4_batch.txt
```

7.1.11.64 ui_ShmooOutputFile

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_ShmooOutputFile`.

Description

This UI user variable can be used to specify the path/file-name of a file used to capture an ASCII version of shmoo/search output. By default, ShmooTool generates a shmoo output window per-controller (per-site). If Multi-DUT Shmoos are enabled a window is generated

per-DUT, per-controller (per-site). Multi-DUT Shmoos are always enabled in Multi-DUT Test Programs.

Note the following:

- `ui_ShmooOutputFile` must only be set from Site code, using `remote_set()`. Setting `ui_ShmooOutputFile` from the Host or User Tool code is silently ignored.
- The `%S` token can be used in the file name specification to add the Site number generating the shmoo to the file name. Using the `%S` token does not cause any additional output files to be created.
- When Multi-DUT Shmoos are enabled, the `%D` token can be used in the file name specification to add the DUT number to the file name. See Example 2: and Example 3:. Using the `%D` token causes an additional output file to be created for each DUT in the test program. Note that no shmoo is sent to any file(s) for DUT(s) which are inactive at the time the scmoo is executed; i.e. the output file may be empty.
- If `%D` is used in when Multi-DUT Shmoos are not enabled the character '1' is added to the file name anyway.
- When Multi-DUT Shmoos are enabled and `%D` is not explicitly added to the file name specification the system software appends `_%D` to the file name, prior to any file name extension. And, a file is opened for each DUT, as noted above.
- Except as noted in the previous bullet, the system software does not add any under-score characters to the file name.
- The system software does not add a file extension to the file name.
- File(s) are created on disk at the time is `ui_ShmooOutputFile` sent to UI, using `remote_set()`.
- Files are closed several ways:
  - Setting `ui_ShmooOutputFile` with an empty file name.
  - Setting `ui_ShmooOutputFile` with a different file name closes the previous file(s).
  - The test program is unloaded.
- Existing files will be silently over-written (clobbered) when `ui_ShmooOutputFile` is set with a file name specification which matches existing files on disk.
- When using `ui_ShmooOutputFile` the actual shmoo output may be spooled in system RAM until the currently executing shmoo is complete, at which time it is flushed to the output file.
- The value is NOT persistent after UI is terminated (via the UI.ini file).

Usage

```
CSTRING_VARIABLE( ui_ShmooOutputFile, "", "" )
```

ui_ShmooOutputFile must only be set from user-written C-code, executed on the Site, using remote_set(). remote_get() can be used to read back the current value. The variable is only valid when sent to UI (Site = -1).

```
remote_set ("ui_ShmooOutputFile", "path", -1);
```

Examples

Example 1:

This example sets ui_ShmooOutputFile from Site code:

```
CString fname = "D:/shmoo_out.txt";
remote_set( "ui_ShmooOutputFile", fname, -1 );
```

This example gets the current value of ui_ShmooOutputFile. This could be executed from Host, Site or User Tools code:

```
CString val = remote_get("ui_ShmooOutputFile", -1 );
output(" Get ui_ShmooOutputFile => %S", val );
```

Example 2:

This example uses the %S token to add the Site number generating each shmoo to the file name. Then, assuming multiple sites execute the same shmoo, each will be saved to a file with its name made unique by including the site number:

```
remote_set( "ui_ShmooOutputFile", "D:/shmoo_out_%S.txt", -1 );
```

Using this example the file names generated will be:

```
D:/shmoo_out_1.txt       for Site-1
D:/shmoo_out_2.txt       for Site-2
Etc.
```

Example 3:

When Multi-DUT Shmoos are enabled the %D token can be used to represent the DUT number to the file name. For example:

```
remote_set( "ui_ShmooOutputFile", "D:/shmoo_out_%S_%D.txt", -1 );
```

Using this example the file names generated will be:

```
D:/shmoo_out_1_1.txt     for Site-1 DUT-1
D:/shmoo_out_1_2.txt     for Site-1 DUT-2
D:/shmoo_out_2_1.txt     for Site-2 DUT-1
Etc.
```

7.1.11.65 ui_ShowOutputTab

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_ShowOutputTab`.

Description

This UI user variable can be used to select a specific tab in UI's output window.

The `value` assigned determines which output tab is selected. The `value` matches the site number paradigm, where:

- 0 = Host tab
- 1 = Controller-1 (Site-1) tab
- 2 = Controller-2 (Site-2) tab
- etc.

Values up to 60 are valid (assuming a Magnum 1 with all 60 sites in use).

Usage

`ui_ShowOutputTab` can only be set from user-written C-code using `remote_set()`. The variable is only valid when sent to UI (site = -1). Other rules apply, see Description:

```
remote_set ("ui_ShowOutputTab", value, -1);
```

where **`value`** identifies which tab is selected. See Description.

Example

In the following example, the Controller-2 (Site-2) tab is selected in UI's output window:

```
remote_set( "ui_ShowOutputTab", 2, -1 );// Send to Ui
```

7.1.11.66 ui_Show

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See ui_Show.

Note: first available in software release h1.1.23.

ui_Show may be added to User Tools managed by ToolLauncher, to enable ui_HideTool and ui_ShowTool to control the user tool display state much the same as UI tools. See ui_ShowTool / ui_HideTool Support.

Note the following:

- ui_Show is only used in the context described in ui_ShowTool / ui_HideTool Support.
- By default, when ui_ShowTool is invoked on a user tool manged by ToolLauncher, if ui_Show is not defined in the tool nothing happens, and ui_HideTool terminates the tool process.
- When ui_Show is defined in a user tool, invoking ui_ShowTool or ui_HideTool on that tool will cause the value of ui_Show to be set and the body code to be executed:
  - ui_ShowTool sets ui_Show = TRUE
  - ui_HideTool sets ui_Show = FALSE

  Then, the body code of ui_Show determines what additional actions are taken. The example shown in ui_ShowTool / ui_HideTool Support causes the user tool to operate like UI's tools, as described in ui_ShowTool.

Usage

```
BOOL_VARIABLE( ui_Show, TRUE, "" ){}
```

ui_ShowTool is set and body code invoked as a side effect of using ui_ShowTool or ui_HideTool. See Description.

Example

See ui_ShowTool / ui_HideTool Support.

7.1.11.67 ui_ShowTool

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_ShowTool`.

This UI user variable can be used to invoke most of UI's Interactive Tools from user C-code.

`ui_ShowTool` is the complement of `ui_HideTool`. Details for both are documented here. `ui_HideTool` was first available in software release h1.1.23.

UI's tool display paradigm is somewhat different than other Windows applications. UI's tools will always be in one of the following states:

- Not started: the tool process is not running
- Started: the tool process is running. The tool is in one of the following states:
  - Tool is visible
  - Tool is visible but minimized; it is seen in the taskbar
  - Tool is hidden; i.e. not displayed and not seen in the taskbar

In normal use, the latter state is entered by first starting the tool and then clicking the cancel button (the  in the upper-right corner of the tool). `ui_HideTool` also puts a tool into this state. As indicated, hiding a UI tool this does not terminate the tool process. Instead, it hides the display; the tool does not appear in the task bar nor on the display. Starting the tool or using `ui_ShowTool` makes the tool visible again. This operation allows the tool to be hidden without losing any information.

Specific rules apply when using `ui_ShowTool` and `ui_HideTool`:

- Both are only usable from user-written C-code.
- Both are only usable after a test program has been loaded on all sites. See Example.

- Both are supported by the following UI tools, by specifying a value from the following table:

Table 7.1.11.67-1 ui_ShowTool / ui_HideTool Values

Value	Tool
"Bitmap"	BitmapTool
"Break"	Breakpoint Monitor
"DBM"	DBMTool
"ECR"	ECRTool
"Front"	FrontPanelTool
"LECTool"	LEC Tool
"LVM"	LVMTool
"Pattern"	PatternDebugTool
"Shmoo"	ShmooTool / SearchTool
"Summary"	Default Summary Tool
"Timing"	TimingTool
"V/I"	Voltage and Current Tool
"Variables"	User Variables Tool
"WafermapTool"	WafermapTool

- ui_ShowTool and ui_HideTool are both ONEOF_VARIABLEs. At any given time, a ONEOF_VARIABLE represents a single value, but has an associated list of legal values. Using ui_ShowTool and ui_HideTool the list of legal values are shown in the table above.
- To show or hide multiple tools requires executing ui_ShowTool and ui_HideTool multiple times, each specifying one tool.
- Beginning in software release h1.1.23, ui_ShowTool and ui_HideTool will also operate on User Tools which are started/managed by ToolLauncher. See ToolLauncher and ui_Show. In this situation, the value of the user tool is the name of the tool's executable file, exclusing *.exe*.

Usage

```
ONEOF_VARIABLE( ui_ShowTool, "", "" ){}
```

ui_ShowTool can only be set from user-written C-code using remote_set(). The variable is only valid when sent to UI (site = -1). Other rules apply, see Description:

```
remote_set ("ui_ShowTool", "value", -1);
```

where **value** identifies which tool to be started. Must be one of the values from the table above, specified as a quoted string.

Example

In the example below, the user code in the HOST_BEGIN_BLOCK waits for all sites to load a test program before using remote_set() to send ui_ShowTool to UI, to start FrontPanelTool. The ui_ProgLoaded code is required to support this operation:

```
// Use ui_ProgLoaded to send a signal to the HOST when a test
// program load has completed on all sites
VOID_VARIABLE( ui_ProgLoaded, "" ) {
   remote_signal( "ProgLoaded", site_num() );
}
HOST_BEGIN_BLOCK( myHBB ){
   //... other code as desired ...
   // Use remote_wait() here to wait until program load has
   // completed on all sites
   remote_wait( "ProgLoaded", INFINITE );
   remote_set( "ui_ShowTool", "Front", - 1 );// Send to Ui
   //... other code as desired ...
}
```

The following code hides FrontPanelTool:

```
remote_set("ui_HideTool", "Front", -1 );
```

7.1.11.68 ui_ShutDown

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See ui_ShutDown.

Description

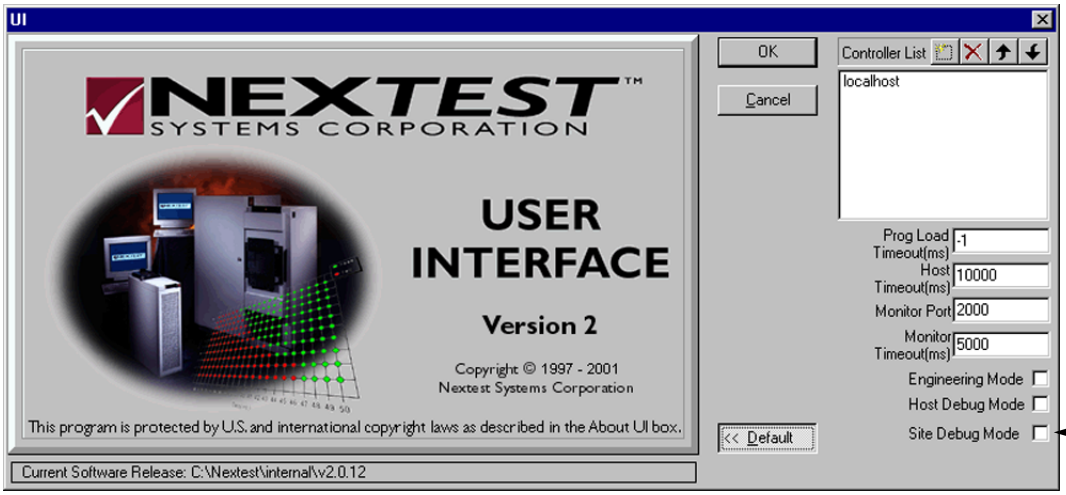
This UI user variable is identical to `ui_Exit`, and exists for backwards compatibility only.

7.1.11.69 ui_SiteDebug

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_SiteDebug`.

Description

Site Debug mode must be set (TRUE) to enable source code debugging using Developer Studio



This mode is normally set using the UI graphic interface (above) but can also be done using `ui_SiteDebug`.

`ui_SiteDebug` defines a bit-wise mask, where a logic-1 in a given bit position enables site debug on the corresponding site. The LSB is site-1. Setting the value to -1 enables site debug for all used sites.

This UI user variable can be used from a Windows command line, or from within a batch file (see `ui_BatchFile`).

It is possible to use `ui_SiteDebug` from C-code but this has limited value because it has no effect until the currently loaded test program is closed, and another test program is loaded, and only during the existing invocation of UI.

The value is NOT persistent after UI is terminated (via the UI.ini file).

A similar UI user variable is available to enable site debug: see ui_HostDebug.

Usage

From command line:

```
Long form:   /U:ui_SiteDebug=<value>
Short form:  /SD=<value>
Short form:  /SD
```

The latter short form assumes value = -1.

From batch file:

```
ui_SiteDebug=<value>
```

where **<value>** defines a bit-wise mask, where a logic-1 in a given bit position enables site debug on the corresponding site. The LSB is site-1. Setting the value to -1 enables site debug for all used sites.

From C-code, intended usage is via remote_set() and remote_get(). The variable is only valid when sent to UI (site = -1).

Example

These three examples perform identically when executed at a Windows command line:

```
C:\> ui /U:ui_SiteDebug=-1
C:\> ui /SD=-1
C:\> ui /SD
```

This example sets the mask from a batch file:

```
ui_SiteDebug=-1
```

This example sets the mask from user-written C-code:

```
remote_set("ui_SiteDebug", -1, -1 );
```

The following example will read back and output the state of this variable from user C-code:

```
CString val = remote_get("ui_SiteDebug", -1 );
output(" Get ui_SiteDebug => %s", val );
```

7.1.11.70 ui_SiteDone

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_SiteDone`.

Note: first available in software release h1.1.23.

Description

This Callback UI User Variable is invoked by UI each time execution of the Sequence & Binning Table completes on a given site.

If `ui_SiteDone` is defined in the test program and/or User Tools its body code will be executed any time UI signals `ui_SiteDone`. The value assigned to `ui_SiteDone` consists of the site which triggered the call-back plus a comma separated string of bin values from that site.

Also see `ui_TestDone`.

Usage

```
CSTRING_VARIABLE( ui_SiteDone, "", "" ) { [body code] }
```

where:

`ui_SiteDone` is a Callback UI User Variable.

`body code` is the C-code the user adds to the body of `ui_SiteDone` in their test program or User Tools.

Example

The following example executes in the Host process to output the information assigned to `ui_SiteDone` by the site which triggered the call-back. This will also execute in the Tool process if this `ui_SiteDone` code is included in a User Tools.

```
CSTRING_VARIABLE( ui_SiteDone, "", "" ) {
   output("  ui_SiteDone => %s", ui_SiteDone );
}
```

The following example is typical of the output generated using the previous example on a two site system:

```
ui_SiteDone => 2:bin1,bin2
ui_SiteDone => 1:bin2,bin1
```

7.1.11.71 ui_SiteLoaded

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_SiteLoaded`.

Description

This Callback UI User Variable is invoked by UI when a test program load has completed on *each* site.

If `ui_SiteLoaded` is defined in the test program and/or User Tools its body code will be executed once UI determines that test program loading has completed on a site. This can be used to synchronize code in the `HOST_BEGIN_BLOCK()` and/or `TOOL_BEGIN_BLOCK()` with site operation.

When invoked, the value of the variable is set to the site which reported the program loaded.

A similar UI user variable reports when *all* sites have reported the test program is loaded. See `ui_ProgLoaded`.

A related UI user variable reports when a site reports the test program is un-loaded. See `ui_SiteUnloaded`.

Usage

```
INT_VARIABLE( ui_SiteLoaded, 0, "" ) { [body code] }
```

where:

`ui_SiteLoaded` is a Callback UI User Variable.

body is the C-code the user adds to the body of `ui_SiteLoaded` in their test program or User Tools.

Example

This simple example outputs a message when `ui_SiteLoaded` is invoked by UI. This code will execute in the Host process if `ui_SiteLoaded` is defined in the test program. This code will execute in the Tool process if `ui_SiteLoaded` is defined in User Tools.

```
INT_VARIABLE( ui_SiteLoaded, 0, "" ) {
   output( "ui_SiteLoaded from Site-%d", ui_SiteLoaded );
}
```

7.1.11.72 ui_SiteMask

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_SiteMask`.

Description

This UI user variable can be used to determine which sites are currently enabled, and modify or set which sites are enabled.

As a practical matter, `ui_SiteMask` can be thought of as indicating which sites will execute the Sequence & Binning Table when **`File Start Testing`** is invoked. `ui_SiteMask` can be set during program loading, to match handler or prober capabilities, or modified later to disable defective sockets in parallel test operations.

`ui_SiteMask` is a bit-wise value where a logic-1 in a given bit position indicates that the site is enabled. The LSB is Site-1.

The value in `ui_SiteMask` always reflects all existing 128-pin sites, even when test sites are slaved together using `Sites-per-Controller` > 1. Thus when evaluating or setting the individual bits of `ui_SiteMask`, C-code should ignore bits based on the value of `Sites-per-Controller`. See Example 2:.

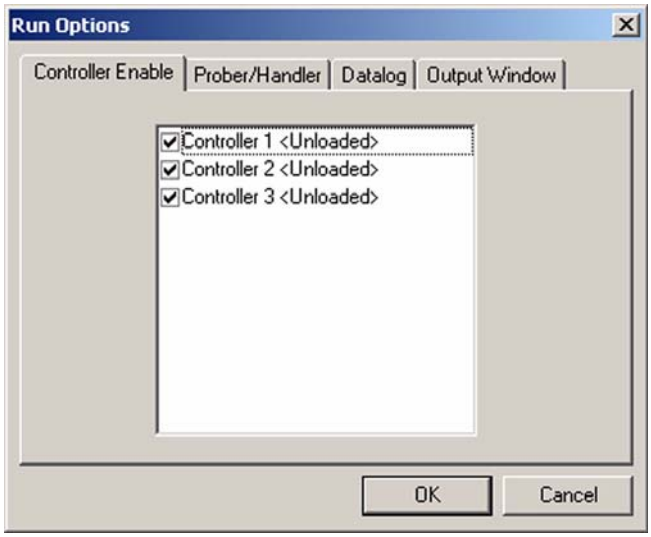
When getting the value of `ui_SiteMask` it is necessary to wait for program loading on all sites to complete. This suggests using `ui_ProgLoaded` or `ui_LoadedMask` to ensure results are valid. See Example 2:.

By default, a site will initially be enabled if all of the following are true:

- The associated Controller is active as set in UI's Controller List (displayed when UI is first started):.

UI's Controller List

- UI was able to communicate with that controller when it started.
- The Controller was not disabled using the *Controller Enable* tab of UI's **Tools Options**.



This UI user variable can be used from a Windows command line, from within a batch file (see `ui_BatchFile`), or from within user-written C-code.

The value is NOT persistent after UI is terminated (via the UI.ini file).

Usage

ui_SiteMask can be set from user-written C-code using remote_set() or remote_send(). remote_get() and remote_fetch() can be used to read back the current value. The variable is only valid when sent to UI (site = -1).

```
UINT64_VARIABLE( ui_SiteMask, 0x1, ""){}
remote_set ("ui_SiteMask", ui_SiteMask, -1);
remote_get ("ui_SiteMask", -1);
```

It is possible to set ui_SiteMask from a command line or batch file (see ui_BatchFile):

- From a command line:

  ```
  Long form: /U:ui_SiteMask=<mask>
  Short form:  /SM=<mask>
  ```

- From a batch file:

  ```
  ui_SiteMask=<mask>
  ```

where **<mask>** is the bit-mask as noted above.

Examples

Example 1:

These two examples perform identically when executed at a Windows command line:

```
C:\> ui /U:ui_SiteMask=1
C:\> ui /SM=1
```

This example sets the mask from a batch file:

```
ui_SiteMask=1
```

This example sets the mask from user-written C-code:

```
remote_set("ui_SiteMask", 1, -1 );
```

The following example will read back and output the state of this variable from user C-code:

```
UINT64_VARIABLE( ui_SiteMask, 0, ""){}
remote_get( "ui_SiteMask", -1 );
output(" Get ui_SiteMask => %I64d", ui_SiteMask );
```

Example 2:

This example uses ui_SiteMask in HOST_BEGIN_BLOCK() code. ui_ProgLoaded is used to signal the HOST_BEGIN_BLOCK() code when program loading has completed on all sites. The HOST_BEGIN_BLOCK() code execution then continues, to retrieve the current value of ui_SiteMask and, compensating for the value of sites_per_controller(), output a message indicating each enabled site:

```
// When UI reports all sites have completed loading the test
// program, send a signal to the HOST_BEGIN_BLOCK() to continue
// execution
VOID_VARIABLE( ui_ProgLoaded, "" ) {
   remote_signal( "AllSitesLoaded", 0 );
}
UINT64_VARIABLE( ui_SiteMask, 0, "" ) {}
HOST_BEGIN_BLOCK( example ) { // See HOST_BEGIN_BLOCK()
   // ... other code here ...
   // HOST_BEGIN_BLOCK() execution waits here for the signal
   // that all sites have completed loading the test program
   if ( remote_wait( "AllSitesLoaded", INFINITE ) == -1 )
        fatal( "ERROR: programs didn't load" );
   // Get the value of ui_SiteMask from UI
   remote_get( "ui_SiteMask", -1 );
   // Output a message for each enabled site. Ignore ganged sites
   // using sites_per_controller()
   for ( int bit = 0; bit < 60; bit += sites_per_controller() )
      if ((ui_SiteMask & ((UINT64) 1 << bit )) == 1)
         output( "Site %d is active", (bit + 1) ); // bit 0 = site-1
   // ... other code here ...
}
```

7.1.11.73 ui_SiteModeCommandLine

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See ui_SiteModeCommandLine.

Description

This UI user variable is used to specify command line arguments which will be executed in each active Site process.

This is targeted at setting the value of one or more user variables in the Site process(es), using command line or batch file methods (see `ui_BatchFile`). Similar capabilities exist for the Host using `ui_HostModeCommandLine` and the Tool using `ui_ToolModeCommandLine`.

A single string value is assigned to `ui_SiteModeCommandLine`, within which the `/s:` and `/S:` delimiters are used to:

- Delimit each user variable value pair
- Designate whether the specified user variable initialization is to occur before (`/s:`) the `CONFIGURATION()` block executes (i.e. before any program initialization has begun) or after (`/S:`) the `INITIALIZATION_HOOK()` executes (i.e. after all program initialization has completed).

Prior to being initialized by `ui_SiteModeCommandLine` the user variable value is determined by the value set in the program source code.

Only one `ui_SiteModeCommandLine` exists, the last one specified replaces any that were previously defined.

The body code of the user variable(s) being initialized executes in each enabled Site process. If multiple values are assigned to a user variable, which can be done using separate instances of `/S:` or `/s:`, the body code will execute once for each value. It is possible to set both `/s:` and `/S:` for the same user variable giving it one value before program initialization and a different value afterwards.

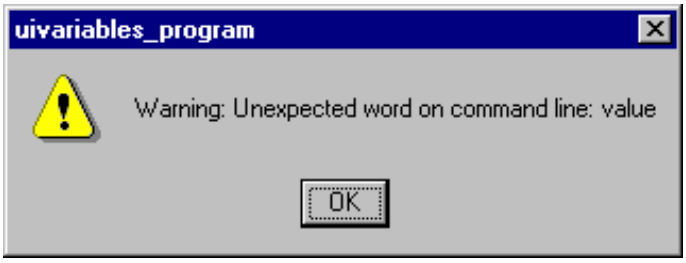
Normally, the command string only executes once, while loading the first test program after UI starts. Any other test programs loaded during the same UI invocation will not be affected. Using the Reload option the initialization will occur for each test program loaded using the current UI invocation.

The value is NOT persistent after UI is terminated (via the UI.ini file).

Two error types are reported, which occur during program loading:

- An invalid user variable name results in a warning message in each enabled Site output window. When this occurs, the test program continues to load/execute. In the example warning message below the invalid user variable name is: `uVar1`:
  Warning: Attempt to set undefined user-variable "uVar1" from site 1 ignored.
  Warning: This message will not be repeated for user-variable "uVar1".

- An invalid command string results in the following error popup (the program name will be different). After the OK button is clicked the test program continues to load/ execute:



Usage

```
CSTRING_VARIABLE( ui_SiteModeCommandLine, "", "" )
```

From command line:

```
Long form:   /U:ui_SiteModeCommandLine="command string"
Short form:  /SC="command string"
```

From batch file:

```
ui_SiteModeCommandLine="command string"
```

where **`command string`** utilizes two delimiters to separate user variable value pairs:

- `/s:`
- `/S:`

Using lowercase **`/s:`** causes the specified user variable to be initialized before the `CONFIGURATION()` block executes (i.e. before any program initialization has begun). Using the uppercase **`/S:`** causes the specified user variable to be initialized after the `INITIALIZATION_HOOK()` executes (i.e. after all program initialization has completed).

A timeout value can optionally be specified for a user variable by appending {time} to the statement. The value -1 sets `INFINITY`. For example, to set the timeout value for the user variable names uVAR1 to 1000mS:

```
C:\> ui /SC="/S:uVAR1=xxx{1000}
```

Example

These two examples perform identically when executed at a Windows command line. Two user variables are initialized, one before program initialization starts (uVAR1) and one after initialization had completed (uVAR2):

```
C:\> ui /U:ui_SiteModeCommandLine="/S:uVAR1=xxx /s:uVAR2=yyy"
C:\> ui /SC="/S:uVAR1=xxx /s:uVAR2=yyy"
```

Given the following user variable definitions exist in the test program, the output noted below will occur when either of the examples above is used:

```
CSTRING_VARIABLE( uVAR1, "?", "") {
   output("Site[%d] uVAR1 => %s ", site_num(), uVAR1 );
}
CSTRING_VARIABLE( uVAR2, "?", "") {
   output("Site[%d] uVAR2 => %s ", site_num(), uVAR2 );
}
```

Output messages in Site window on a single site system (PT):

```
Site[1] uVAR2 => yyy
The test program is loaded
Site[1] uVAR1 => xxx
```

7.1.11.74 ui_SiteUnloaded

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_SiteUnloaded`.

Description

This Callback UI User Variable is invoked by UI when a test program is unloaded on *each* site.

If `ui_SiteUnloaded` is defined in the test program and/or User Tools its body code will be executed once UI determines that a test program has been unloaded on each site. This can be used to synchronize code in the `HOST_END_BLOCK()` and/or User Tools code with site operation.

When invoked, the value of the variable is set to the site which reported the program unloaded.

A similar UI user variable reports when *all* sites have reported the test program is unloaded. See `ui_ProgUnloaded`.

A related UI user variable reports when each site reports the test program is loaded. See `ui_SiteLoaded`.

Usage

```
INT_VARIABLE( ui_SiteUnloaded, 0, "" ) { [body code] }
```

where:

ui_SiteUnloaded is a Callback UI User Variable.

body is the C-code the user adds to the body of ui_SiteUnloaded in their test program or User Tools.

Example

This simple example outputs a message when ui_SiteUnloaded is invoked by UI. This code will execute in the Host process if ui_SiteUnloaded is defined in the test program. This code will execute in the Tool process if ui_SiteUnloaded is defined in User Tools.

```
INT_VARIABLE( ui_SiteUnloaded, 0, "" ) {
   output( "ui_SiteUnloaded from Site-%d", ui_SiteUnloaded );
}
```

7.1.11.75 ui_StartTest

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See ui_StartTest.

Description

This UI user variable is used to send a Start Testing signal to UI.

ui_StartTest can be used from Host code or User Tools code.

ui_StartTest should only be used:

- After test program loading has completed on all enabled sites (see ui_ProgLoaded)
- After a previous start test has completed (see ui_TestDone).

Typically, after test program loading is complete, execution waits (indefinitely) for a Start Testing signal. The Start Testing signal can be invoked several ways:

- From the keyboard, by typing *control-T*
- Using UI's **File Start Testing** menu option

- Using the ui_StartTest user variable in user-written C-code.

Each time Start Testing is invoked, one iteration through the Sequence & Binning Table is executed.

When using automated IC handlers or wafer prober equipment it is common for a code loop in the HOST_BEGIN_BLOCK() to receive a start test signal from the equipment and forward it to UI using ui_StartTest (see Example). The code loop then waits to receive ui_TestDone from UI, indicating that all test sites enabled in the ui_SiteMask have completed testing. Execution then continues to, for example, retrieve binning information from the sites which is forwarded to the handler/prober equipment. The code loop then waits for the equipment to send another start test signal, etc.

Also see ui_StopTest.

Usage

Intended usage is via remote_send() from within user-written C-code, typically from User Dialogs or User Tools. The variable is only valid when sent to UI (site = -1).

Example

This is a simplified example showing the use ui_ProgLoaded, ui_StartTest, and ui_TestDone in a HOST_BEGIN_BLOCK() code loop as described above. Not shown is the equipment specific code which interfaces with a handler/prober to receive start test, and send binning information.

```
VOID_VARIABLE( ui_ProgLoaded, "" ) {
   // Send a signal to the corresponding remote_wait() in
   // the HOST_BEGIN_BLOCK() indicating that all sites have
completed
   // loading the test program
   remote_signal( "AllSitesLoaded", 0 );
}~
VOID_VARIABLE( ui_TestDone, "" ) {
   // Send a signal to the corresponding remote_wait() in
   // the HOST_BEGIN_BLOCK() indicating that all sites have
reported
   // Sequence & Binning Table execution has completed.
   remote_signal( "TestDone", 0 );
}
```

```
HOST_BEGIN_BLOCK( myHBB ) { // See HOST_BEGIN_BLOCK()
   // Wait here for all sites to report the test program is loaded
   remote_wait( "AllSitesLoaded", INFINITE );
   output( "All sites report test program is loaded" );
   while(1) { // Unload the test program to terminate the while(1)
      // Wait here for handler/prober start test signal
      // Send ui_StartTest to UI, which signals all enabled sites
      // to execute the Sequence & Binning Table.  Note that since
      // no body code is needed in ui_StartTest there is no need to
      // define it locally i.e. just send "the name"
      remote_send( "ui_StartTest", -1, TRUE );
      // Wait here for UI notification that testing has completed
      // on all sites
      remote_wait( "TestDone", INFINITE );
      // Gather up any desired binning information here
      // Send to handler/prober, etc.
   }
}
```

7.1.11.76 ui_StartTool

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_StartTool`.

Description

This UI user variable can be used to invoke User Tools from user C-code.

`ui_StartTool` can be used from a Windows command line, from within a batch file (see `ui_BatchFile`), or from user-written C-code.

`ui_StartTool` can also be used to start LEC Tool (*LECTool.exe*) and ScanTool (*scantool.exe*). These tools can't be started using `ui_ShowTool`.

Usage

```
CSTRING_VARIABLE( ui_StartTool, "", "" )
```

ui_StartTool can be set from user-written C-code using remote_set(). remote_get() can be used to read back the value of ui_StartTool. The variable is only valid when sent to UI (site = -1).

```
remote_set ("ui_StartTool", "path", -1);
```

Note that using remote_get() will return the value of ui_StartTool, regardless of whether a tool was actually started or not i.e. the return value is the same if the specified tool did not exist or the path was wrong, etc.

It is possible to set ui_StartTool from a command line or batch file (see ui_BatchFile). In both cases it is possible to start multiple User Tools using multiple instances of ui_StartTool.

From a command line:

```
Long form:   /U:ui_StartTool=<path>
Short form:  /TOOL=<path>
```

From a batch file:

```
ui_StartTool=<path>
```

where **<path>** is the desired path/file-name of the User Tools executable file.

Example

The following example will start the user tool at C:/myTool/Debug/myTool.exe:

```
CString path = "C:/myTool/Debug/myTool.exe;
remote_set( "ui_StartTool", path, -1 );
```

These two examples perform identically when executed at a Windows command line:

```
ui /U:ui_StartTool=C:/myTool/Debug/myTool.exe
ui /TOOL=C:/myTool/Debug/myTool.exe
```

This example starts the same tool from a batch file:

```
ui_StartTool=C:/myTool/Debug/myTool.exe
```

7.1.11.77 ui_StopTest

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See ui_StopTest.

Description

This UI user variable can be used to cause UI to send a Stop Testing signal to the enabled site controllers. In most cases, this will interrupt execution of the Sequence & Binning Table.

The Stop Testing signal can be invoked several ways:

- From the keyboard, by typing *control-S*
- Using UI's **File Stop Testing** menu option
- Using the `ui_StopTest` user variable in user-written C-code

Note: the ability to actually stop testing is limited. In general, testing will stop at the beginning of the next Nextest function to execute after the Stop Test signal sent to the site controllers by UI. If the APG pattern generator is active (i.e. executing a test pattern) it will be stopped and execution will halt. However, if a user-written code segment (or code loop) is executing which does not call any of the Nextest functions that code cannot be halted.

See `ui_StartTest`.

Usage

Intended usage is via `remote_send()` from within user-written C-code, typically from User Dialogs or User Tools. The variable is only valid when sent to UI (site = -1).

Example

```
remote_send( "ui_StopTest", -1, TRUE );
```

7.1.11.78 ui_TestDone

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_TestDone`.

Description

This Callback UI User Variable is invoked by UI each time execution of the Sequence & Binning Table completes on all enabled test sites.

If ui_TestDone is defined in the test program and/or User Tools its body code will be executed any time UI signals ui_TestDone. This is typically used to synchronize code in the Host C-code and/or user tool C-code with site operations.

Two versions (overloads) of ui_TestDone are available:

- The VOID version has no inherent value; i.e. it only executes the body code.
- The CSTRING version is assigned information sent from the last site which completed execution of the Sequence & Binning Table. The body code can use this information as desired. More below.

As indicated, the CSTRING version of ui_TestDone represents a CString variable. When the call-back is triggered the value assigned to ui_TestDone consists of the last site which completed execution of the Sequence & Binning Table plus a comma separated string of bin values from that site. ui_TestDone is useful on single-site systems to deliver binning information to Host code and/or User Tool code at the end of Sequence & Binning Table execution. However, when using multi-site systems, ui_SiteDone should be used instead, to allow binning information from all sites to be delivered to Host code and/or User Tool code.

Also see ui_TestStarted.

Usage

```
VOID_VARIABLE( ui_TestDone, "" ) { [body code] }
CSTRING_VARIABLE( ui_TestDone, "", "" ) { [body code] }
```

where:

ui_TestDone is a Callback UI User Variable.

body code is user code added to the body of ui_TestDone.

Example

The following example outputs a message when ui_TestDone is issued by UI. This code will execute in the Host process if ui_TestDone is defined in the test program. This code will execute in the Tool process if ui_TestDone is defined in User Tools:

```
VOID_VARIABLE( ui_TestDone, "", "" ) {
   output(" ui_TestDone received from UI");
}
```

7.1.11.79 ui_TestProgConfiguration

UI User Variables are also listed in UI User Variables, Alphabetical Listing. See `ui_TestProgConfiguration`.

Description

This UI user variable can be used to specify a `CONFIGURATION()` to be used as the test program loads.

`ui_TestProgConfiguration` is only useful when the test program contains multiple `CONFIGURATION()` blocks. Note that this has *no effect* on `HOST_CONFIGURATION()`, `SITE_CONFIGURATION()`, or `TOOL_CONFIGURATION()` selections.

This UI user variable is targeted for use from a Windows command line, or from within a batch file (see `ui_BatchFile`). While it is possible to set `ui_TestProgConfiguration` from user-written C-code it has no effect on the currently loaded test program, but will affect the next program loaded within the same instance of UI.

The value is NOT persistent after UI is terminated (via the UI.ini file). The value is cleared when a test program is loaded.

Usage

```
CSTRING_VARIABLE( ui_TestProgConfiguration, "", "" )
```

`ui_TestProgConfiguration` can be set from a command line or batch file (see `ui_BatchFile`).

From a command line:

```
Long form:   /U:ui_TestProgConfiguration=<name>
Short form:  /TPC=<name>
```

From a batch file:

```
ui_TestProgConfiguration=<name>"
```

where **<name>** is the desired name of the a `CONFIGURATION()` block in the test program to be loaded next. The name must be exact, and is case sensitive. An invalid name is ignored, with no warnings.

Example

The examples below assume the next test program loaded will contain the following:

```
CONFIGURATION( Config1 ) {
   // ... some code here
}
CONFIGURATION( Config2 ) {
   // ... some code here
}
```

Example 1:

These two examples perform identically when executed at a Windows command line.

```
ui /TPC=Config2
ui /U:ui_TestProgConfiguration=Config2
```

Example 2:

The following is an example of a batch file which will start UI, enable Engineering Mode, load the test program *D:/MinTestProg/Debug/MinTestProg*, and select the Config1 `CONFIGURATION()` block:

```
ui_NoLogo=1
ui_EngineeringMode=1
ui_TestProgConfiguration=Config1
ui_TestProgName=D:/MinTestProg/Debug/MinTestProg
```

If these statements are in the file *`C:\test6_batch.txt`* the batch file can be executed from the command line using

```
C:\ui /BATCH=C:\test6_batch.txt
```

Example 3:

This example sets `ui_TestProgConfiguration` from C-code which can execute in Host, Site or User Tools code. As noted above, this has limited value:

```
CString name = "Config2";
remote_set( "ui_TestProgConfiguration", name, -1 );
```

This example gets the current value of `ui_TestProgConfiguration` from C-code which can execute in Host, Site or User Tools code. Note, however, that loading a test program clears the value in `ui_TestProgConfiguration`, thus this example is only useful before the test program is loaded i.e. from user tool code when the tool is started before the test program:

```
CString val = remote_get("ui_TestProgConfiguration", -1 );
```

```
output(" Get ui_TestProgConfiguration => %s", val )
```

7.1.11.80 ui_TestProgDirPath

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_TestProgDirPath`.

Description

This UI user variable can be used to specify the initial location displayed in UI's **File Open Test Program** browser.

`ui_TestProgDirPath` is targeted for use from a Windows command line, from within a batch file (see `ui_BatchFile`), or from user-written Host, Site or User Tools C-code.

An invalid path is ignored.

The value IS persistent after UI is terminated (via the UI.ini file).

Usage

```
CSTRING_VARIABLE( ui_TestProgDirPath, "", "" )
```

`ui_TestProgDirPath` can be set from user-written C-code using `remote_set()`. `remote_get()` can be used to read back the current value. The variable is only valid when sent to UI (site = -1).

```
remote_set ("ui_TestProgDirPath", "path", -1);
CString path = remote_get ("ui_TestProgDirPath", -1);
```

`ui_TestProgDirPath` can be set from a command line or batch file (see `ui_BatchFile`).

From a command line:

```
Long form:   /U:ui_TestProgDirPath=<path>
Short form:  /TD=<path>
```

From a batch file:

```
ui_TestProgDirPath=<path>"
```

where **<path>** is the desired path to the test program to be loaded next.

Example

Example 1:

These two examples perform identically when executed at a Windows command line.

```
ui /TD=D:/MinTestProg/Debug
ui /U:ui_TestProgDirPath=D:/MinTestProg/Debug
```

Example 2:

The following is an example of a batch file which will start UI, enable Engineering Mode, set a path to the next program loaded = *D:/MinTestProg/Debug*, and load the test program at that location:

```
ui_NoLogo=1
ui_EngineeringMode=1
ui_TestProgDirPath=D:/MinTestProg/Debug
ui_TestProgName=MinTestProg
```

If these statements are in the file *C:\test7_batch.txt* the batch file can be executed from the command line using

```
C:\ui /BATCH=C:\test7_batch.txt
```

Example 3:

This example sets `ui_TestProgDirPath` from C-code which can execute in Host, Site or User Tools code:

```
CString path = "D:/MinTestProg/Debug";
remote_set( "ui_TestProgDirPath", path, -1 );
```

This example gets the current value of `ui_TestProgDirPath` from C-code which can execute in Host, Site or User Tools code:

```
CString path = remote_get("ui_TestProgDirPath", -1 );
output(" Get ui_TestProgDirPath => %s", path )
```

7.1.11.81 ui_TestProgName

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_TestProgName`.

Description

This UI user variable can be used to specify the name of a test program to load.

When using `ui_TestProgName`, if a full path to the test program executable is not specified, the location of the test program will be resolved using the default environmental path value, unless `ui_TestProgDirPath` is used.

`ui_TestProgName` is targeted for use from a Windows command line, or from within a batch file (see `ui_BatchFile`).

Use `remote_get()` of `ui_TestProgName` to get the complete path/program-name to the currently loaded test program from user C-code. No valid application exists for setting `ui_TestProgName` from user-written C-code.

The value is NOT persistent after UI is terminated (via the UI.ini file).

Usage

```
CSTRING_VARIABLE( ui_TestProgName, "", "" )
```

`ui_TestProgName` can be set from a command line or batch file (see `ui_BatchFile`).

From a command line:

```
Long form:   /U:ui_TestProgName=<name>
Short form:  /TP=<name>
```

From a batch file:

```
ui_TestProgName=<name>"
```

where **<name>** is the name of the desired test program executable file.

Example

Example 1:

These two examples perform identically when executed at a Windows command line.

```
ui /TP=D:/MinTestProg/Debug/MinTestProg.exe
ui /U:ui_TestProgName=D:/MinTestProg/Debug/MinTestProg.exe
```

Example 2:

The following is an example of a batch file which will start UI, enable Engineering Mode, and load the test program = `D:/MinTestProg/Debug/MinTestProg.exe`:

```
ui_NoLogo=1
ui_EngineeringMode=1
ui_TestProgName=D:/MinTestProg/Debug/MinTestProg.exe
```

If these statements are in the file *C:\test8_batch.txt* the batch file can be executed from the command line using

```
C:\ui /BATCH=C:\test8_batch.txt
```

Example 3:

This example gets the current value of the currently loaded test program from C-code, which can execute in Host, Site or User Tools code:

```
CString name = remote_get("ui_TestProgName", -1 );
output(" Get ui_TestProgName => %s", name );
```

7.1.11.82 ui_TestStarted

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_TestStarted`.

Description

This Callback UI User Variable is invoked by UI each time it sends the Start Test signal to the test sites.

If `ui_TestStarted` is defined in the test program and/or User Tools its body code will be executed any time UI signals `ui_TestStarted`. This is typically used to synchronize code in the Host C-code and/or user tool C-code with site operations.

Note that the type of this variable was initially `VOID_VARIABLE` but later changed to `UINT64_VARIABLE`. After this change, the value of `ui_TestStarted` is set to a bit-wise mask of the sites which are receiving the start-test signal.

Also see `ui_TestDone`, `ui_SiteDone`.

Usage

```
UINT64_VARIABLE( ui_TestStarted, "" ) { [body code] }
```

where:

`ui_TestStarted` is a Callback UI User Variable.

body code is the C-code the user adds to the body of `ui_TestStarted` in their test program or User Tools.

Example

This simple example outputs a message when `ui_TestStarted` is issued by UI. This code will execute in the Host process if `ui_TestStarted` is defined in the test program. This code will execute in the Tool process if `ui_TestStarted` is defined in User Tools.

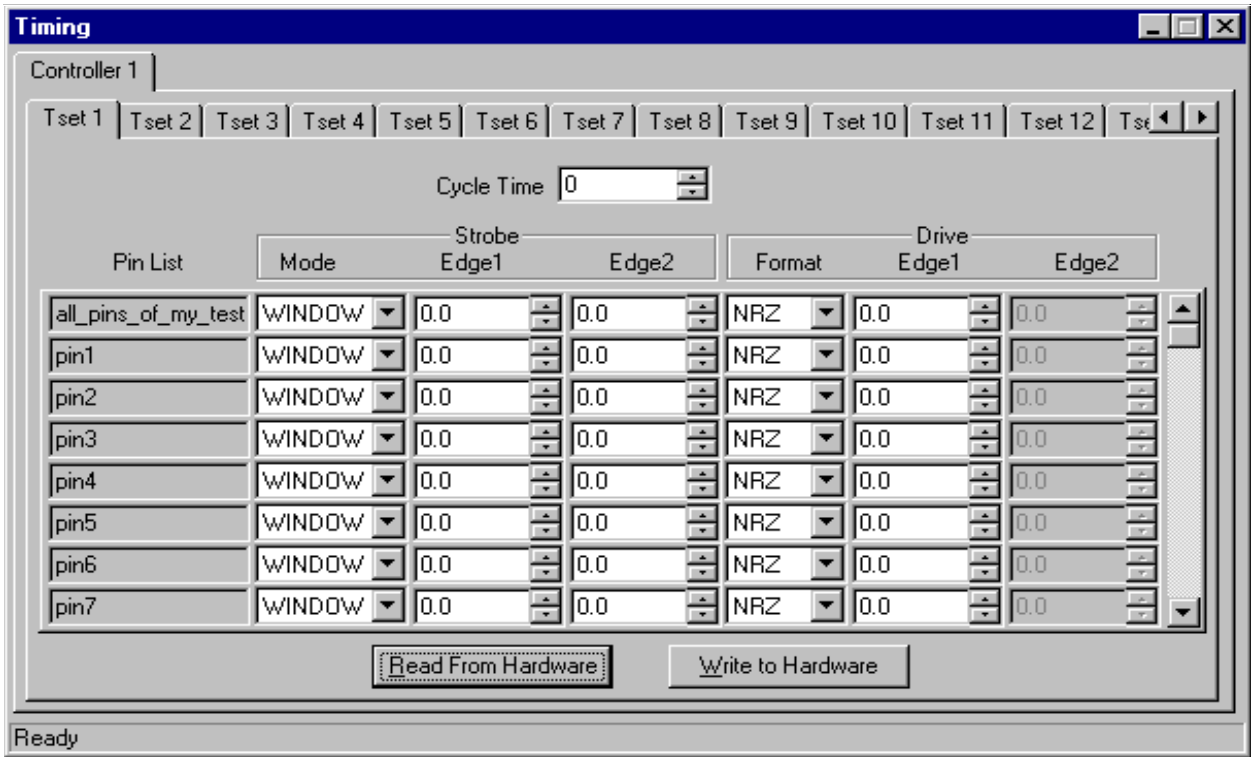
```
UINT64_VARIABLE( ui_TestStarted, "" ) {
   output( "ui_TestStarted received");
   output( " Site mask = 0x%I64x", ui_TestStarted );
}
```

7.1.11.83 ui_TimingToolPinLists

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_TimingToolPinLists`.

Description

By default, TimingTool displays all pin lists defined in the test program, regardless whether the pin lists are used to define timing or not. In the example below, note the pin list named `all_pins_of_my_test_program` is displayed, even though it isn't useful in TimingTool.



User C-code can be used to both specify which pin lists are displayed in TimingTool, and to enable runtime modification of this list.

`ui_TimingToolPinLists` can only be used from user-written C-code.

Note the following:

- Pay close attention to the spelling of the pin list names, including case.
- Pin lists which are not valid for the test program in use are ignored, no warning messages are generated.
- If all pin lists specified are invalid TimingTool won't display any timing values.
- Setting the value to "" (null string) will restore default operation i.e. display all pin lists in the test program.
- If TimingTool is open when `ui_TimingToolPinLists` is modified, it must be terminated and restarted to see the changes.

- The value is NOT persistent after the current test program is unloaded or when UI is terminated (via the UI.ini file).

Usage

```
CSTRING_VARIABLE( ui_TimingToolPinLists, "", "" )
```

`ui_TimingToolPinLists` can be set from user-written C-code using `remote_set()`. `remote_get()` can be used to read back the current value. The variable is only valid when sent to UI (site = -1).

Multiple pin lists are specified as a comma separated `CString`:

```
remote_set ("ui_TimingToolPinLists", "pl_1, pl_2, pl_n", -1);
```

Examples

Example 1:

If interactive changes of the list of pin lists are not needed use `remote_set()` in the Host Begin Block with the desired pin lists coded directly:

```
HOST_BEGIN_BLOCK( your_host_begin_block_name ){
... other code ...
   remote_set(  "ui_TimingToolPinLists",
                "pl_1, pl2, pl_n",
                -1 );
... other code ...
}
```

Example 2:

This code will retrieve the current value of `ui_TimingToolPinLists` and print it as a comma separated string:

```
CString myPL = remote_get ("ui_TimingToolPinLists", -1 );
output(" myPL => %s", myPL );
```

Example 3:

It is also possible to use a `CSTRING_VARIABLE` to specify the list of pin lists. When this is done, the list can be modified interactively using UI's User Variables Tool. In this example, `TimingPinLists` is set at compile-time to an explicit list of pin list names, and invoked in

the HOST_BEGIN_BLOCK(). With no interactive changes, TimingTool will start and display these pin lists:

```
   CSTRING_VARIABLE( TimingPinLists,
                     "pl_1, pl2, pl_n",
                     "Timing Pin Lists" ) {
      remote_set( "ui_TimingToolPinLists", TimingPinLists, -1 );
   }
   HOST_BEGIN_BLOCK( your_host_begin_block_name ) {
   ... other code ...
   invoke (TimingPinLists); // Execute uVar body code
   ... other code ...
}
```

Example 4:

In the following example, the CSTRING_VARIABLE sets ui_TimingToolPinLists to contain every pin list in the test program. The user variable can still be interactively modified using UI's User Variables Tool:

```
   CSTRING_VARIABLE( TimingPinLists,
                     resource_all_names( S_PinList ),
                     "Timing Pin Lists" ) {
      // set it in Ui
      remote_set( "ui_TimingToolPinLists", TimingPinLists, -1 );
   }
```

Note that Initially TimingPinLists contains a comma separated list of every pin list in the test program. This happens during program initialization because the 2nd argument to the CSTRING_VARIABLE definition executes resource_all_names(), which returns a list of all pin lists in the program. TimingPinLists is then set to that list.

7.1.11.84 ui_ToolLoaded

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See ui_ToolLoaded.

Description

This Callback UI User Variable is invoked by UI any time a User Tools has started and has completely initialized.

If `ui_ToolLoaded` is defined in the test program or in User Tools code, its body code will be executed each time UI invokes this user variable. Note that this notification is limited to Host process and User Tools only i.e. it is not invoked in any site process. The value UI assigns to `ui_ToolLoaded` will be the site number of the tool just started.

Note: `ui_ToolLoaded` is not sent by UI until after the tool is completely initialized i.e. after the `TOOL_BEGIN_BLOCK` and `INITIALIZATION_HOOK` executions have completed.

Also see `ui_ToolUnloaded`.

Usage

```
INT_VARIABLE( ui_ToolLoaded, 0, "" ) { [body code] }
```

where:

`ui_ToolLoaded` is a Callback UI User Variable.

body is the C-code the user adds to the body of `ui_ToolLoaded` in their User Tools.

Example

This simple example outputs a message when `ui_ToolLoaded` is invoked by UI. This code will execute in the Host process and in each running user tool process in which `ui_ToolLoaded` is defined:

```
INT_VARIABLE( ui_ToolLoaded, 0, "" ) {
   output( "ui_ToolLoaded for Site Number => %d", ui_ToolLoaded);
}
```

7.1.11.85 ui_ToolModeCommandLine

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_ToolModeCommandLine`.

Description

This UI user variable is used to specify command line arguments when UI is started, but these arguments which effect the first User Tools started by UI.

This is targeted at using command line or batch file methods (see ui_BatchFile) to start UI, where the command line or batch file needs to set the value of one or more user variables used by the first User Tools started from UI.

Several key differences exist between `ui_ToolModeCommandLine` and the rather similar capabilities provided for Site options (using `ui_SiteModeCommandLine`) and Host options (using ui_HostModeCommandLine):

- `ui_ToolModeCommandLine` sets options when starting UI, but which are only effective if and when the first User Tool is subsequently started from UI (using the **`Tools > Open Tool...`** menu option). A User Tool started using any other method does not receive the command line parameters.
- Once the first User Tool is started, the command line arguments assigned to `ui_ToolModeCommandLine` are cleared.
- `ui_ToolModeCommandLine` does not have Reload support.

A single string value is assigned to `ui_ToolModeCommandLine`, within which the `/t:` and `/T:` delimiters are used to:

- Delimit each user variable value pair
- Designate whether the specified user variable initialization is to occur before (`/t:`) the `CONFIGURATION()` block executes (i.e. before any tool initialization has begun) or after (`/T:`) the `INITIALIZATION_HOOK()` executes (i.e. after all tool initialization has completed).

Prior to being initialized by `ui_ToolModeCommandLine` the user variable value is determined by the value set in the program source code.

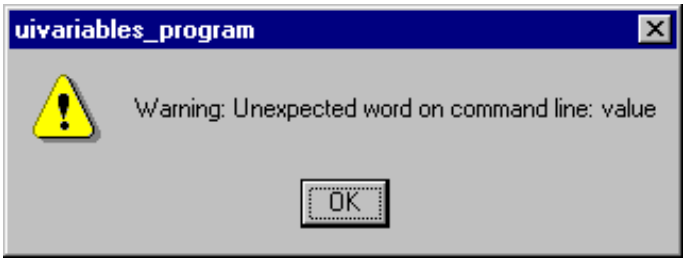
Only one `ui_ToolModeCommandLine` *value* exists, the last one specified replaces any that were previously defined.

The body code of each user variable(s) being initialized executes in the Tool process. If multiple values are assigned to a user variable, which can be done using separate instances of `/T:` or `/t:`, the body code will execute once for each value. It is possible to set both `/t:` and `/T:` for the same user variable giving it one value before program initialization and a different value afterwards.

The value is NOT persistent after UI is terminated (via the *UI.ini* file).

Two error types are reported, which occur during Tool loading:

- An invalid user variable name results in a warning message in the Host output window, when the tool is loading. When this occurs, the tool continues to load/ execute.
- An invalid command string results in the following error popup (the program name will be different). After the OK button is clicked UI will continue to load/execute:



Usage

```
CSTRING_VARIABLE( ui_ToolModeCommandLine, "", "" )
```

From command line:

```
Long form:   /U:ui_ToolModeCommandLine="command string"
Short form:  /TC="command string"
```

From batch file:

```
ui_ToolModeCommandLine="command string"
```

where command string utilizes two delimiters to separate user variable value pairs:

- /t:
- /T:

Using lowercase `/t:` causes the specified user variable to be initialized before the `CONFIGURATION()` block executes (i.e. before any program initialization has begun). Using the uppercase `/T:` causes the specified user variable to be initialized after the `INITIALIZATION_HOOK()` executes (i.e. after all program initialization has completed).

A time out value can optionally be specified for a user variable by appending {time} to the statement. The value -1 sets INFINITY. For example, to set the time out value for the user variable names uVAR1 to 1000mS:

```
C:\> ui /TC="/T:uVAR1=xxx{1000}
```

Example

These two examples perform identically when executed at a Windows command line. Values for two user variables are specified, one to be applied before tool initialization starts (uVAR1) and one after initialization has completed (uVAR2):

```
C:\> ui /U:ui_ToolModeCommandLine="/T:uVAR1=xxx /t:uVAR2=yyy"
C:\> ui /TC="/T:uVAR1=xxx /t:uVAR2=yyy"
```

Given the following user variable definitions exist in the tool, the output noted below will occur when either of the examples above is used:

```
CSTRING_VARIABLE( uVAR1, "?", "") {output("uVAR1 => %s", uVAR1);}
CSTRING_VARIABLE( uVAR2, "?", "") {output("uVAR2 => %s", uVAR2);}
```

Output messages in Host window (the Tools site number may vary):

```
[Tool 2511]: uVAR2 => yyy
[Tool 2511]: uVAR1 => xxx
```

7.1.11.86 ui_ToolUnloaded

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_ToolUnloaded`.

Description

This Callback UI User Variable is invoked by UI any time a User Tools is terminated.

If `ui_ToolUnloaded` is defined in the test program or User Tools code, its body code will be executed each time UI invokes this user variable. Note that this notification is limited to Host process and User Tools only i.e. it is not invoked in any Site processes. The value UI assigns to `ui_ToolUnloaded` will be the site number of the tool just unloaded.

Note: `ui_ToolUnloaded` is not sent by UI until after the tool is completely unloaded i.e. after the `TOOL_END_BLOCK` execution has completed.

Also see `ui_ToolLoaded`.

Usage

```
INT_VARIABLE( ui_ToolUnloaded, 0, "" ) { [body ] }
```

where:

ui_ToolUnloaded is a Callback UI User Variable.

body is the C-code the user adds to the body of ui_ToolUnloaded in their User Tools.

Example

This simple example outputs a message when ui_ToolUnloaded is invoked by UI. This code will execute in each running user tool process in which ui_ToolUnloaded is defined:

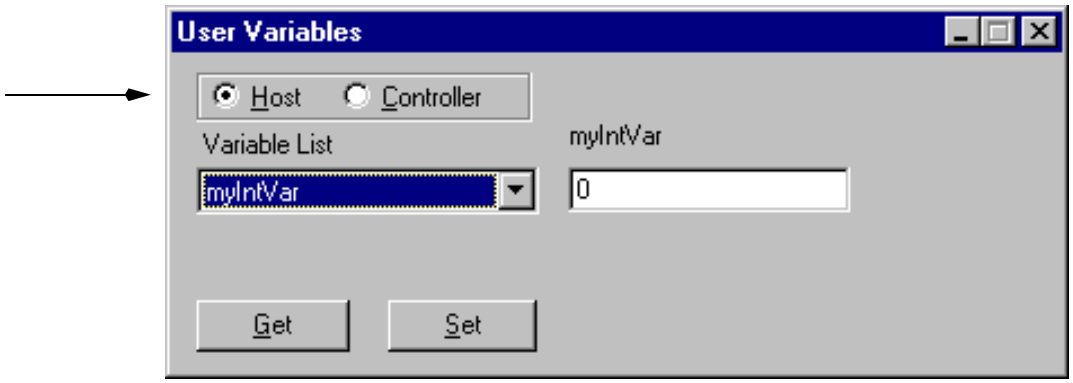
```
VOID_VARIABLE( ui_ToolUnloaded, "" ) {
 output("ui_ToolUnloaded for Site Number => %d", ui_ToolUnloaded);
}
```

7.1.11.87 ui_UserVarSiteMode

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See ui_UserVarSiteMode.

Description

This UI User Variable can be used to select the initial state of the radio button in UI's User Variables Tool, to be either *Host* or *Controller*:



ui_UserVarSiteMode can be used from a Windows command line, from within a batch file (see ui_BatchFile), or from within user-written C-code.

The value IS persistent after UI is terminated (via the UI.ini file).

Usage

ui_UserVarSiteMode can be set from user-written C-code using remote_set(). remote_get() can be used to read back the current value. The variable is only valid when sent to UI (site = -1).

```
remote_set ("ui_UserVarSiteMode", value, -1);
CString val = remote_get("ui_UserVarSiteMode", -1);
```

where **<value>** is 0 to enable the Host radio button, or 1 which enables the Controller radio button.

From command line:

```
Long form:   /U:ui_UserVarSiteMode=<value>
Short form:  /UVS=<value>
Short/short form: /UVS
```

The latter short form assumes value = 1 i.e. Controller is selected.

From batch file:

```
ui_UserVarSiteMode=<value>
```

where **<value>** is 0 to enable the Host radio button, or 1 which enables the Controller radio button.

Example

These two examples perform identically when executed at a Windows command line. Both cause the Controller radio button to be enabled:

```
C:\> ui /U:ui_UserVarSiteMode=1
C:\> ui /UVS=1
```

This example sets ui_UserVarSiteMode from C-code which can execute in Host, Site or User Tools code:

```
remote_set( "ui_UserVarSiteMode", 1, -1 );
```

This example gets the current value of ui_UserVarSiteMode from C-code which can execute in Host, Site or User Tools code:

```
CString val = remote_get("ui_UserVarSiteMode", -1 );
output(" Get ui_UserVarSiteMode => %s", val );
```

7.1.11.88 ui_UserVariableTimeout

All *UI User Variables* are also listed in UI User Variables, Alphabetical Listing. See `ui_UserVariableTimeout`.

Description

This UI user variable can be used to modify the default timeout value applied when executing User Variables body code.

The default value = 5000 i.e. 5 seconds

`ui_UserVariableTimeout` can be used from a Windows command line, from within a batch file (see `ui_BatchFile`), or from within user-written C-code. See Usage below.

The value is NOT persistent after UI is terminated (via the UI.ini file).

Usage

`ui_UserVariableTimeout` can be set from user-written C-code using `remote_set()`. `remote_get()` can be used to read back the current value. The variable is only valid when sent to UI (site = -1).

```
remote_set ("ui_UserVariableTimeout", value, -1);
CString val = remote_get("ui_UserVariableTimeout", -1);
```

where **<value>** is the desired timeout value in milliseconds (mS).

Example

This example sets `ui_UserVariableTimeout` from C-code which can execute in Host, Site or User Tools code:

```
remote_set( "ui_UserVariableTimeout", 5000, -1 );
```

This example gets the current value of `ui_UserVariableTimeout` from C-code which can execute in Host, Site or User Tools code:

```
CString val = remote_get("ui_UserVariableTimeout", -1 );
output(" Get ui_UserVariableTimeout => %s", val );
```

7.2 Host / Site / Tool Communication

Note: see Overview in Binning for an explanation of the different processes noted in the following sections.

The Host process, each Site process, and all Tool processes execute independently. However, there may be occasions when some synchronization or exchange of information is needed between these processes.

Execution synchronization between process can use the `remote_signal()`, `remote_wait()` functions, to cause one process to wait for a signal from another process before execution continues.

Information exchange between processes is done via User Variables:

- The `remote_send()` and `remote_fetch()` functions can be used to synchronize the values of User Variables between processes. Optionally, the body code of the user variable can be executed in the *receiving* process. These functions work with single user variable or with a snapshot (see `SNAPSHOT()`).
- The `remote_set()`function can be used to change the value of User Variables in a remote process. The `remote_get()`function can be used to retrieve the value of User Variables from another process. Optionally, the body code of the user variable can be executed in the *receiving* process.
- The contents of arbitrary user-defined data structures can be sent between processes using the Transferring User-defined Data Structures (Serialization) mechanism.

7.2.1 remote_signal(), remote_wait()

Note: see Overview in Binning for an explanation of the different processes noted below.

Description

The Host process, each Site process, and all Tool processes execute independently. However, there may be occasions when some synchronization is needed between processes.

For example, when using an IC handler or prober, the handler/prober code executes in the Host process, and needs to wait for the test program to completely load on all Sites before the first start test signal is sent. This means that the Host process needs to wait for all Site processes to signal they are ready to execute tests. The `remote_signal()` and `remote_wait()` functions can be used to implement this type of synchronization.

The `remote_signal()` function is used to send a synchronization signal from one process to another. Arguments to this function determine the name of the signal, and which process (site) to send it to.

The `remote_wait()` function is used check if a specific synchronization signal has been received. `remote_wait()` can also be used to check if any of several signals have been received. In either case, an optional timeout can be specified to control how long `remote_wait()` actually waits for a signal before continuing.

Usage

```
void remote_signal( LPCTSTR name, int target );
int remote_wait(  LPCTSTR name,
                  DWORD timeout DEFAULT_VALUE( INFINITE ) );
int remote_wait(  CStringArray &names,
                  DWORD timeout DEFAULT_VALUE( INFINITE ),
                  int *index DEFAULT_VALUE( 0 ) );
```

where:

name is the user-defined name of the synchronization signal. To obtain the desired operation, the **name** must be identical in code used in both the `remote_signal()` process and the `remote_wait()` process. A misspelling of names means the two computers will not communicate on that synchronization signal.

names is an array of user-defined signal names, which can be used when multiple signals are to be monitored. The `remote_wait()` function using **names** will return if any of these signals are received. The **index** parameter will indicate which signal was received (see below).

target is one of the standard site numbers identifying the process that will receive the remote_signal(), as follows:

- 0 = Host process
- 1 through 60 represent Site1 through Site60 processes
- A value > 1024 is a User Tools (see site_num()).

timeout specifies how long the function will wait before proceeding, in milliseconds. If not specified, the **timeout** value defaults to INFINITE, which can also be used explicitly to wait indefinitely. If the proper signal is not received before the **timeout** elapses remote_wait() returns -1. If the proper signal is received before the **timeout** elapses remote_wait() returns the site number of the process that sent the synchronization signal (**name**).

index is used when multiple signals are monitored, to identify which signal was received by remote_wait(). When a signal is received by remote_wait() **index** returns the index in the **names** array of the signal received.

Examples

Example 1:

This simple example has the Host process waiting up to 10 seconds to receive a signal named “Site Ready”. This example would apply only to a one site system, as will be discussed below. The Site process (SITE_BEGIN_BLOCK()) sends the signal when it is ready for Host execution (HOST_BEGIN_BLOCK()) to continue execution.

```
HOST_BEGIN_BLOCK( HBB1 ) {
   // Other code here to set up Host
   if ( int site = remote_wait( "Site Ready", 10000 ) == -1 )
      warning( "Site 1 did not respond.");
   else
      output( "Site Ready signal received from site => %d", site );
   // Other code here to complete Host setup
}
SITE_BEGIN_BLOCK( SBB1 ) {
   // Other code here to set up Site
   remote_signal( "Site Ready", 0 ); // 0 = send to Host process
}
```

In the Host Begin Block, the remote_wait() function waits for 10 seconds (10000 milliseconds) to receive the signal. If the signal is not received before the 10 sec. timeout

elapses, `remote_wait()` returns -1 and the warning message is printed. If the signal is received the Host outputs a message indicating which site sent the signal.

Example 2:

As mentioned above, the previous example would apply only to a one site system. In a multi-site system, all sites execute the identical `SITE_BEGIN_BLOCK()` code, thus the first site which sent the signal would cause the `HOST_BEGIN_BLOCK()` code to proceed, possibly before other sites are ready. Therefore, using a multiple site system the `remote_wait()` code needs to confirm that all sites are ready. The code below does this.

```
// UI sends a signal to the HOST_BEGIN_BLOCK() when all sites
// have reported the test program load has completed.
VOID_VARIABLE( ui_ProgLoaded, "") {
   remote_signal ( "Prog Loaded", 0 );
}
HOST_BEGIN_BLOCK( HBB1 ) {
   // Other host code before waiting for sites to complete
   remote_wait( "Prog Loaded" ); // Uses default INFINITE timeout
}
```

7.2.2 remote_send()

Note: see Overview in Program Execution Control for an explanation of the different processes noted below.

Descriptionc

Note: the term local process is used below to indicate the process which executes `remote_send()`. The term remote process indicates the process which is being modified.

The `remote_send()` function is used to synchronize the values of User Variables between the local process and a remote process. This can be used to synchronize user variable values between Host process, and/or Site(s) process, and/or User Tools processes.

`remote_send()` causes the current value of a user variable in the local process to be assigned to the same user variable in the remote process. Optionally, the body code of the

user variable can be executed in the remote process, and the local process has the option of waiting for the transaction to complete, including execution of the body code, before the local execution continues.

Using `remote_send()` requires that the referenced user variable(s) be *defined* in both local and remote processes. In contrast, `remote_set()` can be used to directly transmit a new value to a user variable in a remote process, without the need to declare that user variable in the local process. Or, if the user variable exists in both processes, `remote_set()` can modify the value in a remote process without changing the value in the local process.

`remote_send()` is used in two contexts:

- Referencing one user variable
- Referencing a snapshot (see `SNAPSHOT()`)

When `remote_send()` references a snapshot all of the user variables defined in the `SNAPSHOT()` are synchronized. The current values of the associated user variables in the local process are assigned to the corresponding user variables in the remote process.

Using the timeout parameter, the local process has the option of waiting for the remote process to receive the data and to complete the execution of the user variable body code before the local process execution continues. When a snapshot is referenced, the sequence of user variable body code execution follows the order the user variables are listed in the `SNAPSHOT()`, and body code execution of each user variable will complete before the body code of the next user variable begins.

Usage

```
BOOL remote_send( VariableProxy v,
                  int site,
                  BOOL invoke,
                  DWORD timeout DEFAULT_VALUE( 0 ) );
BOOL remote_send( Snapshot *snapshot,
                  int site,
                  BOOL invoke,
                  DWORD timeout DEFAULT_VALUE( 0 ) );
```

where:

v represents one user variable to be synchronized. A `VariableProxy` type refers to User Variables and can be:

- A user variable name

- A quoted user variable name (string)
- A pointer to a user variable

snapshot represents a snapshot; i.e. a set of user variables. All user variables in the snapshot will be synchronized. See SNAPSHOT().

site is one of the standard site numbers identifying the remote process i.e. the site in which the user variable or snapshot is to be modified, as follows:

- -1 = UI - User Interface process, used with UI User Variables only
- 0 = Host process
- 1 through 60 represent Site1 through Site60 processes
- A value > 1024 is a User Tools (see site_num())

invoke determines whether the body code of the user variable is executed in the remote process. TRUE will cause the body code to execute, FALSE will not. When a snapshot is referenced the body code of all user variables in the SNAPSHOT() will execute (see above).

timeout specifies how long the function will wait before proceeding, in milliseconds. This includes any time spent executing the user variable body code in the remote process. If not specified, the **timeout** value defaults to zero. The value INFINITE can be specified to wait indefinitely. If the transaction completes before time expires, remote_send() returns TRUE, otherwise it returns FALSE.

Example

This example can be used to allow only one site at a time to perform some action. This example is based on the fact that all access to a particular user variable's body code is serialized by the Host process. All of the action starts with TEST_BLOCK(abc) below.

```
EXTERN_VOID_VARIABLE( need_x ); // Forward declaration
EXTERN_VOID_VARIABLE( got_x ); // Forward declaration
TEST_BLOCK( abc ) {// Runs on site, as part of the Sequence table
   // Cause the Host to execute the body code of need_x, then wait
   // for execution to complete before proceeding.
   remote_send( need_x, 0, TRUE, INFINITE ); // 0 = send to Host
   return TRUE;
}
```

```
// The need_x user variable body code is designed to run in the
// Host process, as triggered by remote_send() from a site.
// Like any user variable body, only one invocation of this body is
// possible at a time; therefore all need_x requests by the sites
// are automatically serialized and synchronized.
VOID_VARIABLE( need_x, "" ) {
   // Execute the body code of got_x on the site which invoked
   // need_x, and wait until that transaction completes. The
   // built-in variable 'sender' is part of every user variable, and
   // can be used to identify which site invoked remote_send() or
   // remote_set().
   remote_send( got_x, sender, TRUE, INFINITE );
   // Execute any host code to process the value in need_x, now that
   // the site 'sender' has completed setting things up.
}
VOID_VARIABLE( got_x, "" ) {
   // Execute site code here needed before allowing the host
   // to continue, as described in need_x above.
}
```

7.2.3 remote_fetch()

Note: see Overview in Binning for an explanation of the different processes noted below.

Description

Note: the term local process is used below to indicate the process which executed `remote_fetch()`. The term remote process indicates the other process.

Like `remote_send()`, the `remote_fetch()` function is used to synchronize the values of User Variables in separate processes (on separate sites). This can be used to synchronize user variables between Host, and/or Site(s), and/or User Tools processes.

Using `remote_fetch()` requires that the referenced user variable(s) be *defined* in both processes. In contrast, remote_get() can be used to read a user variable value from a remote process without the need to create the user variable in the local process. Or, if the user variable exists in both processes, remote_get() can be used to read a user variable value from the remote process without modifying the same user variable in the local process.

`remote_fetch()` *pulls* the current value of the user variable from the remote process and assigns it to the same user variable in the local process. Optionally, the body code of the user variable can be executed in the local process, after the new value has been assigned to the user variable. The local process has the option of waiting for the transaction to complete before execution continues.

`remote_fetch()` is used in two contexts:

- Referencing one user variable
- Referencing a snapshot (see `SNAPSHOT()`)

When `remote_fetch()` references a snapshot all of the user variables defined in the `SNAPSHOT` are synchronized. The current values of the associated user variables in the remote process are assigned to the corresponding user variables in the local process. Using the timeout parameter, the local process has the option of waiting for the entire transaction to complete before execution continues. The sequence of user variable body code execution follows the order the user variables are listed in the `SNAPSHOT()`, and body code execution of each user variable will complete before the body code of the next user variable begins.

Usage

```
BOOL remote_fetch(VariableProxy v,
                  int site,
                  BOOL invoke,
                  DWORD timeout DEFAULT_VALUE( INFINITE ) );
BOOL remote_fetch(Snapshot *snapshot,
                  int site,
                  BOOL invoke,
                  DWORD timeout DEFAULT_VALUE( INFINITE ) );
```

where:

v represents one user variable to be synchronized. A `VariableProxy` type refers to User Variables and can be:

- A user variable name
- A quoted user variable name (string)

- A pointer to a user variable

snapshot represents a snapshot; i.e. a collection of user variables. All user variables in the SNAPSHOT( ) will be synchronized.

site is one of the standard site numbers identifying the remote process i.e. the site *from* which the user variable value, or snapshot, will be read, as follows:

- -1 = UI - User Interface process, used with UI User Variables only
- 0 = Host process
- 1 through 60 represent Site1 through Site60 processes
- A value > 1024 is a User Tools (see site_num( ))

invoke determines whether the body code of the user variable is executed in the local process. TRUE will cause the body code to execute, FALSE will not. When a snapshot is referenced the body code of all user variables in the SNAPSHOT( ) will execute (see above).

timeout specifies how long the function will wait before proceeding, in milliseconds. This includes any time spent executing the user variable body code in the local process. If not specified, the **timeout** value defaults to INFINITE, which can also be used explicitly to wait indefinitely. If the transaction completes before time expires, remote_fetch( ) returns TRUE, otherwise it returns FALSE.

Example

This example shows the Host process fetching the value of the user variable named my_val from Site 2.

```
INT_VARIABLE( my_val, 0, "") {}

HOST_BEGIN_BLOCK( HBB1 ) {
   // Other code here as needed
   if ( ! remote_fetch ( my_val, 2, FALSE, INFINITE ) )
      warning(" Warning: Host did not receive my_val from Site 2");
   else
      output(" my_val from site 2 => %d", my_val);
}
```

7.2.4 remote_set(), remote_get()

Note: see Overview in Binning for an explanation of the different processes noted below.

Description

Note: the term local process is used below to indicate the process which executed `remote_get()`or `remote_set()`. The term remote process indicates the other process.

The `remote_set()` function is used to explicitly set the value of a user variable in a remote process. The `remote_get()` function is used to retrieve the value of a user variable from a remote process.

This allows code in a Host process, Site process, or User Tools process to set or get the value of a user variable from one of the other process.

Unlike `remote_send()` and `remote_fetch()`, which are used to *synchronize* user variable values between processes, the `remote_set()` and `remote_get()` are used to transmit information between processes. Using `remote_send()` or `remote_fetch()` requires that the user variable be *defined* in both the local and remote site. In contrast, `remote_set()` and `remote_get()` can be used to directly transmit and receive values between two processes without the need for a common user variable.

Usage

```
BOOL remote_set(  VariableProxy v,
                  ValueProxy value,
                  int site,
                  BOOL invoke DEFAULT_VALUE( TRUE ),
                  DWORD timeout DEFAULT_VALUE( 0 ) );
CString remote_get( VariableProxy v,
                    int site,
                    BOOL invoke DEFAULT_VALUE( FALSE ),
                    DWORD timeout DEFAULT_VALUE( INFINITE ) );
```

where:

v represents one user variable to be accessed. It is not legal to access a VOID_VARIABLE. A VariableProxy type refers to User Variables and can be:

- A user variable name
- A quoted user variable name (string)
- A pointer to a user variable

value is the value to be set, using remote_set(), in the specified user variable. The value parameter must be of a type compatible with the user variable being modified (BOOL_VARIABLE, FLOAT_VARIABLE, etc. The value parameter is sent to the remote process as a CString, but is converted to the proper type before being assigned to the user variable. Error checking is done for proper type matching.

site is one of the standard site numbers identifying the remote process, as follows:

- -1 = UI - User Interface process
- 0 = Host process
- 1 through 60 represent Site1 through Site60 processes
- A value > 1024 is a User Tools (see site_num()).

invoke determines whether the body code of the user variable is executed in the *receiving* process. TRUE will cause the body code to execute, FALSE will not. The *receiving* process for remote_set() is the remote process specified by the **site** parameter. The **invoke** should always be set = FALSE (default) when using remote_get().

timeout specifies how long the function will wait before proceeding, in milliseconds. This includes any time spent executing the user variable body code. If not specified, the **timeout** value defaults to zero. The value INFINITE can be used to wait indefinitely for the transaction to complete. If the transaction completes before time expires, remote_set() returns TRUE, otherwise it returns FALSE. If the transaction completes before time expires, remote_get() returns the value read from the remote **site** as a CString, which must be converted to the desired type as needed (using atoi(), atof(), etc.). remote_get() returns a NULL string if a **timeout** occurs.

Examples

Example 1:

This example causes the body code of ui_StartTest (see UI User Variables) to execute in the UI - User Interface process (-1). The default timeout (0) is used i.e. execution does not wait.

```
// Cause UI to send a start test signal to all sites
```

```
remote_set( "ui_StartTest", "", -1, TRUE ); // Value is unused
```

Example 2:

This example also references one of the UI User Variables (`ui_SiteMask`). The local process (could be Host, Site, or User Tools) requests the current value of `ui_SiteMask` from UI - User Interface (-1), converts the returned value from `string` to `int` and assigns it to the local variable `mask`.

```
// Get the UI's current sitemask. Don't forget remote_get() returns
// a string
DWORD mask = atoi( remote_get( "ui_SiteMask", -1 ) );
```

Example 3:

This example causes the body code of user variable `remote_message_box` to execute in the Host process to display a message box.

```
// Intended for Host execution
CSTRING_VARIABLE( remote_message_box, "", "" ) {
   // Note that 'sender' is implicitly available in user variable
   // bodies
   AfxMessageBox( vFormat( "The message is %s from %d",
                           remote_message_box, sender ) );
}
SITE_BEGIN_BLOCK( SB1 ) {
   // Display a message box on the Host, wait until it's dismissed
   remote_set(  remote_message_box,
                "Hello, Mr. Host",
                0,
                TRUE,
                INFINITE );
   output( "Message box done" );
}
```

Example 4:

The following example is a contrast between `remote_send()` and `remote_set()`. In this example, to use `remote_send()` requires that `ui_StartTest` be declared in the test program:

```
VOID_VARIABLE( ui_StartTest, "" ) {}
void func() {
   remote_send( ui_StartTest, -1, TRUE );
}
```

Using `remote_set()` is more direct, but the spelling must be exact - no error checking is done at compile-time:

```
void func() {
   // Value doesn't matter for ui_StartTest
   remote_set( "ui_StartTest", "", -1 );
}
```

Example 5:

The following example is a contrast between `remote_fetch()` and `remote_get()`. In this example, to use `remote_fetch()` requires that `ui_SiteMask` be declared in the test program:

```
DWORD_VARIABLE( ui_SiteMask, "", "" ) {}
void func() {
   remote_fetch( ui_SiteMask, -1, FALSE );
}
```

Using `remote_get()` is simpler:

```
void func() {
   // Note: remote_get() returns an ASCII string
   DWORD mask = atoi (remote_get( "ui_SiteMask", -1 ) );
}
```

7.2.5 Transferring Multiple User Variables

Description

A *snapshot* is used to define a *set* of User Variables.

A snapshot can be used in conjunction with `remote_send()` and `remote_fetch()` to synchronize multiple user variables with a single function call.

Using `remote_send()` and `remote_fetch()`, the body code of each user variable in the snapshot can be executed. When this option is used, the order of execution follows the

order the user variables are listed in the snapshot. And, the body code of one user variable executes to completion before execution of the body code of the next user variable begins.

The `SNAPSHOT()` macro is used to create a new snapshot.

The `VARIABLE()` macro is used in the `SNAPSHOT()` macro body-code to add one User Variables to the snapshot.

The `VARIABLES()` macro is used in the `SNAPSHOT()` macro body-code to add all User Variables from a specified User Dialog to the snapshot.

The `REMOVE_VARIABLE()` macro is used in the `SNAPSHOT()` macro body-code to remove one specified User Variables from the snapshot.

The `INCLUDE_SNAPSHOT()` macro is used in the `SNAPSHOT()` macro body-code to add all of the User Variables from an existing snapshot to the new snapshot.

The `EXTERN_SNAPSHOT()` macro is used to make a forward or external declaration.

Usage

```
EXTERN_SNAPSHOT( other_snapshot ) // Defined somewhere else
SNAPSHOT( snapshot_name ) {
   INCLUDE_SNAPSHOT( other_snapshot )
   REMOVE_VARIABLE( some_uvar_from_other_snapshot );
   VARIABLE( single_var );
   VARIABLES( dialog_name );
}
```

where:

`EXTERN_SNAPSHOT` is a Test System Macro used to declare an external snapshot i.e. one created elsewhere (or below a reference in the same file).

`SNAPSHOT` is a Test System Macro used to create a new snapshot.

`snapshot_name` is a user-defined name for the collection of variables in this **`SNAPSHOT`**. To transfer multiple user variables at one time, this name is used as the user variable in a `remote_send()` or a `remote_fetch()`.

INCLUDE_SNAPSHOT is a Test System Macro used to add the user variables contained in another snapshot to the one being created. Multiple lines of INCLUDE_SNAPSHOT may be included in the snapshot macro.

REMOVE_VARIABLE is a Test System Macro used to remove a single user variable from the snapshot. Multiple lines of REMOVE_VARIABLE may be included in the SNAPSHOT macro.

VARIABLE is a Test System Macro used to define a single user variable named **single_var**. Multiple lines of VARIABLE may be included in the SNAPSHOT macro.

VARIABLES is a Test System Macro used to add all user variables associated with a specific User Dialog. Multiple lines of VARIABLES may be included in the SNAPSHOT macro.

dialog_name identifies a User Dialog. All user variables tied to resources in the specified dialog are added to the snapshot.

Example

In the example below, three user variables are defined and added to the snapshot named DUT_type. The num_DUT_pins user variable is removed from the list (just to show how it is done). Then, when the HOST_BEGIN_BLOCK() executes, the snapshot is sent to Site 1 using one remote_send() function call.

```
INT_VARIABLE( num_DUT_pins, 0, "" ) {}
BOOL_VARIABLE( high_speed, FALSE, "" ) {}
FLOAT_VARIABLE( DUT_VCC, 5.0, "" ) {}

SNAPSHOT( DUT_type ) {
   VARIABLE( num_DUT_pins );
   VARIABLE( high_speed );
   VARIABLE( DUT_VCC );
   REMOVE_VARIABLE( num_DUT_pins ); // Removed it
}

HOST_BEGIN_BLOCK( hbb1 ) {
   remote_send( DUT_type, 1, FALSE ) {} // Send to Site 1
}
```

7.2.6 Transferring User-defined Data Structures (Serialization)

Description

The `CALL_SERIALIZE()` macro may be used to transfer arbitrary user-defined variables and/or data structures between Host/Site/Tool processes.

In general, a `CSTRING_VARIABLE` user variable is used to pass the information between processes. In the *source* site (process), the user-defined variables and/or data structures are *serialized* into this user variable. Then `remote_send()`, or `remote_fetch()` are used to transfer the user variable from *source* to *destination* process, after which the information is un-serialized in the *destination* process. The serialization mechanism is built-in to the Microsoft libraries, and can be researched using the on-line help in Developer Studio.

The methodology used has several steps, best described with examples, but also outlined in more detail here:

- Create the user-defined structure(s) as desired. These definitions must be identical in both the source and destination sites (processes):

```
struct myStruct {
   int myInt;
   myType myTypeVar; // Must also spec Serialize()
}
```

- Within each data structure definition specify a `Serialize()` function. The body code of this function must specify how to both serialize (`CArchive.IsStoring`) and un-serialize (`CArchive.IsLoading`) the data elements of interest. Note that only those data elements to be serialized need to be specified i.e. any elements which are not included will be ignored. For data types with built-in serialization support (i.e. `int`, `CString`, etc.), the **>>** and **<<** operators must be used. For data types which don't have **<<**/**>>** support, the `.Serialize` operator must be used:

```
struct myStruct {
   int myInt;
   CString myString;
   myType myTypeVar;

   void Serialize( CArchive &ar ) {
   // The order of the elements below doesn't matter, but they
   // MUST be the same for both parts of the 'if'.
   if ( ar.IsLoading() ) {
```

```
      ar >> myInt;                // Supported variable syntax
      ar >> myString;
   }
   else { // ar.IsStoring()
      ar << myInt;
      ar << myString;
   }
   myTypeVar.Serialize (ar);   // Class/struct syntax
};
```

- The CALL_SERIALIZE() Test System Macro is invoked once for each data structure to be serialized:

  ```
  CALL_SERIALIZE( myStruct )
  ```

  CALL_SERIALIZE enables the **<<** and **>>** operators for use in C-code to serialize and un-serialize data via the CSTRING_VARIABLE container noted below.

- Create variables of the struct as desired:

  ```
  myStruct instance1;
  ```

- Create a CSTRING_VARIABLE to be used as the container for the serialized data. This definition must be shared in both the source and destination sites (processes).

- As needed, serialize the desired data into this CSTRING_VARIABLE in the source process.

  ```
  // All CStringArchive's used in this way must be in their own
  // block, so that their destructor gets a chance to update
  // serialized_myStruct.
  {
     CStringArchive ar( &serialized_myStruct );
     ar << instance1;
  }
  ```

  The example below shows how the body code of this user variable can be used to perform the un-serialize step noted below.

- Use remote_send(), or remote_fetch() to update the user variable between processes.

- Un-serialize the user variable into the appropriate variable in the destination process:

  ```
  // All CStringArchive's used in this way must be in their own
  // block, so that their destructor gets a chance to update
  // serialized_myStruct.
  ```

```
{
   CStringArchive ar( &serialized_myStruct );
   instance1 >> ar;
}
```

Usage

See Description and Examples.

Example

This example is rather large and consists of:

- Program Code
- Dialog Image
- Example Host Output
- Example Site Output

Program Code

This is a complete, single file, test program. The program does not perform any of the normal testing functions but does demonstrate how information in nested user-defined structures can be transferred from Site to Host.

The program defines a dialog (see Dialog Image) containing a single button. This is used to invoke the process, each time the Site changes some struct data values and they are transferred to the Host.

Note: the *WARNING* in the body code of the user variable named `site_get_info`:

```
#include "TestProgApp/public.h"
#include "resource.h"

// Two user-defined structs. The local Serialize() function in each
// is required to support sending data between sites via CSTRING
// uVar.

struct myStruct1 {
   doublemyDouble1, myDouble2;
   int  myInt1, myInt2;
   CString myCString1, myCString2;
```

```
   // Everything below this line is the 'serializer', needed to
   // support sending this struct between sites. By defining this
   // function as part of the struct and using CALL_SERIALIZE
   // below, many details are handled.
   void Serialize( CArchive &ar ) {
      // The order of the elements below doesn't matter, but they
      // MUST be the same for both parts of the 'if'. Syntax is for
      // 'supported' types (see MsDev doc and next struct).
      if ( ar.IsLoading() ){
         ar >> myDouble1;
         ar >> myDouble2;
         ar >> myInt1;
         ar >> myInt2;
         ar >> myCString1;
         ar >> myCString2;
      }
      else { // ar.IsStoring()
         ar << myDouble1;
         ar << myDouble2;
         ar << myInt1;
         ar << myInt2;
         ar << myCString1;
         ar << myCString2;
      }
   }
};
struct myStruct2 {
   float myFloat1;
   myStruct1 mS1a, mS1b; // See above
   BOOL myBool1;

   // Note that syntax used below is different for 'supported'
   // types vs. Class/structs.
   void Serialize( CArchive &ar ) {
      if ( ar.IsLoading() ){
         ar >> myFloat1;        // Variable syntax
         ar >> myBool1;
      }
      else { // ar.IsStoring()
         ar << myFloat1;
```

```
      ar << myBool1;
    }
      mS1a.Serialize( ar );  // Class or struct syntax
      mS1b.Serialize( ar );
  }
};
// Enable the << and >> operators for use in C-code.
CALL_SERIALIZE( myStruct2 )
CALL_SERIALIZE( myStruct1 )
//=================================================================
// Make one
myStruct2 myVar1;
int cnt = 0; // demo only

//=================================================================
// CSTRING uVar used to contain serialized data to be passed
// between sites. Data is put into this uVar by the body code of
// the uVar site_get_info, which is invoked using remote_set() from
// the Host/Tool. Then, the data is retrieved from by the Host/Tool
// using remote_fetch(), which executes this body code to
// un-serialize the data.
CSTRING_VARIABLE( serialized_myVar1, "", "" ) { // Host Execution
  myVar1.Serialize( CStringArchive( serialized_myVar1 ) );
}
//=================================================================
// Print values to see things change. The Example Host Output and
// Example Site Output were generated by code which included
// prettier formatting (was't very readable here).
void print_vals( myStruct2 *v ) {
  output("\n======= ---  Modified %d Times  --- =======", cnt++);
  output("myVar1.myBool1 => %s", v->myBool1 ? "TRUE" : "FALSE");
  output("myVar1.myFloat1 => %3.3f", v->myFloat1);
  output("myVar1.mS1a.myDouble1 => %4.4f", v->mS1a.myDouble1);
  output("myVar1.mS1a.myDouble2 => %4.4f", v->mS1a.myDouble2);
  output("myVar1.mS1a.myInt1 => %d", v->mS1a.myInt1);
  output("myVar1.mS1a.myInt2 => %d", v->mS1a.myInt2);
  output("myVar1.mS1a.myCString1 => %s", v->mS1a.myCString1);
  output("myVar1.mS1a.myCString2 => %s", v->mS1a.myCString2);
}
```

```
//=============================================================
// Host/Tool uses remote_send() to invoke this uVar on the Site.
// The body code serializes the desired data into the uVar
// serialized_myVar1. The Host/Tool uses remote_fetch() to
// retrieve it.
VOID_VARIABLE( site_get_info, "") { // Site execution
   // Modify values on Site.
   myVar1.myBool1 = ! myVar1.myBool1;
   myVar1.myFloat1= myVar1.myFloat1 + 101.101;
   myVar1.mS1a.myDouble1= myVar1.mS1a.myDouble1 * 2;
   myVar1.mS1a.myDouble2= myVar1.mS1a.myDouble1 + myVar1.myFloat1;
   myVar1.mS1a.myInt1= myVar1.mS1a.myInt1 - 1;
   myVar1.mS1a.myInt2= myVar1.mS1a.myInt1 + 1;
   myVar1.mS1a.myCString1= "Updated Site Values ";
   myVar1.mS1a.myCString2  += "X";
   print_vals( &myVar1 );

   // Serialize the data into the CSTRING uVar serialized_myVar1.
   // WARNING: all CStringArchive's used in this way must be scoped
   // to their own block, so that their destructor completely
   // updates the archive (data is spooled). ONLY this code should
   // be within the braces.
   {
      CStringArchive ar( &serialized_myVar1 );
      ar << myVar1;
   }
}
//=============================================================
// Dialog button requests site to return info, which is
// done via the uVar serialized_myVar1.
VOID_VARIABLE( Button1, "") { // HOST execution. See Dialog Image
   // Signal site to serialize the data into the uVar
   // serialized_myVar1
   remote_send(  site_get_info, 1, TRUE, INFINITE );
   // Retrieve the serialized data, and invoke the body code
   // here to un-serialize it in this thread.
   remote_fetch( serialized_myVar1, 1, TRUE, INFINITE );
```

```
   // See the changes in Host window
   print_vals( &myVar1 );
}
DIALOG( myDialog ) {
   IMMEDIATE_CONTROL( IDC_Button1, Button1 ) // See Dialog Image
}
// Invoke dialog only after the program has loaded and
// the initial values are printed
VOID_VARIABLE( ui_ProgLoaded,"" ) {
   run_modeless ( myDialog );
}
HOST_BEGIN_BLOCK ( HBB1 ) {
   // Set some initial values on Host
   myVar1.myBool1 = FALSE;
   myVar1.myFloat1= -1.1;
   myVar1.mS1a.myDouble1= -1.1;
   myVar1.mS1a.myDouble2= -1.1;
   myVar1.mS1a.myInt1= -1;
   myVar1.mS1a.myInt2= -1;
   myVar1.mS1a.myCString1= "Initial Host values";
   myVar1.mS1a.myCString1= "";
   output("\n======= --- Initial Host Values --- =======\\");
   print_vals( &myVar1 );
}
SITE_BEGIN_BLOCK( SBB1 ) {
   // Set some initial values on Site
   myVar1.myBool1 = FALSE;
   myVar1.myFloat1= 0.0;
   myVar1.mS1a.myDouble1= 0;
   myVar1.mS1a.myDouble2= 0.0;
   myVar1.mS1a.myInt1= 0;
   myVar1.mS1a.myInt2= 0;
   myVar1.mS1a.myCString1= "Initial Site values";
   myVar1.mS1a.myCString1= "";
   output("\n======= --- Initial Site Values --- =======\\");
   print_vals( &myVar1 );
}
```

Dialog Image

The dialog ID is "MYDIALOG". The button ID is IDC_Button1.

Example Host Output

```
======= --- Initial Host Values --- =======
======= ---   Modified 0 Times  --- =======
          myVar1.myBool1 => FALSE
         myVar1.myFloat1 => -1.100
    myVar1.mS1a.myDouble1 => -1.1000
    myVar1.mS1a.myDouble2 => -1.1000
       myVar1.mS1a.myInt1 => -1
       myVar1.mS1a.myInt2 => -1
  myVar1.mS1a.myCString1 =>
  myVar1.mS1a.myCString2 =>

======= ---   Modified 1 Times  --- =======
          myVar1.myBool1 => TRUE
         myVar1.myFloat1 => 101.101
    myVar1.mS1a.myDouble1 => 0.0000
    myVar1.mS1a.myDouble2 => 101.1010
       myVar1.mS1a.myInt1 => -1
       myVar1.mS1a.myInt2 => 0
  myVar1.mS1a.myCString1 => Updated Site Values
  myVar1.mS1a.myCString2 => X

======= ---   Modified 2 Times  --- =======
          myVar1.myBool1 => FALSE
         myVar1.myFloat1 => 202.202
    myVar1.mS1a.myDouble1 => 0.0000
    myVar1.mS1a.myDouble2 => 202.2020
       myVar1.mS1a.myInt1 => -2
       myVar1.mS1a.myInt2 => -1
  myVar1.mS1a.myCString1 => Updated Site Values
  myVar1.mS1a.myCString2 => XX
```

Example Site Output

```
======= --- Initial Site Values --- =======
======= ---   Modified 0 Times  --- =======
          myVar1.myBool1 => FALSE
         myVar1.myFloat1 => 0.000
   myVar1.mS1a.myDouble1 => 0.0000
   myVar1.mS1a.myDouble2 => 0.0000
      myVar1.mS1a.myInt1 => 0
      myVar1.mS1a.myInt2 => 0
  myVar1.mS1a.myCString1 =>
  myVar1.mS1a.myCString2 =>
The test program is loaded

======= ---   Modified 1 Times  --- =======
          myVar1.myBool1 => TRUE
         myVar1.myFloat1 => 101.101
   myVar1.mS1a.myDouble1 => 0.0000
   myVar1.mS1a.myDouble2 => 101.1010
      myVar1.mS1a.myInt1 => -1
      myVar1.mS1a.myInt2 => 0
  myVar1.mS1a.myCString1 => Updated Site Values
  myVar1.mS1a.myCString2 => X

======= ---   Modified 2 Times  --- =======
          myVar1.myBool1 => FALSE
         myVar1.myFloat1 => 202.202
   myVar1.mS1a.myDouble1 => 0.0000
   myVar1.mS1a.myDouble2 => 202.2020
      myVar1.mS1a.myInt1 => -2
      myVar1.mS1a.myInt2 => -1
  myVar1.mS1a.myCString1 => Updated Site Values
  myVar1.mS1a.myCString2 => XX
```

7.2.7 SiteMask() Support

Note: see Overview in Binning for an explanation of the different processes noted below.

Description

`SiteMask()` may be used as an argument to many of the Nextest functions to specify which site(s) are to execute the function.

Note: the `SiteMask()` function is NOT usable as a stand-alone function - it is only used as an argument to other Nextest functions.

By default, the functions which communicate directly with tester hardware are executed in site process(es) only. These functions access such basics as voltages, timing, executing tests, accessing the APG, PMU, DPS, etc.

When it is necessary for user-written code executing in Host or Tool processes to communicate with tester hardware two basic methodologies are available:

- Execute User Variables body code using the remote functions: `remote_set()`, `remote_get()`, `remote_send()`, `remote_fetch()`, etc.
- Use the `SiteMask()` versions of the functions which communicate with hardware.

In very early test programs option-1 was the only option. Legacy test programs in which User Dialogs interacted with tester hardware used this methodology. The dialog code executed in the Host process and `remote_send()` was used to invoke user variable body code designed to interacted with the tester hardware in Site processes. Then either the body code sent results back to the Host process, or the Host code used `remote_fetch()` read back any information wanted from the site. This methodology works well but makes the solution somewhat program specific, because the user variables containing the needed body code were compiled into the test program.

Option-2 mostly eliminates the need to use user variable body code solely to interact with tester hardware, and thus simplifies the design of User Dialogs and User Tools.

Most of the Nextest functions which interact with tester hardware have a version which accepts a `SiteMask()` as the first argument to the function. For example:

```
vil ( 2 V );                    // Site execution only
```

```
vil ( SiteMask(0x3), 2 V );   // Host, Tool (or Site) execution
```

The first example can only execute in a Site process. The second can execute in Host or Tool processes. It can also execute in Site processes but this isn't good practice, and will execute slower than normal.

Using a `SiteMask()` argument enables the system software to transparently handle the inter-process communications between Host or Tool, and Site processes. All of the functions which accept a `SiteMask()` argument (see below) use the `SiteMask()` similarly.

The `SiteMask()` function takes one argument, which is a bit-wise value specifying on which sites the function is to be executed. A value of `0x1` causes the function to be executed on site 1 only. A value of `0xF` would execute the command on sites 1,2,3,4.

Thus, the two `vil()` examples above are not equivalent. The first programs `vil()` on all signal pins used in the test program, on all used sites. The second programs `vil()` on all used pins on sites 1 and 2 (`SiteMask(0x3)`).

The functions which can use the `SiteMask()` option are listed in two tables below. Two tables are used to differentiate between functions which *write* to hardware (*setter* functions) and those which *read* from hardware (*getter* functions). Some functions appear in both tables and it is the other arguments and the function return value which distinguish a *setter* from a *getter*.

The *getter* functions must be used with a `SiteMask()` which accesses only one site i.e. `0x1`, `0x2`, `0x4`, `0x8`, etc. This allows the existing functions to *get* (read) a single value as they were originally designed.

The `SiteMask()` option is NOT documented with each of the individual functions; it is an advanced feature and only documented here.

Note: functions in magenta below are not yet documented, or not released. Use with caution.

Table 7.2.7.0-1 *Getter* Functions Supporting SiteMask() Argument

`abase()`	`actualdata()`	`afield()`
`amain()`	`amax()`	`back_voltage()`
`back_voltage_enable()`	bin_get()	`count()`
`cycle()`	`data_reg_width()`	`dmain()` `dbase()`
dbm_size()		

Table 7.2.7.0-1 *Getter* Functions Supporting SiteMask() Argument *(Continued)*

		dps()
dps_current_high()	dps_current_low()	dps_vpulse()
dutadr()	dutdata()	dutmar()
	dutxadr()	dutyadr()
ecr_board_present()	edge_strobe()	erradr()
errmar()	errvar()	errxadr()
erryadr()	expectdata()	funtest()
	getedge1()	getedge2()
getedge3()	getedge4()	
getioedge1()	getioedge2()	
get_xdtopo()	get_ydtopo()	
intadr()		
ipar_force()	ipar_high()	ipar_low()
jamreg()		lbdata()
measure()	negative_clamp()	no_dps()
numx()	numy()	ParametricMode()
partest()	partime()	pattern_paused()
pe_board_mask()	positive_clamp()	prevadr()
prevdata()	prevmar()	prevxadr()
prevyadr()	ReadColRAM()	
ReadNextError()	ReadRowRAM()	reload()
	scandata()	
sdutadr()	sdutxadr()	sdutyadr()
serradr()	serrxadr()	serryadr()
sites_per_controller()	sprevadr()	sprevxadr()
sprevyadr()		test_pin_first_error()
tgmode()	total_all_bins()	
vecdata()	vih()	vihh()
vil()	voh()	vol()

Table 7.2.7.0-1 *Getter* Functions Supporting SiteMask() Argument *(Continued)*

vpar_force()	vpar_high()	vpar_low()
vz()	xbase()	xfield()
xmain()	xmax()	x_fast_axis()
ybase()	yfield()	yindex()
ymain()	ymax()	

Table 7.2.7.0-2 *Setter* Functions Supporting SiteMask() Argument

abase()	afield()	amain()
back_voltage()	back_voltage_enable()	bin_set()
count()	cycle()	data_reg_width()
data_strobe()	dmain() dbase()	
dps_current_high()	dps_current_low()	dps_vpulse()
edge_strobe()	dps()	intadr()
		ipar_force()
ipar_high()	ipar_low()	jamreg()
lbdata()		lvm_error_mode()
	measure()	negative_clamp()
	numx()	numy()
partime()	pipe_clear()	positive_clamp()
reload()	restart()	scandata()
	setedge1()	setedge2()
setedge3()	setedge4()	setioedge1()
setioedge2()	settime()	
tgmode()	timer()	
vecdata()	vih()	vihh()
vil()	voh()	vol()
vpar_force()	vpar_high()	vpar_low()
vz()	WriteColRAM()	WriteMainArray()

Table 7.2.7.0-2 *Setter* Functions Supporting SiteMask() Argument *(Continued)*

WriteRowRAM()	xbase()	xfield()
xmain()	x_fast_axis()	ybase()
yfield()	yindex()	ymain()

Note:

7.3 Resources

7.3.1 Overview

See Resources.

The term **`resource`** is used generically, to refer to key software components of a Magnum 1/2/2x test program.

For example, pin lists, test blocks, bins, etc. each have a Nextest-defined Resource Types. In any given test program there may be zero, one, or many instances of these resource types.

The user may also define resources. For example, `AllPins`, `IOpins`, `Databus`, etc. are examples of user-defined pin list resources.

The reason `resources` are important are the support functions available in the Magnum 1/2/2x software. Using these functions it is possible to:

- Obtain a list of names of every member of a given resource type. See `resource_all_names()`
- Given a pointer to a specific resource lookup its name. See `resource_name()`
- Given the name of a specific resource lookup to a pointer to it. See Resource Find Functions
- The definition of many user-defined resources can be modified at runtime. See `resource_deallocate()` and `resource_initialize()`.
- The automatic runtime initialization of some resources can be inhibited, and managed explicitly. See `resource_ignore()`.
- The selection of Single Resource Types can be changed without reloading the test program. See Resource Use Functions.

In most cases, the term *resource* refers to software objects that are created using *Nextest-defined* macros. These macros are normally executed once, and create a corresponding object which may be referenced once or repeatedly. For example:

- User-defined pin lists are created using the `PINLIST()` macro, which are normally only executed once, as the program loads. Each macro creates a `PinList` object which may be referenced throughout the test program. The

`resource_deallocate()` and `resource_initialize()` functions can be used at runtime to re-execute the original `PINLIST` macro, which can contain conditional code to create different versions of a given pin list each time it is executed.

- The `SEQUENCE_TABLE()` macro, used to define Sequence & Binning Tables, is executed once, as the program loads. This creates a corresponding software state machine which is executed each time a `Start Test` is invoked. The `resource_deallocate()` and `resource_initialize()` functions can be used at runtime to re-execute the currently selected `SEQUENCE_TABLE()` macro (which can contain conditional code), or can be combined with `Resource Use Functions` to select a different *active* SeqBinTable.

However, not all resources have similarly useful runtime capabilities. For example:

- Test blocks are resources. The test block body code and name can't be changed at runtime thus treating test blocks as resources at runtime isn't useful.
- User variables are resources, but in the context of a resource the only thing that can be done at runtime is to modify the user variables value, which is done easily by assigning a new value to the variable.

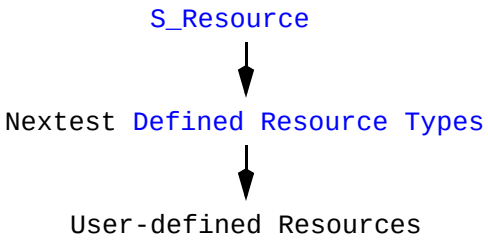
The key point here is that resources enable some useful capabilities, but not universally. The most useful capability are mostly constrained to resources of the following Resource Types:

- `S_Pattern`
- `S_PEBoardList`
- `S_PinList`
- `S_PinScramble`
- `S_ScanPattern`
- `S_SequenceTable`
- `S_VihhMap`

7.3.2 Resource Types

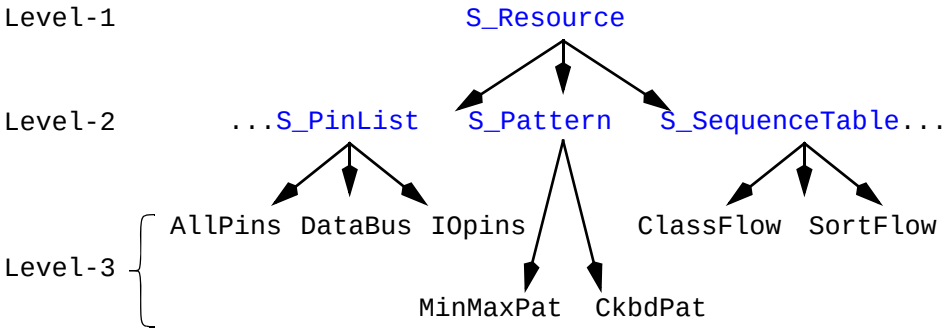
See Resources.

Resources are organized in a three level hierarchy:



Expanding this example, note that the names shown in level-1 and level-2 are Nextest-defined. `S_Resource` is the only level-1 resource, and provides a programmatic handle usable to access all other resources.

In the example below, Level-2 displays only 3 (`S_PinList`, `S_Pattern`, and `S_SequenceTable`) of the possible Defined Resource Types:



The names and quantity of each resource type in level-3 are user-defined. Any given test program may have zero, one, or many level-3 resources defined for each of the Defined Resource Types.

Some of the defined resource types are constrained, at runtime, to a single active instance. This means that the program may define multiple instances of that resource type, but at runtime one active instance must be selected. For example, at runtime only one Sequence & Binning Table can be active. See Single Resource Types and Single Resource Runtime Selection. These are flagged in left column of Defined Resource Types.

The table on the next few pages shows the defined resource types. Note that S_Resource is included in this list. The left column indicates which types are Single Resource Types. Note that these resource types are all type LPCTSTR.

Table 7.3.2.0-1 Defined Resource Types

Single Resource Type?	Resource Type	Reference
No	S_AfterTestingBlock	See AFTER_TESTING_BLOCK()
No	S_ATCBoardList	See ATC_BOARD_LIST in separate ATC Manual
No	S_AVSPinList	See AVS_PINLIST in separate ATC Manual
No	S_BeforeTestingBlock	See BEFORE_TESTING_BLOCK()
Yes	S_Configuration	See Configuration Macros
Yes	S_CurrentShare	See DPS Current Sharing
No	S_Dialog	See User Dialogs
No	S_DutPin	See DUT Pins
Yes	S_HostBeginBlock	See Host Begin Block
Yes	S_HostEndBlock	See Host End Block
Yes	S_HostConfiguration	See HOST_CONFIGURATION()
No	S_InitializationHook	See Initialization Hook
No	S_LogicVector	Each Logic Vector file (LVC file). See Compiling Test Patterns
No	S_Pattern	See Pattern Overview and Naming
Yes	S_PinAssignments	See Pin Assignment Table
No	S_PinList	See Pin Lists
Yes	S_PinScramble	See Pin Scramble Map
No	S_Resource	See note 1.
No	S_ScanPattern	See Scan Test Patterns

Table 7.3.2.0-1 Defined Resource Types *(Continued)*

Single Resource Type?	Resource Type	Reference
Yes	S_SequenceTable	See Sequence & Binning Table
Yes	S_SiteBeginBlock	See Site Begin Block
Yes	S_SiteEndBlock	See Site End Block
Yes	S_SiteConfiguration	See SITE_CONFIGURATION()
No	S_Snapshot	See User Variables
No	S_TestBin	See TEST_BIN()
No	S_TestBinGroup	See TEST_BIN_GROUP()
No	S_TestBlock	See Test Blocks
No	S_ToolBegin	See Tool Begin Block
No	S_ToolConfiguration	See TOOL_CONFIGURATION()
No	S_ToolEnd	See Tool End Block
No	S_Variable	See note 2.
No	S_Variable_BOOL	See User Variables
No	S_Variable_CString	See User Variables
No	S_Variable_double	See User Variables
No	S_Variable_DWORD	See User Variables
No	S_Variable_float	See User Variables
No	S_Variable_int	See User Variables
No	S_Variable_int64	See User Variables
No	S_Variable_OneOf	See User Variables
No	S_Variable_void	See User Variables
Yes	S_VihhMap	See VIHH Maps

Notes:

1) S_Resource can be used as a wild card, referencing all Nextest-defined resource types.

2) S_Variable can be used as a wild card, referencing User Variables of all types.

7.3.3 Resource Name Functions

See Resources.

Description

Given a pointer to an *initialized* resource, the resource_name() function returns the user-defined name of that resource.

The resource may be user-defined or one of the Nextest Defined Resource Types. The types of resources which are *initialized* will be different in each of the different execution contexts: Site, Host, UI, or Tool. See below.

The resource_all_names() function provides a way to obtain a list of names for all *initialized* instances of a specified resource type. Resource types are organized hierarchically, thus the names reported by resource_all_names()will depend on which resource level is referenced. Refer to the diagrams in Resource Types and note the following:

- If the resource argument passed to resource_all_names() is S_Resource the returned list will contain the names of the *initialized* Nextest Defined Resource Types.
- If the resource argument passed to resource_all_names() is one of the Nextest Defined Resource Types other than S_Resource, the returned list will contain the names of *initialized* user-defined resources of the specified resource type.
- The types of resources which are *initialized* will be different in each of the different execution contexts: Site, Host, UI, or TOOL. For example, Sites will initialize PinLists, whereas TOOLs won't. See Overview.

Usage

```
LPCTSTR resource_name( type *obj );
```

These versions of resource_all_names() returns an integer count of the number of names returned in the CStringArray argument.

```
int resource_all_names( LPCTSTR resource, CStringArray *names );
```

```
int resource_all_names(  LPCTSTR resource,
                         int site,
                         CStringArray *names );
int resource_all_names( Resource *resource, CStringArray *names );
```

These versions of `resource_all_names()` returns a comma separated list of names suitable for use in `ONEOF` User Variables.

```
CString resource_all_names( LPCTSTR resource );
CString resource_all_names( LPCTSTR resource, int site );
```

where:

`obj` is a pointer to the resource of interest, which can be a user-defined or Nextest-defined resource.

`resource` is the name of the resource of interest. These can be a user-defined or Nextest-defined resource. The names of Nextest-defined resources are listed in the Defined Resource Types.

`*resource` is a pointer to the resource of interest, which can be a user-defined or Nextest-defined resource.

`names` is a `CStringArray`. The user's program code declares a *variable* of `CStringArray` (***not*** a pointer to a `CStringArray`) and the `resource_all_names()` function adds elements to the array. The value returned by `resource_all_names()` is the number of elements in the array.

`site` is an integer value which identifies the Site to be queried, and specified as follows:

- -1 = UI - User Interface process
- 0 = Host process
- 1 through 32 represent Site1 through Site32 processes
- A value > 1024 is a User Tools (see `site_num()`)

Examples

Example 1:

This example uses the Nextest-defined `S_PinList` resource type. Three pin lists are used in the example, but the members of the pin list are not important to the example.

```
PINLIST ( SomePins ) {
```

```
   // Define pin list members here
}
PINLIST ( MorePins ) {
   // Define pin list members here
}
PINLIST ( MostPins) {
   INCLUDE_PINLIST ( MorePins )
   // Add more pin list members here
}
// Get a list of the names of all PinLists in the program
CStringArray plist_array;
int plist_count = resource_all_names( S_PinList, &plist_array );
output (" This program has %d PinLists", plist_count );
// Process every PinList in the program...
for ( int i = 0; i < plist_count; ++i ) {
   // If the PinList name contains "Mo" print MATCHED + PinList name
   // In this example, this will print "MorePins" and "MostPins"
   if ( plist_array[ i ].Find( "Mo" ) == 0 ) {
      output (" MATCHED pin list => %s", plist_array[ i ]);
   }
   else // Print SKIPPED + PinList name i.e. "SomePins"
      output (" SKIPPED pin list => %s", plist_array[ i ]);
   // Given a PinList name, look-up a pointer to that PinList
   PinList* plist = PinList_find ( plist_array[ i ] );
   // Use the pointer here as needed.
   vol ( 1 V, plist );
}
// Given a pointer to the PinList "SomePins", look-up its name
CString plist_name_str = resource_name ( SomePins );
output (" Pin List Name => %s", plist_name_str );
```

Example 2:

The following example will ***NOT*** work because the `namearray` variable is the wrong type; it is incorrectly declared as a pointer to a `CStringArray` variable rather than a variable of type `CStringArray`.

```
CStringArray *namearray;
int plist_count = resource_all_names( S_PinList, namearray );
```

This is a common programming error which is *difficult to debug*. It is syntactically legal, but functionally defective. The symptoms are unpredictable and often cause programs to crash, basically because memory is being clobbered in some undefined area of the program memory space. Below is the correct syntax:

```
CStringArray namearray;
int plist_count = resource_all_names( S_PinList, &namearray );
```

7.3.4 Resource Find Functions

See Resources.

Description

Given the name of a specific resource, the functions below return a pointer to that resource.

Each function is named after one of the Defined Resource Types. To obtain the desired functionality the user must use the appropriate function for each resource type.

```
AfterTestingBlock_find()
ATCBoardList_find()
AVSPinList_find()
BeforeTestingBlock_find()
Configuration_find()
CurrentShare_find()
Dialog_find()
HostBeginBlock_find()
HostEndBlock_find()
HostConfiguration_find()
InitializationHook_find()
LogicVector_find()
Pattern_find()
PatternSet_find()
PEBoardList_find()
PinAssignments_find()
```

```
PinList_find()
PinScramble_find()
Resource_find()
ScanPattern_find()
SequenceTable_find()
SiteBeginBlock_find()
SiteEndBlock_find()
SiteConfiguration_find()
Snapshot_find()
TestBin_find()
TestBinGroup_find()
TestBlock_find()
ToolBegin_find()
ToolConfiguration_find()
ToolEnd_find()
Variable_find()
Variable_BOOL_find()
Variable_CString_find()
Variable_double_find()
Variable_DWORD_find()
Variable_float_find()
Variable_int_find()
Variable_int64_find()
Variable_OneOf_find()
Variable_void_find()
VihhMap_find()
```

Usage

```
type* type##_find( LPCTSTR instance );
```

where:

type## represents the prefix portion of the function name i.e. everything except `_find()`.

instance is the name of a specific resource, of the type compatible with the `_find()` function name. For the single-resource types (see Defined Resource Types) the currently used resource can be determined by specifying 0 as the instance value.

The returned value is a pointer to the resource. If the resource is not found NULL is returned.

Example

This example returns a pointer to the pin list named `SomePins`. That pointer is then used to program `VOL` on those pins.

```
CString name = resource_select(S_PinList); // resource_select()
if (name) {
   PinList* plist = PinList_find (name);
   vol ( 1 V, plist );
}
```

7.3.5 Resource Control Functions

See Resources.

Description

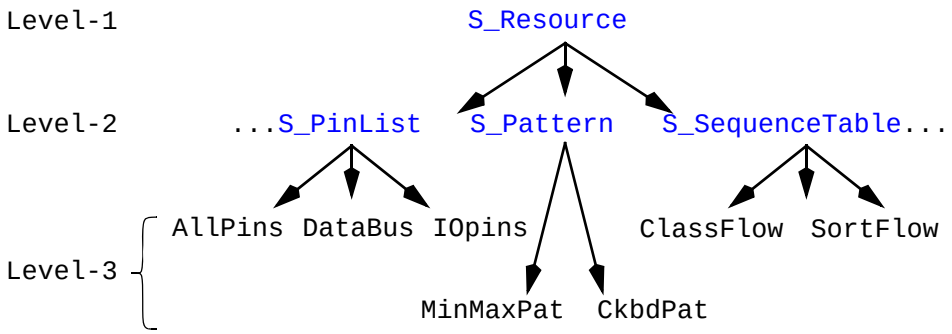
The `resource_deallocate()` function is used to *uninitialize* resources. This resets the specified resource to its uninitialized state (the resource is not destroyed), and sets a flag indicating the resource is uninitialized. An uninitialized resource must be initialized before it can be used in the normal context.

The `resource_initialize()` function causes the re-execution of the macro which was used to originally create the resource. Normally, these macros are only executed once, during program loading. If the macro contains conditional code, re-executing it allows the resource to be defined conditionally i.e. change the resource's values or attributes. The `resource_initialize()` function only (re)initialize resources which are currently uninitialized.

Both `resource_deallocate()` and `resource_initialize()` have an optional argument which can be used to specify a specific resource to process. This argument can be one of the Nextest Defined Resource Types or a user-defined resource (pin list, VIHH

Map, etc.). When no resource argument is specified the default is S_Resource, i.e. all resources in the test program.

When a user-defined resource is specified, only that resource is processed (initialized, or uninitialized). When one of the Nextest Defined Resource Types is specified all of the dependent resources are processed. For examples, note the diagram below:



- If `resource_initialize(DataBus)` or `resource_deallocate(DataBus)` is called, only the `DataBus` pin list resource is processed. And, `resource_initialize()` will only initialize `DataBus` if it is currently uninitialized.
- If `resource_initialize(S_PinList)` or `resource_deallocate(S_PinList)` is called, all pin lists are processed. And, `resource_initialize()` will only initialize pin lists which are currently uninitialized.
- If `resource_initialize(S_Resource)` or `resource_deallocate(S_Resource)` is called, every resource in the program is processed. Again, `resource_initialize()` will only initialize resources which are currently uninitialized.

Note: `resource_deallocate()` and `resource_initialize()` are low level commands. At this level, the system software does not automatically update dependent resources when a given resource is deallocated, modified, and reinitialized. For example, modifying the tester pin channel mapping using `S_PinAssignments` alone will *likely result in defective operation* because many other resources must also be deallocated/initialized to acquire the changes made to pin assignments. Until these dependencies are documented it is recommended that the `S_PinAssignments` resource NOT be specifically reinitialized. Rather, use the one method which guarantees that all dependencies are updated:

```
resource_initialize(S_Resource);
```

With Single Resource Types, to switch to an alternative resource requires that the currently selected resource be uninitialized (`resource_deallocate()`). Then `Resource Use Functions` are called to select the new resource, followed by `resource_initialize()` to initialize that resource and make it usable. This can be done, for example, to switch Sequence & Binning Tables at runtime.

It is possible to inhibit the initialization of resources as the program loads using `resource_ignore()`. This can be used, for example, to prevent patterns from automatically loading as the program loads. Then, later, patterns can be loaded using `resource_initialize(S_Pattern)`.

7.3.5.1 resource_deallocate()

See Resources.

Description

The `resource_deallocate()` function is used to *uninitialize* resources. See detailed introduction in Resource Control Functions.

When not specified, the default resource is `S_Resource` i.e. all resources in the program.

Usage

The `resource_deallocate()` function returns TRUE if the specified resource is found and was initialized, otherwise FALSE is returned.

```
BOOL resource_deallocate (
          LPCTSTR resource DEFAULT_VALUE( S_Resource ) );
BOOL resource_deallocate( Resource *resource );
```

where:

`resource` is optional, and identifies the resource being deallocated or initialized. See Resource Types. If not specified, the argument defaults to `S_Resource` i.e. all resources of all types.

`*resource` is a pointer to a specific resource instance being initialized.

Example

7.3.5.2 resource_initialize()

See Resources.

Description

The `resource_initialize()` function is used to *(re)initialize* resources. See detailed introduction in Resource Control Functions.

When not specified, the default resource is `S_Resource` i.e. all resources in the program.

Specific initialization details are not documented, and vary for each resource type. In general, the following is handled:

- Memory management
- Evaluation and/or set up of *SOME* dependencies (see Note above)
- Initialize default values
- Error checks
- Setting the initialization flag

Usage

The `resource_initialize()` function returns TRUE if the specified resource is found and is uninitialized, otherwise FALSE is returned.

```
BOOL resource_initialize (
   LPCTSTR resource DEFAULT_VALUE( S_Resource ) );
```

```
BOOL resource_initialize ( Resource *resource );
```

where:

resource is optional, and identifies the resource being deallocated or initialized. See Resource Types. If not specified, the argument defaults to S_Resource i.e. all resources of all types.

***resource** is a pointer to a specific resource instance being initialized.

Example

See Example

7.3.5.3 resource_ignore()

See Resources.

Description

The resource_ignore() function can be used to inhibit automatic resource initialization as the test program loads.

This will be most useful in preventing test patterns from automatically loading (S_Pattern, S_ScanPattern). Then, later in the test program, resource_initialize() can be used to cause test patterns to load.

In many cases, the resource_ignore() needs to be executed before resources are first initialized. In these situations this means before Configuring the Tester to the DUT i.e. within the CONFIGURATION() block code. In general, this applies to all resources except patterns, which are initialized after the various begin blocks have executed (HOST_BEGIN_BLOCK(), SITE_BEGIN_BLOCK(), or TOOL_BEGIN_BLOCK()).

Usage

The resource_ignore() function returns TRUE if the specified resource is found and is uninitialized, otherwise FALSE is returned.

```
BOOL resource_ignore( type *obj );
```

where:

obj is a pointer to the resource of interest, which can be a user-defined or Nextest-defined resource.

Example

The following example demonstrates how automatic loading of Memory Test Patterns and Logic Test Patterns can be inhibited.

```
SITE_BEGIN_BLOCK( SBB ) {
   // ... other code here
   resource_ignore( Resource_find ( S_Pattern ));
   // ... other code here
}
```

7.3.6 Resource Use Functions

See Resources.

Description

As noted in Single Resource Runtime Selection, the common method for selecting specific instances of Single Resource Types utilizes the `CONFIGURATION()` macro, which provides compile-time resource selection. Or, if the test program defines more than one instance of a given Single Resource Types, and one is not specified using the `CONFIGURATION()` macro, a dialog is presented at runtime requiring the user to make a selection. The *resource use* functions documented here provide an alternative runtime method.

The *resource use* functions are used to select a specific instance of one of the Single Resource Types and make it the active instance. Each of the Single Resource Types has a specific *use* function (see below).

To use these functions requires the following sequence:

- If necessary, use `resource_deallocate()` to uninitialize the currently selected resource. The currently selected resource can be identified using the appropriate Resource Find Functions and passing '0' as the argument. See Examples.
- Execute one of the *resource use* functions to *select* the desired resource
- `resource_initialize()` to initialize the newly selected resource

Note: `resource_deallocate()` and `resource_initialize()` are low level commands. At this level, the system software does not automatically update dependent resources when a given resource is deallocated, modified, and reinitialized. For example, modifying the tester pin channel mapping using `S_PinAssignments` alone will *likely result in defective operation* because many other resources must also be deallocated/initialized to acquire the changes made to pin assignments. Until these dependencies are documented it is recommended that the `S_PinAssignments` resource NOT be specifically reinitialized. Rather, use the one method which guarantees that all dependencies are udpated:

```
resource_initialize(S_Resource);
```

Each of the resource use functions below is named after one of the Single Resource Types. To obtain the desired functionality the user must use the appropriate function for each resource type.

```
Configuration_use();
CurrentShare_use();
HostBeginBlock_use();
HostEndBlock_use();
HostConfiguration_use();
PinAssignments_use();
PinScramble_use();
SequenceTable_use();
SiteBeginBlock_use();
SiteEndBlock_use();
SiteConfiguration_use();
VihhMap_use();
```

Usage

```
BOOL *type##_use( LPCTSTR instance );
```

where:

`*type##` represents the prefix portion of the function name i.e. everything except *_use()*.

`instance` is the name of a specific resource, of the type compatible with the *_use()* function name.

The returned value is `TRUE` if the specified resource name is valid and the use operation succeeds, otherwise `FALSE` is returned.

Examples

```
// Uninitialize the currently used SeqBinTable
resource_deallocate( S_SequenceTable );

// Select a new SeqBinTable to use
   if (SequenceTable_use("SeqTab2") == FALSE) {
      output ("  ERROR: specified SeqTable not found");
      return FAIL;
}

// Initialize it. This executes the SEQUENCE_TABLE() macro code
// to construct the executable SeqBinTable
resource_initialize( S_SequenceTable );
```

7.3.7 resource_select()

See Resources.

Description

The `resource_select()` function displays a dialog containing the names of all instances of the specified resource, allows user to select one, and the name of the selected resource.

This uses the same dialog that is presented during program loading when two or more instances of a Single Resource Types are defined in the test program and no `CONFIGURATION()` is specified to select one of the instances. However, `resource_select()` will operate on any of the Defined Resource Types (but not user-defined resources).

The `resource_select()` function will not display a dialog in the following situations:

- Zero instances of the specified Defined Resource Types exist. `resource_select()` returns `NULL`.

- One instance of the specified Defined Resource Types exist. `resource_select()` returns the name of that instance without presenting the selection dialog.

Usage

The return value is the name of the selected resource. See above for details.

```
CString resource_select( LPCTSTR resource );
```

where:

`resource` is one of the Defined Resource Types.

Example

This example presents a selection dialog displaying the names of all pin lists in the program.

```
CString name = resource_select(S_PinList);
if (name) {
   PinList* plist = PinList_find (name);
   vol ( 1 V, plist );
}
```

The returned `name` parameter will be `NULL` if the program contains no pin lists, or will be the name of the selected pin list. If name is not NULL a pointer to the pin list is found and VOL is programmed to 1V on those pins. Note that this will not function correctly if any DPS pins are in the selected pin list.

7.3.8 invoke()

See Resources.

Description

The `invoke()` function executes user-written C code associated with various types of Resources.

The `invoke()` function has several versions (overloads), each targeted at specific Resource Types. Only the following Resource Types have `invoke()` support:

- Dialog - invoke the specified User Dialog to start a modal dialog.
- TestBin - invoke the optional function defined for a specified Test Bin.

- HostBeginBlock - invoke the specified Host Begin Block.
- SequenceTable - invoke the specified Sequence & Binning Table.
- TestBlock - invoke the specified TEST_BLOCK.
- Pattern - invoke the Pattern Initial Conditions of the specified test pattern.
- VariableProxy - invoke the body code, if any, of the specified User Variables.

Some versions of the `invoke()` function return a value, others do not. See Usage.

Usage

As noted above, `invoke()` has several versions (overloads). These are all listed below, then documented separately.

```
int invoke( Dialog *dialog );
int invoke( TestBin* obj ); // See Test Bin invoke() Function
void invoke( HostBeginBlock *obj );
TestBin *invoke(  SequenceTable *obj,
                  int firstNode DEFAULT_VALUE( 0 ),
                  int lastNode DEFAULT_VALUE( -1 ) );
int invoke( TestBlock *obj );
void invoke( Pattern *obj );
void invoke( VariableProxy v,
             int sender DEFAULT_VALUE( site_num( ) ) );
```

The `invoke( Dialog )` function starts a modal User Dialog. The value returned by `invoke()` is the value returned by MFC's `CDialog::DoModal()` i.e one of:

- `IDOK` if user clicked **`OK`** button
- `IDCANCEL` is user clicked the **`CANCEL`** button or hit the **`escape`** key
- `IDABORT` if there was a problem
- The value passed to `dismiss( Dialog *dialog, int result )` if used.

The `invoke( HostBeginBlock )` function executes the specified `HostBeginBlock`. This is not recommended.

The `invoke( SequenceTable )` function executes the specified `SequenceTable`. Executing `invoke(0)` will execute the current sequence table. The value returned by `invoke()` is the final Test Bin. The Sequence & Binning Table can also be executed using `ui_StartTest` (see UI User Variables). The `firstNode` and `lastNode` arguments are

both optional, and are used to specify a starting and/or ending test block. When used, the starting and/or ending test block is specified by number, which is the position the test block is found in UI's UI Sequence and Binning sub-window.

The `invoke( TestBlock )` function executes the specified `TestBlock`. The value returned by `invoke()` is the value returned by the test block code. See Test Block Integer Return Values. When executed using `invoke()` many of the built-in support facilities are not operational (Test Numbers, Setup Numbers, breakpoints, etc.). These are available using `invoke( SequenceTable )`. The Before-testing Block and After-testing Block are executed. See Examples for an example of how to execute a single test block plus the Before-testing Block and After-testing Block.

The `invoke( Pattern )` function executes the Pattern Initial Conditions associated with the specified test pattern.

The `invoke( VariableProxy )` function executes the body code of the User Variables named by the `VariableProxy`. A User Variable always has associated body code, which sometimes does nothing. The optional `sender` argument can be used to pass an integer value to the body code, which accesses the value using the built-in `sender` variable which is local to the body code of every User Variable. The default value passed to the body code is the `site_num()` of the process which executes `invoke()`.

Examples

The following example is a user written function which will execute a single test block, specified by the tblock argument, plus the Before-testing Block and After-testing Block. This provides the same functionality as if the **`Run Only This`** option available via the right-mouse context menu in UI's Sequence/Binning Table sub-window:

```
TestBin *run_only_this( TestBlock *tblock ) {
  // Get the current sequence table
  SequenceTable *st = SequenceTable_find( 0 );
  // Find all instances of tblock in the selected sequence table
  IntArray array;
  int size = get_nodes( st, tblock, &array );
  // If the specified tblock is found execute it via the
  // sequence table. This causes the the Before-testing Block and
  // After-testing Block to be executed also.
  TestBin* bin;
  if( size ) {
    bin = invoke( st, array[ 0 ], array[ 0 ] );
    return( bin );
```

```
   }
   else{
      output("ERROR: TestBlock NOT found in current Seq/Bin");
      return( 0 );
   }
}
```

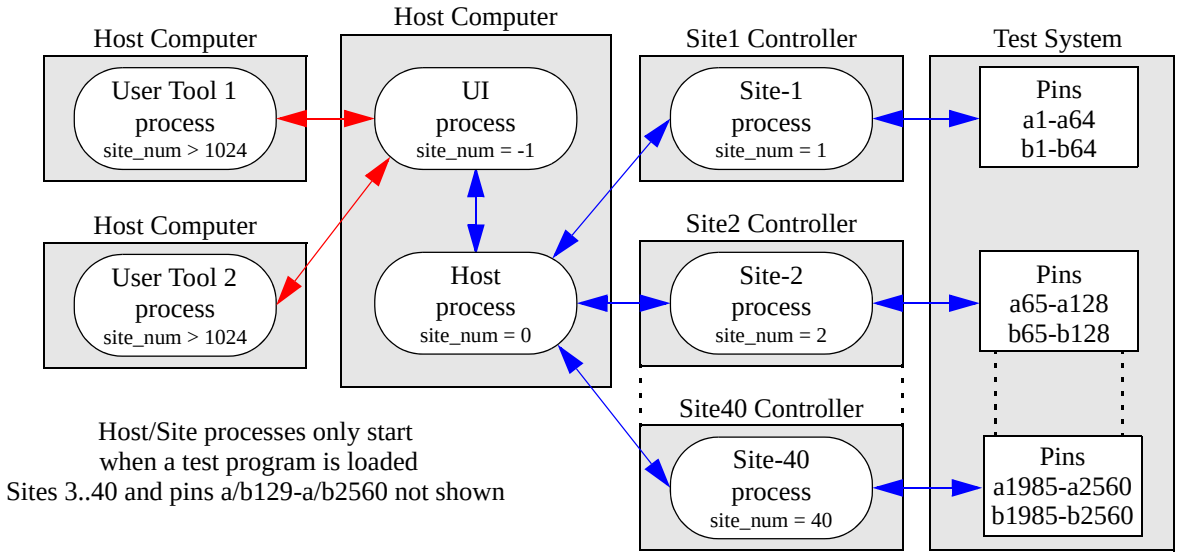
7.4 User Tools

This section contains the following:

7.4.1 Overview

User tools consist of user-written C-code compiled and executed independent of a test program, but which can communicate with UI - User Interface. And, if a test program is loaded, the user tool code can also communicate with the Host and Site instances of the test program.

The following diagrams shows how the various software processes communicate:



Each user tool and the Host and all Site copies of the test program execute as a separate process. Each process has a `site_num()`, which allows use of the various `remote_*` functions (more later). Using the `site_num()` mechanism allows the programmer to basically ignore the communication links between the various processes.

The blue paths (solid lines) are established when a test program is loaded - these are the standard program communications paths. Only Site processes can communicate directly with the tester hardware.

The red paths (dotted lines) are established when a user tool is started. They are used when the tool communicates with UI - User Interface. And, if a test program is loaded user tool code can communicate with the test program, in the Host process and all Site processes.

The tool communication paths are important because not all of the Nextest functions and macros are designed to execute in the tool (or Host) context. For example, the `funtest()` function typically used in Test Blocks executes in the Site process, and cannot be used (as is) to execute a test from a user tool (or from the Host process). As with any software, the programmer (tool creator) must be aware of, and use, the appropriate methods to obtain the desired results - see Creating User Tools.

The following information applies equally to user tool code, and the Host/Site test program code:

- As shown in the diagram above, each is a separate process (separate execution context), and is independent of the UI - User Interface process. The various processes are discussed in the Overview section of Binning.
- Connection is automatically made between the user tool process and UI - User Interface process. Similarly, when a test program is loaded, connections are automatically established between UI - User Interface and the Host and each Site copy of the test program. These connections enable communications between the various processes using User Variables, the `remote_*` functions, and the `site_num()` of each process.
- The user-written C code must contain a reference to the Nextest libraries to allow use of the various Nextest functions and macros. See Creating User Tools.
- Test programs and user tools do not have a user-written `main()`. The `main()` is handled by Nextest software. Initialization capabilities are provided by a set of macros. See User Tool Initialization and Binning.
- User tool code (and Host/Site code) can define and invoke User Dialogs. Each dialog executes in its local process, but can communicate with the other processes using User Variables, the `remote_*` functions, and the `site_num()` of each process.
- User Variables provide for exchange of information between processes. Optional user variable body code allows user-written code in one process to invoke user-written code in a different process.
- User-written code in one process can cause a different process to load a Dynamic Link Library (DLL). For example, user tool code can cause a test program to load a DLL, possibly containing the definitions of User Variables needed by the tool to communicate or control the test program.
- Available inter-process communication functions uses the `site_num()` of each process as an ID. The `remote_signal(), remote_wait()` functions can be used to synchronize execution between processes. The `remote_send()` and `remote_fetch()` functions can be used to synchronize the values of User Variables between processes. The `remote_set()` and `remote_get()` functions can be used to directly access User Variables in other processes
- The functions which directly interact with tester hardware normally execute only in Site process(es). However, special versions of these functions can be executed from user tool code or Host code, allowing the system software to transparently handle any necessary inter-process communication. See SiteMask() Support.
- User-defined pull-down menus can be added to the UI - User Interface menu bar via user tool code, or Host or Site code. Selecting from that menu executes code in the process which added the menu to UI - User Interface. See User Menus in UI.

Note: in any MFC application, it is required that there is one and only one CWinApp-derived instance. Nextest test programs already define a CWinApp-derived instance which is used for many things, including connecting to UI, initializing software structures and tester hardware. This functionality is provided by the include file "TestProgApp.h", which is required in every test program and every user-tool. Therefore, to merge an existing MFC program with Nextest software, it is required that you remove your CWinApp-derived object. In some cases an `INITIALIZATION_HOOK()` can be useful to perform one-time initialization. Often your CWinApp-derived instance is concerned with tasks that are not even relevant, and can simply be deleted.

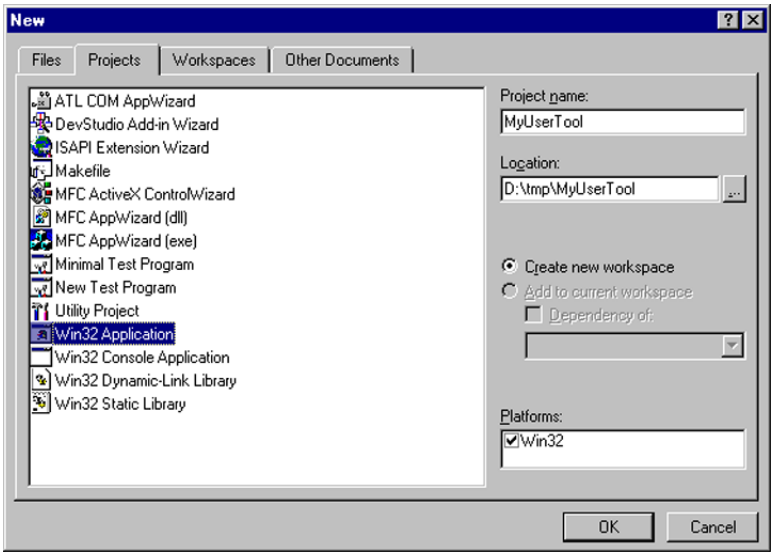
7.4.2 Creating User Tools

Note: the techniques used to create a User Tool using Developer Studio 4.2 were deleted @ 7/2002. If you are using a Nextest software release which depends upon Developer Studio 4.2, refer to the Programmer's Manual shipped with that release.

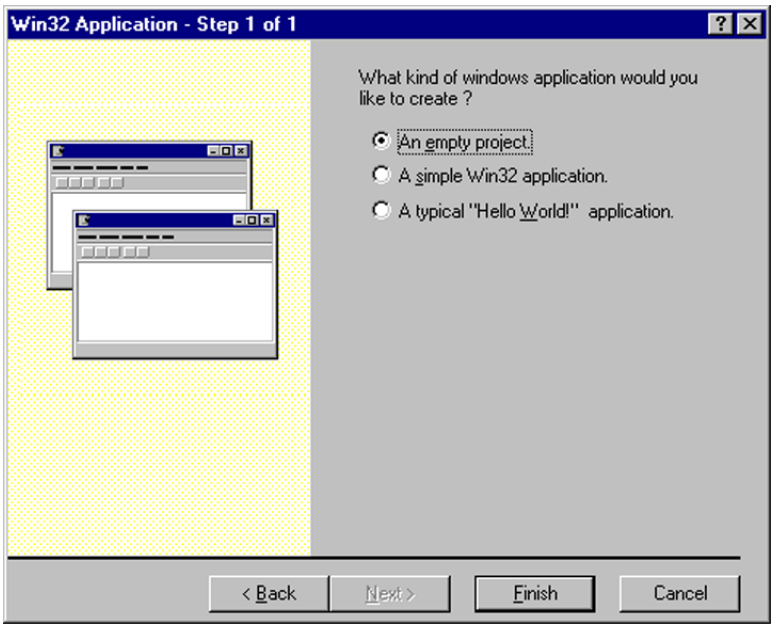
As noted in Overview, u*ser tools* consist of user-written C-code compiled, and executed, independent of a test program, but which can communicate with UI - User Interface. The following steps are used to create a simple user tool, consisting of one source file:

1. Identify (or create) a disk location to store the user tool source files (the project workspace).
2. Start Developer Studio 6.0 (MSDS) - do not open an existing Project Workspace.

3. In Developer Studio, use **File**: **New**, select *Win32 Application*, enter the **Project name** and **Location** information, ensure the **Create new workspace** and **Platforms** Win32 are checked, and click **OK**

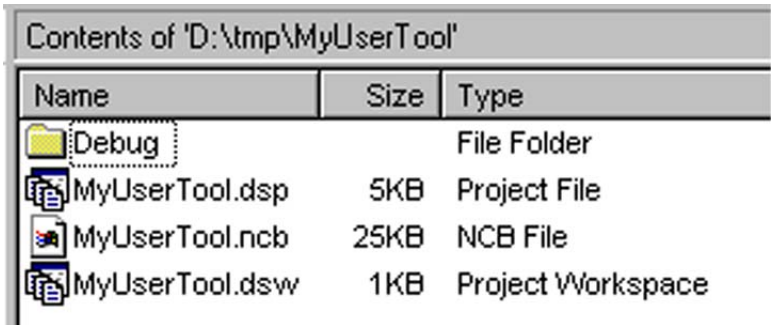


4. In the next dialog, make sure the **An empty project.** option is checked, then click **Finish**.

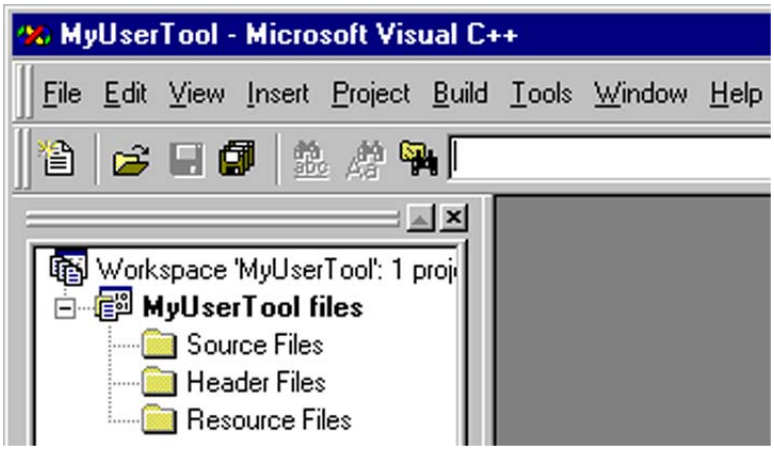


5. Click **OK** in the *New Project Information* dialog (not shown).

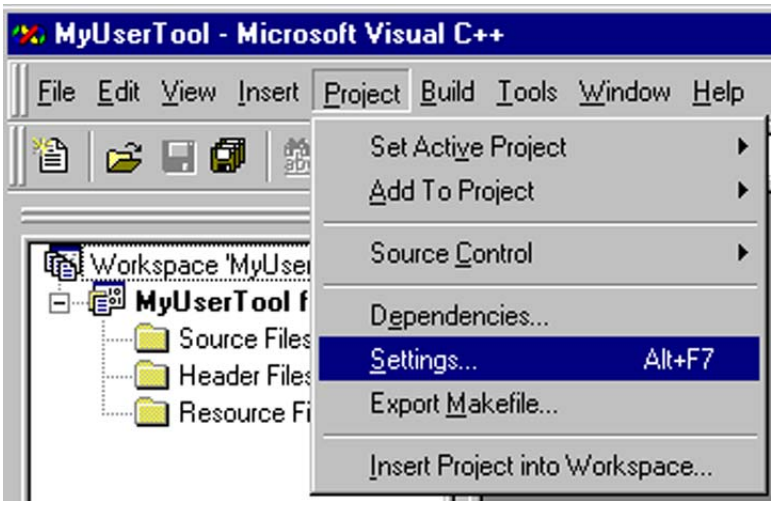
At this point, the project workspace folder is created on disk, at the specified location, and will contain 3 files and an empty Debug folder:



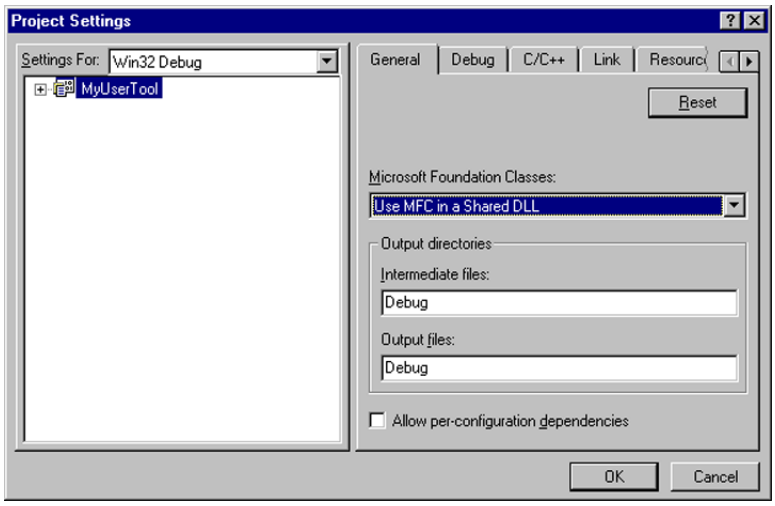
In Developer Studio, the project workspace will contain 3 empty folders:



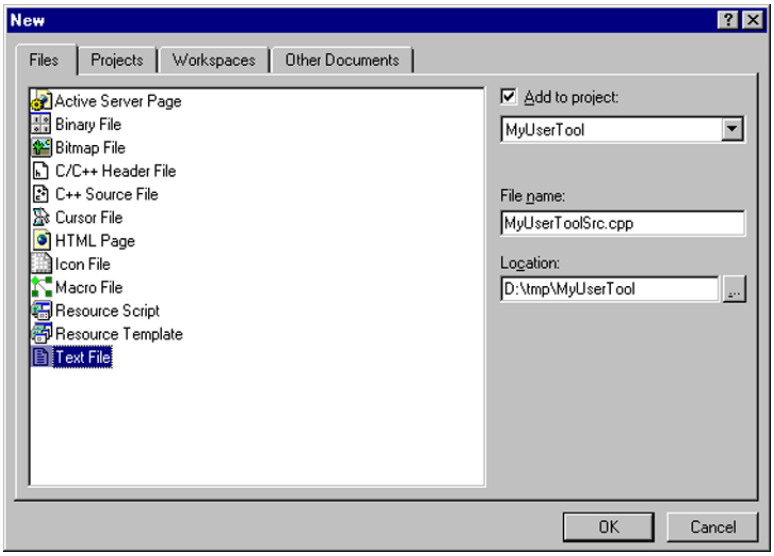
6. In Developer Studio, select **`Project: Settings...`**



In the **Microsoft Foundation Classes** select *Use MFC in a Shared DLL*. Don't change anything else. Click **OK**.



7. In Developer Studio, use **File:New** to create a new text file in the same location as the project. You should only need to select `Text File` and add the **File name**. Name the file using the *.cpp* file name extension:



Note that this step also automatically adds the file to the project.

8. In this file, add the following line at the top of the file:

```
#include "tester.h"
```

Adding this `#include` statement to any source file causes that file to be compiled (built) with the Nextest libraries. This allows user code in that file to access the Nextest

functions and macros, which in turn, enables that code to communicate with UI - User Interface and, if loaded, a test program, tester hardware, and other user tools.

9. In Developer Studio, compile (build) the project. Since no user code has been added yet, no compile-time errors should occur (yet). If compile-time errors do occur, correct them before proceeding.

The remaining steps are to add code to the tool source file(s) to perform the desired actions when the tool executes. This can be as simple as “Hello World” (below) or as complex as time and creativity allow. Insert new source files to the tool as desired - except for the `#include` noted earlier it is just a C program (without a `main()`).

10. Using the example above, add the following to the file: `MyUserToolSrc.cpp`:

```
TOOL_BEGIN_BLOCK()( my_TBB_name ) {
   output("Hello World");
}
```

Compile the program again. No errors should occur. `TOOL_BEGIN_BLOCK()` is a Nextest macro which, if included in user tool code, executes automatically when the tool execution starts.

11. If, when compiling a user tool, the following link-time error occurs it means that at least one instance of the tool is currently running.

```
     Linking...
     LINK : fatal error LNK1168: cannot open Debug/
my_user_tool.exe for writing
     Error executing link.exe.
     Tool_code.exe - 1 error(s), 0 warning(s)
```

Terminating UI will terminate all instances of attached user tools. More details about terminating user tools are noted in Starting/Terminating User Tools.

7.4.3 Starting/Terminating User Tools

Note the following about starting and terminating a user tool:

- UI - User Interface must be running before a user tool can be started. If UI is not running an error will occur and the tool will not start.

- It is not required that a test program be loaded before starting a user tool. Of course, depending on the functions called from the user tool, proper tool operation may require that a test program to be running, and/or a tester hardware be available.
- Any number of user tools can be started/running at a time. Each will have a unique `site_num()` which can be used as a tool ID.
- User tool code will connect/disconnect with UI as needed.
- Unless explicitly coded by the test program (rare, and not good form), unloading a test program does not unload a user tool.
- Terminating UI - User Interface will terminate any connected user tools.
- To terminate a user tool in C-code the `testprogexit()` function must be used.
- It is possible to concurrently execute multiple instances of a given user tool. However this may have side effects. See Single Instance Code Example.

User tools can be starting using the following methods. The *path* and *file names* seen below are a continuation of those used above in Creating User Tools:

1. From a Windows shell command line:

   ```
   D:\tmp\MyUserTool\Debug\my_user_tool -t
   ```

   The **`-t`** option causes the executing program to connect to the UI - User Interface process and execute as a Magnum 1/2/2x user tool. The other methods used to invoke user tool code all use the **`-t`** option, sometimes transparently.

2. Executing a command file (.bat file).

   ```
   ui /tool=D:\tmp\MyUserTool\my_user_tool.exe /nologo
   ```

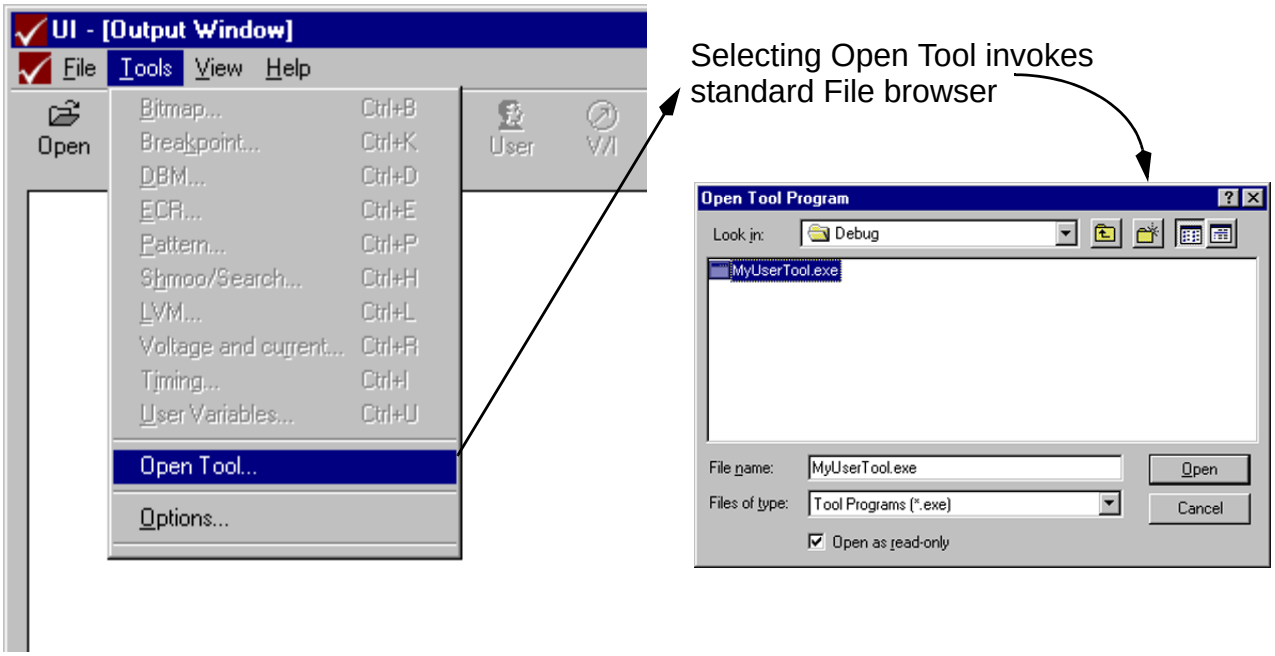
   Also see `ui_BatchFile`.

3. From C-code, using `remote_set()` and `ui_StartTool`

   ```
   CString tool_path = "D:\tmp\MyUserTool\Debug\my_user_tool";
   remote_set( "ui_StartTool", tool_path, -1, TRUE );
   ```

   Using this method, do NOT include the **`-t`** command line option.

4. Manually, using the UI - User Interface **`Tools: OpenTool`** menu selection



In the file browser, locate and select the executable file (`.exe`) of the user tool and click **`Open`** to start the tool. The **`-t`** command line option is handled automatically. Don't forget that executable files will typically be in the *Debug* folder of the user tool Project.

7.4.3.1 Single Instance Code Example

It is possible to concurrently execute multiple instances of a given user tool. However this has at least one ***potential negative side effect***, related to User Menus in UI.

It is common for user tools to add a menu to UI, to enable various tool functions via the menu. And, often the menu provides for terminating the tool. However, only one process can own a given user menu in UI, thus any attempts to add the same menu again are ignored. In other words, invoking a user tool multiple times will only add the menu to UI one time, with the first instance of the tool owning the menu. Terminating the first tool instance will terminate the menu, but any other instances of the same tool will continue to exist, with no menu displayed.

These other instances of the tool can be very annoying. The most common side effect (problem) occurs when trying to re-compile the tool. The following error message appears, indicating that the tool executable is running:

```
Error: Could not delete file "D:\tool_path\Debug\tool_code.exe" :
Access is denied.
```

When this occurs it is possible to terminate individual instances of a tool using the Windows Task Manager, or all tools will be terminated by terminating UI.

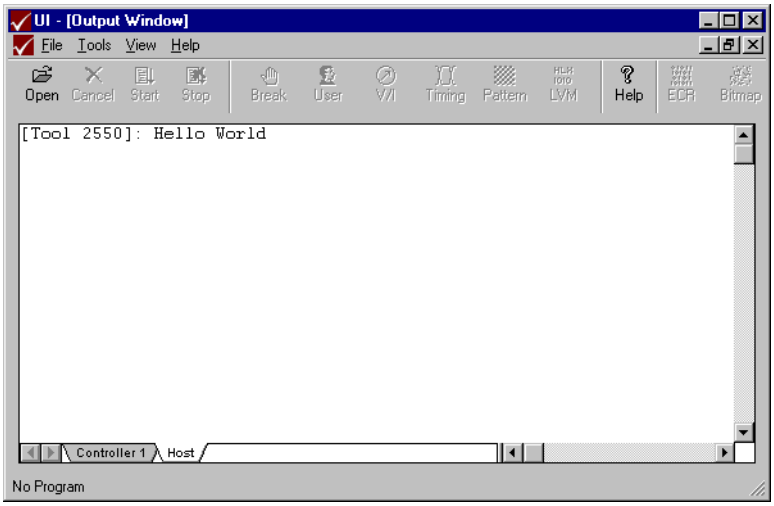
The code below can be called from the user tool code to prevent multiple instances of a given tool from being started:

```
// user-written routine used to ensure a second instance of a user
// tool won't start.
void EnsureSingleInstance(){
   char *name = __argv[ 0 ]; // Note: __argv[0] = tool name
   if ( FindWindow( "static", name ) )
      fatal ("[%s] already running. Terminate to restart.", name );
   else CreateWindow( "static", name, 0, 0, 0, 0, 0, 0, 0,  0, 0 );
}

// TOOL_BEGIN_BLOCK executes every time the tool is started.
TOOL_BEGIN_BLOCK( my_tool_BB ){
   EnsureSingleInstance();  // Prevent starting multiple tool
}                           // instances
```

7.4.4 User Tool Output Messages

When `output(), warning(), fatal(), vFormat()` are called from user tool code the output message will be seen in UI - User Interface's Host output window.

By default, each message will have a prepended prefix indicating that the message was from a user tool and the current `site_num()` of that tool. For example, the output below was generated from the example code on page 2495:



This prefix can be modified from user code using the ui_OutputFormat.

Note that this output message occurred even though a test program was not currently loaded, however, the same format would have been seen if a program was loaded. Also note that the `site_num()` of user tools is dynamic, and will change each time the tool is started. See Execution Context Functions.

7.4.5 User Tool Initialization

The following macros can be used to automatically execute user-written code within a user tool. Any of these macros which are included in the user tool code are called, in the order shown.

- `TOOL_CONFIGURATION()` // Executed only as the tool is started
- `TOOL_BEGIN_BLOCK()` // Executed only as the tool is started
- `INITIALIZATION_HOOK()` // Executed only as the tool is started
- `TOOL_END_BLOCK()` // Executed only as the tool is terminated

These macros are global in scope i.e. they can't be used within the body of a C function or another macro.

7.4.6 User Tool Functions

In general, User Tools communicate with the Host process, and/or the test program executinging in a Site process, and/or UI. See the Overview section of Binning for descriptions of these processes.

User Variables play a key role in this communication, as does the methodology covered in Host / Site / Tool Communication.

The `site_num()` and `OnTool()` functions, documented elsewhere, are also useful.

The remaining functions in this section address needs specific to User Tools.

7.4.6.1 get_all_tools()

The `get_all_tools()` function can be used in any of the Host, Site, or Tool processes to obtain a list of site number(s), one for each User Tools currently attached to UI - User Interface.

> Note: a given User Tool will not appear in the list until that tool has completely initialized, including execution of the tool's `TOOL_CONFIGURATION()`, `TOOL_BEGIN_BLOCK()` and `INITIALIZATION_HOOK()`, if defined.

Usage

```
int get_all_tools( IntArray *array );
```

where:

`array` is a pointer to an existing `IntArray` variable. This variable is modified by `get_all_tools()` to contain the `site_num()` of each User Tools currently connected to UI. See Note: above.

`get_all_tools()` returns the number of tools identified, which also represents the number of values in **`array`**.

Example

```
IntArray tools;
int count = get_all_tools( &tools );
```

```
output("There are %d user tools currently connected", count);
for (int i = 0; i < count; i++)
   output("Tool %d uses site_num => %d", i+1, tools[i] );
```

7.4.7 User Tool Example

The following example implements the following features in a user tool:

- User Menus in UI: two menus are added to UI.
- Via the menu, set `vil()` on all used signal pins using the `SiteMask()` option.
- Start two simple user dialogs. A button in each dialog causes the display in the other dialog to be modified.
- Confirmer dialogs are displayed when terminating either dialog or the user tool.
- The user tool's `site_num()` is obtained and printed
- Single Instance Code Example is implemented and called from the `TOOL_BEGIN_BLOCK()`.
- Except for the visual dialog components, the example code is contained in a single source file.

Note: because the visual dialog components cannot be included below, this code example will not compile. There are 4 dialogs. The 2 confirmer dialogs contain only the default IDCANCEL and IDOK buttons. Only `TOOLDIALOG1` and `TOOLDIALOG2` contain user-defined components: each has one user-defined push buttons, plus two user-defined text items. Images of these dialogs are included at the end of this section, and include the identifiers for each user-defined component.

```
#include "TestProgApp/public.h"
#include "resource.h" // For dialog use
//-------------------------------------------------------------
// Example of code which communicates with hardware. Most functions
// which program tester hardware support using SiteMask() as the
// first argument to the function. This directs the function to
// execute on all Sites enabled with a '1' in the SiteMask value.
// This example uses a mask of 0x1 i.e. write to Site 1 only.
```

```
// Body code converts string value to double value then executes
// vil() with that value, on sites specified by the bit-wise
// SiteMask()
CSTRING_VARIABLE( set_vil_volts, "", "uVar: set_vil_volts" ) {
   output("  set_vil_volts (string) => [%s]", set_vil_volts);
   // In this example, the value of 'set_vil_volts' is established
   // by the last portion of the first argument to menu_add() i.e.
   // everything after the last '/'. This is a string which, in this
   // example, must be converted to a double value to be
// subsequently used as arg-1 to the vil() function.
   double value = ( atof( set_vil_volts ) * (1 V));
   output("  set_vil_volts (double) => %1.3f", value );
   // Set vil on sites enabled via SiteMask() value
   vil( SiteMask( 0x1 ), value, builtin_UsedPins );
}
//-------------------------------------------------------------------
// Create a confirmer dialog to confirm termination of user dialog
// via user-defined IDCANCEL handler.
DIALOG( CONFIRM_EXIT_DIALOG ) {}
VOID_VARIABLE( cancel_handler_void, "" ) {
    if ( invoke( CONFIRM_EXIT_DIALOG ) == IDOK )
     focus( variable );
}
//-------------------------------------------------------------------
EXTERN_DIALOG( TOOLDIALOG1 ) // Forward
EXTERN_DIALOG( TOOLDIALOG2 ) // Forward
EXTERN_CSTRING_VARIABLE( dialog_msg2 ) // Forward

// These strings will be modified by clicking a button in one
// dialog and updating the value and display in the other dialog.
CSTRING_VARIABLE( dialog_msg1, "Original dialog-1 Message", "") {}
CSTRING_VARIABLE( dialog_msg2, "Original dialog-2 Message", "") {}

// Clicking this button in Tool Dialog 1 causes the value in the
// other dialog to change.
VOID_VARIABLE( ToolButton1, "" ) {
   static int times = 0;
   dialog_msg2 =vFormat ("Message changed from dialog-1 => %d
times", ++times);
   redisplay_modeless( TOOLDIALOG2 ); // Update value in dialog

}
```

```
// Clicking this button in Tool Dialog 2 causes the value in the
// other dialog to change.
VOID_VARIABLE( ToolButton2, "" ) {
   static int times = 0;
   dialog_msg1 = vFormat ("Message changed from dialog-2 => %d
times", ++times);
   redisplay_modeless( TOOLDIALOG1 ); // Update value in dialog
}
//-----------------------------------------------------------------
// Optionally used to set up initial conditions in the dialog as it
// is being created. Arg is required by prototype definition but
// not used
void set_initial_dialog_conditions( BOOL xxx) {
   output("  set_initial_dialog_conditions()");
}
//-----------------------------------------------------------------
// Two dialogs invoked as the tool starts, or from the my_Tools
// menu in UI. Each has a button and a message. Clicking the button
// in one dialog causes the message to change in the other dialog.
// The default IDCANCEL is intercepted and handled by code above,
// which presents a confirmer.
DIALOG( TOOLDIALOG1 ) {
   output("  ToolDialog1 from => %d", site_num() );
   TOPMOST( FALSE ) // Allows iconifying, etc.
   CONTROL( IDC_dialog_msg1, dialog_msg1 )
   IMMEDIATE_CONTROL( IDC_ToolButton1, ToolButton1 )
   CONTROL(IDCANCEL, cancel_handler_void);//Intercept Cancel event
   ONINITDIALOG( set_initial_dialog_conditions )
}
DIALOG( TOOLDIALOG2 ) {
   output("  ToolDialog2 from => %d", site_num() );
   TOPMOST( FALSE ) // Allows iconifying, etc.
   CONTROL( IDC_dialog_msg2, dialog_msg2 )
   IMMEDIATE_CONTROL( IDC_ToolButton2, ToolButton2 )
   CONTROL(IDCANCEL, cancel_handler_void);//Intercept Cancel event
   ONINITDIALOG( set_initial_dialog_conditions )
}
```

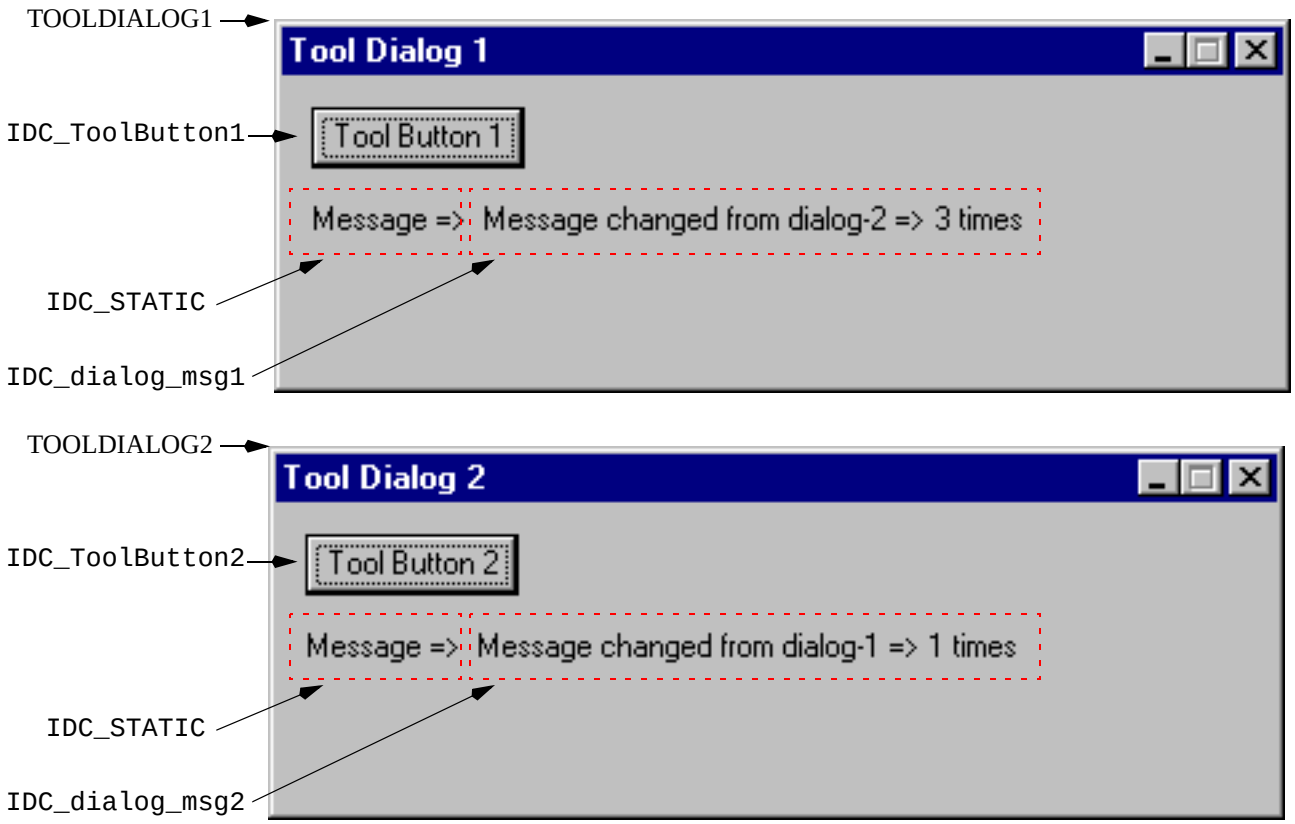
```
//-------------------------------------------------------------
// Used in my_Tool menu to restart one of the dialogs. This can be
// used if the dialogs, which are automatically started when the
// tool starts, are terminated.
VOID_VARIABLE( start_dialog1, "Start Dialog1" ) {
   output("  VOID_VARIABLE: start_dialog1 from => %d", site_num());
   run_modeless( TOOLDIALOG1 );
}
VOID_VARIABLE( start_dialog2, "Start Dialog2" ) {
   output("  VOID_VARIABLE: start_dialog2 from => %d", site_num());
   run_modeless( TOOLDIALOG2 );
}
//-------------------------------------------------------------
// Demonstrates how to automatically set a uVar = tool's site_num()
// value when a test program load completes. This example uses the
// uVar (tool_site_num) in the tool process but it could be in Site
// or Host too.
INT_VARIABLE( tool_site_num, -99, "tool_site_num" ) {
   output(" ui_ProgLoaded triggered uVar body code\\");
   output("using remote_set()");
   output("  Tool site_num => %d", tool_site_num );
}
// ui_ProgLoaded is automatically invoked by UI when a test program
// load completes. In this example, remote_set() is called to set
// the value of the tool_site_num uVar to the tool's site_num()
// value.
VOID_VARIABLE( ui_ProgLoaded, "") {
   output("  VOID_VARIABLE: ui_ProgLoaded from => %d", sender);
   // Using remote_set(), the uVar name as a String means this
   // doesn't get resolved until executed. Quoted name must be
   // identical with uVar name. uVar must exist at destination
   // site to be useful.
   remote_set( "tool_site_num", site_num(), site_num(), TRUE,
INFINITE );
}
//-------------------------------------------------------------
// Create a confirmer dialog to confirm termination of my_Tool.
// This also unconditionally terminates any dialogs created by the
// tool.
```

```
DIALOG( CONFIRM_EXIT_TOOL ){}
VOID_VARIABLE( ExitTool, "Exit Tool" ) {
   output( "  Terminating Tool" );
   if ( invoke( CONFIRM_EXIT_TOOL ) == IDOK )
      testprogexit();
}
//-------------------------------------------------------------------
// Body code executes when my_Tools/Msg-1 is selected in UI. Output
// messages appear in UI's Host window.
CSTRING_VARIABLE( print_tool_msg1, "", "uVar: print_tool_msg1" ) {
   output("  Msg-1");
}
// Body code executes when my_Tools/Msg-2 is selected in UI. Output
// messages appear in UI's Host window.
CSTRING_VARIABLE( print_tool_msg2, "", "uVar: print_tool_msg2" ) {
   output("  Msg-2");
}
//-------------------------------------------------------------------
// uVars used with "My Menu 2"
CSTRING_VARIABLE( menu_load_test_program, "", "Load program" ) {
   CString prog = get_open_file_name();
   remote_set( "ui_Open", prog, -1 );
}
CSTRING_VARIABLE( menu_start_test, "", "Issue Start Test" ) {
   remote_set( "ui_StartTest", "", -1 );
}
CSTRING_VARIABLE( menu_stop_testing, "", "Issue Stop Testing" ) {
   remote_set( "ui_StopTest", "", -1 );
}
CSTRING_VARIABLE( menu_close_program, "", "Close test program" ) {
   remote_set( "ui_Close", "", -1 );
}
CSTRING_VARIABLE( menu_terminate_UI, "", "Terminate UI" ) {
   remote_set( "ui_Exit", "", -1 );
}
//-------------------------------------------------------------------
// Routine to ensure tool won't start if currently running.
// This is trick, using methods not documented here.
```

```
void EnsureSingleInstance( ){
   char *name = __argv[ 0 ]; // __argv[0] = tool name
   if ( FindWindow( "static", name ) )
      fatal( "Only one copy of this tool can be run at a time" );
   else
      CreateWindow( "static", name, 0, 0, 0, 0, 0, 0, 0,  0, 0 );
}
//-------------------------------------------------------------
// These macros all execute automatically.
// Executes automatically, first
CONFIGURATION( C1 ){
   output("  CONFIGURATION:      site number => %d", site_num() );
}
// Executes automatically, after CONFIGURATION.
TOOL_CONFIGURATION( TC1 ){
   output("  TOOL_CONFIGURATION:  site number => %d", site_num() );
}
// Executes automatically, after TOOL_CONFIGURATION. Note that
// toolbar_add() could also be used below.
TOOL_BEGIN_BLOCK( TBB1 ){
    // Prevent multiple tool instances from starting
   EnsureSingleInstance(); // See Single Instance Code Example
   output("  TOOL_BEGIN_BLOCK:    site number => %d", site_num() );
   // Add a menu to UI
   menu_add( "My Menu 1/Msg-1",          print_tool_msg1 );
   menu_add( "My Menu 1/Msg-2",          print_tool_msg2 );
   menu_add( "My Menu 1/Set Vil/0 V",    set_vil_volts );
   menu_add( "My Menu 1/Set Vil/1 V",    set_vil_volts );
   menu_add( "My Menu 1/Start Dialog1",  start_dialog1 );
   menu_add( "My Menu 1/Start Dialog2",  start_dialog2 );
   menu_add( "My Menu 1/Exit Tool",      ExitTool );
   // Add another menu to UI
   menu_add( "My Menu 2/Load Program",   menu_load_test_program );
   menu_add( "My Menu 2/Start Test",     menu_start_test );
   menu_add( "My Menu 2/Stop Test",      menu_stop_testing );
   menu_add( "My Menu 2/Close Program",  menu_close_program );
   menu_add( "My Menu 2/Terminate UI",   menu_terminate_UI );
   // Start two dialogs, each running in its own thread.
```

```
   run_modeless( TOOLDIALOG1 );
   run_modeless( TOOLDIALOG2 );
}
// Executes automatically, as tool terminates.
TOOL_END_BLOCK( TE1 ){
   output("  TOOL_END_BLOCK:      site number => %d", site_num() );
}
// Executes automatically, after TOOL_BEGIN_BLOCK.
INITIALIZATION_HOOK( IH1 ) {
   output("  INITIALIZATION_HOOK: site number => %d", site_num() );
}
```

Example dialog images and component identifiers



7.4.8 ToolLauncher

ToolLauncher extends support for User Tools. The key features of ToolLauncher include:

- User-defined menu items are automatically added to UI's tool bar for each user tool registered with ToolLauncher. Registration is simple, see Tool Registration Requirements.
- Registered tools are not loaded or started until a menu item is selected. This reduces program load time, and memory is not consumed until the tool is actually started.
- When a tool is started, menu management can be mostly automatic, using cookbook code.
- Loading or unloading a test program can enable or disable user menu items, whether the tool is running or not.
- Beginning in software release h1.1.23, `ui_ShowTool` and `ui_HideTool` operate on User Tools which are started/managed by ToolLauncher. However, some user code is required, see ui_ShowTool / ui_HideTool Support.

7.4.8.1 Tool Registration Requirements

See ToolLauncher.

For a User Tools to be automatically detected (registered) by ToolLauncher requires the following:

- Each tool (i.e. *MyTool*) must contain both an executable file (i.e. *...\MyTool\Debug\MyTool.exe*) and a DLL (i.e. *...\MyTool\Debug\MyTool.dll*). Both files must be located in the same folder on disk and have the same root name. If the steps outlined in ToolLauncher DLL Setup are followed properly these two files are automatically set up.
- Each tool and associated DLL must define and share three Required Functions:
  - `setup_menus()`
  - `setup_toolbars()`
  - `site_loaded()`

7.4.8.2 Operation

See ToolLauncher.

This section documents the basic operation of ToolLauncher:

- ToolLauncher is automatically started by UI.
- When started, ToolLauncher registers any User Tools which meets the Tool Registration Requirements, by searching locations specified using the `DEFERRED_TOOL_PATH` environment variable (see Environmental Variables). For each location specified in `DEFERRED_TOOL_PATH`, that folder and all sub-folders are recursively checked for user tools meeting the defined Tool Registration Requirements.
- ToolLauncher loads the DLL for each registered tool and executes the `setup_menus()` function from the DLL. This adds user-defined menu item(s) to UI's tool bar. The DLL is then unloaded.

The following sequence occurs when a menu selection is made for a user tool registered with ToolLauncher:

- ToolLauncher removes all menu item(s) from UI's tool bar which were inserted by the DLL for the selected tool. This is required to allow tool code to subsequently manage its own menu items (more below).
- ToolLauncher starts the user tool, which executes normally (see User Tools).
- Code in the tool may (easily) restore the menu item(s), as desired. See `setup_menus()`.

The following sequence occurs when a user tool which is registered with ToolLauncher is terminated:

- ToolLauncher detects that the tool is terminated.
- The DLL for that tool is loaded and the `setup_menus()` function in the DLL is executed. This adds user-defined menu item(s) to UI's tool bar.
- The DLL is then unloaded.

7.4.8.3 Required Functions

See ToolLauncher.

As noted in Tool Registration Requirements, ToolLauncher requires that each user tool, and its associated DLL, define and share two functions:

- `setup_menus()` - a Nextest-defined callback function containing user-written code which defines how user tool menu item(s) are to be displayed in UI's tool bar. Executed by ToolLauncher to set up menu item(s) before a registered user tool is started or after the tool is terminated (see Operation). May also be executed by the user tool code restore menu items as the tool starts. This will typically occur via the tool's `TOOL_BEGIN_BLOCK`. See Example User Tool.
- `setup_toolbars()` - a Nextest-defined callback function containing user-written code which adds items to UI's toolbar.
- `site_loaded()` - a Nextest-defined callback function containing user-written code used to manage (enable, disable, etc.) existing user tool menu items. `site_loaded()` is executed to notify ToolLauncher when a test program is loaded or unloaded, on the Host or any Site(s). Then, for any registered user tool(s) which are not currently executing, the corresponding DLL is loaded, the `site_loaded()` function is executed, and the DLL is unloaded. `site_loaded()` may also be executed by user tool code as desired. See Example User Tool.

A single instance of these functions (i.e. a single source file) must be shared between the user tool code and the associated DLL (see MenuLayout.cpp). As noted in ToolLauncher DLL Setup, the tool's project workspace will contain two projects: the user tool and the DLL, both of which will include the MenuLayout.cpp file.

7.4.8.4 setup_menus()

See ToolLauncher.

See ToolLauncher, Required Functions.

Description

The `setup_menus()` function is a Nextest-defined callback, targeted to contain user-written code which defines menu items to be displayed in UI's tool bar. This is one of the ToolLauncher Required Functions.

As noted in Operation, `setup_menus()` is executed by ToolLauncher to add items to UI's Tool menu. This occurs when ToolLauncher first registers a user tool and any time a registered tool is terminated. When a given tool is started, ToolLauncher automatically removes all associated menu items from the Tool menu, to allow tool code to manage its own menu items.

Tool code can execute `setup_menus()`, typically from the tool's `TOOL_BEGIN_BLOCK`, to obtain the menu items originally displayed when ToolLauncher started.

Usage

The function prototype is defined by Nextest, but the body code is all user-written:

```
void setup_menus( menu_func mf,
                  BOOL deferred /* = FALSE */ )
  { ... user-written callback body code ... }
```

where:

`mf` is a pointer to a function to be called from within `setup_menus()`. When `setup_menus()` is executed from user tool code, the anticipated usage requires this to be the `menu_add()` function. See Example below and Example User Tool.

`deferred` will be TRUE when `setup_menus()` is executed by ToolLauncher. When `setup_menus()` is executed from user code **`deferred`** should not be specified, which results in FALSE being passed. **`deferred`** allows code within `setup_menus()` to react to how it was called.

Example

This example shows how `setup_menus()` will typically be called from user tool code to add menu item(s) the same as when ToolLauncher first started (which passes the address of `menu_add()` as the **`mf`** argument):

```
setup_menus( menu_add ); // Passing address of menu_add()
```

When executed, the body code of `setup_menus()` will execute the `menu_add()` function each place the **`mf`** token is used.

Also see Example User Tool.

7.4.8.5 setup_toolbars()

See ToolLauncher.

See ToolLauncher, Required Functions.

Description

The `setup_toolbars()` function is a Nextest-defined callback, containing user-written code which defines items to be displayed in UI's tool bar. This is one of the ToolLauncher Required Functions.

As noted in Operation, `setup_toolbars()` is executed by ToolLauncher to add items to UI's tool bar. This occurs when ToolLauncher first registers a user tool and any time a registered tool is terminated. When a given tool is started, ToolLauncher automatically removes all associated items from the tool bar, to allow tool code to manage its own tool bar items.

Tool code can execute `setup_toolbars()`, typically from the tool's `TOOL_BEGIN_BLOCK`, to obtain the tool bar items originally displayed when ToolLauncher started.

Usage

```
void setup_toolbars( toolbar_func tf, BOOL deferred = FALSE );
```

where:

`tf` is a pointer to a function to be called from within `setup_toolbars()`. When `setup_toolbars()` is executed from user tool code, the anticipated usage requires this to be the `toolbar_add()` function. See Example below and Example User Tool.

`deferred` will be `TRUE` when `setup_toolbars()` is executed by ToolLauncher. When `setup_toolbars()` is executed from user code **`deferred`** should not be specified, which results in FALSE being passed. **`deferred`** allows code within `setup_toolbars()` to react to how it was called.

Example

This example shows how `setup_toolbars()` will typically be called from user tool code to add toolbar item(s) the same as when ToolLauncher first started (which passes the address of `toolbar_add()` as the **`tf`** argument):

```
setup_toolbars( toolbar_add ); // Passing address of toolbar_add()
```

When executed, the body code of `setup_toolbars()` will execute the `toolbar_add()` function each place the **`tf`** token is used.

Also see Example User Tool.

7.4.8.6 site_loaded()

See ToolLauncher, Required Functions.

Description

The `site_loaded()` function is a Nextest-defined callback, containing user-written code used to manage (enable, disable, etc.) menu items for user tools registered with ToolLauncher. This is one of the ToolLauncher Required Functions.

Any time a test program is loaded or unloaded, on the Host or any Site(s), ToolLauncher is notified, and the following actions occur:

- For each user tool registered with ToolLauncher which is not currently loaded, the associated DLL is loaded.
- The `site_loaded()` function is executed from the DLL.
- The DLL is unloaded.

This allows menu items for user tools which are not running to be modified when a test program loads or unloads.

`site_loaded()` may also be executed from user tool code, as desired. This allows an executing user tool to use the same code as ToolLauncher, to manage/change menu items as test programs load/unload. See Example below and Example User Tool.

Usage

```
void site_loaded( int site,
                  BOOL loaded,
                  BOOL deferred /* = FALSE */ )
     { ... user-written callback body code ... }
```

where:

`site` indicates which site triggered the callback. Host = 0, Site-1 = 1, etc. This is automatic when `site_loaded()` is executed from ToolLauncher. If `site_loaded()` is called from user tool code the value of **`site`** is determined by that code.

`loaded` is TRUE if `site_loaded()` was triggered because a test program was loaded or FALSE if unloaded. This is automatic when `site_loaded()` is executed from ToolLauncher. If `site_loaded()` is executed from user code the value of **`loaded`** is determined by that code.

deferred is TRUE if site_loaded() is called by ToolLauncher, indicating that the associated user tool is not currently executing (i.e. execution is deferred). When site_loaded() is executed from user code **deferred** should not be specified, which results in FALSE being passed. **deferred** allows code within site_loaded() to react to how it was called.

Example

This example shows how site_loaded() might be used in user tool code:

```
VOID_VARIABLE( ui_ProgLoaded, ""){
   site_loaded( 0, TRUE );
}
VOID_VARIABLE( ui_ProgUnloaded, ""){
   site_loaded( 0, FALSE );
}
INT_VARIABLE( ui_SiteLoaded, 0, ""){
   site_loaded( ui_SiteLoaded, TRUE );
}
INT_VARIABLE( ui_SiteUnloaded, 0, "" ){
   site_loaded( ui_SiteUnloaded, FALSE );
}
```

The effect of this code is to cause the same results as when site_loaded() is executed by ToolLauncher.

Also see Example User Tool.

7.4.8.7 ui_ShowTool / ui_HideTool Support

See ToolLauncher, User Tools, Required Functions.

As indicated above, beginning in software release h1.1.23, ui_ShowTool and ui_HideTool operate on User Tools which are started/managed by ToolLauncher.

However, in order for operation to be consistent with that seen with UI's tools a bit of user code is required, as follows:

```
BOOL_VARIABLE( ui_Show, TRUE, "" ){
   if ( ui_Show )
     run_modeless( myDialog );
```

```
    else
      kill_modeless( myDialog ); // Dismiss the tool dialog
    // Or, use testprogexit() to terminate the tool process
}
```

Note the following:

- The ui_Show user variable, and related code, must be added to the user tool code. It should NOT be added to the MenuLayout.cpp file required by ToolLauncher.
- The purpose of ui_Show is to allow UI to invoked the tool if/when ui_ShowTool is applied to the tool and to hide the tool when ui_HideTool is applied to the tool. The code in the example above performs these tasks.
- If the code above does not exist, UI will not have the mechanism needed for ui_ShowTool to start the tool and and ui_HideTool will kill the tool process rather than just dismiss the dialog; the latter operation is described in the ui_ShowTool documentation.

7.4.8.8 MenuLayout.cpp

See ToolLauncher.

Note: the *MenuLayout.cpp* file included here is an example implementation (recommended), demonstrating key features of the ToolLauncher Required Functions source file. Note that ToolLauncher DLL Setup assumes the *MenuLayout.cpp* file name is used.

As noted in Tool Registration Requirements, ToolLauncher requires that the Required Functions be shared between the user tool code and the associated DLL. The example below is contained in a single source file named *MenuLayout.cpp,* and contains *only* the Required Functions (highly recommended):

```
// This code is shared by the user tool and associated DLL
#include "deferred_tool.h" // Replaces "tester.h"

void setup_menus( menu_func mf, BOOL deferred ) { // setup_menus()
   mf( "MyTools/Func1", "MyFunc1", "C-a", FALSE );
   mf( "MyTools/Func2", "MyFunc2", "C-b", FALSE );
   mf( "MyTools/MyDialog", "StartMyDialog", "C-c");
}
```

```
void setup_toolbars( toolbar_func tf, BOOL deferred ){
// Set up toolbar items here. See setup_toolbars()
}
void site_loaded( int site, BOOL loaded, BOOL deferred ) {
   output( "  site_loaded() reports program %s on Site => %d %s",
      loaded ? "Loaded" : "UNloaded", site );
   // Execute the following only when a test program is loaded or
   // unloaded on the Host. This will only occur after all sites
   // are loaded or unloaded.
   if ( site == 0 ) {
      menu_enable( "MyTools/Func1", loaded );
      menu_enable( "MyTools/Func2", loaded );
   }
}
```

Note the following:

- A different `#include` must be used in any source file(s) which reference `setup_menus()`, `setup_toolbars()` or `site_loaded()`:

  ```
  #include "deferred_tool.h"
  ```

  The `deferred_tool.h` file includes and thus replaces:

  ```
  #include "tester.h"
  ```

  Note: collisions with user additions to the *tester.h* file should be rare, but are possible, and must be resolved by moving the problem definitions outside the Nextest include file structure.

- The *MenuLayout.cpp* file contains *only* the ToolLauncher Required Functions: `setup_menus()`, `setup_toolbars()` and `site_loaded()`. The prototypes of both functions are defined by Nextest, and cannot be changed.

- In `setup_menus()`, the `mf` function is called once to define each user tool menu item; the arguments define the menu options. ToolLauncher will add these menu items as outlined in Operation. The following code from the `setup_menus()` example above defines one menu entry:

  ```
  mf( "MyTools/Func1", "MyFunc1", "C-a", FALSE );
  ```

  where:

  "`MyTools/Func1`" defines the menu to modify or create (`MyTools`) and the menu item text to be added (`Func1`)

”MyFunc1” identifies a user variable. The body code of this user variable will be executed when the menu item is selected. Note that the user variable name must be "quoted". This is because these user variable(s) are not defined (and should not be defined) within the scope of `setup_menus()`.

“C-a“ defines a keyboard shortcut (control-A) which will activate the menu item.

FALSE specifies that the menu item will be disabled. If this argument is not specified the menu item is enabled.

Note: user menus are documented in detail in User Menus in UI.

- User tool code may also call `setup_toolbars()` to insert items into UI’s toolbar.
- User tool code may also call `setup_menus()` to execute the same code as ToolLauncher. From user code, the `mf` function should be the address of `menu_add()`. As the example shows, multiple menu items can be added by calling `mf()` multiple times.
- The example `site_loaded()` code above does the following:
  - Always outputs a message showing the values of the passed arguments.
  - Enables two menu items when a test program is loaded on the Host.
  - Disables two menu items when a test program is unloaded on the Host.

The Example User Tool uses this same code.

7.4.8.9 ToolLauncher DLL Setup

See ToolLauncher.

As noted in Tool Registration Requirements, the Required Functions must be shared between the user tool code and the associated DLL, and the DLL and tool executable files must be located together. How this is managed is documented here.

User tools to be supported by ToolLauncher will contain two projects:

- The user tool project
- The DLL project

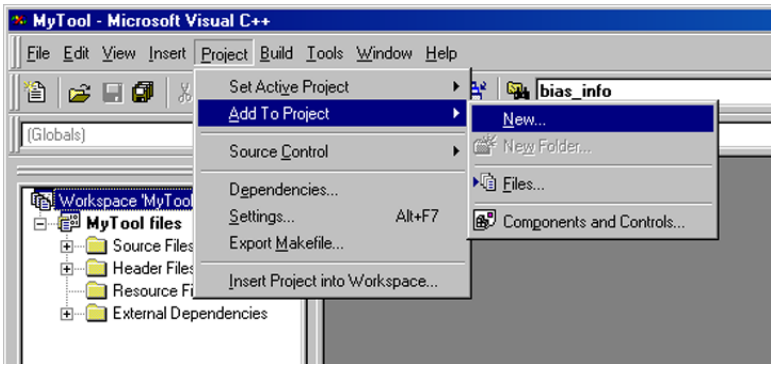
Both projects will contain the MenuLayout.cpp file.

Before dealing with the DLL, the user tool project is first set up normally, as documented in the *User Tools*. Create the MenuLayout.cpp file and add it to the user tool project. Build the tool and correct any compile-time errors.

Note: the procedures below all assume the user tool is named *MyTool*, and that the MenuLayout.cpp file name is used.

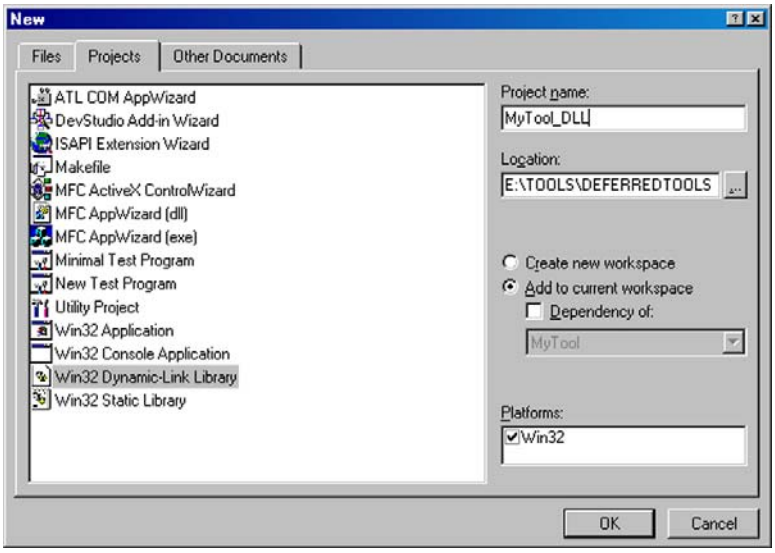
To add the DLL, perform the following:

1. In the same workspace as the user tool, add the DLL project:
   - In the *File View* window, select the workspace, then select **Project -> Add to Project -> New...**:

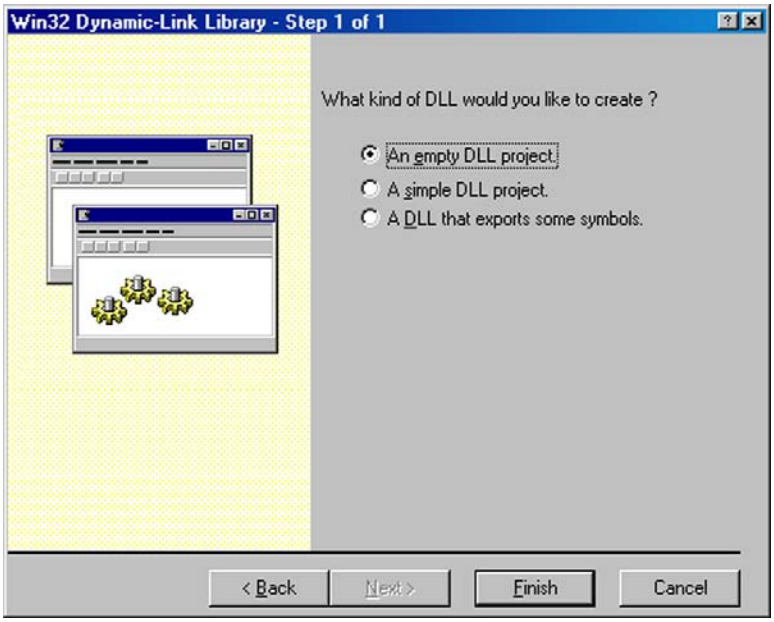


   - In the *New* dialog, select the **Projects** tab, then select **Win32 Dynamic Link Library**.
   - In the **Project name** field, enter the project name for the DLL. The name specified here does not matter - the setup done in step-5 causes the actuall DLL to be renamed when it is copied to the final location. However, do make this name clearly indicate the project is a DLL. Don't change any other fields.

- Click OK:



- In the Wizard dialog which appears next, the default option (An empty DLL project) is correct. Click **Finish**.

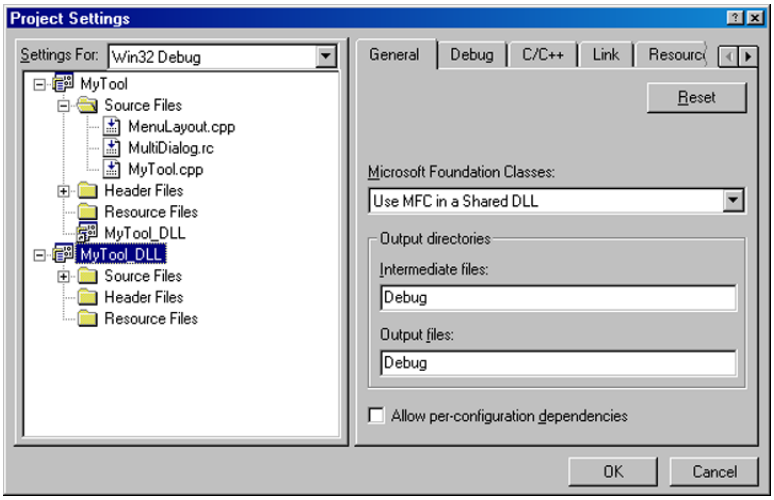


- Terminate the New Project Information dialog which appears next.

2. Set the DLL project to use MFC as a Shared DLL:

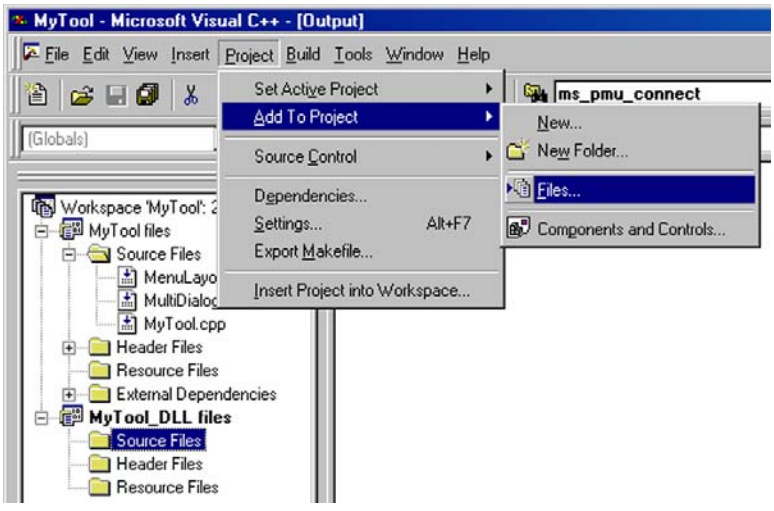
   - In the *File View* window, select the DLL project.
   - Select **Project -> Settings...**

- In the Project Settings dialog, ensure the DLL project is selected in the left panel.
- In the Project Settings dialog, locate and select the **`General`** tab
- In the Microsoft Foundation Classes pull-down menu select:
  `Use MFC in a Shared DLL`
- Click OK



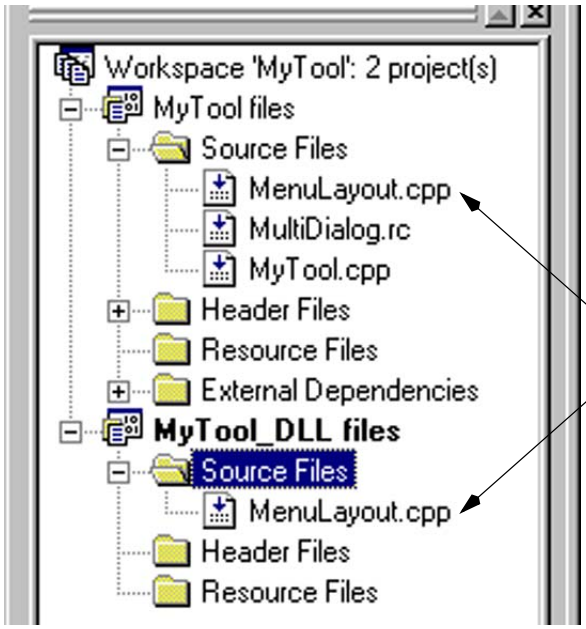
3. Add the MenuLayout.cpp file containing the Required Functions to the DLL project:

- In the *File View* window, select the **`Source Files`** folder from newly added DLL project (not the tool project).
- Select **`Project -> Add to Project -> Files...`**



- In the file browser, select the MenuLayout.cpp file and click OK.

- Confirm the MenuLayout.cpp appears in both the tool project and the DLL project:



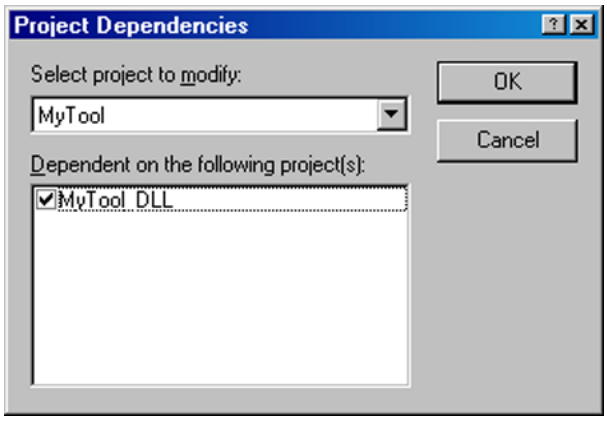
MenuLayout.cpp shared with both the user tool andthe DLL projects.

4. Set up a dependency between the user tool project and the DLL:

   This step is optional. Doing this ensures that invoking **`Build -> Rebuild All`** will *always* compile both projects, in the proper order. If this step is skipped, the user is responsible for manually selecting the project to compile, and the tool *\Debug* folder must exist before custom build step (below) for the DLL will execute.

   - Select **`Project -> Dependencies...`**
   - In the Project Dependencies dialog make sure the tool project is dependent on the DLL project.

- Click OK:

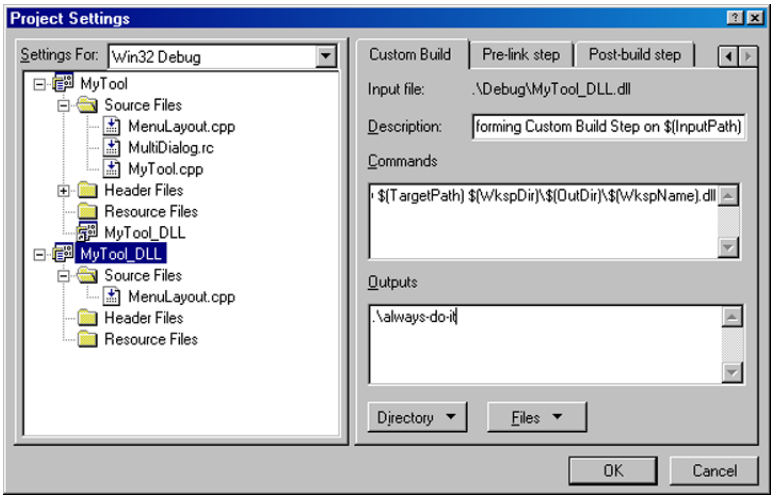


5. Set up a custom build step for the DLL project:

   This will copy the DLL's *.exe* file to the tool's *\Debug* folder so that it is in the same location as the tool's *.exe* file (as noted in Tool Registration Requirements). The *.dll* file is also renamed to match the *.exe* file name.

   - In the *File View* window, select the DLL project.
   - Select **Project -> Settings...**.
   - In the Project Settings dialog, ensure the DLL project is selected in the left panel.
   - In the Project Settings dialog, locate and select the **Custom Build** tab.
   - In the **Commands** window enter the following command, exactly as shown (use copy/paste but don't get the end-of-line):
     `copy "$(TargetPath)" "$(WkspDir)\$(OutDir)\$(WkspName).dll"`
   - In the **Outputs** window enter the following command, exactly as shown(use copy/paste but don't get the end-of-line):
     `.\always-do-it`

- Click OK.



6. Set the user tool project (not the DLL project) as the active project:
   - In the *File View* window, select the user tool project.
   - Using the right-mouse select **`Set as Active Project`**
   - Confirm the user tool project is selected.
7. **`Select Build -> Rebuild All`** to compile both the tool project and the DLL.

7.4.8.10 Example User Tool

See ToolLauncher.

Most of the important details about this example are discussed previously in earlier sections. Other details can be found in the User Tools.

This example adds one menu (*MyTools*) and 3 menu items (*Func1*, *Func2*, *MyDialog*) to UI's tool bar:

Note the following

- The *Func1* and *Func2* items are initially disabled (greyed out). The *MyDialog* item is enabled.
- *Func1* and *Func2* will be enabled any time a test program is loaded, and disabled any time a test program is not loaded. This is true whether the tool is loaded or not.
- In this example, *Func1* and *Func2* only output a message when the menu item is selected. When *MyDialog* is selected the dialog (see MultiDialog.rc File below) starts (its display may be minimized). The dialog does nothing interesting.

This example is contained in the following files:

- MyTool.cpp
- MenuLayout.cpp File
- MultiDialog.rc File

Note that the information and images seen in ToolLauncher DLL Setup was obtained using example below.

MyTool.cpp

```
// This is user tool code.
#include "deferred_tool.h" // Replaces "tester.h"

EXTERN_DIALOG( MyDialog ) // Forward

CSTRING_VARIABLE( MyFunc1, "MyFunc1", "") {
   output( "  Executing %s from site=>%d",
      resource_name( variable ), sender );
}
```

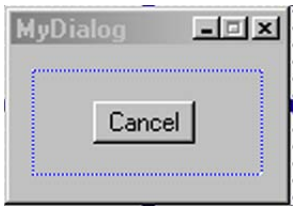
```
CSTRING_VARIABLE( MyFunc2, "MyFunc2", "") {
   output( "  Executing %s from site=>%d",
      resource_name( variable ), sender );
}
CSTRING_VARIABLE( StartMyDialog, "StartMyDialog", "") {
   output(" Executing %s from site=>%d",
      resource_name( variable ), sender);
   run_modeless( MyDialog ); // Start my tool dialog
}
DIALOG( MyDialog ) {
   TOPMOST( FALSE ) // Allows iconifying, etc.
}
// If the tool is running, make it enable menu items the same as
// ToolLauncher would if the tool was not running.
VOID_VARIABLE( ui_ProgLoaded, ""){ site_loaded( 0, TRUE ); }
VOID_VARIABLE( ui_ProgUnloaded, ""){ site_loaded( 0, FALSE ); }
INT_VARIABLE( ui_SiteLoaded, 0, ""){
   site_loaded( ui_SiteLoaded, TRUE );
}
INT_VARIABLE( ui_SiteUnloaded, 0, "" ){
   site_loaded( ui_SiteUnloaded, FALSE );
}
// When the tool starts, set up the menu and/or toolbar items the
// same as ToolLauncher did.
TOOL_BEGIN_BLOCK( TBB1 ){
   setup_menus( menu_add );          // Optional
   setup_toolbars( toolbar_add );    // Optional

}
```

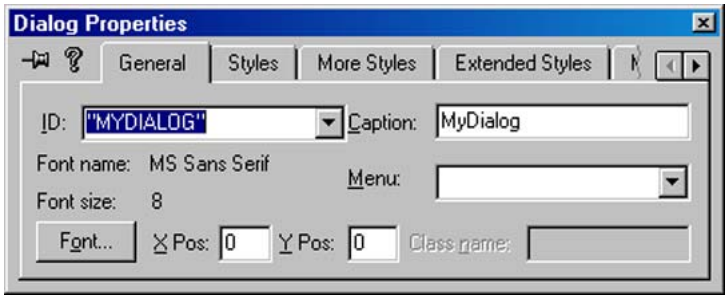
MenuLayout.cpp File

Use the code from MenuLayout.cpp.

MultiDialog.rc File

The dialog below, containing only the Cancel button, is one of the tools (MyDialog) started from this code:





7.5 User Dialogs

This section includes:

7.5.1 Overview

Terminology Used

The terminology used in this section is a mix of formal terms used by Microsoft Developer Studio and more common or generic terms used by the rest of the world. These latter terms are only used when the formal terms are obscure and/or risk confusion.

User Dialog	A graphic dialog and related C-code created by the user.
Dialog Component	A sub-component of a user dialog: a button, text box, radio button, etc. This term used here in place of “Dialog Resource” to aid clarity.
Dialog Resource	The formal term used in Developer Studio (tools, etc.) for a Dialog Component.
User Variable	A Nextest-created software object consisting of a variable, with optional body code which can be selectively executed. User variables are a key component in making user dialogs easy to use. See User Variables.
MSDS	Microsoft Developer Studio
MFC	Microsoft Foundation Class Libraries. This is the underlying library of functions used to support the complete set of Visual graphic capabilities in Visual C++. In most cases, user-created dialogs do NOT directly use MFC library calls. See Supported Dialog Components

7.5.2 Supported Dialog Components

The Microsoft Visual C++ Developer Studio (MSDS) supports a broad range of predefined graphic dialog components, including buttons, text boxes, radio buttons, etc. To fully utilize these resources requires a broad knowledge of the Microsoft Foundation Class Libraries (MFC) included with MSDS. This is not a trivial learning experience and is not supported by Nextest.

Instead, the Nextest software provides a set of macroS and functions which, when combined with User Variables, make the most commonly used dialog components accessible to the typical user. However, by necessity, the scope of support provided by Nextest software is a subset of the full capabilities available via the MFC libraries.

The various combinations of user variable vs. supported dialog components are:

User Variable Type	Supported Control Types
BOOL_VARIABLE	Checkbox
CSTRING_VARIABLE	Edit Box, Static Text, Bitmap[4]
DOUBLE_VARIABLE	Edit Box, Static Text, Progress[3], Bitmap[3,4]
DWORD_VARIABLE	Edit Box, Static Text, Progress[3], Bitmap[3,4]
FLOAT_VARIABLE	Edit Box, Static Text, Progress[3], Bitmap[3,4]
INT_VARIABLE	Edit Box, Static Text, Progress[3], Bitmap[3,4]
INT64_VARIABLE	Edit Box, Static Text, Progress[3], Bitmap[3,4]
ONEOF_VARIABLE	ListBox, Dropdown ComboBox, Drop List ComboBox, Tab, Static Text, RadioButton[2], Bitmap[3,4]
VOID_VARIABLE	Button, Bitmap[4], Group Box[2]
Notes 1) No immediate mode 2) Maverick-I/-II Software release v1.26.21 or later 3) Maverick-I/-II Software release v2.4.7 or later. All Magnum 1/2/2x releases. 4) See Bitmap Usage.	

Nextest software does not provide any support for the following dialog components:

- Horizontal or Vertical Scroll Bars, other than as automatically supported by the components above.
- Animate
- Tab Control
- Tree Control
- List Control
- Hot Key
- Slider
- Spin
- Custom Control

- Rich Edit
- Date/Time Picker
- Month Calendar
- IP Address
- Extended Combo Box

7.5.3 Creating a User Dialog

Definition

A user dialog consists of two part

- Creating the Dialog C-code
- Creating the Dialog Graphic

The C-code is created/edited using the same methods used for the other test program source code. The test program created using the Program Wizard contains a number of simple example dialogs, with supporting C-code contained in the file *dialogs.cpp*.

The dialog's graphic components are edited visually, using different Developer Studio tools.

It does not matter whether the C-code or graphic components are created first, however error free compiling may not be possible until both parts are completed.

Note: it is recommended that user dialogs be created incrementally, by adding only a few features at a time, and compiling, and testing often. Occasionally, when things go wrong, the only course of recovery is to delete things, both code and graphic components, until the problem is corrected.

7.5.3.1 Creating the Dialog C-code

Nextest software provides a set of Test System Macro and functions which, when combined with User Variables, make the most commonly used dialog components accessible to the typical user. These are listed below, and elsewhere in this section of the manual.

Note: Nextest does NOT provide any level of direct support for the Microsoft Foundation Class Libraries (MFC). All Nextest support for user dialogs is provided through the macros and functions documented in this manual.

The Program Wizard stores the example user dialog C-code in the file *dialogs.cpp*, and this file name will be used here when referring to dialog source code. However, the file name is totally up to the user.

The `DIALOG()` macro is used to create and name a user dialog. Multiple dialogs can be created in the test program, each with a unique name. The remainder of the macros/functions below are used within the `DIALOG()` macro body-code.

The `CONTROL()` macro is used in the body-code of the `DIALOG()` macro to used to link a *non-immediate* dialog resource to one `BOOL_VARIABLE`, `DOUBLE_VARIABLE`, `DWORD_VARIABLE`, `FLOAT_VARIABLE`, `INT_VARIABLE`, `INT64_VARIABLE`, `UINT64_VARIABLE` or `CSTRING_VARIABLE` User-defined User Variables. The term non-immediate means that activating that dialog component does not cause an (immediate) execution of the associated user variable body code i.e. any selection or value change made in the dialog component must be explicitly *read*, using `update_variable()` or `update_variables()`. Using the `CONTROL()` macro defaults to displaying numerical values in decimal. However, when entering numerical values in the dialog component either decimal, or hex, values can be entered, using the appropriate syntax (*0x* prefix for hex).

The `HEX_CONTROL()` macro is similar to `CONTROL` except that it defaults to displaying hexadecimal numerical values. Both decimal and hex values can be entered using the appropriate syntax.

The `IMMEDIATE_CONTROL()` macro is used in the body-code of the `DIALOG()` macro to link an immediate dialog resource to a `VOID_VARIABLE`, `BOOL_VARIABLE`, or `ONEOF_VARIABLE` User-defined User Variables. The term *immediate* means that invoking that dialog component causes an immediate execution of the associated user variable body code.

The `TOPMOST()` macro is used in the body-code of the `DIALOG()` macro to determine whether the dialog will remain on-top of other UI dialogs (TRUE) or will allow other dialogs to be on top of it (FALSE).

The `ONINITDIALOG()` macro is used in the body-code of the `DIALOG()` macro to specify a function which will be executed when the dialog is first created. The common application is to set up the dialog's initial conditions. Function required prototype is:

```
void funcID( BOOL created )
```

where the name of the function is user-defined, and is the argument passed to the `ONINITDIALOG()`. During dialog creation the specified function is executed twice, once before any dialog resources are created (`created = FALSE`) and once after all dialog resources are created (`created = TRUE`). In situations where the initial values to be displayed in the dialog are not the default values of the User-defined User Variables tied to the dialog components this function can be used to establish the desired values. See User Tool Example.

The `GRAPHIC()` macro is used in the body-code of the `DIALOG()` macro to ???.

The `hex_display()` function is used in the body-code of the `DIALOG()` macro to modify the numerical base used to display integer values. More below.

Usage

```
DIALOG( dialog_name ) { body_code }
CONTROL( control_id, uVar )
HEX_CONTROL( control_id, uVar )
IMMEDIATE_CONTROL( control_id, uVar )
TOPMOST( mode )
ONINITDIALOG( func )
GRAPHIC( id, func )
hex_display( uVar, TRUE )
```

where:

`dialog_name` is used to link the main graphic dialog to user-written C-code, and to invoke the dialog. The name must *exactly* match the Dialog Properties ID, which can be viewed and edited using the *Dialog Properties* display. See Creating the Dialog Graphic.

`control_id` represents the resource ID of the dialog component (button, list box, etc.) being associated with the specified user variable, **`uVar`**. In the *MSDS Resource View*, display the target dialog and double-click on a dialog component (resource) to display it's properties, including it's ID. Except for the predefined `IDCANCEL` and `IDOK` resource IDs, a given dialog component is not programmatically useful unless it is tied to a user variable.

`uVar` is the name of one user-defined User-defined User Variables.

`mode` specifies the desired numerical display mode of the dialog component tied to the user variable **`uVar`**. Specifying TRUE = hexadecimal. FALSE = decimal. This can be used to

override the default radix setup using the `CONTROL()` and `HEX_CONTROL()` macros. This function only applies when:

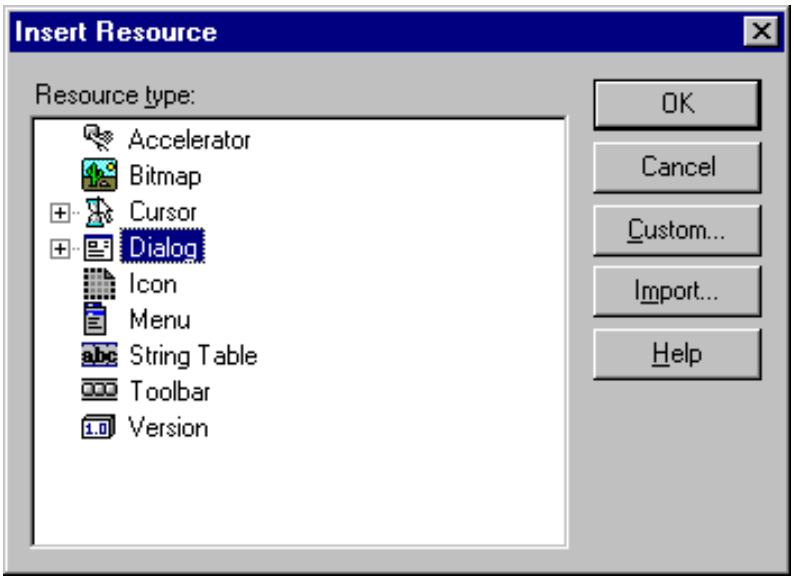
- The user variable is an `INT_VARIABLE`, `DWORD_VARIABLE` or `INT64_VARIABLE`
- The associated dialog component displays numerical values: text box, edit box, etc.

Note that the user may enter values in the dialog component using hex or decimal regardless of the `hex_display()` mode setting. But, if the value displayed in the dialog is updated from C-code (`update_variable()` or `update_variables()`) the new value displayed will be in the default radix as modified by `hex_display()`. The prefix 0x is used to enter hex values.

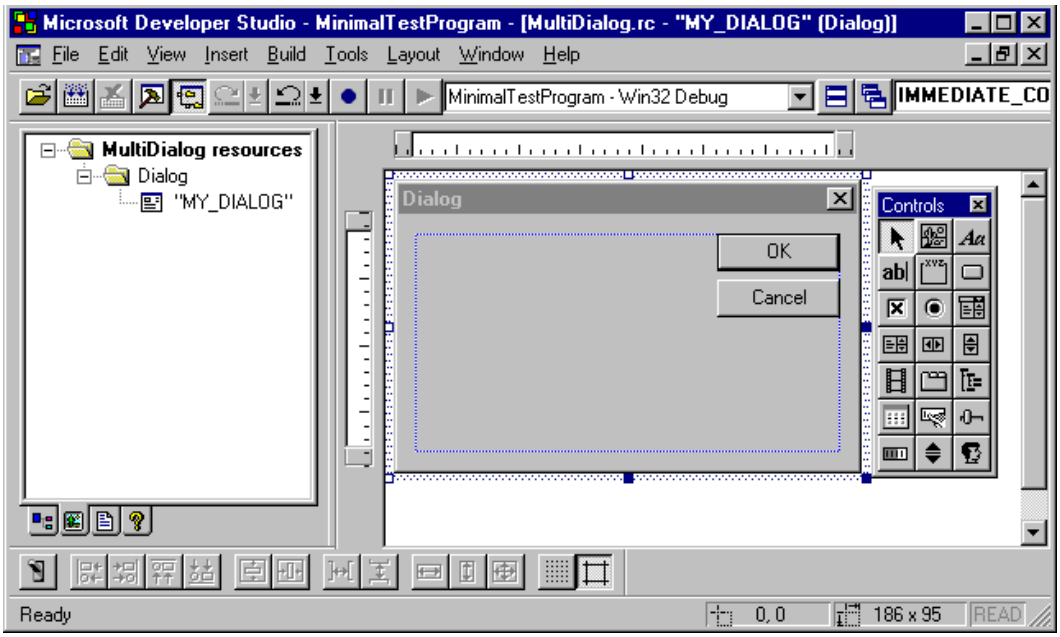
7.5.3.2 Creating the Dialog Graphic

The following steps outline how to create a new dialog in Developer Studio. The examples below reflect a test program which contains no dialogs. If the program already contains user dialogs the images below will be different, but the process is the same.

1. Open the *Project Workspace* of the target test program.
2. Select **`Insert: Resource.`**
3. In the *Insert Resource* display, left click on Dialog, and click OK



4. In the *Project Workspace* window, select the *ResourceView* tab
5. Click the **+** to open the Dialog folder and note the new dialog resource appears in the list of dialogs. Double click on the dialog icon to display the dialog resource for editing.



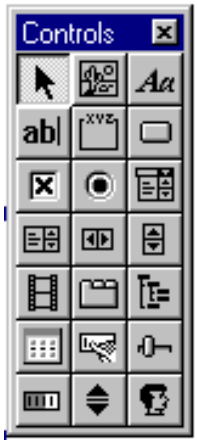
6. Double click on the background of the dialog, and enter the name of the dialog in the ID field. This name must be in *DOUBLE QUOTES*. It is important that this name exactly match the name used in the corresponding `DIALOG()` macro.

This ID must exactly match the dialog name used in the C-code. It must be double quoted here (unlike the ID for other dialog components).

7. Click anywhere outside the Dialog Properties box to cause it to close. No explicit **Save** is required (or possible).
8. The dialog properties can be edited at any time, using this Dialog Properties box. Click on the  icon for an explanation of each property seen under each tab.

7.5.3.3 Adding Dialog Components to the Dialog

Dialog components are added to the main dialog using drag/drop methods. The source of dialog components is the Controls Tool Bar.



This tool bar may be Hidden, by you or a prior user. If it is not visible, select **View: Toolbars...** and make sure the *Controls* option is checked. The Controls Tool Bar may be positioned anywhere in the *Developer Studio* window.

Example

To add a Combo Box, for example, to the dialog:

1. In the Controls Tool Bar, locate the Edit Box icon 
2. Left click-and-drag into the main dialog display.
3. Resize and position the component as desired. This can also be changed at any time.

4. Double-click on the Edit Box component to display the Edit Box Properties dialog.

5. Edit the ID to a meaningful name. In this example, the name was changed to IDC_options_list. Note that the IDC_ prefix is a convention, allowing easy identification of dialog component IDs from other program identifiers. This ID (name) will be used as argument-2 to one of `CONTROL()`, `IMMEDIATE_CONTROL()`, etc. macros in the C-code. The ID (name) for dialog components is NOT quoted (unlike the ID for the parent dialog).

6. Position and size the Edit Box as desired. Use the *Dialog Tool Box* controls as desired.

This tool bar may be hidden, by you or a previous user. If it is not visible, select **`View: Toolbars...`** and make sure the *Dialog* option is checked. The Dialog Tool Bar may be positioned anywhere in the *Developer Studio* window.

7.5.3.4 IDCANCEL and IDOK

By default, each new dialog will automatically contain a Cancel and OK button. The ID property of the OK button is `IDOK`. The ID property of the Cancel button is `IDCANCEL`. Both `IDOK` and `IDCANCEL` have predefined functionality i.e. no user C-code is needed to support these components.

However, it is possible for user-written C-code to replace the built-in functionality. This is done by defining an `IMMEDIATE_CONTROL()` with `IDCANCEL` or `IDOK` as the first argument. The second argument must be the name of an existing `VOID_VARIABLE`. It is the

body code of the VOID_VARIABLE which will execute when the OK or Cancel button is clicked. It is this body code which takes ***all*** responsibility for terminating the dialog (or not).

The `focus()` function is used to set the focus to a specific resource in a dialog. It is also used to terminate a dialog. Note the following:

- When used to terminate a dialog, the syntax is fixed regardless of the dialog name, ID, etc. Use the following in the body-code of the user variable:

  ```
  focus(variable); // Literally "variable" not the uVar name
  ```

  When the variable is a VOID_VARIABLE it must be tied to IDCANCEL (see IDCANCEL and IDOK).
- When used to move the focus to a specific dialog component the name of the User Variables linked to the component is specified i.e. `focus(uVar_name)`.
- When `focus(uVar_name)` is called from the body-code of a user variable execution immediately exits the body-code, AND any other body-code in the current execution hierarchy. In other words, if the body code of *uVar1* invokes the body code of *uVar2* which invokes the body code if *uVar3*, if `focus(any_uVar_name)` is called in *uVar3* no more body code from any of the three user variables will be executed.
- When it is desired to undefine the focus use `focus(0);`

Usage

```
void focus( VariableProxy v );
```

where:

v identifies which variable will receive the focus, see Description for more details.

Example

```
VOID_VARIABLE( my_CANCEL_var, "Dismiss current dialog" ) {
   if ( update_values( current_dialog() )) { // Optional
      focus( variable );
   }
}
```

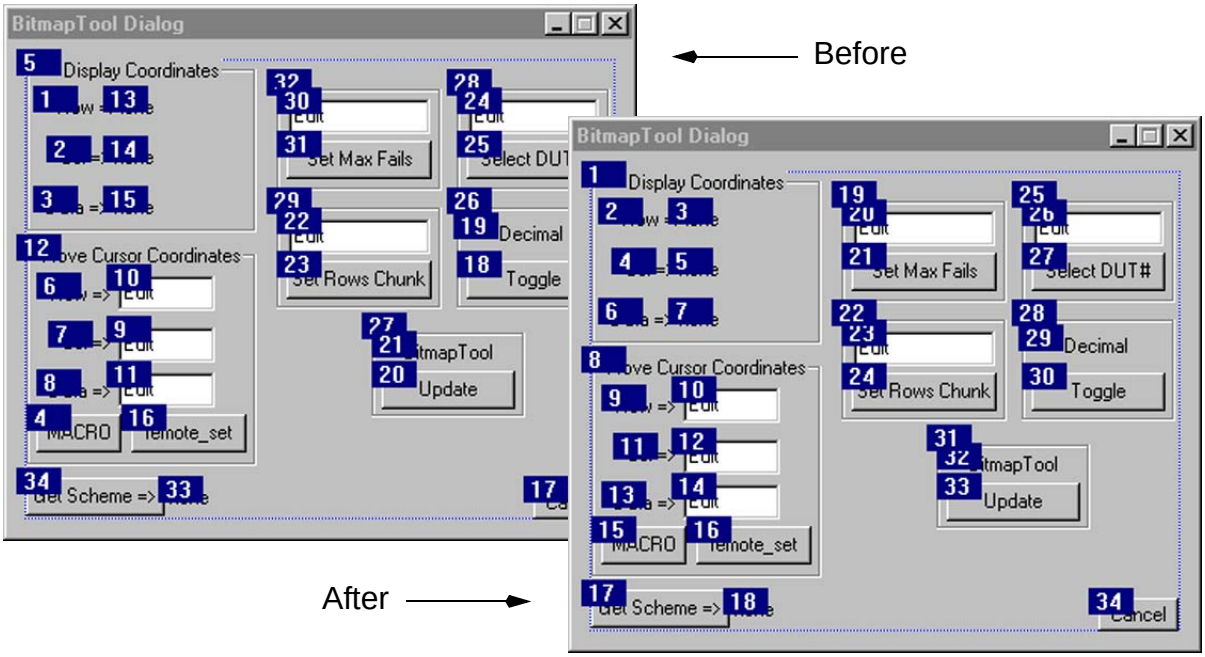
7.5.4 Setting Tab Order

Description

In use, it is possible to move the cursor between components of User Dialogs using the tab key. This section documents how to configure how the cursor moves when the tab key is used.

In most cases a will configured tab order is only a convenience to the end user. However, a contiguous tab order is required when using Radio Buttons and ONEOF User Variables.

In the images below, the left dialog shows the order tab movement would occur before any invervention. The order is somewhat random, likely base on the order the dialog components were added. The right image shows how the order was modified.



Usage

The process begins by displaying the dialog in Developer Studio. Type control-D (^d) to display the existing tab order.

To set the order starting with position 1 just start clicking on dialog components.

To modify the order at other than position 1, *shift-left click* to select a position number as a reference. The next left-click will set that dialog component to the next number and shift all higher order numbers accordingly. The Developer Studion on-line help covers this in more detail (search for *tab order*).

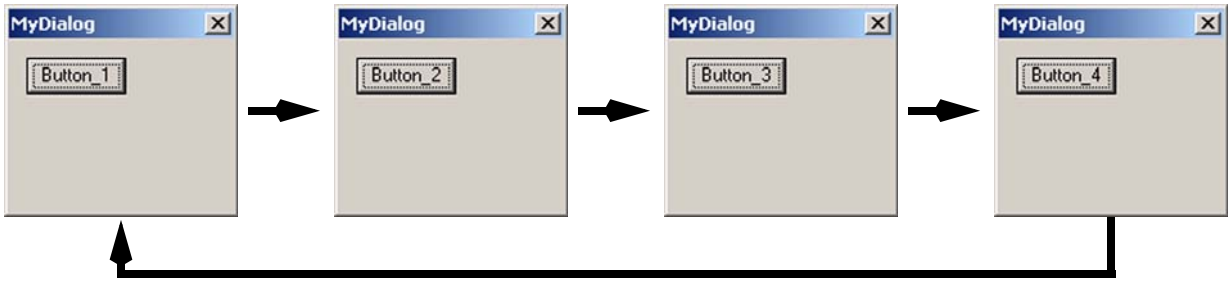
7.5.4.1 Dialog Editor Tips

- In general, the default styles are appropriate. Test any non-default styles carefully!
- Control-Z undo works when editing the dialog graphic
- Use the arrow keys to move dialog components in small increments
- Use the Dialog tools to align, size, etc. dialog components

7.5.5 Changing Dialog Button Text

Note: first available in h2.2.7/h1.2.7.

The example below shows how to change the text displayed in a dialog button control. In this example, each time the button is clicked the text is changed, cycling through 4 values:



Below, this button's resource ID = IDC_BUTTON1

Example

```
#include "tester.h"
#include "resource.h"
```

```
VOID_VARIABLE( Button_1, "" ){
   CString t = get_window_text(  Button_1 );
   if( t == "Button_1") set_window_text( Button_1, "Button_2" );
   else
      if( t == "Button_2") set_window_text( Button_1, "Button_3" );
   else
      if( t == "Button_3") set_window_text( Button_1, "Button_4" );
   else
      if( t == "Button_4") set_window_text( Button_1, "Button_1" );
}

DIALOG( dialog ){
   CONTROL( IDC_BUTTON1, Button_1 )
}
SITE_BEGIN_BLOCK( SB1 ){
   invoke( dialog );
}
```

7.5.6 Creating Bitmap Dialog Components

A bitmap graphic can be created in a user dialog. The example below creates a bitmap used to display a BIG RED FAIL in the dialog. Supporting C-code can conditionally make a dialog component, in this case the bitmap image, visible or invisible. Using this bitmap the application C-code would hide the *Fail* bitmap and show a *Pass* bitmap when appropriate.

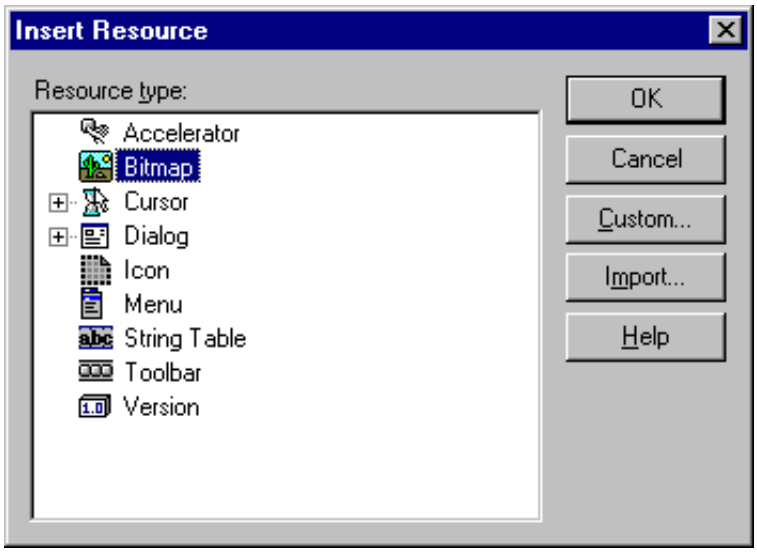
To use Bitmaps in a dialog requires the following basic steps. The first two steps are further outlined in detail below.

- Create the bitmap and add it to the project
- Add a Picture dialog component to the dialog and link it to the bitmap
- Create the C-code needed to use the bitmap. This is optional if the bitmap image serves no purpose other than as a static image.

To create and add a bitmap to a user dialog do the following steps:

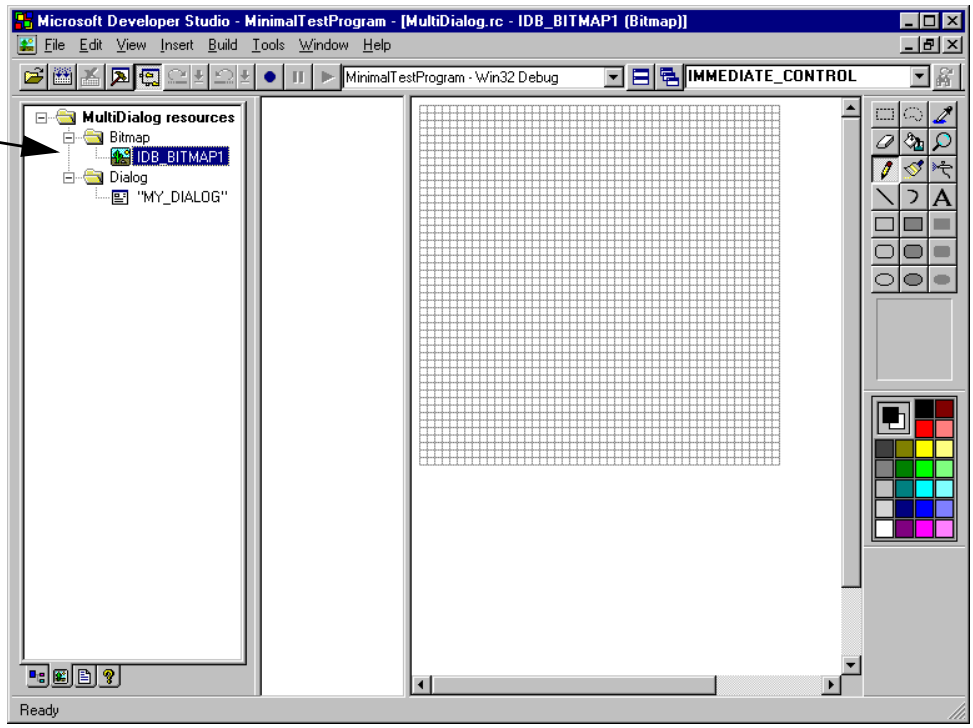
1. Create the desired bitmap and insert it into the project:
   a. Select **Insert: Resource...**

b. In the *Insert Resource* dialog click the Bitmap icon and click *OK*.

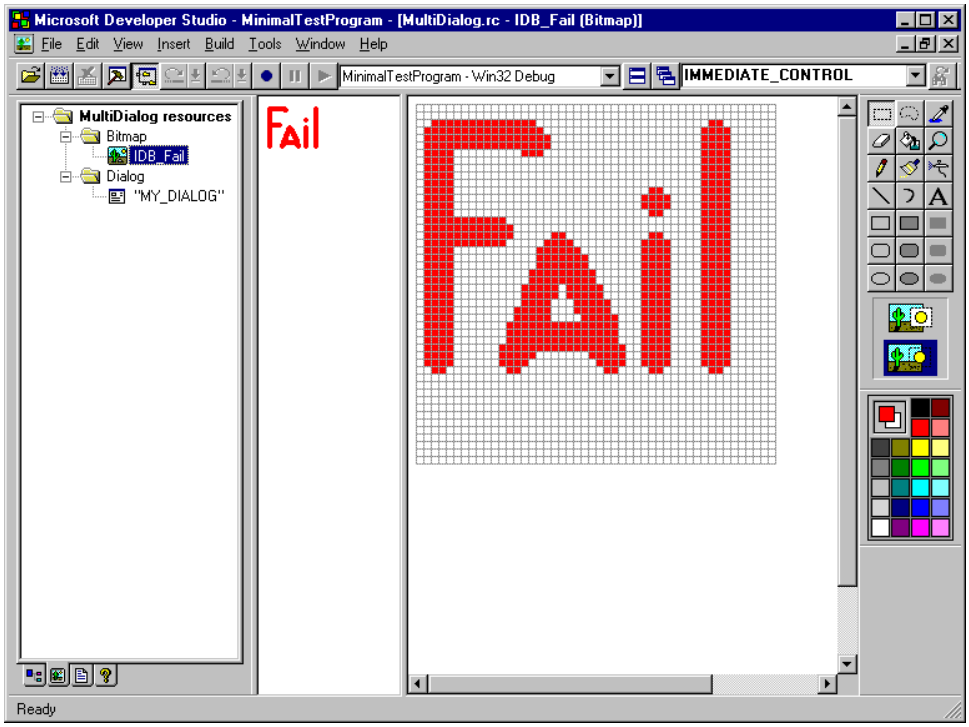


c. Click the *ResourceView* tab and note that a new *Bitmap* resource appeared. If this is the first Bitmap in this test program, it will be named `IDB_Bitmap1` by default. Note also that the MSDS bitmap editor is displayed and ready to use.

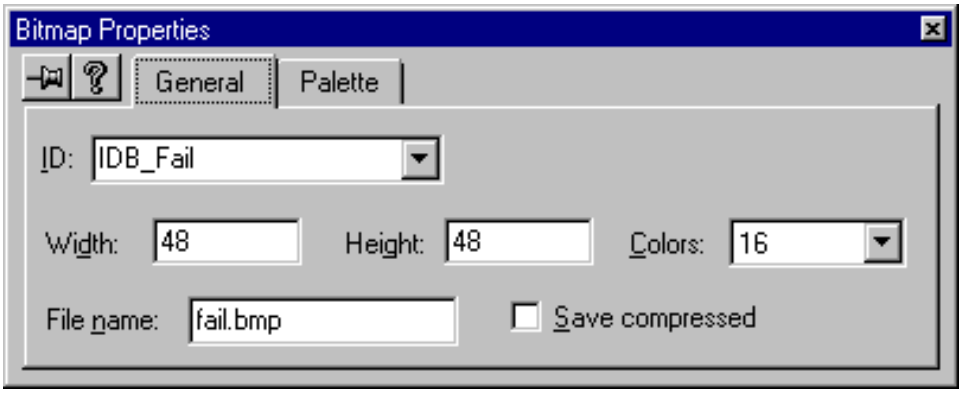
The new Bitmap resource.



2. Using the MDSD bitmap editor, create the *picture* you want. In this example, the BIG RED FAIL is created.

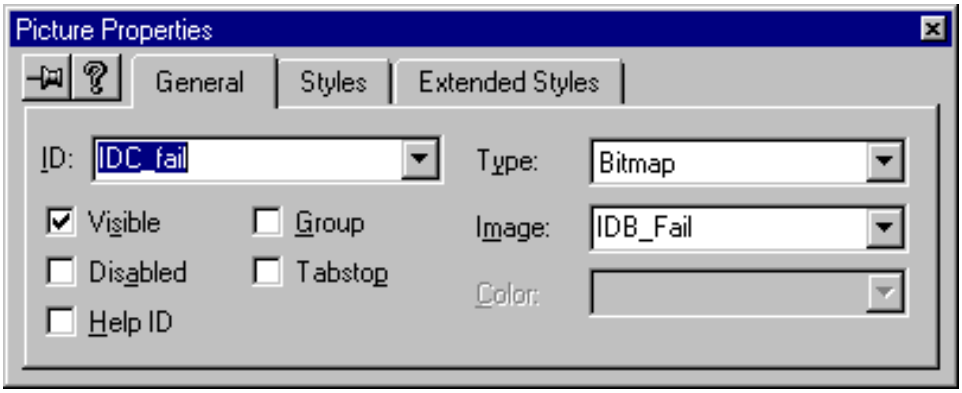


3. Double-click on the ***border*** of the drawn bitmap image to display the property editor. Change the ID to something meaningful. In the example, the ID was changed to *IDB_fail*. The *IDB_* prefix is the preferred convention for all bitmap IDs. Note, that the bitmap will be saved to a file named after the ID. The file will be located with the other program source files.



To add a Picture dialog component to the dialog and link it to the bitmap do the following:

1. In the MSDS ResourceView, double-click on the target dialog to open it for editing.
2. From the *Controls* tool bar select and add a *Picture* component to the dialog.
3. Double-click on the edge of the picture component to display the Picture Properties box.
   a. Edit the **ID** to a meaningful name. IDC_ is the preferred prefix for all dialog component IDs.
   b. In the **Type** window select Bitmap.
   c. In the **Image** window select the specific bitmap to be linked to this *Picture*.
   d. Review and select other properties as needed. The most basic property is whether the image is initially visible. Supporting C-code can conditionally make the image visible or invisible, which is commonly done to, for example, hide a *Fail* bitmap and show a *Pass* bitmap (and vice versa). It is the Picture ID (`IDC_fail`) which is referenced in the C-code to change the visibility state. The example properties looked like this:

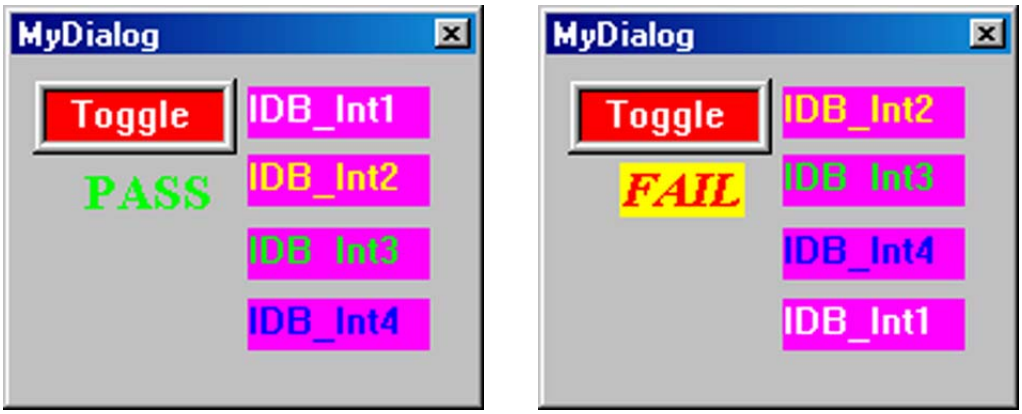


   e. Optionally write the C-code to use the Picture.

7.5.7 Bitmap Usage

In the context of User Dialogs, a bitmap is a [simple] user-created image which can be displayed or hidden, or used as a button. This allows the user to display pictorial objects in a

dialog or easily implement a custom button image. For example, the dialog below displays 6 bitmaps, one of which is used as a button (Toggle) the others are for display only:



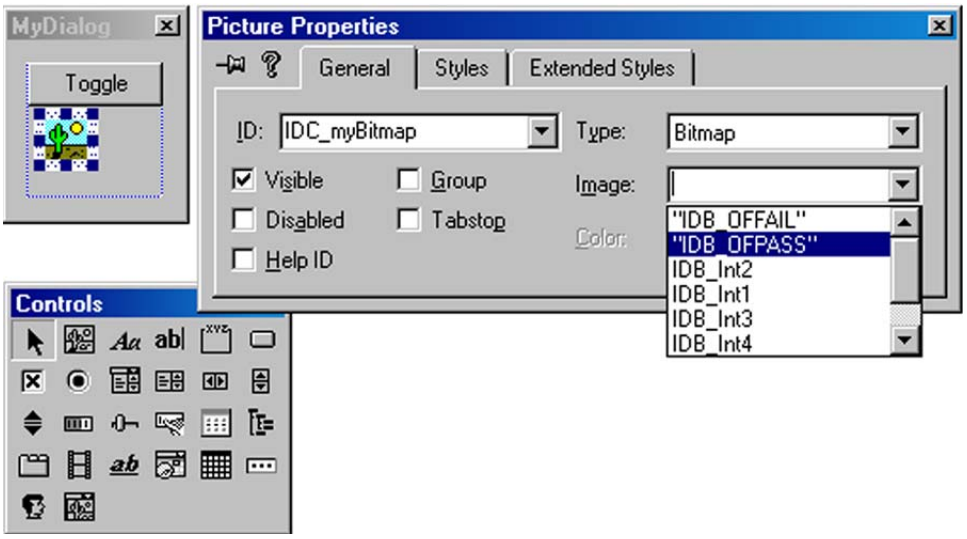
The code in Example 2: implements this dialog. In this example, when the `Toggle` bitmap is clicked, the following occurs:

- The *Pass* bitmap is replaced with the *FAIL* bitmap (or vice versa)
- The other 4 bitmaps are changed.

Bitmap support increased in Maverick software release v2.4.7. Prior to this release:

- A bitmap could only be associated with `CSTRING_VARIABLE`s, which were then used to hide/show the desired bitmap. This is capability is seen in Example 1: below.

- Bitmaps were always explicitly inserted into the user dialog using the resource `Control` dialog, and selecting and inserting a `Picture` resource into the dialog. Then, in the `Pictures Properties` dialog, the Type is changed to *Bitmap*, and the `Image` is selected:



Starting in Maverick software release v2.4.7, and for all Magnum 1/2/2x releases, the following features are available:

- The association of a given bitmap with a specific user variable can be done in program code.
- Bitmaps can be associated with most user variable types. Refer to Supported Dialog Components which shows the types of user variables which support the progress resource.
- The value of the user variable can be used to select the desired bitmap to be displayed.

These new features are seen in the two dialogs above, which are implemented in Example 2:

As noted above, and in Supported Dialog Components, a bitmap can be associated with most types of User-defined User Variables. Usage rules vary depending on the underlying type of the user variable:

- `CSTRING_VARIABLE`: if the value assigned to the variable is the (quoted) resource ID of a bitmap that bitmap will be displayed when the dialog or the control is updated. See Example 2:
- `VOID_VARIABLE`: if associated with a bitmap, clicking that bitmap invokes the body code of the variable.

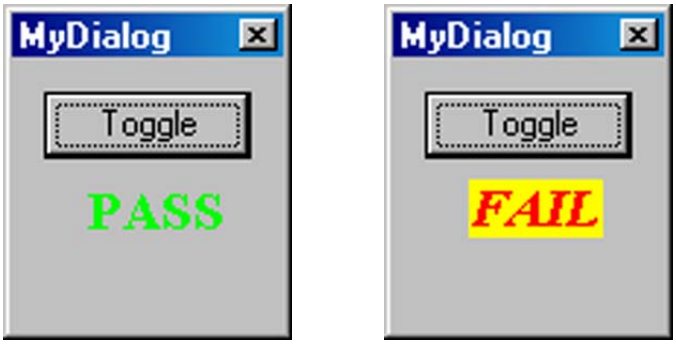
- DOUBLE_VARIABLE, FLOAT_VARIABLE, INT64_VARIABLE, INT_VARIABLE, and DWORD_VARIABLE all share he same rules. If the value of the variable is assigned to a bitmap resource ID, that bitmap will be displayed when the dialog or the control is updated.
- ONEOF_VARIABLE: the value list can represent n-bitmaps, where each value is the (quoted) resource ID of one bitmap. If the value of the variable is changed, the associated bitmap will be displayed when the dialog or the control is updated.

These are all demonstrated using Example 2:.

Examples

Example 1:

This example is supported in all software releases. The code below was used to implement the dialog below, shown in the two states which result when the `Toggle` button is clicked:



Note that the association of bitmap-to-user variable is done via the *Picture Properties* ID in the dialog resource editor (i.e. it isn't seen in C source code):

```
CSTRING_VARIABLE( uvTestPass, "uvTestPass", "") {}
CSTRING_VARIABLE( uvTestFail, "uvTestFail", "") {}
```

Clicking the button labled *Toggle* alternates hiding one bitmap and showing the other:

```
VOID_VARIABLE( uvToggle, "") {
   static BOOL toggle = TRUE;
   if (toggle = !toggle) {
      ShowWindow( get_HWND( uvTestPass ), SW_SHOW );
      ShowWindow( get_HWND( uvTestFail ), SW_HIDE );
   } else {
      ShowWindow( get_HWND( uvTestPass ), SW_HIDE );
```

```
      ShowWindow( get_HWND( uvTestFail ), SW_SHOW );
   }
}
DIALOG( MyDialog ) {
   IMMEDIATE_CONTROL( IDC_uvToggle, uvToggle )// Button
   CONTROL( IDC_uvTestPass, uvTestPass )     // Bitmap
   CONTROL( IDC_uvTestFail, uvTestFail )     // Bitmap
}
```

Example 2:

This example implements the dialog shown at the beginning of this section. It uses the features described above:

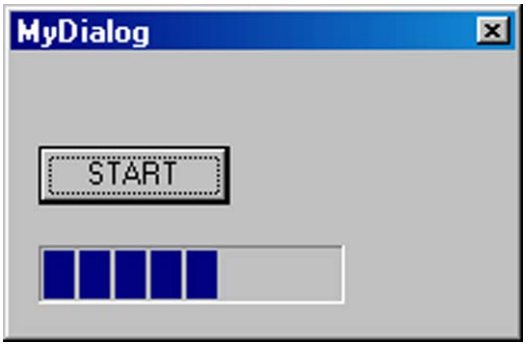
```
EXTERN_DIALOG( MyDialog1 ) // Forward
INT_VARIABLE( uvInt1, IDB_Int1, ""){}
INT_VARIABLE( uvInt2, IDB_Int2, ""){}
INT_VARIABLE( uvInt3, IDB_Int3, ""){}
INT_VARIABLE( uvInt4, IDB_Int4, ""){}
ONEOF_VARIABLE(uvOfTest,"IDB_OfPass,IDB_OfFail,IDB_OfBusy",""){}
VOID_VARIABLE( uvToggle1, "") {
   static int v = 0;
   switch (v) {
   case 0 :
      uvInt1 = IDB_Int1;
      uvInt2 = IDB_Int2;
      uvInt3 = IDB_Int3;
      uvInt4 = IDB_Int4;
      uvOfTest = "IDB_OfBusy"; v++; break;
   case 1 :
      uvInt1 = IDB_Int2;
      uvInt2 = IDB_Int3;
      uvInt3 = IDB_Int4;
      uvInt4 = IDB_Int1;
      uvOfTest = "IDB_OfFail"; v++; break;
   case 2 :
      uvInt1 = IDB_Int3;
      uvInt2 = IDB_Int4;
      uvInt3 = IDB_Int1;
      uvInt4 = IDB_Int2;
      uvOfTest = "IDB_OfBusy"; v++; break;
```

```
      case 3 :
         uvInt1 = IDB_Int4;
         uvInt2 = IDB_Int1;
         uvInt3 = IDB_Int2;
         uvInt4 = IDB_Int3;
         uvOfTest = "IDB_OfPass"; v = 0; break;
      }
      update_controls( MyDialog1 );
   }
   DIALOG( MyDialog1 ) {
      IMMEDIATE_CONTROL( IDC_uvToggle1, uvToggle1 )
      CONTROL( IDC_uvOfTest, uvOfTest )
      CONTROL( IDC_Int1, uvInt1 )
      CONTROL( IDC_Int2, uvInt2 )
      CONTROL( IDC_Int3, uvInt3 )
      CONTROL( IDC_Int4, uvInt4 )
   }
```

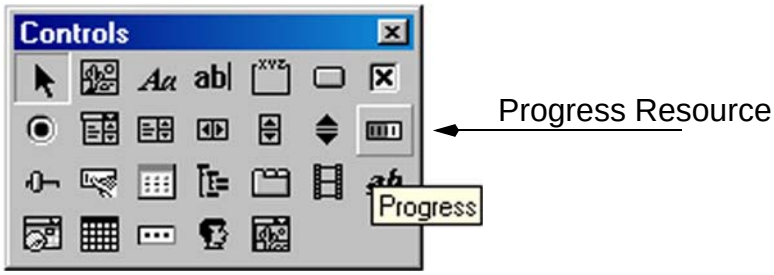
7.5.8 Dialog Progress Resource

The User Dialog below contains two resources:

- A button resource labeled START
- A *progress* resource i.e. the standard bar graph used to indicate some software activity or progress.



A progress resource object is added to User Dialogs the same way as other resource types, using the `Controls` dialog to select the resource type, then clicking in the user dialog to insert and locate the new resource. The image below shows the Progress resource::



Refer to Supported Dialog Components which shows the types of user variables which support the progress resource.

The number of segments displayed in a progress bar is determined by the value assigned to the associated user variable:

User Variable = 0 | User Variable = 50 | User Variable = 100

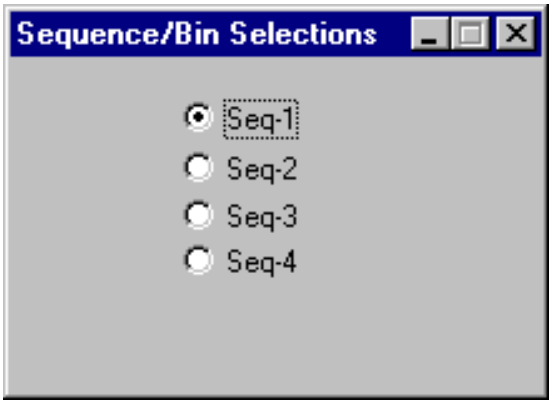
The following example code implements the dialog above. When the `Start` button is clicked, the progress resource is set to 0, then incrementally modified up and down. Try it.

```
INT_VARIABLE( progress, 50, "") {}
VOID_VARIABLE( TEST_START, "") {
   output("  TEST_START");
   for( int j = 0; j < 10; j++) {
      for( int i = 0; i < j*10; i++){
         progress = i;
         update_control(progress);
         Sleep( 10 );
      }
      for( i = j*10; i >= 0; i--){
         progress = i;
```

```
            update_control(progress);
            Sleep( 10 );
         }
      }
   }
   DIALOG( MyDialog3) { // Progress resource
      IMMEDIATE_CONTROL( IDC_TEST_START, TEST_START )
      CONTROL( IDC_PROGRESS1, progress )
   }
```

7.5.9 Radio Buttons and ONEOF User Variables

Radio buttons must be associated with `ONEOF_VARIABLE` user variables. The following example shows 4 radio buttons with the names Seq-1 through Seq-4. The example below uses this dialog:.



The following steps should be followed to use radio buttons:

1. Determine the number of options which are needed, and define a unique name for each option. In the example above, 4 options are shown with the names Seq-1, Seq-2, etc.

2. Create a ONEOF_VARIABLE user variable and include the names in the comma separated list:

```
ONEOF_VARIABLE( my_var, "Seq-1,Seq-2,Seq-3,Seq-4", "") {...}
```

   The names in the ONEOF list will be automatically displayed by the radio buttons (more below).

3. In the visual dialog, add the same number of radio buttons as there are options defined for the ONEOF_VARIABLE user variable. The example above shows this. The Label property of the radio buttons doesn't matter.

4. Only the first *Dialog Radio Button Properties ID* is actually used in the test program. The default ID for radio buttons must be used: IDC_RADIO1. Any radio button subseqently added will have sequential IDs (IDC_RADIO2, IDC_RADIO3, etc.). Note that this ordering is important, as noted below.

5. These radio buttons be contiguous in the tab order setting. See Setting Tab Order.

6. Any values set in *Caption* field of the *Radio Button Properties* dialog are replaced by the values assigned in the comma separated list of values defined for the ONEOF_VARIABLE. However, the size of the display field seen in the dialog must be set manually.

7. In the dialog definition code, associate the ONEOF_VARIABLE user variable with the first radio button in the series:

```
DIALOG( my_dialog ) {
   IMMEDIATE_CONTROL( IDC_RADIO1, my_var );
}
```

   At this point, proper operation assumes that since there are 4 ONEOF values there will also be 4 radio buttons, with the ID of each sequentially named after the one seen above i.e. IDC_RADIO2, IDC_RADIO3, IDC_RADIO4. Having too few radio buttons vs. ONEOF values is harmless. Having too many is *bad*.

8. Note that the IMMEDIATE_CONTROL() macro was used above. This means that the body code of the associated ONEOF user variable will execute immediately when a new radio button is selected. If this is not desired, use the CONTROL() macro instead, plus update_variable() to explicitly read the radio button value from C-code.

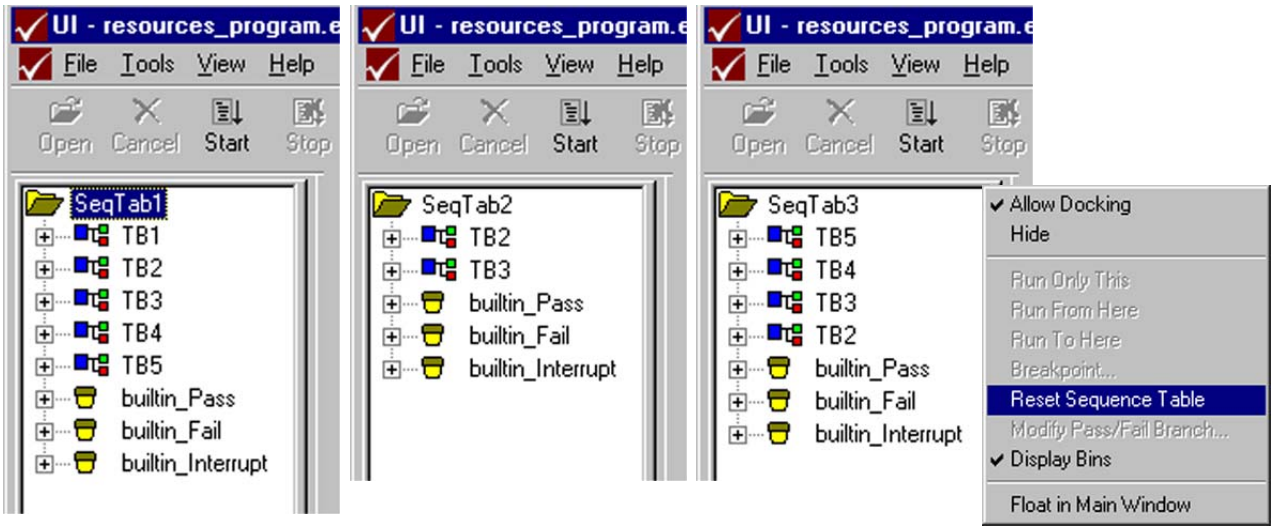
9. Invoke the dialog as desired.

Example

The example below uses radio buttons and supporting code to switch between Sequence/ Binning tables using a dialog. Note that this switching can be done any time test execution is not occurring, and as many times as desired.

The radio button display above determined the following images. The associated code is included below.

The images below show the Sequence/Binning table display in UI for each of the first three options selected using the radio buttons above and the code below:



The example above has the following parts:

- Dialog Code
- Test Block Code
- Sequence/Binning Code

Dialog Code

```
#include "tester.h"
#include "resource.h"

ONEOF_VARIABLE(seqbin_selection, "Seq-1,Seq-2,Seq-3,Seq-4", "x"){
   if (OnHost()) // Seq/Bin table must be mod on both HOST/SITE
      remote_send( seqbin_selection, 1, TRUE, INFINITE);

   // Uninitialize currently selected Seq/Bin table
   resource_deallocate( S_SequenceTable );
```

```
   // Select which new Seq/Bin table is to be used
   if (seqbin_selection == "Seq-1") SequenceTable_use("SeqTab1");
   if (seqbin_selection == "Seq-2") SequenceTable_use("SeqTab2");
   if (seqbin_selection == "Seq-3") SequenceTable_use("SeqTab3");
   if (seqbin_selection == "Seq-4") SequenceTable_use("SeqTab4");

   // Initialize new S/B table. This executes the SeqBin macro code
   // again to construct a new executable SeqBinTable.
   resource_initialize( S_SequenceTable );
}
DIALOG( my_dialog ) {
   IMMEDIATE_CONTROL( IDC_RADIO1, seqbin_selection );
}
```

Test Block Code

```
#include "tester.h"
TEST_BLOCK( TB1 ) {
   output(" Executing Test Block => %s", current_test_block() );
   return PASS;
}
TEST_BLOCK( TB2 ) {
   output(" Executing Test Block => %s", current_test_block() );
   return PASS;
}
TEST_BLOCK( TB3 ) {
   output(" Executing Test Block => %s", current_test_block() );
   return PASS;
}
TEST_BLOCK( TB4 ) {
   output(" Executing Test Block => %s", current_test_block() );
   return PASS;
}
TEST_BLOCK( TB5 ) {
   output(" Executing Test Block => %s", current_test_block() );
   return PASS;
}
```

Sequence/Binning Code

```
#include "tester.h"

SEQUENCE_TABLE( SeqTab1 ) {
   SEQUENCE_TABLE_INIT
   TEST( TB1,NEXT, STOP)
   TEST( TB2,NEXT, STOP)
   TEST( TB3,NEXT, STOP)
   TEST( TB4,NEXT, STOP)
   TEST( TB5,STOP, STOP)
}
SEQUENCE_TABLE( SeqTab2 ) {
   SEQUENCE_TABLE_INIT
   TEST( TB2,NEXT, STOP)
   TEST( TB3,STOP, STOP)
}
SEQUENCE_TABLE( SeqTab3 ) {
   SEQUENCE_TABLE_INIT
   TEST( TB5,NEXT, STOP)
   TEST( TB4,NEXT, STOP)
   TEST( TB3,NEXT, STOP)
   TEST( TB2,STOP, STOP)
}
SEQUENCE_TABLE( SeqTab4 ) {
   SEQUENCE_TABLE_INIT
   TEST( TB4,STOP, STOP)
}
```

7.5.10 Sliders & Scroll-bars

Note: first available in software release h3.5.xx.

The example User Dialog below contains several resources:

- A horizontal slider button resource labeled `Value 1` with an adjacent check-box labeled `Toggle Range`.

- A vertical slider button resource labeled `Value 2` with an adjacent check-box labeled `Toggle Range`.
- A horizontal scroll-bar resource labeled `Value 3` with an adjacent check-box labeled `Toggle Range`.
- A vertical scroll-bar resource labeled `Value 4` with an adjacent check-box labeled `Toggle Range`.
- The OK and Cancel buttons.

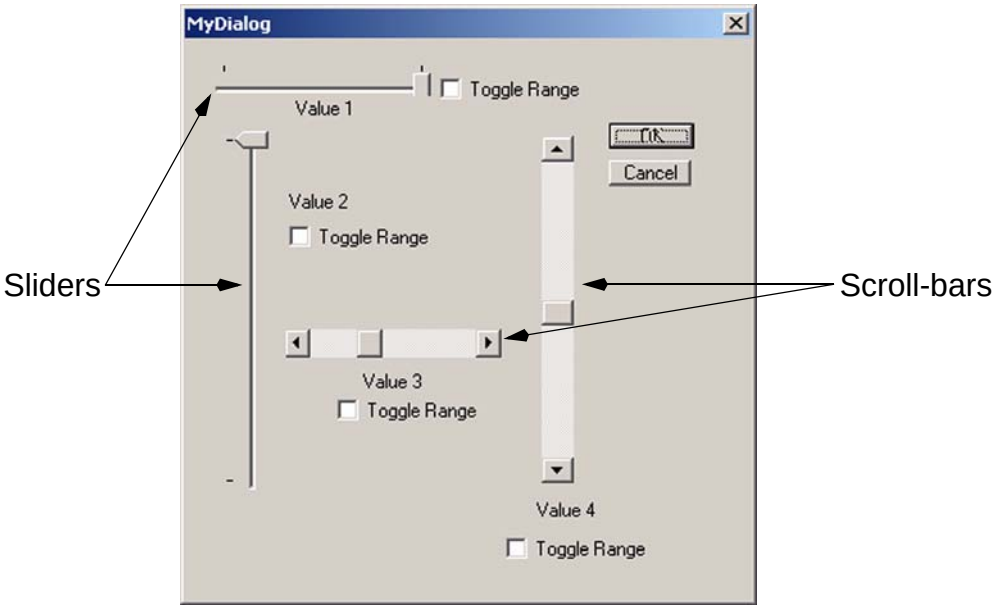


Figure-171: Dialog with Sliders & Scroll-bars

A slider or scroll-bar resource object is added to User Dialogs the same way as other resource types, using the `Controls` dialog to select the resource type, then clicking in the

User Dialog to insert and locate the new resource. The image below shows the related resource selections :

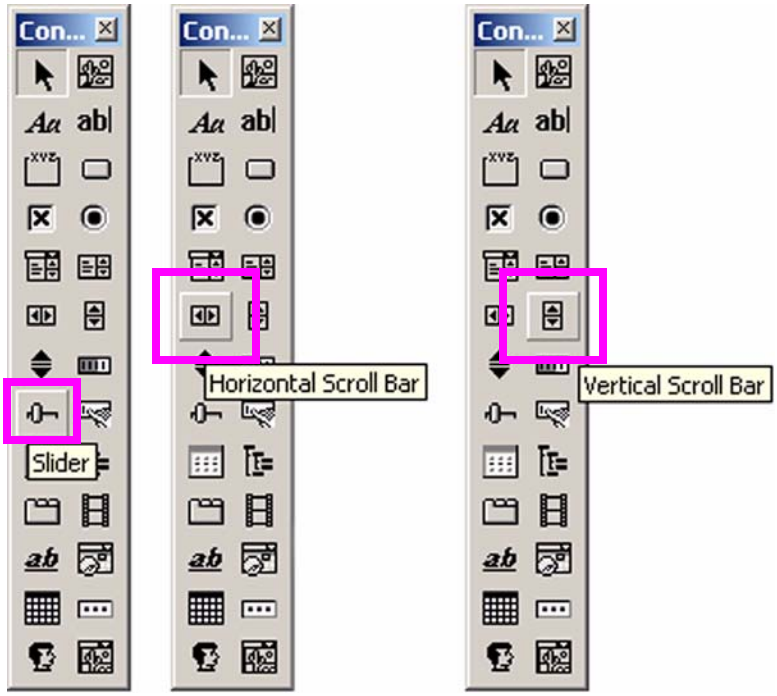


Figure-172: Slider & Scroll-bar Resource Selection

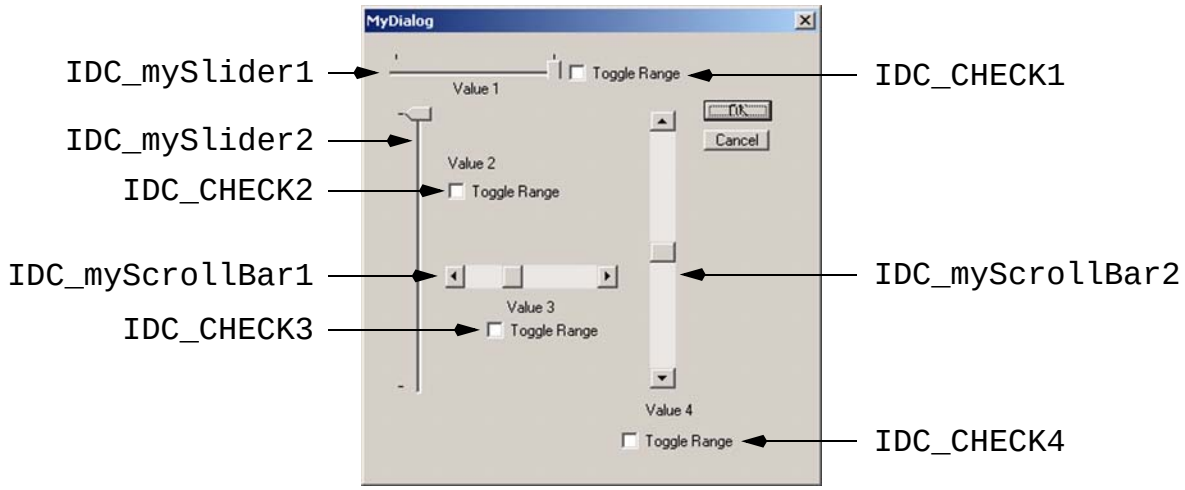
Rules:

- Sliders and/or scroll-bars represent positive integer values and may be associated with the following types of User Variables: `INT_VARIABLE`, `DWORD_VARIABLE`, `INT64_VARIABLE`, `UINT64_VARIABLE`.
- Slider and/or scroll-bar resources are always treated as immediate; i.e. clicking on or moving the slider or scroll-bar control immediately sends an event to the User Dialog, which causes the body-code of the associated User Variable to execute. Use the `IMMEDIATE_CONTROL()` macro to map a slider or scroll-bar resource to User Variables. This behavior cannot be changed.
- The default range of sliders and scroll-bars is 0 to 100, regardless of the initial value assigned to the associated User Variable. This range can be changed as noted below. Alternatively, the body code of the associated User Variable can scale and/or modify the value as desired.
- As indicated above, after a given User Dialog is invoked, the range represented by a member slider or scroll-bar can be changed, however the method used is different for sliders vs. scroll-bars (thank Microsoft):

- For sliders, the `SendMessage()` function is used, if necessary twice: once to set the range minimum value and again to set range maximum value, see Example.
- For scroll-bars, the `SetScrollRange()` function is used, see Example.

Example

The example code below implements the dialog shown in Dialog with Sliders & Scroll-bars and below. For reference, the dialog resource identities are shown below:




```
#include "tester.h"
#include "resource.h"

EXTERN_DIALOG( MyDialog )

INT_VARIABLE( mySlider1, 2019, "" ){
   output( "mySlider1 = %d", mySlider1 );
}

INT_VARIABLE( mySlider2, -44, "" ){
   output( "mySlider2 = %d", mySlider2 );
}

INT_VARIABLE( myScrollBar1, 33, "" ){
   output( "myScrollBar1 = %d", myScrollBar1 );
}

INT_VARIABLE( myScrollBar2, 51, "" ){
   output( "myScrollBar2 = %d", myScrollBar2 );
}
```

```
// Control to toggle max range value of mySlider1
BOOL_VARIABLE( mySlider1_range_toggle, FALSE, "" ){
   output("mySlider1_range_toggle => %s",
      (mySlider1_range_toggle? "TRUE":"FALSE") );
   // Change only the max range value
   SendMessage( get_HWND( mySlider1 ),
                TBM_SETRANGEMAX,
                TRUE,
                (mySlider1_range_toggle ? 1000 : 10) );
   update_variable( mySlider1 );
}
// Control to toggle both min and max range value of mySlider2
BOOL_VARIABLE( mySlider2_range_toggle, FALSE, "" ){
   output("mySlider2_range_toggle => %s",
      (mySlider2_range_toggle? "TRUE":"FALSE") );
   SendMessage( get_HWND( mySlider2 ),
                TBM_SETRANGEMIN,
                TRUE,
                (mySlider2_range_toggle ? 0 : 150) );
   SendMessage( get_HWND( mySlider2 ),
                TBM_SETRANGEMAX,
                TRUE,
                (mySlider2_range_toggle ? 10000 : 250 ) );
   update_variable( mySlider2 );
}
// Control to toggle max range value of myScrollBar1
BOOL_VARIABLE( myScrollBar1_range_toggle, FALSE, "" ){
   output("myScrollBar1_range_toggle => %s",
      (myScrollBar1_range_toggle? "TRUE":"FALSE") );
   SetScrollRange(get_HWND( myScrollBar1 ),
                  SB_CTL,
                  0,
                  (myScrollBar1_range_toggle ? 1000 : 10 ),
                  TRUE );
   update_variable( myScrollBar1 );
}
// Control to toggle both min and max range value of myScrollBar2
BOOL_VARIABLE( myScrollBar2_range_toggle, FALSE, "" ){
   output("myScrollBar2_range_toggle => %s",
      (myScrollBar2_range_toggle? "TRUE":"FALSE") );
```

```
   SetScrollRange(get_HWND( myScrollBar2 ),
                  SB_CTL,
                  (myScrollBar2_range_toggle ? 0 : 35 ), // Min
                  (myScrollBar2_range_toggle ? 275 : 55 ), // Max
                  TRUE  );
   update_variable( myScrollBar2 );
}
DIALOG( MyDialog ){
   CONTROL( IDC_mySlider1, mySlider1 )
   IMMEDIATE_CONTROL( IDC_CHECK1, mySlider1_range_toggle )
   CONTROL( IDC_mySlider2, mySlider2 )
   IMMEDIATE_CONTROL( IDC_CHECK2, mySlider2_range_toggle )
   CONTROL( IDC_myScrollBar1, myScrollBar1 )
   IMMEDIATE_CONTROL( IDC_CHECK3, myScrollBar1_range_toggle )
   CONTROL( IDC_myScrollBar2, myScrollBar2 )
   IMMEDIATE_CONTROL( IDC_CHECK4, myScrollBar2_range_toggle )
}

HOST_BEGIN_BLOCK( HB1 ){
   invoke( MyDialog );
}
```

7.5.11 User Dialog Functions

- Transferring Values to/from Dialog Resources
- `for_each()`
- `top_most()`

7.5.11.1 Transferring Values to/from Dialog Resources

Definition

The functions documented here are used to transfer values to/from dialog resources from/to User Variables.

The `update_variable()` function is used to set the value of the specified user variable to the value currently in its associated dialog component. The user variable value is only modified in the Host process. Two versions of `update_variable()` are available: one version which accepts an explicit dialog argument, to be used when the program contains more than one dialog, and one version which has no dialog argument, which can be used when a single dialog exists in the test program.

The `update_variables()` function is similar except that it updates all of the user variables assiociated with the specified dialog. The user variable values are only modified in the Host process.

The `update_control()` function is used to write the value of the specified user variable (in the Host process) to its associated dialog component. Two versions of `update_control()` are available: one version which accepts an explicit dialog argument, to be used when the program contains more than one dialog, and one version which has no dialog argument, which can be used when a single dialog exists in the test program.

The `update_controls()` function is similar except that it updates all of a dialogs components with the values in their associated user variables (in the Host process).

Usage

```
BOOL update_variables( Dialog *dialog );
BOOL update_variable( VariableProxy variable );
BOOL update_variable( Dialog *dialog, VariableProxy variable );
BOOL update_controls( Dialog *dialog );
BOOL update_controls();
BOOL update_control( VariableProxy variable );
BOOL update_control( Dialog *dialog, VariableProxy variable );
```

where:

`dialog` identifies the dialog of interest, to be read from or updated.

`variable` identifies a specific user variable.

These functions return `TRUE` when the operation is successful, otherwise `FALSE` is returned. This is important with specifying a user variable as a `CString` value because the compiler cannot check to see if the name is valid.

Example

```
INT_VARIABLE( int_val, 1, ""){}

DIALOG( some_dialog) {
   CONTROL( IDC_name, int_val);
}

// Some user C-code executing in the host process
... other code here ...

update_variable ( int_val ); // Get current value from dialog
output ("The current dialog value for name is => %s", int_val );
int_val++;
update_control( int_val ); // Update dialog /w new value
... other code here ...
```

7.5.11.2 for_each()

Description

This version of the `for_each()` function provides an iteration capability used to process dialog components (resources).

When called, `for_each()` iterates over each component, of the specified User Dialog, which is tied to a User Variable (using the `CONTROL/IMMEDIATE_CONTROL` macros).

For each iteration (for each dialog component) the specified user-defined call-back function is executed, and passed a pointer to the `DialogEntry` information about that component. The `DialogEntry` definition is:

```
struct DialogEntry {
   int id;
   Variable *variable;
   BOOL callFunc;
   BOOL immediate;
   BOOL hexdisplay;
   void (*graphic)( window w );
   window window;
   CString name;
};
```

Note: some of the `DialogEntry` parameters will not be useful in user-written C-code.

Usage

```
void for_each( Dialog *dialog,
               BOOL (*func)( DialogEntry *entry ) );
```

where:

`dialog` is the dialog of interest.

`*func` is a pointer to a user-written C-function to be called for each iteration of the `for_each()` function. The function prototype is:

```
BOOL func( DialogEntry *entry )
```

where:

`func` is the user-defined name of the call-back function.

`entry` is a pointer to a variable, created by the system software, used to pass information about the dialog component to the call-back.

The call-back function code should return TRUE to continue the iteration, or can return FALSE to terminate iteration.

Example

???

7.5.11.3 top_most()

Description

The `top_most()` function was added to allow user-written C-code to set or modify the top-most attribute of a User Dialog.

When a dialog's top-most attribute is TRUE, that dialog will remain visible at all times (except when another top-most dialog is invoked later). If FALSE, other windows and dialogs can be placed on top of the dialog, partially or completely hiding it.

Usage

```
void top_most( Dialog *dialog, BOOL mode );
```

where:

`dialog` is the User Dialog of interest.

`mode` is TRUE to set the top-most attribute and FALSE to reset the top-most attribute.

Example

???

7.5.12 Grid Usage

See Creating a User Dialog, Supported Dialog Components.

Note: first available in software release h1.1.23.

This section covers the following:

- Overview
- Adding a Grid to a Dialog
- GRID_CONTROL() Macro
- ONINITDIALOG: Defining the Grid
- Grid Functions
  - Types, Enums, etc.
  - `grid_create()`
  - `grid_setup()`
  - `grid_fixed_col_width_set()`
  - `grid_fixed_row_height_set()`
  - `grid_column_pixel_width_set()`
  - `grid_row_pixel_height_set()`
  - `grid_initialize()`
  - `grid_update()`
  - `grid_focus_cell_get()`
  - `grid_reset()`
- Grid Call-back Functions

7.5.12.1 Overview

See Grid Usage.

A Grid is a graphic resource which can be used in User Dialogs. A Grid is a matrix of cells which can display color and/or text under control of user written code. For example, the following images show a Grid in three user dialogs:

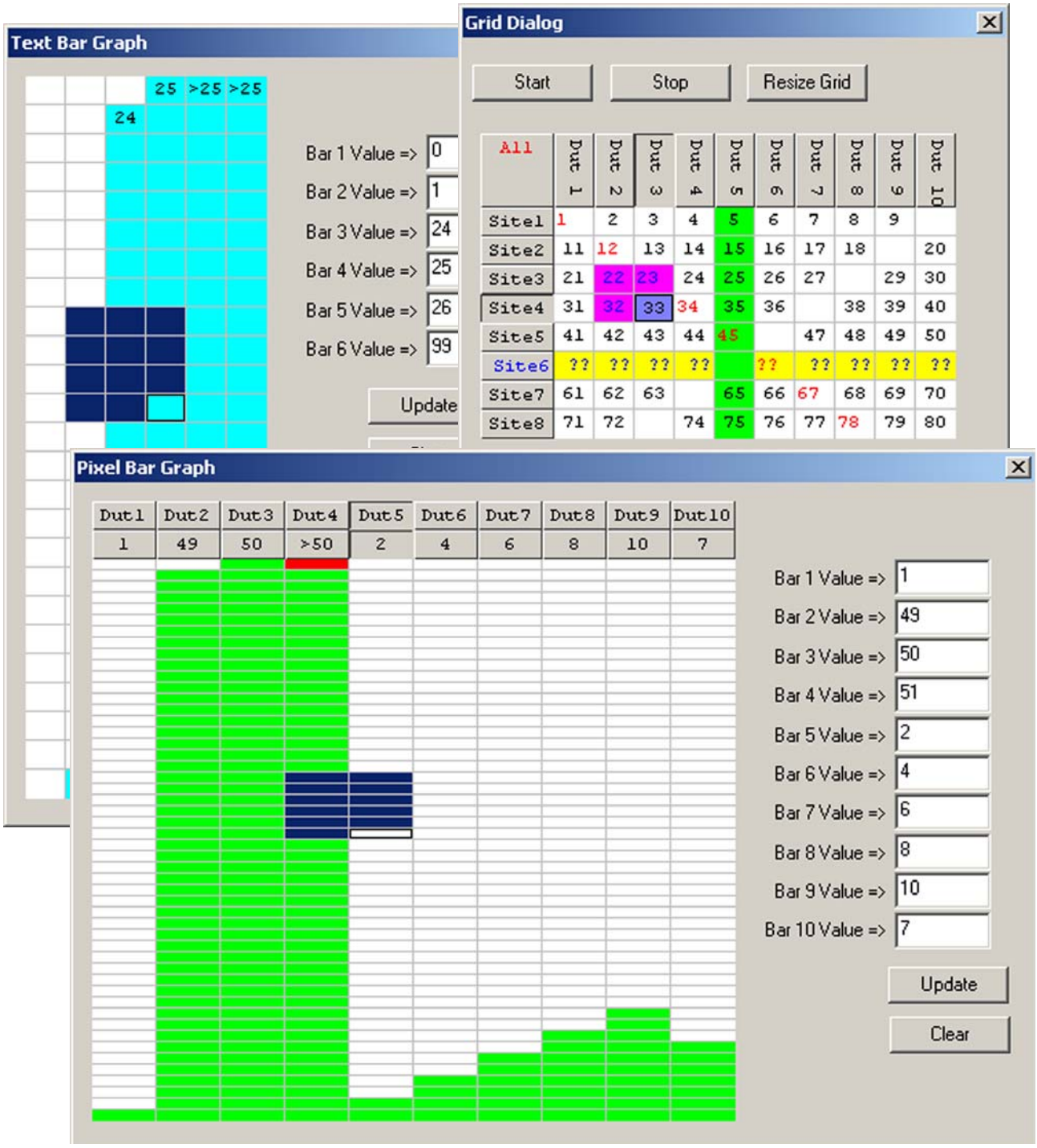


Figure-173: Example Dialogs with Grid

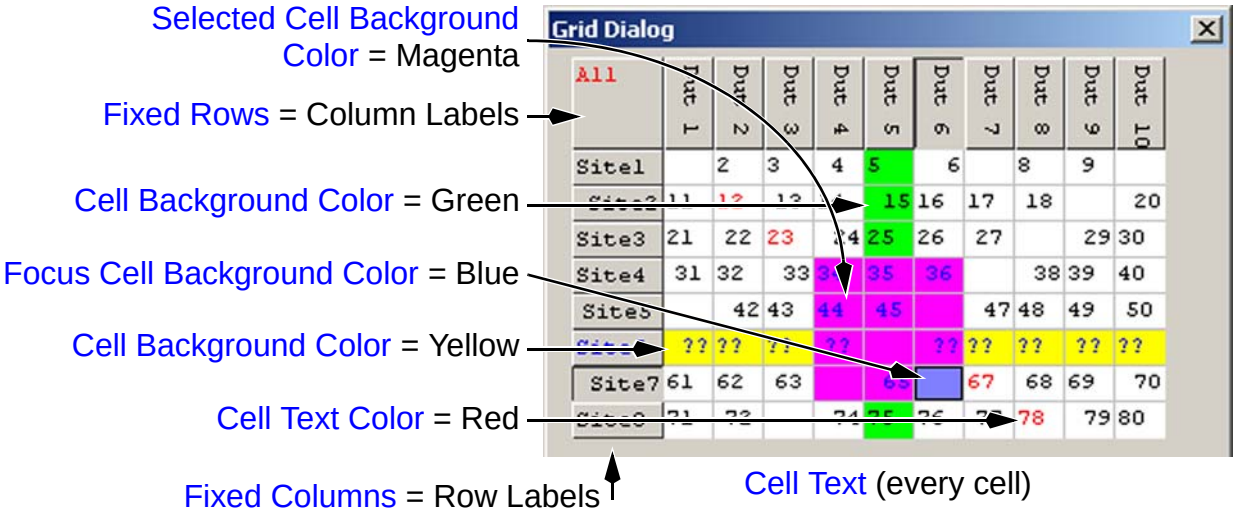
In the previous image, the left-most dialog implements a vertical bar-graph in which cells selectively use color and text to display the value of an associated integer value. The upper right dialog demonstrates many of the available programmable attributes including cell text, cell background color, selected cell color, focussed cell color, etc. The lower-right dialog also contains a bar-graph in which the main cells are defined using pixel dimensions rather than text dimensions. Note that these dialogs also include other features which are not specific to Grid use but were used to experiment with the grid options, cell content, cell color, etc.

Using a Grid requires the following sequence:

- Graphically Adding a Grid to a Dialog.
- Programmatically adding the grid using the GRID_CONTROL() Macro. This also registers the Grid Call-back Functions which are used to determine many of the various Grid Attributes.
- Using ONINITDIALOG: Defining the Grid. This includes the number of rows/ columns, number of fixed rows/columns, row height and column width, etc.

Normally, the previous steps are performed once, to define the Grid. Then `grid_update()` is executed to refresh/modify the information displayed by the Grid. The various call-back functions specified using the GRID_CONTROL() Macro are executed to actually update the Grid's display.

As indicated, a Grid has many attributes which are defined and controlled by user code:



Cell Format: text in some cells is left justified, in others center or right justified

Figure-174: Grid Attributes

The following table describes these Grid attributes:

Attribute	Description
Fixed Rows	Zero or more rows fixed at the top of the Grid, typically used for column label(s). The number of fixed rows is set using the `fixed_rows` argument to `grid_setup()`, executed by ONINITDIALOG: Defining the Grid. The height of fixed rows is set using the `grid_fixed_row_height_set()` function, executed by ONINITDIALOG: Defining the Grid. The text orientation (vertical vs. horizontal) is set by the `verticalLabel` argument to the GRID_CONTROL() Macro. Text content is controlled by the GridCellTextCallback Call-back Function. Text color is set by the GridTextColorCallback Call-back Function. Text justification cannot be modified. The call-backs are registered using the GRID_CONTROL() Macro and executed during `grid_initialize()` and `grid_update()`.
Fixed Columns	Zero or more columns fixed at the left of the Grid, typically used for row label(s). The number of fixed columns is set using the `fixed_cols` argument to `grid_setup()`, executed by ONINITDIALOG: Defining the Grid. The width of fixed columns is set using the `grid_fixed_col_width_set()` function, executed by ONINITDIALOG: Defining the Grid. Text orientation is always horizontal. Text content is controlled by the GridCellTextCallback Call-back Function. Text color is set by the GridTextColorCallback Call-back Function. Text justification is set by the GridCellFormatCallback Call-back Function. The call-backs are registered using the GRID_CONTROL() Macro and executed during `grid_initialize()` and `grid_update()`.
Cell Text	The text displayed in each cell. Controlled by the GridCellTextCallback Call-back Function which is registered using the GRID_CONTROL() Macro and executed during `grid_initialize()` and `grid_update()`.
Cell Text Color	The color of text displayed in each cell. Controlled by the GridTextColorCallback Call-back Function which is registered using the GRID_CONTROL() Macro and executed during `grid_initialize()` and `grid_update()`.

Attribute	Description
Cell Format	The text justification for a most cells. Controlled by the GridCellFormatCallback Call-back Function which is registered using the GRID_CONTROL() Macro and executed during `grid_initialize()` and `grid_update()`.
Cell Background Color	The background color of non-fixed cells. Controlled by the GridBackgndColorCallback Call-back Function which is registered using the GRID_CONTROL() Macro and executed during `grid_initialize()` and `grid_update()`.
Selected Cell Text Color	The color of text in non-fixed cell(s) which are selected. Controlled by the GridSelectedTextColorCallback Call-back Function which is registered using the GRID_CONTROL() Macro and executed during `grid_initialize()` and `grid_update()`.
Selected Cell Background Color	The background color of selected non-fixed cell(s). Controlled by the GridSelectedBackgndColorCallback Call-back Function which is registered using the GRID_CONTROL() Macro and executed during `grid_initialize()` and `grid_update()`.
Focus Cell Background Color	The background color of the non-fixed cell with which currently has focus. Controlled by the GridFocusBackgndColorCallback Call-back Function which is registered using the GRID_CONTROL() Macro and executed during `grid_initialize()` and `grid_update()`.

During `grid_initialize()` and `grid_update()`, the various Grid Call-back Functions functions are executed once for each cell in the Grid and receive the row and column coordinate of a cell. These coordinate values allow user code to conditionally determine the value to be set for a given cell. For example, the text and background color for a cell a row/column 3/1 can be set differently than that in row/column 3/2, etc.

In all cases, these call-back functions are executed for every cell in the Grid, including any fixed row or fixed column cells. The Grid cell numbering system is shown in the image below:

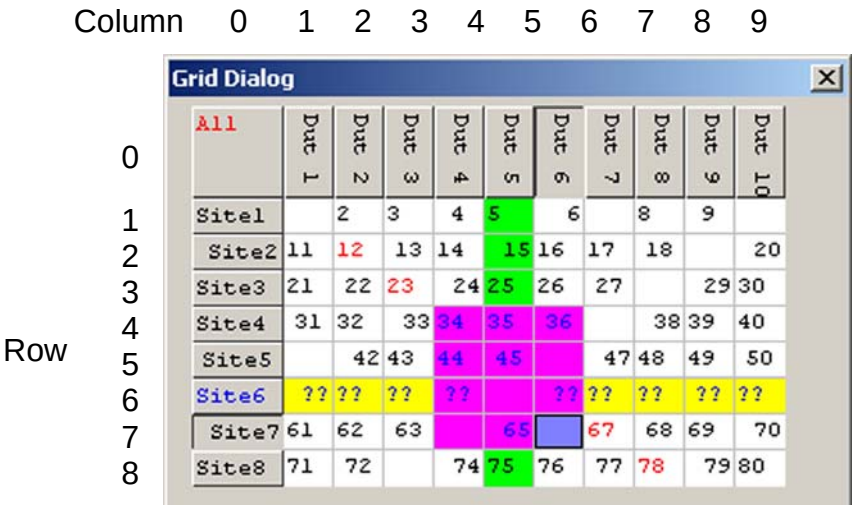


Figure-175: Grid Row/Column Numbering

7.5.12.2 Adding a Grid to a Dialog

See Overview.

Adding a Grid resource to User Dialogs is done using the Developer Studio resource editor, much like adding a Button, edit box, etc. However, a Grid has more details to manage than the other dialog resources. The images below show the key editing controls used to add a Grid to a dialog:

:

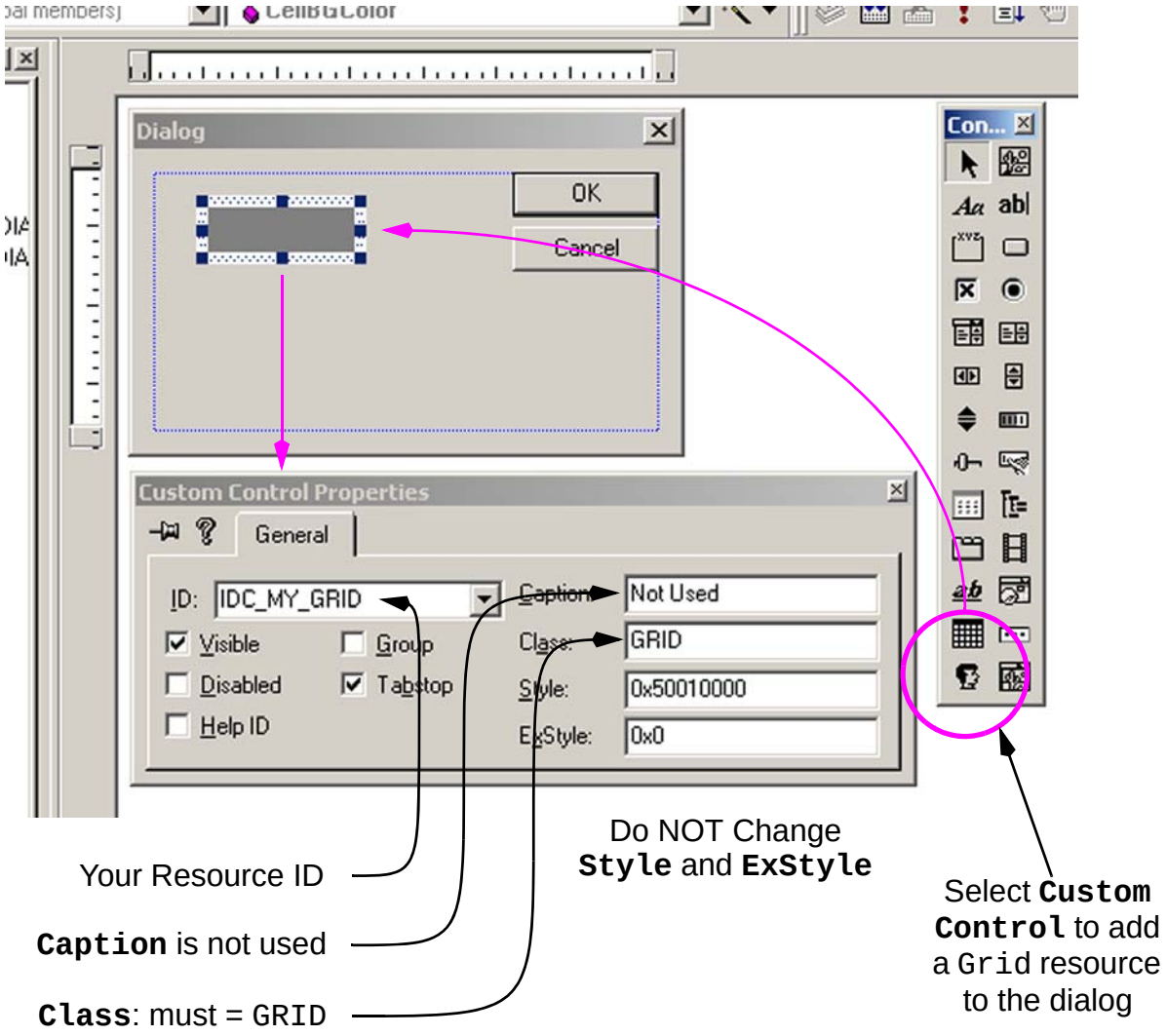


Figure-176: Resource Editor Grid Controls

Note the following:

- See Creating the Dialog Graphic for an overview of the general process of adding a resource to a dialog.
- Using the **Controls** dialog select the Custom Control and click in the dialog frame to insert the Grid resource.
- Note that the image displayed for a Grid resource is as shown above; i.e. it will not look like the resulting grid seen in the actual dialog.

- Position the upper-left corner of the Grid resource to the desired location. It is OK to resize the visual Grid resource but note that the final Grid size is solely determined by its configuration, as set using various Grid Functions in ONINITDIALOG: Defining the Grid.

Note: using a Grid, an iterative sequence is required to obtain the desired dialog layout. The upper-left corner of each Grid is visually positioned in the dialog, using the resource editor, whereas the Grid's width and height are solely determined by user code, using `grid_setup()`, `grid_fixed_col_width_set()`, `grid_fixed_row_height_set()`, `grid_column_pixel_width_set()`, `grid_row_pixel_height_set()`. All other dialog resources are positioned and sized visually, using the resource editor. The actual dialog layout can only be tested by executing the test program and viewing the dialog.

- Double-click the Grid resource to display its properties (the **`Custom Control Properties`** dialog), as shown above. In this dialog make only the following changes:
  - Change the resource ID, as desired, to a meaningful identifier. This becomes the first argument passed to the GRID_CONTROL() Macro, to programmatically associate the grid to the dialog.
  - Change the **`Class`** value to `GRID`. This makes the **`Custom Control`** a Grid.
  - Do NOT change the **`Style`** and **`ExStyle`** values.
  - The **`Caption`** is not used and can be ignored.

7.5.12.3 GRID_CONTROL() Macro

See Overview, Adding a Grid to a Dialog.

Description

The GRID_CONTROL( ) macro is used to programmatically add a Grid resource to a dialog and to identify the call-back functions which will execute when updating the Grid's display. See Creating the Dialog C-code for an overview of this topic.

For example:

```
DIALOG( myDialog) {
   CONTROL( ...other controls... )
   IMMEDIATE_CONTROL( ...other controls...)
   GRID_CONTROL(
        IDC_MY_GRID,
        FALSE, // TRUE = Vertical text in Fixed Rows
        0, // Set = 0, reserved for future
        myCellTextCallback,
        myCellTextFormatCallback,
        myCellTextColorCallback,
        mySelectedCellTextColorCallback,
        myCellClickedCallback,
        myCellBGColorCallback,
        mySelectedCellBGColorCallback,
        myFocusBGColorCallback )
   ONINITDIALOG( myGridInitFunc )
}
```

- Note that the GRID_CONTROL() Macro has a number of arguments, each of which must be specified. The value 0 can be specified for any of the various call-back functions which are not needed by a given Grid application.
- In any dialog containing a Grid, use the ONINITDIALOG( ) macro to specify the a function used to initialize the various Grid attributes. Details are covered in ONINITDIALOG: Defining the Grid. Note that an ONINITDIALOG function may also contain code unrelated to Grid applications.

Usage

```
GRID_CONTROL(
   int IDCustom,
   BOOL verticalLabel,
   GridReadCallback readFunc,
   GridCellTextCallback cellTextFunc,
   GridCellFormatCallback cellFormatFunc,
   GridTextColorCallback textColorFunc,
   GridSelectedTextColorCallback selTextColorFunc,
   GridCellClickedCallback cellClickedFunc,
   GridBackgndColorCallback backColorFunc,
   GridSelectedBackgndColorCallback selBackColorFunc,
   GridFocusBackgndColorCallback focusBackColorFunc );
)
```

where:

IDCustom is the resource ID of the Grid, as set using the **Custom Control Properties** dialog. The example in Figure-176: uses IDC_MY_GRID.

verticalLabel determines whether text in Fixed Rows (only) is displayed vertically or horizontally. The upper-right example in Figure-173: has vertical text (verticalLabel = TRUE) and the lower example has horizontal text (verticalLabel = FALSE). This setting affects all fixed rows. Setting **verticalLabel** = FALSE requires that the Grid's row width, set using grid_setup(), be adequate to display the widest string used in any fixed row cell. Setting **verticalLabel** = TRUE requires that the fixed row height, set using grid_row_pixel_height_set(), be adequate to display the tallest string used in any fixed row cell. Both grid_setup() and grid_row_pixel_height_set() are executed in the ONINITDIALOG: Defining the Grid.

readFunc is reserved for future applications. This value should be set = 0.

cellTextFunc is optional and used to register a user-written GridCellTextCallback Call-back Function. This function determines the Cell Text displayed in every Grid cell. Set this argument = 0 if no text is to be displayed in the Grid (including in any Fixed Rows and/or Fixed Columns). See GridCellTextCallback Call-back Function for more details.

cellFormatFunc is optional and used to register a user-written GridCellFormatCallback Call-back Function. This function determines the text justification (left, center, right) for any text displayed in the main cell area or in Fixed Columns. Default = center. Set this argument = 0 if no text justification is required. See GridCellFormatCallback Call-back Function for more details.

textColorFunc is optional and used to register a user-written GridTextColorCallback Call-back Function. This function determines the Cell Text Color displayed in every Grid cell. Set this argument = 0 if no non-default text color is to be displayed in the Grid (including in any Fixed Rows and/or Fixed Columns). See GridTextColorCallback Call-back Function for more details.

selTextColorFunc is optional and used to register a user-written GridSelectedTextColorCallback Call-back Function. This function determines the Selected Cell Text Color displayed in selected non-fixed Grid cells. Set this argument = 0 if no non-default text color is to be displayed in selected cells. See GridSelectedTextColorCallback Call-back Function for more details.

cellClickedFunc is optional and used to register a user-written GridCellClickedCallback Call-back Function. If registered, this call-back is executed any time the user clicks a Grid cell. The row/column coordinates of the cell *with focus* are passed to the call-back function. See GridCellClickedCallback Call-back Function for more details.

backColorFunc is optional and used to register a user-written GridBackgndColorCallback. This function determines the Cell Background Color displayed in every non-fixed Grid cell. Set this argument = 0 if no non-default background cell color is to be displayed in the Grid. See GridBackgndColorCallback for more details.

selBackColorFunc is optional and used to register a user-written GridSelectedBackgndColorCallback. This function determines the Selected Cell Background Color displayed in every selected non-fixed Grid cell. Set this argument = 0 if no non-default background cell color is to be displayed in selected Grid cells. See Selected Cell Background Color for more details.

focusBackColorFunc is optional and used to register a user-written GridFocusBackgndColorCallback. This function determines the Focus Cell Background Color displayed in the non-fixed Grid cell which has focus. Set this argument = 0 if no non-default background cell color is to be displayed in the Grid cell with focus. See GridFocusBackgndColorCallback for more details.

Example

See Description.

7.5.12.4 ONINITDIALOG: Defining the Grid

See Overview, Adding a Grid to a Dialog.

Description

In any dialog containing a Grid, some aspects of the Grid are defined using an ONINITDIALOG function. This is a user-written function which is identified in the dialog definition code using the ONINITDIALOG() macro. The function name and code are user-written, the function prototype is defined by Nextest:

```
void funcID (BOOL created)
```

where the name of the function is user-defined, and is the argument passed to the ONINITDIALOG() macro. During dialog creation the specified function is executed twice, once before any dialog resources are created (created = FALSE) and once after all dialog resources are created (created = TRUE). The Grid definition code must only execute in the latter case (see Example).

The Grid-related tasks performed in the ONINITDIALOG function code must follow the following sequence:

- Get the handle to the Dialog containing the Grid. This is needed by grid_create(), next.
- Execute grid_create() to create the Grid and get a handle to it.
- Execute grid_setup() to define various required Grid attributes:
  - Number of Grid rows
  - Number of Grid columns
  - Number of Fixed Rows
  - Number of Fixed Columns
  - Row height, in characters
  - Column width, in characters

  Note that the row height and column width may be subsequently redefined using pixel dimensions using grid_row_pixel_height_set() and grid_column_pixel_width_set() and that fixed rows/column sizes may be set using grid_fixed_row_height_set() and grid_fixed_col_width_set().

  Also note that a Grid's physical display size is affected by this function and several others. It may be necessary to iteratively adjust the dialog's size and/or the location of other dialog components to obtain the desired visual display. See Note:.
- If appropriate, execute grid_fixed_col_width_set() once for each fixed column, to adjust the width of any Fixed Columns.

- If appropriate, execute `grid_fixed_row_height_set()` once for each fixed row, to adjust the height of any Fixed Rows. This height must be adjusted if fixed row text is displayed vertically vs. horizontally (set by the `verticalLabel` argument of `GRID_CONTROL()`).
- Execute `grid_initialize()` to update the initial Grid display. Any registered Grid Call-back Functions which affect the Grid display will be executed, once for each cell in the Grid. These user-written Grid Call-back Functions solely determine the text, colors, etc. displayed in each cell.

Example

The following example is used to describe the various Grid-related details. This example `ONINITDIALOG` function is referenced, by name, in the GRID_CONTROL() Macro example:

```
HWND hMyGrid; // Handle to a Grid. Also used elsewhere by Grid APIs

void myGridInitFunc( BOOL created ){

   if (!created) return; // Dialog must be created to define a Grid

   // Get handle to Dialog containing the Grid
   HWND hDialog = get_HWND( myDialog );

   // Create a specific Grid and get a handle to it
   hMyGrid = grid_create( hDialog, IDC_MY_GRID ); // grid_create()

   // Set required Grid attributes
   grid_setup(  hMyGrid, // grid_setup()
                8,   // Number of Grid rows
                10,  // Number of Grid columns
                1,   // Number of Fixed Rows
                1,   // Number of Fixed Columns
                1,   // Row height, in characters
                3 ); // Column width, in characters

   // Set width of first (0) fixed width column, in characters
   grid_fixed_col_width_set( hMyGrid, 0, 6 );

   // Set height of first (0) fixed height row. in characters
   grid_fixed_row_height_set( hMyGrid, 0, 1 );

   grid_initialize( hMyGrid ); // grid_initialize()
}
```

7.5.12.5 Grid Functions

See Overview, ONINITDIALOG: Defining the Grid.

This section covers the following. They are documented in the order normally used:

- Types, Enums, etc.

The following functions are normally only used in the user-written ONINITDIALOG function, to define initial Grid attributes. See ONINITDIALOG: Defining the Grid:

- `grid_create()`
- `grid_setup()`
- `grid_fixed_col_width_set()`
- `grid_fixed_row_height_set()`
- `grid_column_pixel_width_set()`
- `grid_row_pixel_height_set()`
- `grid_initialize()`

The following functions are used after a dialog is created:

- `grid_update()` - update the information displayed by the Grid. Executes any registered Grid Call-back Functions.
- `grid_focus_cell_get()` - get the cell coordinates for the cell with the focus. See Grid Row/Column Numbering.
- `grid_reset()` - resets the selection and moves focus to the first non-fixed cell.

7.5.12.6 Types, Enums, etc.

See Grid Functions, Grid Call-back Functions.

Description

The following structure is used in support of various Grid Functions and Grid Call-back Functions.

Usage

The `GridCell` structure is used by various Grid Functions and Grid Call-back Functions to identify a specific cell in a Grid. Location 0, 0 is the top left cell in the grid (see Grid Row/ Column Numbering):

```
struct GridCell {
   int row;
   int col;
};
```

7.5.12.7 grid_create()

See Grid Functions, ONINITDIALOG: Defining the Grid.

Description

The `grid_create()` function is used to create a Grid and return a handle to it. Note the following:

- The `grid_create()` function is executed in the user-written ONINITDIALOG function (see ONINITDIALOG: Defining the Grid).
- A given user dialog may contain more than one Grid. The returned Grid's handle is used as an argument to other Grid-related functions to identify a specific Grid. Since some of these functions will be executed outside the context of the `ONINITDIALOG` function the scope of the handle variable (`HWND hMyGrid` in the earlier example) must be considered.

Usage

```
HWND grid_create( HWND hDlg, int nIDCustom );
```

where:

`hDlg` is a previously initialized `HWND` variable identifying the dialog containing the target Grid. See Example.

`nIDCustom` is the Grid's resource ID, as set using the **`Custom Control Properties`** dialog. See Resource Editor Grid Controls. In that example the resource ID is `IDC_MY_GRID`.

`grid_create()` returns a handle to the Grid being created.

Example

See Example.

7.5.12.8 grid_setup()

See Grid Functions, ONINITDIALOG: Defining the Grid.

Description

The `grid_setup()` function is used define a Grid's initial attributes. Note the following:

- The `grid_setup()` function is executed in the user-written ONINITDIALOG function (see ONINITDIALOG: Defining the Grid).
- A Grid's physical display size is affected by this function and several others. It may be necessary to iteratively adjust the dialog's size and/or the location of other dialog components to obtain the desired visual display. See Note:.
- Grid cells do not change size based on content which is later displayed in that cell. The row width and column height must be sized anticipating the largest string which will be displayed in any cell.

Usage

```
void grid_setup(HWND hGrid,
                int rows,
                int cols,
                int fixed_rows,
                int fixed_cols,
                int rowCharHeight,
                int colCharWidth );
```

where:

`hGrid` is a handle to the target Grid, as returned by `grid_create()`.

`rows` and **`cols`** specifies the number of non-fixed rows and columns in the Grid.

`fixed_rows` and **`fixed_cols`** specifies the number of Fixed Rows and Fixed Columns in the Grid.

rowCharHeight specifies the height of rows, in characters. Row height may subsequently be set in pixel counts using grid_row_pixel_height_set(). Fixed row height may subsequently be set using grid_fixed_row_height_set().

colCharWidth specifies the width of columns, in characters. Column width may subsequently be set in pixel counts using grid_column_pixel_width_set(). Fixed column width may subsequently be set using grid_fixed_col_width_set().

Example

See Example.

7.5.12.9 grid_fixed_col_width_set()

See Grid Functions, ONINITDIALOG: Defining the Grid.

Description

The grid_fixed_col_width_set() function is used to specify the width of a Grid's Fixed Columns. Note the following:

- grid_fixed_col_width_set() is executed in the user-written ONINITDIALOG function, see ONINITDIALOG: Defining the Grid.
- grid_fixed_col_width_set() function is normally executed once for each fixed column.
- If grid_fixed_col_width_set() is not used, the width of any Fixed Columns is set by the colCharWidth argument to grid_setup().
- A Grid's physical display size is affected by this function and several others. It may be necessary to iteratively adjust the dialog's size and/or the location of other dialog components to obtain the desired visual display. See Note:.

Usage

```
void grid_fixed_col_width_set( HWND hGrid, int col, int numChars );
```

where:

hGrid is a handle to the target Grid, as returned by grid_create().

col is the zero-based column index which identifies which of possibly several Fixed Columns is being set.

numChars identifies the width of the fixed column, in characters.

Example

See Example.

7.5.12.10 grid_fixed_row_height_set()

See Grid Functions, ONINITDIALOG: Defining the Grid.

Description

The `grid_fixed_row_height_set()` function is used to specify the height of a Grid's Fixed Rows. Note the following:

- `grid_fixed_row_height_set()` is executed in the user-written ONINITDIALOG function, see ONINITDIALOG: Defining the Grid.
- `grid_fixed_row_height_set()` function is normally executed once for each fixed row.
- If `grid_fixed_row_height_set()` is not used, the width of any Fixed Rows is set by the `rowCharHeight` argument to `grid_setup()`.
- A fixed row's height is affected by the `verticalLabel` argument to `GRID_CONTROL()`; i.e. whether fixed row text is displayed vertically vs. horizontally.
- A Grid's physical display size is affected by this function and several others. It may be necessary to iteratively adjust the dialog's size and/or the location of other dialog components to obtain the desired visual display. See Note:.

Usage

```
void grid_fixed_row_height_set(HWND hGrid,
                               int row,
                               int numChars );
```

where:

hGrid is a handle to the target Grid, as returned by `grid_create()`.

row is the zero-based row index which identifies which of possibly several Fixed Rows is being set.

numChars identifies the height of the fixed row, in characters.

Example

See Example.

7.5.12.11 grid_column_pixel_width_set()

See Grid Functions, ONINITDIALOG: Defining the Grid.

Description

The `grid_column_pixel_width_set()` function is used to specify the width of a Grid's non-fixed columns in pixels. Note the following:

- By default, the width of a Grid's non-fixed column is set by the `colCharWidth` argument to `grid_setup()`. The `grid_column_pixel_width_set()` function may subsequently be used to set the column width in pixels.
- `grid_column_pixel_width_set()` is executed in the user-written `ONINITDIALOG` function, see ONINITDIALOG: Defining the Grid.
- A fixed column's width is not affected by `grid_column_pixel_width_set()`.
- A Grid's physical display size is affected by this function and several others. It may be necessary to iteratively adjust the dialog's size and/or the location of other dialog components to obtain the desired visual display. See Note:.

Usage

```
void grid_column_pixel_width_set(
            HWND hGrid,
            int numpixels,
            BOOL bAdjustRect );
```

where:

hGrid is a handle to the target Grid, as returned by `grid_create()`.

numpixels specifies width of the Grid's columns, in pixels.

bAdjustRect is used for internal purposes. **bAdjustRect** should always be set = TRUE.

Example

The example below shows the ONINITDIALOG function which generated the Grid shown in the bottom dialog in Figure-173: . Note the use of grid_column_pixel_width_set() to set the Grid's non-fixed columns width and the use of grid_row_pixel_height_set() to set the Grid's non-fixed row height:

```
HWND hPixelBarGraph; // Handle to Grid

void PixelBarGraphCreate ( BOOL created ){
  if (!created) return;

  // Get handle to Dialog containing Grid
  hPixelBarGraph = get_HWND( PixelBarGraph_Dialog );

  // Create a specific Grid and get a handle to it
  hPixelBarGraph = grid_create( hPixelBarGraph, // grid_create()
                                IDC_PIXEL_BAR_GRAPH );

  // Set required Grid attributes
  grid_setup( hPixelBarGraph, 50, 10, 2, 0, 1, 1); // grid_setup()

  // Set height for two Fixed Rows
  grid_fixed_row_height_set( hPixelBarGraph, 0, 1 );
  grid_fixed_row_height_set( hPixelBarGraph, 1, 1 );

  // Reset height of rows in pixels, grid_row_pixel_height_set()
  grid_row_pixel_height_set( hPixelBarGraph, 7, TRUE );

  // Reset width of columns in pixels
  grid_column_pixel_width_set( hPixelBarGraph, 40, TRUE );

  grid_initialize( hPixelBarGraph ); // grid_initialize()
}
```

7.5.12.12 grid_row_pixel_height_set()

See Grid Functions, ONINITDIALOG: Defining the Grid.

Description

The grid_row_pixel_height_set() function is used to specify the height of a Grid's non-fixed rows in pixels. Note the following:

- By default, the height of a Grid's non-fixed rows is set by the `rowCharHeight` argument to `grid_setup()`. The `grid_row_pixel_height_set()` function may subsequently be used to set the row height in pixels.
- `grid_row_pixel_height_set()` is executed in the user-written `ONINITDIALOG` function, see ONINITDIALOG: Defining the Grid.
- A fixed row's height is not affected by `grid_row_pixel_height_set()`.
- A Grid's physical display size is affected by this function and several others. It may be necessary to iteratively adjust the dialog's size and/or the location of other dialog components to obtain the desired visual display. See Note:.

Usage

```
void grid_row_pixel_height_set(
          HWND hGrid,
          int numpixels,
          BOOL bAdjustRect );
```

where:

`hGrid` is a handle to the target Grid, as returned by `grid_create()`.

`numpixels` specifies height of the Grid's rows, in pixels.

`bAdjustRect` is used for internal purposes. **`bAdjustRect`** should always be set = TRUE.

Example

See Example.

7.5.12.13 grid_initialize()

See Grid Functions, ONINITDIALOG: Defining the Grid.

Description

The `grid_initialize()` function is used to initialize a new Grid and update its contents. Note the following:

- `grid_initialize()` is executed once, at the end of the user-written `ONINITDIALOG` function, see ONINITDIALOG: Defining the Grid.

- Executing `grid_initialize()` causes all of the Grid Call-back Functions (registered using the GRID_CONTROL() Macro) which affect Grid cell attributes to be executed, once for each cell in the Grid. This determines, for example, the text and color displayed in each cell. User code will subsequently execute `grid_update()` to modify or refresh the Grid content.

Usage

```
void grid_initialize( HWND hGrid );
```

where **`hGrid`** is a handle to the target Grid, as returned by `grid_create()`.

Example

See Example and Example.

7.5.12.14 grid_update()

See Grid Functions, ONINITDIALOG: Defining the Grid.

Description

The `grid_update()` function is used to refresh a Grid's displayed contents. Note the following:

- `grid_update()` may only be used after a given Grid has been defined and initialized via ONINITDIALOG: Defining the Grid.
- Executing `grid_update()` causes all of the Grid Call-back Functions (registered using the GRID_CONTROL() Macro) which affect Grid cell attributes to be executed, once for each cell in the Grid. This determines, for example, the text and color displayed in each cell. Code in these call-back functions is the only mechanism which allows user code to change the contents or attributes of a Grid's cells (text, color, selected color, etc.).

Usage

```
void grid_update( HWND hGrid );
```

where **`hGrid`** is a handle to the target Grid, as returned by `grid_create()`.

Example

The following example shows the data array displayed in the Grid of the upper-right dialog in Figure-173: . Below the array is the GridCellTextCallback Call-back Function used to display the array contents in the Grid. Last is the GridCellClickedCallback Call-back Function executed any time the user clicks in the Grid. It clears the array location associated with the clicked cell (the cell *with focus*), then executes `grid_update()` to refresh the Grid display; effectively clearing any cell clicked in the main Grid cell area:

```
static CString celldata [num_sites][num_duts] = {
   {"1", "2", "3", "4", "5", "6", "7", "8", "9", "10"},
   {"11", "12", "13", "14", "15", "16", "17", "18", "19", "20"},
   {"21", "22", "23", "24", "25", "26", "27", "28", "29", "30"},
   {"31", "32", "33", "34", "35", "36", "37", "38", "39", "40"},
   {"41", "42", "43", "44", "45", "46", "47", "48", "49", "50"},
   {"??", "??", "??", "??", "??", "??", "??", "??", "??", "??"},
   {"61", "62", "63", "64", "65", "66", "67", "68", "69", "70"},
   {"71", "72", "73", "74", "75", "76", "77", "78", "79", "80"}
};
// Example GridCellTextCallback Call-back Function
CString CellText( GridCell cell ) {
   if ( ( cell.row == 0 ) && ( cell.col == 0 ) )
      return "All";
   if ( ( cell.col == 0 ) && ( cell.row > 0 ) )// Row heading
      return vFormat( "Site%d", cell.row );
   if ( ( cell.row == 0 ) && ( cell.col > 0 ) ) // Col heading
      return vFormat( "Dut %d", cell.col );
   if( cell.col == num_cols - cell.row)
      return "";
   return( celldata[cell.row -1][cell.col -1] );
}
// Example GridCellClickedCallback Call-back Function
// Executed when Grid cell is clicked
void CellClicked( GridCell cell ){
   celldata[cell.row -1][cell.col -1] = "";
   grid_update( hSiteDutGrid );
}
```

7.5.12.15 grid_focus_cell_get()

See Grid Functions, ONINITDIALOG: Defining the Grid.

Description

The `grid_focus_cell_get()` function may be used to get the cell with focus from a specified Grid. Note the following:

- `grid_focus_cell_get()` may only be used after a given Grid has been defined and initialized via ONINITDIALOG: Defining the Grid.

Usage

```
GridCell grid_focus_cell_get( HWND hGrid );
```

where **`hGrid`** is a handle to the target Grid, as returned by `grid_create()`.

`grid_focus_cell_get()` returns the cell with focus as a `GridCell`.

Example

```
GridCell c = grid_focus_cell_get( hPixelBarGraph );
output(" Focus currently at row => %d, col => %d", c.row, c.col );
```

7.5.12.16 grid_reset()

See Grid Functions, ONINITDIALOG: Defining the Grid.

Description

The `grid_reset()` function may be used to clear any Grid selections and move the focus to the first non-fixed cell (upper-left non-fixed cell). Note the following:

- `grid_reset()` may only be used after a given Grid has been defined and initialized via ONINITDIALOG: Defining the Grid.

Usage

```
void grid_reset( HWND hGrid );
```

where **`hGrid`** is a handle to the target Grid, as returned by `grid_create()`.

Example

```
grid_reset( hPixelBarGraph );
```

7.5.12.17 Grid Call-back Functions

See Overview, GRID_CONTROL() Macro.

With one exception (GridCellClickedCallback Call-back Function) the Grid call-back functions are used determine the following attributes displayed in each Grid cell:

Attribute	Call-back Function
Cell Text	GridCellTextCallback Call-back Function
Cell Format	GridCellFormatCallback Call-back Function
Cell Text Color	GridTextColorCallback Call-back Function
Selected Cell Text Color	GridSelectedTextColorCallback Call-back Function
Cell Background Color	GridBackgndColorCallback Call-back Function
Selected Cell Background Color	GridSelectedBackgndColorCallback Call-back Function
Focus Cell Background Color	GridFocusBackgndColorCallback Call-back Function

Review the individual call-back function for more details.

7.5.12.18 GridCellTextCallback Call-back Function

See Overview, Grid Call-back Functions, GRID_CONTROL() Macro.

Description

This call-back allows user-written code to set the Cell Text displayed in the current cell of the Grid being updated. Note the following:

- This is the only mechanism allowing user code to modify a Grid cell's text content.

- The call-back is registered by the `cellTextFunc` argument to the `GRID_CONTROL()` macro.
- If registered, this call-back is executed during `grid_initialize()` and `grid_update()`, once for each cell in the Grid, including cells in Fixed Rows and Fixed Columns.

Usage

The user's call-back function must conform to the following prototype:

```
CString( *GridCellTextCallback )( GridCell );
```

where **`GridCell`** identifies the current cell.

The function must return a `CString`, which defines the Cell Text to be displayed in **`GridCell`**.

Example

The following code uses a two dimensional array to store values displayed in the main cells of the Grid shown in Figure-175: . The code in the `myCellText` call-back function returns different values for the upper-left cell (0/0 = "All"), the Fixed Rows and Fixed Columns, and the cells in the main Grid area. A diagonal of cells is left empty, to demonstrate that the data in the array may not always be used:

```
static CString celldata [8][10] = {
   {"1", "2", "3", "4", "5", "6", "7", "8", "9", "10"},
   {"11", "12", "13", "14", "15", "16", "17", "18", "19", "20"},
   {"21", "22", "23", "24", "25", "26", "27", "28", "29", "30"},
   {"31", "32", "33", "34", "35", "36", "37", "38", "39", "40"},
   {"41", "42", "43", "44", "45", "46", "47", "48", "49", "50"},
   {"??", "??", "??", "??", "??", "??", "??", "??", "??", "??"},
   {"61", "62", "63", "64", "65", "66", "67", "68", "69", "70"},
   {"71", "72", "73", "74", "75", "76", "77", "78", "79", "80"}
};
CString myCellText( GridCell cell ) {
   if ( ( cell.row == 0 ) && ( cell.col == 0 ) )
      return "All";
   if (( cell.col == 0 ) && ( cell.row > 0 )) // Fixed row heading
      return vFormat( "Site%d", cell.row );
   if (( cell.row == 0 ) && ( cell.col > 0 )) // Fixed col heading
      return vFormat( "Dut %d", cell.col );
```

```
   if( cell.col == num_cols - cell.row) // Blank diagonal
      return "";
   return( celldata[cell.row -1][cell.col -1] );
}
```

7.5.12.19 GridCellFormatCallback Call-back Function

See Overview, Grid Call-back Functions, GRID_CONTROL() Macro.

Description

This call-back allows user-written code to set the Cell Format of the current cell of the Grid being updated. This allows the Cell Text to be horizontally justified in the cell. Note the following:

- User code in the call-back must return one of the following, which determines the horizontal justification for any text in the current cell:
  - `DT_CENTER`
  - `DT_LEFT`
  - `DT_RIGHT`

  These values are defined in the Microsoft libraries and are the only values supported by this call-back. Other values are silently ignored.
- If no call-back is registered text is center justified in the main Grid cells and in Fixed Columns.
- This call-back does not affect text in Fixed Rows (column labels)
- This call-back does affect text in Fixed Columns (row labels).
- Cell 0/0 is not affected.
- The call-back is registered by the `cellFormatFunc` argument to the `GRID_CONTROL()` macro.
- If registered, this call-back is executed during `grid_initialize()` and `grid_update()`, once for each cell in the Grid.

Usage

The user's call-back function must conform to the following prototype:

```
UINT( *GridCellFormatCallback )( GridCell );
```

where **GridCell** identifies the current cell.

The function must return a UINT (one of the values noted above), which defines the Cell Format to be applied to **GridCell**.

Example

The following code was used in the Grid example shown in Figure-175: . In that image note that some cells have left-justified text and some have center-justified text or right-justified text:

```
UINT myCellTextFormat ( GridCell cell ){
   int m = (cell.col + cell.row) % 3;
   switch(m) {
      case 0: return DT_CENTER;
      case 1: return DT_LEFT;
      case 2: return DT_RIGHT;
   }
   return DT_CENTER;
}
```

Note that this example isn't much use in the real world, but does demonstrate the related controls.

7.5.12.20 GridTextColorCallback Call-back Function

See Overview, Grid Call-back Functions, GRID_CONTROL() Macro.

Description

This call-back allows user-written code to set the Cell Text Color displayed in the current cell of the Grid being updated. Note the following:

- This call-back affects text in all cells, including in Fixed Rows (column labels) and in Fixed Columns (row labels). Cell 0/0 is affected.
- The Cell Text Color of non-fixed cells may be superseded by Selected Cell Text Color if the current cell is selected and a GridSelectedTextColorCallback Call-back Function is registered.
- The call-back is registered by the `textColorFunc` argument to the `GRID_CONTROL()` macro.

- If registered, this call-back is executed during `grid_initialize()` and `grid_update()`, once for each cell in the Grid, including cells in Fixed Rows and Fixed Columns.

Usage

The user's call-back function must conform to the following prototype:

```
COLORREF( *GridTextColorCallback )( GridCell );
```

where **`GridCell`** identifies the current cell.

The function must return a `COLORREF`, which defines the Cell Text Color to be displayed in **`GridCell`**. This is best done using the `RGB` macro, which takes three integer values ranging from 0 to 255 representing red, green and blue. For example:

```
return RGB( 0, 255, 0 ); // Returns 100% Green
```

Or, the Microsoft `GetSysColor()` function can be used to get the default Windows text color (see Example).

Example

The following code was used in the Grid example shown in Figure-175: . In that image note that diagonal cells have Red text, cells in row 6 have Blue text, and the rest have the Windows default text color:

```
COLORREF myCellTextColor( GridCell cell ){
   if ( cell.row == cell.col ) // Set diagonal cells = Red text
      return RGB( 255, 0, 0 );
   if( cell.row == 6 )         // Set row 6 cells = Blue text
      return RGB( 0, 0, 255 );
   // All other cells get the default Windows text color
   return ( GetSysColor( COLOR_WINDOWTEXT ) );
}
```

7.5.12.21 GridSelectedTextColorCallback Call-back Function

See Overview, Grid Call-back Functions, GRID_CONTROL() Macro.

Description

This call-back allows user-written code to set the Selected Cell Text Color of text in the current cell of the Grid being updated, if the cell is currently selected. Note the following:

- This call-back affects the color of text in non-fixed cells only; i.e. cells in Fixed Rows (column labels) and in Fixed Columns (row labels) are not affected.
- If the current cell is selected the Selected Cell Text Color supersedes a color set using the GridTextColorCallback Call-back Function.
- The call-back is registered by the `selTextColorFunc` argument to the `GRID_CONTROL()` macro.
- If registered, this call-back is executed during `grid_initialize()` and `grid_update()`, once for each non-fixed cell in the Grid.
- The call-back must return the Selected Cell Text Color to be applied to the current cell if it is selected.

Usage

The user's call-back function must conform to the following prototype:

```
COLORREF( *GridSelectedTextColorCallback )( GridCell );
```

where **`GridCell`** identifies the current cell.

The function must return a `COLORREF`, which defines the Selected Cell Text Color to be displayed in **`GridCell`**. This is best done using the `RGB` macro, which takes three integer values ranging from 0 to 255 representing red, green and blue. For example:

```
return RGB( 0, 0, 255 ); // Returns 100% Blue
```

Or the Microsoft `GetSysColor()` function can be used to get the default Windows text color (see Example).

Example

The following code was used in the Grid example shown in Figure-175: . In that image note that the selected cells (magenta and dark blue) have blue text:

```
COLORREF mySelectedCellTextColor ( GridCell cell ){
   return RGB( 0, 0, 255 ); // Selected text = Blue
}
```

7.5.12.22 GridCellClickedCallback Call-back Function

See Overview, Grid Call-back Functions, GRID_CONTROL() Macro.

Description

This call-back allows user-written code to execute when a cell is clicked in the Grid. Note the following:

- If registered, this call-back is executed any time the user clicks a Grid cell.
- The row/column coordinates of the clicked cell (*with focus*) are passed to the call-back function (see Grid Row/Column Numbering). If a cell in Fixed Rows or Fixed Columns is clicked the cell with focus will NOT be the cell which is clicked; it will be the first cell in the row or column.
- The call-back is registered by the `cellClickedFunc` argument to the `GRID_CONTROL()` macro.

Usage

The user's call-back function must conform to the following prototype:

```
void( *GridCellClickedCallback )( GridCell );
```

where **`GridCell`** identifies the clicked cell, see Grid Row/Column Numbering.

Example

The following code depends on the data array used in Example. It outputs the row/column coordinates of a clicked cell (with focus). It then clears the value stored for that selected cell in the data array and updates the Grid, effectively clearing any non-fixed cell which is clicked:

```
void myCellClicked( GridCell cell ){
   output("Clicked: Row %d, Col %d", cell.row, cell.col);
   celldata[cell.row -1][cell.col -1] = "";
   grid_update( hSiteDutGrid ); // grid_update()
}
```

7.5.12.23 GridBackgndColorCallback Call-back Function

See Overview, Grid Call-back Functions, GRID_CONTROL() Macro.

Description

This call-back allows user-written code to set the Cell Background Color of the current cell of the Grid being updated. Note the following:

- This call-back does not affect text in Fixed Rows (column labels) or in Fixed Columns (row labels). Cell 0/0 is not affected.
- The Cell Background Color may be superseded by Selected Cell Background Color if the current cell is selected and a GridSelectedBackgndColorCallback Call-back Function is registered.
- The Cell Background Color may be superseded by Focus Cell Background Color if the current cell has focus and a GridFocusBackgndColorCallback Call-back Function is registered.
- The call-back is registered by the `backColorFunc` argument to the `GRID_CONTROL()` macro.
- If registered, this call-back is executed during `grid_initialize()` and `grid_update()`, once for each non-fixed cell in the Grid.
- The call-back must return the Cell Background Color to be applied to the current cell.

Usage

The user's call-back function must conform to the following prototype:

```
COLORREF( *GridBackgndColorCallback )( GridCell );
```

where **`GridCell`** identifies the current cell.

The function must return a `COLORREF`, which defines the Cell Background Color to be displayed in **`GridCell`**. This is best done using the `RGB` macro, which takes three integer values ranging from 0 to 255 representing red, green and blue. For example:

```
return RGB( 255, 255, 0 ); // Returns Yellow
```

Or the Microsoft `GetSysColor()` function can be used to get the default Windows background color (see Example).

Example

The following code was used in the Grid example shown in Figure-175: . In that image note that the background color of column 5 is green and row 6 is yellow. Also note that the color of the selected cells (magenta) and focus cell (blue) supersedes the colors set using this call-back function:

```
COLORREF myCellBGColor( GridCell cell ){
   if ( cell.col == 5 )        // Col 5 = Green
      return RGB( 0, 255, 0 );
   if( cell.row == 6 )         // Row 6 = Yellow
      return RGB( 255, 255, 0 );
   return ( GetSysColor( COLOR_WINDOW ) );
}
```

7.5.12.24 GridSelectedBackgndColorCallback Call-back Function

See Overview, Grid Call-back Functions, GRID_CONTROL() Macro.

Description

This call-back allows user-written code to set the Selected Cell Background Color of the current cell of the Grid being updated, if the cell is selected. Note the following:

- This call-back does not affect cells in Fixed Rows (column labels) or in Fixed Columns (row labels). Cell 0/0 is not affected.
- The Selected Cell Background Color may be superseded by Focus Cell Background Color if the current cell has focus and a GridFocusBackgndColorCallback Call-back Function is registered.
- The Selected Cell Background Color supersedes the color set by GridBackgndColorCallback Call-back Function.
- The call-back is registered by the `selBackColorFunc` argument to the `GRID_CONTROL()` macro.
- If registered, this call-back is executed during `grid_initialize()` and `grid_update()`, once for each non-fixed cell in the Grid.
- The call-back must return the Selected Cell Background Color to be applied to the current cell if it is selected.

Usage

The user's call-back function must conform to the following prototype:

```
COLORREF( *GridSelectedBackgndColorCallback )( GridCell );
```

where **`GridCell`** identifies the current cell.

The function must return a `COLORREF`, which defines the Selected Cell Background Color to be displayed if **`GridCell`** is selected. This is best done using the `RGB` macro, which takes three integer values ranging from 0 to 255 representing red, green and blue. For example:

```
return RGB( 0, 255, 0 ); // Returns 100% Green
```

Or the Microsoft `GetSysColor()` function can be used to get the default Windows background color (see Example).

Example

The following code was used in the Grid example shown in Figure-175: . In that image note that the background color of the selected cells is magenta and that the focus cell background color (blue-ish) supersedes the colors set using this call-back function. Also note that if the cell at location 2/2 was selected it would display the default Windows background color:

```
COLORREF mySelectedCellBGColor( GridCell cell ){
   if( cell.row == 2 && cell.col == 2)
      return ( GetSysColor( COLOR_WINDOW ) );
   return RGB( 255, 0, 255 ); // Magenta
}
```

7.5.12.25 GridFocusBackgndColorCallback Call-back Function

See Overview, Grid Call-back Functions, GRID_CONTROL() Macro.

Description

This call-back allows user-written code to set the Focus Cell Background Color of the current cell of the Grid being updated, if the cell has focus. Note the following:

- This call-back does not affect cells in Fixed Rows (column labels) or in Fixed Columns (row labels). Cell 0/0 is not affected.

- The Focus Cell Background Color supersedes the color set by a GridBackgndColorCallback Call-back Function or a GridSelectedBackgndColorCallback Call-back Function.
- The call-back is registered by the `focusBackColorFunc` argument to the `GRID_CONTROL()` macro.
- If registered, this call-back is executed during `grid_initialize()` and `grid_update()`, once for each non-fixed cell in the Grid.
- The call-back must return the Focus Cell Background Color to be applied to the current cell if it has focus.

Usage

The user's call-back function must conform to the following prototype:

```
COLORREF( *GridFocusBackgndColorCallback )( GridCell );
```

where **`GridCell`** identifies the current cell.

The function must return a `COLORREF`, which defines the Focus Cell Background Color to be displayed if **`GridCell`** has focus. This is best done using the `RGB` macro, which takes three integer values ranging from 0 to 255 representing red, green and blue. For example:

```
return RGB( 0, 255, 0 ); // Returns 100% Green
```

Or the Microsoft `GetSysColor()` function can be used to get the default Windows background color (see Example).

Example

The following code was used in the Grid example shown in Figure-175: . In that image note that the background color of the selected cells is magenta and that the focus cell background color (Blue-ish) supersedes the colors set using the selected call-back function. Also note that if the cell at location 1/1 had focus it would be set to the default Windows background color:

```
COLORREF mySelectedCellBGColor( GridCell cell ){
   if( cell.row == 1 && cell.col == 1)
      return ( GetSysColor( COLOR_WINDOW ) );
   return RGB( 128, 128, 255 ); // Blue-ish
}
```

7.6 STDF Software

Note: first available in software release h2.3.19 (Magnum 1), i2.3.19 (Magnum ICP), h3.5.xx (Magnum 2/2x).

This section describes the Nextest software which supports the Teradyne *Standard Test Data Format (STDF) Specification*.

This section includes:

- stdf_PMR_add()
- stdf_PRR_add()
- stdf_PTR_add()
- stdf_RDR_add()
- stdf_SBR_add()
- stdf_SDR_add()
- stdf_TSR_add()
- stdf_WCR_add()
- stdf_WIR_add()
- stdf_WRR_add()

7.6.1 Overview

See STDF Software.

The Teradyne *Standard Test Data Format (STDF) Specific*ation defines simple, flexible, portable data format standard targeted at test result data.

The *Standard Test Data Format (STDF) Specification* is documented separately. For convenience, some information from the STDF specification is included in this document, however, it is not complete: the user *must* refer to the STDF specification for complete information, important to any use of STDF.

Note: *Test Data Format (STDF) Specification Version 4* was used as the basis for this document.

From the Nextest software viewpoint, the STDF software operates as follows:

- Each test program will use the STDF File Functions and STDF Record Add Functions as appropriate to log the desired test result data to a user-specified STDF file on disk.
  - The STDF functions do not operate when executed in a Site process: a warning is issued. Conversely, the STDF functions are designed to operate when executed from the Host process (typically via the Host Begin Block) or a user tool process (see User Tools). Data to be logged which exists in a Site process must be transferred to the Host or user tool process using methods which are not covered in this document. Nextest Applications has representative examples.

- The STDF File Functions are used to open, write-to and close an STDF file on disk.
  - The `stdf_file_open()` function is used to open an STDF file on disk. Only one STDF file can be open at a time. Specific file naming rules apply, see `stdf_file_open()` and the Standard Test Data Format (STDF) Specification. When `stdf_file_open()` is executed, presuming the file is open successfully, the required File Attributes Record (FAR) is automatically generated and written to the file; i.e. there are no provisions for user-code to manipulate a FAR record.
  - The `stdf_file_write()` function is used to write any records currently in the STDF record heap (populated using the STDF Record Add Functions, more below) to the currently open STDF file.
  - The `stdf_file_close()` function is used to close the currently open STDF file. If any records exist in the STDF record heap when `stdf_file_close()` is executed they are written to the STDF file before it is closed.
- The STDF Record Add Functions are used to explicitly store an STDF record in Host computer memory. For convenience this is called the *STDF record heap*.
  - As noted above, the first required record (File Attributes Record - FAR) is automatically generated by the STDF software and written to the STDF file first. All other record types are explicitly managed by user-written code.
  - The STDF software automatically adds the required record header to each recorded added to the heap. This typically consists of the REC_LEN, REC_TYP, REC_SUB record fields. See Standard Test Data Format (STDF) Specification.
  - The user is responsible for adding records to the heap in a valid sequence, as defined in the Standard Test Data Format (STDF) Specification.
  - In this document, details of most record types are simplified. DO refer to the Standard Test Data Format (STDF) Specification for important additional details for each record type being used. Many record types have both required information fields and fields which may not be applicable but must be set to a specific value when not needed. These details and other important rules are NOT documented here; see the Standard Test Data Format (STDF) Specification for each record type.
  - It is NOT necessary for an STDF file to be open to add records to the STDF record heap. It is necessary for an STDF file to be open to write the STDF record heap to a disk file. See STDF File Functions.
  - The STDF record heap is cleared each time it is written to the STDF file. Additional records may then be added to the heap, followed by additional writes to STDF file. Etc.

- Any records in the STDF record heap at the time the test program is terminated are *lost*. The user is responsible for saving the record heap to disk file before the program is terminated.
- Some STDF data types used as arguments, or components of arguments, to the STDF Record Add Functions consist of variable-length arrays. See Data Type Codes and Representation. The Standard Test Data Format (STDF) Specification specifies the maximum supported size for each of these variable-length arrays (255 bytes, 65,535 bits, etc.). The STDF Code Example declares arrays using these maximum sizes.
- The arrays used as arguments to the STDF Record Add Functions each have specific requirements for the first array element (array[0]). These values are automatically set by the STDF software..

Note: the arrays used in the STDF software are basic C-language arrays. In use, user-written code must declare and initialize each array before it can be passed to the STDF Record Add Functions. During the process of initializing these arrays it is *very important* to prevent over-running an array boundary, which is always **BAD**. Careless programming can cause incorrect and erratic program operation and can be very difficult to diagnose. This is the user's responsibility.

7.6.2 STDF Record Types

See STDF Software, Overview.

The following table lists (alphabetically) the defined STDF record types. Refer to the Standard Test Data Format (STDF) Specification for additional details:

Table 7.6.2.0-1 STDF Record Types and Add Functions

Token	Record Type	SDTD-Add Function
ATR	Audit Trail Record	`stdf_ATR_add()`
BPS	Begin Program Section Record	`stdf_BPS_add()`
DTR	Datalog Text Record	`stdf_DTR_add()`
EPS	End Program Section Record	`stdf_EPS_add()`

Table 7.6.2.0-1 STDF Record Types and Add Functions

Token	Record Type	SDTD-Add Function
FAR	File Attributes Record	The FAR record is automatically generated by the STDF software and added to the STDF file by stdf_file_open().
FTR	Functional Test Record	stdf_FTR_add()
GDR	Generic Data Record	Generic Data Record (GDR) Functions
HBR	Hardware Bin Record	stdf_HBR_add()
MIR	Master Information Record	stdf_MIR_add()
MPR	Multiple-Result Parametric Record	stdf_MPR_add()
MRR	Master Results Record	stdf_MRR_add()
PCR	Part Count Record	stdf_PCR_add()
PGR	Pin Group Record	stdf_PGR_add()
PIR	Part Information Record	stdf_PIR_add()
PLR	Pin List Record	stdf_PLR_add()
PMR	Pin Map Record	stdf_PMR_add()
PRR	Part Results Record	stdf_PRR_add()
PTR	Parametric Test Record	stdf_PTR_add()
RDR	Retest Data Record	stdf_RDR_add()
SBR	Software Bin Record	stdf_SBR_add()
SDR	Site Description Record	stdf_SDR_add()
TSR	Test Synopsis Record	stdf_TSR_add()
WCR	Wafer Configuration Record	stdf_WCR_add()
WIR	Wafer Information Record	stdf_WIR_add()
WRR	Wafer Results Record	stdf_WRR_add()

7.6.3 Data Type Codes and Representation

See STDF Software, Overview.

The following table duplicates information formally documented in the Standard Test Data Format (STDF) Specification. It is included here for convenience only. The codes shown below are used when describing arguments to the STDF Record Add Functions. Some of the data types below warrant special attention: see Note::

Table 7.6.3.0-1 Data Type Codes and Representation

Code	Description	C Type Specifier
C*12	Fixed length character string (this example has length = 12 but any length is supported). If a fixed length character string does not fill the entire field, it must be left-justified and padded with spaces.	char[12]
C*n	Variable length character string where **n** defines the length. The first byte = unsigned count of bytes to follow (maximum of 255 bytes). *Important*: see Note:.	char[n]
C*f	Variable length character string where **f** defines the length. The string length is stored in another record field. *Important*: see Note:.	char[f]
U*1	One byte unsigned integer.	unsigned char
U*2	Two byte unsigned integer.	unsigned short
U*4	Four byte unsigned integer.	unsigned long
I*1	One byte signed integer.	char
I*2	Two byte signed integer.	short
I*4	Four byte signed integer.	long
R*4	Four byte floating point number.	float
R*8	Eight byte floating point number.	long float (double)
B*6	Fixed length bit-encoded data (this example has length = 6 but any length is supported).	char[6]

Table 7.6.3.0-1 Data Type Codes and Representation

Code	Description	C Type Specifier
V*n	Variable data type field. The data type is specified by a code in the first byte, and the data follows (maximum of 255 bytes). *Important*: see Note:.	
B*n	Variable length bit-encoded field. First byte = unsigned count of bytes to follow (maximum of 255 bytes). The first data item is in the least significant bit of the second byte of the array. *Important*: see Note:.	char[]
D*n	Variable length bit-encoded field. First two bytes = unsigned count of bits to follow (maximum of 65,535 bits). The first data item is in the least significant bit of the third byte of the array. Unused bits in the high end of the last byte must be zero. *Important*: see Note:.	char[]
N*1	Unsigned integer data stored in a nibble (nibble = 4 bits of a byte). The first item is in the low 4 bits (low nibble), the second item in the high 4 bits. If an odd number of nibbles is indicated, the high nibble of the byte must be zero. Only whole bytes can be written to the STDF file.	char
kxTYPE jxTYPE	Array of data of the type specified. The value of **k** or **j** is the number of elements in the array, which is defined in an earlier field in the record. For example, an array of unsigned short integers is defined as kxU*2 or jxU*2.	TYPE[]

7.6.4 STDF File Functions

See STDF Software, Overview

This section documents the functions used to open, write to and close an STDF file. See Overview.

This section includes:

- stdf_file_open()

- stdf_file_write()
- stdf_file_close()

7.6.4.1 stdf_file_open()

See STDF File Functions, Overview.

Description

The stdf_file_open() function is used to open an STDF file on disk, in preparation for writing STDF records from the STDF record heap to the file using stdf_file_write(). See Overview.

Note the following:

- stdf_file_open() will operate only when executed in the Host process or user tool process. See Overview.
- Only one (1) STDF file can be open at a time. Executing stdf_file_open() when an STDF file is already open generates a warning but is otherwise ignored.
- No file clobber checks are made; i.e. any existing file with the specified file name will be silently over-written.
- When stdf_file_open() is executed, presuming the file is open successfully, the required File Attributes Record (FAR) is automatically generated and written to the file. There are no provisions for user-code to manipulate the FAR record.
- The fname argument identifies the target file name to be opened. fname may include a drive letter, folder, and file name; i.e. an absolute path/filename similar to *d:/myFolder/mySTDFfile*. If an absolute path is not specified, the file will be located relative to the test program executable file location.

Note: the following file name rules are excerpts from the Standard Test Data Format (STDF) Specification. The information below is simplified to include only information which applies when using the Windows operating system. Additional information and important rules are documented in the *STDF Filenames* section of the Standard Test Data Format (STDF) Specification.

- An STDF file name must have the following format:

```
filename.STD[string]
```

where

`filename` is any string consisting of 1 to 39 of the ASCII characters A - Z, a - z, and 0 - 9, plus the underscore (_). The first character must be alphabetic.

`.STD[string]` is a string beginning with the characters *.STD*, and continuing with characters that are legal for **`filename`**. The **`string`** cannot be longer than 39 characters.

Also note:

- In earlier versions of the STDF specification, the dollar sign ($) was a legal filename character. It is no longer supported, because its use is incompatible with certain operating systems.
- The STDF filename can contain only a single period.

Usage

```
bool stdf_file_open( char* fname );
```

where **`fname`** identifies the target file to be opened. See Description for important rules.

`stdf_file_open()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.4.2 stdf_file_write()

See STDF File Functions, Overview.

Description

The `stdf_file_write()` function is used to write STDF records from the STDF record heap to an STDF file, previously opened using `stdf_file_open()`. Note the following:

- `stdf_file_write()` will operate only when executed in the Host process or user tool process. See Overview.

- `stdf_file_write()` requires that an STDF file have been previously opened using `stdf_file_open()`. Executing `stdf_file_write()` without an open STDF file generates a warning but is otherwise ignored (no STDF records in the STDF record heap are lost).
- `stdf_file_write()` writes all records currently in the STDF record heap to the open STDF file. See Overview. This clears the STDF record heap; additional records may then be added to the heap, followed by additional writes to STDF file. Etc.
- `stdf_file_close()` also writes all records currently in the STDF record heap to the open STDF file then closes the file.

Usage

```
bool stdf_file_write();
```

`stdf_file_write()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.4.3 stdf_file_close()

See STDF File Functions, Overview.

Description

The `stdf_file_close()` function is used to close the currently open STDF file. If any STDF records remain in the STDF record heap they are written to the STDF file before it is closed. Note the following:

- `stdf_file_close()` will operate only when executed in the Host process or user tool process. See Overview.
- `stdf_file_close()` requires that an STDF file have been previously opened using `stdf_file_open()`. Executing `stdf_file_close()` without an open STDF file generates a warning but is otherwise ignored (no STDF records in the STDF record heap are lost).

Usage

```
bool stdf_file_close();
```

`stdf_file_close()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5 STDF Record Add Functions

See STDF Software, Overview

This section documents the functions used to add STDF records to the STDF record heap in preparation for writing them to an STDF file. See Overview.

Separate functions are used to add each record type, however, for the most part these functions operate identically: the main difference is the record type being added, which determines the arguments for each function.

These functions are documented in alphabetical order. However, the user is responsible to adding records to the STDF record heap in a valid sequence, as defined in the Standard Test Data Format (STDF) Specification.

This section includes the following:

- `stdf_ATR_add()` = add a ATR record to the STDF record heap.
- `stdf_BPS_add()` = add a BPS record to the STDF record heap.
- `stdf_DTR_add()` = add a DTR record to the STDF record heap.
- `stdf_EPS_add()` = add a EPS record to the STDF record heap.
- `stdf_FTR_add()` = add a FTR record to the STDF record heap.
- Generic Data Record (GDR) Functions = add a GDR record to the STDF record heap.
- `stdf_HBR_add()` = add a HBR record to the STDF record heap.
- `stdf_MIR_add()` = add a MIR record to the STDF record heap.
- `stdf_MPR_add()` = add a MPR record to the STDF record heap.
- `stdf_MRR_add()` = add a MRR record to the STDF record heap.
- `stdf_PCR_add()` = add a PCR record to the STDF record heap.

- `stdf_PGR_add()` = add a PGR record to the STDF record heap.
- `stdf_PIR_add()` = add a PIR record to the STDF record heap.
- `stdf_PLR_add()` = add a PLR record to the STDF record heap.
- `stdf_PMR_add()` = add a PMR record to the STDF record heap.
- `stdf_PRR_add()` = add a PRR record to the STDF record heap.
- `stdf_PTR_add()` = add a PTR record to the STDF record heap.
- `stdf_RDR_add()` = add a RDR record to the STDF record heap.
- `stdf_SBR_add()` = add a SBR record to the STDF record heap.
- `stdf_SDR_add()` = add a SDR record to the STDF record heap.
- `stdf_TSR_add()` = add a TSR record to the STDF record heap.
- `stdf_WCR_add()` = add a WCR record to the STDF record heap.
- `stdf_WIR_add()` = add a WIR record to the STDF record heap.
- `stdf_WRR_add()` = add a WRR record to the STDF record heap.

7.6.5.1 stdf_ATR_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_ATR_add()` function is used to add an ATR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Used to record any operation that alters the contents of the STDF file. The name of the program and all its parameters should be recorded in the ASCII field provided in this record* [the `cmd_line` argument]*. Typically, this record will be used to track filter programs that have been applied to the data. Marks the beginning of a new program section (or sequencer) in the job plan.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_ATR_add( unsigned long MOD_TIM, char *CMD_LINE );
```

where:

MOD_TIM specifies the date and time of STDF file modification. The data type of this argument = U*4 in Data Type Codes and Representation.

CMD_LINE specifies the "*Command line of program*" (per the STDF specification). The data type of this argument = C*n in Data Type Codes and Representation. *Important*: see Note:.

`stdf_ATR_add()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.2 stdf_BPS_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_BPS_add()` function is used to add a BPS record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Marks the beginning of a new program section (or sequencer) in the job plan.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_BPS_add( char *CMD_LINE );
```

where **CMD_LINE** identifies the "*Program section (or sequencer) name*" (per the STDF specification). The data type of this argument = C*n in Data Type Codes and Representation. *Important*: see Note:.

`stdf_BPS_add()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.3 stdf_DTR_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_DTR_add()` function is used to add a DTR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Contains text information that is to be included in the datalog printout. DTRs may be written under the control of a job plan: for example, to highlight unexpected test results. They may also be generated by the tester executive software: for example, to indicate that the datalog sampling rate has changed. DTRs are placed as comments in the datalog listing.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_DTR_add( char *TEXT_DAT );
```

where **TEXT_DAT** identifies the "*ASCII text string*" (per the STDF specification). The data type of this argument = `C*n` in Data Type Codes and Representation. *Important*: see Note:.

`stdf_DTR_add()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.4 stdf_EPS_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_EPS_add()` function is used to add a EPS record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Marks the end of the current program section (or sequencer) in the job plan.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_EPS_add( void );
```

`stdf_EPS_add()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.5 stdf_FTR_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_FTR_add()` function is used to add a FTR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Contains the results of the single execution of a functional test in the test program. The first occurrence of this record also establishes the default values for all semi-static*

information about the test. The FTR is related to the Test Synopsis Record (TSR) by test number, head number, and site number.

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_FTR_add( FTRBlock data );
```

where **data** is a user-defined FTRBlock variable initialized to define the various parameters which describe an FTR record, as shown below. Some fields are required, others may not be applicable but must be set to a specific value when not needed, see Standard Test Data Format (STDF) Specification:

struct FTRBlock {	Standard Test Data Format (STDF) Specification Description	Data Type in Data Type Codes and Representation
float TEST_NUM;	Test number	U*4
unsigned char HEAD_NUM ;	Test head number	U*1
unsigned char SITE_NUM;	Test site number	U*1
char *TEST_FLG;	Test flags (fail, alarm, etc.)	B*1 (see B*6)
char *OPT_FLAG;	Optional data flag	B*1 (see B*6)
float CYCL_CNT;	Cycle count of vector	U*4
float REL_VADR;	Relative vector address	U*4
float REPT_CNT;	Repeat count of vector	U*4
float NUM_FAIL;	Number of pins with 1 or more failures	U*4
int XFAIL_AD;	X logical device failure address	I*4
int YFAIL_AD;	Y logical device failure address	I*4
int VECT_OFF;	Offset from vector of interest	I*2
unsigned int RTN_ICNT;	Count (j) of return data PMR indexes	U*2
unsigned int PGM_ICNT;	Count (k) of programmed state indexes	U*2
unsigned int *RTN_INDX;	Array of return data PMR indexes	jxU*2 (see jxTYPE)
char *RTN_STAT;	Array of returned states	jxN*1 (see jxTYPE)
unsigned int *PGM_INDX;	Array of programmed state indexes	kxU*2 (see kxTYPE)

`unsigned char *PGM_STAT;`	Array of programmed states	`k`xN*1 (see kxTYPE)
`int FAIL_PIN_number_of_chars;`	Number of characters in FAIL_PIN.	I*4
`char *FAIL_PIN;`	Failing pin bitfield. A char[] array containing up to 8198 elements.	D*n
`char *VECT_NAM;`	Vector module pattern name	C*n
`char *TIME_SET;`	Time set name	"
`char *OP_CODE;`	Vector Op Code	"
`char *TEST_TXT;`	Descriptive text or label	"
`char *ALARM_ID;`	Name of alarm	"
`char *PROG_TXT;`	Additional programmed information	"
`char *RSLT_TXT;`	Additional result information	"
`unsigned char PATG_NUM;`	Pattern generator number	U*1
`int SPIN_MAP_number_of_chars;`	Number of characters stored in SPIN_MAP.	I*4
`unsigned char *SPIN_MAP;`	Bit map of enabled comparators. A char[] array containing up to 8198 elements.	D*n

```
}
```

`stdf_FTR_add()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.6 Generic Data Record (GDR) Functions

See STDF Record Add Functions, Overview.

Description

These functions are used to assemble a GDR record and write it to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Contains information that does not conform to any other record type defined by the STDF specification. Such records are intended to be written under the control of job plans executing on the tester. This data may be used for any purpose that the user desires.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

The general model for assembling a GDR record is to use the various _add() functions as desired to assemble one GDR record in the STDF record heap (separate _add() functions are used to allow a given GDR record to contain a user-defined number of values of different data types). Once the record is assembled the stdf_GDR_write_record() function is used to write it to the STDF file.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_GDR_unsigned_byte_add( int bytes, int data );
bool stdf_GDR_signed_byte_add( int bytes, int data );
bool stdf_GDR_floating_point_add( float data );
bool stdf_GDR_double_add( double data );
bool stdf_GDR_char_add( int number_of_chars, char *data );
bool stdf_GDR_binary_add( int bytes, char *data );
bool stdf_GDR_bit_encoded_add( int bytes, char *data );
bool stdf_GDR_nybble_add( char data );
bool stdf_GDR_write_record();
```

where:

bytes is used is two contexts:

- Using stdf_GDR_unsigned_byte_add() and stdf_GDR_signed_byte_add(), **bytes** is used because the data type being added has more than one supported size (see Data Type Codes and Representation). In this context, legal values are 1, 2 and 4. For example, 3 sizes of unsigned ints are supported (U*1, U*2, U*4). Thus, when using stdf_GDR_unsigned_byte_add(), the value 2 is specified for the **bytes** argument when adding a 2 byte int (U*2) value to the GDR record.
- Using stdf_GDR_binary_add() and stdf_GDR_bit_encoded_add(), **bytes** specifies the number of values in the **data** array to be added.

data specifies the value(s) to be added to the GDR record. When the **data** argument is a pointer (i.e. *data) it represents an array of values of the specified type.

number_of_chars is used when adding a character string to the GDR record. It specifies the number of characters in the **data** array to be added.

These functions return TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.7 stdf_HBR_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_HBR_add()` function is used to add a HBR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Stores a count of the parts "physically" placed in a particular bin after testing. (In wafer testing, "physical" binning is not an actual transfer of the chip, but rather is represented by a drop of ink or an entry in a wafer map file.) This bin count can be for a single test site (when parallel testing) or a total for all test sites. The STDF specification also supports a Software Bin Record (SBR) for logical binning categories. A part is "physically" placed in a hardware bin after testing. A part can be "logically" associated with a software bin during or after testing.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_HBR_add(  unsigned char HEAD_NUM,
                    unsigned char SITE_NUM,
                    unsigned short HBIN_NUM,
                    unsigned int HBIN_CNT,
                    char HBIN_PF,
                    char *HBIN_NAM );
```

where:

HEAD_NUM identifies the "*Test head number*" (per the STDF specification). The data type of this argument = U*1 in Data Type Codes and Representation.

SITE_NUM identifies the "*Test site number*" (per the STDF specification). The data type of this argument = U*1 in Data Type Codes and Representation.

HBIN_NUM identifies the "*Hardware bin number*" (per the STDF specification). The data type of this argument = U*2 in Data Type Codes and Representation.

HBIN_CNT identifies the "*Number of parts in bin*" (per the STDF specification). The data type of this argument = U*4 in Data Type Codes and Representation.

HBIN_PF identifies the "*Pass/fail indication*" (per the STDF specification). The data type of this argument = C*1 in Data Type Codes and Representation (see C*12).

HBIN_NAM identifies the "*Name of hardware bin*" (per the STDF specification). The data type of this argument = C*n in Data Type Codes and Representation. *Important*: see Note:.

stdf_HBR_add() returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.8 stdf_MIR_add()

See STDF Record Add Functions, Overview.

Description

The stdf_MIR_add() function is used to add a MIR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *The MIR and the MRR (Master Results Record) contain all the global information that is to be stored for a tested lot of parts. Each data stream must have exactly one MIR, immediately after the FAR (and the ATRs, if they are used). This will allow any data reporting or analysis programs access to this information in the shortest possible amount of time.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_MIR_add( MIRBlock data );
```

where:

where **data** is a user-defined MIRBlock variable initialized to define the various parameters which describe an MIR record, as shown below. Some fields are required, others may not be applicable but must be set to a specific value when not needed, see Standard Test Data Format (STDF) Specification:

struct MIRBlock {	Standard Test Data Format (STDF) Specification Description	Data Type in Data Type Codes and Representation
unsigned long SETUPTIME;	Date and time of job setup	U*4
unsigned long FIRSTPARTTESTTIME;	Date and time first part tested	U*4
unsigned char STATIONNUMBER;	Tester station number	U*1
char MODE_COD;	Test mode code (e.g. prod, dev)	C*1 (see C*12)
char RTST_COD;	Lot retest code	C*1 (see C*12)
char PROT_COD;	Data protection code	C*1 (see C*12)
unsigned short BURN_TIM;	Burn-in time (in minutes)	U*2
char CMOD_COD;	Command mode code	C*1 (see C*12)
char LOT_ID[255];	Lot ID (customer specified)	C*n
char PART_TYP[255];	Part Type (or product ID)	"
char NODE_NAM[255];	Name of node that generated data	"
char TSTR_TYP[255];	Tester type	"
char JOB_NAM[255];	Job name (test program name)	"
char JOB_REV[255];	Job (test program) revision number	"
char SBLOT_ID[255];	Sublot ID	"
char OPER_NAM[255];	Operator name or ID (at setup time)	"
char EXEC_TYP[255];	Tester executive software type	"
char EXEC_VER[255];	Tester exec software version number	"
char TEST_COD[255];	Test phase or step code	"
char TST_TEMP[255];	Test temperature	"
char USER_TXT[255];	Generic user text	"
char AUX_FILE[255];	Name of auxiliary data file	"

```
   char PKG_TYP[255];              Package type                          "
   char FAMLY_ID[255];             Product family ID                     "
   char DATE_COD[255];             Date code                             "
   char FACIL_ID[255];             Test facility ID                      "
   char FLOOR_ID[255];             Test floor ID                         "
   char PROC_ID[255];              Fabrication process ID                "
   char OPER_FRQ[255];             Operation frequency or step           "
   char SPEC_NAM[255];             Test specification name               "
   char SPEC_VER[255];             Test specification version number     "
   char FLOW_ID[255];              Test flow ID                          "
   char SETUP_ID[255];             Test setup ID                         "
   char DSGN_REV[255];             Device design revision                "
   char ENG_ID[255];               Engineering lot ID                    "
   char ROM_COD[255];              ROM code ID                           "
   char SERL_NUM[255];             Tester serial number                  "
   char SUPR_NAM[255];             Supervisor name or ID                 "
}
```

`stdf_MIR_add()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.9 stdf_MPR_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_MPR_add()` function is used to add a MPR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Contains the results of a single execution of a parametric test in the test program where that test returns multiple values. The first occurrence of this record also establishes the default values for all semi-static information about the test, such as limits, units, and scaling. The MPR is related to the Test Synopsis Record (TSR) by test number, head number, and site number.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_MPR_add( MPRBlock data );
```

where **data** is a user-defined MPRBlock variable initialized to define the various parameters which describe an MPR record, as shown below. Some fields are required, others may not be applicable but must be set to a specific value when not needed, see Standard Test Data Format (STDF) Specification:

struct MPRBlock {	Standard Test Data Format (STDF) Specification Description	Data Type in Data Type Codes and Representation
unsigned long TEST_NUM;	Test number	U*4
unsigned char HEAD_NUM;	Test head number	U*1
unsigned char SITE_NUM;	Test site number	U*1
unsigned char TEST_FLG;	Test flags (fail, alarm, etc.)	B*1 (see B*n)
unsigned char PARM_FLG;	Parametric test flags (drift, etc.)	B*1 (see B*n)
short RTN_ICNT;	Count (j) of PMR indexes See note	U*2
short RSLT_CNT;	Count (k) of returned results See note	U*2
unsigned char *RTN_STAT;	Array of returned states RTN_ICNT = 0	jxN*1 (see kxTYPE)
double *RTN_RSLT;	Array of returned results RSLT_CNT = 0	kxR*4 (see kxTYPE)
char TEST_TXT[255];	Test description text or label length byte = 0	C*n
char ALARM_ID[255];	Name of alarm length byte = 0	C*n
unsigned char OPT_FLAG;	Optional data flag See note	B*1 (see B*n)
int RES_SCAL;	Test results scaling exponent OPT_FLAG bit 0 = 1	I*1
int LLM_SCAL;	Low limit scaling exponent OPT_FLAG bit 4 or 6 = 1	I*1
int HLM_SCAL;	High limit scaling exponent OPT_FLAG bit 5 or 7 = 1	I*1
double LO_LIMIT;	Low test limit value OPT_FLAG bit 4 or 6 = 1	R*4
double HI_LIMIT;	High test limit value OPT_FLAG bit 5 or 7 = 1	R*4
double START_IN;	Starting input value (condition) OPT_FLAG bit 1 = 1	R*4

`double INCR_IN;`	Increment of input condition OPT_FLAG bit 1 = 1	R*4
`short *RTN_INDX;`	Array of PMR indexes RTN_ICNT = 0	jxU*2 (see kxTYPE)
`char UNITS[255];`	Test units length byte = 0	C*n
`char C_RESFMT[255];`	ANSI C result format string length byte = 0	"
`char C_LLMFMT[255];`	ANSI C low limit format string length byte = 0	"
`char C_HLMFMT[255];`	ANSI C high limit format string length byte = 0	"
`double LO_SPEC;`	Low specification limit value OPT_FLAG bit 2 = 1	R*4
`double HI_SPEC;`	High specification limit value OPT_FLAG bit 3 = 1	R*4

```
};
```

`stdf_MPR_add()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.10 stdf_MRR_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_MRR_add()` function is used to add a MRR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *The Master Results Record (MRR) is a logical extension of the Master Information Record (MIR). The data can be thought of as belonging with the MIR, but it is not available when the tester writes the MIR information. Each data stream must have exactly one MRR as the last record in the data stream.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_MRR_add(  unsigned long FINISH_T,
                    char DISP_COD,
                    char *USR_DESC,
                    char *EXC_DESC );
```

where:

FINISH_T specifies the "*Date and time last part tested*" (per the STDF specification). The data type of this argument = U*4 in Data Type Codes and Representation.

DISP_COD specifies the "*Lot disposition code*" (per the STDF specification). The data type of this argument = C*1 in Data Type Codes and Representation (see C*12).

USR_DESC specifies the "*Lot description supplied by user*" (per the STDF specification). The data type of this argument = C*n in Data Type Codes and Representation. *Important*: see Note:.

EXC_DESC specifies the "*Lot description supplied by exec*" (per the STDF specification). The data type of this argument = C*n in Data Type Codes and Representation. *Important*: see Note:.

stdf_MRR_add() returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.11 stdf_PCR_add()

See STDF Record Add Functions, Overview.

Description

The stdf_PCR_add() function is used to add a PCR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Contains the part count totals for one or all test sites. Each data stream must have at least one PCR to show the part count.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_PCR_add(  unsigned char HBIN_NAM,
                    unsigned char SITE_NUM,
                    unsigned int PART_CNT,
                    unsigned int RTST_CNT,
                    unsigned int ABRT_CNT,
                    unsigned int GOOD_CNT,
                    unsigned int FUNC_CNT );
```

where:

HBIN_NAM specifies the "*Test head number*" (per the STDF specification). The data type of this argument = U*1 in Data Type Codes and Representation.

SITE_NUM specifies the "*Test site number*" (per the STDF specification). The data type of this argument = U*1 in Data Type Codes and Representation.

PART_CNT specifies the "*Number of parts tested*" (per the STDF specification). The data type of this argument = U*4 in Data Type Codes and Representation.

RTST_CNT specifies the "*Number of parts retested*" (per the STDF specification). The data type of this argument = U*4 in Data Type Codes and Representation.

ABRT_CNT specifies the "*Number of aborts during*" (per the STDF specification). The data type of this argument = U*4 in Data Type Codes and Representation.

GOOD_CNT specifies the "*Number of good (passed) parts tested*" (per the STDF specification). The data type of this argument = U*4 in Data Type Codes and Representation.

FUNC_CNT specifies the "*Number of functional parts tested*" (per the STDF specification). The data type of this argument = U*4 in Data Type Codes and Representation.

`stdf_PCR_add()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.12 stdf_PGR_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_PGR_add()` function is used to add a PGR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Associates a name with a group of pins. See "Using the Pin Mapping Records".*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_PGR_add(  short GRP_INDX,
                    char *GRP_NAM,
                    short INDX_CNT,
                    short *PMR_INDX);
```

where:

`GRP_INDX` specifies the "*Unique index associated with pin group*" (per the STDF specification). The data type of this argument = `U*2` in Data Type Codes and Representation.

`GRP_NAM` specifies the "*Name of pin group*" (per the STDF specification). The data type of this argument = `C*n` in Data Type Codes and Representation. *Important*: see Note:.

`INDX_CNT` specifies the "*Count (k) of PMR indexes*" (per the STDF specification). The data type of this argument = `U*2` in Data Type Codes and Representation.

`PMR_INDX` specifies the "*Array of indexes for pins in the group*" (per the STDF specification). The data type of this argument = `kxU*2` in Data Type Codes and Representation (see `kxTYPE`).

`stdf_PGR_add()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.13 stdf_PIR_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_PIR_add()` function is used to add a PIR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Acts as a marker to indicate where testing of a particular part begins for each part tested by the test program. The PIR and the Part Results Record (PRR) bracket all the stored information pertaining to one tested part.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_PIR_add( int HEADNUM, int SITENUM );
```

where:

`HEADNUM` specifies the "*Test head number*" (per the STDF specification). The data type of this argument = `U*1` in Data Type Codes and Representation.

`SITENUM` specifies the "*Test site number*" (per the STDF specification). The data type of this argument = `U*1` in Data Type Codes and Representation.

`stdf_PIR_add()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.14 stdf_PLR_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_PLR_add()` function is used to add a PLR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Defines the current display radix and operating mode for a pin or pin group. See "Using the Pin Mapping Records".*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_PLR_add(  short GRP_CNT,
                    short *GRP_INDX,
                    short *GRP_MODE,
                    unsigned char *GRP_RADIX,
                    char *PGM_CHAR,
                    char *RTN_CHAR,
                    char *PRGM_CHAL,
                    char *RTN_CHAL );
```

where:

GRP_CNT specifies the "*Count (k) of pins or pin groups*" (per the STDF specification). The data type of this argument = `U*2` in Data Type Codes and Representation.

GRP_INDX specifies the "*Array of pin or pin group indexes*" (per the STDF specification). The data type of this argument = `kxU*2` in Data Type Codes and Representation (see `kxTYPE`).

GRP_MODE specifies the "*Operating mode of pin group*" (per the STDF specification). The data type of this argument = `kxU*2` in Data Type Codes and Representation (see `kxTYPE`).

GRP_RADIX specifies the "*Display radix of pin group*" (per the STDF specification). The data type of this argument = `kxU*1` in Data Type Codes and Representation (see `kxTYPE`).

PGM_CHAR specifies the "*Program state encoding characters*" (per the STDF specification). The data type of this argument = kxC*n in Data Type Codes and Representation (see kxTYPE).

RTN_CHAR specifies the "*Return state encoding characters*" (per the STDF specification). The data type of this argument = kxC*n in Data Type Codes and Representation (see kxTYPE).

PRGM_CHAL specifies the "*Program state encoding characters*" (per the STDF specification). The data type of this argument = kxC*n in Data Type Codes and Representation (see kxTYPE).

RTN_CHAL specifies the "*Return state encoding characters* " (per the STDF specification). The data type of this argument = kxC*n in Data Type Codes and Representation (see kxTYPE).

stdf_PLR_add() returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.15 stdf_PMR_add()

See STDF Record Add Functions, Overview.

Description

The stdf_PMR_add() function is used to add a PMR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Provides indexing of tester channel names, and maps them to physical and logical pin names. Each PMR defines the information for a single channel/pin combination. See "Using the Pin Mapping Records".*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_PMR_add(  unsigned short PMR_INDX,
                    unsigned short CHAN_TYP,
                    char *CHAN_NAM,
                    char *PHY_NAM,
                    char *LOG_NAM,
                    unsigned char HEAD_NUM,
                    unsigned char SITE_NUM );
```

where:

PMR_INDX specifies the "*Unique index associated with pin*" (per the STDF specification). The data type of this argument = U*2 in Data Type Codes and Representation.

CHAN_TYP specifies the "*Channel type*" (per the STDF specification). The data type of this argument = U*2 in Data Type Codes and Representation.

CHAN_NAM specifies the "*Channel name*" (per the STDF specification). The data type of this argument = C*n in Data Type Codes and Representation. *Important*: see Note:.

PHY_NAM specifies the "*Physical name of pin*" (per the STDF specification). The data type of this argument = C*n in Data Type Codes and Representation. *Important*: see Note:.

LOG_NAM specifies the "*Logical name of pin*" (per the STDF specification). The data type of this argument = C*n in Data Type Codes and Representation. *Important*: see Note:.

HEAD_NUM specifies the "*Head number associated with channel*" (per the STDF specification). The data type of this argument = U*1 in Data Type Codes and Representation.

SITE_NUM specifies the "*Site number associated with channel*" (per the STDF specification). The data type of this argument = U*1 in Data Type Codes and Representation.

stdf_PMR_add() returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.16 stdf_PRR_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_PRR_add()` function is used to add a PRR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Contains the result information relating to each part tested by the test program. The PRR and the Part Information Record (PIR) bracket all the stored information pertaining to one tested part.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_PRR_add( PRRBlock data );
```

where **`data`** is a user-defined `PRRBlock` variable initialized to define the various parameters which describe an PRR record, as shown below. Some fields are required, others may not be applicable but must be set to a specific value when not needed, see Standard Test Data Format (STDF) Specification:

```
struct PRRBlock {
   unsigned char HEAD_NUM;
   unsigned char SITE_NUM;
   char *PART_FLG;

   int NUM_TEST;
   int HARD_BIN;
   int SOFT_BIN;
   int X_COORD;
   int Y_COORD;
   unsigned long TEST_T;
```

Standard Test Data Format (STDF) Specification Description	Data Type in Data Type Codes and Representation
Test head number	U*1
Test site number	U*1
Part information flag	B*1 (see B*n)
Number of tests executed	U*2
Hardware bin number	U*2
Software bin number	U*2
(Wafer) X coordinate	I*2
(Wafer) Y coordinate	I*2
Elapsed test time in milliseconds	U*4

```
   char *PART_ID;                Part identification          C*n
   char *PART_TXT;               Part description text         "
   char *PART_FIX;               Part repair information      B*n
}
```

`stdf_PRR_add()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.17 stdf_PTR_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_PTR_add()` function is used to add a PTR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Contains the results of a single execution of a parametric test in the test program. The first occurrence of this record also establishes the default values for all semi-static information about the test, such as limits, units, and scaling. The PTR is related to the Test Synopsis Record (TSR) by test number, head number, and site number.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_PTR_add( PTRBlock data );
```

where **`data`** is a user-defined `PTRBlock` variable initialized to define the various parameters which describe an PTR record, as shown below. Some fields are required,

others may not be applicable but must be set to a specific value when not needed, see Standard Test Data Format (STDF) Specification:

	Standard Test Data Format (STDF) Specification Description	Data Type in Data Type Codes and Representation
`struct PTRBlock {`		
`unsigned long TEST_NUM;`	Test number	U*4
`unsigned char HEAD_NUM;`	Test head number	U*1
`unsigned char SITE_NUM;`	Test site number	U*1
`unsigned char TEST_FLG;`	Test flags (fail, alarm, etc.)	B*1 (see B*6)
`unsigned char PARM_FLG;`	Parametric test flags (drift, etc.)	B*1 (see B*6)
`float RESULT;`	Test result	R*4
`char *TEST_TXT;`	Test description text or label	C*n
`char *ALARM_ID;`	Name of alarm	"
`unsigned char OPT_FLAG;`	Optional data flag	B*1 (see B*6)
`int RES_SCAL;`	Test results scaling exponent	I*1
`int LLM_SCAL;`	Low limit scaling exponent	I*1
`int HLM_SCAL;`	High limit scaling exponent	I*1
`float LO_LIMIT;`	Low test limit value	R*4
`float HI_LIMIT;`	High test limit value	R*4
`char *UNITS;`	Test units	C*n
`char *C_RESFMT;`	ANSI C result format string	"
`char *C_LLMFMT;`	ANSI C low limit format string	"
`char *C_HLMFMT;`	ANSI C high limit format string	"
`float LO_SPEC;`	Low specification limit value	R*4
`float HI_SPEC;`	High specification limit value	R*4
`}`		

`stdf_PTR_add()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.18 stdf_RDR_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_RDR_add()` function is used to add a RDR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Signals that the data in this STDF file is for retested parts. The data in this record, combined with information in the MIR, tells data filtering programs what data to replace when processing retest data.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_RDR_add(  unsigned short NUM_BINS,
                    unsigned char* RTSTBINS );
```

where:

`NUM_BINS` specifies the "*Number (k) of bins being retested*" (per the STDF specification). The data type of this argument = `U*2` in Data Type Codes and Representation.

`RTSTBINS` specifies the "*Array of retest bin numbers*" (per the STDF specification). The data type of this argument = `kxU*2` in Data Type Codes and Representation (see `kxTYPE`).

`stdf_RDR_add()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.19 stdf_SBR_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_SBR_add()` function is used to add a SBR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Stores a count of the parts associated with a particular logical bin after testing. This bin count can be for a single test site (when parallel testing) or a total for all test sites. The STDF specification also supports a Hardware Bin Record (HBR) for actual physical binning. A part is "physically" placed in a hardware bin after testing. A part can be "logically" associated with a software bin during or after testing.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_SBR_add(  unsigned char HEAD_NUM,
                    unsigned char SITE_NUM,
                    unsigned short SBIN_NUM,
                    unsigned int SBIN_CNT,
                    char SBIN_PF,
                    char *SBIN_NAM );
```

where:

`HEAD_NUM` specifies the "*Test head number*" (per the STDF specification). The data type of this argument = `U*1` in Data Type Codes and Representation.

`SITE_NUM` specifies the "*Test site number*" (per the STDF specification). The data type of this argument = `U*1` in Data Type Codes and Representation.

`SBIN_NUM` specifies the "*Software bin number*" (per the STDF specification). The data type of this argument = `U*2` in Data Type Codes and Representation.

`SBIN_CNT` specifies the "*Number of parts in bin*" (per the STDF specification). The data type of this argument = `U*4` in Data Type Codes and Representation.

SBIN_PF specifies the "*Pass/fail indication*" (per the STDF specification). The data type of this argument = `C*1` in Data Type Codes and Representation (see `C*12`).

SBIN_NAM specifies the "*Name of software bin*" (per the STDF specification). The data type of this argument = `C*n` in Data Type Codes and Representation. *Important*: see Note:.

`stdf_SBR_add()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.20 stdf_SDR_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_SDR_add()` function is used to add a SDR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Contains the configuration information for one or more test sites, connected to one test head, that compose a site group.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_SDR_add( SRDBlock data );
```

where **`data`** is a user-defined `SRDBlock` variable initialized to define the various parameters which describe an SDR record, as shown below. Some fields are required,

others may not be applicable but must be set to a specific value when not needed, see Standard Test Data Format (STDF) Specification:

	Standard Test Data Format (STDF) Specification Description	Data Type in Data Type Codes and Representation
`struct SDRBlock {`		
`unsigned char HEAD_NUM;`	Test head number	`U*1`
`unsigned char SITE_GRP;`	Site group number	`U*1`
`unsigned char SITE_CNT;`	Number (k) of test sites in site group	`U*1`
`unsigned char *SITE_NUM;`	Array of test site numbers	`kxU*1 (see kxTYPE)`
`char HAND_TYP[255];`	Handler or prober type	`C*n`
`char HAND_ID[255];`	Handler or prober ID	"
`char CARD_TYP[255];`	Probe card type	"
`char CARD_ID[255];`	Probe card ID	"
`char LOAD_TYP[255];`	Load board type	"
`char LOAD_ID[255];`	Load board ID	"
`char DIB_TYP[255];`	DIB board type	"
`char DIB_ID[255];`	DIB board ID	"
`char CABL_TYP[255];`	Interface cable type	"
`char CABL_ID[255];`	Interface cable ID	"
`char CONT_TYP[255];`	Handler contactor type	"
`char CONT_ID[255];`	Handler contactor ID	"
`char LASR_TYP[255];`	Laser type	"
`char LASR_ID[255];`	Laser ID	"
`char EXTR_TYP[255];`	Extra equipment type field	"
`char EXTR_ID[255];`	Extra equipment ID	"
`}`		

`stdf_SDR_add()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.21 stdf_TSR_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_TSR_add()` function is used to add a TSR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Contains the test execution and failure counts for one parametric or functional test in the test program. Also contains static information, such as test name. The TSR is related to the Functional Test Record (FTR), the Parametric Test Record (PTR), and the Multiple Parametric Test Record (MPR) by test number, head number, and site number.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_TSR_add( TSRBlock data );
```

where **`data`** is a user-defined `TSRBlock` variable initialized to define the various parameters which describe an TSR record, as shown below. Some fields are required, others may not be applicable but must be set to a specific value when not needed, see Standard Test Data Format (STDF) Specification:

```
struct TSRBlock {
   unsigned char HEAD_NUM;
   unsigned char SITE_NUM;
   char TEST_TYP;

   unsigned long TEST_NUM;
   unsigned long EXEC_CNT;
   unsigned long FAIL_CNT;
   unsigned long ALRM_CNT;
   char *TEST_NAM;
   char *SEQ_NAME;
   char *TEST_LBL;
   char OPT_FLAG;

   float TEST_TIM;
   float TEST_MIN;
```

Standard Test Data Format (STDF) Specification Description	Data Type in Data Type Codes and Representation
Test head number	U*1
Test site number	U*1
Test type	C*1 (see C*12)
Test number	U*4
Number of test executions	U*4
Number of test failures	U*4
Number of alarmed tests	U*4
Test name	C*n
Sequencer (program segment/flow) name	"
Test label or text	"
Optional data flag	B*1 (see B*6)
Average test execution time in seconds	R*4
Lowest test result value	R*4

```
   float  TEST_MAX;                Highest test result value          R*4
   float  TEST_SUMS;               Sum of test result values          R*4
   float  TEST_SQRS;               Sum of squares of test result values R*4
}
```

`stdf_TSR_add()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.22 stdf_WCR_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_WCR_add()` function is used to add a WCR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Contains the configuration information for the wafers tested by the job plan. The WCR provides the dimensions and orientation information for all wafers and dice in the lot. This record is used only when testing at wafer probe time.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
   bool stdf_WCR_add(  double WAFR_SIZ,
                       double DIE_HT,
                       double DIE_WD,
                       int WF_UNITS,
                       char WF_FLAT,
                       int CENTER_X,
```

```
                    int CENTER_Y,
                    int POS_X,
                    int POS_Y);
```

where:

WAFR_SIZ specifies the "*Diameter of wafer in WF_UNITS*" (per the STDF specification). The data type of this argument = R*4 in Data Type Codes and Representation.

DIE_HT specifies the "*Height of die in WF_UNITS*" (per the STDF specification). The data type of this argument = R*4 in Data Type Codes and Representation.

DIE_WD specifies the "*Width of die in WF_UNITS*" (per the STDF specification). The data type of this argument = R*4 in Data Type Codes and Representation.

WF_UNITS specifies the "*Units for wafer and die dimensions*" (per the STDF specification). The data type of this argument = U*1 in Data Type Codes and Representation.

WF_FLAT specifies the "*Orientation of wafer flat*" (per the STDF specification). The data type of this argument = C*1 in Data Type Codes and Representation (see C*12).

CENTER_X specifies the "*X coordinate of center die on wafer*" (per the STDF specification). The data type of this argument = I*2 in Data Type Codes and Representation.

CENTER_Y specifies the "*Y coordinate of center die on wafer*" (per the STDF specification). The data type of this argument = I*2 in Data Type Codes and Representation.

POS_X specifies the "*Positive X direction of wafer*" (per the STDF specification). The data type of this argument = C*1 in Data Type Codes and Representation (see C*12).

POS_Y specifies the "*Positive Y direction of wafer*" (per the STDF specification). The data type of this argument = C*1 in Data Type Codes and Representation (see C*12).

`stdf_WCR_add()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.23 stdf_WIR_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_WIR_add()` function is used to add a WIR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Acts mainly as a marker to indicate where testing of a particular wafer begins for each wafer tested by the job plan. The WIR and the Wafer Results Record (WRR) bracket all the stored information pertaining to one tested wafer. This record is used only when testing at wafer probe. A WIR/WRR pair will have the same HEAD_NUM and SITE_GRP values.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_WIR_add(  int HEAD_NUM,
                    int SITE_GRP,
                    unsigned long START_T,
                    char *WAFER_ID );
```

where:

`HEAD_NUM` specifies the "*Test head number*" (per the STDF specification). The data type of this argument = `U*1` in Data Type Codes and Representation.

`SITE_GRP` specifies the "*Site group number*" (per the STDF specification). The data type of this argument = `U*1` in Data Type Codes and Representation.

`START_T` specifies the "*Date and time first part tested*" (per the STDF specification). The data type of this argument = `U*4` in Data Type Codes and Representation.

`WAFER_ID` specifies the "*Wafer ID*" (per the STDF specification). The data type of this argument = `C*n` in Data Type Codes and Representation. *Important*: see Note:.

`stdf_WIR_add()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.5.24 stdf_WRR_add()

See STDF Record Add Functions, Overview.

Description

The `stdf_WRR_add()` function is used to add a WRR record to the STDF record heap.

In the Standard Test Data Format (STDF) Specification this record type is described as:

> *Contains the result information relating to each wafer tested by the job plan. The WRR and the Wafer Information Record (WIR) bracket all the stored information pertaining to one tested wafer. This record is used only when testing at wafer probe time. A WIR/WRR pair will have the same HEAD_NUM and SITE_GRP values.*

The Standard Test Data Format (STDF) Specification contains important additional information about this record type.

Important usage rules which apply to all STDF Record Add Functions are documented in Overview.

Usage

```
bool stdf_WRR_add( WRRBlock data );
```

where **`data`** is a user-defined `WRRBlock` variable initialized to define the various parameters which describe an WRR record, as shown below. Some fields are required, others may not be applicable but must be set to a specific value when not needed, see Standard Test Data Format (STDF) Specification:

`struct WRRBlock {`	Standard Test Data Format (STDF) Specification Description	Data Type in Data Type Codes and Representation
`int HEAD_NUM;`	Test head number	`U*1`
`int SITE_GRP;`	Site group number	`U*1`
`unsigned long FINISH_T;`	Date and time last part tested	`U*4`
`unsigned long PART_CNT;`	Number of parts tested	`U*4`
`unsigned long RTST_CNT;`	Number of parts retested	`U*4`
`unsigned long ABRT_CNT;`	Number of aborts during testing	`U*4`
`unsigned long GOOD_CNT;`	Number of good (passed) parts tested	`U*4`
`unsigned long FUNC_CNT;`	Number of functional parts tested	`U*4`
`char *WAFER_ID;`	Wafer ID	`C*n`

```
   char *FABWF_ID;                Fab wafer ID                          "
   char *FRAME_ID;                Wafer frame ID                        "
   char *MASK_ID;                 Wafer mask ID                         "
   char *USR_DESC;                Wafer description supplied by user    "
   char *EXC_DESC;                Wafer description supplied by exec    "
}
```

`stdf_WRR_add()` returns TRUE if the operation was successful, otherwise FALSE is returned.

Example

See STDF Code Example.

7.6.6 STDF Code Example

See STDF Software, Overview.

This example compiles and generates a valid STDF file on disk, however many of the values below are silly and used to show valid C syntax for each supported record type. The output file name = `../testSTDF_1.std` (relative to the test program executable file). To keep the example simple, none of the `bool` values returned from the STDF functions are tested for errors (not recommended).

```
#include "tester.h"
FTRBlock my_FTR_record;
MIRBlock my_MIR_record;
MPRBlock my_MPR_record
PRRBlock my_PRR_record;
PTRBlock my_PTR_record;
SRDBlock my_SDR_record;
TSRBlock my_TSR_record;
WRRBlock my_WRR_record;
time_t time_now;
bool ok;

// The code below must execute from the Host process or user tool
// process (see Overview). This example puts it into the
// HOST_BEGIN_BLOCK, which will be typcial.
HOST_BEGIN_BLOCK( HB1 ) {
  // In the real world, there will be many FTRs
```

```
  my_FTR_record.TEST_NUM = 101;
  my_FTR_record.HEAD_NUM = 1;
  my_FTR_record.SITE_NUM = 1;
  my_FTR_record.TEST_FLG = 42;
  my_FTR_record.OPT_FLAG = 43;
  my_FTR_record.CYCL_CNT = 3535;
  my_FTR_record.REL_VADR = 19;
  my_FTR_record.REPT_CNT = 0;
  my_FTR_record.NUM_FAIL = 8;
  my_FTR_record.XFAIL_AD = 33;
  my_FTR_record.YFAIL_AD = 44;
  my_FTR_record.VECT_OFF = 0;
#define RTN_SIZE 1
  unsigned short rtn_idx[ RTN_SIZE ] = { 1 };
  unsigned char rtn_stat[ RTN_SIZE ] = { '6' };
  my_FTR_record.RTN_ICNT = (sizeof( rtn_idx ) / sizeof(short)) ;
  my_FTR_record.RTN_INDX = rtn_idx;
  my_FTR_record.RTN_STAT = rtn_stat;
#define PGM_SIZE 1
  unsigned short pgm_idx[ PGM_SIZE ] = { 1 };
  unsigned char pgm_stat[ PGM_SIZE ] = { '6' };
  my_FTR_record.PGM_ICNT = (sizeof( pgm_idx ) / sizeof(short));
  my_FTR_record.PGM_INDX = pgm_idx;
  my_FTR_record.PGM_STAT = pgm_stat;
  my_FTR_record.FAIL_PIN_number_of_chars = 5;
  strcpy( my_FTR_record.FAIL_PIN, "12345" );
  strcpy( my_FTR_record.VECT_NAM, "myMemPat_1" );
  strcpy( my_FTR_record.TIME_SET, "LooseAC" );
  strcpy( my_FTR_record.OP_CODE, "VEC" );
  strcpy( my_FTR_record.TEST_TXT, "Read Seg Address-n" );
  strcpy( my_FTR_record.ALARM_ID, "Fire" );
  strcpy( my_FTR_record.PROG_TXT, "This example is silly" );
  strcpy( my_FTR_record.RSLT_TXT, "Get a real example" );
  my_FTR_record.PATG_NUM = 1;
  my_FTR_record.SPIN_MAP_number_of_chars = 5;
  strcpy( my_FTR_record.SPIN_MAP, "12345" );

  time( &time_now );
  my_MIR_record.SETUPTIME = time_now;
  my_MIR_record.FIRSTPARTTESTTIME = time_now;
  my_MIR_record.STATIONNUMBER = 101;
```

```
my_MIR_record.MODE_COD = 'M';
my_MIR_record.RTST_COD = 'R';
my_MIR_record.PROT_COD = 'P'  ;
my_MIR_record.BURN_TIM = 5;
my_MIR_record.CMOD_COD = 'C';
strcpy( my_MIR_record.LOT_ID, "Parking" );
strcpy( my_MIR_record.PART_TYP, "XYZZY" );
strcpy( my_MIR_record.NODE_NAM, "Lymph" );
strcpy( my_MIR_record.TSTR_TYP, "Magnum 2x" );
strcpy( my_MIR_record.JOB_NAM, "MostestHotestJob" );
strcpy( my_MIR_record.JOB_REV, "19" );
strcpy( my_MIR_record.SBLOT_ID, "None" );
strcpy( my_MIR_record.OPER_NAM, "Mike P" );
strcpy( my_MIR_record.EXEC_TYP, "UI" );
strcpy( my_MIR_record.EXEC_VER, "h3.4.xx" );
strcpy( my_MIR_record.TEST_COD, "preFinal" );
strcpy( my_MIR_record.TST_TEMP, "3.14K" );
strcpy( my_MIR_record.USER_TXT, "Oops" );
strcpy( my_MIR_record.AUX_FILE, "Rasp" );
strcpy( my_MIR_record.PKG_TYP, "EggCrate" );
strcpy( my_MIR_record.FAMLY_ID, "AdamsFamily" );
strcpy( my_MIR_record.DATE_COD, "Jurassic" );
strcpy( my_MIR_record.FACIL_ID, "Metropolis" );
strcpy( my_MIR_record.FLOOR_ID, "Basement 43" );
strcpy( my_MIR_record.PROC_ID, "LostWax" );
strcpy( my_MIR_record.OPER_FRQ, "TooooFast" );
strcpy( my_MIR_record.SPEC_NAM, "Mil_0U812" );
strcpy( my_MIR_record.SPEC_VER, "Final_9.3.1.3a" );
strcpy( my_MIR_record.FLOW_ID, "200GPM" );
strcpy( my_MIR_record.SETUP_ID, "769" );
strcpy( my_MIR_record.DSGN_REV, "Prelim_692.3.2" );
strcpy( my_MIR_record.ENG_ID, "427" );
strcpy( my_MIR_record.ROM_COD, "SpaceInvaders" );
strcpy( my_MIR_record.SERL_NUM, "0.0.1" );
strcpy( my_MIR_record.SUPR_NAM, "RobnAdlr" );

my_MPR_record.TEST_NUM = 1;
my_MPR_record.HEAD_NUM = 1;
my_MPR_record.SITE_NUM = 1;
my_MPR_record.TEST_FLG = 0x04;
my_MPR_record.PARM_FLG = 0x00;
```

```
unsigned char rtn_states[1] = { '6' };
my_MPR_record.RTN_ICNT =
   ( sizeof(rtn_states) / sizeof(unsigned char) );
my_MPR_record.RTN_STAT = rtn_states;
double rtn_results[1] = { 0.3e-3 };
my_MPR_record.RSLT_CNT =
   ( sizeof(rtn_results) / sizeof(double));
my_MPR_record.RTN_RSLT = rtn_results;
strcpy( my_MPR_record.TEST_TXT, "myTest" );
strcpy( my_MPR_record.ALARM_ID, "noAlarm" );
my_MPR_record.OPT_FLAG = 0x00;
my_MPR_record.RES_SCAL = 1.0e3;
my_MPR_record.LLM_SCAL = 1.0e3;
my_MPR_record.HLM_SCAL = 1.0e3;
my_MPR_record.LO_LIMIT = 0.1e-6;
my_MPR_record.HI_LIMIT = 31.95e-3;
my_MPR_record.START_IN = 10.0e-3;
my_MPR_record.INCR_IN = 0.5e-6;
short rtn_index[1] = { 9 };
my_MPR_record.RTN_INDX = rtn_index;
strcpy( my_MPR_record.UNITS, "uA" );
char tmp1[32];
sprintf( tmp1, "%7.2f uA", tmp1 );
strcpy( my_MPR_record.C_RESFMT, tmp1 );
strcpy( my_MPR_record.C_LLMFMT, tmp1 );
strcpy( my_MPR_record.C_HLMFMT, tmp1 );
my_MPR_record.LO_SPEC = 0.0;
my_MPR_record.HI_SPEC = 23.3;

// In the real world, there will be many PTRs
my_PTR_record.TEST_NUM = 21;
my_PTR_record.HEAD_NUM = 1;
my_PTR_record.SITE_NUM = 1;
my_PTR_record.TEST_FLG, '3';
my_PTR_record.PARM_FLG, '0';
my_PTR_record.RESULT = +23.32e-6;
strcpy( my_PTR_record.TEST_TXT, "Functional Shorts Test" );
strcpy( my_PTR_record.ALARM_ID, "None" );
my_PTR_record.OPT_FLAG = 'X';
my_PTR_record.RES_SCAL = -3;
my_PTR_record.LLM_SCAL = -6;
```

```
my_PTR_record.HLM_SCAL = -3;
my_PTR_record.LO_LIMIT = -1.32e-6;
my_PTR_record.HI_LIMIT = +44.0e-3;
strcpy( my_PTR_record.UNITS, "uA" );
strcpy( my_PTR_record.C_RESFMT, "%7.2f" );
strcpy( my_PTR_record.C_LLMFMT, "%7.2f" );
strcpy( my_PTR_record.C_HLMFMT, "%7.2f" );
my_PTR_record.LO_SPEC = -1.0e-6;
my_PTR_record.HI_SPEC = +40.0e-3;
```

```
my_SDR_record.HEAD_NUM = 1;
my_SDR_record.SITE_GRP = 1;
unsigned char sites[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
my_SDR_record.SITE_CNT = (sizeof(sites)/sizeof(unsigned char));
my_SDR_record.SITE_NUM = sites;
strcpy( my_SDR_record.HAND_TYP, "HandSocket" );
strcpy( my_SDR_record.HAND_ID, "UsuallyLeft" );
strcpy( my_SDR_record.CARD_TYP, "Hoyle Bicycle" );
strcpy( my_SDR_record.CARD_ID, "KingOfClubs" );
strcpy( my_SDR_record.LOAD_TYP, "Bricks" );
strcpy( my_SDR_record.LOAD_ID, "Mike P" );
strcpy( my_SDR_record.DIB_TYP, "None" );
strcpy( my_SDR_record.DIB_ID, "n/a" );
strcpy( my_SDR_record.CABL_TYP, "HDMI" );
strcpy( my_SDR_record.CABL_ID, "JimCarry" );
strcpy( my_SDR_record.CONT_TYP, "Needle" );
strcpy( my_SDR_record.CONT_ID, "SpaceNeedle" );
strcpy( my_SDR_record.LASR_TYP, "StarWars" );
strcpy( my_SDR_record.LASR_ID, "R2D2" );
strcpy( my_SDR_record.EXTR_TYP, "PlazmaNuker" );
strcpy( my_SDR_record.EXTR_ID, "Bruce" );
```

```
my_TSR_record.HEAD_NUM = 14;
my_TSR_record.SITE_NUM = 162;
my_TSR_record.TEST_TYP = 'F';
my_TSR_record.TEST_NUM = 12345;
my_TSR_record.EXEC_CNT = 432;
my_TSR_record.FAIL_CNT = 6666;
my_TSR_record.ALRM_CNT = 5432;
strcpy( my_TSR_record.TEST_NAM, "TestThisOnce" );
strcpy( my_TSR_record.SEQ_NAME, "TestItAgain" );
strcpy( my_TSR_record.TEST_LBL, "ReverseSequence" );
```

```
  my_TSR_record.OPT_FLAG = 0x3f;
  my_TSR_record.TEST_TIM = 3.92;
  my_TSR_record.TEST_MIN = 1.87;
  my_TSR_record.TEST_MAX = 9.987;
  my_TSR_record.TEST_SUMS = 333;
  my_TSR_record.TEST_SQRS = 56768;

  my_WRR_record.HEAD_NUM = 1;
  my_WRR_record.SITE_GRP = 100;
  time( &time_now );
  my_WRR_record.FINISH_T = time_now;
  my_WRR_record.PART_CNT = 45678;
  my_WRR_record.RTST_CNT = 45677;
  my_WRR_record.ABRT_CNT = 0;
  my_WRR_record.GOOD_CNT = 34567;
  my_WRR_record.FUNC_CNT = 45678;
  strcpy( my_WRR_record.WAFER_ID, "Mint" );
  strcpy( my_WRR_record.FABWF_ID, "FruityMint" );
  strcpy( my_WRR_record.FRAME_ID, "Picture" );
  strcpy( my_WRR_record.MASK_ID, "NixonMask" );
  strcpy( my_WRR_record.USR_DESC, "ThatRoundOne" );
  strcpy( my_WRR_record.EXC_DESC, "ThatCircularOne" );

  ok = stdf_file_open( "../testSTDF_1.std" );
  if( !ok ) output( "ERROR: stdf_file_open() returned FALSE" );

  time( &time_now );
  ok = stdf_ATR_add( time_now, "ui /nologo /E"  );
  ok = stdf_MIR_add( my_MIR_record );
  ok = stdf_PMR_add( 0, 1, "Chan-1" , "Clock" , "Clock" , 1, 1 );

  short myPins[10] = { 19, 12, 3, 22, 63, 1, 7, 15, 32, 3 };
  ok = stdf_PGR_add( 1,
                     "myPins",
                     (sizeof(myPins) / sizeof(short)),
                     myPins );

#define PSIZE 3
  short pIndx[ PSIZE ] = { 1, 2, 3 };
  short pMode[ PSIZE ] = { 20, 20, 20 };
  unsigned char pRadix[ PSIZE ] = { 0, 10, 16 };
  char pChar[ PSIZE ] = { 'F', 'P', 'X' };
  char pChal[ PSIZE ] = { 'l', 's', 'x' };
  char pRtnChar [ PSIZE ] = { 'l', 's', 'x' };
```

```
char pRtnChal [ PSIZE ] = { 'l', 's', 'x' };
ok = stdf_PLR_add(   3, pIndx, pMode, pRadix,
                     pChar, pChal,
                     pRtnChar, pRtnChal );
unsigned short rbins[10] = { 1, 1, 3, 2, 2, 1, 1, 5, 1, 3 };
ok = stdf_RDR_add( (sizeof( rbins ) / sizeof( short) ), rbins );
ok = stdf_SDR_add( my_SDR_record );
time( &time_now );
ok = stdf_WIR_add( 1, 12, time_now, "VanillaWafer" );
ok = stdf_BPS_add( "RedundancyEval" );

ok = stdf_FTR_add( my_FTR_record );
ok = stdf_PTR_add( my_PTR_record );

// Log per-DUT info. Bracket each DUT with PIR/PRR.
// Special datalog for DUT 3 only.
char tmp[32];
for( int dut = 1; dut < 10; ++dut ){
  ok = stdf_PIR_add( 1, dut );
  my_PRR_record.HEAD_NUM = 1;
  my_PRR_record.SITE_NUM = dut;
  strcpy( my_PRR_record.PART_FLG, "3" );
  my_PRR_record.NUM_TEST = 91;
  my_PRR_record.HARD_BIN = 3;
  if( dut == 3 )
    sprintf( tmp, "Special Datalog -> Dut-%d", dut );
  ok = stdf_DTR_add( tmp );
  my_PRR_record.SOFT_BIN = 33;
  my_PRR_record.X_COORD = -32768;
  my_PRR_record.Y_COORD = -32768;
  my_PRR_record.TEST_T = 3.456;
  sprintf( tmp, "Dut-%d", dut );
  strcpy( my_PRR_record.PART_ID, tmp );
  strcpy( my_PRR_record.PART_TXT, "UglyPart" );
  strcpy( my_PRR_record.PART_FIX, "S" );
  ok = stdf_PRR_add( my_PRR_record );
}

ok = stdf_GDR_unsigned_byte_add( 1, 255 );
ok = stdf_GDR_unsigned_byte_add( 2, 65535 );
ok = stdf_GDR_unsigned_byte_add( 4, 4294967295 );
ok = stdf_GDR_signed_byte_add( 1, -255 );
ok = stdf_GDR_signed_byte_add( 2, -65535 );
```

```
  ok = stdf_GDR_signed_byte_add( 4, -4294967295 );
  ok = stdf_GDR_floating_point_add( (float) -1.234e-9);
  ok = stdf_GDR_double_add( +1.234e-6 );
  char name[] = "myName";
  ok = stdf_GDR_char_add( sizeof( name ), name );
  char myBits[] = { 0x0, 0x1, 0x7F };
  ok = stdf_GDR_binary_add(   (sizeof(myBits)/sizeof(char)),
                              myBits );
  ok = stdf_GDR_bit_encoded_add( (sizeof(myBits)/sizeof(char)),
                                 myBits );
  ok = stdf_GDR_nybble_add( 0xF );
  ok = stdf_GDR_write_record();
  ok = stdf_HBR_add( 1, 1, 3, 509, 'P', "ThisBin" );
  ok = MPRBlock( my_MPR_record );
  ok = stdf_SBR_add( 1, 1, 309, 69, 'P', "ThatSWbin" );
 ok = stdf_WCR_add(300, 10, 10, 3, 'U', -32768, -32768, 'R', 'D');
  ok = stdf_PCR_add( 1, 1, 209, 29, 2, 207, 209 );
  ok = stdf_file_write();
  ok = stdf_EPS_add();
  ok = stdf_file_write();
  ok = WRRBlock( my_WRR_record );
  time( &time_now );
  ok = stdf_MRR_add( time_now, 'P', "BigLot", "BossBigLot" );
  ok = stdf_file_write();
  ok = stdf_TSR_add( my_TSR_record );
  ok = stdf_file_close();
  if( !ok ) output( "ERROR: stdf_file_close() returned FALSE" );
}
```

7.7 Excel Related Functions

The functions documented in this section allow a Magnum 1/2/2x test program to invoke and send data and selected commands to Excel.

7.7.1 Overview

See Excel Related Functions.

The functions documented in this section allow a Magnum 1/2/2x test program to invoke and send data and selected commands to Excel.

Note: these Excel functions exist solely to support Magnum 1/2/2x Timing Calibration software and are thus, by design, limited in scope and capability. The Excel automation library contains thousands of functions most of which are not supported by corresponding Magnum 1/2/2x functions. Additional functionality (for example, to create a chart, etc.) must be added by the user, using Excel macros developed, tested, and debugged within Excel. Then, the `RunMacro()` function may be used to execute any number of Excel macros from a Magnum 1/2/2x test program.

The following notes apply ONLY to instances of Excel invoked from a Magnum 1/2/2x test program:

- The functions documented here were testing using the following versions of Excel: Excel97, Excel2000.
- The Excel functions documented here can be invoked from either the host or site process.
- Excel related error checking is very (very) limited in the Magnum 1/2/2x software. In many cases, when an error occurs, the test program and/or UI will crash or hang, and it may be necessary to use the Windows task manager to terminate all offending processes.

Note: Excel itself may issue warnings, errors, and other forms of dialog popups. Often, these are not noticed by the user because they are displayed behind other graphic windows, etc. Depending on the nature of the situation, it can appear as though the test program and/or UI have stopped functioning correctly (hung). However, before using the task manager to terminate the test program and/or UI, go to the windows task bar and select Excel (to maximize it). This will also cause any pending dialogs to display. Often, once the proper action is taken using Excel, proper test program operation will resume.

- It is legal to invoke more than one workbook from the test program. All will be available within the same single instance of Excel. Use normal Excel controls to switch between workbooks. Some of the functions documented here limit or otherwise affect how workbook(s) operate. Again, Excel support is limited.
- Instances of Excel which were invoked from test program code ***MUST NOT*** be terminated from Excel; if this is done the test program *may* hang and require using the Windows task manager to terminate the test program and/or UI.

- Any workbook modified from the test program and not saved to disk may, when Excel is terminated, prompt the user to save or discard the changes. See `ReleaseExcel(), QuitExcel()`.
- Cell indexing is 0 based i.e. cell A1 is row 0, column (0, 0).

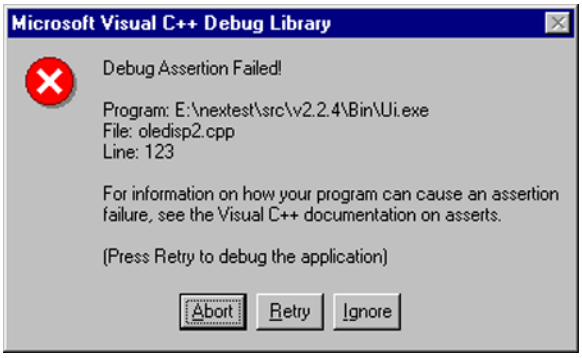
7.7.2 InvokeExcelEx()

See Excel Related Functions.

Description

The `InvokeExcelEx()` function is used to invoke Excel from a Magnum 1/2/2x test program. Excel will start and, if visible, contain no workbook(s). Use `AddWorkBook()` to create a new workbook or `OpenWorkBookEx()` to open an existing workbook stored on disk.

It is ***NOT*** legal to invoke more than one instance of Excel at a time. If this is done, the test program will crash, displaying the following:.



Use `ReleaseExcel(), QuitExcel()` to terminate an instance of Excel before invoking Excel again.

Note: Instances of Excel which were invoked from test program code MUST NOT be terminated from Excel; if this is done the test program may hang and require using the Windows task manager to terminate the test program and/or UI.

The `InvokeExcel()` function is included for backwards compatibility but is deprecated. Please use `InvokeExcelEx()`.

Usage

```
BOOL InvokeExcelEx( BOOL visible );
void InvokeExcel( BOOL visible ); // Deprecated
```

where:

visible determines whether Excel will be displayed (TRUE) or not (FALSE). FALSE means not displayed (not minimized) i.e. Excel is not accessible by or visible to the user.

InvokeExcelEx() returns TRUE if no errors occur, otherwise FALSE is returned. Error checking is very limited.

Example

The following example is used to demonstrate all of the Excel Related Functions.

Invoke Excel and do show the display:

```
if( ! InvokeExcelEx( TRUE ) )
   output("ERROR: invoking Excel");
```

Create a new Excel workbook.

```
AddWorkBook();
```

Add one worksheet named *Summary*. The cell grid-line will be visible. The row headings and column headings will be displayed.

```
AddWorkSheet( "Summary", TRUE, TRUE ); // AddWorkSheet()
```

Open an existing workbook named *test.xls*.

```
if ( ! OpenWorkBookEx( "C:/test.xls" ) ) { // OpenWorkBookEx()
   QuitExcel( TRUE, FALSE ); // ReleaseExcel(), QuitExcel()
   ReleaseExcel();
}
```

Set the column width of column-0 to 30:

```
SetColumnWidth( 0, 30 ); // SetColumnWidth()
```

Set the cell value at row/column B2 = 45:

```
AddVal( 45, 1, 1 ); // AddVal()
```

Set the cell value at row/column D9 = “Summary”:

```
AddVal( "Summary", 8, 3 ); // AddVal()
```

To fill a range of cells, use the AddArray() function rather than calling AddVal() in a C for loop. The MAKE_2D_ARRAY macro must be used to create the array (but doesn't initialize it):

```
int start_row = 0;
int end_row = 7;
int start_col = 0;
int end_col = 2;

// MAKE_2D_ARRAY is a Nextest Macro, see AddArray()
MAKE_2D_ARRAY( double,
               array,
               (end_row - start_row +1),
               (end_col - start_col +1) );

// Initialize the array
for( int row= start_row; row < end_row; row++)
   for( int col = start_col; col < end_col; col++)
      array[row][col] = (row * 2.5 + col);

AddArray( array, 0, 0 ); // AddArray()
```

Update the Excel display:

```
UpdateScreen( TRUE ); // UpdateScreen()
```

Save the workbook to disk:

```
SaveAs("c:/summary.xls"); // SaveAs()
```

Get the value from one cell (row-3/col-7 in this example):

```
double val;
if( GetVal( 3, 7, double &val ) ) // GetVal()
   output(" val => %d", val );
else
   output("ERROR: getval() returned an error");
```

To get the values from a range of contiguous cells use the GetArray() function rather than calling GetVal() in a C loop. This example uses the array defined using MAKE_2D_ARRAY above:

```
if( ! GetArray( row, col, array ) ) // GetArray()
   output("ERROR: GetArray() returned FALSE");
```

Terminate Excel. Order is specific:

```
QuitExcel(TRUE, FALSE); // ReleaseExcel(), QuitExcel()
ReleaseExcel();
```

7.7.3 OpenWorkBookEx()

See Excel Related Functions.

Description

The `OpenWorkBookEx()` function is used to open an existing Excel file stored on disk.

The `OpenWorkBook()` function is included for backwards compatibility but is deprecated. Please use `OpenWorkBookEx()`.

Usage

```
BOOL OpenWorkBookEx( LPCTSTR filename );
void OpenWorkBook( LPCTSTR filename ); // Deprecated
```

where:

`filename` specifies the disk, path and file name of the target Excel workbook.

`OpenWorkBookEx()` returns `TRUE` if no errors occur, otherwise `FALSE` is returned.

Example

See Example.

7.7.4 AddWorkBook()

See Excel Related Functions.

Description

The `AddWorkBook()` function is used to add a new workbook to the single instance of Excel invoked using `InvokeExcelEx()`. Note the following:

- While it is possible to add any number of workbooks, there are no programmatic mechanisms (no functions) for switching between workbooks, or for closing the active workbook. If Excel is displayed (see `InvokeExcelEx()`) it is possible to manually switch between workbooks, and the active workbook becomes the target for any Excel functions subsequently executed. However, due to the limitations noted, it is not normally useful to open more than one workbook at a time.

- Workbook naming is automatic, and has no significance until saved to disk.

Usage

```
void AddWorkBook();
```

Example

See Example.

7.7.5 AddWorkSheet()

See Excel Related Functions.

Description

The `AddWorkSheet()` function can be used to add a single worksheet to the currently active workbook (see `AddWorkBook()`). Note the following:

- The `AddWorkSheet()` function only affects the active workbook in the single instance of Excel invoked from the test program (see `InvokeExcelEx()`).
- The new worksheet is only added to the active workbook i.e. if multiple workbooks exist in the open Excel only the visible one will be modified.
- The `AddWorkSheet()` function *deletes* other worksheets in the active workbook. This can cause ***LOSS OF DATA*** if the workbook is not saved to disk. No warnings are displayed.

Usage

```
void AddWorkSheet( LPCTSTR sheetName, BOOL grid, BOOL heading );
```

where:

`sheetName` specifies the name of the worksheet displayed in Excel.

`grid` is used to define whether grid lines are to be displayed. Legal values are `TRUE` and `FALSE`.

`heading` is used to define whether row and column headings are to be displayed. Legal values are `TRUE` and `FALSE`.

Example

See Example.

7.7.6 SelectWorkSheet()

See Excel Related Functions.

Description

The `SelectWorkSheet()` function is used to specify the active Excel worksheet. The following functions only access the active worksheet:

- `SetColumnWidth()`
- `AddVal()`
- `GetVal()`
- `AddArray()`
- `GetArray()`
- `UpdateScreen()`
- `RunMacro()`

Usage

```
BOOL SelectWorkSheet( LPCTSTR sheetName );
```

where:

`sheetName` is the name of the target worksheet.

`SelectWorkSheet()` returns `TRUE` if the worksheet name specified is valid, otherwise `FALSE` is returned.

Example

In the following example does the following:

- Start Excel
- Open a Excel workbook named *c:/mybook.xls*
- Select worksheet named *Sheet2*
- Get an integer value (val) from the cell at row = 0, col = 1:

```
// The following is test block code...
InvokeExcel( TRUE ); // Make Excel visible

LPCTSTR wb = "c:/mybook.xls";
if ( ! OpenWorkBookEx( wb ) ) {
   output( "ERROR: failed to open workbook => %s", wb );
   return( FALSE );
}

LPCTSTR sht = "Sheet2";
if ( ! SelectWorkSheet( sht ) ) {
   output( "ERROR: invalid worksheet specified => %s", sht );
   return( FALSE );
}

int val;
GetVal( 0, 1, &val );
```

7.7.7 GetActiveSheet()

See Excel Related Functions.

Description

The `GetActiveSheet()` function is used to get the name of the currently active worksheet in Excel.

This is useful in the body code of `ui_ExcelAppEvent`.

Usage

```
void GetActiveSheet( CString* name );
```

where:

name is a pointer to an existing `CString` variable used to return the name of the currently active Excell worksheet.

Example

```
CString name;
GetActiveSheet( &name );
```

7.7.8 GetActiveCell()

See Excel Related Functions.

Description

The `GetActiveCell()` function is used to get the coordinates of the currently selected cell in Excel.

This is useful in the body code of `ui_ExcelAppEvent`.

Usage

```
void GetActiveCell( int* row, int* col );
```

where:

`row` and **`col`** are the addresses of existing int variables used to return the coordinates of the currently selected cell in Excel.

Example

```
int row, col;
GetActiveCell( &row, &col );
```

7.7.9 GetSelectionRange()

See Excel Related Functions.

Description

The `GetSelectionRange()` function is used to get the coordinates of the range(s) currently selected in Excel. Multiple ranges can be selected, and are returned.

This is useful in the body code of `ui_ExcelAppEvent`.

Usage

```
typedef CArray< RECT, RECT > RectArray;
void GetSelectionRange( RectArray* range_rects );
```

where:

range_rects returns 0 or more range selections.

Example

```
RectArray my_array;
GetSelectionRange( &my_array );
int num_rects = my_array.GetSize();
output("Number of ranges selected => %d", num_rects );
for( int i = 0; i < num_rects; ++i ) {
   RECT r = my_array[ i ];
   output(" Range => %d", i );
   output("   Start_row => %d", r.top );
   output("   Start_col => %d", r.left );
   output("   End_row => %d", r.bottom );
   output("   End_col => %d", r.right );
}
```

7.7.10 UpdateScreen()

See Excel Related Functions.

Description

The UpdateScreen() function causes Excel to repaint its display.

Usage

```
void UpdateScreen( BOOL val );
```

where:

val determines whether the Excel screen is updated. Legal values are TRUE and FALSE.

Example

See Example.

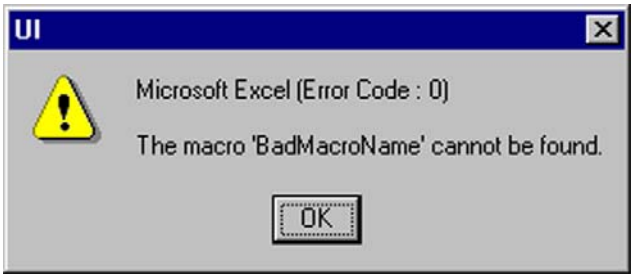
7.7.11 RunMacro()

See Excel Related Functions.

Description

As mentioned in Note:, the Magnum 1/2/2x software has limited support for Excel. The `RunMacro()` function can be used to greatly extend the existing capabilities by causing Excel to execute any currently loaded macro, by name.

If the specified macro does not exist UI will display a dialog similar to the following. In this example, the macro name passed to `RunMacro()` was *BadMacroName*:



No other actions are taken by UI or the test program, which will continue to execute.

Usage

```
void RunMacro( LPCTSTR name );
```

where:

name is the name of the Excel macro to be executed.

Example

See Example.

7.7.12 SaveAs()

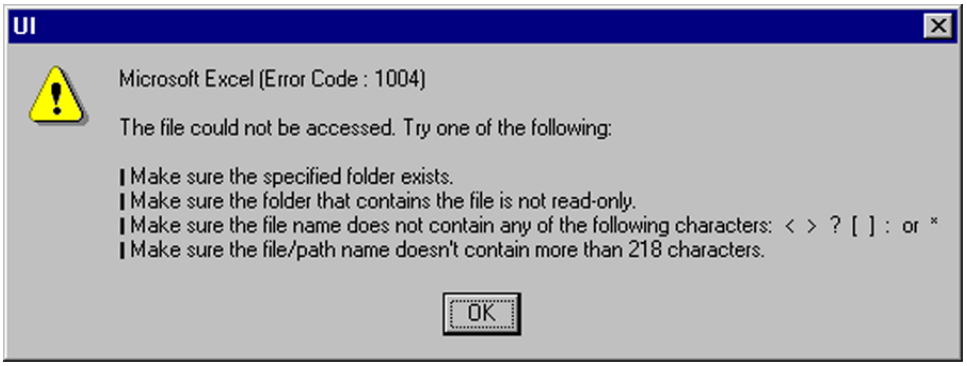
See Excel Related Functions.

Description

The `SaveAs()` function can be used to save the active workbook to disk, as an Excel `.xls` file.

> Note: no file clobbering checks are made i.e. an existing file of the same name will be over-written.

If any folders in the specified path do not exist an error similar to the following will be displayed:



Usage

```
void SaveAs( LPCTSTR filename );
```

where:

`filename` specifies the disk, path and file name to be written to disk. Note that any file name extension, including `.xls`, must be explicitly specified.

Example

See Example.

7.7.13 ReleaseExcel(), QuitExcel()

See Excel Related Functions.

Description

The `QuitExcel()` function is used to terminate Excel, similar to using Excel's **`File->Exit`** function. Proper operation also requires executing the `ReleaseExcel()` function (next item).

The `ReleaseExcel()` function causes the Excel process to be terminated. This is necessary when Excel is invoked from a software application.

Note: proper operation requires that the `ReleaseExcel()` function be executed after the `QuitExcel()` function.

Note: Instances of Excel which were invoked from test program code MUST NOT be terminated from Excel; if this is done the test program may hang and require using the Windows task manager to terminate the test program and/or UI.

Usage

```
void ReleaseExcel( );
void QuitExcel( BOOL force, BOOL save );
```

where:

`force` specifies whether Excel should be terminated unconditionally. Legal values are TRUE and FALSE. If TRUE is specified the **`save`** argument is valid (see below). If FALSE is specified, operation is similar to invoking Excel's **`File->Exit`** function i.e. Excel will prompt the user if unsaved changes exist.

`save` is valid only when the **`force`** argument is TRUE. **`save`** specifies whether modifications should be discarded (FALSE) or saved to disk (TRUE). If TRUE is specified and Excel does not already know into which file to save the workbook, Excel will prompt the user with a standard file browser.

Example

See Example.

7.7.14 Excel Value Set/Get Functions

These functions are used to manipulate values and attributes within the active Excel worksheet. See `AddWorkSheet()`.

- `SetColumnWidth()`
- `AddVal()`
- `GetVal()`
- `AddArray()`
- `GetArray()`

7.7.14.1 SetColumnWidth()

See Excel Related Functions and Excel Value Set/Get Functions.

Description

The `SetColumnWidth()` function can be used to set the width of a single column in the active Excel worksheet (see `AddWorkSheet()`).

Usage

```
void SetColumnWidth( int col, float width );
```

where:

`col` specifies the target column. Values are zero based.

`width` specifies the target width. The value used here is the same as when using Excel's **`Format->Column->Width`** control.

Example

See Example.

7.7.14.2 AddVal()

See Excel Related Functions and Excel Value Set/Get Functions.

Description

The `AddVal()` function can be used to set (not add-to) the value of a single cell in the active Excel worksheet (see `AddWorkSheet()`).

See `GetVal()`.

Usage

Three versions are available to support different data types:

```
void AddVal( int val, int row, int col );
void AddVal( double val, int row, int col );
void AddVal( CString val, int row, int col );
```

where:

`val` specifies the desired value. Note that the data types supported are `int`, `double`, and `CString`.

`row` and **`col`** identify the target cell to modify. Both values are zero based. Any previous value in the target cell is lost. Cell formatting is set to match the data type.

Example

See Example.

7.7.14.3 GetVal()

See Excel Related Functions and Excel Value Set/Get Functions.

Description

The `GetVal()` function can be used to get the value of a single cell in the active Excel worksheet (see `AddWorkSheet()`).

See `AddVal()`.

Usage

Three versions are available to support different data types:

```
BOOL GetVal( int row, int col, int *val );
BOOL GetVal( int row, int col, double *val );
BOOL GetVal( int row, int col, CString *val );
```

where:

`row` and **`col`** are the zero-based coordinates of the cell to be accessed.

`GetVal()` returns `TRUE` if no errors occur, otherwise `FALSE` is returned. Errors can include: specifed cell is invalid; Excel returns an error.

Example

See Example.

7.7.14.4 AddArray()

See Excel Related Functions and Excel Value Set/Get Functions.

Description

The `AddArray()` function can be used to set (not add-to) the value of a range of cells in the active Excel worksheet (see `AddWorkSheet()`).

See `GetArray()`.

Usage

```
void AddArray( array, int row, int col );
```

where:

`array` is the array of values. **`array`** must be created using the `MAKE_2D_ARRAY()` macro (see below).

`row` and **`col`** identify the upper left corner of the range of cells to be filled. The number of cells to be filled is determined by the size of **`array`**. Values are zero based.

`MAKE_2D_ARRAY()` is a Nextest macro used to create a 2D array of `int`, `double`, or `CString` (only) values. `MAKE_2D_ARRAY()` does not initialize the array.

```
MAKE_2D_ARRAY( type,
               array,
               num_rows,
               num cols )
```

where:

type specifies the data type to be stored in the array. Only `int`, `double`, and `CString` are supported.

array is the array variable being created. Must be a legal C identifier.

num_rows and **num_cols** specify the size of the array.

Example

See Example.

7.7.14.5 GetArray()

See Excel Related Functions and Excel Value Set/Get Functions.

Description

The `GetArray()` function can be used to get multiple values from a range of cells in the active Excel worksheet (see `AddWorkSheet()`).

See `AddArray()`.

Usage

```
BOOL GetArray( int row, int col, array );
```

where:

row and **col** identify the upper left corner of the range of cells to be read. The number of cells read is determined by the size of **array**. Row/col values are zero based.

array is the array used to return the specified values. **array** must be created using the `MAKE_2D_ARRAY()` macro (see below).

`MAKE_2D_ARRAY()` is a Nextest macro used to create a 2D array of `int`, `double`, or `CString` (only) values. `MAKE_2D_ARRAY()` does not initialize the array.

```
MAKE_2D_ARRAY( type,
               array,
               num_rows,
               num cols )
```

where:

`type` specifies the data type to be stored in the array. Only `int`, `double`, and `CString` are supported.

`array` is the array variable being created. Must be a legal C identifier.

`num_rows` and **`num_cols`** specify the size of the array.

`GetArray()` returns TRUE if no errors occur, otherwise FALSE is returned. Errors can include: not enouth memory for **`array`**; specifed range is invalid; Excel returns an error.

Example

See Example.

7.7.15 Excel Event Detection

See Excel Related Functions.

When the user clicks in an Excel worksheet an event can now be received by UI, which will execute a user-written call-back function to act on that event.

There are 2 parts to this facility:

- Enable the `ui_ExcelAppEvent` call-back function using `EnableExcelAppEvents()`.
- Write the callback code. See `ui_ExcelAppEvent`.

7.7.15.1 EnableExcelAppEvents()

See Excel Event Detection, Excel Related Functions.

Description

The `EnableExcelAppEvents()` function is used to enable or disable the `ui_ExcelAppEvent` call-back function. Note the following:

- When `ui_ExcelAppEvent` is enabled, if the user clicks in an Excel spreadsheet, an event is generated and sent to UI, at which time the `ui_ExcelAppEvent` user-written body code is executed.
- UI will invoke `ui_ExcelAppEvent` in Host and/or Site and/or User Tools processes.
- If `ui_ExcelAppEvent` is not enabled the Excel event has no effect.
- The value assigned to `ui_ExcelAppEvent` at the time it is executed will reflect the nature of the event. See `ui_ExcelAppEvent`.

Usage

```
BOOL EnableExcelAppEvents( BOOL enable );
```

where:

`enable` specifies whether the `ui_ExcelAppEvent` call-back function is enabled (TRUE) or disabled (FALSE).

`EnableExcelAppEvents()` returns TRUE if successful, otherwise FALSE is returned.

Example

```
if( EnableExcelAppEvents( TRUE ) == FALSE )
   output(" ERROR: EnableExcelAppEvents() returned FALSE");
```

7.8 Debug Hook and Pin Status Hook

This section documents functions which support using user-written C-code to:

- Generate debug messages. See `install_debug_hook()`.
- Log failing pin information. See `install_pinstatus_hook()`

Both methods provide a mechanism which executes code which is separate from Test Blocks but executes during Sequence & Binning Table execution.

7.8.1 install_debug_hook()

Description

The `install_debug_hook()` function is used to register a user-written call-back function.

If a debug hook call-back is registered, during Sequence & Binning Table execution (only) the call-back will automatically be executed ***both*** before ***and*** after:

- Any of the Nextest *test* functions execute i.e. `funtest()`, `partest()`, `ac_partest()`, `test_supply()`, `ac_test_supply()`, `hv_test_supply()`, `hv_ac_test_supply()`, `ptu_partest()`. These are referred to as *test functions* below. In this case, the `current_test()` function will return the name of the test function, `current_setup()` returns `NULL`, and `current_test_block()` will return the name of the Test Block being executed
- Any Nextest function which increments the setup number executes i.e. all Nextest functions except the *test* functions noted previously. These are referred to as *setup functions* below. In this case, the `current_setup()` function will return the name of the setup function, `current_test()` returns `NULL`, and `current_test_block()` will return the name of the Test Block being executed.
- Each Test Block executes. In this case, the `current_test_block()` function will return the name of the Test Block being executed.
- Each Test Bin executes. The `current_test_block()` will return the name of the Test Bin being executed

The debug hook call-back is targeted at several applications:

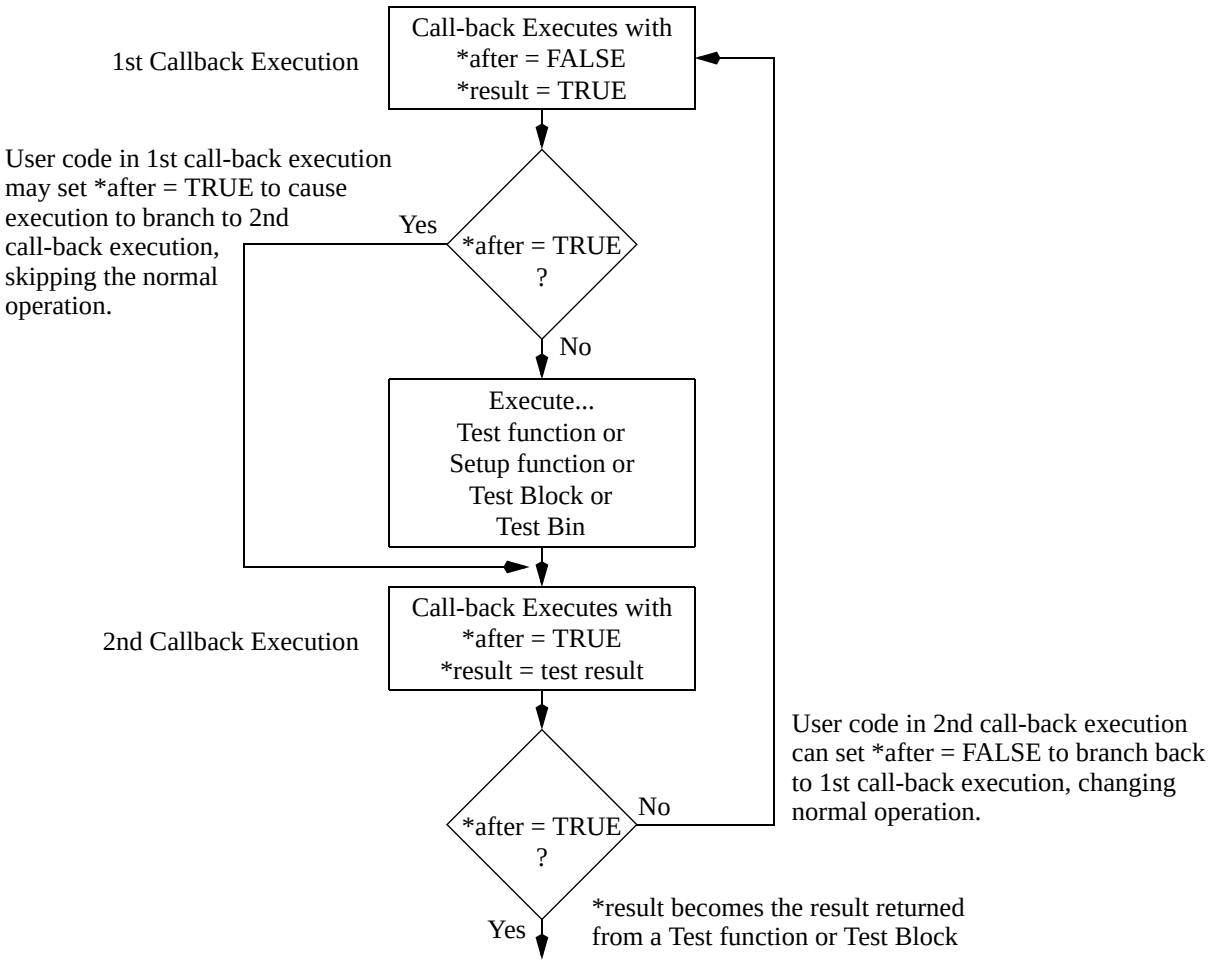
- Enable user-written C-code, written in a function separate from Test Block code, to generate execution trace messages.
- Enable [conditional] execution and/or loops of individual code statements, with controlling code in a function separate from Test Block code.

At each execution, the call-back function receives two parameters:

- `*after` = a pointer to an existing `BOOL` variable intialized to indicate whether the call-back is being invoked before (`*after` = FALSE) or after (`*after` = TRUE) the [test function, setup function, Test Block, or Test Bin] executes. As noted below, the call-back code can modify `*after` to control subsequent execution options.
- `*result` = a pointer to an `int` value which ultimately determines the result returned from the test function, or a Test Block. The call-back code can conditionally execute code based on `*result` and can also modify `*result`. If

`*result` is modified during the 2nd call-back execution (`*after == TRUE`), it determines the value returned by the test function, or a Test Block. The value in `*result` has no direct effect on execution sequence.

If a debug hook call-back is registered, execution follows the model below. Remember, if a call-back is registered, it will automatically be executed ***both*** before ***and*** after every test function, every setup function, every Test Block and every Test Bin:



Several functions are available which are useful within the call-back function. See `test_number()`, `setup_number()`, and `current_test_block()`. Also see `current_setup()`, and `current_test()`, which are *only* useful when executed within the debug call-back function.

To un-register the call-back invoke the `install_debug_hook()` function and pass `NULL` as the argument.

Usage

The following function is used to register the user-written call-back function:

```
debug_hook_type install_debug_hook( debug_hook_type hook );
```

where:

hook is a pointer to a user-written function with the following prototype:

```
void func ( BOOL &after, int &result ){}
```

Note: **`after`** and **`result`** are created by the system software and used as noted in Description.

`install_debug_hook()` returns a pointer to the previous call-back function, if one was registered, otherwise `NULL` is returned.

Example

The example has the following three parts:

- Call-back Registration Code
- Call-back Function Code
- Test Block Code
- Example Output

The example includes usage of: `current_setup()` and `current_test()`.

Call-back Registration Code

Use `install_debug_hook()` to register the call-back function. This must be done in a site process i.e. `SITE_CONFIGURATION()`, `SITE_BEGIN_BLOCK()`, `INITIALIZATION_HOOK()`, test block code, or the body code of a User-defined User Variables executed on a Site.

```
install_debug_hook( myDebugFunc );
```

Call-back Function Code

The call-back function is named by the user, but the prototype must conform to the declaration noted in Usage.

```
void myDebugFunc ( BOOL &after, int &result ) {
  output("myDebugFunc ( after = %s, result = %d)",
           after ? "TRUE":"FALSE",
           result);
```

```
   output(" current_setup      => %s", current_setup() );
   output(" current_test       => %s", current_test() );
   output(" current_test_block => %s", current_test_block() );
}
```

Test Block Code

The following test block was executed to generate the Example Output. Note that no references to the call-back exist in the test block:

```
TEST_BLOCK( TB1 ) {
   vil( 0.00 V );
   vz(  0.00 V,  PL_ALL );
   vpar_high( 0 V );
   test_result = partest( pass_vg, PL_1_DPS );
   test_result |= partest( pass_nicl, PL_ALL );
   return test_result;
}
```

Example Output

```
myDebugFunc ( after = FALSE, result = 1)
    setup number        => 0
     test number        => 0
    current_setup       =>
    current_test        =>
    current_test_block  => TB1
myDebugFunc ( after = FALSE, result = 1)
    setup number        => 1
     test number        => 0
    current_setup       => vil
    current_test        =>
    current_test_block  => TB1
myDebugFunc ( after = TRUE, result = 1)
    setup number        => 1
     test number        => 0
    current_setup       => vil
    current_test        =>
    current_test_block  => TB1
myDebugFunc ( after = FALSE, result = 1)
    setup number        => 2
     test number        => 0
```

```
    current_setup       => vz
    current_test        =>
    current_test_block  => TB1
myDebugFunc ( after = TRUE, result = 1)
    setup number        => 2
     test number        => 0
    current_setup       => vz
    current_test        =>
    current_test_block  => TB1
myDebugFunc ( after = FALSE, result = 1)
    setup number        => 4
     test number        => 0
    current_setup       => vpar_high
    current_test        =>
    current_test_block  => TB1
myDebugFunc ( after = TRUE, result = 1)
    setup number        => 4
     test number        => 0
    current_setup       => vpar_high
    current_test        =>
    current_test_block  => TB1
myDebugFunc ( after = FALSE, result = 1)
    setup number        => 0
     test number        => 1
    current_setup       =>
    current_test        => partest
    current_test_block  => TB1
myDebugFunc ( after = TRUE, result = 1)
    setup number        => 0
     test number        => 1
    current_setup       =>
    current_test        => partest
    current_test_block  => TB1
myDebugFunc ( after = FALSE, result = 1)
    setup number        => 0
     test number        => 2
    current_setup       =>
    current_test        => partest
    current_test_block  => TB1
myDebugFunc ( after = TRUE, result = 1)
    setup number        => 0
```

```
      test number         => 2
     current_setup        =>
     current_test         => partest
     current_test_block   => TB1
myDebugFunc ( after = TRUE, result = 1)
     setup number         => 0
      test number         => 2
     current_setup        =>
     current_test         =>
     current_test_block   => TB1
myDebugFunc ( after = FALSE, result = 1)
     setup number         => 0
      test number         => 0
     current_setup        =>
     current_test         =>
     current_test_block   => builtin_Pass
myDebugFunc ( after = TRUE, result = 1)
     setup number         => 0
      test number         => 0
     current_setup        =>
     current_test         =>
     current_test_block   => builtin_Pass
TestDone...bin = builtin_Pass
```

7.8.1.1 current_setup()

Description

If a call-back function is registered using `install_debug_hook()`, when the call-back executes before or after a *setup function* the name of that function can be obtained using the `current_setup()`. This can be useful for tracing program execution using the debug hook call-back.

A setup function is any Nextest function which increments the setup number i.e. all Nextest functions except the *test* functions (i.e. except `funtest()`, `partest()`, `ac_partest()`, `test_supply()`, `ac_test_supply()`, `hv_test_supply()`, `hv_ac_test_supply()`, `ptu_partest()`).

`current_setup()` returns `NULL` when the call-back is executing before/after a test function, a Test Block, or a Test Bin. See `install_debug_hook()`.

The `current_test()` function serves a similar for *test* function.

Note: `current_setup()` is only useful within the scope of the call-back function registered using `install_debug_hook()`.

Usage

```
LPCTSTR current_setup();
```

See Description.

Example

See Example.

7.8.1.2 current_test()

Description

If a call-back function is registered using `install_debug_hook()`, when the call-back executes before or after a *test function* the name of that function (*not* the test name passed to the function) can be obtained using the `current_test()`. This can be useful for tracing program execution using the debug hook call-back.

A test function is one of the Nextest functions which increment the test number i.e. `funtest()`, `partest()`, `ac_partest()`, `test_supply()`, `ac_test_supply()`, `hv_test_supply()`, `hv_ac_test_supply()`, `ptu_partest()`.

`current_test()` returns `NULL` when the call-back is executing before/after a setup function, a Test Block, or a Test Bin. See `install_debug_hook()`.

The `current_setup()` function serves a similar for *setup* functions.

Note: `current_test()` is only useful within the scope of the call-back function registered using `install_debug_hook()`.

Usage

```
LPCTSTR current_test();
```

See Description.

Example

See Example.

7.8.2 install_pinstatus_hook()

Description

The `install_pinstatus_hook()` function is used to register a user-written call-back function.

If a pin status hook call-back is registered, during Sequence & Binning Table execution (only) the call-back function will automatically be executed after any *test* function executes i.e. `funtest()`, `partest()`, `ac_partest()`, `test_supply()`, `ac_test_supply()`, `hv_test_supply()`, `hv_ac_test_supply()`, `ptu_partest()`.

Each time it executes, the call-back function receives the following parameters. In simple terms, the information displayed in the FrontPanelTool is made available via arguments to the call-back function:

- The `PASS/FAIL` result of the test just executed.
- An array of pin status information about each pin tested.
- The size of the array (number of array elements).
- A flag indicating whether all pins in the array are DPS pins.

To un-register the call-back invoke the `install_pinstatus_hook()` function and pass `NULL` as the argument.

Usage

The following function is used to register the user-written call-back function:

```
pinstatus_hook_type
   install_pinstatus_hook( pinstatus_hook_type hook );
```

where:

hook is a pointer to a user-written function with the following prototype:

```
void func ( BOOL result, PinStatus *data, int size, int flags );
```

where:

func is the user-defined name of the call-back function.

result is the result of the test just executed. The system software passes it to the call-back.

data is a pointer to an array of PinStatus information. The system software creates the array, and passes a pointer to the call-back. The pointer points to the first element in the array and **size** indicates the number of elements in the array. The contents of the array are in pin number order i.e. t_1, t_2, etc. regardless of the nature of the test performed. Each element of the **data** array will contain a value from the PinStatus enumerated type:

```
enum PinStatus { ps_untested = 0, ps_passed, ps_failed };
```

When the test function just executed was a functional test, only ps_untested and ps_failed are used. For DC tests all three values are used.

flags currently only indicates whether all pins in the array are DPS pins (0x1) or not (0x0). Future software enhancements may use additional flag bits as necessary. The flags field is only valid after performing PMU or DPS tests. The system software passes it to the call-back.

install_pinstatus_hook() returns a pointer to the previous call-back if one was registered, otherwise NULL is returned.

Example

The example has three parts:

Call-back Registration Code
Call-back Function Code
Test Block Code
Example Output

Call-back Registration Code

Use install_pinstatus_hook() to register the call-back function. This must be done in a site process i.e. SITE_CONFIGURATION(), SITE_BEGIN_BLOCK(), INITIALIZATION_HOOK(), test block code, or the body code of a User-defined User Variables executed on a Site.

```
install_pinstatus_hook( myPinStatusFunc );
```

Call-back Function Code

For proper operation, the code below requires that the `PinList*` tested in the previous PMU or DPS test be accessible to the call-back code. In this example, the `PinList*` named `testedPins` is used in the call-back code below but is used and declared in the Test Block Code:

```
void myPinStatusFunc(  BOOL result,
                       PinStatus *data,
                       int size,
                       int flags ) {
  if (result == PASS ) return; // Nothing to do
  output("  myPinStatusFunc()");
  output("    result => %s", result ? "TRUE" : "FALSE" );
  output("    size   => %d", size );
  output("    flags  => 0x%x", flags );
  output("    FAILED on the following pins");
  if( flags & 0x1) output("      All pins tested were DPS pins");
  HDTesterPin tpin;
  int ignore, pnum;
  LPCTSTR pname;
  for (   int p = 0; // For each pin tested...
          pin_info( testedPins, p, &ignore, &pnum, &pname );
          p++ ) {
    if ( p > size ) {
      output("ERROR: contents of testedPins exceeds\n");
      output("       PinStatus array size. Exiting.\n");
      return;
    }
    if( data[p] == ps_failed) {
      output("        %s (pin => %d)", pname, pnum );
    }
  }
}
```

Test Block Code

The following test block executes `partest()` 3 times, thus the call-back will be executed 3 times:

```
// Make "testedPins" global. It is referenced in both partest()
// below and in the Call-back Function Code.
PinList* testedPins;
BOOL test_result;

TEST_BLOCK( TB_continuity ) {
   back_voltage_enable ( TRUE );
   back_voltage(0 V);
   ipar_force (-100 UA);
   vpar_high(-100 MV);
   vpar_low(-2 V);
   vclamp( 1 V, -3 V );
   partime( 0 MS );

   testedPins = PL_PE3;
   test_result = partest(pass_nivl, testedPins );

   testedPins = PL_13;
   test_result &= partest(pass_nivl, testedPins );

   testedPins = PL_Vcc_13;
   test_result &= partest( pass_vg, testedPins );

   return test_result;
}
```

Example Output

The `myPinStatusFunc()` call-back function was executed 3 times:

```
myPinStatusFunc()
    result => FALSE
    size   => 48
    flags  => 0x0
    FAILED on the following pins
myPinStatusFunc()
    result => FALSE
    size   => 48
    flags  => 0x0
    FAILED on the following pins
        P3 (pin => 3)
        P4 (pin => 4)
        P5 (pin => 5)
        P6 (pin => 6)
        P10 (pin => 10)
```

```
        P14 (pin => 14)
        P15 (pin => 15)
        P16 (pin => 16)
        P34 (pin => 34)
        P35 (pin => 35)
        P36 (pin => 36)
        P37 (pin => 37)
myPinStatusFunc()
    result => FALSE
    size   => 3
    flags  => 0x1
    FAILED on the following pins
      All pins tested were DPS pins
        Vcc_1 (pin => 1)
TestDone...bin = builtin_Pass
```

7.9 MonitorApp

Description

- Unless Debugging With Developer Studio, *MonitorApp* is the program which loads test programs on Site controllers (UI - User Interface loads the program on Host computers). See the Overview in Binning.
- *MonitorApp* must be running before UI - User Interface will start.
- *MonitorApp* is started automatically when UseRel executes (normally when the user logs-in if the Nextest software is installed).
- *MonitorApp* is automatically terminated and restarted when UseRel or UseDLLs is executed.
- It may be necessary to manually terminate and restart the *MonitorApp* process when manipulating Environmental Variables. See Terminating & Restarting MonitorApp.

7.9.1 Terminating & Restarting *MonitorApp*

The following methods can be used to manually terminate *MonitorApp*:

- From a command line:
  ```
  monitorapp kill
  monitorapp
  ```
- Using the Windows Task Manager, locate and select the `MonitorApp.exe` process then click on the **End Process** button.

The following methods can be used to manually restart *MonitorApp*:

- From a command line:
  ```
  monitorapp
  ```
- Double click the *MonitorApp* icon from the *Bin* directory of the current software release
- Execute UseRel or UseDLLs from the *Utils* directory of the desired software release.

7.10 Environmental Variables

This section contains the following:

7.10.1 Nextest Environment Variables

In the context of Nextest software *Environment Variables* are used as noted:

Table 7.10.1.0-1 Nextest Environment Variables

Variable	Value	Purpose
DEFERRED_TOOL_PATH		Set to one or more locations (disk folder names) which are checked by UI software to identify User Tools which are to be automatically loaded. See ToolLauncher.
PATH		UseRel pre-pendeds the location of the Nextest software release in use to the PATH environment variable, which also contains other values set by other means. Both UI and Developer Studio depend on these Nextest settings.
PATTERN_PATH		Set to one or more locations (disk folder names) which are checked when loading logic and/or scan patterns. See Pattern Load PATH.
SIMULATED_APG	1	To simulate using Maverick-I APG features. Use only when SIMULATED_PTI = 1. Ignored when SIMULATED_HD = 1 and SIMULATED_HD = 2.
	2	To simulate using Maverick-II APG features. Use only when SIMULATED_PTI = 1. Ignored when SIMULATED_HD = 1 and SIMULATED_HD = 2.
SIMULATED_HD	3	Enable Magnum 2x simulation. Over-rides SIMULATED_PTI. See Magnum 1/2/2x Simulation Setup. Also enables ECR Simulation.
	2	Enable Magnum 2 simulation. Over-rides SIMULATED_PTI. See Magnum 1/2/2x Simulation Setup. Also enables ECR Simulation.
	1	Enable Magnum 1 simulation. Over-rides SIMULATED_PTI. See Magnum 1/2/2x Simulation Setup. Also enables ECR Simulation.
	0	Disable Magnum 1/2/2x Simulation.

Table 7.10.1.0-1 Nextest Environment Variables *(Continued)*

Variable	Value	Purpose
SIMULATED_LVM	1	Magnum 1/2/2x only. Enables use of LVMTool and vecdata()in simulation mode.
	0	Disable use of LVMTool and and vecdata() in simulation mode.
SIMULATED_PE	1	To simulate using Maverick-I Pin Electronics features. Ignored when SIMULATED_HD = 1 and SIMULATED_HD = 2.
	2	To simulate using Maverick-II Pin Electronics features. Ignored when SIMULATED_HD = 1 and SIMULATED_HD = 2.
SIMULATED_PTI	1	Must be set = 1 to simulate Maverick-I/-II. Ignored SIMULATED_HD = 1 and when SIMULATED_HD = 2.
SIMULATED_SITES	1 to 40	Magnum 1/2/2x only. Specifies the number of sites to be simulated. See Magnum 1/2/2x Simulation Setup.

7.10.2 Environmental Variable Scope

For environment variables to be useful the application which needs to access a given environment variable must execute within the scope of that variable. Similarly, before a change made to an environment variable value can be seen, any executing applications which are to use the new value must be terminated and restarted after the variable is set.

In the context of a Magnum 1/2/2x test program, the following hierarchy must be considered when the scope of environment variables is considered. These are listed in the order each process normally starts (test programs vs. User Tools don't matter):

- System Environment
- *MonitorApp*
- UI - User Interface
- Any test program which uses the environment variable
- Any User Tools which use the environment variable

- Developer Studio (Site and/or Host debug only)

The important dependencies can be stated as follows:

- The System Environment sets the initial values for defined environment variables. These can be reviewed using the Windows Control Panel: **`Start: Settings: Control Panel`**, locate and double-click on the *System* icon and select the Environment tab (NT). Using Windows 2000, or XP look harder.. its there somewhere.
- Except as noted below, *MonitorApp* loads test programs on Site (not Host) computer(s). *MonitorApp* inherits the value of environment variables from the environment in which it is invoked. Normally, *MonitorApp* is started automatically, when the user logs-in to the computer, thus all environment variables are normally obtained from the System Environment.
- It is possible to manually terminate and restart *MonitorApp*, using several methods (see Terminating & Restarting MonitorApp). Any changes made to environment variable(s) will be inherited by *MonitorApp* when it is started in the same scope in which the environment variable(s) were modified. For example, using the same command shell to both modify the variable(s) and restart *MonitorApp* will ensure proper inheritance. An example which won't ensure proper inheritance is to set the environment variable(s) from a command shell and restart *MonitorApp* from Explorer.
- UI - User Interface obtains the value of environment variables from the environment in which it is invoked. Normally, UI is started from the Start Menu and thus inherits environment variables from the System Environment. Since UI will not start unless *MonitorApp* is running any changes to environment variables can be properly inherited if both *MonitorApp* and UI are restarted within the same scope.
- Except as noted below, the Host copy of a test program is loaded by UI - User Interface, and thus inherits environment variables from UI.
- When Debugging With Developer Studio the Host and/or Site processes being debugged inherit environment variables from *Developer Studio*, which inherits from the System Environment if started from the Start Menu.
- User Tools inherit environment variables from the scope in which they are invoked, which can be from a shell, from User Menus in UI, or from a batch file. User Tools are an advanced feature in the Nextest software, and it is assumed that issues of environment variable scope are well known to the developer.

7.10.3 Setting Environment Variables

Environment variables can be set, or modified, using the following methods:

1. Control Panel Method. This is persistent, and affects all applications started, from the System Environment, after the variable is set. Applications already executing, including existing shells, *MonitorApp*, etc. will not see changes made.
2. Command Line Method. Persistent only to the command shell in which the variable is set. Any applications started from this shell inherit the variable. Applications already executing when the variable is set do not see the change.
3. Batch File Method. Persistent only for the duration of the batch file. Any applications started from the batch file inherit the variable. Applications already executing when the variable is set do not see the change.
4. C-code Method i.e. from test program code or User Tools code. Persistent for the duration of the program.

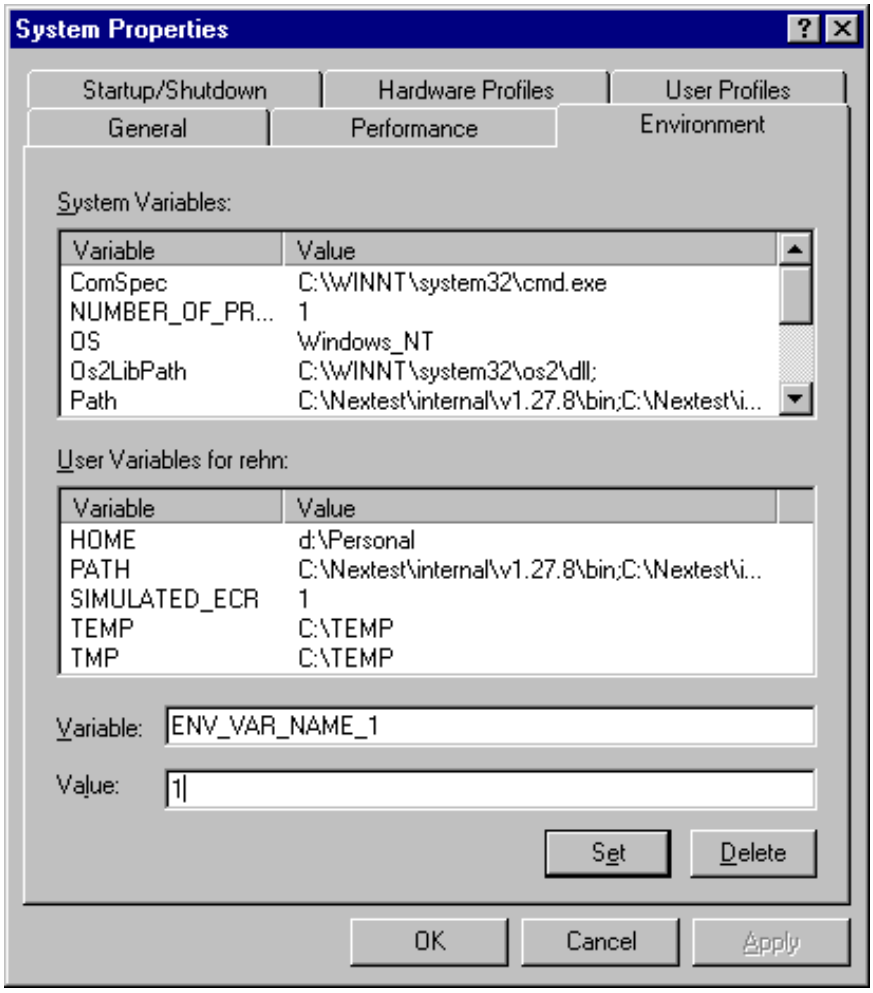
Control Panel Method

Note: the exact steps vary depending on which Windows operating system is in use. The description below applies to Windows NT.

The following method uses the System Control Panel to view, create, modify, or eliminate an environment variable. See Environmental Variable Scope.

1. Invoke the Windows Control Panel: **Start: Settings: Control Panel**
2. Locate and double-click on the *System* icon
3. Select the Environment tab
4. In the dialog, to modify an existing environment variable select it in the "**User Variables for ...**" window. To create a new variable type its name in the **Variable** text window.
5. In the **Value** window enter the desired value.
6. Click **Set** to create or modify the environment variable.
7. Repeat for additional variables. Click **OK** when done.

The example below creates, or modifies, the environment variable named ENV_VAR_NAME_1 and sets its value = 1:



Command Line Method

Type the following commands to set (or modify) the two environmental variables from a command line. See Environmental Variable Scope.

```
monitorapp kill
set ENV_VAR_NAME_1=1
set ENV_VAR_NAME_2=0
monitorapp
```

This example also terminates and restarts *MonitorApp* to ensure the next invocation of UI - User Interface will inherit the new environment variable values.

To completely eliminate an environment variable from the command line type:

```
set ENV_VAR_NAME_1=
```

Note there is no value after the equals sign.

Batch File Method

This example batch file sets two environment variables, and also terminates and restarts *MonitorApp* to ensure the next invocation of UI - User Interface will inherit the new environment variable values (see Environmental Variable Scope). The methods used to invoke a batch file are not documented here.

```
File: some_batch_file.bat
   monitorapp kill
   set ENV_VAR_NAME_1=1
   set ENV_VAR_NAME_2=0
   monitorapp
```

To completely eliminate an environment variable from a batch file:

```
   set ENV_VAR_NAME_1=
```

Note there is no value after the equals sign.

C-code Method

The following code creates, or modifies, the environment variable named ENV_VAR_NAME_1, from C code. This could be test program code and/or User Tools code:

```
// Set or modify ENV_VAR_NAME_1
   int putenvOK = putenv( "ENV_VAR_NAME_1=1" );
   if (putenvOK == -1)
      output ("ERROR: putenvOK( ENV_VAR_NAME_1=1 )");
```

The following code gets the current value of the environment variable named ENV_VAR_NAME_1, from C code. This could be test program code and/or User Tools code:

```
// Get ENV_VAR_NAME_1 value
   char *env_val;
   env_val = getenv( "ENV_VAR_NAME_1" );
```

```
if( env_val != NULL )
   output( " ENV_VAR_NAME_1 => %s", env_val );
```

To completely eliminate an environment variable from C code:

```
envOK = putenv( "ENV_VAR_NAME_1=" );
```

Note there is no value after the equals sign.

7.11 Invoking a File Browser

Description

The `get_open_file_name()` function is used to invoke a standard Windows file browser dialog suitable for selecting a file to be opened, allowing subsequent user-code to read from the file. The function checks that the specified path and file name exist and display an error if not. A `CString` value is returned containing the full path/file name selected. If **`Cancel`** is invoked from the dialog a NULL string is returned.

The `get_save_file_name()` function is used to invoke a standard Windows file browser dialog suitable for selecting a file to be saved, allowing subsequent user-code to write to the file. The function checks if the specified file exists and prompts for over-write confirmation if it does. A `CString` value is returned containing the full path/file name selected. If over-write is denied or if **`Cancel`** is invoked a NULL string is returned.

Usage

```
CString get_open_file_name();
CString get_save_file_name();
```

where both functions return the path/filename of the file selected in the browser or NULL. See Description.

Example

```
CString infile = get_open_file_name();
CString outfile = get_save_file_name();
```

7.11.1 Obsolete: current_dialog()

In very early test programs the function `current_dialog()` was commonly used as an argument to the following functions:

```
update_value()
update_control()
get_HWND()
immediate()
hex_display()
```

This argument was required to allow these functions to identify to which user dialog a given user variable was linked. This method is no longer required since each user variable now knows to which dialog it is linked.

This change allows these functions to be simplified. Below are examples of both the old methods (obsolete) and new methods (recommended):

```
old: update_value( current_dialog( ), variable );
new: update_value( variable );

old: update_control( current_dialog( ), variable );
new: update_control( variable );

old: HWND h = get_HWND( current_dialog( ), variable );
new: HWND h = get_HWND( variable );

old: immediate( current_dialog( ), variable, TRUE );
new: immediate( variable, TRUE );

old: hex_display( current_dialog( ), variable, TRUE );
new: hex_display( variable, TRUE );
```

In the situation where a pointer to a user dialog is needed (for some other application) the following can be used:

```
old: Dialog *d = current_dialog();
new: Dialog *d = current_dialog( variable );
```

It is recommended that return values from these functions are checked. For example:

```
old:
if ( ! update_value( current_dialog(), variable ) )
   warning("couldn't update variable %s in dialog %s",
            resource_name( variable ),
            resource_name( current_dialog() ) );

new:
if ( ! update_value( variable ) )
   warning(  "couldn't update variable %s in dialog %s",
              resource_name( variable ),
              resource_name( current_dialog( variable ) ) );
```

Or, more succinctly:

```
old: VERIFY( update_value( current_dialog(), variable ) );
new: VERIFY( update_value( variable ) );
```

7.12 DUT Board TDR Functions

Overview

Time Domain Reflectometry (TDR) is a technique used to measure the electrical path length of a transmission line.

TDR is used in Nextest software to measure the electrical path length of DUT board signal pin interconnects. The `DutBoardTDR.exe` program is included in each Nextest software release for this purpose. The resulting TDR values (deskew values) are stored in an EEPROM located on the standard DUT boards. Any time a test program is loaded, the system software automatically reads the DUT board EEPROM and uses the TDR values to

adjust user programmed timing values to compensate for DUT board signal path delays. By doing so, drive and strobe signals appear at the DUT at the programmed time. This concept is often called *deskewing the DUT board*.

The `DutBoardTDR` program is designed to TDR a passive 50-ohm transmission line which is open-circuit at the DUT socket. If a given pin connection contains relays in the signal path, or other passive or active circuitry in the signal path, TDR will not operate as desired, and using the resulting TDR value will result in inaccurate timing. To handle this situation, a user-written test program must be executed to modify the values stored in the DUT board EEPROM for pins which do not meet the TDR requirements noted above. This can be a special test program executed solely for this purpose, or the necessary code can be put into all test programs. In either case, the functions and macros documented in this section are used.

The following functions and macros are documented in detail later in this section:

- `TDR_BLOCK()`: all TDR operations must occur within a `TDR_BLOCK()`.
- `db_tdr()`: invoke TDR on a specified list of pins.
- `db_read_tdr()`: read the DUT board EEPROM into a TDRarray, a software structure local to the `TDR_BLOCK()`.
- `db_write_tdr()`: write the contents of the TDRarray to the DUT board EEPROM. Only the EEPROM contents affect user timing values. Note that the `db_write_tdr()` function can only be executed *ONE TIME* in the program.
- `db_set_tdr(), db_get_tdr()`: explicitly set or get the TDR value for one specified pin. This accesses the software TDRarray only.
- `db_get_pins()`: read the DUT board EEPROM and get the number of pins for which TDR data is currently stored in the EEPROM.
- `db_get_date()`: read the DUT board EEPROM and get the date on which the EEPROM was last written.

Using these functions, the user's test program can properly configure any DUT board relays, perform TDR on all or some pins, use user-written deskew methods to compensate for any active circuitry on the DUT board, set special deskew values for special pins, etc.

Performing TDR from the test program can also be used as a DUT board diagnostic. User code can read and store values from the DUT board EEPROM, perform a new TDR, and compare EEPROM values with the new values to detect when signal connections have changed significantly. A shorted pin or resistive load will show up as an unexpectedly large TDR measurement. A broken connection may show up as a smaller than expected TDR measurement (smaller than a good connection) but this depends upon where the defect exists relative to the DUT socket.

Note: except as noted, the functions noted above MUST only be executed within the TDR_BLOCK(). Executing these functions elsewhere in a test program may overwrite user specified timing and voltage settings or have other undesirable side effects. The system software does not check this rule.

7.12.1 TDR_BLOCK()

See DUT Board TDR Functions.

The TDR_BLOCK( ) macro was created to allow user-written code to execute the TDR related functions at the proper time during the test program load and initialization sequence.

Note: except as noted, all TDR related functions ***MUST*** be executed within the TDR_BLOCK. User-defined timing will be corrupted if this rule is violated, with unpredictable (undesirable) test results. The system software does not check this rule.

Usage

The functionality of the TDR_BLOCK is outlined below:

1. During the program load process the optional TDR_BLOCK executes after CONFIGURATION(), and SITE_CONFIGURATION(), and before SITE_BEGIN_BLOCK(). The TDR_BLOCK macro only executes in Site processes.
2. In the TDR_BLOCK, before any user code executes, the system software creates a software structure, called TDRarray, and initializes it to contain 0nS deskew values for all pins. The 0nS values are only set once, on entry to the TDR_BLOCK. The db_tdr(), db_read_tdr(), and db_set_tdr() functions can subsequently change the contents of TDRarray.
3. The TDRarray is temporary storage whose scope is limited to the TDR_BLOCK itself. It is not accessible except using the functions documented in this section.
4. The code in the TDR_BLOCK code is then executed. This is the opportunity for user code to configure the DUT board, execute TDR functions as needed to define a complete set of TDR data for the target DUT board, put explicit values in the TDRarray, etc.

Note: within the TDR_BLOCK, user code must ***NOT*** modify any pin voltages or timing values. The quality of Nextest TDR operations depends upon voltage/timing values set up by Nextest software.

Note: withing the TDR_BLOCK, user code must ***NOT*** configure the ECR (i.e. must not use `ecr_config_set()`) or modify the APG branch-on-error source (don't execute `mar_error_choice_set()`).

5. Last, user code optionally executes `db_write_tdr()` to cause TDR data to be written to the DUT board EEPROM. Remember, only the contents of the DUT board EEPROM is used to establish final user timing values.

Note: user code is responsible for ensuring that the TDRarray contains valid deskew timing values for all used signal pins ***BEFORE*** `db_write_tdr()` is executed. The system software does not check and `db_write_tdr()` can only be executed once without unloading and reloading the test program.

Note: valid TDR values are between -10nS and +10nS. Values outside these limits ***MUST NOT*** be used. Violating this rule can cause timing edges to occur at incorrect times. The system software does not check this rule.

Note: the db_write_tdr() function must only be executed once within the TDR_BLOCK. Violation of this rule can result in inaccurate user timing values. The system software does not check this rule.

6. After the TDR_BLOCK execution ends the system software reads the TDR data from the DUT board EEPROM and initializes the system timing tables using these TDR values.

The test program can define multiple TDR blocks, for example, to account for different DUT board designs (rarely necessary). When this is done, one TDR block must be selected, using the USE_TDR_BLOCK( ) macro, in a `CONFIGURATION()` or `SITE_CONFIGURATION()` block. If this is not done, the system software will present a dialog, requiring the user to make the selection. This can be quite obnoxious since a dialog is presented for each site in the system.

Example

This example uses most of the functions documented in this section and serves as the example for those functions. Note the following:

- The example includes a TDR_BLOCK, a user variable, and a local C-function. The latter two are not required, but are included to show how errors might be handled.
- The TDR_BLOCK code invokes TDR on a pin list named *all_pins*.
- When the user code in the TDR_BLOCK detects an error two things happen:
  - A message is output in the appropriate UI Controller window
  - A message dialog is displayed in the Host process.
  If many errors occur this error dialog could become quite annoying.
- Errors reported include:
  - A TDR measurement fails given the min/max search range values specified for db_tdr().
  - A measured TDR value is outside a separate set of user-defined limits.
  - An error occurred when writing the final TDR values to the EEPROM on the DUT board.
- Some of the code below is architected to make this documentation more readable.

```
#define HOST 0 // For code readability only

// UserVariable which displays an error dialog on Host using
// message sent from Site code
CSTRING_VARIABLE( HostMessage, "", "" ) {
   AfxMessageBox( HostMessage );
}

// RemoteDialog() is a local function used to send an error
// message from Site to Host for display in an error dialog
void RemoteDialog( LPCTSTR message ) {
   HostMessage = message;
   remote_send( HostMessage, HOST, TRUE, INFINITE );
}

TDR_BLOCK( tdr ) {

   CString msg; // For readability in this documentation

   // TDR search range min/max limits (reused later)
   double min_limit = -10.0 NS;
   double max_limit = -1.0 NS;
```

```
   // db_tdr() performs TDR on the pin list = all_pins
   output ("Executing db_tdr()...");
   BOOL tdr_OK = db_tdr( all_pins, min_limit, max_limit );
   HDTesterPin tpin;
   if( ! tdr_OK ) {
      output("Bad TDR measurement(s) on Site- %d", site_num() );
      for( int index = 0;pin_info( all_pins, index, &tpin ); // See
pin_info()
           ++index ) {
         double result = db_get_tdr( tpin ); // db_get_tdr()
         if( result == 0 )
            output("  Bad pin => %s", testerpin_name( tpin ) );
      }
      msg = "db_tdr() returned FALSE on Site-";
      msg += vFormat( "%d\n", site_num() );
      msg += "See messages in Site window";
      HostMessage = msg;
      remote_send( HostMessage, HOST, TRUE, INFINITE );
   }
   else{   // db_tdr() returned TRUE
           // Check measured TDR values against user limits
      BOOL ok = TRUE;
      min_limit = -3.4 NS;
      max_limit = -2.5 NS;
      for(    int index = 0;
              pin_info( all_pins, index, &tpin ); // See pin_info()
              ++index ) {
         double result = db_get_tdr( tpin ); // db_get_tdr()
         if(  ( result < min_limit ) ||
              ( result > max_limit )) {
            ok = FALSE;
            msg = vFormat("Bad TDR value on Site- %d", site_num() );
            msg += vFormat(", tester pin: %s.",
                 testerpin_name( tpin ) );
            HostMessage = msg;
            remote_send( HostMessage, HOST, TRUE, INFINITE );
         }
      }
      if( ok ){ // Program DUT board EEPROM
         BOOL EE_write_OK = db_write_tdr();
```

```
        if( ! EE_write_OK ) { // The EEPROM write had an error...
           msg = "TDR write to DUT board EEPROM failed on Site-";
           msg += vFormat("%d\n", site_num());
           output("%s", msg);
           HostMessage = msg;
           remote_send( HostMessage, HOST, TRUE, INFINITE );
        }
     }
  }
}
```

7.12.2 db_tdr()

See DUT Board TDR Functions.

Description

The `db_tdr()` function is used to invoke TDR on a user specified pin list. Note the following:

- `db_tdr()` is only usable in the scope of a `TDR_BLOCK`, as described in `TDR_BLOCK()`.
- The specified pin list can be any subset of tester PE channels, including all pins.
- The TDR results are stored in the TDRarray, as described in `TDR_BLOCK()`. The TDR result is a time value relative to a default of zero, which represents the reference established by the Nextest timing calibration (TCAL) program (*TimingCal.exe*) using the Nextest TCAL DUT Board. Smaller numbers (more negative) indicate shorter path lengths. Larger numbers (more positive) indicate longer path lengths.
- The `db_tdr()` function can be executed any number of times within the `TDR_BLOCK()`, as needed to collect the necessary deskew data for the necessary pins. If a given pin is TDR'ed more than once any previous value in the TDRarray for that pin are overwritten by the new value.

Note: it is the responsibility of user code to ensure that all appropriate pins have valid deskew data stored in the TDRarray before writing the data to the DUT board EEPROM using `db_write_tdr()`. The system software does not check this rule.

Note: valid TDR values are between -10nS and +10nS. Values outside these limits ***MUST NOT*** be used. Violating this rule can cause timing edges to occur at incorrect times. The system software does not check this rule.

- The `db_tdr()` function requires that two limit values be specified. These represent the user-defined minimum and maximum TDR value expected for the pins being deskewed. The `db_tdr()` function returns TRUE if the TDR measurement(s) for all specified pins are between these limits, otherwise FALSE is returned and an error message is output in the appropriate UI Controller window. A separate error message is output for each pin failing these limits.

Note: even though a given TDR value is outside the specified limits it is stored in the TDRarray and will be written to the DUT board EEPROM if `db_write_tdr()` is executed.

- `db_tdr()` takes approximately 30 seconds to complete on a single-site system. Up to several minutes are required for multi-site systems.

Usage

```
BOOL db_tdr( PinList* pPinList, double min_val, double max_val );
```

where:

`pPinList` specifies the pins to be TDR'ed.

`min_val` and **`max_val`** specify the user-defined minimum and maximum TDR limits. These determine whether `db_tdr()` returns TRUE, i.e. the TDR measurement(s) for all pins in **`pPinList`** are between these limits, or FALSE. An error message is output in the appropriate UI Controller window for each pin failing these limits.

Example

See Example.

7.12.3 db_read_tdr()

See DUT Board TDR Functions.

Description

The db_read_tdr() function reads the DUT board EEPROM to retrieve the embedded date and generate a checksum of the TDR data. Note the following:

- db_read_tdr() is only usable in the scope of a TDR_BLOCK, as described in TDR_BLOCK().
- db_read_tdr() checks that the date stored with the TDR data is valid.
- Using a checksum, db_read_tdr() checks that the EEPROM data is valid.
- If the EEPROM data is valid, it is stored in the TDRarray noted in TDR_BLOCK(), and db_read_tdr() returns TRUE. If the data is invalid, the TDRarray is not modified.
- The scope of the TDRarray is limited to the TDR_BLOCK.

After the db_read_tdr() function has successfully read the EEPROM into the TDRarray the db_get_pins() function may be used to determine how many pins have TDR data in the TDRarray, and db_get_tdr() may be used to get the TDR value for one pin. To save a copy of the entire TDRarray requires that user code define an appropriate data structure and use db_get_tdr() in a loop to get data for the desired pins.

Usage

```
BOOL db_read_tdr();
```

where:

db_read_tdr() returns TRUE or FALSE as noted in Description.

Example

The following example will operate correctly only within the scope of a TDR_BLOCK, as described in TDR_BLOCK():

```
// Read EEPROM into TDRarray
BOOL EE_read_OK = db_read_tdr();

if ( ! EE_read_OK )
   output("ERROR: db_read_tdr() returned FALSE");
else {
   double val;
   HDTesterPin tpin;
   for (   int index = 0;
           pin_info( all_pins, index, &tpin );
           ++index ) {
```

```
      // See db_get_tdr()
      val = db_get_tdr( tpin );
      output( " TDR measurement for pin => %s = %5.3f",
         testerpin_name( tpin ), val );
   }
}
```

7.12.4 db_write_tdr()

See DUT Board TDR Functions.

Description

The `db_write_tdr()` function writes the contents of the TDRarray to the DUT board EEPROM. The current date/time and a checksum are also written. Note the following:

- `db_write_tdr()` is only usable in the scope of a `TDR_BLOCK`, as described in `TDR_BLOCK()`.
- Writes the contents of the TDRarray to the DUT board EEPROM. The current date/time and a checksum are also written.
- The TDR data is read back from the EEPROM and compared it with the contents of the TDRarray. If everything matches `db_write_tdr()` returns `TRUE`, otherwise `FALSE` is returned.
- For the contents of the TDRarray to have any effect on the user's test program the TDR data must be written to the EEPROM, using `db_write_tdr()`. The TDRarray is temporary storage whose scope is limited to the `TDR_BLOCK()` itself.
- User code is responsible for ensuring that the TDRarray contains valid deskew timing values for all used signal pins ***BEFORE*** `db_write_tdr()` is executed. The system software does not check.

Note: valid TDR values are between -10nS and +10nS. Values outside these limits ***MUST NOT*** be used. Violating this rule can cause timing edges to occur at incorrect times. The system software does not check this rule.

Note: the db_write_tdr() function must only be executed once within the TDR_BLOCK. Violation of this rule can result in inaccurate user timing values. The system software does not check this rule.

Usage

```
BOOL db_write_tdr();
```

where:

db_write_tdr() returns TRUE or FALSE as noted in Description.

Example

See Example

7.12.5 db_set_tdr(), db_get_tdr()

See DUT Board TDR Functions.

Description

The db_set_tdr() function sets a value in the TDRarray for the specified pin. This can be used to place a user-defined deskew value in the TDRarray before it is written to the EEPROM using db_write_tdr().

Note: valid TDR values are between -10nS and +10nS. Values outside these limits ***MUST NOT*** be used. Violating this rule can cause timing edges to occur at incorrect times. The system software does not check this rule.

The db_get_tdr() function gets and returns a value from the TDRarray. This can be used when making a copy of a TDRarray, for performing comparisons between values, etc.

Note the following:

- These functions are only usable in the scope of a TDR_BLOCK, as described in TDR_BLOCK().

- The values in the TDRarray are set to 0nS at the start of the TDR_BLOCK execution, and can subsequently be modified using db_tdr(), db_read_tdr(), and db_set_tdr().
- The TDRarray is temporary storage whose scope is limited to the TDR_BLOCK() itself.

Usage

```
void db_set_tdr( HDTesterPin Pin, double Value );
double db_get_tdr( HDTesterPin Pin );
```

where:

Pin specifies one tester pin and must be one of the HDTesterPin enumerated types in the range of . Values must be signal pins (not DPS, etc.)

Value is the desired deskew value to be set in the TDRarray. Units may be used (see Specifying Units).

db_get_tdr() returns the deskew value from the TDRarray for the specified pin.

Example

See Example

7.12.6 db_get_pins()

See DUT Board TDR Functions.

Description

The db_get_pins() function reads the TDRarray (see TDR_BLOCK()) and returns a count of the number of pins of TDR data stored in the EEPROM.

The number of pins returned will be 128 pins * the value set by sites_per_controller().

Usage

```
int db_get_pins();
```

where:

`db_get_pins()` returns a value as noted in the Description.

Example

```
int count = db_get_pins();
output("DUT board EEPROM contains %d TDR values", count );
```

7.12.7 db_get_date()

See DUT Board TDR Functions.

Description

The `db_get_date()` function returns the date/time the DUT board EEPROM was last programmed with data from the TDRarray (see `TDR_BLOCK()`).

Usage

```
CString db_get_date();
```

where:

`db_get_date()` returns the date as a `CString`, in the form of "`mm/dd/yyyy/ hh:mm`". An empty string is returned if the EEPROM date is not valid or otherwise not readable. See `db_read_tdr()`.

Example

```
CString date_time = db_get_date();
output(" EEPROM was last programmed => %s", date_time );
```

7.13 Miscellaneous

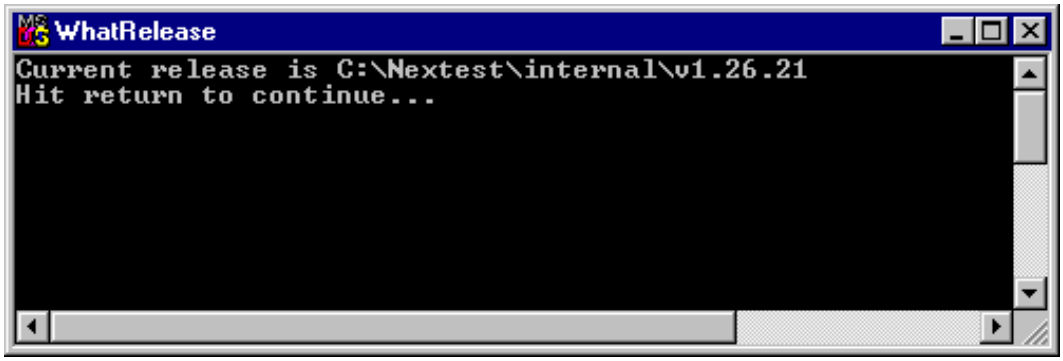
7.13.1 WhatRelease

Definition

The command "`WhatRelease`" is added to the *Windows Start* menu when the Nextest software is installed using UseRel or by executing the self extracting archive which installs the software.

Executing "`WhatRelease`" executes the `.../Utils/WhatRelease.bat` script, which displays a window showing the version of the software active on that computer.

The example below shows that software revision **`v1.26.21`** is currently active.



7.13.2 UseRel

Description

The `UseRel` batch file (*...release\Utils\UseRel.exe*) is used to select a Nextest software release and start the various processes needed to use that release.

Executing `UseRel` from a given software release enables that release for use. To switch to a different software release requires executing `UseRel` again, from the desired release's *Utils* directory.

Before executing `UseRel` it is necessary to terminate any instances of UI - User Interface and Developer Studio.

Executing `UseRel.bat` from the target release's *Utils* folder does the following:

1. Updates the PATH environment variable to point to the target release. This also removes any PATH components pointing to a previously used software release.
2. Updates the *Developer Studio* registry keys, to point to the target release's *include* files and libraries.
3. Updates the *Windows Start* menu to execute the target release's version of UI and point to the target release's version of the Programmer's Manual.
4. Kills *MonitorApp* if it is running, and starts the target release's version.

Note: `UseRel` ***does not*** update any shortcuts created by the user, whether on the desktop or elsewhere.

In software release h2.2.xx/h1.2.xx `UseRel` was updated to prompt the user to execute StartServer. This occurs when `UseRel` is executed on a Host connected to a test system.

7.13.3 UseDLLs

Description

The `UseDLLs` batch file can be executed to start UI - User Interface using a different software release than is currently set up i.e. without running UseRel (and rebooting on multi-site systems).

`UseDLLs` only affects the current UI - User Interface session that it starts i.e. it affects program loading and execution only, not compiling programs. When a test program is loaded using this instance of UI the DLLs from the new release will be used. Terminating UI completes the operation of `UseDLLs.bat`, leaving none of the changes behind.

Note: since the *Windows Start* menu is ***not*** modified, invoking UI from the start menu will continue to use the software release set up the last time UseRel was executed. To use the alternate (new) release again, execute `UseDLLs.bat` again.

To execute UseDLLs use the *Windows Explorer*, and locate the desired (target) software release. UseDLLs.bat is located in the release Utils folder, i.e. in the same location as UseRel.bat. Double-click on UseDLLs.bat to execute it and start UI using the target software release.

To use UseDLLs.bat with a software release which didn't originally contain it, UseDLLs.bat can be executed from a command line, with a command-line argument that should be the full path to the desired release.

Executing UseDLLs.bat from the target release's *Utils* folder (or command line) does the following:

1. Restarts the target release's version of *MonitorApp*.
2. On a VT and GT, restarts *MonitorApp* on each Site controller using the PATH to the target release.
3. Starts the target release's version of UI - User Interface. Any test programs loaded using this instance of UI will use the DLLs from the target release. Terminating this instance of UI completes the operation of UseDLLs.bat, leaving no changes behind.

7.13.4 Automatic Stack Trace Generator

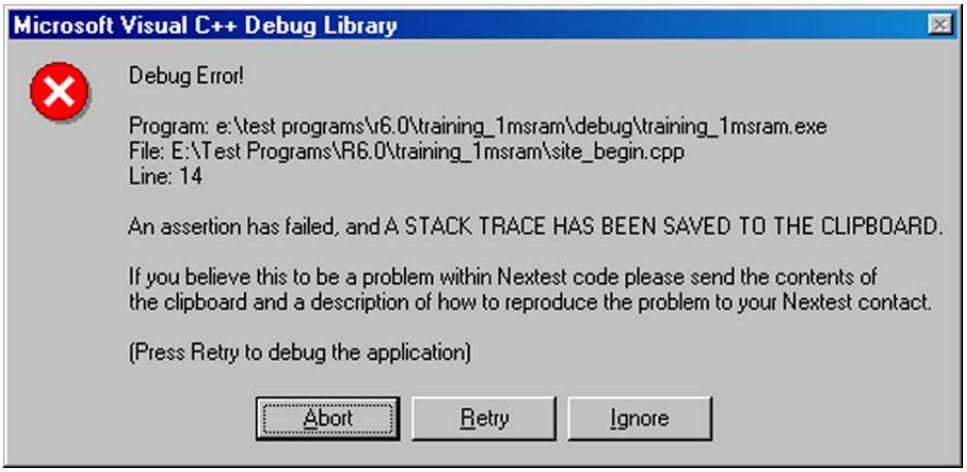
Note: new in software release v.2.4.4.

The MFC ASSERT( ) function is used in software to detect error conditions. It can be used in any code. Nextest uses the ASSERT( ) function to make sure certain error conditions cannot be ignored. Users can use the ASSERT( ) function as desired.

During program execution, when an ASSERT( ) is triggered two things occur:

- A warning dialog is displayed, showing which source file contained the triggered ASSERT( ). This allows the user to determine whether the ASSERT( ) came from user-written code or Nextest (or Microsoft) written code.
- A *stack trace* is automatically generated and stored on the clipboard. This stack trace shows the hierarchy of function calls which occurred up to the ASSERT( ).

A typical ASSERT( ) warning dialog is shown below:



In this example, note that the source code file which contained the ASSERT( ) function was user-written, i.e.

```
E:\TestPrograms\R6.0\BMT_1msram\site_begin.cpp
```

Below is the first line of the stack trace from the ASSERT( ) noted above:

```
1 static ____sbb_1msram_SiteBeginBlock_callback()
[E:\Test Programs\R6.0\BMT_1msram\site_begin.cpp:34 + 30]
```

This also shows the source file which contained the ASSERT( ), which was located on line 34 of that file.

When an ASSERT( ) is triggered, if it is *NOT* triggered from user-written code, the information in the stack trace should be forwarded to Nextest, typically by email to user's Applications Support engineer. This helps Nextest to evaluate the nature of the error, and determine whether any software changes are required.

Note: if the ASSERT( ) is triggered from user-written code, Nextest cannot assist in evaluating the problem.

When an ASSERT( ) occurs, to capture the stack trace do the following:

- Do ***NOT*** terminate the ASSERT( ) dialog. The stack trace is captured to the clipboard, and is cleared when the dialog is terminated.
- In a text editor of your choice, perform **`Edit->Paste`** into a blank page. The editor can be Notepad, Wordpad, Word, etc. If the ASSERT( ) dialog appeared in a site window on a multi-site system (ST, SST, VT, GT) many of the standard editor applications are not available - use Developer Studio.

- Save the file to disk.
- Email the file to your Nextest applications engineer. Be sure to include details about how the problem can be reproduced, which software release was in use, etc.


NEXTEST™
SYSTEMS CORPORATION

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z