

# eTPU Simulator Reference Manual

*by*

*John Diener, Andrew Klumpp, Tony Zabinski,  
Keith Kroker, and Michael Schwager*

ASH WARE, Inc.

Version 4.61

2012-4-3

(C) 1994-2012 ASH WARE, Inc



ASH WARE Inc.



# Table of Contents

Foreword	9
<b>Part 1 Overview</b>	<b>11</b>
1.1 On-Line Help Contents .....	14
<b>Part 2 Demo Descriptions</b>	<b>15</b>
<b>Part 3 Software Upgrades</b>	<b>19</b>
3.1 Version 3.0 to 4.3 Enhancements .....	21
<b>Part 4 The Project File</b>	<b>27</b>
<b>Part 5 Source Code Files</b>	<b>29</b>
5.1 Source Code Search Rules .....	30
<b>Part 6 Script Commands Files</b>	<b>33</b>
6.1 The "ETEC_cpp.exe" Preprocessor .....	34
6.2 The Primary Script Command Files .....	35
6.3 ISR Script Commands Files .....	36
6.4 The Startup Script Commands File .....	38
6.5 The Build Script File .....	38
6.6 File Format and Features .....	39
Script Directives, Define, Ifdef, Include .....	40
Script Directives, Define, Ifdef, Include .....	41
Script Enumerated Data Types .....	42
Script Integer Data Types .....	42
Referencing Memory in Script Files .....	43
Assignments in Script Commands Files .....	43
Operators and expressions in Script Commands Files .....	44
Syntax for global access of eTPU Function Variables .....	45
Syntax for eTPU Channel Hardware Access .....	45

Syntax for eTPU ALU Register Access .....	47
String within a string supports formatted symbolic information .....	48
Comments in Script Commands Files .....	49
Decimal, Hexadecimal, and Floating Point Notation in Script Files .....	49
String Notation .....	49
<b>6.7 Script Commands Groupings .....</b>	<b>50</b>
Clock Control Script Commands .....	51
Timing Script Commands .....	52
Verify Timing Script Commands .....	53
Memory Modify Script Commands .....	54
Memory Verify Script Commands .....	54
Register Write Script Commands .....	55
Register Verify Script Commands .....	56
Symbol Write Script Commands .....	56
Verify Symbol Value Script Commands .....	58
System Script Commands .....	59
File Script Commands .....	61
Trace Script Commands .....	63
Code Coverage Script Commands .....	64
Channel Function Select Register Commands .....	68
Channel Priority Register Commands .....	68
Pin Control Script Commands .....	68
Pin Transition Behavior Script Commands .....	69
Thread Script Commands .....	72
Disable Messages Script Commands .....	73
eTPU System Configuration Commands .....	73
eTPU Timing Configuration Commands .....	74
eTPU STAC Bus Script Commands .....	75
eTPU Global Data Write/Verify Commands .....	76
eTPU Channel Data Script Commands .....	78
eTPU Channel Address Script Commands .....	79
eTPU Engine Data Script Commands .....	79
eTPU Channel Function Mode Script Command .....	81
eTPU Event Vector Entry Condition (Standard/Alternate) Commands .....	81
eTPU Interrupt Script Commands .....	82
eTPU/TPU Host Service Request Register Script Commands .....	83
eTPU/TPU Interrupt Association Script Commands .....	83
External Logic Commands .....	84
Build Script Commands .....	85
<b>6.8 Automatic and Pre-Defined Define Directives .....</b>	<b>91</b>
<b>6.9 Listing of Script Enumerated Data Types .....</b>	<b>94</b>
Script FILE_TYPE Enumerated Data Type .....	94
Script VERIFY_FILES Enumerated Data Type .....	94
Script FILE_OPTIONS Enumerated Data Type .....	95

---

Trace Options Enumerated Data Types .....	96
Base Time Options Enumerated Data Type .....	96
Build Script TARGET_TYPE Enumerated Data Type .....	97
Build Script TARGET_SUB_TYPE Enumerated Data Type .....	97
Build Script ADDR_SPACE Enumerated Data Type .....	98
Build Script READ_WRITE Enumerated Data Type .....	99
eTPU Register Enumerated Data Types .....	100
<b>Part 7 Trace Buffer and Files</b>	<b>103</b>
<b>Part 8 Test Vector Files</b>	<b>105</b>
8.1 Node Command .....	108
8.2 Group Command .....	109
8.3 State Command .....	110
8.4 Frequency Command .....	110
8.5 Wave Command .....	110
8.6 Engine Example .....	111
<b>Part 9 Functional Verification</b>	<b>115</b>
9.1 Data Flow Verification .....	116
9.2 Pin Transition Behavior Verification .....	117
9.3 Code Coverage Analysis .....	122
9.4 Regression Testing (Automation) .....	125
9.5 Console Mode .....	126
9.6 Command Line Options .....	126
Using the -d (define) Option and Escape Characters .....	128
9.7 File Location Considerations .....	129
9.8 Test Termination .....	130
9.9 Cumulative Logged Regression Testing .....	130
9.10 Regression Test Example .....	131
<b>Part 10 Action Tags</b>	<b>133</b>
10.1 Print Action Tag .....	134
10.2 Timer Action Commands .....	134

---

10.3	Write Value Action Tag .....	135
<b>Part 11</b>	<b>External Logic Simulation</b>	<b>137</b>
<b>Part 12</b>	<b>Integrated Timers</b>	<b>141</b>
<b>Part 13</b>	<b>Workshops</b>	<b>143</b>
<b>Part 14</b>	<b>The Logic Analyzer</b>	<b>145</b>
14.1	Executing to a Precise Time .....	146
14.2	Waveform Selection .....	147
14.3	The Active Waveform .....	148
14.4	Left and Right Vertical Cursors .....	148
14.5	Displaying Behavior Verification Data .....	150
14.6	Mouse Functionality .....	150
14.7	Vertical Yellow Context Time Cursor .....	151
14.8	Scroll Bars .....	152
14.9	Display Pane Boundary Time Indicators .....	153
14.10	Data Storage Buffer Start Indicator .....	153
14.11	Current Time Indicator .....	153
14.12	Auto Scroll .....	153
14.13	Button Controls .....	153
14.14	Timing Display .....	154
14.15	Data Storage Buffer .....	155
<b>Part 15</b>	<b>Operational Status Windows</b>	<b>157</b>
15.1	Generic Windows .....	157
Source Code Windows	.....	158
Script Commands Window	.....	160
Watch Windows	.....	161
Local Variable Windows	.....	164
Call Stack Window	.....	165
Thread Window	.....	166

---

Trace Window .....	170
Complex Breakpoint Window .....	172
Memory Dump Window .....	173
Timers Window .....	174
<b>15.2 eTPU-Specific Windows .....</b>	<b>175</b>
eTPU Configuration Window .....	176
eTPU Channel Frame Window .....	177
eTPU Global Timer and Angle Counters Window .....	178
eTPU Host Interface Window .....	180
eTPU Channel Hardware Window .....	180
eTPU Scheduler Window .....	183
eTPU Execution Unit Registers Window .....	184

## **Part 16 Dialog Boxes 185**

16.1 File Open, Save, and Save As Dialog Boxes .....	185
16.2 Auto Build Batch File Options Dialog Boxes .....	187
16.3 Goto Time Dialog Box .....	189
16.4 Goto Angle Dialog Box .....	190
16.5 Occupy Workshop Dialog Box .....	191
16.6 IDE Options Dialog Box .....	191
16.7 Workshop Options Dialog Box .....	193
16.8 MtDt Build Options Dialog Box .....	194
16.9 Message Options Dialog Box .....	195
16.10 Source Code Search Dialog Box .....	197
16.11 Reset Options Dialog Box .....	199
16.12 Logic Analyzer Options Dialog Box .....	200
16.13 Channel Group Dialog Box .....	201
16.14 Complex Breakpoint Conditional Dialog Box .....	201
16.15 Trace Options Dialog Box .....	201
16.16 Local Variable Options Dialog Box .....	203
16.17 License Options Dialog Box .....	204
16.18 Memory Tool Dialog Box .....	204
16.19 Insert Watch Dialog Box .....	205
16.20 Watch Options Dialog Box .....	206
16.21 The 'About' Dialog Box .....	206

<b>Part 17 Menus</b>	<b>207</b>
17.1 Loading Files Menu .....	207
17.2 Stepping MtDt .....	209
17.3 Running MtDt .....	211
17.4 Breakpoints .....	212
17.5 Workshops .....	213
17.6 Opening View Windows .....	214
17.7 Window Options Menu .....	214
17.8 Setting Simulator Options .....	215
17.9 Help Menu .....	217
<b>Part 18 Hot Keys, Toolbar, Status Window</b>	<b>219</b>
<b>Part 19 Supported Targets and Available Products</b>	<b>221</b>
19.1 eTPU/CPU System Simulator .....	221
19.2 eTPU2 Stand-Alone Simulator .....	222
19.3 eTPU Stand-Alone Simulator .....	222
19.4 eTPU2 Simulation Engine Target .....	222
19.5 eTPU Simulation Engine Target .....	222
<b>Part 20 Building the Target Environment</b>	<b>223</b>





# 1

## Overview

A variety of products are derived from the Multiple Target Development Tool (Mtdt) technology. This manual covers all the current Mtdt products. These products include single and multiple target simulators and hardware debuggers.

Mtdt has been developed using an object orientated and layered approach such that the bulk of the code within Mtdt can be applied to any target. It matters little if the target is big endian or little endian; 8, 16, 32, or 64 bits; hardware debugger or simulation engine. This allows us to quickly and inexpensively provide support for additional targets.

### Targets and Products

From a tools development standpoint there is very little to differentiate one target from another. CPUs execute code while peripherals have a host interface and wiggle and respond to I/O pins. Hardware interfaces provide different means to the same end. Current Mtdt targets include an eTPU2 simulation engine, an eTPU simulation engine, a TPU simulation engine, a CPU32 simulation engine, a CPU16 simulation engine, and a CPU32 across a BDM port. Support for additional targets continue to be added.

But from a customer's standpoint there are specific requirements that must be met. Individual products are therefore derived from Mtdt to meet these customer requirements.

As single target products we offer an eTPU2Stand-Alone Simulator, an eTPU Stand-Alone Simulator, a TPU Stand-Alone Simulator, a CPU32 Stand-Alone Simulator a CPU16 Stand-Alone Simulator, and a 683xx Hardware Debugger. As full system simulation

products we offer a CPU32/TPU System Simulator and a CPU16/TPU System Simulator.

Although not offered as a specific product, MtDt supports mixing hardware and simulated targets. This would allow, for instance, a software model of a new CPU to be linked to a real peripheral across a BDM Port. Driver code could be developed for a CPU that exists only as a software model but controls real hardware.

## Concurrently Developing Code for Multiple Targets

Code for multiple targets can be developed, and debugged, concurrently. Interactions between and among multiple targets are modeled precisely and accurately. All the normal debugging techniques such as single stepping, setting breakpoints, stepping into functions, etc., are available for each target. The IDE supports instantaneous target switching such that it is possible, for example, to run to a CPU32 breakpoint, and then switch to a TPU and single step it. All the while, all targets are kept fully synchronized.

## The Freescale eTPU and TPU

Of special interest are the Enhanced Time Processor Unit (eTPU) and TPU from Freescale. For those wishing more information on the eTPU, refer to the Freescale user manuals. For those wishing to develop their own TPU microcode, it is highly recommended that you obtain the book *“TPU Microcoding for Beginners”* from AMT Publishing. Do not be misled by the title. This book is essential for beginners and experts alike. The eTPU and TPU training seminars are also highly recommended.

## Script Commands Files

MtDt script commands files have several purposes. Each target has a primary script file used to automate things like loading code, initialization, and functional verification. ISR script files can be associated with specific interrupts and execute only when that interrupt is activated. Startup script commands files are executed following an MtDt-controlled reset or when the application is initially launched. The primary purposes of the startup script commands file is initialization and getting the target into the correct state for loading code when MtDt is launched. MtDt build script files instantiate and connect targets. For most users no knowledge of MtDt build script files is required because the standard build script files provided by ASH WARE provide all the required features. But power users may tailor the MtDt build script files to take advantage of MtDt's advanced full-system simulation capabilities.

## Test Vector Files

Test vector files provide the user with one means of exercising TPU simulation targets with complex test vectors. While a script commands file functionally represents the CPU interface, a test vector file represents the external interface. And whereas a script commands file provides a broad range of functions, the test vector file provides the narrow but powerful capability of driving nodes to "high" or "low" states and assigning application-specific names to nodes and node groups.

A second method of exercising the TPU simulation target is to create a model of the external system using a dedicated modeling CPU target. Using this method the user need not necessarily use test vector files.

## Sharing Memory between Multiple Targets

The key ingredient to the full system simulation capability is shared memory. Targets expose their address spaces to other targets. Two levels of address space exposure are possible: extrinsic and intrinsic.

Extrinsic exposure is the most powerful but is not supported by all targets. Extrinsicly mapped address spaces allow memory accesses to that section of the memory map to occur in a target different from the one in which the access originated. For example, a simulated TPU might perform a write to data memory. The section of data memory could be extrinsicly mapped to a CPU32 across a BDM port. The access would be transferred across the BDM port and would actually occur in the CPU32's address space.

Intrinsic exposure is less powerful but is supported by all targets. A target that cannot redirect a memory access to another target is said to be intrinsic. For example a CPU32 hardware such as a 68332 exposed across a BDM port is intrinsic. All memory accesses initiated by the CPU32 must occur within the address space of that CPU32, and cannot, for example, be redirected to memory owned by a simulated TPU.

## Miscellaneous Capabilities

MtDt has a number of additional features not yet mentioned. These include project sessions, source code files, functional verification, external logic simulation, integrated timers, multiple workshops, a rich set of dialog boxes, a standard Windows-style menu system, and a Windows-style IDE with hot keys, a toolbar, and target status indicators.

### 1.1 On-Line Help Contents

The following help topics are available.

- Overview
- Software Deployment, Upgrades, and Technical Support
- Version 4.30, 4.00 and 3.70 through 3.00 Enhancements
- Supported Targets and Available Products
- Project Sessions
- Source Code Files
- Script Commands Files
- Script Commands Groups
- Test Vector Files
- Functional Verification
- External Logic Simulation
- Integrated Timers
- Workshops
- Operational Status Windows
- Dialog Boxes
- Menus
- Hot Keys, Toolbar, Status Window
- Building the Target Environment

# 2

## Demo Descriptions

This section covers demos that are available for both the eTPU Simulator and ETEC Compiler. All demos simulate and compile with demonstration versions (non-purchased) of the ASH WARE simulator and compiler.

### **Freescale Set 1 - Uart Demo**

- Use of the Freescale's Set12 UART function..
- Use of header file, 'etec\_to\_etpuc\_uart\_conv.h' to convert between the automatically generated 'etpuc\_set1\_defines.h' and the standard Freescale API interface file.

### **Freescale Set 2 - Engine Demo**

- Use of the Freescale's Set2, Cam, Crank, Fuel, Spark, Knock functions.
- Use of the identical auto-generated headers that Freescale uses for it's host-side API

### **Freescale Set 3 - ASDC Demo**

## 2. Demo Descriptions

---

- Use of the Freescale's Set3 ASDC, PWMF, and PWMMDC functions.
- Use of the identical auto-generated headers that Freescale uses for its host-side API

### **Freescale Set 4 - ASAC Demo**

- Use of the Freescale's Set12 ASAC, PWMF, and PWMMAC functions.
- Use of ASH WARE's auto-defines file, 'etpuc\_set4\_defines.h'.

### **Data Types Demo**

- A variety of data types and data scopes commonly used in the eTPU.
- Run-time initialization of data using the ETEC-generated initialization file, 'DataTypes\_idata.h'.

### **Auto-Defines Demo**

- Use of files 'etec\_sim\_autodef.h' and 'etec\_sim\_autodef\_private.h' to write & verify channel, global, and engine memory. Note that engine memory is available in the eTPU2 only.

### **Templates Demo**

- A variety of templates (empty code) which are excellent starting point when developing new eTPU functions.
- Legacy and ETEC mode functions.
- Standard and Alternate entry tables.

### **System Configuration Demo**

- An (optional) system configuration file sets the system parameters such as clock

frequency, processor family, which functions run on which channels, channel priority, etc.

- Optionally, the maximum allowed worst case latency (WCL) for each channel can be specified
- Build fails if WCL requirements are not met.
- Analyses file shows resulting system behaviours such as WCTL, and WCL for each channel.

### **Stepper Motor System Simulator Demo**

- System simulator demo (both CPU and eTPU are simulated)
- Freescale's host-side API on a simulated CPU.
- The ASH WARE `<>_defines` file used in the host-side API.
- Freescale's Set 1 Stepper Motor (SM) function.

### **UART ETEC Mode System Simulator Demo**

- System simulator demo (both CPU and eTPU are simulated.)
- Use of the superior ETEC mode style of programming.
- Conversion of Freescale's UART function to ETEC mode.
- Freescale's host-side API used on a simulated CPU.
- The Auto-generated header files similar to those used in the Freescale standard functions.
- The ASH WARE generated '`<>_idata.h`' file for initializing DATA memory.
- The ASH WARE generated '`<>_scm.h`' file for initializing CODE memory.



# 3

## Software Upgrades

### World Wide Web Software Deployment

All ASH WARE software is now deployed directly from the World Wide Web. This is done using the following procedure.

- Download and install a demo version of the desired software product. All software products are available at a demonstration level.
- Purchase the software product(s) either from ASH WARE or one of our distributors.
- E-mail to ASH WARE the license file, named "AshWareComputerKey.ack", found in the installation directory.
- Wait until you receive an e-mail notification that the information from your license file has been added to the installation utility.
- Download and re-install again. The software product(s) you purchased are now fully functional. All other software products are still available at a demonstration level.

### World Wide Web Software Upgrades

All versions since 2.1 can be upgraded directly through the World Wide Web. The

### 3. Software Upgrades

---

following procedure is required when performing this upgrade. Note that versions prior to 2.1 cannot be upgraded via the World Wide Web.

- Upon receiving notification from ASH WARE that a new version of the [eTPU Simulator](#) is available, download and re-install MtDt.

After the initial software upgrade, ASH WARE no longer requires a new software key.

## Network Floating Licenses

Version 4.30 and above support a floating license capability. A central License Server has a pool of one or more licenses. Client computers request floating licenses from the License Server. The License Server issues licenses until its pool of licenses has been depleted. When a client computer no longer needs a license it becomes available for the License Server to distribute to a different client.

When a client requests a license it normally exits if no license is available. However, it may choose instead to wait a certain user-defined amount of time for a license to become available. If a license does become available the software is able to operate. If, after the user-specified amount of time is exceeded, no license becomes available then the software exits. This is especially useful in automated testing where the test would otherwise fail if no floating license were available. The amount of time to wait for a floating license to become available is specified by the `-NetworkRetry` parameter passed on the command line. See the Command Line Parameters section for specifying this parameter.

## Dongle Licensing

A dongle is a physical device that attaches to a USB port on your computer. With a dongle license you can run the software in a fully-functional mode as long as the dongle is connected to your computer. If you unplug the dongle, then the software will run only in a demo-limited mode.

A dongle effectively replaces the license file in that you can move the software and dongle to a different computer and it will run in fully-functional mode without any interaction with ASH WARE. This is particularly valuable in (say) aerospace in which the software must be functional for many decades. Since the software can be moved between computers with no interaction with ASH WARE, this constitutes a stable long-term development and maintenance scenario.

## Hearing about Software Releases

In order to be notified about ASH WARE's software releases, be sure to provide your e-mail address to ASH WARE. This will ensure that you are automatically alerted to production and beta software releases. Otherwise you will have to periodically check the ASH WARE Web site to find out about new software releases. Note that your e-mail address and other contact information will never be released outside of ASH WARE. Further, ASH WARE will only add you to our e-mail list if you specifically request us to do so.

The eTPU Simulator automatically displays an informational message when your software subscription is close to expiration. Note that the software license has no expiration so it is legal to use the eTPU Simulator beyond the software subscription expiration date. The software subscription entitles you to free technical support and Web-based software upgrades.

## Technical Support Contact Information

With the purchase of this product comes a one-year software subscription and free technical support. This technical support is available through Email, the World Wide Web, and telephone. Contact information is listed below.



- (503) 533-0271 (phone)
- [www.ashware.com](http://www.ashware.com)
- [support@ashware.com](mailto:support@ashware.com)

### 3.1 Version 3.0 to 4.3 Enhancements

#### **Version 4.30 Enhancements**


-  **4.30** Support for Network Floating Licensing.
-  **4.30** Support for Dongle Licensing.


#### **Version 4.00 Enhancements**


-  **4.00** eTPU2 script commands to write/verify Global Data.
-  **4.00** eTPU2 Stand-Alone Simulator Product (status=PRODUCTION)


### 3. Software Upgrades

---

 Support for the new STMicroelectronics/Freescale SPC563Mxx eTPU2 microcontrollers

 eTPU2 script commands to write Engine Relative RAM.

 eTPU2 script command to write the TCR1 Clock Source field (TBCR.TCR1CS).

 Script commands to write the STAC bus which support sharing of the TCR1 and TCR2 global counters across eTPU engines.

#### Version 3.80 Enhancements

eTPU2 script command to write the Engine Relative Base Address register.

#### Version 3.70 Enhancements

CRITICAL CHANGE: eTPU Engine ‘A’ and ‘B’ ISR Auto-define eTPU2 Stand-Alone Simulator Product2\_SIM (status=ALPHA)  
Support for the new STMicroelectronics/Freescale SPC563Mxx eTPU2 microcontrollers

#### Version 3.60 Enhancements.

In Dual-eTPU simulation configurations, one eTPU engine can drive the I/O pins of the other eTPU Engine. See the eTPU/TPU External Logic Commands section for more information.

Added support for MPC5553, MPC5534, MPC5565, MPC5567, and MCF5270/1 derivatives.

Instrument your code with named timers `TIMER_ACTION_HELP` to verify that critical timing indices are met. Use timing verification script commands to verify that traversal times meet application requirements. View the latest traversal times in the Watch window.

#### Version 3.50 Enhancements.

.eTPU-C provides a built-in type called `chan_struct` that allows access to such channel-specific settings as IPAC, OPAC, PDCM, TBS, etc. This symbolic access to channel settings has now been exposed in the eTPU Simulator. For example, to issue an TDL event on channel five, write `val("@5.ASHchannel.TDLA", "1");`

.eTPU-C register types are now supported in symbolic processing - watch windows, `print_to_trace()`, `verify_val()`, `write_val()`, etc. For example, `ertA`, `tcr1`, `p15_0`, etc.

.A `verify_str_ex()` script command has been added. It is very much like the `verify_str()` script command with the addition of a fourth argument – a length. Think `strncmp`.

.The `verify_str()` script command now supports the `>=` and `<=` operators. Added system macros for determining the Byte Craft compiler version, `__COMPILER_VERSION__`, and the ASH WARE simulator version, `__MTDT_VERSION__`, which are useful in script command used to verify or report this information.

```
print_to_trace("Using compiler version %s\n", __COMPILER_VERSION__);  
print_to_trace("Using simulator version %s\n", __MTDT_VERSION__);
```

Added an `at_code_tag_ex()` script command to add a timeout thereby handling more cleanly the situation in which the tag is never hit.

Added the eTPU/CPU System Simulator product

Added a “@ASH@print\_to\_trace” capability whereby formatted symbolic information (think `printf`) can be exported to the trace window every time an area of your code is traversed. GREAT FOR AUTOMATED TESTING!

Extended the existing `print_to_trace()` script command to support formatted symbolic information using a new “string within a string” format.

Added global eTPU ChannelVariable access support to the Watches window and various script commands.

Added a new “String within a String” format to extend the `print()`; and `print_to_trace()`; script commands such that they now support formatted symbolic information.

Added the GNU CPP pre-processor, “`cpp.exe`”, to script command files, thereby supporting capabilities such as global variable and static local initialization, among other things!

Added the ability to pass multiple `#defines` from the command line, and also to define strings and values (previously on a single value-less `#define` was supported.)

Added a command line switch that suppresses the “Source Code Missing” message when running automated tests

Improved tracing data by adding such things as the ME (Match Enable) bit value to the trace file, thereby supporting our companion latency analyses tool (see our website for a description.)

Added a `verify_version()`; script command to warn the user if there are dependencies on a particular simulator version.

#### Version 3.42 Enhancements

Improved the “Drag-And-Drop” of time in the logic analyzer window

The `dump_file()`; script can now append to any existing file.

The `verify_files()`; script is useful for logged regression testing using a “gold file”.

When building the simulation environment using an eTPU simulation model, the

### 3. Software Upgrades

---

eTPU sub-type can be specified. This supports version-specific features and bug simulation.

Added the ability to specify the target Version (such as MPC5554 eTPU).

Specify the full 32-bit read result on unused memory.

Added a new auto-define that helps conditional parsing of script command files.

Run the simulator until a channel's thread boundary

#### Version 3.41 Enhancements

Display current angle on status bar

Execute to a user-specified angle

Extended eTPU code coverage to include event vectors

Inferred event vector coverage

Accumulate coverage over multiple tests

View channel nodes

Write and verify `chan_data16()`; commands

View critical thread execution indices

Clear Worst Case Thread Indices Commands

Resizing the Pin Transition Buffer Testing

Cumulative Logged Regression Testing

#### Version 3.40 and 3.30 Enhancements

Added a new Complex Breakpoint Window

eTPU Stand-Alone Simulator Product

Multiple target scripting

Added ability to view thread group activity in the logic analyzer window

Improved Regression Testingsection

Added the `#include` directive to script commands files

Added `#ifdef`, `#ifndef`, `#else`, `#endif` preprocessing to script command files

Added automatic defines to script command files

#### Version 3.20 Enhancements

Testing automation via command line capabilities

Verification and modification of symbolic data

Superior Logic Analyzer Windowincluding precise event timing measurement

Improved simulation speed by between 2x and 2.6x depending on loading conditions

Superior tracing including pin transitions, parameter RAM I/O, and capture/match events saved to the trace buffer

Trace data can be saved to a file.

The application-wide font can be specified.

#### Version 3.10 Enhancements

- Replaced the `set_cpu_frequency()`; command with the `set_clk_period()`; script command
- Additional script commands
- New call stack window
- Improved local variables window
- Improved trace window
- New local variable options dialog box
- Additional IDE options
- Additional operation status windows

#### **Version 3.0 Enhancements**

- Source code search rules
- Enhanced script files
- Definitions using `#define` declaration
- Enumerated types
- Simple expressions
- Improved assignment operators
- Breakpoints
- Complete grammar check at load-time
- Editable register style windows
- Integrated timers `TIMERS_HELP`
- Watch, local variable, and memory dump windows
- Workshops



# 4

## The Project File

The Project Open and project Save As dialog boxes allow association of the eTPU Simulator configuration with a project file. All settings are stored in this file, including the active source code, script file, selected font spaces per tab, enabled/disabled messages, etc.

To open an existing project file, either double click on the file from Microsoft Windows Explorer or, from the Files menu, select the project Open submenu. Then select the name and existing project file. Various settings from the just-opened project file are loaded into the eTPU Simulator. Note that before opening the new project file, you are given the option to save the currently-active settings to the currently-active project file.

To create a new project file, from the Files menu, select the Save As submenu. Then specify a new file name.

At startup, global settings are retrieved from the last-active project file, assuming that no project file was passed on the command line. Otherwise, the project file passed on the command line is opened.

The paths to all other files are stored and retrieved relative to the project file. This allows the entire set of project files to be bundled and moved together as a group.

Whenever the eTPU Simulator is exited, the configuration is automatically written to the active project file.



# 5

## Source Code Files

User executable files are generated from source code using compilers, assemblers, and linkers. For TPU simulation engines Freescale's TPU Microcode Assembler can be used. For other targets, industry standard compilers such as Introl, Diab-Data, or GNU can be used.

These executable files are loaded into the target's memory space. The eTPU Simulator also loads the associated source code files and displays them in source code windows, highlighting the line associated with the instruction being executed. Several hot keys allow the user to set breakpoints, execute to a specific line of code, or execute until a point in time.

The executable code is loaded by selecting the Executable, Open submenu from the Files menu and following the instructions of the Load Executable dialog box. The loaded source code file is then displayed in a context window. The window can be scrolled, re-sized, minimized, etc. Help is available for the source code file window and is accessed by depressing the <F1> function key when the window is active.

### **Theory of Operation**

The eTPU Simulator associates the user's source code with the executable code generated by the compiler, assembler, and linker. This ability provides the following important MtDt functionality.

## 5. Source Code Files

---

- Highlight the active instruction
- Set/toggle breakpoints
- Execute to cursor

In order for the eTPU Simulator to read the executable code, the source code must be compiled, assembled, and linked. The resulting executable file can then be loaded into the eTPU Simulator. The name of the executable file varies quite a bit from one vendor and tool to another. Generally, TPU microcode executable files have a .LST suffix while a compiler/linker might produce a file named A.OUT.

### 5.1 Source Code Search Rules

Source code files may be contained in multiple directories. In order to provide source-level debugging, the eTPU Simulator must be able to locate these files. Source code search rules provide the mechanism for these files to be located.

The search rules are as shown below. These rules are performed in the order listed. If multiple files with the same name are located in different directories the first encountered instance of that file, per the search rules, will be used.

- Search relative to the directory where the main build file, such as A.OUT is located.
- Search relative to the directories established for the specific target associated with the source code.
- Search relative to the global directories established for all targets.

In the search rules listed above the phrase "search relative to the directory . . ." is used. What does this mean? It means that if the file is specified as "..\Dir1\Dir2\FileName.xyz", start at the base directory and go up one, then look down in directory "Dir2" and search in this directory for the file, "FileName.xyz".

Note that the search rules apply only to source code files in which an exact path is not available. If an exact path is available, the source code file will be searched only at that exact path. If an exact path is provided and the file is not located at that exact path, the search will fail.

The Source Code Search Options dialog box allows the user to specify the global directories search list as well as the search lists associated with each individual target.

### **Absolute and Relative Paths**

The eTPU Simulator accepts both absolute and relative paths.

An absolute path is one in which the file can be precisely located based solely that path. The following is an absolute path.

```
C:\Compiler\Library\
```

A relative path is one in which the resolution of the full path requires a starting point. The following is an example of a relative path.

```
..\ControlLaws\
```

Relative paths are internally converted to absolute paths using the main build file as the starting point. As an example, suppose the main build file named A.OUT is located at the following location.

```
C:\MainBuild\TopLevel\A.out
```

Now assume that in the search rules the following relative path has been established.

```
..\ControlLaws\
```

Now assume that file Spark.C is referenced from the build file A.OUT. Where would the eTPU Simulator search for this file? The following location would be searched first because this is where the main build file, A.OUT, is located.

```
C:\MainBuild\TopLevel\
```

If file Spark.C were not located at the above location, then the following location would be searched. This location is established by using the location of the main build file as the starting location for the ..\ControlLaws\ relative path.

```
C:\MainBuild\ControlLaws\
```



# 6

## Script Commands Files

### Overview

Script commands files provide a number of important capabilities to the user. In some cases, script commands files provide a mechanism whereby actions available within the GUI can be automated. In other cases they are used by the eTPU Simulator to build a full system out of various targets. They also can be used in place of missing-but-required pieces of a complete system such as in a TPU Stand-Alone Simulator project in which script commands files take the place of the host CPU.

We have created a single universal scripting language rather than a distinct scripting language for each application and target. The basic program construction is used for each. Only the predefined symbol table that is available differentiates the language for each application.

Each file is arranged as a sequential array of commands, i.e., the eTPU Simulator executes the script commands in sequential order. This allows the eTPU Simulator to know when to execute the commands. Timing commands cause the eTPU Simulator to cease executing commands until a particular point in time. At that point in time, the eTPU Simulator begins executing subsequent script commands until it reaches the next timing script command. Timing commands are not allowed in startup or MtDt build script files.

The following script help topics are found later in this section.

## 6. Script Commands Files

---

- Script Commands File Format
- Script Command Groups
- Multiple Target Scripts
- Automatic and Predefined #define Directives
- Predefined Enumerated Data Types

### Types of Script Commands Files

There are four types of script commands files. These are listed below, and a description of each can be found later in this section.

- The Primary Script Commands Files
- ISR Script Commands Files
- The Startup Script Commands File
- The MtDt Build Commands File

The eTPU Simulator can have only a single active MtDt build batch file. Each target may have only a single primary script commands file and a single startup scripts commands file active at any one time. ISR script commands files are associated with interrupts. Although each interrupt may have only a single associated ISR script commands file, it is important to note that each script commands file may be associated with multiple interrupts.

### Similarities to the C Programming Language

The script commands files are intended to be a subset of the "C" programming language. In fact, with very little modification these files will compile under C. The Script2C.exe utility is included for making the required conversions to compile script files in C.

## 6.1 The "ETEC\_cpp.exe" Preprocessor

The ETEC C PreProcessor (ETEC\_cpp.exe) provides enhanced preprocessing capabilities that significantly increase the power of the scripting language. In most cases this capability is transparent to the user, though one side affect is that the preprocessing stage is case sensitive.

This feature defaults to being active, though it can be disabled, see the IDE Options dialog box.

One application of the preprocessor is to support initialization of global variables in the eTPU. This is done as follows. The Byte Craft compiler supports a macro capability that results in a series of macros as shown below for global variable initialization.

```
__etpu_globalinit32(0x0000,0x70123456)
__etpu_globalinit32(0x0004,0x71ABCDEF)
__etpu_globalinit32(0x0008,0x72000000)
```

The example above was output into the auto-generated header files by the following Byte Craft command:

```
#pragma write h, (::ETPUglobalinit32);
```

Using the following macro expansion, it is possible change the above macros into a form supported by ASH WARE.

```
#define __etpu_globalinit32(location, value) \
*((ETPU_DATA_SPACE U32 *) location) = value;
```

The eTPU Simulator then sees the following script commands which it can handle.

```
*((ETPU_DATA_SPACE U32 *) 0x0000 ) = 0x70123456;
*((ETPU_DATA_SPACE U32 *) 0x0004 ) = 0x71ABCDEF;
*((ETPU_DATA_SPACE U32 *) 0x0008 ) = 0x72000000;
```

## 6.2 The Primary Script Command Files

The eTPU Simulator automatically executes a primary script commands file if one is open. A new or alternate script commands file must be opened before it is available to the eTPU Simulator for execution. The desired script commands file is opened via the Files menu by selecting the Scripts, Open submenu and providing the appropriate responses in the Open Primary Script File dialog box. The eTPU Simulator displays the open or active script commands file in the target's configuration window. Only one primary script commands file may be active at one time. Help is available for this window when it is active and can be accessed by depressing the <F1> function key.

### 6.3 ISR Script Commands Files

Currently, the ability to associate a script commands file with an interrupt is limited to the eTPU and TPU simulation targets.

Script commands files can be associated with interrupts. When the interrupt associated with a particular eTPU/TPU channel becomes asserted the ISR script commands file associated with that channel gets executed.

In the eTPU, ISR script commands files can be associated with channel and data interrupts as well as with the global exceptions.

There are some differences between the primary script commands file and ISR script commands files. Some important considerations are listed below.

- ISR script commands files are associated with channels using the `load_isr`, and similar script commands.
- The primary script commands file begins execution after a eTPU Simulator reset whereas ISR script commands files execute when the associated interrupt becomes both asserted and enabled.
- The primary script commands file is preempted by the ISR script commands files.
- ISR script commands files are not preempted, even by other ISR script commands files and even if the (discouraged) use of timing commands with these ISR script commands files is adapted.
- Only a single primary script commands file can be active at any given time. Each interrupt source can have only a single ISR script commands file associated with it.
- 
- Within the eTPU's interrupt service routine the ISR script commands file should clear the interrupt. This is accomplished using the `clear_chan_intr(X)`, the `clear_this_intr()`, or similar script command. Failure to clear the interrupt request causes an infinite loop.
- A single ISR script commands file can be associated with multiple interrupt sources such as eTPU/TPU channels. To make the ISR script commands file portable across multiple channels be sure to use the `clear_this_intr()` or similar script command.
- Do not use the `clear_this_intr()` script command in the primary script commands file because the primary script commands file does not have an

eTPU/TPU channel context.

- Use of timing commands within an ISR script commands file is discouraged. This would be analogous to putting delays in a CPU's ISR routine. Such a delay would have a detrimental effect on CPU latency and in the case of the eTPU/TPU Simulator would be considered somewhat poor form.
- eTPU/TPU channels need not have an association with an ISR script commands file.

There is an automatic define that can be used to determine which channel the script command is associated with. This script command appears as follows.

```
#define _ASH_WARE_<TargetName>_ISR_ X
```

Where TargetName is the name of the target (generally TpuSim, eTPU\_A, or eTPU\_B), and X is the number of the channel associated with the executing script. The following shows a couple examples of its use.

```
#ifdef _ASH_WARE_TPUSIM_ISR_
print("this is an ISR script running on a target TPUSIM");
#else
print("this is not an ISR running on TPUSIM");
#endif

write_par_ram(_ASH_WARE_TPUSIM_ISR_,2,0x41);
write_par_ram(_ASH_WARE_TPUSIM_ISR_,3,8);
clear_this_cisr();
```

### **Critical Change in Version 3.70 and Later!**

Beginning with MtDt Version 3.70, support for the Freescale and ST Microelectronics eTPU2 forced a change in the eTPU engine naming convention that affects the ISR auto #define. Freescale originally referred to the two eTPU engines as eTPU1 and eTPU2. Unfortunately, this naming convention clashes with the name of the new eTPU derivative, 'eTPU2.' The original eTPU is referred to as 'eTPU' and the new eTPU2 is referred to as 'eTPU2.'

Automatically-defined #defines within ISR script commands running on eTPU 'Engine A' and 'Engine B' are now as follows.

**Is:**

```
#define _ASH_WARE_ETPU_A_ISR_ <ChanNum>
#define _ASH_WARE_ETPU_B_ISR_ <ChanNum>
```

## 6. Script Commands Files

---

Was:

```
#define _ASH_WARE_ETPU1_ISR_ <ChanNum>
#define _ASH_WARE_ETPU2_ISR_ <ChanNum>
```

### 6.4 The Startup Script Commands File

Startup scripts provide the capability to get the target into a known state following the eTPU Simulator-controlled reset. It is particularly useful for the case in which the eTPU Simulator is configured to load an executable image when launched. In the 683xx Hardware Debugger, for instance, startup scripts can be used to configure the SIM registers so that the executable image can be immediately loaded when it is launched.

Each target can have an associated startup script. Make sure the desired target is active. From the Files menu select the Scripts, Startup submenu.

A report file is generated each time the startup script is executed. This file has the same name as the startup script; its extension is "report."

There are some restrictions to startup scripts. These are listed below.

- No windows are supported.
- Flow control, such as breakpoints and single step, is not supported.
- Certain script commands are not supported. Restricted commands are noted as such in the reference section.

### 6.5 The Build Script File

MtDt supports both debugging and simulation of a variety of targets. Since the single MtDt application can both simulate and debug a variety of simulation and hardware targets, how does MtDt know what to do?

Build batch files provide the instructions that MtDt requires to create and glue together the various copies of hardware and simulation targets. Although the user is encouraged to modify copies of these files when required, in many cases the standard build batch files loaded during MtDt installation will suffice.

Typical build script files might instantiate a CPU32 and a TPU3, then instantiate RAM and ROM for the CPU32, and then cause the TPU's memory to reside in the CPU's address space.

### MtDt Build Script Commands File Naming Conventions

In order to prevent future installations of ASH WARE software from overwriting build script files that you have created or modified, ASH WARE recommends that you follow the following naming convention.

- Build script files from ASH WARE begin with "zzz."
- Build script files not from ASH WARE don't begin with "zzz."

The string "zzz" was chosen so that these files would appear at the end of a directory listing sorted by file name. ASH WARE recommends that when you modify a build batch file you remove the letters "zzz" from the file name. When you create a new file, ASH WARE recommends that the file name not begin with the string "zzz." The following is a list of some of the build batch files loaded by the ASH WARE installation utility.

- zzz\_1TpuSim.MtDtBuild
- zzz\_2Sim32\_1TpuSim.MtDtBuild
- zzz\_1Sim32.MtDtBuild
- zzz\_1Bdm32.MtDtBuild

How would you modify the zzz\_1Sim32.MtDtBuild file to create a larger RAM? ASH WARE recommends the following procedure.

- Copy file zzz\_1Sim32.MtDtBuild to file BigRamSim32.BuildScript
- Modify file BigRamSim32.BuildScript

## 6.6 File Format and Features

The script commands file must be ASCII text. It may be generated using any editor or word processor (such as WordPerfect or Microsoft Word) that supports an ASCII file storage and retrieval capability.

The following is a list of script command features.

- Multiple-target scripts
- Directives
- Enumerated data types
- Integer data types
- Referenced memory

## 6. Script Commands Files

---

- Assignment operators
- Operators and expressions
- Comments
- Numeric Notation
- String notation

### Script Commands Format

The command is case sensitive (though this is currently not enforced) and, in general, has the following format:

```
command([data1],[data2],[data3]);
```

The contents within the parenthesis, data1, data2, and data3, are command parameters. The actual number of such data parameters varies with each particular command. Data parameters may be integers, floating point numbers, or strings. Integers are specified using either hexadecimal or decimal notation. Floating point parameters are specified using floating point notation, and strings are specified using string notation. Hexadecimal and decimal are fully interchangeable.

#### 6.6.1 Script Directives, Define, Ifdef, Include

In a multiple target environment each target has its own script commands file, and each of these files runs independently. As such, the scripts in each of these files act by default on its own target. For example a script that modifies memory will do so in the memory space of the target in which that script commands file is executing. But there are situations in which it is convenient to be able to have a script command within a single script commands file act on the various other targets in the system besides the one in which that script commands file is executing.

```
<TargetName>.<ScriptCommand>
```

The specific target for which a script command will run is specified as shown above.

```
eTPU_B.write_chan_hsrr (LAST_CHAN, 1);  
wait_time(10);  
verify_mem_u32(ETPU_DATA_SPACE, SLAVE_SIGNATURE_ADDR,  
              0xffffffff, DATA_TRANSFER_INTR_SIGNATURE);  
eTPU_B.verify_data_intr(LAST_CHAN, 1);  
eTPU_A.verify_data_intr(LAST_CHAN, 0);
```

In this example, a host service request is applied to target eTPU\_B. Ten microseconds later script commands verify that the host service request generated a data interrupt on

eTPU\_A but not eTPU\_B.

## 6.6.2 Script Directives, Define, Ifdef, Include

The #define directive

Script commands files may contain the C-style define directive. The define directive starts with the pound character "#" followed by the word "define" followed by an identifier and optional replacement text. When the identifier is encountered subsequently in the script file, the identifier text is replaced by the replacement text. The following example shows the define directive in use.

```
#define THIS_CHANNEL 8
#define THIS_FUNC 4
set_chan_func(THIS_CHANNEL, THIS_FUNC);
```

Since the define directive uses a straight text replacement, more complicated replacements are also possible as follows.

```
#define THIS_SETUP 8,4
set_chan_func(THIS_SETUP);
```

There are a number of automatic and predefined define directive as described in the like-named section.

### The #include <FileName.h> directive

Allows inclusion of multiple files within a single script file. Note that included files do not support things like script breakpoints, script stepping, etc.

```
#include "AngleMode.h"
```

In this example, file AngleMode.h is included into the script commands file that included it.

### The #ifdef, #ifndef, #else, #endif directives

These directives support conditional parsing of the text between the directives.

```
//===== That is all she wrote!!
#ifdef _ASH_WARE_AUTO_RUN_
exit();
# print else
("All tests are done!!");
#endif // _ASH_WARE_AUTO_RUN_
```

The above directive is commonly found at the very end of a script commands file that is

## 6. Script Commands Files

---

part of an automated test suite. It allows behavior dependent on the test conditions. Note that `_ASH_WARE_AUTO_RUN_` is automatically defined when the eTPU Simulator is launched in such a way that it runs without user input. In this case, upon reaching the end of the script file the eTPU Simulator is closed when it is part of an automated test suite and otherwise a message is issued to the user.

### 6.6.3 Script Enumerated Data Types

Enumerated types are only indirectly available to the user. Many defined functions have arguments that require specific enumerated data as arguments. Since the user cannot currently prototype new functions or recast enumerated data, this discussion is only indirectly germane. Despite these limitations, internally enumerated data types are defined for many script commands and the tighter checking and version independence provided by enumerated data types make them an important aspect of script files.

In general, the enumerated data types are defined for each specific target or script file application.

```
enum TARGET_TYPE
    TPU_SIM, ETPU_SIM,    // eTPU/TPU Targets
    SIM32, BDM32,        // CPU32 targets
    // *** END OF PARTIAL LISTING ***
};
```

Note that in C++ it would be possible to pass an integer as the first argument and at worst a warning would be generated. In fact, in C++, even the warning could be avoided by casting the integer as the proper enumerated data type. This is not possible in a script file because of tighter checking and because casting is not supported.

### 6.6.4 Script Integer Data Types

In order to maximize load-time checking, script command files support a large number of integer data types. This allows "constant overflow" warnings to be identified at load-time rather than at run-time. In addition, since the scripting language supports a variety of CPUs with different fundamental data sizes, the script command data types are designed to be target independent. This allows use of the same script files on any target without the possibility of data type errors related to different data sizes.

The following is a list of the supported data types along with the minimum and maximum value.

- U1 valid range is 0 to 1
- U2 valid range is 0 to 3
- U3 valid range is 0 to 7
- U4 valid range is 0 to 15
- U5A valid range is 0 to 16
- U5B valid range is 0 to 31
- U8 valid range is 0 to 0xFF
- U16 valid range is 0 to 0xFFFF
- U32 valid range is 0 to 0xFFFFFFFF
- U64 valid range is 0 to 0xFFFFFFFFFFFFFFFF

### 6.6.5 Referencing Memory in Script Files

Memory can be directly accessed by referencing an address. Two parameters must be available for this construct: an address and a memory access size. In addition, there is an implied address space, which for most targets is supervisor data. For some targets the address space may be explicitly overridden.

- (U8 \*) ADDRESS // References an 8-bit memory location
- (U16 \*) ADDRESS // References a 16-bit memory location
- (U32 \*) ADDRESS // References a 32-bit memory location
- (U64 \*) ADDRESS // References a 64-bit memory location
- (U128 \*) ADDRESS // References a 128-bit memory location

The following are examples of referenced memory constructs. Note that these examples do not form complete script commands and therefore in this form would cause load errors.

```
*((U8 *) 0x20 // Refers to an 8-bit byte at addr 0x20
*((U24 *) 0x17) // Refers to a 24-bit word at addr 0x17
*((U32 *) 0x40 // Refers to a 32-bit word at addr 0x40
```

### 6.6.6 Assignments in Script Commands Files

Assignments can be used to modify the value of referenced memory, a practice commonly referred to as "bit wiggling." Using this it is possible to set, clear, and toggle specific groups of bits at referenced memory. The following is a list of supported assignment

## 6. Script Commands Files

---

operators.

- =	Assignment
- +=, -=	Arithmetic addition and subtraction
- *=, /=, %=	Arithmetic multiply, divide, and remainder
- <<=, >>=	Bitwise shift right and shift left
- &=,  =, ^=	Bitwise "and," "or," and "exclusive or"
- <<=, >>=	Bitwise "shift left" and "shift right"

The following examples perform assignments on memory.

```
// Writes a 44 decimal to the 8 bit byte at address 17
*((U8 *) 0x17) = 44;
// Writes a 0xAABBCC to the 24 bit word at address 0x31
*((U24 *) 0x31) = 0xAABBCC;
// Sets bits 31 and 15 of the 32-bit word at addr 0x200
*((U32 *) 0x200) |= 0x10001000;
// Increments by one the 16-bit word at address 0x3300
*((U16 *) 0x3300) += 1;
```

Using an optional memory space qualifier, memory from a specific address space can be modified. See the Build Script ADDR\_SPACE Enumerated Data Type section for a listing of the various available address spaces.

```
// Sets the TPU I/O pins for channel 15 and channel 3
*((TPU_PINS_SPACE U16 *) 0x0) |= ((1<<15) + (1<<3));
// Sets the TPU's HSQR bits such that channel 7's is a 11b
*((TPU_DATA_SPACE U32 *) 0x14) |= (3<<(7*2));
// Injects a new opcode into the TPU's code space
*((TPU_CODE_SPACE U32 *) 0x20) = 0x12345678;
```

### 6.6.7 Operators and expressions in Script Commands Files

Operators can be used to create simple expressions in script commands files. Note that these simple expressions must be fully resolved at load time. The precedence and ordering is the same as in the C language. The following is a list of the supported operators.

- +, -	Arithmetic addition and subtraction
- *, /, %	Arithmetic multiply, divide and remainder
- &,  , ^, ~	Bitwise AND, OR, and EXCLUSIVE OR
- <<, >>	Bitwise shift left and shift right

The following example makes use of simple expressions to specify the channel base.

```
#define PARAMETER_RAM 0x100
#define BYTES_PER_CHAN 16
#define SPARK_CHAN_ADDR PARAMETER_RAM + BYTES_PER_CHAN * 5
// Write a 77 (hex) byte to address 150 (hex)
*((U8 *) SPARK_CHAN_ADDR) = 0x22+0x55;
```

The normal C precedence rules can be overridden using brackets as follows.

```
write_chan_func(1, 3+4*2); // Set chan 1 to function 11
write_chan_func(1, (3+4)*2); // Set chan 1 to function 14
```

### 6.6.8 Syntax for global access of eTPU Function Variables

Although eTPU channel variables reside in statically allocated memory, scope-wise within eTPU-C they are treated more like local variables. The syntax covered in this section allows developers to access channel variables at any time, for debug or verification purposes, not just when within function scope.

The following syntax supports global reference of channel variables by symbolic name. The syntax has the form @<channel # / name>.<function var name>. Either a raw channel number can be used, or the name assigned to the channel in the Vector file works to reference channel variables on a channel.

```
@PWM3.DutyCycle
@5.Period
verify_val("@PWM2.DutyCycle", "=", "3500");
// @ASH@print_to_trace("PPWA channel 3 high time = 0x%x\n", @3.
HighTime);
```

This syntax can be used with the symbolic script commands such as `verify_val()`, in the Watch window, as well as in the `@ASH@print_to_trace()`; action command.

### 6.6.9 Syntax for eTPU Channel Hardware Access

eTPU-C provides a built-in type called `chan_struct` that allows access to such channel-specific settings as IPAC, OPAC, PDCM, TBS, etc. This symbolic access to channel settings has now been exposed in the eTPU Simulator. Typically, a `chan_struct` variable is defined in a standard header file (`etpuc.h`), e.g.

```
chan_struct channel;
```

This variable `channel` can then be used to access channel settings in the watch window, or

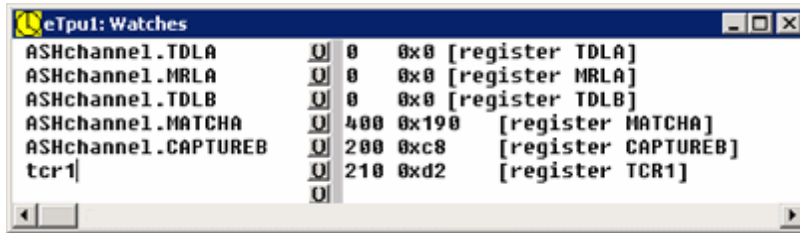
## 6. Script Commands Files

---

script command such as `print_to_trace()` and `verify_val()`, with the syntax:

```
channel.OPACA  
channel.PDCM
```

In addition to any `chan_struct` variables explicitly defined in the code, the Simulator always predefines a variable called `ASHchannel` of type `chan_struct`. Thus this special access is available even when the predefined headers are not used. This is shown in the watch window, below.



When a `chan_struct` variable is accessed as described above, e.g.,

```
print_to_trace("\\"Current channel PDCM is %d\\", ASHchannel.  
PDCM");
```

the value accessed is always from the current channel, as indicated by the `chan` register. There are times where it is useful to access channel fields for other than the active channel. The following script commands could be used to help test thread handling when both `TDLA` and a link service request (`LSR`) are set:

```
write_val("ASHchannel.TDLA", "1");  
write_val("ASHchannel.LSR", "1");
```

As written above, they only apply to the current channel, or whatever the `chan` register is currently set to if no thread is active. Similar to channel variables, the channel relative syntax can be applied to `chan_struct` type variables -

@<channel number / channel vector name>:. Setting `TDLA` and `LSR` on channel 5, which for example we assume is named "P\_IN" in the vector file, the script should be something like as follows:

```
write_val("@5.ASHchannel.TDLA", "1");  
write_val("@P_IN.ASHchannel.LSR", "1");
```

Note that when writing channel settings, great care must be taken. For example, setting `TDLB` without setting `TDLA` may result in undefined behavior. Not all channel fields described in the `chan_struct` type definition are supported, and several additional ones have been added - the supported list is as follows:

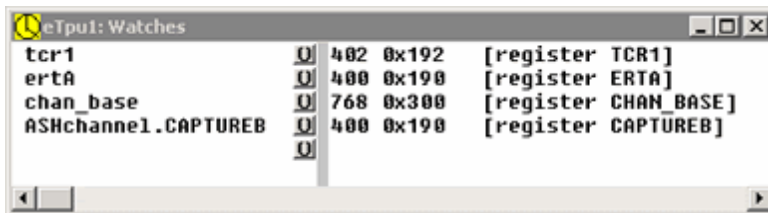
```
IPACA
```

IPACB  
 LSR  
 MRLA  
 MRLB  
 MTD  
 OPACA  
 OPACB  
 PDCM  
 TBSA  
 TBSB  
 FLAG0  
 FLAG1  
 FM0  
 FM1  
 PSS  
 PSTI  
 PSTO  
 TDLA  
 TDLB  
 MATCHA  
 MATCHB  
 CAPTUREA  
 CAPTUREB

Note that the fields FM0, FM1 and PSS refer to sampled values and thus only apply to the current channel and thread (time slot transition). PSS is re-sampled when the chan register is written.

### 6.6.10 Syntax for eTPU ALU Register Access

TPU-C register types are now supported in symbolic processing in script commands such as `print_to_trace()`, `verify_val()`, `write_val()`, etc., and also in the watch window as shown below.



In order for this to work, the registers must be exposed in a header file.

## 6. Script Commands Files

---

In standard the eTPU-C headers (etpuc.h) several register variables are exposed by default as follows:

```
register_chan chan;
register_erta erta;
register_ertb ertb;
register_tcr1 tcr1;
register_tcr2 tcr2;
register_tpr tpr;
register_trr trr;
register_chan_base chan_base;
```

Others can be defined per the eTPU-C syntax, for example:

```
register_diob diob;
register_mach mach ;
register_pl5_0 pl5_0; // lower 16 bits of p register
```

See the standard eTPU-C headers for details on the syntax of the register types. Once defined, such variables can be referenced in the watch window or symbolic script commands just like other variables.

```
print_to_trace("\\"Current channel is %d\\", "chan");
```

Note that write-only registers like link do not provide meaningful data.

### 6.6.11 String within a string supports formatted symbolic information

The “string within a string” formatted supports generation of formatted symbolic information by certain script commands. The format is a “C” string with a second embedded C string. The embedded sting must use the backslash escape character to begin and end the embedded string. Two example uses of this are shown below.

```
write_chan_hsrr ( TEST1_CHAN, 5);
wait_time(0.12);
// Send the results to the trace window
print_to_trace("\\"TEST RESULTS (trace):\n"
               "    A=%d\n"
               "    B=%d\n"
               "    C=%d\\",A,B,C");
// Same as above ... but to the screen
print          ("\\"TEST RESULTS (screen):\n"
               "    A=%d\n"
               "    B=%d\n"
               "    C=%d\\",A,B,C");
```

## 6.6.12 Comments in Script Commands Files

Legacy "C" and the new "C++" style comments are supported, as follows.

```
// This is a comment.
set_tdl(3);

/* This is a legacy C-style comment.
This is also a comment.
This is the end of the multiple-line comment. */
set_tdl(/* more comment */ 3);
```

## 6.6.13 Decimal, Hexadecimal, and Floating Point Notation in Script Files

Decimal and Hexadecimal notation are interchangeable.

```
357      // Decimal Notation
0x200    // Hexadecimal Notation
```

In certain cases floating point notation is also supported.

```
3.3e5    // Floating Point
```

## 6.6.14 String Notation

The following is the accepted string notation.

```
"STRING"
```

The characters between the first quote and the second quote are interpreted as a string.

```
"File.dat"
```

This denotes a string with eight characters and termination character as follows, 'F', 'i', 'l', 'e', '.', '\'', 'd', '\'', 'a', '\'', 't', '\'', '\0'.

### Concatenation

It is often desirable to concatenate strings. The following example illustrates a case in which this is particularly useful.

```
#define TEST_DIR  "..\\TestDataFiles\\"
read_behavior_file (TEST_DIR "Test.bv");
vector(TEST_DIR "Example");
```

### C-Style Escape Sequences

## 6. Script Commands Files

---

In the C language, special characters are specified within a string using the backslash character, '\'. For instance, a new-line character is specified with the backslash character followed by the letter "n", or '\n'. The character following the backslash character is treated as a special character. The following special characters are supported.

```
\\ References a backslash character
"..\\File.dat"
```

### Planned Obsolescence of Single Backslash within Strings

In previous versions of this software a C-style escape sequence was not supported and a single backslash character was treated as a just that, a single backslash character. In anticipation of future software versions supporting enhanced C-style escape sequences, the single backslash character within a string now causes a warning. ASH WARE recommends using a double-backslash to ensure compatibility with future versions of this software.

```
//The following string causes a warning.
"..\\File.dat"
```

## 6.7 Script Commands Groupings

Listed below are the available script command functional groups.

### All Target Types Script Commands

- Clock control script commands
- Timing script commands
- Verify traversal time commands
- Modify memory script commands
- Verify memory script commands
- Register write script commands
- Register verification script commands
- Symbol value write script commands
- Symbol value verification script commands
- System script commands
- File script commands
- Trace script commands
- Code coverage script commands

### eTPU/TPU Script Commands

- Channel function select register script commands
- Channel priority register script commands

- Host service request register script commands
- Interrupt association script commands
- External boolean logic script commands
- Pin control and verification script commands
- Pin transition behavior script commands
- Disable messages script commands
- Clear Worst Case Thread Indices Commands

### **eTPU Script Commands**

- System configuration commands
- Timing configuration commands
- STAC Bus commands
- Global Data Commands
- Channel data commands
- Channel base address commands
- Engine data commands
- Channel function mode (FM) commands
- Channel event vector entry commands
- Interrupt script commands

### **Build Script Commands**

- Build script commands

## **6.7.1 Clock Control Script Commands**

The script commands described in this section provide control over the clock and frequency settings.

The `set_cpu_frequency();` script command has been deprecated. Instead, use the `set_clk_period()` script command described in this section. A warning message is generated when this command is used. This message can be disabled from the Message Options dialog box.

```
set_clk_period(femtoSecondPerClkTick);
```

The script command listed above sets the target's clock period in femto-seconds per clock tick. Note that one femto second is  $1e-15$  of a second or one billionth of a micro-second. A simple conversion is to invert the desired MHz and multiply by a billion.

```
// 1e9/16.778 = 59601860 femto-seconds  
set_clk_period(59601860);
```

In this example the CPU clock frequency is set to 59,601,860 femto-seconds, which is

## 6. Script Commands Files

---

16.778 MHz.

### 6.7.2 Timing Script Commands

```
at_code_tag(TagString);  
at_code_tag_ex(TagString, Timeout, TimeoutAction);
```

These commands prevent subsequent commands from executing until the target hits the source code that contains the string, <TagString>. Note that all source code files are searched and that the string should be unique so that it is found at just one location (for otherwise the command will fail).

```
at_code_tag("$$MyTest1$$");  
verify_val("failFlag", "=", "0");
```

In the example above, the target executes until it gets to the point in the source code that contains the text, \$\$MyTest1\$\$ and then verifies that the variable named failFlag is equal to zero. It is important to note that the variable could be local to the function that contains the tag string such that it may be only briefly in scope. The only scoping requirement is that the variable is valid right when the target is paused to examine this variable.

The extended version of the command, at\_code\_tag\_ex(), also supports a timeout (in microseconds) such that if the tagged source code is not traversed in the amount time specified by Timeout, then the action specified by TimeoutAction will occur. Supported actions are FAIL\_ON\_TIMEOUT, FAIL\_ON\_TAG, and ALWAYS\_PASS. FAIL\_ON\_TIMEOUT causes a verification failure (and subsequent test suite failure) if the tagged code is not traversed in the specified time. FAIL\_ON\_TAG is just the opposite and is used when the tagged code is NOT expected to be traversed. ALWAYS\_PASS allows the script command execution to proceed on the first of either the tagged code being traversed or on the timeout, and no verification error is generated in either case.

```
at_code_tag_ex("$$MyTest2$$", 4.5, FAIL_ON_TIMEOUT);
```

In example above, the target executes until the point in the source code is traversed that contains the text, \$\$MyTest2\$\$\$. If in 4.5 microseconds the tagged code has NOT yet been traversed than a verification failure results. Script command execution continues on the first of the tagged text being traversed (the expected case) or the timeout (the failing case.)

```
at_time(T);
```

When this command is reached, no subsequent commands are executed until T microseconds from the simulation's start time. At that time the script commands following

the `at_time` statement are executed.

```
wait_time(T);
```

No script commands are executed until the simulation's current time plus the `T` microseconds.

```
wait_time(33.5); // (assume current time=50 microseconds)
set_link(5.0);
wait_time(100.0);
set_link(2.0);
```

In this example at 83.5 microseconds (from the start of the simulation) channel 5's Link Service Latch (LSL) will be set. No script commands are executed for an additional 100 microseconds. At 183.5 microseconds (from the start of the simulation) channel 2's LSL will be set.

### 6.7.3 Verify Timing Script Commands

The verify timing script command verifies that timing requirements are met. The format is as follows.

```
verify_timer_clks("TimingTag", MinSysClks, MaxSysClks);
```

The “ActionTag” parameter is a string that must match a similarly-named timing region within your code. The `MinSysClk` and `MaxSysClk` parameters describe the allowable minimum and maximum number of system clocks (inclusive) that execution of the code is allowed to take in order for the verification test to pass. If the code takes less time than the minimum or more time than the maximum to execute then a verification error occurs.

If the code has been traversed multiple times than the verification command verifies the last full traversal.

The following is an example of a region of code marked for timing. See the Timer Action Commands section for more information on naming timing regions.

```
int MyFunc(int x)
{
    int y;           // @ASH@timer_start("Test A");
    y = x + 25;
    return y        // @ASH@timer_stop("Test A");
}
```

In the following example, the last full traversal of the above code is verified to have taken between 10 and 20 system clocks. A verification error occurs if the code has never been fully traversed, if the last traversal took 9 or fewer system clocks, or if the last traversal

## 6. Script Commands Files

---

took 21 or more system clocks.

```
verify_timer_clks("Test A", 10, 20);
```

Note that on the eTPU there are two system clocks per instruction cycle, so the following #define can improve the test's readability.

```
#define CYCLES *2 // A cycle is two clocks on the eTPU
verify_timer_clks("Test A", 5 CYCLES, 10 CYCLES );
```

### Related Information

- Naming timing regions in source code
- Verifying traversal times a script command file
- View named timing regions timing using the Watch Window
- List named timing regions in the Insert Watch Dialog Box

### 6.7.4 Memory Modify Script Commands

Memory is modified within script commands using the assignment operator. See the Assignments in Script Commands Files section for a description.

### 6.7.5 Memory Verify Script Commands

Verify memory script commands provide the mechanism for verifying the values of the target memory. The first argument is an address space-enumerated type. The second argument is the address at which the value should be verified. The third argument is a mask that allows certain bits within the memory location to be ignored. The fourth argument is the value that the memory location must equal.

```
verify_mem_u8(enum ADDR_SPACE, U32 address, U8 mask, U8 val);
verify_mem_u16(enum ADDR_SPACE, U32 address, U16 mask, U16 val);
verify_mem_u24(enum ADDR_SPACE, U32 address, U24 mask, U24 val);
verify_mem_u32(enum ADDR_SPACE, U32 address, U32 mask, U32 val);
```

This command uses the following algorithm.

- Read the memory location in the specified address space and address.
- Perform a logical "and" of the mask with the value that was read from memory.
- Compare the result to the expected value.
- If the expected value is not equal to the masked value, generate a verification error.

The following example verifies register values of a CPU32 target.

```
verify_mem_u8(CPU32_SUPV_DATA_SPACE, 0x7, 0xc0, 0x80);
```

In the example above, a script file, which is acting on a CPU32 target, verifies that the two most significant bits found at address 0x7 are equal to 10b. The lower 14 bits are ignored. If the bits are not equal to 10b, a script failure message is generated and the target's script failure count is incremented.

```
verify_mem_u16(CPU32_SUPV_DATA_SPACE, 0x100, 0xffff, 0x55aa);
```

In the example above, a 16-bit (two-byte) memory at address 100h is verified to equal 0x55aa. By using a mask of FFFFh, the entire word is verified.

```
verify_mem_u24(ETPU_DATA_SPACE, 0x101, 0xffffffff, 0x555aaa);
```

In the example above, a 24-bit (three-byte) memory at address 101h is verified to equal 0x555aaa. By using a mask of 0xFFFFFFFF, the entire word is verified.

```
verify_mem_u32(CPU32_SUPV_DATA_SPACE, 0x200, 1<<27, 1<<27);
```

In the example above, bit 27 of a 32-bit (four-byte) memory location at address 200h is verified to be set. All other bits except bit 27 are ignored.

### Related Topics

See the eTPU Channel Data Script Commands section which covers both writing and verifying eTPU memory.

## 6.7.6 Register Write Script Commands

Write register script commands provide the mechanism for changing the values of the target registers. The first argument is a register-enumerated type with a definition that depends on the specific target and register width on which the script command is acting. The second argument is the value to which the register will be set.

```
write_reg4(U4, enum REGISTERS_U4);  
write_reg8(U8, enum REGISTERS_U8);  
write_reg16(U16, enum REGISTERS_U16);  
write_reg24(U24, enum REGISTERS_U24);  
write_reg32(U32, enum REGISTERS_U32);  
write_reg64(U64, enum REGISTERS_U64);
```

The following example modifies register values of a CPU16 target.

```
write_reg4(0x12, REG_ZK);  
write_reg16(0x5557, REG_PC);
```

In the example above, a script file, which is acting on a CPU16 target, writes 12

## 6. Script Commands Files

---

hexadecimal to the index register Z's address extension register and writes 5557 hexadecimal to the program counter.

### 6.7.7 Register Verify Script Commands

Verify register script commands provide the mechanism for verifying the values of the target registers. The first argument is a register-enumerated type with a definition that depends on the specific target and register width on which the script command is acting. The second argument is the value against which the register will be verified.

```
verify_reg1(enum REGISTERS_U1, U1);
verify_reg4(enum REGISTERS_U4, U4);
verify_reg5(enum REGISTERS_U5, U5);
verify_reg8(enum REGISTERS_U8, U8);
verify_reg16(enum REGISTERS_U16, U16);
verify_reg24(enum REGISTERS_U24, U24);
verify_reg32(enum REGISTERS_U32, U32);
verify_reg64(enum REGISTERS_U64, U64);
```

The following example verifies register values of a CPU16 target.

```
verify_reg4(REG_ZK, 0x12);
verify_reg16(REG_PC, 0x5557);
```

In the example above, a script file, which is acting on a CPU16 target, verifies that the index register Z's address extension register is equal to 12 hexadecimal and verifies that the program counter is equal to 5557 hexadecimal. If either of these conditions fails to verify, a script failure message is generated and the target's script failure count is incremented.

### 6.7.8 Symbol Write Script Commands

Write symbol value script commands provide a mechanism for writing data to simulated/target memory using the symbolic names from the source code. The `write_val()` command is for writing data of a basic type to a symbolically referenced memory location.

```
write_val("symbolExprString", "exprString");
```

The expression string (`exprString`) can be a numerical constant or a simple symbolic expression. Constants can be supplied as decimal signed integers, unsigned hexadecimal numbers (prefixed with '0x'), floating point numbers, or as a character (e.g. 'A'). A symbolic expression can be just a local/global symbol or a simple expression such as `*V` (de-reference), `&V` (address of V), `V[constant]`, `V.member` or `V->member`, where V is a

symbol of the appropriate type. The special @ channel variable (eTPU only) reference syntax is also supported. "symbolExprString" must be a symbolic expression as described above, an "l-value" in compiler parlance. The type of the symbolic expression must be a basic type – char, int, float, etc. If the types of the two sides differ then C type conversion rules are followed before writing the data to memory.

Two other forms of the write symbol value script command are also supported that directly take the numerical value to write as an argument.

```
write_val_int("symbolExprString", U32 val);
write_val_fp("symbolExprString", double val);
```

These forms allow the value to be input as a constant expression, perhaps using a series of macros, thereby providing more flexibility.

The write\_str() command is provided as a shorthand way to write a string into the memory pointed to by a symbolic expression of pointer type..

```
write_str("pointerExprString", "stringExprString");
```

The pointer expression string (pointerExprString) is symbolic expression as described above, but of type pointer rather than a basic type. It can be a pointer to any type, and is implicitly type-cast to the char\* type. "stringExprString" can either be a string constant, or it can be of type char array or char pointer. In either case, write\_str() function like the C library function strcpy(). Use write\_str with caution; no effort is made to check that the destination buffer has sufficient space available, and the resulting bug induced by such a buffer overflow can be extremely difficult to debug.

A key concept is that of symbol scope. A variable defined within a particular function goes out of scope if that function is not being executed. To get around this, a script command can be set to execute when the function becomes active using the at\_code\_tag (); script command. See the Timing Script Commands section for a description.

```
at_code_tag("&&Test1Here&&");
write_val("FailFlag", "0");
```

In the above example the target is run until it gets to the address associated with the source code that contains the text &&Test1Here&&. Once this address is reached, symbol FailFlag is set equal to zero.

```
at_code_tag("&&Test23Here&&");
write_str("PlayerBuffer", "Michael Jordan");
```

In this case the string "Michael Jordan" is written to the buffer named "PlayerBuffer". If the buffer has insufficient space to hold this string, a bug that is difficult to identify would result.

## 6. Script Commands Files

---

See the Global eTPU Channel variableAccess section for information on accessing eTPU channel variables using the format shown below.

```
@<chan num/name>.<function var name>
```

### 6.7.9 Verify Symbol Value Script Commands

These verify symbol value commands have a similar syntax to those commands described in the Write Symbol Value Script Command section.

```
verify_val("exprString", "testOpString", "exprString");
```

The expression strings (exprString) can be a numerical constant or a simple symbolic expression. Constants can be supplied as decimal signed integers, unsigned hexadecimal numbers (prefixed with '0x'), floating point numbers, or as a character (e.g. 'A'). A symbolic expression can be just a local/global symbol or a simple expression such as \*V (de-reference), &V (address of V), V[constant], V.member or V->member, where V is a symbol of the appropriate type. The special @ channel variable (eTPU only) reference syntax is also supported. If the types of the two sides differ, C type conversion rules are followed before performing the test operation.

"testOpString" is a C test operator. Supported operators are ==, !=, >, >=, <, <=, &&, and ||.

If the result of the specified operation on the expressions is 0, or false, a verification error is generated.

```
at_code_tag("&&Test1Here&&");  
verify_val("FailFlag", "==", "0");
```

In the example above, the target is run until it gets to the address associated with the source code that contains the text &&Test1Here&&. Once this address is reached, the value of symbol FailFlag is read and a verification error is generated if it does not equal zero.

Two other forms of the verify symbol value script command are also supported that directly take the numerical value to compare against as an argument.

```
verify_val_int("exprString", "testOpString", U32 val);  
verify_val_fp("exprString", "testOpString", double val);
```

These forms allow the test value to be input as a constant expression, perhaps using a series of macros, thereby providing more flexibility.

Separate script commands are available to compare and verify string values.

```
verify_str("expr1", "testOp", "expr2");
verify_str_ex("expr1", "testOp", "expr2", len);
```

The “expr1” and “expr2” parameters can either be a string constant, or can be of type char array or char pointer. If strings are resolved from both parameters then they are compared using the comparator specified in "testOp". If the outcome of this is true (non-zero), the verification test passes; otherwise a failure is reported. Supported comparator operators are ==, !=, >, and <, >= and <=. Greater-than and less-than operations follow the same rules as the strcmp() standard library function.

The extended version of this command, verify\_str\_ex(); also support a length specifier, len. Think strncmp where the comparison acts only on the first “len” characters and the remainder are ignored.

```
at_code_tag("^^StringTest1^^");
verify_str("somePtr", "==", "Hello World");
```

In the example above, the target is run until it gets to the address associated with the source code that contains the text ^^StringTest1^. Once this address is reached, the ASCII values are read until the terminating character, a byte of value zero, is reached. The resulting string is compared, case-sensitively, with the string "Hello World". If the two strings are not equal, a verification error is generated.

```
at_code_tag("^^StringTest2^^");
verify_str_ex("somePtr", "<=", "Hello World", 5);
```

In the example above, the first five letters of two strings are compared and verification error results unless somePtr is less than or equal to “Hello”.

See the Global eTPU Channel variableAccess section for information on accessing eTPU channel variables using the format shown below.

```
@<chan num/name>.<function var name>
```

### 6.7.10 System Script Commands

```
system(commandString);
verify_system(commandString, returnVal);
```

These commands invoke the operating system command processor to execute an operating system command, batch file, or other program named by the string, <commandString>. The first command, system, ignores the return value, while the second command, verify\_system, verifies that the value returned is equal to the expected value, returnVal. If the returned value is not equal to the expected value than a script verification error is generated.

## 6. Script Commands Files

---

```
system("copy c:\\temp\\report.txt check.txt");  
verify_system("fc check.txt expected.txt", 0);
```

In this example the operating system is invoked to generate a file named check.txt from a file named report.txt. The file check.txt is then compared to file expected.txt using the fc utility. A script verification error is generated if the files do not match.

```
exit();
```

This shuts down the eTPU Simulator and sets the error level to non-zero if any verification tests failed. If all tests pass, the error level is set to zero. The error level can be examined by a batch file that launched the eTPU Simulator, thereby supported automated testing. See the Regression Testing section for a detailed explanation of and examples showing how this command can be used as part of an automated test suite.

```
print(messageString);
```

This command is geared toward promoting camaraderie between coworkers. This command causes a dialog box to open that contains the string, <messageString>. The truly devious practical joker will find a way to combine this script command with sound effects.

```
print("Hit any key to fuse all P-wells "  
      "with all N-substrates in your target silicon");
```

In the example above, your coworker at the adjacent lab bench pauses for a certain amount of healthy introspection. A well-placed and timed bzilch-chord can significantly enhance its effect.

```
print("\n"TEST RESULTS:\n"  
      "      A=%d\n"  
      "      B=%d\n"  
      "      C=%d\n",A,B,C);
```

Formatted symbolic information can be generated using as described in the “String within a String” section. See the example shown above.

```
verify_version(verHi, verLo, verBuildChar, messageString);
```

This command generates a warning message if the eTPU Simulator version is earlier than that specified by <verHi>, <verLo> and <verBuildChar>. If an earlier than required version of the eTPU Simulator is actually running then a dialog appears that contains the text specified by <messageString>.

```
verify_version(3,50,'B',  
              "This demo application that illustrates GLOBAL INITIALIZATION\n"  
              "will not work in this early version of the eTPU Simulator." );
```

In the example above the message, “this demo ...” appears if run on a eTPU Simulator version earlier than version 3.50, build B. We recommend this script command be placed

first in the script file so that it gets processed before any parse errors or unsupported command errors can occur.

### 6.7.11 File Script Commands

These commands support loading and saving files via script commands.

```
load_executable("filename.executable");
```

This example loads the executable image and related source files found in file filename.executable. Any open source files in the previously-active image are closed. The file path is resolved relative to the directory in which the project file is located.

```
load_executable("A.Out");
```

This example loads the executable found in file A.Out in the same directory where the project file is located.

```
vector("filename.Vector");
```

The eTPU Simulator test vectors found in file filename.Vector are loaded by this script command. Test vector files normally have a "vector" suffix. Note that the "vector" suffix is preferable. Test vector files can also be loaded via the Open Test Vector File dialog box which is opened from the Files menu by selecting the Vector, Open submenu.

```
vector("UART.Vector");
```

This example loads the test vectors found in file UART.Vector in the directory where the project file is located. The file path is resolved relative to the directory in which the project file is located.

```
dump_file(startAddress, stopAddress, enum ADDR_SPACE,  
          "filename.dump", enum FILE_TYPE,  
          enum DUMP_FILE_OPTIONS);
```

This command creates the file filename.dump of type FILE\_TYPE, using the options specified by DUMP\_FILE\_OPTIONS. The file is created from the image located between startAddress and stopAddress, out of the address space ADDR\_SPACE.

```
dump_file(0, 0xffff, CPU32_SUPV_DATA_SPACE, "Dump.S19",  
          SRECORD, DUMP_FILE_DEFAULT);  
#define MY_OPTIONS NO_ADDR + NO_SYMBOLS  
dump_file(0, 0xffff, CPU32_SUPV_CODE_SPACE, "Dump.dat",  
          DIS_ASM, MY_OPTIONS);
```

The downside of the dump\_file command is that it does a one-time dump of the entire file overwriting any previous file. An alternative to this is to continuously write the trace data

## 6. Script Commands Files

---

to a file as it is generated. See the Trace Script Commands Section.

This example creates a Motorola SRECORD file, Dump.S19, from the first 64K of the CPU32's supervisor data space. The default options are used for this dump. An assembly file, Dump.dat, is also created from the first 64K of supervisor code space and both address mode and symbolic information are excluded. Assuming the target processor is a CPU32, the generated assembly code is for the CPU32.

```
verify_files("fileName1.dat", "fileName2.dat",
            enum VERIFY_FILES_RESULT);
```

This script command verifies two file named fileName1.dat and fileName2.dat. In addition to checking for matching or mismatching, this script command also can verify non-existence of either file or both files. Expected results are specified with the VERIFY\_FILES\_RESULT enumeration.

```
verify_files("new.dat", "gold.dat", FILE1_MISSING );
dump_file(0, 0x28, ETPU_DATA_SPACE, "new.dat", IMAGE, DATA8
);
verify_files("new.dat", "gold.dat", FILESMATCH );
```

In the above example a new file named “new.dat” is generated, and is compared against file “gold.dat” to make sure that they match. By first verifying that file “new.dat” does not exist the fact that file “new.dat” is actually generated by the dump command is also verified.

```
dump_file(0, 0x28, ETPU_DATA_SPACE, "new.dat", IMAGE,
DATA8);
verify_files("new.dat", "gold.dat", FILES_MISMATCH );
wait_time(100);
dump_file(0, 0x28, ETPU_DATA_SPACE, "new.dat", IMAGE,
DATA8 | FILE_APPEND);
verify_files("new.dat", "gold.dat", FILES_MATCH );
```

The above example illustrates use of the FILE\_APPEND options to store multiple data snapshots into a single file. By first verifying that the files mismatch, then that the files match, it proves that it is this verification process that actually generated the passing results.

### Related Topics:

Trace Script Commands

## 6.7.12 Trace Script Commands

```
start_trace_stream("FileName.Trace", enum TRACE_EVENT_OPTIONS,
                  enum TRACE_FILE_OPTIONS, enum BASE_TIME,
                  U32 numTrailingDigits);
```

This command saves the target's trace buffer to file `FileName.trace` with the options set by `TRACE_EVENT_OPTIONS`., `TRACE_FILE_OPTION` , and `BASE_TIME_OPTIONS`. The time field has `numTrailingDigits` trailing digits.

Note that all events specified in the script command must be currently enabled within the eTPU Simulator for the command to succeed.

Note also that the `BASE_TIME` and `numTrailingDigits` digits apply ONLY if the `TRACE_FILE_OPTION` is set to "PARSEABLE". If the specified file type is "VIEWABLE" than these options are taken from the trace window settings so that the trace file looks like the trace window.

```
end_trace_stream();
```

This command stops tracing to a stream and closes the file so that it can be opened by a text viewer that requires write permission on the file.

```
print_to_trace(Message)
```

This command prints the string `Message` to the trace assuming it is enabled in the Trace Options ... dialog box.

```
at_time(400);
start_trace_stream("Stream.Trace", ALL-DIVIDER - MEM_READ,
                  PARSEABLE, US, 3, 1);
print_to_trace("*****\n"
              "*****  START OF TEST          \n"
              "*****\n");
wait_time(1500);
end_trace_stream();
```

The above example begins streaming all trace data except dividers and memory reads to a trace file named "Stream.Trace". Time is recorded in micro-seconds with three trailing digits following the period such that the least significant digit represents nano-seconds. The `isResetTime` field is set to "true" so that script execution time of 400 micro-seconds is subtracted from the time and the clocks field.

```
save_trace_buffer("FileName.trace", enum TRACE_EVENT_OPTIONS,
                  enum TRACE_FILE_OPTION, enum BASE_TIME,
                  U32 numTrailingDigits);
```

Formatted symbolic information can be generated using as described in the "String within a

## 6. Script Commands Files

---

String” section. The following is an example of this format.

```
print_to_trace("\nTEST RESULTS:\n"
              "    A=%d\n"
              "    B=%d\n"
              "    C=%d\n",A,B,C);
```

This command is the same as the `start_trace_stream` command except that the current trace buffer is saved to a file.

```
save_trace_buffer("ThisTraceFile.Trace",
                 STEP + MEM_READ, VIEWABLE,
                 US, 3);
```

In this example a file named `ThisTraceFile.trace` is generated. Only step events and memory read events are saved. The selected file format is optimized for viewing rather than parsing. Because this is a "VIEWABLE" file, the `base_time` and `numTrailingDigits` fields are ignored.

### 6.7.13 Code Coverage Script Commands

An important index of test suite completeness is the coverage percentage that a test suite achieves. The eTPU Simulator provides several script commands that aid in the determination of coverage percentages. In addition, script commands provide the capability to verify that minimum coverage percentages have been achieved. A discussion of this topic is found in the Code Coverage Analysis section. The following are the script commands that provide these capabilities.

```
write_coverage_file("Report.Coverage");
verify_file_coverage("MyFile.uc",instPct,braPct);
verify_all_coverage(instPct,braPct);

// eTPU-Only
verify_file_coverage_ex("MyFile.c",instPct,braPct,entPct);
verify_all_coverage_Ex(instPct,braPct,entPct);
```

The `write_coverage_file(...)` command generates a report file that lists the coverage statistics. Statistics for individual files are listed as well as a cumulative file for the entire loaded code.

The `verify_file_coverage(...)`; and `verify_file_coverage_ex(...)`; commands are used as part of automation testing of a specific source file. The `instPct` and `braPct` parameters are the minimum required branch and coverage percentages in order for the test to pass. The `entPct` parameter is the minimum require entry percentage and is available only in the

eTPU simulator. These parameters are both expressed in floating point notation. The valid range of coverage percentage is zero to 100. Note that for each branch instruction there are two possible branch paths: the branch can either be taken or not taken. Therefore, in order to achieve full branch coverage, each branch instruction must be encountered at least twice and the branch must both be taken and not taken.

The `verify_all_coverage(...)`; and `verify_all_coverage_ex(...)`; are similar to the `verify_file_coverage` commands except these commands focus on the entire build rather than specific source code modules. As such, they are less useful as a successful testing strategy will focus on specific modules rather than on the entire build.

Note that this capability is also available directly from the file menu.

```
wait_time(100);  
verify_file_coverage("toggle.uc",92.5,66.5);  
verify_all_coverage(33.3,47.5);  
write_coverage_file("record.Coverage");
```

The code in this example waits 100 microseconds and then verifies that at least 92.5 percent of the instructions associated with file `toggle.uc` have been executed and 66.5 percent of the possible branch paths associated with file `toggle.uc` have been traversed. In addition, the example verifies that at least 33.3 percent of all instructions have been executed and that 47.5 percent of all branch paths have been traversed. A complete report of instruction and branch coverage is written to file `record`.

### Inferred Event Vector Coverage

In eTPU applications it is often difficult to get complete (100%) event vector coverage. There are two situations in difficulties may be encountered.

The first situation would be a valid and expected thread that is difficult to reproduce in a simulation environment. For example when measuring the time at which a rising edge occurs, it may be difficult to generate a test case for when the input pin is a zero, because a thread handler will normally execute immediately such that the pin is still high.

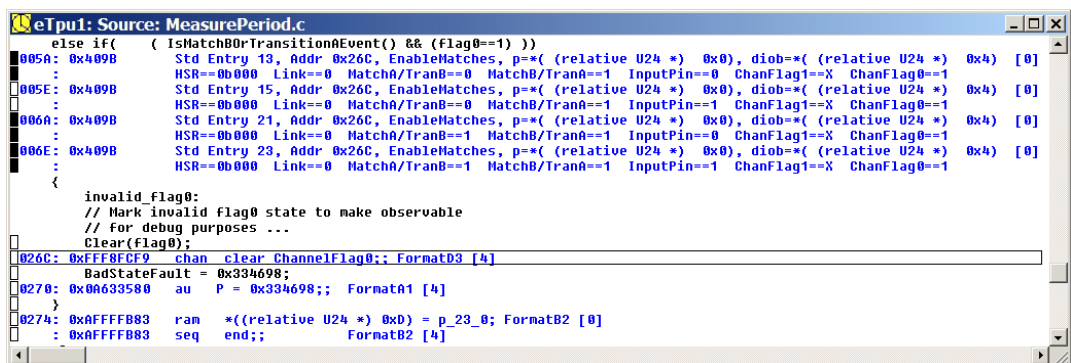
But an event vector handling the case of a rising edge and a low pin is valid. For instance, a rising edge followed by a falling edge could occur before a thread executing in another channel completes. Now the thread handling this rising edge executes with a low pin state. It is therefore important to test this case, but how? The solution to achieving event vector coverage for this case is to be clever in designing a test. For example, you might inject two very short pulses into two channels running the same function. The channels will be serviced sequentially, so if you keep the pulse width shorter than the thread length than the second thread will execute with the input pin low.

## 6. Script Commands Files

The second situation in which it may be difficult to achieve complete event vector coverage is when there are multiple event vectors that handle invalid cases. For instance, all functions must handle links, even when a link is not part of the functions normal operation. Such a link could occur if there was a bug in another function. Since there are number of such invalid situations, they are typically grouped. As such, it may be justified to bundle these together using the following script command. This command allows coverage of a single event vector to count as coverage for other (inferred) event vectors.

```
infer_entry_coverage(FuncNum, FromEntryIndex, ToEntryIndex);
```

Consider the following thread labeled, "invalid\_flag0". This thread is never expected to occur because the function clears flag0 at initialization, and flag0 is never set. So thus state which handles a match or transition event in which the flag0 condition is set should never execute.



```
eTpu1: Source: MeasurePeriod.c
else if( ( IsMatchBORTransitionEvent() && (flag0==1) ))
005A: 0x4098 Std Entry 13, Addr 0x26C, EnableMatches, p=((relative U24 *) 0x0), diob=((relative U24 *) 0x4) [0]
      : HSR==0b000 Link==0 MatchA/TranB==0 MatchB/TranA==1 InputPin==0 ChanFlag1=X ChanFlag0==1
005E: 0x4098 Std Entry 15, Addr 0x26C, EnableMatches, p=((relative U24 *) 0x0), diob=((relative U24 *) 0x4) [0]
      : HSR==0b000 Link==0 MatchA/TranB==0 MatchB/TranA==1 InputPin==1 ChanFlag1=X ChanFlag0==1
006A: 0x4098 Std Entry 21, Addr 0x26C, EnableMatches, p=((relative U24 *) 0x0), diob=((relative U24 *) 0x4) [0]
      : HSR==0b000 Link==0 MatchA/TranB==1 MatchB/TranA==1 InputPin==0 ChanFlag1=X ChanFlag0==1
006E: 0x4098 Std Entry 23, Addr 0x26C, EnableMatches, p=((relative U24 *) 0x0), diob=((relative U24 *) 0x4) [0]
      : HSR==0b000 Link==0 MatchA/TranB==1 MatchB/TranA==1 InputPin==1 ChanFlag1=X ChanFlag0==1
{
  invalid_flag0:
  // Mark invalid flag0 state to make observable
  // for debug purposes ...
  Clear(flag0);
026C: 0xFFFFCF9 chan_clear_ChannelFlag0; FormatD3 [4]
      : BadStateFault = 0x334698;
0270: 0x0A633580 au P = 0x334698; FormatA1 [4]
      : }
0274: 0xAFFFFB83 ram *((relative U24 *) 0xD) = p_23_0; FormatB2 [0]
      : 0xAFFFFB83 seq end;; FormatB2 [4]
```

A test has been written to excersize this thread, and one can see that Event vector 15 has been covered because the box on the left is white. But entries 13, 21, and 23 have not been executed because the boxes on the left are still black. Since this is an invalid case that actually should never execute, it is considered sufficient to infer coverage of entries 13, 21, and 23, as long as event vector 15 is covered. This is done using the following script command.

```
infer_entry_coverage(MEASURE_PERIOD_FUNC, 15, 13);
infer_entry_coverage(MEASURE_PERIOD_FUNC, 15, 21);
infer_entry_coverage(MEASURE_PERIOD_FUNC, 15, 23);
```

Although there are a number of restrictions listed below that are enforced by the eTPU Simulator, the most important restriction is not enforced. Namely, that this coverage by inference should only used for invalid cases where the thread exists purely as Built In Test (BIT) and would not in normal operation be expected to execute. In fact, when testing to the very highest software testing standards, 100 percent event vector coverage should be achieved without the use of this script command.

- Execution of an event vector that is covered by inference results in a verification failure
- The FromEntryIndex's thread and the ToEntryIndexthread thread must be the same.

### Cumulative Coverage

To produce the highest quality software it is imperative that testing cover 100% of instructions, branches, and event vectors (for eTPU targets). Additionally, for quality control purposes, this coverage should be proven using the verify\_coverage scripts. But most test suites consist of multiple tests, such that the coverage is achieved only after all tests have run. The cumulative coverage scripts provide the ability to prove that the entire test suite cumulatively has achieved 100% coverage.

The typical testing procedure might work as follows. A series of tests is run, and at the end of each test the coverage data is stored. At the end of the very last test, the coverage data from all previous tests are loaded such that the resulting coverage is an accumulation of the coverage of all previous tests. Then the verify\_coverage script command is run proving that all tests have passed. The following illustrates this process.

```
... Run Test A.  
save_cumulative_file_coverage("MyFunc.c", "TestA.CoverageData");  
  
... Run Test B.  
save_cumulative_file_coverage("MyFunc.c", "TestB.CoverageData");  
  
...  
  
... Run Test M.  
save_cumulative_file_coverage("MyFunc.c", "TestM.CoverageData");  
  
... Run Test N.  
load_cumulative_file_coverage("MyFunc.c", "TestA.CoverageData");  
load_cumulative_file_coverage("MyFunc.c", "TestB.CoverageData");  
...  
load_cumulative_file_coverage("MyFunc.c", "TestM.CoverageData");  
verify_file_coverage("MyFunc.c ",100,100,100);
```

## 6. Script Commands Files

---

### 6.7.14 Channel Function Select Register Commands

```
write_chan_func(ChanNum,Val);
```

This command sets channel CHANNUM to function Val.

```
#define MY_FUN_NUM 0x10  
write_chan_func(7, 0x10);
```

In this example channel 7 is set to function 10 (decimal). All other channel function selections remain unchanged.

In the Byte Craft eTPU "C" compiler the function number can be automatically generated using the following macro.

```
#pragma write h, ( #define MY_FUNC_NUM ::ETPUfunctionnumber(Pwm));
```

### 6.7.15 Channel Priority Register Commands

```
write_chan_cpr(ChanNum,Val);
```

This command writes the priority assignment Val to the CPR for channel ChanNum.

```
write_chan_cpr(6,2);
```

In this example a middle priority level (2=middle priority) is assigned to channel 6 by writing a two to the CPR bits for channel 6. All other TPU channel priority assignments remain unchanged.

### 6.7.16 Pin Control Script Commands

eTPU input pins are normally controlled using test vector files. eTPU output pins and TPU I/O pins configured as outputs are normally controlled by the eTPUPU and are verified using master behavior verification test files described in the Functional Verification section.

Therefore, these commands are not the primary method for controlling and verifying pin states. Instead, these commands serve as a secondary capability for pin state control and verification.

```
write_chan_input_pin(ChanNum, Val);  
write_tcrclk_pin(Val);  
verify_chan_output_pin(ChanNum, Val);  
verify_ouput_buffer_disabled();  
verify_ouput_buffer_enabled();
```

These commands write either ChanNum's or the TCRCLK pin to value Val, verify that ChanNum's pin is equal to Val, or verify that an output buffer is enabled or disabled.

```
write_chan_input_pin(25, 0);  
write_tcrclk_pin(1);
```

In this example, channel 25's input pin is cleared to a low, and the TCRCLK pin is set high.

```
verify_chan_output_pin(5, 1);  
verify_ouput_buffer_disabled();  
wait_time(5);  
verify_chan_output_pin(5, 0);  
verify_ouput_buffer_enabled();
```

In this example channel 5's output pin is verified to have a falling transition within a 5-micro-second window. It is also verifying that the pin is acting like an open-drain (active low, passive high.)

### 6.7.17 Pin Transition Behavior Script Commands

The pin transition behavior capabilities allow the user to generate behavioral models of the source code and to verify the source code against these saved behavioral models. The script commands allow the user to both create pin transition behavioral model and automate the verification process. Script command capabilities include the ability to save and load pin transition behavior files, the ability to enable continuous verification against these models and control tolerance of tests, and the ability to perform a complete verification of all recorded behavior at once.

NEW : Enhanced behavior verification starting with release 5.00 provides much more flexible and useable pin transition verification capabilities. With enhanced behavior verification pin transition data is saved in a new format that is .csv (comma separated value) format for compatibility with many other tools. However, the enhanced behavior verification can read original style .bv files as well as the new .ebv files. Existing scripting commands apply to original .bv files only where noted.

A more complete discussion of functional verification is given in the Functional Verification chapter while a discussion of the specifics of pin transition behavioral modeling is given in the Pin Transition Behavior Verification section. With the new enhanced behavior verification the loaded master pin transition behavior data can be viewed in the logic analyzer, see the Logic Analyzer section.

### Deprecated Behavior Verification Script Commands

```
read_behavior_file("filename.bv");
```

This command loads the pin transition behavior file into the master pin transition behavior buffer. This buffer forms a behavioral model of the pin transition behavior of the source

## 6. Script Commands Files

---

code. Only old-style .bv files can be read with this command.

```
verify_all_behavior();
```

This command verifies all recorded pin transition behavior against the master pin transition behavior buffer. It generates a behavior verification error message and increments the behavior failure count for each deviation from the behavioral model. Only old-style .bv files can be verified with this command.

```
enable_continuous_behavior();
```

This command enables continuous verification of pin transition behavior against the master pin transition behavior buffer. During source code simulation each functional deviation generates a behavior verification error message and causes the behavior verification failure count to be incremented. This is useful for identifying the specific areas in which the microcode behavior has changed. This command only applies to verification with old-style .bv files. Enhanced behavior verification automatically runs in continuous mode.

```
disable_continuous_behavior();
```

This command disables continuous verification of pin transition behavior against the master pin transition behavior buffer. Note that pin transition behavior is still recorded in the pin transition behavior buffer. This command only applies to verification with old-style .bv files. Enhanced behavior verification automatically runs in continuous mode.

```
resize_pin_buffer(<NumPinTransitions>);
```

This command resizes the pin transition buffer. The default size is 100K transitions.

```
resize_pin_buffer(500000);
```

In this example the pin transition buffer size is changed such that it can hold 500K pin transitions. This script command should only be executed at time zero.

Note that resizing the pin transition buffer can have serious affects on performance. For instance it can cause a long delay when the eTPU Simulator is reset. It can also significantly slow down the logic analyzer redraw rate, such that the simulation speed is bound by the redraw rate. Simulation speed reductions can be obviated by hiding or minimizing the logic analyzer while the eTPU Simulator runs, such that redraws are not required, thereby improving simulation speed.

The effective pin transition buffer size can also be increased in other ways. This is discussed in the Logic Analyzer Options Dialog Box section.

## Enhanced Behavior Verification Script Commands

The test tolerance commands apply whether an old-style .bv file or new .ebv file is being

used as the master, however, the file manipulation script commands only apply to .ebv files.

```
create_ebehavior_file("filename.ebv");
```

This command creates an enhanced behavior data file with the specified name. .ebv file creation in a script command file has the following limitations:

- only one can be created
- cannot have the same file name as a running (under verification) .ebv file
- must occur at simulation time 0 before any wait\_time() or at\_time() script commands
- must occur after any external gate instantiation or after any vector file commands

```
add_ebehavior_pin("<pin name>");
```

By default, when an enhanced behavior data file is created all pins will be saved out (on the TPU, that is channel 0 - 15 and \_tcr2, while on the eTPU, that is all 32 channel input, 32 channel output pins, and \_tcrclk). With this script command, the user can select just the pins desired to be saved to the .ebv file, as this is generally just a small subset. The <pin name> argument is the name provided in the .vector file through the "node" command, or is the default name of the pin (e.g. \_ch3.out is the channel 3 output pin on the eTPU, or \_ch10 is the channel 10 pin on the TPU). These script commands must immediately proceed the create\_ebehavior\_file() script command. Once one add\_ebehavior\_pin() script command is used, all pins of interest must be added with this script command as the default of all is disabled.

```
close_ebehavior_file();
```

Closes an enhanced behavior file that was created, saving off any remaining buffered data to file. When using the created .ebv file, verification should be stopped at the same time as when the .ebv file data stopped being recorded; see stop\_ebehavior\_file().

```
run_ebehavior_file("filename.ebv");
```

This script command opens the specified enhanced behavior verification file. The loaded file is often referred to as the "master file" or "gold file". By default, all pins found in the .ebv file are enabled for verification. The script command has the same limitations as create\_ebehavior\_file() script command, in that it must be called out at a simulation time of 0, etc. The default tolerance for all pin transition data being verified is two system clocks and zero offset.

```
set_ebehavior_pin_tolerance("<pin name>", <time tolerance mode>, <time offset in us>, <time tolerance in us>);
```

Sets the time tolerance allowed between the loaded master file and the current simulation run for the specified pin. The default is to verify all the pins in the .ebv file (or .bv file if an

## 6. Script Commands Files

---

old-style behavior data file is loaded), but once one of these script commands is specified then each pin to be verified must be specified with a `set_ebehavior_pin_tolerance()` script command. The `<pin name>` argument is the name provided in the `.vector` file through the "node" command, or is the default name of the pin. The only available time tolerance mode currently available is "EBV\_ABSOLUTE" - this means that pin transition times in the gold file will be compared directly to the pin transition times that occur in the current simulation run, taking into account the offset and tolerance. The time offset in microseconds is an adjustment made to pin transition data in the gold file before comparing to the simulation pin transition time. For example, if additional code in the initialization of an eTPU function has caused it to start outputting a signal 2us later than previously, then a time offset of 2us can be specified and a smaller time tolerance used in the comparison. The time offset can be positive or negative. The time tolerance controls the maximum amount of difference between the master/gold file pin transition time and the simulation pin transition time before a behavior verification error is thrown. Behavior verification throws an error if the absolute value of the time difference between a gold file pin transition time and the current simulation pin transition time exceeds the specified tolerance. In general, these script commands should immediately follow the `run_ebehavior_file()` script command, although tolerances can be changed on the fly during simulation.

```
set_ebehavior_global_tolerance(<time tolerance mode>, <time
offset in us>, <time tolerance in us>);
```

This script commands sets behavior verification tolerances for all pins of interest (default to all, or those specified with `set_ebehavior_pin_tolerance`). The arguments - mode, time offset and time tolerance are described in the `set_ebehavior_pin_tolerance()` documentation.

```
stop_ebehavior_file();
```

Used to end enhanced behavior verification and close the `.ebv` file being used as the master/gold file. For the pins being verified, the gold file and current simulation must match at this time or a behavior verification error will occur. The `stop_behavior_file()` script command should occur at the same time as the `close_ebehavior_file()` command did during enhanced behavior file creation.

### 6.7.18 Thread Script Commands

The eTPU Simulator stores worst-case latency information for each channel. This is very useful for optimizing system performance. But in some applications the initialization code, which generally does not contribute to worst case latency, experiences the worst case thread length. In this case, it is best to ignore the initialization threads when considering the worst case thread length for a function. This command ignoring the initialization threads.

```
// Initialize the channels.
write_chan_hsrr ( RCV_15_A, ETPU_ARINC_RX_INIT);
write_chan_hsrr ( RCV_15_B, ETPU_ARINC_RX_INIT);

// Wait for initialization to complete,
// then reset the worst case thread indices.
wait_time(100);
clear_worst_case_threads();
```

In this example the channels are issued a host service request, then after 100 microseconds (presumably sufficient time to initialize) the threads indices are reset.

### 6.7.19 Disable Messages Script Commands

Note that this command has been DEPRECATED! Please use the `-Quiet` and the `-ILF5` command line options to suppress generation of these messages in an automated regression testing environment.

Pin transition verification failures and script command verification failures result in a non-zero return code when the application exits, as well as display of a dialog box to inform the user of the failure. This dialog box can be disabled in the Messages dialog box, but this must be done manually each time the application is launched. In certain cases, such as tool qualification under DO178B, it is desirable to disable display of the dialog box such that the test of the verification test can be automated. This is done as follows.

```
disable_message( PIN_TRANSITION_FAILURE );
disable_message( SCRIPT_FAILURE );
```

THESE COMMANDS SHOULD BE USED WITH EXTREME CAUTION! They remove observe-ability from verification failures.

In all cases except automation of test of the verification tests, it is preferable to disable the display of the verification messages manually from within the Messages dialog box.

### 6.7.20 eTPU System Configuration Commands

```
write_entry_table_base_addr(Addr);
```

This command writes the Event Vector Table's address. It writes a value into the ETPUECR register's ETB field that corresponds to address Addr.

```
#define MY_ENTRY_TABLE_BASE_ADDR 0x800
write_entry_table_base_addr(MY_ENTRY_TABLE_BASE_ADDR);
```

## 6. Script Commands Files

---

In the above example the event vector table is placed at address 0x800.

```
write_engine_relative_base_addr(Addr);
```

This (eTPU2 ONLY!) command writes the Engine Relative Base Address. It writes a value into the ETPUECR register's ERBA field that corresponds to address Addr.

```
#define MY_ENGINE_ADDR 0xA00
write_engine_relative_base_addr(MY_ENGINE_ADDR);
```

In the above example the engine relative space is placed (allocated) at address 0xA00.

```
write_scheduler_priority_passing_disable(Val);
```

This (eTPU2 ONLY!) command writes the Scheduler Priority Passing Disable bit. It writes the specified 0 or 1 value into the SPPDIS bit of the ETPUECR register. The default value is 0.

```
write_scheduler_priority_passing_disable(1);
```

In the above example priority passing in the scheduler is disabled.

```
write_global_time_base_enable();
```

The command enables the time bases for all the eTPUs.

```
write_entry_table_base_addr(Addr);
```

In the Byte Craft eTPU "C" Compiler the event vector table base address can be automatically generated using the following macro.

```
#pragma write h, ( #define MY_ENTRY_TABLE_BASE_ADDR ::ETPUentrybase(0));
```

In the Byte Craft eTPU "C" Compiler the event vector table base address is specified as follows:

```
#pragma entryaddr 0x800;
```

This command writes the SCMOFFDATAR register. This register is the opcode that gets executed when the eTPU executes from an SCM address that is not populated with actual memory.

```
write_scm_off_data( Val );
```

### 6.7.21 eTPU Timing Configuration Commands

```
write_angle_mode(Val);
write_tcr1_control(Val);
write_tcr2_control(Val);
New
4.00 write_tcr1_source(Val);
```

These commands write their respective field values in the ETPUTBCR register.

```
write_tcr1_prescaler(Prescaler);
write_tcr2_prescaler(Prescaler);
```

These commands write the prescaler 'Prescaler' to the ETPUTBCR register. Valid values for TCR1 are 1..256 and for TCR2 are 1..64.

```
write_angle_mode(0);           // Disable
write_tcr1_control(1);        // System clock/2
write_tcr2_control(2);        // TCR2's clk is falling TCRCLK Pin
write_tcr1_prescaler(1);      // Fastest ... divide by 1
write_tcr2_prescaler(64);     // Slowest ... divide by 64
```

In this example angle mode is disabled, the TCR1 counter is programmed to be equal to the system clock divide by two (system clock divided by two, prescaler is divides by one), and TCR2 is programmed to be the system clock divided by 512 (system clock divided by 8 with a 64 prescaler.)

```
write_tcr1_control(2);        // System clock/1 (eTPU2 Only!)
write_tcr1_source(1);         // System Clock/1 (eTPU2 Only!)
write_tcr1_prescaler(1);      // Fastest ... divide by 1
```

In the above example system clock/1 is the input to the TCR1 prescaler. This is ONLY available in the eTPU2!

```
set_angle_indices(<DegreesPerCycle>, <TeethPerCycle>);
```

This command supports specification of angle indices required to display current angular information in various portions of the visual interface including the, "Global Time and Angle Counters" window. In a typical automotive application the angle hardware is used as a PLL on the actual engine. Typically two engine revolutions are defined as a single "cycle" so a cycle is defined as 720 degrees. Also, a typical crank has 36 teeth and rotates twice per engine revolution. The following script command generates this configuration.

```
// This configures the visualization of the crank
#define DEGREES_PER_CYCLE 720
#define TEETH_PER_CYCLE 72
set_angle_indices(DEGREES_PER_CYCLE, TEETH_PER_CYCLE);
```

This command configures angle visualization for a cycle of 720 degrees, and a crank with 36 teeth which rotates twice per cycle yielding 72 teeth per cycle.

### 6.7.22 eTPU STAC Bus Script Commands

The STAC Bus for sharing time bases (TCR1, TCR2) between eTPU engines on dual-eTPU microcontrollers can be configured with the following script commands:

```
write_stac_tcr1_enable();
```

## 6. Script Commands Files

---

```
write_stac_tcr1_assignment(Mode);
write_stac_tcr1_server(ServerID);
write_stac_tcr2_enable();
write_stac_tcr2_assignment(Mode);
write_stac_tcr2_server(ServerID);
```

The enable commands allow the specified time base to be exported to or imported from the STAC Bus. Whether the time base acts as a server or client is determined by Mode, where a Mode of 0 is client operation, and a Mode of 1 is server operation. When in client mode, the ServerID determines where the time base is to be imported from. The server numbers for the time bases are hardcoded in hardware as follows:

```
0 - eTPU_A TCR1
1 - eTPU_B TCR1
2 - eTPU_A TCR2
3 - eTPU_B TCR2
```

A typical usage of the STAC Bus is to export the TCR2 time base (angle) from eTPU\_A to eTPU\_B. This could be accomplished with the following set of script commands (assumes other time base configuration already complete):

```
// configure STAC Bus
eTPU_A.write_stac_tcr2_enable(1);
eTPU_A.write_stac_tcr2_assignment(1); // server
eTPU_B.write_stac_tcr2_enable(1);
eTPU_B.write_stac_tcr2_assignment(0); // client
eTPU_B.write_stac_tcr2_server(2); // import eTPU_A's TCR2

// enable timers
write_global_time_base_enable(1);
```

### 6.7.23 eTPU Global Data Write/Verify Commands

```
write_global_data32(AddrOffset, Val);
write_global_data24(AddrOffset, Val);
write_global_data16(AddrOffset, Val);
write_global_data8 (AddrOffset, Val);

verify_global_data32(AddrOffset, Val);
verify_global_data24(AddrOffset, Val);
verify_global_data16(AddrOffset, Val);
verify_global_data8 (AddrOffset, Val);
```

These commands write global data at address AddrOffset to value Val, or verify that the data at address AddrOffset matches value Val. Note that 32-bit numbers must be located

on a double even address boundary (0, 4, 8, ...) that 24-bit numbers must be located on a single-odd boundary (1, 5, 9, ...), that 16-bit accesses must be located on even boundaries (0,2,4,...) and that 8-bit numbers can be on any address boundary.

The address is specified as an offset from the base of parameter RAM.

```
write_global_data32 (0x20, 0xC6E2024A );
verify_global_data32(0x20, 0xC6E2024A );
verify_global_data24(0x21, 0xE2024A );
verify_global_data16(0x20, 0xC6E2 );
verify_global_data16(0x22, 0x024A );
verify_global_data8 (0x20, 0xC6 );
verify_global_data8 (0x21, 0xE2 );
verify_global_data8 (0x22, 0x02 );
verify_global_data8 (0x23, 0x4A );
```

In this example data at an address offset of 0x20 (relative to that eTPU's engine base address) word is written with a 32-bit value 0xC6E2024A (hex). The written value is then verified as 32-, 24-, and 8-bit sizes.

Note the ETEC eTPU C Compiler automatically generates all needed global variable address data into the auto-defines file as a series of macros; no explicit user effort is required.

Bitwise access to engine-space parameter RAM is supported with a set of matching functions to those above.

```
write_global_bits32(AddrOffset, BitOffsetFromMSB, BitSize, Val);
write_global_bits24(AddrOffset, BitOffsetFromMSB, BitSize, Val);
write_global_bits16(AddrOffset, BitOffsetFromMSB, BitSize, Val);
write_global_bits8 (AddrOffset, BitOffsetFromMSB, BitSize, Val);

verify_global_bits32(AddrOffset, BitOffsetFromMSB, BitSize, Val);
verify_global_bits24(AddrOffset, BitOffsetFromMSB, BitSize, Val);
verify_global_bits16(AddrOffset, BitOffsetFromMSB, BitSize, Val);
verify_global_bits8 (AddrOffset, BitOffsetFromMSB, BitSize, Val);
```

These script commands allow the writing and verification of bitfields within the specified unit. The bit offset is from the MSB of the unit. For example, to set a `_Bool` bit that is in the LSB of an 8-bit unit, the script command would be:

```
// set _Bool at LSB of 8-bit unit to 1
write_global_bits8( 0x10, 7, 1, 1 );
```

## 6. Script Commands Files

---

### 6.7.24 eTPU Channel Data Script Commands

```
write_chan_data32(ChanNum, AddrOffset, Val);
write_chan_data24(ChanNum, AddrOffset, Val);
write_chan_data16(ChanNum, AddrOffset, Val);
write_chan_data8 (ChanNum, AddrOffset, Val);
verify_chan_data32(ChanNum, AddrOffset, Val);
verify_chan_data24(ChanNum, AddrOffset, Val);
verify_chan_data16(ChanNum, AddrOffset, Val);
verify_chan_data8 (ChanNum, AddrOffset, Val);
```

These command write channel ChanNum's data at address AddrOffset to value Val, or verify that the data at the specified parameter RAM memory location is value Val. Note that 32-bit numbers must be located on a double even address boundary (0, 4, 8, ...) that 24-bit numbers must be located on a single-odd boundary (1, 5, 9, ...), that 16-bit accesses must be located on even boundaries (0,2,4,...) and that 8-bit numbers can be on any address boundary.

```
#define UART_CHAN 12
write_chan_data32 ( UART_CHAN, 0x20, 0xC6E2024A );
verify_chan_data32( UART_CHAN, 0x20, 0xC6E2024A );
verify_chan_data24( UART_CHAN, 0x21, 0xE2024A );
verify_chan_data16( UART_CHAN, 0x20, 0xC6E2 );
verify_chan_data16( UART_CHAN, 0x22, 0x024A );
verify_chan_data8 ( UART_CHAN, 0x20, 0xC6 );
verify_chan_data8 ( UART_CHAN, 0x21, 0xE2 );
verify_chan_data8 ( UART_CHAN, 0x22, 0x02 );
verify_chan_data8 ( UART_CHAN, 0x23, 0x4A );
```

In this example channel 12's data at an address offset of 0x20 (relative to that channel's base address) word is written with a 32-bit value 0xC6E2024A (hex). The written value is then verified as 32-, 24-, and 8-bit sizes.

In the Byte Craft eTPU "C" Compiler the address offset can be generated using the following #pragma in the code:

```
#pragma write h, ( #define MY_ADDR_OFFSET ::ETPUlocation(Pwm, MyFuncVar));
```

The ETEC eTPU C Compiler automatically generates all needed address data into the auto-defines file; no explicit user effort is required.

Bitwise access to parameter RAM is supported with a set of matching functions to those above.

```
write_chan_bits32(ChanNum, AddrOffset, BitOffsetFromMSB, BitSize, Val);
write_chan_bits24(ChanNum, AddrOffset, BitOffsetFromMSB, BitSize, Val);
```

```

write_chan_bits16(ChanNum, AddrOffset, BitOffsetFromMSB, BitSize, Val);
write_chan_bits8 (ChanNum, AddrOffset, BitOffsetFromMSB, BitSize, Val);
verify_chan_bits32(ChanNum, AddrOffset, BitOffsetFromMSB, BitSize, Val);
verify_chan_bits24(ChanNum, AddrOffset, BitOffsetFromMSB, BitSize, Val);
verify_chan_bits16(ChanNum, AddrOffset, BitOffsetFromMSB, BitSize, Val);
verify_chan_bits8 (ChanNum, AddrOffset, BitOffsetFromMSB, BitSize, Val);

```

These script commands allow the writing and verification of bitfields within the specified unit. The bit offset is from the MSB of the unit. For example, to set a `_Bool` bit that is in the LSB of an 8-bit unit, the script command would be:

```

// set _Bool at LSB of 8-bit unit to 1
write_chan_bits8( BOOL_CHAN, 0x10, 7, 1, 1 );

```

### 6.7.25 eTPU Channel Address Script Commands

```

write_chan_base_addr(ChanNum, Addr);

```

This command writes channel `ChanNum`'s address `Addr`. Note that this writes the CPBA register value.

```

#define PWM1_CHAN 3
#define PWM2_CHAN 4
#define PWM1_CHAN_ADDR 0x300
#define PWM2_CHAN_ADDR (PWM1_CHAN_ADDR + PWM_RAM)
write_chan_base_addr(PWM1_CHAN, PWM1_CHAN_ADDR);
write_chan_base_addr(PWM2_CHAN, PWM2_CHAN_ADDR);

```

In this example channel 3's data will start at address 0x300. Note channel variables and static local variables use this. Each channel should be given its own

### 6.7.26 eTPU Engine Data Script Commands

These engine data script commands are only available on the eTPU2 products.

```

write_engine_data32(AddrOffset, Val);
write_engine_data24(AddrOffset, Val);
write_engine_data16(AddrOffset, Val);
write_engine_data8 (AddrOffset, Val);

verify_engine_data32(AddrOffset, Val);
verify_engine_data24(AddrOffset, Val);
verify_engine_data16(AddrOffset, Val);
verify_engine_data8 (AddrOffset, Val);

```

These commands write engine data at address `AddrOffset` to value `Val`, or verify that the

## 6. Script Commands Files

---

data at address AddrOffset matches value Val. Note that 32-bit numbers must be located on a double even address boundary (0, 4, 8, ...) that 24-bit numbers must be located on a single-odd boundary (1, 5, 9, ...), that 16-bit accesses must be located on even boundaries (0,2,4,...) and that 8-bit numbers can be on any address boundary.

The address is formed by adding the engines base address (see ERBA.) with the address formed in the by the Addroffset field. See the System Configuration Commands section for information on how the Engine Relative Base Address (ECR.ERBA) field is writtenwrite\_engine\_data32 (0x20, 0xC6E2024A );

```
verify_engine_data32(0x20, 0xC6E2024A );
verify_engine_data24(0x21, 0xE2024A );
verify_engine_data16(0x20, 0xC6E2 );
verify_engine_data16(0x22, 0x024A );
verify_engine_data8 (0x20, 0xC6 );
verify_engine_data8 (0x21, 0xE2 );
verify_engine_data8 (0x22, 0x02 );
verify_engine_data8 (0x23, 0x4A );
```

In this example data at an address offset of 0x20 (relative to that eTPU's engine base address) word is written with a 32-bit value 0xC6E2024A (hex). The written value is then verified as 32-, 24-, and 8-bit sizes.

Note the ETEC eTPU C Compiler automatically generates all needed engine variable address data into the auto-defines file as a series of macros; no explicit user effort is required.

Bitwise access to engine-space parameter RAM is supported with a set of matching functions to those above.

```
write_engine_bits32(AddrOffset, BitOffsetFromMSB, BitSize, Val);
write_engine_bits24(AddrOffset, BitOffsetFromMSB, BitSize, Val);
write_engine_bits16(AddrOffset, BitOffsetFromMSB, BitSize, Val);
write_engine_bits8 (AddrOffset, BitOffsetFromMSB, BitSize, Val);

verify_engine_bits32(AddrOffset, BitOffsetFromMSB, BitSize, Val);
verify_engine_bits24(AddrOffset, BitOffsetFromMSB, BitSize, Val);
verify_engine_bits16(AddrOffset, BitOffsetFromMSB, BitSize, Val);
verify_engine_bits8 (AddrOffset, BitOffsetFromMSB, BitSize, Val);
```

These script commands allow the writing and verification of bitfields within the specified unit. The bit offset is from the MSB of the unit. For example, to set a \_Bool bit that is in the LSB of an 8-bit unit, the script command would be:

```
// set _Bool at LSB of 8-bit unit to 1
```

---

```
write_engine_bits8( 0x10, 7, 1, 1 );
```

### 6.7.27 eTPU Channel Function Mode Script Command

```
write_chan_mode(ChanNum, ModeVal);
```

This command writes channel ‘ChanNum’ to function mode ‘ModeVal’. Note that this modifies the FM field of the CxSCR register. This is a two-bit field so valid values are 0, 1, 2, and 3.

```
#define PWM1_CHAN 17
write_chan_mode(PWM1_CHAN, 3);
```

In this example, channel 17’s function mode is set to 3

### 6.7.28 eTPU Event Vector Entry Condition (Standard/Alternate) Commands

```
write_chan_entry_condition(ChanNum, Val);
```

This command writes channel ChanNum’s event vector (entry) condition to Val. Note that this writes the CxCR register's ETCS field. A value of 0 designates the standard table and 1 designates alternate. Note that each function has a set value and that this value MUST match that of the eTPU function to which the channel is set.

```
#define UART_STANDARD_ENTRY_VAL    0
#define PWM_ALTERNATE_ENTRY_VAL    1

write_chan_entry_condition(UART1_CHAN, UART_STANDARD_ENTRY_VAL);
write_chan_entry_condition(UART2_CHAN, UART_STANDARD_ENTRY_VAL);

write_chan_entry_condition(PWM1_CHAN, PWM_ALTERNATE_ENTRY_VAL);
write_chan_entry_condition(PWM2_CHAN, PWM_ALTERNATE_ENTRY_VAL);
```

In this example the UART channels are programmed to use the standard event vector table and the PWM channels are programmed to use the alternate event vector table.

In the Byte Craft eTPU "C" Compiler the event vector condition (alternate/standard) for the eTPU function is specified as follows.

```
#pragma ETPU_function Pwm, alternate;

void Pwm ( int24 Period, int24 PulseWidth )
{
  ...
}
```

In the Byte Craft eTPU "C" Compiler the event vector mode can be automatically

## 6. Script Commands Files

---

generated using the following macro.

```
#pragma write h, ( #define PWM_ALTERNATE_ENTRY_VAL ::ETPUentrytype(Pwm));
```

Note that setting of the event vector table's base address is covered in the System configuration commands `SYSTEM_CFG_CMDS` section.

```
write_chan_entry_pin_direction(ChanNum, Val);
```

This command writes channel `ChanNum`'s event vector pin direction to `Val`. Note that this writes the `CxCR` register's `ETPD` field. A value of 0 uses the channel's input pin and a value of 1 uses the output pin.

```
#define ETPD_PIN_DIRECTION_INPUT      0
#define ETPD_PIN_DIRECTION_OUTPUT    1

write_chan_entry_pin_direction(UART1_CHAN, ETPD_PIN_DIRECTION_INPUT);
write_chan_entry_pin_direction(UART2_CHAN, ETPD_PIN_DIRECTION_OUTPUT);
```

In this example the `UART1` chan event vector table thread selection is based on the input pin, and `UART2` event vector thread selection is based on the output pin.

See the System Configuration Commands section for information on setting the entry table's base address.

### 6.7.29 eTPU Interrupt Script Commands

Interrupts can cause special script ISR file to execute as described in the Script ISR section.

```
clear_chan_intr(ChanNum);
clear_chan_overflow_intr(ChanNum);
clear_data_intr(ChanNum);
clear_data_overflow_intr(ChanNum);
```

These commands clear the interrupts for channel `ChanNum`. It is equivalent to setting the bit associated with the channel in the `CICX`, `DTRC`, `CIOC`, or `DTROC` fields.

```
verify_chan_intr(ChanNum, Val);
verify_chan_overflow_intr(ChanNum, Val);
verify_data_intr(ChanNum, Val);
verify_data_overflow_intr(ChanNum, Val);
verify_illegal_instruction(Val);
verify_microcode_exception(Val);
```

These commands verify that the respective interrupts are either asserted (`Val==1`) or de-asserted (`Val==0`).

```
clear_global_exception();
```

This command clears the global exception along with the exception status bits. It is equivalent to setting the GEC field in the ETPUMCR field.

```
disable_chan_intr(ChanNum);
enable_chan_intr(ChanNum);
disable_data_intr(ChanNum);
enable_data_intr(ChanNum);
```

These commands enable/disable the interrupt for channel ChanNum. Note that if you associate a script ISR file with an interrupt, these commands allow or prevent that file from running on assertion of the interrupt

```
clear_this_intr();
```

This command can only be run from within a script ISR file. It clears the interrupt that caused the command to execute.

### 6.7.30 eTPU/TPU Host Service Request Register Script Commands

```
write_chan_hsrr(ChanNum,Val);
```

This command writes channel ChanNum's HSRR bits to value Val.

```
write_chan_hsrr(4,0);
```

In this example channel 4's HSRR bits are written to zero. If channel 4 had a pending host service request, it would be cleared.

### 6.7.31 eTPU/TPU Interrupt Association Script Commands

Interrupt association script commands associate a script commands file with the firing of interrupts such that when the interrupt is both enabled and active, the script commands file executes. See the Script Commands Files chapter for a description of the use of ISR script commands files

```
load_chan_isr("filename.eTpuCommand", ChanNum); // eTPU-Only
load_data_isr("filename.eTpuCommand", ChanNum); // eTPU-Only
load_exception_isr("filename.eTpuCommand"); // eTPU-Only
```

In order for the ISR script to actually execute the ISR must be enabled. The following script commands enable and disable ISRs for both the eTPU and TPU.

```
enable_chan_intr( chanNum ); // eTPU Only
disable_chan_intr( chanNum ); // eTPU Only
enable_data_intr( chanNum ); // eTPU Only
```

## 6. Script Commands Files

---

```
disable_data_intr( chanNum );           // eTPU Only
```

This commands loads ISR script commands file filename.TpuCommand (or filename.eTpuCommand) and associates them with the various types of interrupts from channel ChanNum.

```
close_chan_isr(ChanNum); // eTPU only
close_data_isr(ChanNum); // eTPU only
close_exception_isr();   // eTPU only
```

This eTPU example loads the file ISR\_5.TpuCommand and associates it with the interrupt from channel 5. Any asserted and enabled interrupt on channel 5 during that 2ms window will cause the script commands in the file to be run.

```
load_data_isr("ISR_22.eTpuCommand", 22);
enable_data_intr( 22 );
wait_time(5000);
close_data_isr(22);
disable_data_intr( 22 );
```

This eTPU DATA isr example loads the file ISR\_22.eTpuCommand and associates it with the data interrupt from channel 22. If the interrupt for channel 22 is both asserted and enabled within the first 5ms, then the script commands in the file will run.

```
load_exception_isr("GlobalExc.eTpuCommand"); // eTPU-Only
```

This eTPU example loads the file GlobalExc.eTpuCommand and associates it with the global interrupt.

### 6.7.32 External Logic Commands

Boolean logic that is external to the eTPU/TPU is instantiated through the use of place\_x script commands. Several types of external logic are available. The script command used to instantiate each type of logic is listed below. See the External Logic Simulation chapter for a detailed description of the use of external Boolean logic gates.

- place\_buffer(X,Y) Instantiates a buffer follower
- place\_inverter(X,Y) Instantiates an inverter
- place\_and\_gate(X,Y,Z) Instantiates an "and" gate
- place\_or\_gate(X,Y,Z) Instantiates an "or" gate
- place\_xor\_gate(X,Y,Z) Instantiates an "exclusive or" gate
- place\_nand\_gate(X,Y,Z) Instantiates a "nand" gate
- place\_nor\_gate(X,Y,Z) Instantiates a "nor" gate

- `place_nxor_gate(X,Y,Z)` Instantiates an "inverting exclusive or" gate
- `remove_gate(Z);` Removes the gate that drives channel Z

The eTPU has up to two pins per channel which (depending on the specific device) may or may not actually be connected together or to from outside of the microcontroller. In any case, indexes are defined as follows for the eTPU.

- 0 to 31 Channels 0 through 31 inputs, respectively
- 32 to 63 Channels 0 through 31 outputs, respectively
- 64 TCRCLK pin

```
place_and_gate(5,33,64);
```

This example places an "and" gate with eTPU channels 5's input pin and eTPU channel 2's output pin as inputs and the TCRCLK pin as the output.

### Two eTPU Engine Configurations

In two eTPU configurations it is possible to place gates between the two eTPU's pins. This is done using the following syntax.

- 128 to 159 Other eTPU's channels 0 through 31 input pins
- 160 to 191 Other eTPU's channels 0 through 31 output pins
- 192 Other eTPU's TCRCLK pin

```
place_xor_gate(160, 161,5);
```

This example places a 'XOR' gate from eTPU B's channel 0 and 1 output pins to eTPU A's channel 5 input pin.

### 6.7.33 Build Script Commands

```
instantiate_target(enum TARGET_TYPE, "TargetName");  
instantiate_target_ex1(enum TARGET_TYPE,  
enum TARGET_SUB_TYPE,  
"TargetName");
```

These commands instantiate a target enum TARGET\_TYPE and assigns it the name TargetName. The extended version of this command also supports a target sub type, enum TARGET\_SUB\_TYPE\_TARGET\_SUB\_TYPE. Subsequent references to this target

## 6. Script Commands Files

---

use the target name specified in this command. The second (extended) version of this command supports sub-targets.

An issue that can be important is that the eTPU is used in a variety of microcontroller families (such as Power Architecture and Coldfire) and also in a variety of versions of microcontrollers within each family (such as MPC5554-rev B, MCF5234 rev 0) and that the amount of DATA and CODE memory as well as the errata can vary greatly within microcontroller families and within versions. The ASH WARE simulation model supports selection of specific microcontroller versions via the TARGET\_SUB\_TYPE enumeration.

For the eTPU a particularly interesting file is, “zzz\_eTpuVersions.Dat,” which is found in the BuildScripts directory. This supports selection of the eTPU version (and associated errata and memory sizes) via a #define, which the user can select in a dialog box. See the Build Script Options Dialog Box section.

```
instantiate_target(SIM32, "Host");
instantiate_target_ex1(ETPU_SIM, MPC5554_B_1, "eTpu1");
```

This example instantiates two simulation models, a CPU32 simulation model, and an eTPU simulation model. The name “Host” is assigned to the CPU32 and the name “eTpu1” is assigned to the eTPU. Subsequent build script commands use these names when referring to these targets. These names are also referenced in a variety of other places such as in workshops and the menu system.

```
add_mem_block("TargetName", StartAddress, StopAddress,
              "BlockName", enum ADDR_SPACE);
```

This command adds a memory block to a range of simulated memory. The memory appears between stopAddress and startAddress in the memory space ADDR\_SPACE. The name BlockName is assigned to this block and is used by other script commands when referencing this block. The block name must be unique within its target, but other targets can have a block with this same block name. The size of the memory block is equal to one plus stopAddress minus startAddress. Only a single copy of this memory is created, regardless of how many address spaces the block occupies.

```
add_mem_block("MyCpu", 0, 0xFFFF, "RAM1", ALL_SPACES);
```

In this example a 64K block of simulated memory is created and the name "RAM1" is assigned to this memory block. This memory is accessible from all of the CPU's address spaces.

```
add_non_mem_block("TargetName", StartAddress, StopAddress,
                  enum ADDR_SPACE);
```

This command adds a blank block of simulated memory to the target TargetName. It indicates that no physical memory exists in the specified memory range and specified

address spaces, ADDR\_SPACE. The name BlockName is assigned to this block and is used by other script commands when referencing this block. The block name must be unique within its target, but other targets can have a block with this same block name.

Since no memory is actually modeled by this command, it effectively uses almost none of your computer's virtual memory. This is important since the entire four GB address space must be represented by memory blocks, regardless of whether or not the simulation target actually supports this large of an address space.

```
#define DATA_SPACE      CPU32_SUPV_DATA_SPACE  \
                        + CPU32_USER_DATA_SPACE
add_non_mem_block("MyCpu", 0x1000, 0xffffffff,
                  "Empty", DATA_SPACE);
```

This example specifies that no physical memory exists above the first 64K for both user and supervisor data spaces. Data space is defined by the define declaration as consisting of a combination of the supervisor data space and the user data space.

```
set_block_to_off("TargetName", "BlockName",
                 enum ADDR_SPACE, enum READ_WRITE);
```

This command allows accesses of a simulated memory blocks can be turned off using this script command. Using this command a read-only memory device such as a ROM can be created. Accesses to target TargetName within the block BlockName and specified address spaces ADDR\_SPACE and read and/or write cycles <enum READ\_WRITE> are turned off. A turned-off write access behaves exactly like a normal write access except the actual memory is not written. A turned-off read cycle behaves exactly like a regular read cycle except that the value returned is the OFF\_DATA constant defined for the entire block. The affected address spaces and read/write cycles must be subsets of the referenced memory block.

```
add_mem_block("MyCpy", 0, 0xFFFF, "ROM", ALL_SPACES);
#define ALL_WRITES  RW_WRITE8 + RW_WRITE16 + RW_WRITE32
set_block_to_off("MyCpu", "ROM", ALL_SPACES, ALL_WRITES);
```

This example creates a 64K memory device and configures it to be a read-only or "ROM" memory device.

```
set_block_off_data32("TargetName", "BlockName",
                    enum ADDR_SPACE, OFF_DATA);
set_block_off_data("TargetName", "BlockName",
                   enum ADDR_SPACE, OFF_DATA);
```

These command specifies that read cycles to the target TargetName within the block BlockName return the data <OFF\_DATA> but only if the block is either a "non\_mem" block or a block in which the read cycles have been set to off. The affected address

## 6. Script Commands Files

---

spaces must be a subset of the address spaces to which the referenced memory block applies. The first command sets an eight bit value, the second sets a 32-bit value.

```
add_non_mem_block("eTpu1", 0x4000, 0xFFFFFFFF,
"UnusedCode",
                ETPU_CODE_SPACE);
set_block_off_data32("eTpu1", "UnusedCode",
ETPU_CODE_SPACE,
                0xF7F757FA);
```

In this example the address space between 0x4000 and FFFFFFFF hexadecimal is specified to contain no memory. Quad read cycles to this block will return the specified off data, 0xF7F757FA hexadecimal, at every quad address.

```
set_block_to_bus_fault("TargetName", "BlockName",
                enum ADDR_SPACE, enum READ_WRITE);
```

This command results in bus faults for accesses to the target TargetName within the block BlockName for the applicable address spaces, ADDR\_SPACE, and read/write cycles enum READ\_WRITE. The effected address spaces must be a subset of the spaces to which the referenced memory block applies.

```
add_mem_block("MyCpu32", 0x10000,0xFFFFFFFF, "Unused",
                ALL_SPACES);
set_block_to_bus_fault("MyCpu32", "Unused", ALL_SPACES,
RW_ALL);
```

In this example, a memory block has been added to represent the unused address space above 64K. Any access to this memory block results in a bus fault.

```
set_block_to_priv_viol("TargetName", "BlockName",
                enum ADDR_SPACE, enum READ_WRITE);
```

This command results in privilege violations for accesses to the target TargetName for the memory block BlockName for the applicable address spaces ADDR\_SPACE, and read and/or write cycles enum READ\_WRITE. The affected address spaces must be a subset of the address spaces to which the referenced memory block applies.

```
#define    ALL_DATA_SPACE    CPU32_SUPV_DATA_SPACE \
                + CPU32_USER_DATA_SPACE
add_mem_block("MyCpu32", 0x10000,0x1FFFF, "Protected",
                ALL_DATA_SPACE);
set_block_to_priv_viol("MyCpu32", "Protected",
                CPU32_USER_DATA_SPACE, RW_ALL);
```

In this example, data space accesses to the simulated memory while at the supervisor privilege level will succeed whereas accesses at the user privilege level will result in a privilege violation.

```
set_block_to_addrs_fault("TargetName", "BlockName",
                        enum ADDR_SPACE, enum READ_WRITE);
```

This command results in address faults for odd accesses to the target TargetName within the block BlockName for the applicable address spaces ADDR\_SPACE, and read and/or write cycles enum READ\_WRITE. The affected address spaces must be a subset of the address spaces to which the referenced memory block applies. Even accesses will not result in an address fault.

```
set_block_to_addrs_fault("MyCpu16", "EvenMem",
                        CPU16_CODE_SPACE, RW_ALL);
```

In this example code space accesses to odd addresses are configured to result in an address fault.

```
set_block_timing("TargetName", "BlockName",
                enum ADDR_SPACE, enum READ_WRITE,
                ClockPerEvenAccess, ClocksPerOddAccess);
```

This command sets the timing for the target TargetName in the block BlockName. This command applies only to memory spaces ADDR\_SPACE, and for the read and/or write cycles enum READ\_WRITE. Even accesses are set to ClocksPerEvenAccess while odd accesses are set to ClocksPerOddAccess.

```
#define NOT_DATA_SPACE ALL_SPACES - CPU16_DATA_SPACE
add_mem_block("MyCpu16", 0, 0xFFFF, "SlowMem", ALL_SPACES);
set_block_timing("MyCpu16", "SlowMem", CPU16_DATA_SPACE,
                RW_READ, 4, 8);
set_block_timing("MyCpu16", "SlowMem", CPU16_DATA_SPACE,
                RW_WRITE, 2, 3);
set_block_timing("MyCpu16", "SlowMem", NOT_DATA_SPACE,
                RW_ALL,
                5, 6);
```

In this example the even data reads are set to take four clocks while odd data reads are set to take eight clock cycles. Even data writes take two clock cycles while odd accesses take three. All non-data even reads and writes take five clocks while odd take six.

This example illustrates an important aspect of timing design. A separate copy of the timing data is kept for each address space and for both read and write cycles. So even though only a single memory block was created in this example, timing data for each address space is able to be individually specified.

```
set_block_to_dock("FromTargetName", "BlockName",
                enum ADDR_SPACE, "ToTargetName",
                AddressOffset);
```

## 6. Script Commands Files

---

This script establishes a memory share between a docking target FromTargetName and a "docked-to" second target ToTargetName. Memory accesses for the docking target actually occur in the second target, while this command has no effect on the second target's accesses.

Docking target accesses within the block BlockName in the address space ADDR\_SPACE are projected to the "docked-to" target at an offset address AddressOffset.

The address range corresponds exactly to a previously defined block within the docking target. There is no such requirement for the "docked-to" target.

```
add_mem_block("Cpu_A", 0x0, 0xFFFF, "Shared", ALL_SPACES);
add_non_mem_block("Cpu_B", 0x600, 0x6FF, "ShareRange",
                  ALL_SPACES);
set_block_to_dock("Cpu_B", "ShareRange", ALL_SPACES,
                  "Cpu_A", 0x250);
```

In this example a memory share is setup between targets Cpu\_A and Cpu\_B. The memory that is shared resides in Cpu\_A. The shared block is accessed by Cpu\_B between addresses 600 and 6FF hexadecimal. An offset of 250 hexadecimal is applied to the address of each of Cpu\_B's accesses such that from the perspective of Cpu\_A the accesses occur between 850 and 94F hexadecimal.

Note that, as required, the set\_block\_to\_dock script command has the identical address range as a previous add\_non\_mem\_block script command. Interestingly, there is no such restriction on the Cpu\_A target.

```
set_block_dock_space("TargetName", "BlockName",
                    enum ADDR_SPACE DockFromSpace,
                    enum READ_WRITE,
                    enum ADDR_SPACE DockToSpace);
```

This command supports an address space transformation for a docked memory access. Read and/or write cycles enum READ\_WRITE from target TargetName between within the block BlockName in the address spaces enum ADDR\_SPACE DockFromSpace are transformed to occur in address space enum ADDR\_SPACE DockToSpace. The DockToSpace argument must specify a single space.

It is important to fully specify all shared memory accesses between dissimilar targets. Docks with unspecified address space transformations result in indeterminate results. For instance, a CPU16 sharing memory with a CPU32 could easily result in an opcode being fetched out of data space, even though both targets have both code and data spaces. Assumptions about similarity of address spaces between dissimilar targets simply should

not be made.

```
add_non_mem_block("MyCpu", 0x1000, 0x1003, "ShareRange",
                 ALL_SPACES);
set_block_to_dock("MyCpu", "ShareRange", ALL_SPACES,
                 "MyTpu", 0-0x1000);
set_block_dock_space("MyCpu", "ShareRange", ALL_SPACES,
                    RW_ALL, TPU_PINS_SPACE);
```

In this example a target MyCpu is docked to target MyTpu. An address space transformation is specified such that accesses to any of the CPU's address spaces occur in the TPU's PINS address space.

```
check("TargetName", "ReportFileName");
```

This command does a check on the simulated memory for a target TargetName and creates a report file ReportFileName. The check invoked by this command occurs whether or not this script command is included in the script file. Use of this command allows you to specify the name of the report file and limit the scope of the check to a single target.

```
check("MyCpu", "C:\\Temp\\CpuBuildReport.txt");
check("MyTpu", "TpuBuildReport.txt");
```

In this example report files named C:\\Temp\\CpuBuildReport.txt and TpuBuildReport.txt are generated for the MyCpu and MyTpu targets, respectively. Note that C-style double backslashes are required when separating directory names.

## 6.8 Automatic and Pre-Defined Define Directives

### ASH WARE Specific Script

It is often desirable to conditionally parse (or not parse) portions of a script command file depending on whether or not it is in the ASH WARE development environment. The following #define is automatically prepended when parsing any script file, and therefore can be used to control the aforementioned conditional parsing.

```
#define _ASH_WARE_SCRIPT_ 1
```

This #define is prepended prior to parsing every script file.

```
#ifndef _ASH_WARE_SCRIPT_
void RunEngineDemo()
{
#endif
```

## 6. Script Commands Files

---

The code above causes the function declaration to be ignored by the ASH WARE parser.

### Target-Specific Scripts

It is often desirable to have a single script commands file run on multiple targets. In this case target-dependent behavior is accomplished using the target define. The target define is generated using the target name as follows.

```
#define _ASH_WARE_<TargetName>_ 1
```

TargetName is defined in the build batch file and is found in a pull-down menu in the upper right hand side of the eTPU Simulator.

```
#ifdef _ASH_WARE_DBG32_  
set_crystal_frequency(32768);  
#endif // _ASH_WARE_DBG32_
```

In this example the set\_crystal\_frequency(); script command executes only if the script command is running under a target named DBG32.

### Determining the Tool Versions

The compiler version and simulator version are available as the system macros `__COMPILER_VERSION__` and `__MTDT_VERSION__`. These resolve as strings and are available in script commands such as `print_to_trace()` and `verify_trace()`. See below.

```
print_to_trace("Using compiler version %s and simulator version %s",  
              __COMPILER_VERSION__, __MTDT_VERSION__);  
verify_str("__MTDT_VERSION__", "=", "TPU Simulator, Version 3.50 Build D");
```

These can also be used in `@ASH@print` action commands that are embedded in the source code files, as follows.

```
// @ASH@print_to_trace("Compiler version is %s\n", __COMPILER_VERSION__);
```

### Determining the Auto-Run Mode

The eTPU Simulator is often launched as part of an automated test suite. Under these conditions the test starts running and executes to completion (assuming no failures) with no user intervention. The following is automatically defined when the application is launched in auto-run mode.

```
_ASH_WARE_AUTO_RUN_
```

The following is typically found at the end of a script file used as part of an automated test suite.

```
#ifdef _ASH_WARE_AUTO_RUN_
```

```

exit();
#else
print("All tests are done!!");
#endif // _ASH_WARE_AUTO_RUN_

```

## Determining the Interrupt Number

Channel interrupts for channels 0..15 are numbered 0..15.

ISR script commands execute in response to an enabled and asserted interrupt as described in the ISR Script Commands Files section. On the eTPU/TPU each of these script commands has a unique number, as follows.

- Channel interrupts for channels 0..31 are numbered 0..31.
- Data interrupts for channels 0..31 are numbered 31..63.
- The global exception interrupt number is 64.

The 'define' is formed using the target name, as follows.

```

_ASH_WARE_<TargetName>_ISR_

```

When running under a target named, "ETPU", the ISR script loaded for channel 25's channel interrupt is automatically defined as follows.

```

#define _ASH_WARE_ETPU_ISR_ 25

```

If this same script command file is also loaded for the DATA interrupt, then the automatic define would be as follows.

```

#define _ASH_WARE_ETPU_ISR_ 57

```

An example of how this can be used is as follows

```

#define THIS_ISR_NUM    (_ASH_WARE_ETPU_ISR_)
#define THIS_CHAN_NUM  (_ASH_WARE_ETPU_ISR_ & 0x1F)
clear_this_intr();
// Write a signature to indicate that this ISR ran
write_chan_data24 ( THIS_CHAN_NUM, 0xD, 0xFD12A4 + THIS_ISR_NUM);

```

## Passing Defines from the Command Line

When launching the eTPU Simulator it is often useful to pass #define directives to the primary script commands file from the command line. This is explained in detail in the Regression Testing section.

The following command line is found in the batch files used as part of the automated testing of the eTPU simulator

```

echo Running ALUOP B6 Tests ...

```

## 6. Script Commands Files

---

```
eTpuSimulator.exe -pAutoRun.ETpuSysSimProject -d_TEST_ALUOP_B6_  
if %ERRORLEVEL% NEQ 0 ( goto errors )
```

In the primary script file that is part of this project the following command is used to load the executable file that is specific to this test.

```
#ifdef _TEST_ALUOP_B6_  
load_executable("AluopB6.gxs");  
#endif // _TEST_ALUOP_B6_
```

### eTPU Target Pre-Defined Define Directives

The following define directives are automatically loaded and available for the eTPU Simulation target.

```
#define ETPU1 MPC5554_B_1  
#define ETPU2 MPC5554_B_2
```

## 6.9 Listing of Script Enumerated Data Types

Some script commands have pre-defined enumerated types.

### 6.9.1 Script FILE\_TYPE Enumerated Data Type

The following enumerated data type is used to specify the file type used in various script commands. This specifies a dis-assembly, Freescale S Record, Intel Hexadecimal, or C data structure file type.

```
enum FILE_TYPE {DIS_ASM, SRECORD, IHEX, IMAGE, C_STRUCT, }
```

### 6.9.2 Script VERIFY\_FILES Enumerated Data Type

The following enumerated data type is used to specify the expected results of a file comparison.

```
enum VERIFY_FILES_RESULT {  
    FILES_MATCH, FILES_MISMATCH,  
    FILE1_MISSING, FILE2_MISSING, BOTH_FILES_MISSING  
}
```

### 6.9.3 Script FILE\_OPTIONS Enumerated Data Type

The following enumerated data type is used to specify the options when dumping data to a file. The available options depend on the type of file being dumped.

```
enum DUMP_FILE_OPTIONS {

    // Disables listing of address information in dis-assembly
    // and "C" data structure files:
    NO_ADDR,

    // Disables listing of hexadecimal dump, addressing mode,
    // and symbol data
    // in dis-assembly files:
    NO_HEX, NO_ADDR_MODE, NO_SYMBOLS,

    // Adds a #pragma format "val" to each dis-assembly line
    // This is helpful for non-deterministic assembly languages
    // to cause the assembler to generate a deterministic opcode
    YES_PRAGMA,

    // Adds a blank line between assembly lines
    // Handy for finding opcode boundaries in parallel instr sets
    // such as eTPU where a single opcode's dis-assembly
    // can span multiple lines.
    YES_BLANK_LINES,

    // Selects the endian ordering for image
    // and "C" data structure files:
    ENDIAN_MSB_LSB, ENDIAN_LSB_MSB,

    // Selects data size in image and "C" data structure files:
    DATA8, DATA16, DATA32,

    // Selects decimal data instead of hexadecimal
    // for "C" data structure files:
    OUT_DEC,

    // Append to file if it already exists
    // (default is to overwrite any existing file)
    // Available only for Cstruct and Image files
    FILE_APPEND,
```

## 6. Script Commands Files

---

```
// Specifies default options:  
DUMP_FILE_DEFAULT,  
};
```

### 6.9.4 Trace Options Enumerated Data Types

The following enumerated data type is used to specify the event options when saving a trace buffer to a file.

```
enum TRACE_EVENT_OPTIONS {  
// All targets  
STEP, EXCEPTION, MEM_READ, MEM_WRITE, DIVIDER, PRINT,  
  
// eTPU and TPU Targets, only  
TPU_TIME_SLOT, TPU_NOP, TPU_PIN_TOGGLE,  
TPU_STATE_END, TPU_MATCH_CAPTURE,  
TPU_TCR1_COUNTER, TPU_TCR2_COUNTER,  
// NOTE: Same as (TPU_TCR1_COUNTER|TPU_TCR2_COUNTER)  
TPU_TCR_COUNTER,  
  
// All options  
ALL,  
}
```

The following enumerated data type is used to specify the file format options when saving a trace buffer to a file.

```
enum TRACE_FILE_OPTIONS {  
VIEWABLE, // This format is optimized for viewing  
PARSEABLE, // This format is optimized for parsing  
}
```

### 6.9.5 Base Time Options Enumerated Data Type

The following enumerated data type is used to specify the base time for various script commands.

```
enum BASE_TIME { US, NS, PS, }
```

### 6.9.6 Build Script TARGET\_TYPE Enumerated Data Type

The following enumerated data type is used to specify the target type. No mathematical manipulation of this data type is valid.

```
enum TARGET_TYPE {
    TPU_SIM, TPU_DBG,
    ETPU_SIM, ETPU_DBG,
    SIM32, BDM32,
    SIM16, BDM16,
    SIMPPC, DBGPPC,
}
```

### 6.9.7 Build Script TARGET\_SUB\_TYPE Enumerated Data Type

The following enumerated data type is used to specify the target's sub type. No mathematical manipulation of this data type is valid.

```
enum TARGET_SUB_TYPE {
// eTPU2 SIM - Single Engine
New 4.30 MPC5632M_0_A
New 4.30 MPC5633M_0_A
New 4.30 MPC5634M_0_A
New 4.30 SPC563M54_0_A
New 4.30 SPC563M60_0_A
New 4.30 SPC563M64_0_A

// eTPU2 SIM - Dual Engine
New 4.30 MPC5674_2_A // Rev-2, Engine A
New 4.30 MPC5674_2_B // Rev-2, Engine B
New 4.30 MPC5674_0_A // Rev-0, Engine A
New 4.30 MPC5674_0_B // Rev-0, Engine B
New 4.30 JPC563M60_1_A, // Rev-0, Engine A
New 4.30 JPC563M60_1_B, // Rev-0, Engine B

// eTPU SIM - Dual Engine
MPC5566_0_1, // Rev-0, Engine A
MPC5566_0_2, // Rev-0, Engine B
```

## 6. Script Commands Files

---

```
MPC5554_B_1, // Rev-B, Engine A
MPC5554_B_2, // Rev-B, Engine B

// eTPU SIM - Single Engine - 55xx
MPC5567_0_1, // Rev A
MPC5565_0_1, // Rev A
MPC5553_A_1, // Rev A
MPC5534_0_1, // Rev 0

// eTPU SIM - Single Engine - Coldfire
MPC5571_0_1, // Rev 0, MPC5570 and MPC5571
MCF5232_0_1, // Rev 0
MCF5233_0_1, // Rev 0
MCF5234_0_1, // Rev 0
MCF5235_0_1, // Rev 0

// NOTE: THE TPU SIMULATOR DOES NOT SUPPORT THIS
ENUMERATION!
}
```

### 6.9.8 Build Script ADDR\_SPACE Enumerated Data Type

This enumerated data type is used when specifying the applicable address spaces for various build script commands.

```
enum ADDR_SPACE {

// TPU
TPU_CODE_SPACE, TPU_DATA_SPACE, TPU_PINS_SPACE,
TPU_UNUSED_SPACE,

// eTPU
ETPU_CODE_SPACE, ETPU_CTRL_SPACE, ETPU_DATA_SPACE,
ETPU_DATA_24_SPACE,ETPU_NODE_SPACE, ETPU_UNUSED_SPACE,

// CPU32
CPU32_USER_CODE_SPACE, CPU32_SUPV_CODE_SPACE,
CPU32_USER_DATA_SPACE, CPU32_SUPV_DATA_SPACE,
CPU32_UNUSED_SPACE,

// CPU16
CPU16_CODE_SPACE, CPU16_DATA_SPACE, CPU16_UNUSED_SPACE,
```

```
//
ALL_SPACES,
};
```

In the following very specific cases, mathematical manipulation of this enumerated data type is allowed.

- Single instances of values referencing the same target may be added to each other.
- Single instances of values referencing the same target may be subtracted from ALL\_SPACES.

The following are some valid mathematical manipulations of this data type.

```
// The following references
// a CPU32's USER and SUPERVISOR code spaces
CPU32_USER_CODE_SPACE + CPU32_SUPV_CODE_SPACE
// The following references
// all used CPU32 address spaces
ALL_SPACES - CPU32_UNUSED_SPACE
```

The following are some invalid valid mathematical manipulations of this data type.

```
// !!INVALID!! CPU32 and CPU16 cannot be intermixed
CPU32_USER_CODE_SPACE + CPU16_DATA_SPACE
// !!INVALID!!
// The same value cannot be added or subtracted to itself
TPU_PINS_SPACE + TPU_PINS_SPACE
```

### 6.9.9 Build Script READ\_WRITE Enumerated Data Type

This enumerated data type is used when specifying the applicable read and/or write cycles for various build script commands.

```
enum READ_WRITE {
RW_READ8,    RW_READ16,   RW_READ24,
RW_READ32,   RW_READ64,   RW_READ128
RW_WRITE8,   RW_WRITE16,  RW_WRITE24,
RW_WRITE32,  RW_WRITE64,   RW_WRITE128,
RW_ALL,
};
```

Some mathematical manipulations are allowed. Single instances of all but the RW\_ALL values can be added together and single instances of each value may be subtracted from RW\_ALL.

```
#define SOME_READS RW_READ8 + RW_READ16 + RW_READ32
```

## 6. Script Commands Files

---

```
#define NON_ACCESS32S RW_ALL - RW_WRITE32 - RW_READ32
```

In this example, ALL\_READS is defined as any read access, be it an 8-, 16-, or a 32-bit read cycle. NON\_ACCESS32S is defined as all 8-, 16-, 24-, 64-, and 128-bit read and write cycles.

### 6.9.10 eTPU Register Enumerated Data Types

The eTPU register enumerated data types provide the mechanism for referencing the eTPU registers. These enumerated data types are used in commands that reference the TPU registers such as the register write commands that are defined in the Write Register Script Commands section.

The following enumeration provides the mechanism for referencing the eTPU's registers.

```
enum REGISTERS_U32 { REG_P_31_0, };

enum REGISTERS_U24 {
REG_A, REG_B, REG_C_REG, REG_D, REG_DIOB, REG_SR, REG_ERTA,
REG_ERTB, REG_TCR1, REG_TCR2, REG_TICK_RATE, REG_MACH,
REG_MACL, REG_P,
};

enum REGISTERS_U16 {
REG_TOOTH_PROGRAM, REG_RETURN_ADDR, REG_P_31_16,
REG_P_15_0,
},

enum REGISTERS_U8 {
REG_LINK, REG_P_31_24, REG_P_23_16, REG_P_15_8, REG_P_7_0,
},

enum REGISTERS_U5 { REG_CHAN, },

enum REGISTERS_U1 {
REG_Z, REG_C_FLAG, REG_N, REG_V,
REG_MZ, REG_MC, REG_MN, REG_MV,
},
```

The following are examples of how the enumerated register types are used.

```
write_reg32( 0x12345678, REG_P );
verify_reg32( REG_P, 0x12345678 );
write_reg24( 0x123456, REG_A );
```

```
verify_reg24( REG_A, 0x123456 );
write_reg16( 0x1234, REG_RETURN_ADDR );
verify_reg16( REG_RETURN_ADDR, 0x1234 );
write_reg5 ( 0x12, REG_CHAN );
verify_reg5 ( REG_CHAN, 0x12 );
write_reg1 ( 0x1, REG_Z );
verify_reg1( REG_Z, 0x1 );
```



# 7

## Trace Buffer and Files

### Overview

Trace files have two primary purposes; they are useful for generation of a file that appears identical to the trace window but can be loaded into a file viewer to access advanced search capabilities, and they are used to load into a post-processing facility for advanced trace analyses. The various capabilities and settings are focused on these two purposes.

### Generating Viewable Files

Viewable trace files can be generated by selecting the Trace buffer, Save As ... submenu from the Files menu. Note that the trace buffer is about five times larger than what appears in the trace window. A viewable trace file can also be generated using script commands. See the Trace Script Commands section.

Because viewable files are appropriate only for things like advanced search capabilities, no error or warning is generated if the underlying trace buffer has overflowed.

### Generating Parseable Files

Parseable files can be generated only by using the trace commands described in the Trace Script Commands section. To ensure generation of deterministic parseable trace files, these files can be generated only if the selected trace events are enabled within the eTPU Simulator and the trace buffer has not overflowed when generating a trace file from the buffer.

## 7. Trace Buffer and Files

---

For large trace files it is best to use the streaming capability, thereby avoiding possible trace buffer overflow issues.

### **Parsing the Trace File**

All post processing on the trace files should be done on files generated using the "parseable" option.

Although the file format is intended to be self-explanatory, it is purposely left undocumented to retain flexibility for future enhancements. Instead, it is recommended that those wishing to post-process the trace files use the trace file parse source code available from ASH WARE. The public methods in the TraceParser class are documented and will remain stable for future releases.

Post processing of the trace file is an excellent way to analyze important performance indices. For instance, eTPU and TPU latency calculations such as minimum, maximum, average, and standard deviation would be one excellent application.

### **Trace Buffer Size Considerations**

The trace buffer is a set size. To increase the effective size it is often desirable to disable certain types of events. Disabling and enabling of trace events is accomplished by selecting the Trace submenu from the Options menu.

# 8

## Test Vector Files

### Overview

The eTPU and TPU simulation engine provides a complex test vector generation capability. This allows the user to exercise the simulated eTPU with signals similar to those found in a typical eTPU environment. Test vectors are read from user-supplied test vector files and normally have a ".Vector" extension. Test vector files are used for creation of "high" or "low" states, typically at the eTPU's I/O pins. Note that this capability is available only for the eTPU simulation engine.

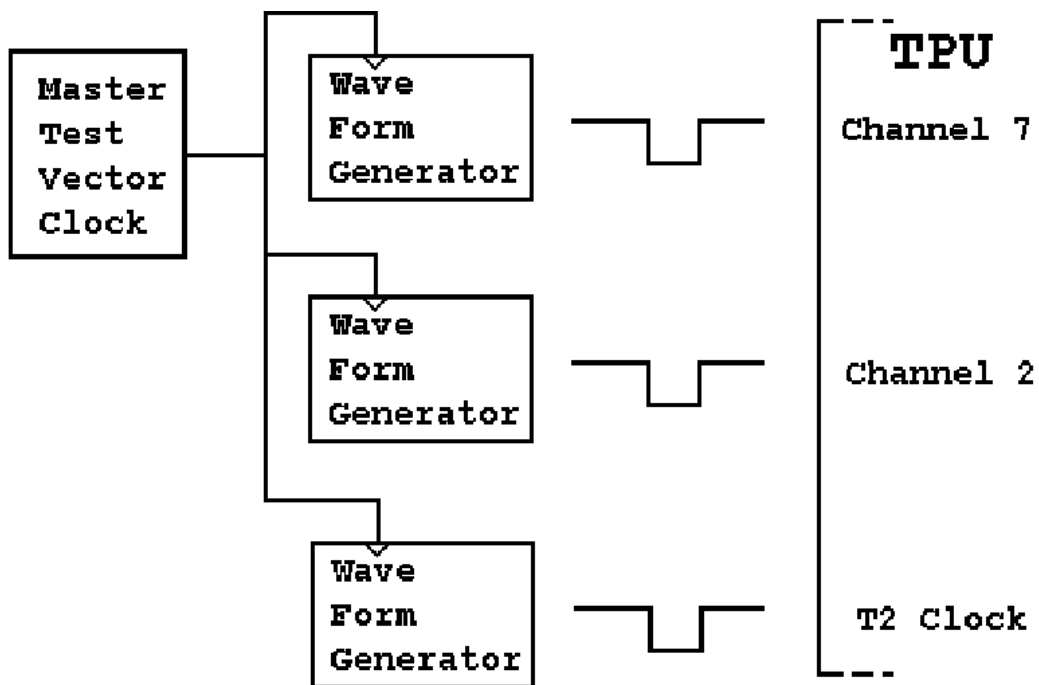
It is helpful to compare test vector files to primary and ISR script commands files. Roughly, test vector files represent the external interface to the eTPU, while script commands files represent the CPU interface. Test vector files are treated quite differently from script commands files. While script commands file lines are executed sequentially and at specific simulated times, test vector files are loaded all at once. Test vector files are used solely to generate complex test vectors on particular eTPU Simulator nodes. As the eTPU Simulator executes, these test vectors are driven unto the specified nodes.

Test vector files are currently available only for TPU and eTPU targets and are target centric in that they can act only on the specific target into which they have been loaded. Future versions of this software will globalize test vector files so that they can act on multiple targets and on any addressable bit in any target's address space.

## Loading and Editing the Test Vector File

The test vector file is loaded into the eTPU Simulator from the “Files” menu by selecting the “Vector, Open ...” sub menu. The test vector file is an ASCII file that the user can edit with any text editor. The configuration window shows the currently-loaded test vector file. It is preferable to keep the test vector file in the same directory as the project file.

## Test Vector Generation Functional Model



The test vector generation model consists of a single master test vector clock and a number of wave form generators. The same master test vector clock clocks all wave form generators. The wave form generators cannot produce waveforms whose frequencies exceed that of the master test vector clock.

A wave form might consist of a loop in which a node is driven high for 10 master test vector clock periods then low for 15. The loop could be set up to run forever.

Test vector files provide the following functionality.

- The master test vector clock frequency is specified.
- The wave form generators are created and defined.
- Wave form outputs are connected to TPU nodes.
- Descriptive names are assigned to TPU nodes (i.e., TPU channel 7's pin is named UART\_RCV).
- Multiple TPU nodes are grouped (i.e., group COMM consists of UART\_RCV1 and UART\_RCV2).
- Complex Boolean states are defined.

Before being parsed, test vector files are run through a C Preprocessor. Thus the files can use the #include mechanism as well as macros and other preprocessor directives and capabilities.

## Command Reference

The following test vector commands are available.

- Node
- Group
- State
- Frequency
- Wave

In addition there is an example of the waveforms generated for an automobile engine monitor system.

## Comments

Test vector files may contain the object-oriented, C++ style double slash comments. A comment field is started with two sequential slash characters, //. All text starting from the slashes and continuing to the end of the line is interpreted as comment. In the following example the first line is a comment while the second is an acceptable test vector command.

```
// This is a comment.  
node AngleP11 ch0.in
```

### 8.1 Node Command

```
node <Name> <Node>
```

The node commands assign the user defined name, “Name” to a node, “Node”. Note that depending on the microcontroller, the input and output from each channel may (or may not) be brought to external pins. Please refer to the Freescale literature for the specific microcontroller being used.

#### Standard eTPU Nodes

- ch0.in Channel 0's input pin
- ch0.out Channel 0's output pin
- ch1.in Channel 1's input pin
- ch1.out Channel 0's output pin
- ...
- ch31.in Channel 31's input pin
- ch31.out Channel 31's output pin
- tcrclk eTPU external clock input pin

In the example shown below the name A429Rcv is assigned to the eTPU's channel 5 input pin.

```
node A429Rcv ch4.in
```

#### Standard TPU Nodes

- CH0 Channel 0's I/O pin
- CH1 Channel 1's I/O pin
- ...
- CH15 Channel 15's I/O pin
- TCR2 TPU Counter 2 gate/clock input pin

In the example shown below the name UART is assigned to the TPU's channel 5 I/O pin.

```
define UART CH5
```

#### Thread Activity Nodes

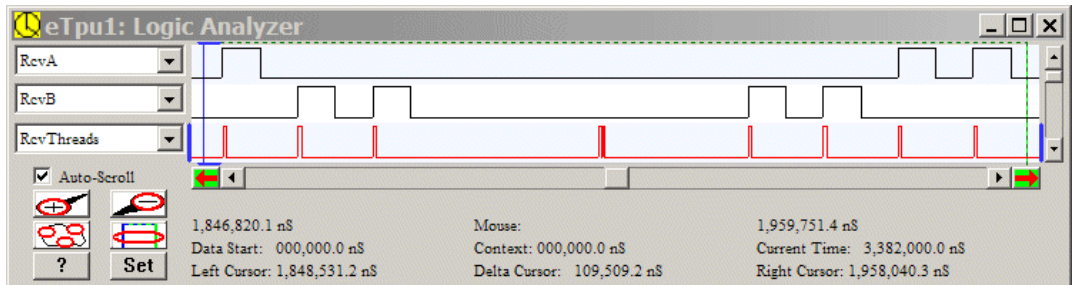
Thread activity can be extremely useful in both understanding and debugging eTPU Simulator functions. The eTPU Simulator provides the following thread activity nodes.

- ThreadsGroupA
- ThreadsGroupB
- ThreadsGroupC
- ThreadsGroupD
- ThreadsGroupE

- ThreadsGroupF
- ThreadsGroupG
- ThreadsGroupH

These nodes can be renamed. In the following example, the ThreadsGroupB node is assigned the name A429RcvThreads. Note that the primary purpose of renaming these nodes is to provide a more intuitive picture in the logic analyzer window, as shown below. See the Logic Analyzer Options Dialog section for more information on how to specify groups for monitoring of TPU and eTPU thread activity.

```
node RcvThreads ThreadsGroupB
```



## 8.2 Group Command

```
group <GROUP_NAME> <NODE 1> [NODE 2] ... [NODE N]
```

The group command assigns multiple nodes, “NODE N” to a group name GROUP\_NAME. These nodes can be referred to later by this group name. The group name may contain any ASCII printable text. These group names are case sensitive (though this is currently not enforced.) Up to 30 nodes may be grouped.

```
define ADDRESS1 ch5
define ADDRESS2 ch7
define DATA ch3
group PORT1 ADDRESS1 ADDRESS2 DATA
```

In this example a group with the name PORT1 is associated with TPU channel pins 5, 7, and 3.

### 8.3 State Command

```
state <STATE_NAME> <BIT_VALUE>
```

The state command assigns a bit value BIT\_VALUE to a user-defined state name STATE\_NAME. State names may contain any ASCII printable text. These state names are case sensitive (though this is currently not enforced.) Bit values must consist of a sequence zeros and ones. The total number of zeros and ones must be between one and 30.

```
state NULL 0110
```

In this example a user-defined state NULL is associated with the bit pattern 0110.

### 8.4 Frequency Command

```
frequency <FREQUENCY>
```

The frequency command sets the master test vector clock to FREQUENCY which is a floating point number whose terms are million cycles per second (MHz). All test vectors are set by this frequency. Since the entire test vector file is loaded at once, if a test vector file contains multiple frequency commands, only the last frequency command is used and all previous frequency commands are ignored.

```
frequency 1
```

In this example the master test vector generation frequency is set to one MHz. This is a convenient test vector frequency because its period of one microsecond makes timing easy to calculate.

### 8.5 Wave Command

The wave command creates and defines a new wave form generator. There is no limit to the number of wave commands that may be used in a test vector file. The number of wave form generators is equal to the number of wave commands found in the test vector file.

```
wave <GROUP> <STATE REPEAT> <(<STATE REPEAT <...>>  
REPEAT> end
```

The wave command causes the nodes of GROUP to be stimulated by the state and repeat count pairs STATE REPEAT. Multiple states and repeat counts are allowed. A special infinite repeat count, \*, generates an infinite repeat count. This command may span

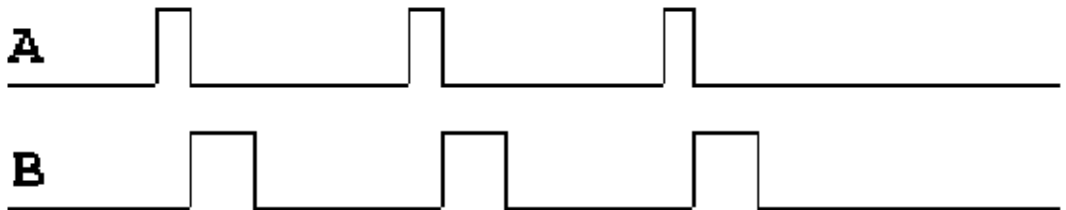
multiple lines. An end statement must terminate the command.

```

wave OUTPUTS
  (OFF 5   DRIVE_A 1   DRIVE_B 2) 3
  OFF *
end

```

The resulting wave form from this example is shown below. The wave form begins with signal A and signal B being off for five master test vector clock periods. Signal A is then driven for one period. Then signal B is driven for two periods. These three states constitute a loop which executes three times. After the loop has executed three times the OFF state is driven forever. Note that the NODE, GROUP, STATE, and FREQUENCY commands are omitted from this example for simplification purposes.



## 8.6 Engine Example

```

// File:      ENGINE.Vector
// AUTHOR:    Andrew M. Klumpp, ASH WARE.
// DATE:      950404
//
// DESCRIPTION:
//   This generates the test vectors associated with a four cylinder
//   car. The four spark plugs fire in the order 1,3,2,4. For convenience
//   an engine frequency is chosen such that one degree corresponds to
//   10 microseconds (10 microseconds will be written as 10us).
//   A test vector frequency is chosen such that one degree corresponds
//   to one time-step.
//   The test vector frequency is the eTPU Simulator's
//   internal test vector timebase.
//   Within each engine revolution two spark plugs fire.

// ASSIGN NAMES TO PINS
//   Assign the descriptive names to the synch and spark plug signals.
node Synch      ch3
node Spark1     ch8
node Spark2     ch11
node Spark3     ch6

```

## 8. Test Vector Files

---

```
node Spark4          ch4

// ASSOCIATE PINS WITH A GROUP

// Make a group named SYNCH with only the SYNCH TPU channel as a member
group SYNCH Synch

// Make a group named SPARKS that consists of the four spark plug signals
group SPARKS Spark4 Spark3 Spark2 Spark1

// DEFINE THE SYNCH STATES
// The synch signal can be either pulsing (1) or waiting (0).
state synch_pulse 1
state synch_wait  0

// DEFINE THE SPARK FIRE STATES
// There are five states:
//   There is one state for each of the four spark plugs firing.
//   There is one state for none of the spark plugs firing.
state FIRE4      1000
state FIRE3      0100
state FIRE2      0010
state FIRE1      0001
state NO_FIRE    0000

// SET THE TEST VECTOR BASE FREQUENCY
//   In order to have a convenient relationship between time and degrees
//   an engine revolution is made to be 3600us such that one degree
//   corresponds to 10us.
// Thus a convenient test vector time-step period of 10us is chosen.
// frequency = 1/period = 1/10us = 0.1MHz
// (Frequency is expressed in MHz; this is a modification of a
//  previous version of the User Manual.)
frequency 0.1

// CREATE/DEFINE THE SYNCH WAVE FORM
// This generates a one degree (10us) pulse every 360 degrees (3600us).
wave SYNCH (synch_pulse 1  synch_wait 364) * end

// CREATE/DEFINE THE SPARK WAVE FORM
// Each spark is equally spaced every 1/2 revolution
// which is 180 degrees (1800us).
// Each spark plug triple fires and each fire lasts one degree (10us).
//
// In addition there is a 17 degree (170us) lag of this wave form
// relative to the synch wave form.
wave SPARKS
    no_fire 17 // This creates a 17 degree lag
```

```

(           // Enclose the following in a bracket to generate a loop/

// The first plug fire cycle lasts five degrees (50us).
// The spark plug fires three times.
fire1 1   no_fire 1   fire1 1   no_fire 1   fire1 1

// The delay between fire cycles is 180 degrees
// less the five degree fire cycle.
// 180-5=175 degrees
no_fire 175

// The third plug fires next.
fire3 1   no_fire 1   fire3 1   no_fire 1   fire3 1

// Give another 175 degree delay.
no_fire 175

// The second plug fires next.
fire2 1   no_fire 1   fire2 1   no_fire 1   fire2 1

// Give another 175 degree delay.
no_fire 175

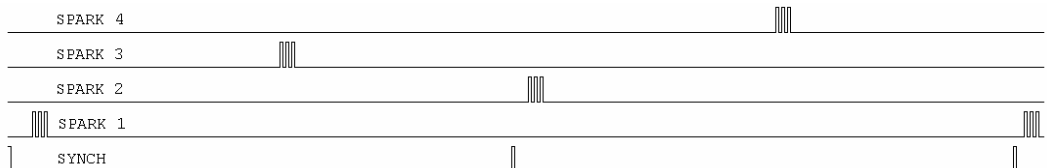
// The fourth plug fires next.
fire4 1   no_fire 1   fire4 1   no_fire 1   fire4 1

// Give another 175 degree delay.
no_fire 175

) * // Enclose the loop and put the infinity character, *.
end

```

The following wave form is generated from the above example.

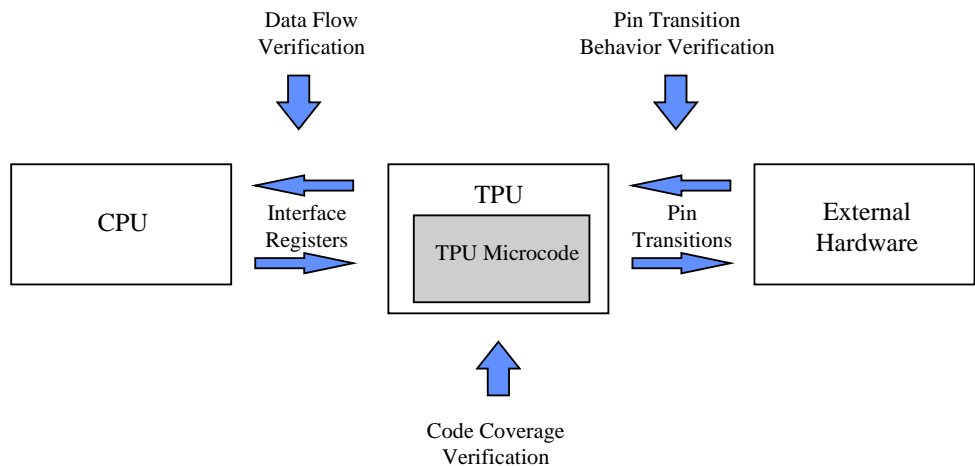




## 9

# Functional Verification

Functional verification supports an automated method for verifying that behavioral requirements are met by the code. These capabilities can be grouped as data flow verification, pin transition behavior verification, and code coverage verification. The following diagram shows a hardware perspective of functional verification. Pin transition verification is applicable only to eTPU and TPU Simulation.



## 9. Functional Verification

---

Data flows between the eTPU and the CPU via interface registers. Data flow verification provides the capability of verifying this data flow.

Pin transitions are generated by the eTPU or by external hardware. Pin transition verification capabilities allow the user to verify this pin transition behavior.

Code coverage provides the capability to determine the thoroughness of a test suite. Code coverage verification allows the user to verify that a test suite thoroughly exercises the microcode.

### A Full Life-Cycle Verification Perspective

The following describes a typical software life-cycle. Initially, a set of requirements is established. Then the code is designed to meet these requirements. Following design, the code is written and debugged. A set of formal tests is then developed to verify that the software meets the requirements. The software is then released.

Now the software enters a maintenance stage. In the maintenance stage, changes must be made to support new features and perhaps to fix bugs. Along with this, the formal tests must be modified and rerun. Then the software must be re-released.

This life-cycle can be described as having three stages: development, verification, and maintenance. All of these three stages are supported. The IDE and GUI are primarily of interest in the initial development phase. Verification involves developing a set of repeatable and automated tests that show that the requirements are being met. In the maintenance stage these previously-developed automated tests are rerun to prove that requirements are still being met when bug fixes and enhancements cause the code base to change.

There is a significant emphasis on automation such that a complete test suite can be run, sometimes spanning hours, and a series of tests results in a single 'pass' or 'fail' result.

## 9.1 Data Flow Verification

Data flow verification is one of the verification capabilities for which an overview is given in the Functional Verification chapter. Data flows between the TPU and the CPU primarily across the Channel Interrupt Service Request (CISR) register and the parameter RAM. The data flow verification capabilities address data flow across these registers.

The eTPU parameter RAM data flow is verified using the `verify_chan_data24(X,Y,Z)` and

verify\_chan\_bits24(X,Y,Z,V) script commands described in the eTPU Parameter RAM Script Commands section.

The data flow across the CISR register is verified using the verify\_intr(X,Y) script command described in the eTPU Interrupt Script Commands Script Commands section.

In the following example data flow across channel 10's CISR and parameter RAM is verified at a simulated time of 100 microseconds and again at 250 microseconds.

```
// Wait for the eTPU Simulator to run 100 micro-seconds.
at_time(100);
// Verify that channel 10's CISR is set.
verify_cisr(0xa,1);
// Verify that channel 10's parameter 2 bit 14 is set.
verify_ram_bit(0xa,2,14,1);
// Verify that channel 10's parameter 3 is 1000 hex.
verify_ram_word(0xa,3,0x1000);
// Verify that channel 10's parameter 5 is 1500 hex.
verify_ram_word(0xa,5,0x1500);
// Clear channel 10's CISR.
clear_cisr(0xa);
// Wait for the eTPU Simulator
// to run an additional 150 microseconds.
wait_time(150);
// Verify that channel 10's CISR is set.
verify_cisr(0xa,1);
// Verify that channel 10's parameter 2 bit 14 is cleared.
verify_ram_bit(0xa,2,14,0);
// Verify that channel 10's parameter 3 is 3000 hex.
verify_ram_word(0xa,3,0x3000);
// Verify that channel 10's parameter 5 is 3500 hex.
verify_ram_word(0xa,5,0x3500);
```

## 9.2 Pin Transition Behavior Verification

Pin transition behavior verification is one of the verification capabilities for which an overview is given in the Functional Verification chapter. Pin transition behavior verification capabilities include the ability to save recorded pin transition behavior to enhanced pin transition behavior (.ebv) files, the ability to load saved pin transition behavior files into the eTPU Simulator, and the ability to verify that the most current pin transition behavior matches the saved behavior. Old-style behavior verification files (.bv) can still be read and used for verification, but can no longer be created. All pin transition behavior verification

## 9. Functional Verification

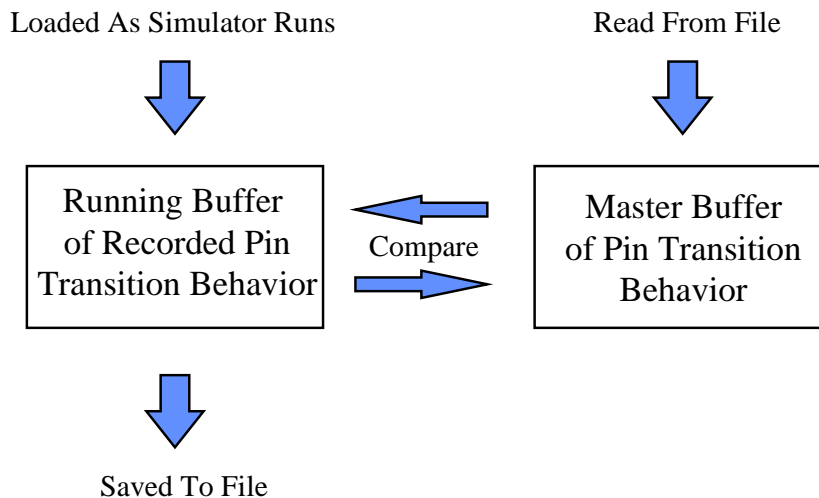
---

must now be managed via scripting; GUI menu options are no longer available.

From a behavioral model perspective these capabilities correlate to the ability to create behavioral models and the ability to compare source microcode against these behavioral models. The emphasis in the eTPU Simulator is on the automation of these modeling capabilities through script commands, as the capabilities are no longer available from the menus.

### Pin Transition Buffers

There are two pin transition storage buffers in the eTPU Simulator as shown in the following diagram.



The running buffer is filled as the eTPU Simulator runs. Whenever a pin transition occurs, information regarding this transition is stored in this buffer, which is also used to draw the signals in the logic analyzer window. This pin transition information can be selectively recorded to file using the `create_ebehavior_file("filename.ebv")`, `add_ebehavior_pin("<pin name>")` and `close_ebehavior_file()` script commands; also see Enhanced Pin Transition Behavior Verification below.

### Deprecated Pin Transition Behavior Verification

Legacy .bv pin transition data files can still be used by the new enhanced behavior verification; .bv files can no longer be generated. The master buffer can be loaded only with pin transition behavior data from a previously saved file. This file forms a behavioral model of the source microcode. Changes can be made in the source microcode and the changed microcode can be verified against these behavioral models. This file is loaded using the `read_behavior_file("filename.bv")` script command.

There are two options for verifying pin transition behavior against the previously-generated behavioral model. The first option is to continuously check the running pin transition behavior buffer against the master pin transition behavior buffer. This is selected by the `enable_continuous_behavior()` script command that follow a behavior file load. The second option is to perform a complete check of the running buffer against the master buffer all at once. This is selected using the `verify_all_behavior()` script command at the end of a simulation run. Time tolerances for pin transitions default to 2 system clocks, but can be adjusted using the new enhanced behavior verification `set_ebehavior_pin_tolerance()` and `set_ebehavior_global_tolerance()` script commands.

A count of failures is displayed in the Configuration window. This count is incremented whenever a behavior verification failure occurs. By default, each failure will generate an error dialog, but this can be disabled by de-selecting Options -> Messages... -> Behavior verification failure.

There are several considerations regarding these behavioral models. The first is buffer size. The maximum behavioral buffer size is currently set at 100,000 records, though this may increase in future releases. If the number of recorded pin transitions equals or exceeds this buffer size then the buffer rolls over and verification against this buffer is not possible - the buffer should be re-sized to hold the master file data.

The second consideration is TCR2 pin recording. Normally TCR2 pin transitions are not written in these buffers. This is because the recording of TCR2 pin transitions very quickly fills up the buffer and causes the buffer to quickly roll over. When the buffer rolls over verification is not possible with legacy .bv files.

### **Enhanced Pin Transition Behavior Verification**

The new enhanced pin transition behavior verification makes use of a user-friendly data file format and provides more options on how master file pin transitions are compared to transitions in the current simulation. With the original behavior verification capability, anything other than an exact match, clock cycle for clock cycle, was considered a failure. The reality is that often times even very small code changes can result in a signal being shifted by a clock cycle or two, which previously would result in failure. Enhanced

## 9. Functional Verification

---

behavior verification provides options for controlling the tolerance used in comparisons, so that small changes that are functionally correct can pass verification. Additionally, master file pin transition data can now be viewed in the Logic Analyzer Window, making it easier to track down problems when a failure does occur.

The first step in making use of pin transition behavior verification is to create a master file, also called a .ebv file for its default extension. These enhanced behavior verification files are text and in comma-separated value (CSV, or .csv) format. All behavior verification is controlled from the scripting environment - to create a master file use the `create_ebehavior_file("filename.ebv")` script command. It must execute at simulation time zero, but after any vector file load or pin buffer placements.

```
// save off all pins
create_ebehavior_file("Engine.EBV");
```

By default all pins are saved to the .ebv file. On the TPU, this is all 16 channel pins and the TCR2 pin. On the eTPU, this is all 32 channel input pins, all 32 output pins, and the TCRCLK pin. In many cases, only a small subset of pins are of interest. The `add_ebehavior_pin("<pin name>")` script command allows the user to save data only on those pins of interest. The pin name comes from any node naming in the vector file, or is the underlying default pin name (e.g. "ch3" is the channel 3 pin on the TPU, or "\_ch5.out" is the channel 5 output pin on the eTPU).

```
// save off only the fuel injection output signals
create_ebehavior_file("Engine_Injection.EBV");
// add_ebehavior_pin commands must immediately follow the
create command
add_ebehavior_pin("Injector1");
add_ebehavior_pin("Injector2");
add_ebehavior_pin("Injector3");
add_ebehavior_pin("Injector4");
```

After the creation setup commands are complete, and the script file exercises the software, the .ebv file needs to be closed and any unsaved data flushed out. Note that data may be saved in several steps as the simulation runs - enhanced behavior verification is not subject to buffer size limitations due to its continuous file management. This is done with the `close_ebehavior_file()` script command.

```
close_ebehavior_file(); // close EBV being saved
```

As mentioned, the .ebv file is in comma-separated value format. The first line contains the column header information, which is time units for the first column, followed by pin names of all saved pin data. After that, each line contains a simulation time value in microseconds followed by the value of each pin. Below is an example of the first few lines of an .ebv file which contains data on just two pins.

```

TIME (US), PWM_A, PWM_B,
000000000000.000000, 0, 1,
000000000019.980000, 1, 1,
000000000039.980000, 0, 1,
000000000049.980000, 0, 0,

```

The comma-separated value allows the file to be easily parsed by other existing tools, such as spreadsheet applications.

Once a master .ebv file has been created, it can be used to verify pin behavior stays within an expected tolerance range on subsequent simulation runs. Typically this is done after a software change has been made that is not supposed to affect pin transition behavior. With enhanced behavior verification, only a continuous verification model is supported. As with .ebv file creation, all control is via script commands. First, which .ebv file to be used for the verification run must be specified - this is done with `run_ebehavior_file("filename.ebv")`. By default, all pins found in the .ebv file are verified with a default transition error tolerance equivalent to two system clocks.

```

// test all pins
run_ebehavior_file("PWM_gold.EBV");

```

The above command should be issued at simulation time zero, at the same place in the script file as the .ebv file was generated with the `create_ebehavior_file()` command. The simulation should run, and at the end, at the same time as when the .ebv file was saved and closed, the verification should be stopped with the `stop_ebehavior_file()` script command.

```

stop_ebehavior_file(); // stop EBV verification

```

Tolerances can be adjusted, and the pins being verified can be specified, using the enhanced behavior verification pin tolerance script commands. With the `set_ebehavior_global_tolerance()` script command, all pins under test can have their allowed error tolerance set.

```

run_ebehavior_file("Engine_gold.EBV");
// verify all pins in the .ebv file to a 2us tolerance
set_ebehavior_global_tolerance(EBV_ABSOLUTE, 0.0, 2.0);

```

The `set_ebehavior_pin_tolerance()` script command serves a dual purpose. Once one of these commands is specified, only pins specified with `set_ebehavior_pin_tolerance()` script commands will be verified.

```

run_ebehavior_file("PWM_gold.EBV");
// test only PWM_A and PWM_C pins w/ appropriate tolerance
set_ebehavior_pin_tolerance("PWM_A", EBV_ABSOLUTE, 0.0,
1.1);
set_ebehavior_pin_tolerance("PWM_C", EBV_ABSOLUTE, 0.0,
1.1);

```

## 9. Functional Verification

---

Although the tolerances can be adjusted at any time during simulation, if using `set_ebehavior_pin_tolerance()` to configure a subset of pins to test, these commands should directly follow the `run_ebehavior_file()` command. Tolerances can then be adjusted later, either by individual pin or globally.

Currently there is one tolerance type - "EBV\_ABSOLUTE". This means the absolute time of each transition in the master file is compared to the absolute time of the matching transition in the current simulation run. In the comparison process, first the offset is applied to the master file transition time, and then the difference between the two times is calculated. If the absolute value of this result is less than the configured tolerance, the behavior is considered valid and simulation continues. If it is greater than the tolerance, a behavior error is thrown - this may or may not trigger a pop-up error dialog depending upon the IDE messages configuration.

With enhanced behavior verification, an `.ebv` file can be created while simultaneously using another one for verification. This could be handy in the sense that after running a simulation session the user has a new current pin transition file - it could be used as an input into other tools, or if verification is successful, it could be copied as the new "gold", or master file.

```
// create and verify simultaneously
create_ebehavior_file("Powertrain_current.EBV");
run_ebehavior_file("Powertrain_gold.EBV");

// ... simulation ...

// finish behavior verification
close_ebehavior_file(); // close EBV being saved
stop_ebehavior_file(); // stop EBV verification
```

When enhanced behavior verification is used with a dual-eTPU simulation model, an `.ebv` file and the associated commands apply to a single eTPU engine. In other words, if pin transition behavior verification is to be done on each eTPU, there must be a separate `.ebv` file for each.







### 9.3 Code Coverage Analysis

Code coverage analysis is one of the verification capabilities for which an overview is given in the Functional Verification chapter.

There are two aspects to code coverage. The first aspect is the code coverage visual interface while the second aspect is the coverage verification commands.

## Code Coverage Visual Interface

The visual interface is enabled and disabled using within the IDE Options dialog box. When this is enabled, black boxes appear as the first character of each source code line that is associated with a microinstruction. As the code executes, and the instruction coverage changes, these black boxes change to reflect the change in the coverage. This is summarized below.

-  A black box indicates a 'C' source line or assembly instruction that has not been executed.
-  An orange box indicates 'C' source line that has been partially executed.
-  A blue box indicates an assembly branch instruction in which neither branch path has been traversed.
-  A green box indicates an assembly branch instruction where the branch path has been traversed.
-  A red box indicates an assembly branch instruction where the non-branch path has been traversed.
-  A white box indicates a 'C' source line or assembly instruction that has been executed.

## Code Coverage Verification Commands

The code coverage verification commands, described in the Code Coverage Script Commands section provide the capability to verify both instruction and branch coverage percentages on both an individual file basis and a complete microcode build. If the required coverage has not been achieved then a verification error message is generated and the count of script failures found in the configuration window is incremented.

## Code Coverage Report Files

Code coverage report files can be generated using the `write_coverage_file("filename.Coverage")` script command described in the Code Coverage Script Commands section. Code coverage report files can also be generated directly from the File menu by selecting the Coverage submenu. This is described in the Files Menu section.

The top of the code coverage report file contains a title line, a copyright declaration line, and a time stamp line.

Following this generic information is a series of sections. The first section provides coverage data on the entire microcode build. Succeeding sections provide coverage

## 9. Functional Verification

---

information on each file used to create the microcode build.

Each section contains a number of lines that provide the following information. The instruction line lists the total number of instructions. Both regular and branch instructions are counted, but entry table information is not. The instruction hits line lists the number of instructions that have been fully covered. The instruction coverage percent line lists the percent of instructions that have been covered. The branches line lists the total number of branch paths. This is always an even number because for each branch instruction there are two possible paths (branch taken and branch not taken.) If the branch path has been traversed then this counts as a single branch hit. Conversely if the non-branch path has been traversed then this also counts as a single branch hit. The branch instruction is considered fully covered when both the branch-path and the non-branch-path have been traversed. The branch coverage percent line contains the percentage of branch paths that have been traversed.

Flushed instructions for which a NOP has been executed are not counted as having been covered.

An example of such a file follows.

```
//// Code coverage analysis file.
//// Copyright 1996-2012 ASH WARE Inc.
//// Sun June 02 09:30:30 1996

Total
  Instructions:          135
  Instruction Hits:      57
  Instruction Coverage Percent:  42.2
  Branches:             60
  Branch Hits:          16
  Branch Coverage Percent: 26.7
MAKE.ASC
  Instructions:          2
  Instruction Hits:      0
  Instruction Coverage Percent:  0.0
  Branches:             0
  Branch Hits:          0
  Branch Coverage Percent: 100.0
toggle.UC
  Instructions:          5
  Instruction Hits:      5
  Instruction Coverage Percent: 100.0
  Branches:             0
```

```
Branch Hits:          0
Branch Coverage Percent: 100.0
pwm.UC
Instructions:         24
Instruction Hits:      14
Instruction Coverage Percent: 58.3
Branches:             12
Branch Hits:          3
Branch Coverage Percent: 25.0
linkchan.uc
Instructions:         8
Instruction Hits:      0
Instruction Coverage Percent: 0.0
Branches:             0
Branch Hits:          0
Branch Coverage Percent: 100.0
uart.UC
Instructions:         58
Instruction Hits:      38
Instruction Coverage Percent: 65.5
Branches:             30
Branch Hits:          13
Branch Coverage Percent: 43.3
```

## 9.4 Regression Testing (Automation)

Regression Testing supports the ability to launch the eTPU Simulator from a DOS command line shell. Command line parameters allow specific tests to be run. From the command line the project file that is run and the primary script file(s) that are loaded into each target are specified. Command line parameters also are used to specify that the target system automatically start running with no user intervention, and to accept the license agreement thereby bypassing the dialog box that would otherwise open up and require user intervention. A script command is used to terminate the eTPU Simulator once all the tests have been run.

Upon termination, the eTPU Simulator sets the error level to zero if no verification tests failed, and otherwise error level is set to be non zero. This error level is the application's termination code and can be queried within a batch file running under the operating system's DOS shell. By launching the eTPU Simulator multiple times, each time with a different set of tests specified, and by checking the error level each time the eTPU Simulator terminates, multiple tests can be run automatically and a single pass (meaning all

## 9. Functional Verification

---

tests passed) or fail (meaning one or more tests failed) result can be determined.

Note that this only works in operating systems that support access to exit codes from a batch file. **Windows 98 does not support this.** True operating systems such as Windows XP Professional, Windows 2000 Professional, and Windows NT 4.0. do support automation.

### 9.5 Console Mode

A console mode version of the eTPU Simulator supports regression testing without the GUI. The console mode version of the executable contains the '\_CL' suffix on the file name.

### 9.6 Command Line Options

When launching the eTPU Simulator the last-loaded project file is automatically loaded and the project file contains most of the settings such as the name of the script file to load, which messages to suppress, etc. However, it can be useful to specify the project file to load or even to override settings contained in the project file. This is by specifying settings at the command line when launching the eTPU Simulator. This is especially useful when building regression tests.

Setting	Option	Example
Display Help Prints a list of all the Command Line Parameters	-h	-h
Project File Loads the specified project file	-p<ProjectFileName>	-pTest.ETpuSimProject Loads project file "Test. ETpuSimProject"
Script File Loads the specified script	-s<TargetName>	-sRarChunkTest.Cmd Loads script file

Setting	Option	Example
file (single target).		"RarChunkTest.Cmd"
Script File Loads the specified script file into the specified target (multiple targets.)	-s<Targ>@ @<Script>	-sMyScript.Cmd@ @Cpu32 Loads script "MyScript.Cmd" into the CPU32 simulation model named "Cpu32"
Define as true In script commands file, #define DefinedText 1	-d<DefinedText>	-dCODE_BASELINE Passes the #define CODE_BASELINE 1 to the script file
Define as value In script commands file, #define DefTxt Val	-d<DefTxt>=<Val>	-dMY_VAR=55 Passes the #define MY_VAR 55 to the script file
Build Define In MtDt build script, define the DefinedText as true	-bd<DefinedText>	-bdMPC5674_2 Defines MPC5674 as 'true' in the build define such that the MPC5674 rev 2 model is loaded.
Log File Logs messages to end of file, "FileName.log"	-lf5<FileName.log>	-lf5Error.log Logs messages to file "Error.log"
Test Name Used in conjunction with the log file to append a test result to the end of the log file.	-tn=<TestName>	"-tnPulse Width Test" Appends 'Pulse Width Test Passes' (or 'fails') to the end of the log file.

## 9. Functional Verification

---

Setting	Option	Example
Suppress Warning #1 Suppress the “Source Code Missing” warning	-ws1	-ws1
No Popup Dialogs Disables display of dialog boxes	-Quiet	-Quiet
Accept License Skips the startup license agreement dialog box	-IAcceptLicense	-IAcceptLicense
Automatically Run Sets the target system to running when the eTPU Simulator is launched	-AutoRun	-AutoRun
Network Retry Time Time to wait (in seconds) for a network license if none is available	-NetworkRetry=<<Sec>	-NetworkRetry=30 Retries for 30 seconds if unable to make a network connection

### 9.6.1 Using the –d (define) Option and Escape Characters

There are issues with passing a quote character into the simulator in Windows because Windows uses the quote character to bunch multiple pieces of text into a single command line parameter. Consider the case where a filename is to be passed into the eTPU simulator. A good way of doing this is to define a string, then use the `load_executable()`; script command as follows.

```
#define CODE_TO_LOAD "MyCode.Cod"  
load_executable( CODE_TO_LOAD );
```

It is possible to pass the `CODE_TO_LOAD` #define into the simulator from the command line (instead of having it in the script commands file) using the following command line

parameter.

```
"-dCODE_TO_LOAD=\ "MyCode.Cod\ " "
```

There are four quote characters in the command line parameter shown above. The first and last quote characters are used by Windows to bunch everything between them together into a single command line parameter which would (for example) allow spaces to appear within the single parameter.

But the filename, MyCode.Cod, is a string, and strings must be surrounded by quote characters in the script commands file. This is accomplished by preceding the quote character with the backslash character. Windows interprets this backslash-quote combination literally and the quote character is thereby passed along with the filename into the simulator.

## 9.7 File Location Considerations

Although this discussion is equally applicable to the MtDt as a whole it is important to point out how MtDt locates files within the context of Regression Testing.

MtDt uses a "project file relative" approach to searching and finding almost all files. This means that the user should generally locate the project files near the source code and script files within the directory structure. Consider the following directory structure.

```
C:\BaseDir\SubDirA\Test.Sim32Project  
C:\BaseDir\SubDirB\Test.Cpu32Command
```

To load the script command file, Test.Cpu32Command, the following option could be used

```
-s..\SubDirB\Test.Cpu32Command
```

By employing this "project file relative" approach the testing environment can be moved around without having to modify the tests and therefore the files names can be smaller and easier to use.

Note that this does NOT apply to the log file. The log file is written to the "current working directory" which is to say, in the directory from which the tests are launched. This exception to the normal directory locations is used so that multiple project files can exist in different sub-directories, and the test results can all be logged to the same log file.

### 9.8 Test Termination

Termination of MtDt and the passing of the test results to the command line batch file is a key element of Regression Testing. At the conclusion of a script file, MtDt can be shut down using the exit script command, as described in the System Commands section. This command causes the application's termination error level to be set to be non-zero if any verification tests failed, or zero if all the tests passed.

The overall strategy in ensuring that a zero error level truly represents that all tests have run error free and to completion is to treat any unusual situation as a failure. Specifically, a failing non-zero termination code will result unless the following set of conditions has been met.

- No verification tests are allowed to fail in any target.
- All targets must have executed all their script commands.
- MtDt must terminate through the exit(); script command. Abnormal termination such as detection of a fatal internal diagnostic error results in a non-zero error level.

### 9.9 Cumulative Logged Regression Testing

Cumulative logged regression testing supports the ability to run an entire test suite without user intervention, even if one or more of the tests fail. This capability overcomes the problem in which a failure halts the entire test suite until acknowledged by the user. Using this capability, the alert is logged to a file rather than being displayed in a dialog box.

Test completion occurs when the exit() script command is encountered. At this time, a PASS or FAIL indicator is appended to the end of the log file. Because it is appended to the end of the log file, the normal usage would be to delete the log file prior to beginning the test suite. Then, upon completion of a test run, the log file grows. At the end of the test suite, the log file can be perused, and any failing tests are quickly identified.

Note that only certain types of failures bypass the normal message dialog box. For instance, if the failure log file itself cannot be written, then this generates a failure message in a dialog box which must be manually acknowledged.

This capability is invoked using a combination of two Command Line Parameters, shown below.

```
-LF5MyLogFile.log -Quiet
```

The first command, **-LF5MyLogFile.log** specifies that message are appended to the end of a log file named, "MyLogFile.log."

The second command, **-Quiet**, specifies that the dialog boxes that normally carry test errors or warnings are not displayed. Note that this command only works in conjunction with **-AutoRun**, **-IAcceptLicense**, and **-LF5<LogFileName.log.>** If any of these options is not selected, then this **-Quiet** command is ignored. Note also that if the user halts the simulation, then this option is disabled such that messages shown in dialog boxes will require manual acknowledgement.

It is convenient to name each test run. That is, when MtDt is launched, the command line parameter shown below applies a name the test run. This name shows up in the log file. This allows the particular test run that is causing any failures to be easily identified when perusing the log file.

```
-tn<TestName>
```

Note that command line parameters do not handle spaces will. To include spaces in the name, enclose the parameter in quotes, as shown below. In the following example, the name, "Angle Mode" is specified.

```
"-tnAngle Mode"
```

## 9.10 Regression Test Example

The keys to successful Regression Testing is the ability to launch MtDt multiple times within a batch file that runs in a DOS command line shell and to verify within this batch file that the tests that were automatically run had no errors. These multiple launches of MtDt, and the tests contained therein, form a test suite.

The following is a batch file used to launch the TPU Simulator multiple times. Note that there is only a single target such that no target must be specified on the command line. Had this been a test running in a multiple-target environment, the target name would have to be specified along with each script file.

```
echo off

// Use the CONSOLE version of the simulator
set exe="C:\Program Files\ASH WARE\TPU Simulator\TpuSimulator_CL.exe"

%exe% -pTest1.TpuSimProject -sTest1.TpuCommand -AutoRun -IAcceptLicense
if %ERRORLEVEL% NEQ 0 ( goto errors )

%exe% -pTest2.TpuSimProject -sTest2.TpuCommand -AutoRun -IAcceptLicense
```

## 9. Functional Verification

---

```
if %ERRORLEVEL% NEQ 0 ( goto errors )

echo *****
echo          SUCCESS, ALL TESTS PASS

echo *****
goto end

:errors
echo *****
echo          YIKES, WE GOT ERRORS!!
echo *****
:end
```

If the above test were named TestAll.bat then the test would be run by opening a DOS shell and typing the following command

```
C:\TestDir\TestAll
```

# 10

## Action Tags

An action tag is an identifier embedded in the source code as a comment that alerts MtDt to perform a specified action when code execution has reached that point in the source code. The action tag is “@ASH@”. When target code is loaded into MtDt, the application scans the source code for action tags – thus if MtDt cannot locate the source code the associated action tags will not get activated.

The full form of an action tag includes the action – “@ASH@<action>”. Code execution momentarily pauses when the associated source code is reached, and the requested action is performed by MtDt. Simulation then re-starts as if nothing happened.

Currently only a few action tag commands are available: print action, timer action and write value actions.

Action tags that are embedded in source code are associated with the underlying executable code as follows. The search for executable code begins at the line that contains the action tag and moves downwards in the source code file. If no executable code is found at or below the line where the action tag appears, then the search continues upwards. If there is no executable code associated with the source code file at all, then the action tag fails.

See <http://www.etpu.org> for an example application.

### 10.1 Print Action Tag

The Print action command is similar to the “C” languages printf() function, and has essentially the same syntax. The resulting text appears as a line in the Trace window. When coupled with the @ASH@ action tag in the source code file looks similar to the following.

```
// @ASH@print_to_trace("action tag test 1");  
/* @ASH@print_to_trace("action tag test 2"); */  
// @ASH@print_to_trace("variable A = %d\n", A);
```

As with the “C” printf, the first argument is the format string. The ASH WARE Print command uses the same syntax for the format string, which can then be followed by a varying number of arguments. The Print command checks that the number of conversion characters in the format string matches the number of parameters that follow the format string and issues an error if there is a mismatch. The parameters can be constants or simple expressions (variables). With code compiled using tools that support more advanced debugging information, simple expressions such as struct.member, structPtr->member, array[2], and \*pointer are supported.

The output of the Print command always goes to the Trace window, and using the start\_trace\_stream(); script command it can also be directed to a file. Directing it to a file can be extremely useful for verification and automated testing.

See the Global eTPU Channel variableAccess section for information on accessing channel variables using the format shown below.

```
@<chan num/name>.<function var name>
```

The format specifier, %, is used to denote that a parameter value is to be inserted in the resulting text. The % character must always be followed by a valid conversion character such as %d. If the % character is not followed by a conversion character then a warning message is generated and any automated tests will fail. The % character is generated by two consecutive % characters.

### 10.2 Timer Action Commands

The timer action commands provide a method for instrumenting source code to verify that time critical paths are being met.

```
// @ASH@timer_start("Test 1");  
// @ASH@timer_stop("Test 1");
```

The passed parameter is the test name and can contain any text with the restriction that

each test must have both a `timer_start` and a `timer_stop` action command. In other words, timer action commands must come in pairs such that each named test has both a start and a stop. Additionally, only one start and one stop is permitted for each test.

In order for a timing measurement to be considered valid, the following must occur. The code containing the start tag must be first and the code containing the `stop_tag` must be traversed next. In other words, the traversal must occur in pairs of start/stop, start/stop, etc. If this order is broken (a stop before a start, two starts in a row, or two stops in a row) then the action timer is invalidated and any verification scripts that thereafter test the timers will result in verification failures.

The test tag is case sensitive such that the following tag,

```
// @ASH@timer_start("TestTag");
```

is a different test from the following tag

```
// @ASH@timer_start("TESTTAG");
```

On both target-reset and on code-reload, all timing measurements are reset.

### Related Information

- Naming timing regions in source code
- Verifying traversal times a script command file
- View named timing regions timing using the Watch Window
- List named timing regions in the Insert Watch Dialog Box

## 10.3 Write Value Action Tag

When the source code containing a write value action tag is traversed, the specified symbolic write value script command (`write_val`, `write_val_int` or `write_val_fp`) is exercised.

```
// @ASH@write_val("s24", "0x123456");  
/* @ASH@write_val_int("varui32", 0x10101010); */  
// @ASH@write_val_fp("ratio", 0.232323);
```

See the Print Action Tag section for more information on referencing channel frame variables.



# 11

## External Logic Simulation

External logic simulation is eTPU centric since it may be used solely on the I/O pins of eTPU targets and only on an intra-eTPU basis. Future versions of the software will extend this capability to work between any addressable bits in any target's address space.

Boolean logic that is external to the TPU can be incorporated in a simulation. The logic is simulated with a single pass per step (each step is two CPU clocks). Logic is placed via the script commands files using external logic commands.

There are a number of limitations to the Boolean logic.

- There are only two logic states, one and zero.
- The logic is simulated with a single pass per step (each step is two CPU clocks).
- All output states are calculated before they are written, and therefore all calculations are based on the pre-calculated states. Thus it takes multiple passes for state changes to ripple through sequentially connected logic.
- All Boolean logic inputs and outputs must be TPU channel I/O pins.
- Behavior of connected Boolean logic outputs and TPU channel pins configured as outputs is undefined.
- Behavior of pins connected to Boolean logic outputs and also driven by test vectors is undefined.

A typical and appropriate use of Boolean logic in conjunction with test vectors is to connect

## 11. External Logic Simulation

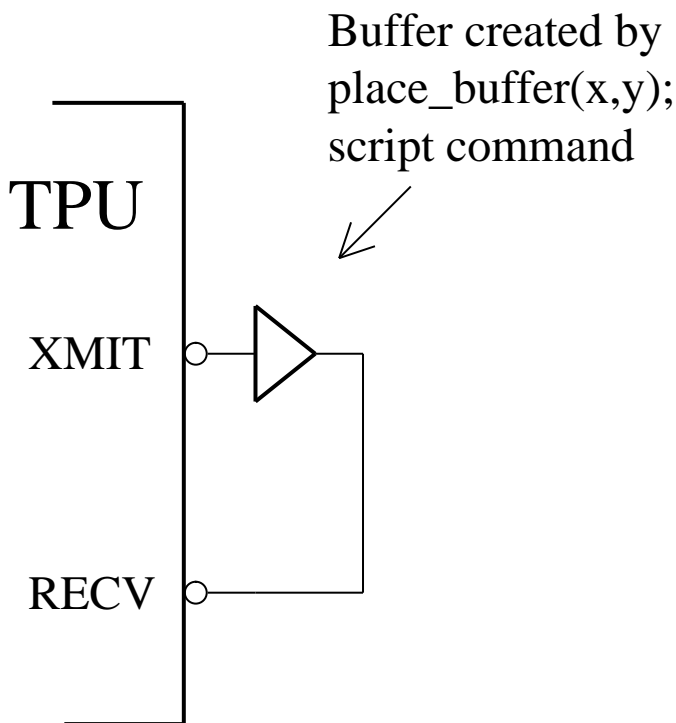
---

a TPU channel pin that is configured as an input to the input of the Boolean logic. This pin is then driven by a test vector file.

### Example 1: Driving the TCR2 Pin

In the following example a TPU channel drives the TCR2 pin. The buffer is instantiated from within a script commands file using the `place_buffer(X,Y)` script command. The buffer's input is connected to the TPU channel's pin and the buffer's output to the TCR2 pin. The X variable is set to five to make TPU channel five the buffer's input. The Y variable is set to 16 as this is the index that MtDt uses for the TCR2 pin.

```
place_buffer(5,16);
```

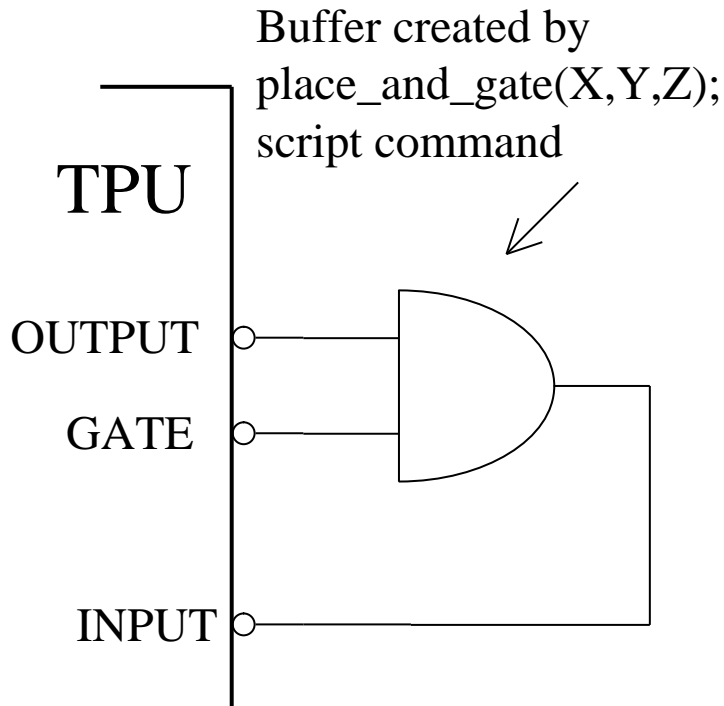


### Example 2: Multi-Drop Communications

In the following multi-drop communications example TPU channel 5 is a communications channel output. TPU channel 7 is used as a gate to enable and disable the output. Channel 1 is the communications input. An idle line is low. An AND gate is instantiated

using the `place_and_gate(X,Y,Z)` script command. The gate pin causes an idle (LOW) state on the input by going low. By going high the gate pin causes the state on the output channel to be driven unto the input channel.

```
place_and_gate(5,7,1);
```





# 12

## Integrated Timers

Integrated timers measure the amount of time that code takes to execute. This capability is available for all simulated targets as well as for hardware targets outfitted with the requisite support hardware.

Integrated timers work with a combination of a special Timers window as well as the source code windows. Specific timer stop and timer start addresses are specified from within the source code windows. Completed timing measurements are viewable within the Timers window.

The timer number within a green circle on the far left side of the source code window identifies the start address. Similarly, the timer number within a red circle on the far left side of the window identifies the stop address. It is possible for the same line of source code to contain multiple timer start and stop addresses. In this case, only a single circle/timer number will be displayed though all timers still continue to function.

MtDt supports 16 timers. Each timer has the following states: armed, started, finished, overrun, and disabled. The state of each timer is displayed in the Timers window. Each timer can be armed in several ways. Specification of a new start or stop address within a source code window automatically arms the timer. The timer can also be armed from within the timer window by clicking on the state field or placing the cursor in the state field and typing the letter 'A'.

When the target hits the start address of an armed timer, the timer state changes to 'started'. When the target hits the stop address the timer state changes to 'finished'.

## 12. Integrated Timers

---

MtDt then computes the amount of time it took to get from the timer start address to the timer stop address, and this information is displayed in both clock ticks and micro-seconds in the timer's window.

An overrun occurs when a timer exceeds the capacity of the timing device. For the simulated timers the limitation is  $1E96$  femto seconds of simulated time. This corresponds to  $1E72$  years of simulated time. This is significantly less ominous than Y2K.

### **Virtual Timers in Hardware Targets**

For targets that do not support the full number of timers that MtDt supports, the concept of virtual timers is introduced. MtDt remembers the start and stop addresses of all 16 virtual timers. The arming of any virtual timer causes a hardware timer to be assigned to that virtual timer. If more virtual timers are armed than are actually supported by the target hardware's timing device then the last-armed virtual timer is automatically disabled.

# 13

## Workshops

Workshops bring order to a chaotic situation. The problem is that with multiple targets, each of which have many windows, the number of windows may become overwhelming. In fact, it may become so overwhelming that without workshops, the Multiple Target Simulator would be nearly unusable.

Workshops allow you to group windows together and view only those windows belonging to that group. Generally it is best to group by target, so that each workshop is associated with a specific target, though this is completely configurable by the user. Some windows, such as watch windows and the logic analyzer windows, are often made visible in multiple workshops. Menus and toolbar buttons allow instantaneous switching between workshops and selection of the active target. The name of the active target is also prominently displayed in the top-right toolbar button.

Closely coupled with workshops is the concept of the active target. It is generally best to associate targets with workshops. Thusly, when the workshop is switched, the active target is also automatically switched to the one associated with the newly activated workshop. The active target is important in that MtDt acts on the active target in a variety of situations. For instance, if the user commands a single step, the active target is the one that gets single stepped and all other targets are treated as slave devices and are stepped however much is required in order to cause the active target's simulation to progress by one step. MtDt makes use of the active target when a new executable code image is loaded. Clearly the user needs to have the ability to select a new executable image into a single specific target. But which target should this be? MtDt automatically selects the active target as the one into which the executable code image is to be loaded.

## 13. Workshops

---

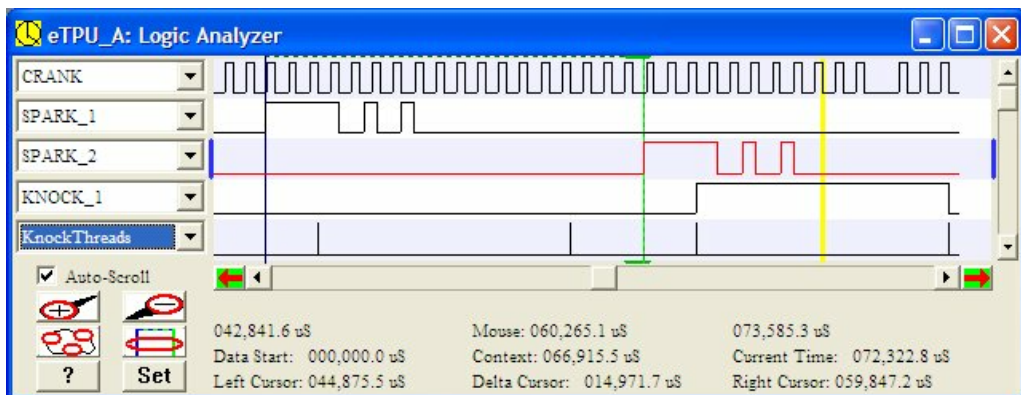
To associate workshops with targets, see the Workshops Options Dialog Box section. In that section there are descriptions of putting workshop buttons on the toolbar, renaming workshops or automatically giving a workshop the same name of its associated target, and associating workshops with targets.

When a target is assigned to a workshop, the windows associated with that target are automatically made visible within the assigned workshop. It is often desirable to override this, either to make individual windows visible in multiple workshops or to remove window from specific targets. See the Occupy Workshop Dialog Box section for a detailed description of how this is done.

# 14



## The Logic Analyzer

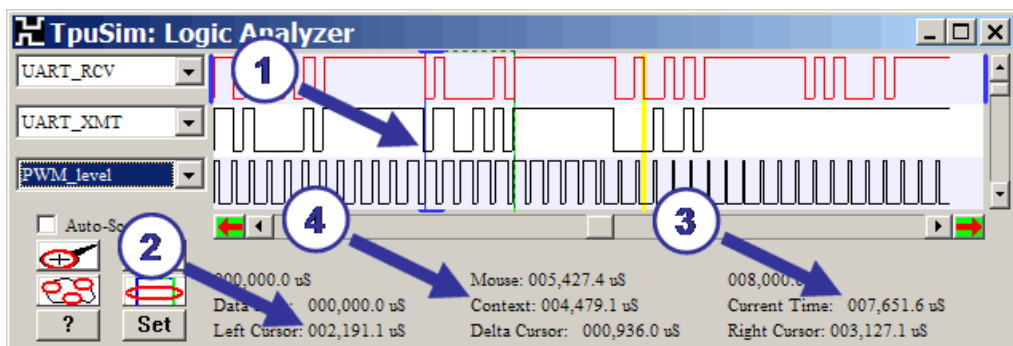
The Logic Analyzer displays node transitions in a graphical format similar to that of a classical logic analyzer. The display can be continuously updated as MtDt runs, or the user can turn off the automatic display update and scroll through previously-recorded transition data.



## 14.1 Executing to a Precise Time

It is informative to consider how to execute the simulator to the precise time of the falling edge indicated by the arrow 1, shown in the picture below.

- Use the keyboards up and down arrows to highlight the UART\_RCV waveform.
- Drag the "Left Cursor" near to the desired edge using the mouse.
- Snap to the actual edge using the left and right arrow keys on your keyboard. In the picture below, this edge is shown to occur at 2,191.1 microseconds.
- Place the cursor over the field showing the time of the left cursor indicated by the second arrow. An open hand  will appear indicating that this "time" can be copied.
- Begin dragging the time by holding the left mouse button down to form a closed hand .
- Drag the time (closed hand) over the waveforms
- Drop the time unto the waveforms by releasing the left mouse button.
- The simulator will reset and execute to precisely 2,191.1 microseconds.
- Verify this observing the time in the "current time" field shown by arrow 3.



A similar method can be used to execute to a "context" time. The context time is shown at arrow 4 above. The "context" time is the time at which some "context" event occurs. For instance, in the trace window, click on any stored event. By clicking on the event, the time at which that event occurred becomes the "context" time. Similarly, in the thread window, click on a worst case thread, and the time at which the first instruction of the worst case

thread occurred becomes the context time. This context time can then be dragged into the current time field causing the simulator to execute to that precise time.

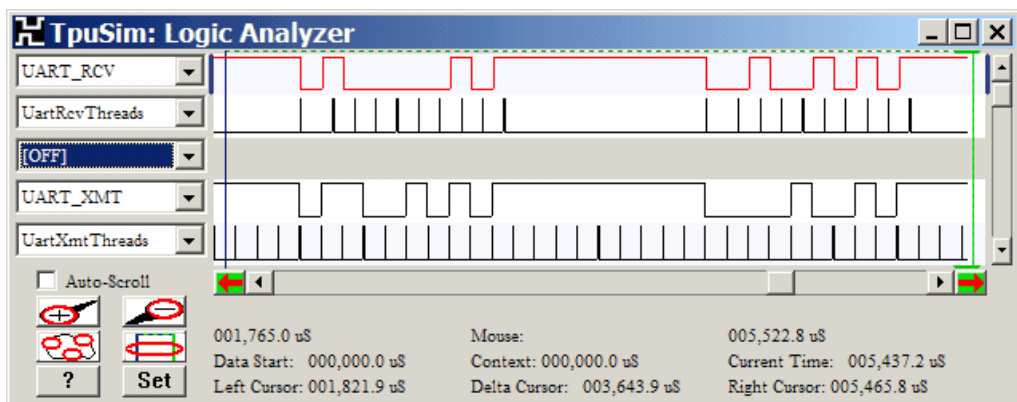
## 14.2 Waveform Selection

The Logic Analyzer displays nodes from the pin transition trace buffer as a waveform. Nodes are added to the display when they are selected from within the drop-down list box located to the left of the waveform display panel. The nodes that may be displayed include the TPU channel I/O pins and the TCR2 pin as well as any user-defined nodes. User-defined nodes are those defined within Test Vector Files.

The waveform display panel may not be large enough to display all the desired nodes. The display can be scrolled vertically using the vertical scroll bar.

### Viewing Thread Activity Nodes

Thread activity can be viewed in the logic analyzer window as shown below. There are eight user-configured thread nodes. Each of these thread nodes can display the thread activity for one or more channels. See the Logic Analyzer Dialog box section for more information on configuring thread nodes.

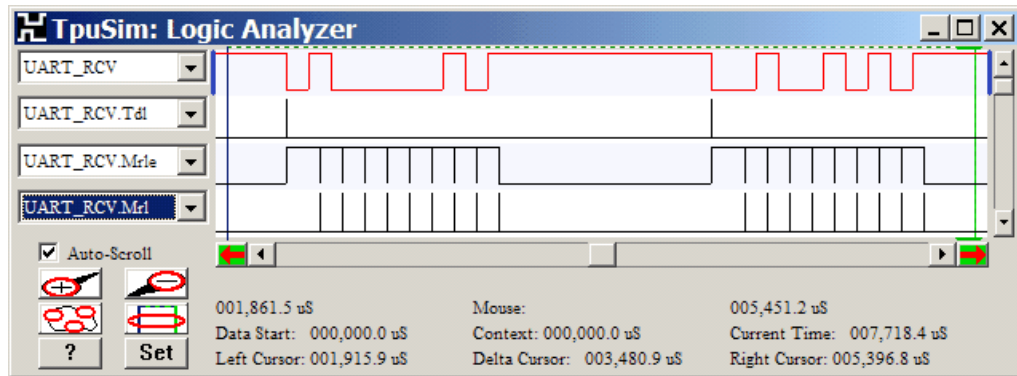


### Viewing Channel Nodes

Several eTPU and TPU channel nodes can be viewed in the logic analyzer as shown below. These include the Host Service Request (HSR,) Link Service Request (LSR,) Match Recognition Latch (output and enable MRL and MRLE,) and Transition Detection Latch (TDL.) Only the nodes from a single channel can be stored and viewed at a time. The channel for which the nodes will be stored is defined in the Logic Analyzer Dialog box

## 14. The Logic Analyzer

---



### 14.3 The Active Waveform

The active waveform is red, whereas all others are black. Use the <up> and <down> arrows on the keyboard to switch the active waveform.

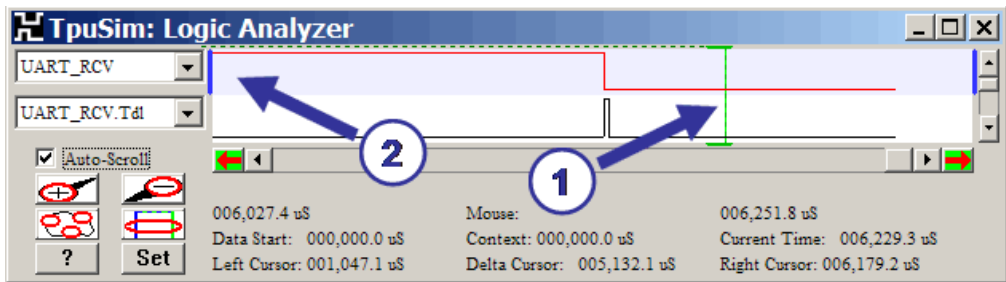
### 14.4 Left and Right Vertical Cursors

The logic analyzer has two cursors; a left cursor and a right cursor. The left cursor is blue whereas the right cursor is green.

One of the two cursors is always "selected" and this selected cursor is displayed as a solid line. Conversely, the unselected cursor is displayed as a dashed line. The concept of an active cursor is important because the active cursor is snapped to edges on the active waveform using the keyboard's left and right arrow keys. To switch between the selected and unselected cursor hold the <CTRL> key while pressing either the left or right arrow keys on the keyboard.

#### Grabbing an Out-of-View Cursor

Occasionally the left or right cursor may be out of view as the left cursor is in the picture seen below. The right cursor is at arrow 1. But the left cursor is out of view because the waveform is only displaying a quarter microsecond of waveform at around six microseconds, but the left cursor is at 1 microsecond. Therefore the left cursor is to the left of the visible area. To bring the left cursor back into view, click the left mouse button at the location indicated by arrow 2.



The simulation can execute to precisely the left or right cursor's time by dragging and dropping the time into the current time, as explained in the, Executing to a Precise Time section.

The cursor can also be moved by dragging and dropping a time into a vertical cursor's time indicator.

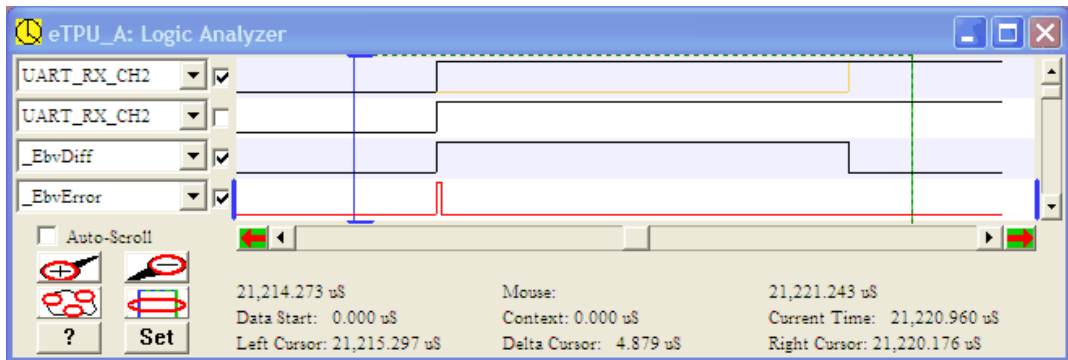
#### Left, Right, and Delta Cursor Time Indicators

The left and right time cursor time indicators show the time associated with the respective vertical cursors. The Delta Cursor indicator shows the difference in time between the left cursor and the right cursor.

These time indicators are capable of extremely precise time measurement with accuracies approaching one femto-second. This is because the cursors can be snapped to the precise pin transition time using your keyboard's left and right arrow keys.

### 14.5 Displaying Behavior Verification Data

Behavior Verification Data appears as a yellow line in the waveform pane, as shown in the top waveform in the illustration found below.



The check box to the left of each waveform is used to enable and disable display of the underlayed behavior verification data. The check box in the second waveform is unselected which is why the behavior verification waveform does not appear. Note that if no behavior verification data is loaded then no yellow waveform will appear, even if the behavior verification checkbox is enabled.

The `_EbvDiff` and `_EbvError` waveforms show where differences and errors in the behavior verification data. These signals are a logical 'or' of all channels errors such that if there is an error or difference in any of the 32 channels then the signal is active (high.)

The reason that `_EbvError` and `_EbvDiff` are not identical is because of the tolerance that is allowed in each of these channels. See the Pin Transition Behavior Script Commands section for a description on setting up global and pin-specific behavior verification tolerances.

### 14.6 Mouse Functionality

In the waveform display panel the mouse has two purposes. It is used to provide a goto time function and also is used to move the left and right cursors.

To get MtDt to execute to a particular point in time, move the mouse to a location corresponding to that time and click the right mouse button. If the selected time is earlier than the current Simulator time MtDt automatically resets before executing. The simulation

runs until the selected time is reached.

The second mouse function is to move the left and right cursors. These cursors are displayed as blue and green vertical lines. The user may move either cursor using the mouse. The times (or CPU clocks) associated with both cursors as well as the delta between cursors are displayed in the cursor time indicators.

Note that the left and right cursors can also be moved using the left and right arrow keys on your keyboard. This causes the cursors to snap to edges on the active waveform.

Cursor display and movement follow these rules:

- Cursors may be located beyond the edge of the wave form display panel.
- If the left cursor gets beyond the right display edge it is automatically moved back into the display.
- If the right cursor gets beyond the left display edge it is automatically moved back into the display.
- Cursors outside the display retain their timing association.
- To allow the user to "pick up" a cursor that is not in the display the cursor is considered to be just outside the display edge for the "drag and drop" operation.

## 14.7 Vertical Yellow Context Time Cursor

The Vertical Yellow Context Time Cursor

The vertical yellow context cursor is the time of some referenced event occurring elsewhere in the IDE. For instance, open a trace window and scroll backward through the trace data. The vertical yellow context cursor will show the time at which the trace window event occurred.

This context capability is a powerful debugging tool. The source code associated with the trace data also pops into view. This allows you to line up previously executed source code, the trace data, and in some cases the call stack data. If you want to re-execute to that context time, use the mouse to drag the time from the context time indicator into the current time indicator, as explained in the, Executing to a Precise Time section.

### Context Time Indicator

The context time indicator displays the time with the IDE-wide context time. For example,

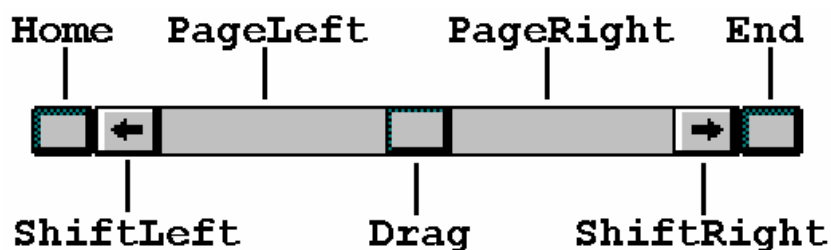
## 14. The Logic Analyzer

---

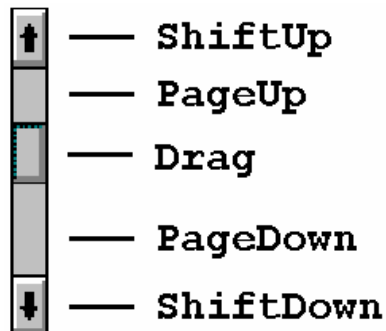
select a past-event in a trace window. The time associated with this event becomes the context time. This context time appears as the vertical yellow context cursor in the waveform display. Drag and drop this time into the current time, as explained in the, Executing to a Precise Time section.

### 14.8 Scroll Bars

The horizontal scroll bar provides the functionality shown in the picture below. Note that this is a standard Windows scroll bar except that it has the additional functionality of the Home and End keys. These keys move the view to the start or end of the simulation.



The vertical scroll bar provides the functionality shown in the picture below. This provides the user with the ability to scroll through the 30 available nodes that the Logic Analyzer supports.



## 14.9 Display Pane Boundary Time Indicators

The left and right time indicators show the time associated with the left-most and right-most visible portions of the display pane.

The display pane can be changed so that a different section of the waveform is in view. For example, use the mouse to drag the current time by depressing the left mouse button over the current time indicator. Holding the left mouse button down, move the mouse over the right time indicator, then release the left mouse button.

## 14.10 Data Storage Buffer Start Indicator

The data start indicator shows the time associated with the very first data in the buffer. This time indicator generally shows zero because the buffer can usually hold all the previously saved pin transition data. But occasionally the buffer overflows so that the very oldest data must be discarded, and in this case the indicator shows the non-zero time associated with the oldest valid data.

## 14.11 Current Time Indicator

The current time indicator is the latest time associated with any target in the system.

Drag and drop any time into this field, as explained in the, Executing to a Precise Time section.

## 14.12 Auto Scroll

Selecting auto-scroll causes the Logic Analyzer to continuously update the view as the simulation runs. De-selection causes the Logic Analyzer to cease updating the view as the simulation runs.

## 14.13 Button Controls

**Zoom In** 

Selecting the Zoom In button narrows the view to the transition data at the center of the

## 14. The Logic Analyzer

---

display.

**Zoom Out** 

Selecting the Zoom Out button widens the view so that more transition data is displayed.

**Zoom Back** 

Selecting the Zoom Back button restores the previous view. The last five views are always saved. MtDt stores the view information in a circular buffer so that if the Zoom Back button is selected five times the original view is restored. Note that only the view and not the cursors are affected. This allows the user to shift views while retaining the timing information associated with the vertical cursors.

**Zoom To Cursors** 

Selecting the Zoom To Cursors button causes the view to display the transition data between the left and right cursors.

**Setup** 

Selecting the Setup button causes the Logic Analyzer Options dialog box to be opened. This dialog box accesses various setup options as described in the Logic Analyzer Options Dialog Box section.

**Help** 

Selecting the Help button accesses the Logic Analyzer section of the on-line help.

### 14.14 Timing Display

Below the Logic Analyzer's horizontal scroll bar are two fields that display the time (or CPU clock counts) corresponding to the left-hand and right-hand sides of the wave form display panel.

The display region can be modified using the horizontal scroller located below the wave form view panel. It can also be modified using the buttons described in the Button Controls section found earlier in the description of the Logic Analyzer.

Below the left side of the wave form display panel are the three fields that display the time (or CPU clock counts) associated with the left cursor, right cursor, and delta cursor.

"Delta" refers to the time (or CPU clock counts) difference between the left and right cursors.

Below the right side of the wave form display are three fields that display the time (or CPU clock counts) associated with the current Simulator time, the oldest available transition data, and the mouse. The mouse field is visible only when the window's cursor is within the wave form display panel.

### 14.15 Data Storage Buffer

As MtDt runs, transition information is continuously stored in a data storage buffer. Data is stored only when there is an actual transition on a node. When no transitions occur, no buffer space is used. This is an important consideration since the user can significantly increase the effective data storage by disabling the logging of the TCR2 input pin (assuming it is active). This is disabled from within the Logic Analyzer Options dialog box.

All buffer data is retained as long as it predates the current Simulator time and there is enough room to store it. Therefore, if MtDt time is reset to zero, all buffer data is lost. When the amount of data reaches the size of the buffer (i.e., the buffer becomes full), new data overwrites the oldest data. In this fashion, the buffer always contains continuous transition data starting from the current time and going backward.



# 15

## Operational Status Windows

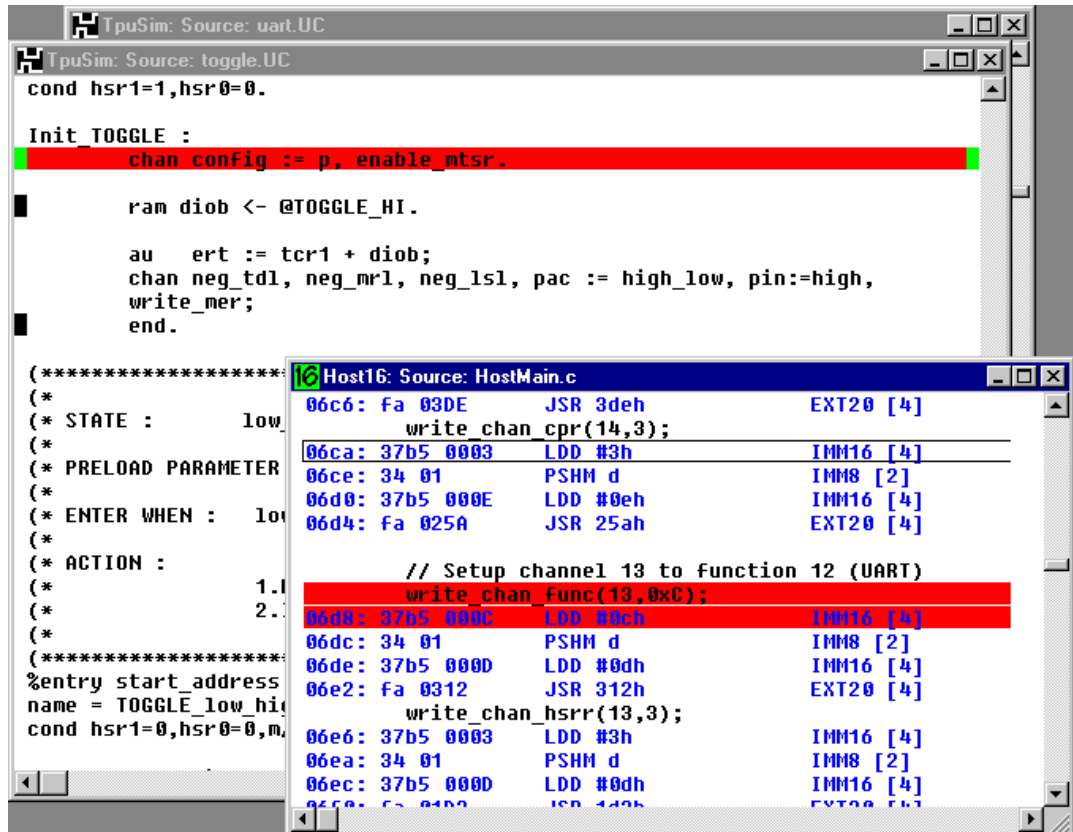
The target state is displayed in various operational status windows. These windows correspond to the various functional blocks associated with the specific target. Each of these windows can be re-sized, scrolled, iconized, minimized, and maximized. Multiple instances of each window may be opened.

### 15.1 Generic Windows

The following windows display generic information.

## 15. Operational Status Windows

### 15.1.1 Source Code Windows



These windows display the executable source code that is currently loaded in the target. Each window displays one source file.

The executable source code is loaded into the target's memory via the Files menu. To load the executable source code, select the Executable, Open submenu and follow the instructions of the Load Executable dialog box.

As the executable source code is executed the source line corresponding to the active instruction is highlighted.

Usually the source file is too large to be displayed in its entirety. The user can use the scroll bars to view different sections of the file. As the code is executed the source code line corresponding to the active instruction appears highlighted in the window.

The user can move the cursor within the file using the Home, End, up arrow, down arrow, Page Up, and Page Down keys. When adding or toggling breakpoints, and using the Goto-Cursor function, the cursor location is an important reference. MtDt searches first down, then up, starting from the cursor, to find a source line corresponding to an instruction.

The executable source code can be quickly reloaded from the Files menu by selecting the Executable, Fast submenu. This is normally required when the user has made a change to the executable source code and has re-built it

### **Mixed Assembly View**

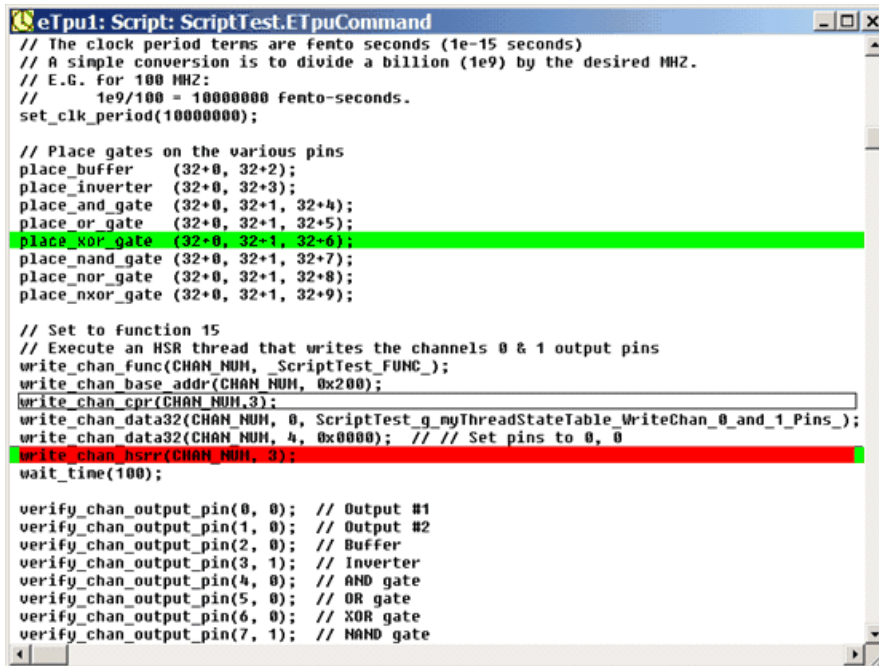
To make visible the dis-assembled instructions associated with each line of source code, select the Toggle, Assembly Mixed submenu from the Options menu. For TPU targets, the 32-bit hexadecimal equivalent of the micro-instruction is displayed. For CPU targets, the actual disassembly is displayed.

### **Coverage Indicators**

The little black, green, red, and white indicators on the left side of the source code window are graphical indications of code coverage as explained in the Code Coverage Analyses section.

## 15. Operational Status Windows

### 15.1.2 Script Commands Window



```
eTpu1: Script: ScriptTest.ETpuCommand
// The clock period terms are fento seconds (1e-15 seconds)
// A simple conversion is to divide a billion (1e9) by the desired MHZ.
// E.G. for 100 MHZ:
// 1e9/100 = 10000000 fento-seconds.
set_clk_period(10000000);

// Place gates on the various pins
place_buffer (32+0, 32+2);
place_inverter (32+0, 32+3);
place_and_gate (32+0, 32+1, 32+4);
place_or_gate (32+0, 32+1, 32+5);
place_xor_gate (32+0, 32+1, 32+6);
place_nand_gate (32+0, 32+1, 32+7);
place_nor_gate (32+0, 32+1, 32+8);
place_nxor_gate (32+0, 32+1, 32+9);

// Set to function 15
// Execute an HSR thread that writes the channels 0 & 1 output pins
write_chan_func(CHAN_NUM, _ScriptTest_FUNC_);
write_chan_base_addr(CHAN_NUM, 0x200);
write_chan_cpr(CHAN_NUM, 3);
write_chan_data32(CHAN_NUM, 0, ScriptTest_g_myThreadStateTable_WriteChan_0_and_1_Pins_);
write_chan_data32(CHAN_NUM, 4, 0x0000); // // Set pins to 0, 0
write_chan_hsr(CHAN_NUM, 3);
wait_time(100);

verify_chan_output_pin(0, 0); // Output #1
verify_chan_output_pin(1, 0); // Output #2
verify_chan_output_pin(2, 0); // Buffer
verify_chan_output_pin(3, 1); // Inverter
verify_chan_output_pin(4, 0); // AND gate
verify_chan_output_pin(5, 0); // OR gate
verify_chan_output_pin(6, 0); // XOR gate
verify_chan_output_pin(7, 1); // NAND gate
```

Although there are several types of script commands windows, only two styles can be viewed within a window: primary and ISR. The primary script commands file window displays the open or active primary script commands file whereas an ISR script commands file window displays a script commands file that is associated with a TPU channel interrupt. See the Script ISR section. for an explanation of these two types of script commands files.

A list of the available script commands functional groups is given in the Script Commands Groupings section.

The primary script commands file is loaded from the Files menu by selecting the Scripts, Open submenu and following the instructions of the Open Primary Script File dialog box.

The primary script commands file can be reread at anytime. This is done from the Files menu by selecting the Script, Fast submenu. MtDt re-executes the file, starting from the first command. When the primary script commands file is reread, MtDt state is not modified.

When MtDt is reset, the user has the option of re-initializing the primary script commands

file, opening a new or modified primary script commands file, or taking no action. If no action is taken, primary script commands file execution will continue from where it left off before the reset. The user selects the desired option via the Reset submenu in the Options menu. This submenu activates the Reset Options dialog box.

When the primary script commands file is reread, its execution starts back at the first line. This allows the user to modify and then rerun a series of script commands without exiting MtDt.

Only one primary script commands file may be active at a time. Multiple ISR script commands files may be open at once, but only a single ISR script commands can be associated with each TPU channel. Note that each ISR script commands file can be associated with multiple TPU channels.

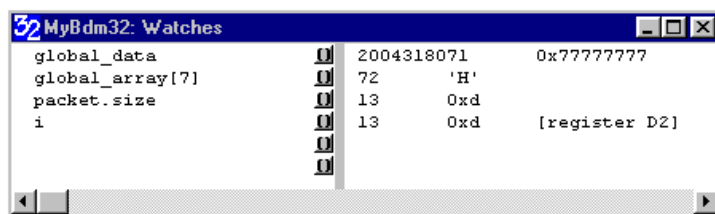
### Debugging Capabilities in Script Command Windows

These capabilities are available starting in version 3.20. Script commands file windows support breakpoints. Similar to source code windows, breakpoints can be set, cleared, enabled, and disabled. See the Breakpoints Menu section for a complete explanation.

A goto line capability is supported within the script commands file windows. Although this can be activated from the Run menu as described in the Run Menu section, a more convenient method is to right-click on the desired line. Either method causes the target execution to continue until the desired script command is executed. Note that if the active line in the script file is beyond the selected "goto" line the target is reset and then runs to the desired line.

A single step capability is also provided in script commands files. See the Step Menu section for a detailed explanation.

#### 15.1.3 Watch Windows



The Watches window displays symbolic data specified by the user. Both local and global variables can be displayed within this window. See the Local Variable window to

## 15. Operational Status Windows

---

automatically display local variables.

The watches window consists of a series of lines of text used to display user-specified watches. Watches can be removed, moved up or moved down. New watches can also be inserted before any current watch, or at the bottom of the list. The Insert Watch function accesses the Insert Watch dialog box which allows you to select and/or modify previously defined watches. These functions are accessed from the Options Menu `HELP_MENU_OPTIONS` by selecting an action listed in the Watch submenu. An equivalent of the Watch submenu is also accessible by right-clicking the mouse from within a Watches window.

The Watches window has a user-specified symbol on the far left. This is the symbol whose resolved value the user wishes to be displayed. To the right of the user-defined symbol is an options button. This button accesses the Watch Options dialog box. In future versions of this software, this dialog box will allow individual settings for the watch to be specified.

To the right of the options button is a vertical separator bar. You can drag the vertical separator bar left or right using the cursor. To the right of the vertical separator bar is the symbol resolution field. If MtDt can resolve a value from the user-specified symbol, MtDt automatically displays it in this field. Otherwise, MtDt displays a message indicating that the user-specified symbol could not be resolved.

The user can edit the user-resolution field. In future versions of this software, this will cause the actual variable values within the targets to be modified.

### Symbolic Data Options

The Watches window supports both global and local variables. Variables are resolved by looking at the innermost local scope first, followed by any outer scopes, in order, then followed by any static variables, and finally the global scope.

Currently, a subset of C syntax is supported for the left-hand side symbol input:

- Pointers can be dereferenced with the '\*' operator.
- The address of variables can be found with the '&' address operator.
- Array elements can be accessed with the '[' operator, where the subscript is an integer.
- Structure members can be accessed via the '.' or '->' operators.

In the current release, only a single operator per watch is supported. Future versions will support a more full-feature C syntax. Note that code must be compiled with symbolic

debug information for this functionality to be available.

See the Global eTPU Channel variableAccess section for information on accessing eTPU channel variables using the format shown below.

```
@<chan num/name>.<function var name>
```

### Viewing Named Timer Region Information

Code can be instrumented with named timing regions. Traversal time information across these regions can be viewed in the watch window using the following format.

```
@AshTimer.TimingRegionName
```

The traversal time and the number of system clocks in the last traversal are listed. Also the number of times traversed and the cumulative amount of time spend in the regions is also listed. Depending on the target the number of instruction cycles spend in the traversal region may also be listed.

A list of the named regions that are available for selection is in the Insert Watch dialog box .

### Viewing Print Action Command Output

Print action commands support instrumentation of code to output code execution information. Think printf. This information is normally output to the trace buffer, and from the trace buffer can be piped to trace files for post processing. But it is also possible to view this information in the watch window using the following command.

```
@AshTimer.FormatString
```

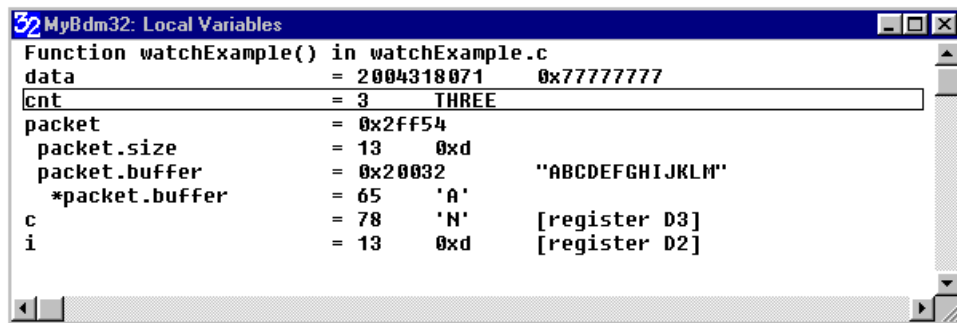
Where the format string matches exactly a Print Action Command's format string from within the source code.

See the Print Action Command section for more information on how to instrument your code.

A list of the available print action command strings is available for selection is in the Insert Watch dialog box.

## 15. Operational Status Windows

### 15.1.4 Local Variable Windows



The Local Variables window automatically displays all local variables in the current context, along with their current values. Variable names are listed in a column on the left-hand side of the window. Values are displayed in a matching column on the right-hand side. The displayed format is relevant to the variable type. Additionally, if the variable is assigned a register, the register is output.

Based upon type, variables are automatically expanded. For example, if a variable is of type `int*`, the dereferenced pointer is displayed on the next line as an integer. Default expansion is up to three levels deep, with pointers, arrays, and structures/unions/bitfields supported. An alternative expansion level can be specified. See the Local Variable Options Dialog Box section for more information.

Future enhancements will give the user control over expansion and collapse of variables. Note that in order for this feature to be available, code must be compiled with symbolic debug information.

In order for this window to correctly identify and display local variables, the proper options for each compiler must be chosen. For instance, debugging information needs to be included and certain stack frame requirements must be met. In certain cases, highly optimized code may cause erratic behavior in this window. See the ASH WARE Web page for a detailed explanation of the correct compiler settings for each target, and limitations when using certain specific compiler settings.

Note that as the target executes, the contents of the Local Variables window will usually change quite a bit. This is because each function generally has a unique set of local variables that are displayed. As the target moves from function to function, only the local variables of the currently executed function are displayed.

Global variables are not displayed within this window. See the description of the Watches

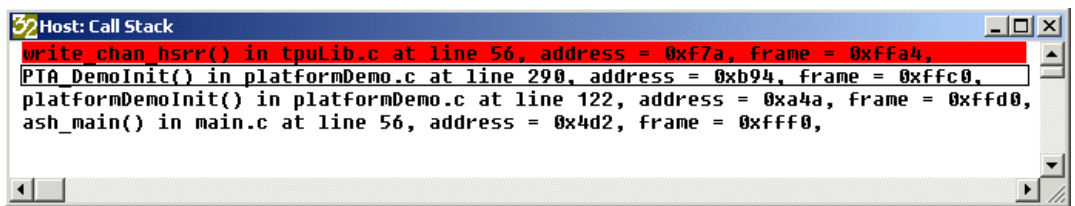
window for information on how to display global variables.

### Local Variable Window Automation

In conjunction with the Call Stack window the Local Variables window can display the local variables of functions that have been pushed onto the call stack. In a Call Stack window, move the cursor to a line associated with a function that has been pushed onto the call stack. This causes the local variables from that function's context to be displayed in the Local Variables window.

Occasionally it is desirable to lock the local variable to display the local variables of a particular function. To do this, you must disable the automation. This is done within a Local Variables window by opening the Local Variable Options Dialog Box and selecting the "lock the local variables ..." option. Locking and unlocking of the current function's context can also be done more quickly by selecting either the "lock" or the "unlock" options from the popup menu that appears when you right-click the mouse within a Local Variable window.

## 15.1.5 Call Stack Window



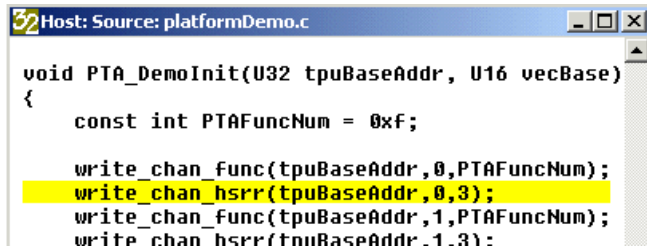
The Call Stack window displays the function, source code file name and line number, address, and stack frame pointer associated with stacked functions as shown above.

### Call Stack, Local Variables, and Source Window Automation

It is often not sufficient to simply view the functions on the call stack. The Call Stack window works in conjunction with the source code and the local variable window to show both the source code line and the stacked local variables associated with any stacked function.

For instance, the second line of the above Call Stack has been selected. This line is associated with the PTA\_DemoInit function. When you move the cursor to this line the source code file shown below automatically pops into view and the source code line scrolls into view and is highlighted with yellow.

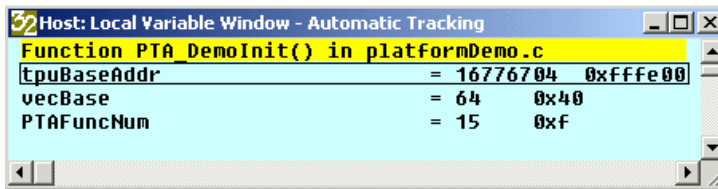
## 15. Operational Status Windows



```
Host: Source: platformDemo.c
void PTA_DemoInit(U32 tpuBaseAddr, U16 vecBase)
{
    const int PTAFuncNum = 0xf;

    write_chan_func(tpuBaseAddr, 0, PTAFuncNum);
    write_chan_hsrr(tpuBaseAddr, 0, 3);
    write_chan_func(tpuBaseAddr, 1, PTAFuncNum);
    write_chan_hsrr(tpuBaseAddr, 1, 3);
}
```

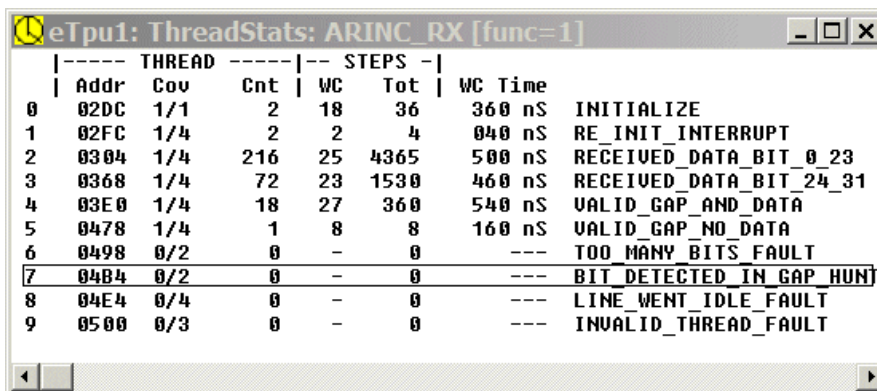
Upon selection of the Call Stack window's second line, the Local Variables window is also affected as shown below. The highlighted function, PTA\_DemoInit had several local variables that were stored on the stack. The values of these local variables are automatically displayed in the Local Variables window.



```
Host: Local Variable Window - Automatic Tracking
Function PTA_DemoInit() in platformDemo.c
tpuBaseAddr = 16776704 0xffffe00
vecBase = 64 0x40
PTAFuncNum = 15 0xf
```

### 15.1.6 Thread Window

This window has a number of user-selectable views and options. The window shown above has been configured to track all instances of the function, "ARINC\_RX." Since multiple channels can run this particular function, the data shown by this window is an accumulation of all the channels that have been assigned this function.



Thread	Addr	Cov	Cnt	WC	Tot	WC Time	Description
0	02DC	1/1	2	18	36	360 nS	INITIALIZE
1	02FC	1/4	2	2	4	040 nS	RE_INIT_INTERRUPT
2	0304	1/4	216	25	4365	500 nS	RECEIVED_DATA_BIT_0_23
3	0368	1/4	72	23	1530	460 nS	RECEIVED_DATA_BIT_24_31
4	03E0	1/4	18	27	360	540 nS	VALID_GAP_AND_DATA
5	0478	1/4	1	8	8	160 nS	VALID_GAP_NO_DATA
6	0498	0/2	0	-	0	---	TOO MANY BITS FAULT
7	04B4	0/2	0	-	0	---	BIT DETECTED IN GAP HUNT
8	04E4	0/4	0	-	0	---	LINE_WENT_IDLE_FAULT
9	0500	0/3	0	-	0	---	INVALID_THREAD_FAULT

TPU and eTPU code executes in response to events. This event-response code is known

as a thread. Individual threads are assigned to each event and event combination. A key performance index of your code is the amount of time each thread takes to execute. Therefore, this thread window is an important tool for determining the performance of your code.

Two important columns are "WC Steps and "WC Time." These display the worst case number of execution steps (including flushes) and execution time (also including flushes) for that thread. Note that this thread will normally execute many, many times in the course of a simulation run, and each time the thread executes it may take a different path, such that the execution time may vary. The worst case number shows the very worst (longest) amount of time that the thread takes to execute through the entire simulation run.

### Finding a Thread's Source Code

To find the line of source code for each thread, move the cursor within the thread window. The first source code line of that thread automatically pops into view and turns yellow, as shown below.

The screenshot shows two windows from the eTPU Simulator. The top window displays source code for `etpu1: Source: ..\eTPU\etpuc_arinc_rx.c`. The line `RECEIVED_DATA_BIT_24_31: ClearTransLatch();` is highlighted in yellow. The bottom window displays `etpu1: ThreadStats: ARINC_RX [func=1]` with the following table:

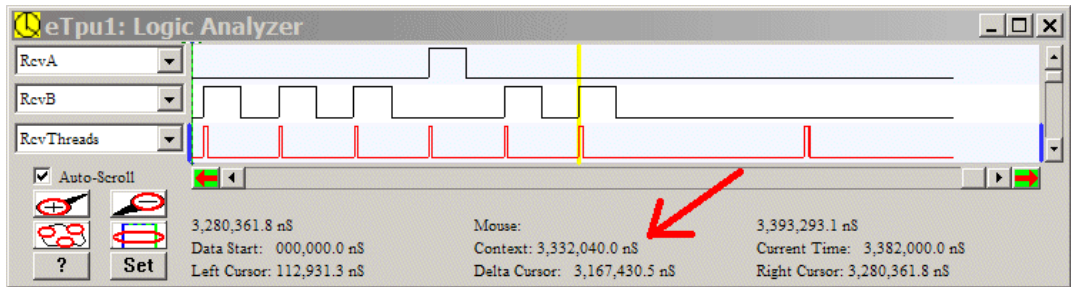
THREAD				STEPS			
Addr	Cov	Cnt	WC	To	WC Time		
0	02DC	1/1	2	18	360	360 nS	INITIALIZE
1	02FC	1/4	2	2	4	040 nS	RE_INIT_INTERRUPT
2	0304	1/4	216	25	4365	500 nS	RECEIVED_DATA_BIT 0 23
3	0368	1/4	72	23	1530	460 nS	RECEIVED_DATA_BIT 24 31
4	03E0	1/4	18	27	360	540 nS	VALID_GAP_AND_DATA
5	0478	1/4	1	8	8	160 nS	VALID_GAP_NO_DATA
6	0498	0/2	0	-	0	---	TOO_MANY_BITS_FAULT
7	04B4	0/2	0	-	0	---	BIT_DETECTED_IN_GAP_HUNT
8	04E4	0/4	0	-	0	---	LINE_WENT_IDLE_FAULT
9	0500	0/3	0	-	0	---	INVALID_THREAD_FAULT

### Re-Executing the Worst Case Thread

If the thread has executed at least once, such that there is a worst case thread time

## 15. Operational Status Windows

available, then the time at which the thread occurs appears as a yellow vertical line in the logic analyzer window, and the time at which the thread occurred is shown as the "context time" in the logic analyzer window, as shown below. To re-execute this thread, "grab" this context time by moving the cursor over the context time (such that an open hand appears) and depressing the left mouse button (such that a closed hand appears). With the left mouse button depressed, move the closed-hand cursor to the right. Position the cursor over the field labeled, "current time," and then release the left mouse button. The simulation will reset, then run until it reaches the "context time," which was the time at which the worst-case thread executed.



The window shown below is similar to the window set to the ARINC\_FX function, except that it is configured to show only the information from the channel named, "RcvA." Note that this name has been assigned to this channel in the test vector file using the NODE command, as defined in the test vector file section. Because it displays only the information from a single channel, it may not reflect the true worst case.

The screenshot shows the ThreadStats window for RcvA [chan=31]. It displays a table with columns for Thread ID, Address, Coverage, Count, Worst Case (WC), Total (Tot), Worst Case Time (WC Time), and Thread Name.

Thread	Addr	Cov	Cnt	WC	Tot	WC Time	Thread Name
0	02DC	1/1	1	18	18	360 nS	INITIALIZE
1	02FC	1/4	2	2	4	040 nS	RE_INIT_INTERRUPT
2	0304	1/4	115	25	2330	500 nS	RECEIVED_DATA_BIT_0_23
3	0368	1/4	25	23	529	460 nS	RECEIVED_DATA_BIT_24_31
4	03E0	1/4	9	16	123	320 nS	VALID_GAP_AND_DATA
5	0478	0/4	0	-	0	---	VALID_GAP_NO_DATA
6	0498	0/2	0	-	0	---	TOO_MANY_BITS_FAULT
7	04B4	0/2	0	-	0	---	BIT_DETECTED_IN_GAP_HUNT
8	04E4	0/4	0	-	0	---	LINE_WENT_IDLE_FAULT
9	0500	0/3	0	-	0	---	INVALID_THREAD_FAULT

One issue with this window is that the worst case threads during initialization are often much worse than is seen during execution of the function. Since these initialization threads are not normally an issue it is helpful to be able to clear out all thread data following

initialization. This is done using the script commands described in the Thread Script Commands section

Note that in the above window the Cov (coverage) column shows  $\frac{1}{4}$  for most of the threads. This is because each of these threads has been configured to respond to four different event vector combinations, yet the simulation run to this point has covered only one of these. Which of these event vector combinations has been covered? Select the "ungroup" option to see.

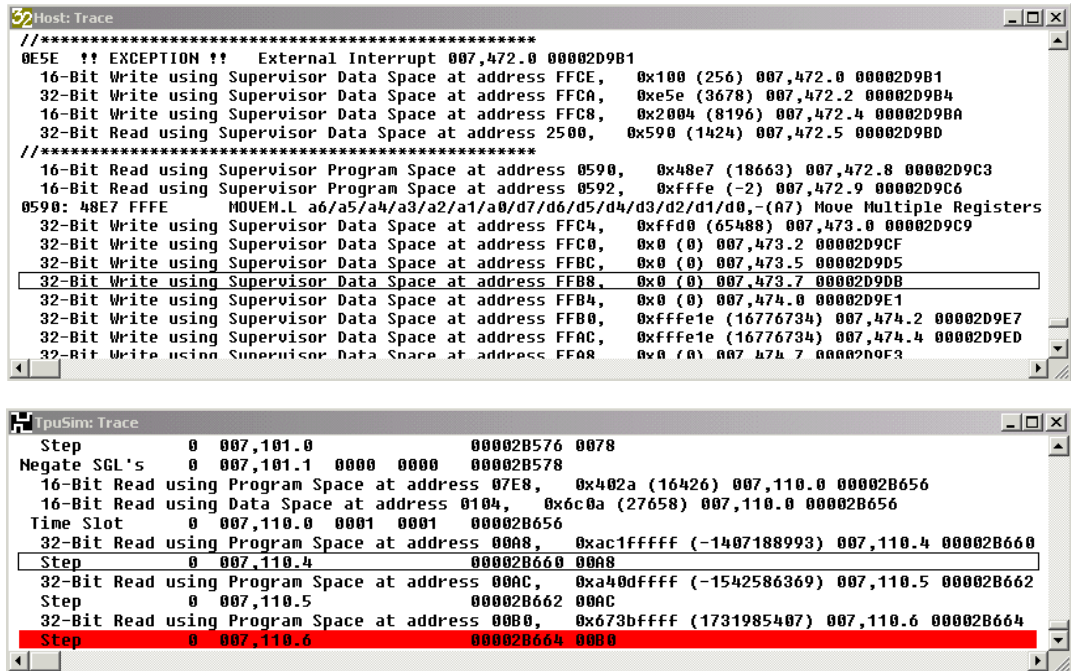
Table	Addr	THREAD	Cnt	WC	Tot	WC Time	
0	0040	02FC	2	2	4	040 nS	RE_INIT_INTERRUPT
1	0042	02FC	0	-	0	---	RE_INIT_INTERRUPT
2	0044	02FC	0	-	0	---	RE_INIT_INTERRUPT
3	0046	02FC	0	-	0	---	RE_INIT_INTERRUPT
4	0048	02DC	1	18	18	360 nS	INITIALIZE
5	004A	0500	0	-	0	---	INVALID_THREAD_FAULT
6	004C	0500	0	-	0	---	INVALID_THREAD_FAULT
7	004E	0500	0	-	0	---	INVALID_THREAD_FAULT
8	0050	04E4	0	-	0	---	LINE_WENT_IDLE_FAULT
9	0052	0478	0	-	0	---	VALID_GAP_NO_DATA
10	0054	04E4	0	-	0	---	LINE_WENT_IDLE_FAULT
11	0056	03E0	9	16	123	320 nS	VALID_GAP_AND_DATA
12	0058	04E4	0	-	0	---	LINE_WENT_IDLE_FAULT
13	005A	0478	0	-	0	---	VALID_GAP_NO_DATA
14	005C	04E4	0	-	0	---	LINE_WENT_IDLE_FAULT
15	005E	03E0	0	-	0	---	VALID_GAP_AND_DATA
16	0060	0304	0	-	0	---	RECEIVED_DATA_BIT_0_23
17	0062	04B4	0	-	0	---	BIT_DETECTED_IN_GAP_HUNT
18	0064	0368	0	-	0	---	RECEIVED_DATA_BIT_24_31
19	0066	0498	0	-	0	---	TOO_MANY_BITS_FAULT
20	0068	0304	115	25	2330	500 nS	RECEIVED_DATA_BIT_0_23
21	006A	04B4	0	-	0	---	BIT_DETECTED_IN_GAP_HUNT
22	006C	0368	25	23	529	460 nS	RECEIVED_DATA_BIT_24_31
23	006E	0498	0	-	0	---	TOO_MANY_BITS_FAULT
24	0070	0304	0	-	0	---	RECEIVED_DATA_BIT_0_23
25	0072	0478	0	-	0	---	VALID_GAP_NO_DATA
26	0074	0368	0	-	0	---	RECEIVED_DATA_BIT_24_31
27	0076	03E0	0	-	0	---	VALID_GAP_AND_DATA
28	0078	0304	0	-	0	---	RECEIVED_DATA_BIT_0_23
29	007A	0478	0	-	0	---	VALID_GAP_NO_DATA
30	007C	0368	0	-	0	---	RECEIVED_DATA_BIT_24_31
31	007E	03E0	0	-	0	---	VALID_GAP_AND_DATA

The "group" and "ungroup" commands allow the same threads from different event vector response combinations to be grouped together. The window shown has this option set to ungroup. Note that there are 32-event vector combinations. To get good testing coverage all of these event combinations should be tested. By setting the window to show all event

## 15. Operational Status Windows

response combinations it becomes clear that the simulation run on this window is not fully exercising the function, and as such the event vector coverage is poor.

### 15.1.7 Trace Window



```
Host: Trace
//*****
0E5E !! EXCEPTION !! External Interrupt 007,472.0 00002D9B1
16-Bit Write using Supervisor Data Space at address FFCE, 0x100 (256) 007,472.0 00002D9B1
32-Bit Write using Supervisor Data Space at address FFCA, 0xe5e (3678) 007,472.2 00002D9B4
16-Bit Write using Supervisor Data Space at address FFC8, 0x2004 (8196) 007,472.4 00002D9B8
32-Bit Read using Supervisor Data Space at address 2500, 0x590 (1424) 007,472.5 00002D9B0
//*****
16-Bit Read using Supervisor Program Space at address 0590, 0x48e7 (18663) 007,472.8 00002D9C3
16-Bit Read using Supervisor Program Space at address 0592, 0xffff (-2) 007,472.9 00002D9C6
0590: 48E7 FFFE MOVEM.L a6/a5/a4/a3/a2/a1/a0/d7/d6/d5/d4/d3/d2/d1/d0,-(A7) Move Multiple Registers
32-Bit Write using Supervisor Data Space at address FFC4, 0xffd0 (65488) 007,473.0 00002D9C9
32-Bit Write using Supervisor Data Space at address FFC0, 0x0 (0) 007,473.2 00002D9CF
32-Bit Write using Supervisor Data Space at address FFB8, 0x0 (0) 007,473.5 00002D9D5
32-Bit Write using Supervisor Data Space at address FFB8, 0x0 (0) 007,473.7 00002D9DB
32-Bit Write using Supervisor Data Space at address FFB4, 0x0 (0) 007,474.0 00002D9E1
32-Bit Write using Supervisor Data Space at address FFB0, 0xffffe1e (16776734) 007,474.2 00002D9E7
32-Bit Write using Supervisor Data Space at address FFAC, 0xffffe1e (16776734) 007,474.4 00002D9ED
32-Bit Write using Supervisor Data Space at address FFA8, 0x0 (0) 007,474.7 00002D9F3

TpuSim: Trace
Step 0 007,101.0 00002B576 0078
Negate SGL's 0 007,101.1 0000 0000 00002B578
16-Bit Read using Program Space at address 07E8, 0x402a (16426) 007,110.0 00002B656
16-Bit Read using Data Space at address 0104, 0xc0a (27658) 007,110.0 00002B656
Time Slot 0 007,110.0 0001 0001 00002B656
32-Bit Read using Program Space at address 00A8, 0xac1ffff (-1407188993) 007,110.4 00002B660
Step 0 007,110.4 00002B660 00A8
32-Bit Read using Program Space at address 00AC, 0xa40ffff (-1542586369) 007,110.5 00002B662
Step 0 007,110.5 00002B662 00AC
32-Bit Read using Program Space at address 00B0, 0x673bffff (1731985407) 007,110.6 00002B664
Step 0 007,110.6 00002B664 00B0
```

The Trace window displays information relating to the instructions that were executed. The Trace window displays all information that has been stored in the trace buffer. See the Trace Options Dialog Box section for information on how to modify the information that is stored in this buffer.

The contents of the trace window can be saved to a trace file. This is helpful for post-processing of the trace data. See the Trace Buffer and Files section for information on how this is done.

For most simulation models, the data flow between processor and memory can be displayed. Timing information is included but it should be noted that the most ASH WARE simulation models use an intuitive rather than a true timing model so these will vary slightly relative to the real CPU, TPU, or other target model type.

Note that in hardware targets, like the 683xx Hardware Debugger, this window will not function correctly because many hardware targets do not contain trace buffers. Therefore, MtDt does not know which instructions executed last. The instructions that are displayed are those from previous single-steps of the target. Gaps in the trace buffer are noted, and the displayed execution times are an approximation based on the real time clock of the PC.

### **Saving Trace Data to a File**

The trace data can be saved to a file either directly through the GUI or from within a script commands file. This is explained in the TRACE BUFFER AND FILES section

### **TPU Simulation Considerations**

The TPU simulation model provides several additional events that can be viewed in the Trace window. These are in addition to the normal data flow between TPU and memory and opcode execution, etc., that are available for most targets. These events include execution steps, time state transitions, active channel transitions, and four CPU clock NOPs. This window serves two purposes. It shows the thread of execution, and it can be used to analyze channel service timing.

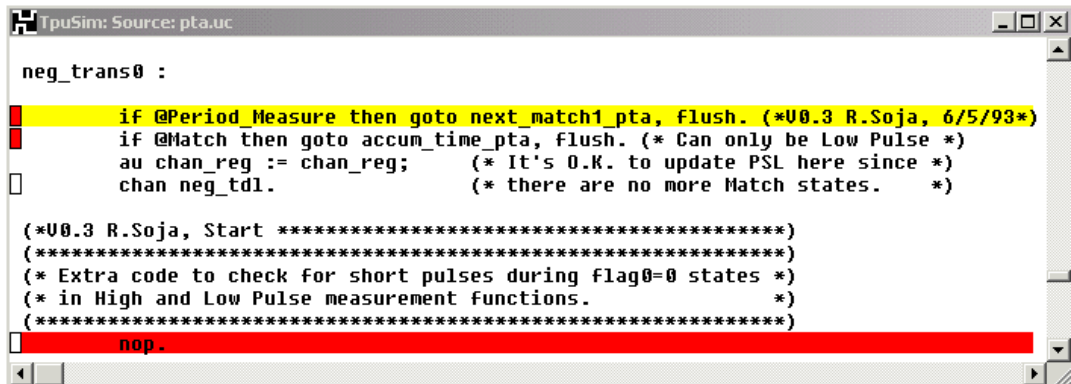
When the TPU services a particular channel, a 10-clock time slot transition occurs. The states of the Service Request Latch (SRL) and Service Grant Latch (SGL), as well as a timestamp and a channel number, are displayed for each time slot transition. The SRL is the fourth field from the left, while the SGL field is displayed to the right of the SRL field.

When all pending service requests from a particular priority level have been serviced, the TPU negates the service grant bits associated with those channels. This requires four CPU clocks. For this event MtDt displays the timestamp and the states of the SRL and the SGL.

### **Trace/Source Code Automation**

The trace window and the Source Code window(s) can be used together. Select a line in the Source Code Window associated with an instruction execution. The source code file associated with this line automatically pops into view and the associated line scrolls into view and is highlighted in yellow. In the TPU trace window shown above, the seventh line has been selected. The line is automatically displayed and highlighted yellow as shown below.

## 15. Operational Status Windows

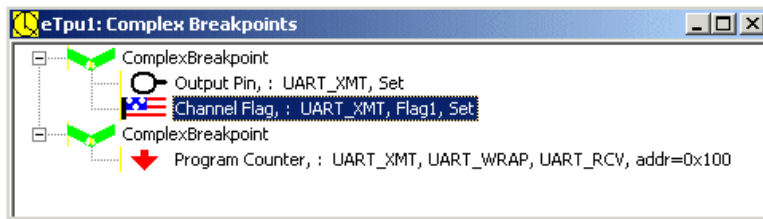


```
TpuSim: Source: pta.uc

neg_trans0 :
if @Period_Measure then goto next_match1_pta, flush. (*U0.3 R.Soja, 6/5/93*)
if @Match then goto accum_time_pta, flush. (* Can only be Low Pulse *)
au chan_reg := chan_reg;      (* It's O.K. to update PSL here since *)
chan neg_td1.                 (* there are no more Match states. *)

(*U0.3 R.Soja, Start *****
*****
* Extra code to check for short pulses during flag0=0 states *
* in High and Low Pulse measurement functions. *
*****
nop.
```

### 15.1.8 Complex Breakpoint Window

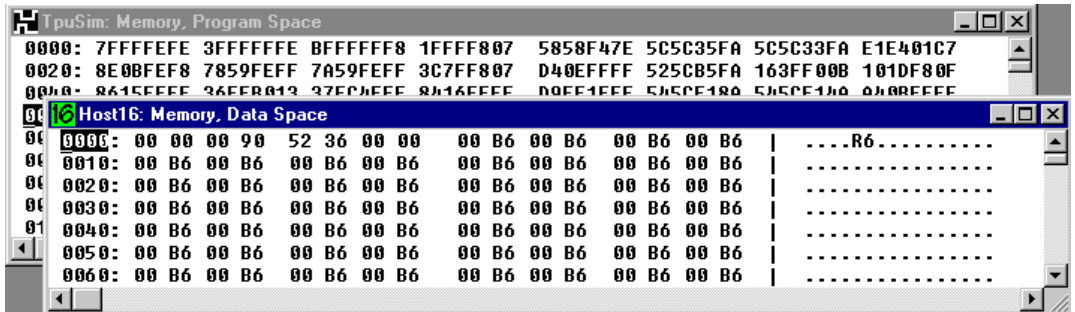


Complex breakpoints are added, removed, and modified from within the complex breakpoint window. Complex breakpoints support the ability to halt the target on the occurrence of one or more combinations of conditions. Each complex breakpoint operates independently of all other complex breakpoints.

Each complex breakpoint can have one or more conditionals. When multiple conditionals are added to the same breakpoint then all conditionals must simultaneously resolve to "true" in order for the complex breakpoint to halt the target(s). Conditionals are added and modified using the Complex Breakpoint Conditional dialog box.

Depending on the target, conditionals can include input/output and clock pins, thread activity, host service requests, and program counter value, variable values or tests, etc.

### 15.1.9 Memory Dump Window



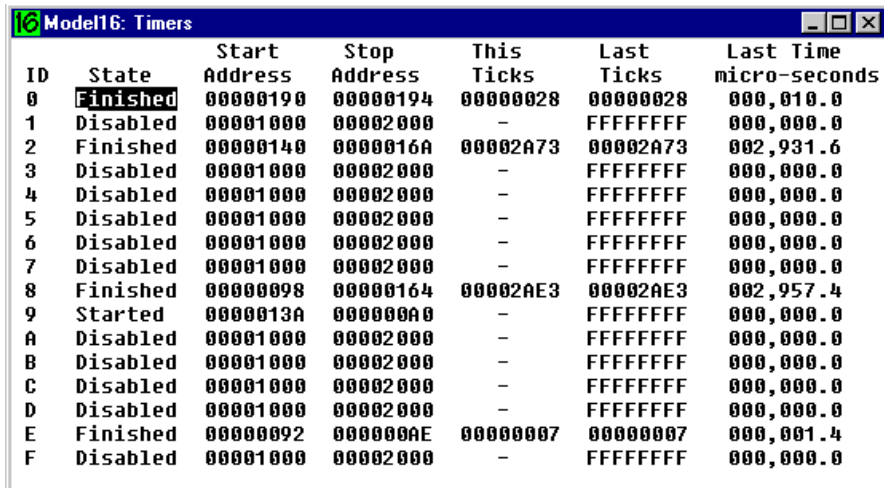
The Memory Dump Window displays memory. A number of options are available. To change a viewing option, activate a memory window and from the Options menu by select the desired option from the Memory submenu.

- Memory is viewable in 8-, 16-, or 32-bit mode.
- The ASCII-equivalent text on the right side can be turned off or on.
- The address space can be specified.
- The base address of the window can be specified.

A common development tool deficiency is that the vertical scroll bar is unusable because it causes too much memory to be traversed. MtDt addresses this problem by limiting the scroll range of the vertical scroll bar. A memory dump window displays only a small amount of the total address space. The portion of memory that the memory window displays is specified by the base address parameter.

## 15. Operational Status Windows

### 15.1.10 Timers Window



ID	State	Start Address	Stop Address	This Ticks	Last Ticks	Last Time micro-seconds
0	Finished	00000190	00000194	00000028	00000028	000,010.0
1	Disabled	00001000	00002000	-	FFFFFFFF	000,000.0
2	Finished	00000140	0000016A	00002A73	00002A73	002,931.6
3	Disabled	00001000	00002000	-	FFFFFFFF	000,000.0
4	Disabled	00001000	00002000	-	FFFFFFFF	000,000.0
5	Disabled	00001000	00002000	-	FFFFFFFF	000,000.0
6	Disabled	00001000	00002000	-	FFFFFFFF	000,000.0
7	Disabled	00001000	00002000	-	FFFFFFFF	000,000.0
8	Finished	00000098	00000164	00002AE3	00002AE3	002,957.4
9	Started	0000013A	000000A0	-	FFFFFFFF	000,000.0
A	Disabled	00001000	00002000	-	FFFFFFFF	000,000.0
B	Disabled	00001000	00002000	-	FFFFFFFF	000,000.0
C	Disabled	00001000	00002000	-	FFFFFFFF	000,000.0
D	Disabled	00001000	00002000	-	FFFFFFFF	000,000.0
E	Finished	00000092	000000AE	00000007	00000007	000,001.4
F	Disabled	00001000	00002000	-	FFFFFFFF	000,000.0

The Timers window displays the state of the 16 ASH WARE timers. See the Integrated Timers chapter for a detailed explanation.

Each row of the window contains information regarding a single timer. The ID field on the far left indicates which of the 16 timers the row represents. The State field is to the right of the ID field. This field indicates the current state of the timer. The possible states are generally disabled, armed, started, finished, and overrun. Double-clicking with the left mouse button forces the timer into the next state. The timer can also be forced into a specific state by typing the first letter of the desired state.

The Start Address and Stop Address fields are to the right of the State field. These fields contain the addresses that, when accessed by the target, cause the state to change. For instance, a timer in the armed state changes to the started state when the start address is executed. Then, when the stopped state is executed, the timer progresses to the finished state.

The This Ticks and the Last Ticks fields are to the right of the Stop Address field. These fields indicate how many clock ticks of the target have occurred for any timer calculation. Why are these split into two fields? This allows the last calculated timing to be retained while a new timer calculation is underway. When a timer is re-armed, the This Ticks field goes blank, while the Last Ticks field retains the previous value.

To the right of the Last Ticks field is the Last Time field. The Last Time field is identical to the Last Ticks field except that the timer result is displayed in micro-seconds instead of

clock ticks.

Note that the Last Ticks and This Ticks fields use the target's current clock period in the calculation. Only the start time and stop time are retained by the timers. This means that if the clock period changes during the course of a calculation, these clock tick calculations will not be correct. Note also that the time calculation on simulated targets does not use the clock period so this field will be correct even if the clock period is changed.

### 15.2 eTPU-Specific Windows

The following windows are available in the eTPU Simulator and eTPU System Simulator.

## 15. Operational Status Windows

### 15.2.1 eTPU Configuration Window



```
eTPU_A: Configuration
General
CPU Frequency: 100.000 MHz
CPU Clock: 000000000
Latest thread is [ ] steps (0 clocks)
Active=50.000%, Threads=0, Steps=0
Entry Table Base Addr: 0000 [ETB=00]
TCRCLK Input Pin: High
Active Channel T[0]..T[-4]: 0, 0, 0, 0
Address: 0000, Rtn: 0000
ETPU Phase: Not Supported
Interrupts
Microcode Global Exception: clr
Illegal Instruction: clr
Verification:
Continuous Behavior = Off
Behavior Failures: 00000000
Script Failures: 00000000
Versions:
Simulator: 3.70, Build A
Model: JPC563M60-1 engine A (eTPU-2), Memory: 12K CODE, 2.5K DATA.
Code: ETEC_cc 1.10 Build A : ELF/DWARF 2.0, Built for eTPU-2
Entry Point On Pin Direction (ETPD) Supported
Out-Of-Range Opcode (SCMOFFDATAR) Supported
Files:
Application C:\Mtdt\Gui\TESTFILES\Mtdt.exe
Project: C:\Mtdt\TestsHigh\ETpu\AutoRunOregon1ETpu.ETpuSimProject
Mtdt Build: C:\Mtdt\Gui\TESTFILES\BuildScripts\zzz_1ETpuSim.MtdtBuild
Source Code: ..\..\DemosSpc563Mxx\ETPU2\etpu2_code.elf
Script: DemoLimitsETpu2.ETpuCommand
Script Report: DemoLimitsETpu2.Report
Script Startup: -No File Open-
Test Vector: -No File Open-
Build: -No File Open-
Master Behavior: -No File Open-
Semaphores:
Semaphore 0 Free -
Semaphore 1 Free -
Semaphore 2 Free -
Semaphore 3 Free -
```

**General** The CPU frequency is displayed. The eTPU executes at half of this CPU frequency in that a single eTPU instruction takes two CPU clocks to execute. The 'CPU clocks' field displays the number of CPU clock ticks that have occurred since the last reset. The latest thread in both steps (opcode execution) and NOPs and CPU clocks is displayed. The execution unit activity level since reset is shown. This is the total number of CPU clocks divided by the number in which the eTPU was either in a Time Slot Transition or in a thread. The current execution unit address and the address to which the unit will begin executing should a return statement be encountered are displayed.

**Interrupts** The state of the Microcode Global Exception (MGE1/MGE2) and the Illegal Instruction Flag (ILF1/ILF2) are displayed which are both part of the ETPUMCR register.

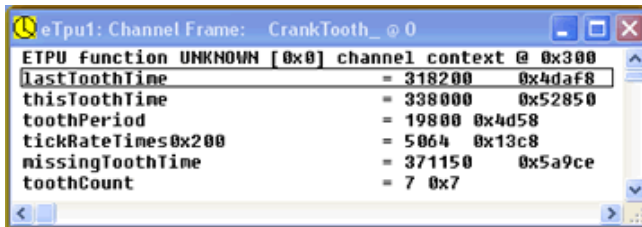
**Verification** The state of continuous verification (enabled or disabled) is displayed as well as the number of pin transition behavior and script command failures are displayed.

**Versions** The eTPU simulator version displays the major and minor release letters along with the build letter. The model displays the microcontroller version and engine (some microcontrollers contain dual execution units designated ‘A’ and ‘B’) and the size of the code and data memories. The ‘code’ line contains compiler-specific information such as compiler name, file format (such as .COD or ELF/DWARF 2.0) as well as the intended eTPU target ( the original eTPU-1 has fewer capabilities than the eTPU-2). Certain eTPU minor revisions added support for entry table pin direction control (ETPD) and the opcode that is returned when an opcode is fetched from an invalid address (SCMOFFDATAR.)

**Files** The simulator executable with full path, project file, and MtDt build file, eTPU source code file are displayed. The name of the primary script file and the corresponding parse report file are displayed. The startup script file is a special file that runs to completion immediately following reset. It is generally not used. The test vector file and build batch file and any master behavior verification file names (gold file) are displayed.

**Semaphores** The state (locked or free) of each of the four semaphores, along with an indicator of which eTPU is locking the semaphore, is displayed.

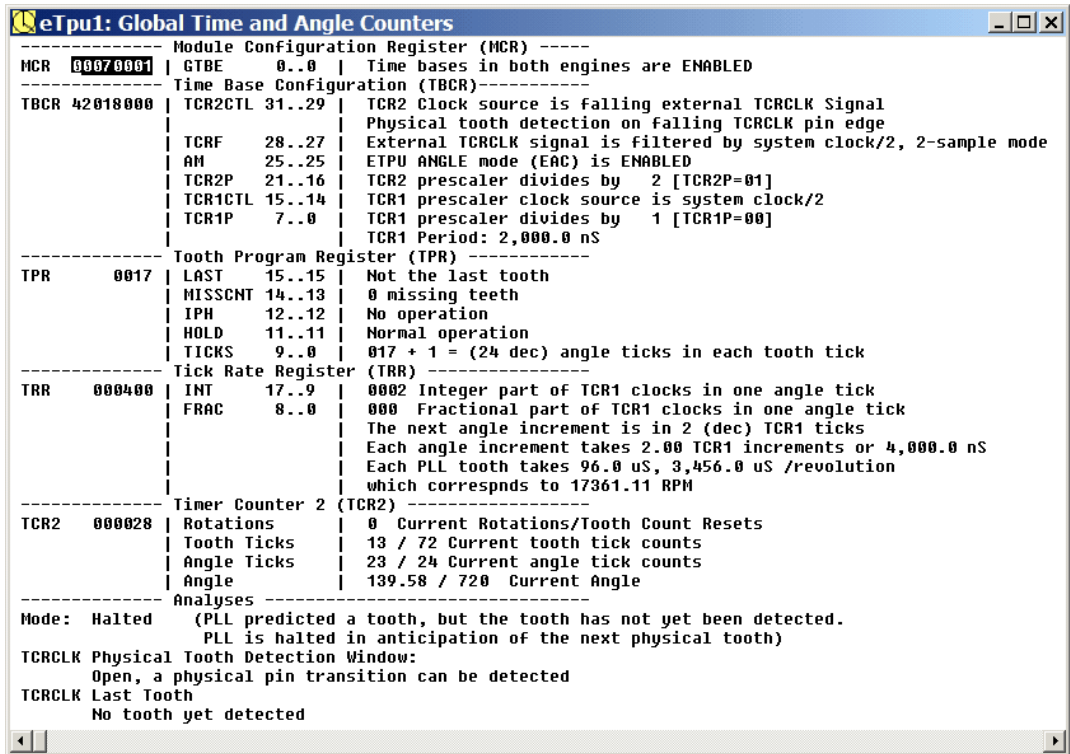
## 15.2.2 eTPU Channel Frame Window



The Channel Frame window shows the channel function and static local variables belonging to a particular channel. Right click the mouse in the window to change the channel.

## 15. Operational Status Windows

### 15.2.3 eTPU Global Timer and Angle Counters Window



```
eTpu1: Global Time and Angle Counters
----- Module Configuration Register (MCR) -----
MCR 00070001 | GTBE 0..0 | Time bases in both engines are ENABLED
----- Time Base Configuration (TBCR) -----
TBCR 42018000 | TCR2CTL 31..29 | TCR2 Clock source is falling external TCRCLK Signal
| | | Physical tooth detection on falling TCRCLK pin edge
| TCRF 28..27 | External TCRCLK signal is filtered by system clock/2, 2-sample mode
| AM 25..25 | ETPU ANGLE mode (EAC) is ENABLED
| TCR2P 21..16 | TCR2 prescaler divides by 2 [TCR2P=01]
| TCR1CTL 15..14 | TCR1 prescaler clock source is system clock/2
| TCR1P 7..0 | TCR1 prescaler divides by 1 [TCR1P=00]
| | | TCR1 Period: 2,000.0 nS
----- Tooth Program Register (TPR) -----
TPR 0017 | LAST 15..15 | Not the last tooth
| MISSCNT 14..13 | 0 missing teeth
| IPH 12..12 | No operation
| HOLD 11..11 | Normal operation
| TICKS 9..0 | 017 + 1 = (24 dec) angle ticks in each tooth tick
----- Tick Rate Register (TRR) -----
TRR 000400 | INT 17..9 | 0002 Integer part of TCR1 clocks in one angle tick
| FRAC 8..0 | 000 Fractional part of TCR1 clocks in one angle tick
| | | The next angle increment is in 2 (dec) TCR1 ticks
| | | Each angle increment takes 2.00 TCR1 increments or 4,000.0 nS
| | | Each PLL tooth takes 96.0 uS, 3,456.0 uS /revolution
| | | which corresponds to 17361.11 RPM
----- Timer Counter 2 (TCR2) -----
TCR2 000028 | Rotations | 0 Current Rotations/Tooth Count Resets
| Tooth Ticks | 13 / 72 Current tooth tick counts
| Angle Ticks | 23 / 24 Current angle tick counts
| Angle | 139.58 / 720 Current Angle
----- Analyses -----
Mode: Halted (PLL predicted a tooth, but the tooth has not yet been detected.
PLL is halted in anticipation of the next physical tooth)
TCRCLK Physical Tooth Detection Window:
Open, a physical pin transition can be detected
TCRCLK Last Tooth
No tooth yet detected
```

The Module Configuration Registers (MCR) Global Time Base Enable (GTBE) field indicates if the time bases in the eTPU(s) are enabled.

The Time Base Configuration Register's (TBCR) TCR1CTL and TCR2CTL fields display the configuration of the TCR1 and TCR2 counters. The TCR1P and TCR2P fields show both the field value and the actual divisor of the prescaler. The TCRF displays the filtering options though these are not simulated by the eTPU Simulation engine. The Angle Mode (AM) bit indicates if angle mode is enabled.

The Tooth Program Register (TPR) is only used in angle mode. This register may be written only by the eTPU code. The LAST bit indicates that angle mode should be reset on the next tooth. The MISSCNT field indicates how many incoming teeth are expected to be missing such that the angle mode PLL will continue to count synthesized teeth and not wait for incoming physical tooth signals. The Insert Physical tooth (IPH) field allows the eTPU code to tell the angle mode hardware to proceed as if an incoming physical tooth had been detected. The HOLD field forces the angle mode logic to halt as if more

incoming physical teeth had been detected than were expected. The TICKS field specifies the number of angle ticks (TCR2 increments) that are in each physical tooth. As such, this is effectively the PLL multiplier for the difference between the frequency of the incoming teeth, and the frequency of the synthesized angle ticks.

The Tick Rate Register (TRR) should be updated by the eTPU code on each incoming physical tooth. It is calculated based on the time difference between the last two incoming physical teeth and specifies the time of each angle tick in TCR1 ticks. In order to reduce error, both an integer (INT) and fractional (FRAC) portion of the ratio of TCR1 counter ticks to TCR2 ticks is supported.

In angle mode the TCR2's tick rate is proportional to angle instead of time. As such the TCR2 counter may be reset when it completes a cycle, which is every 720 degrees in a typical car. The number of such cycles since the last reset is shown, as are the number of PLL-synthesized teeth and the number of synthesized angle ticks. Additionally, the current angle is shown which is based on a default of 720 degrees per cycle. The default teeth per cycle and degrees per cycle can be overridden using the `set_angle_indices()`; script command as explained in the eTPU Time Base Configuration Script Commands section.

The STAC bus is used share the TCR1 and TCR2 global counters between eTPU engines such that (say) the angle (TCR2) used in both eTPU\_A and eTPU\_B is identical. Script commands to do this are in the STAC Bus Script Commands section.

Some analysis is provided of the operational state of angle mode. Angle mode is in normal, wait, or high speed depending on whether the PLL is on track, ahead, or behind. Channel 0 can be programmed to form a sampling window and the open or closed state of this window is displayed. The edge which the angle mode hardware is displayed, and it should be noted that this edge must be programmed to correspond the edge that is also shown which is the edge being detected by channel 0 and which forms a detection window. Spurious operation could result if these do not correspond. Also, channel 0 can be programmed to form a detection window and the state of this detection window (closed, opened) is displayed.

## 15. Operational Status Windows

### 15.2.4 eTPU Host Interface Window

	1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
CFSR:	00	00	00	00	00	00	00	00	00	00	00	06	05	05	05	05	04	04	04	04	04	04	00	03	03	03	03	03	03	00	02	01
CPR:	-	-	-	-	-	-	-	-	-	-	-	HI	HI	HI	HI	HI	HI	HI	HI	HI	HI	HI	HI	HI	HI	HI	HI	HI	HI	HI	HI	
HSRR:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
FM:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
Entry:	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	
CISR:	0	0	0	0	0	0	0	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
CIER:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
CIOR:	0	0	0	0	0	0	0	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
CDTRSR:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
CDTRER:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
CDTROSRS:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

For each channel the following information is shown. The Channel Function Select Register (CFSR) shows the eTPU Function, the Channel Priority Register (CPR) shows the priority for that channel, the Host Service Request Register (HSRR) shows the state of the service request. The Function Mode (FM) shows the state of these two bits, the Entry shows if the event vector (entry) table for that channel is being treated as the standard or alternate event vector table.

The Channel Interrupt Status Register (CISR) shows if this interrupt has been issued by the eTPU, the CIER indicates if the interrupt is enabled and the CIOR indicates if an attempt to set the interrupt when it was already set occurred and therefore there was an overflow.

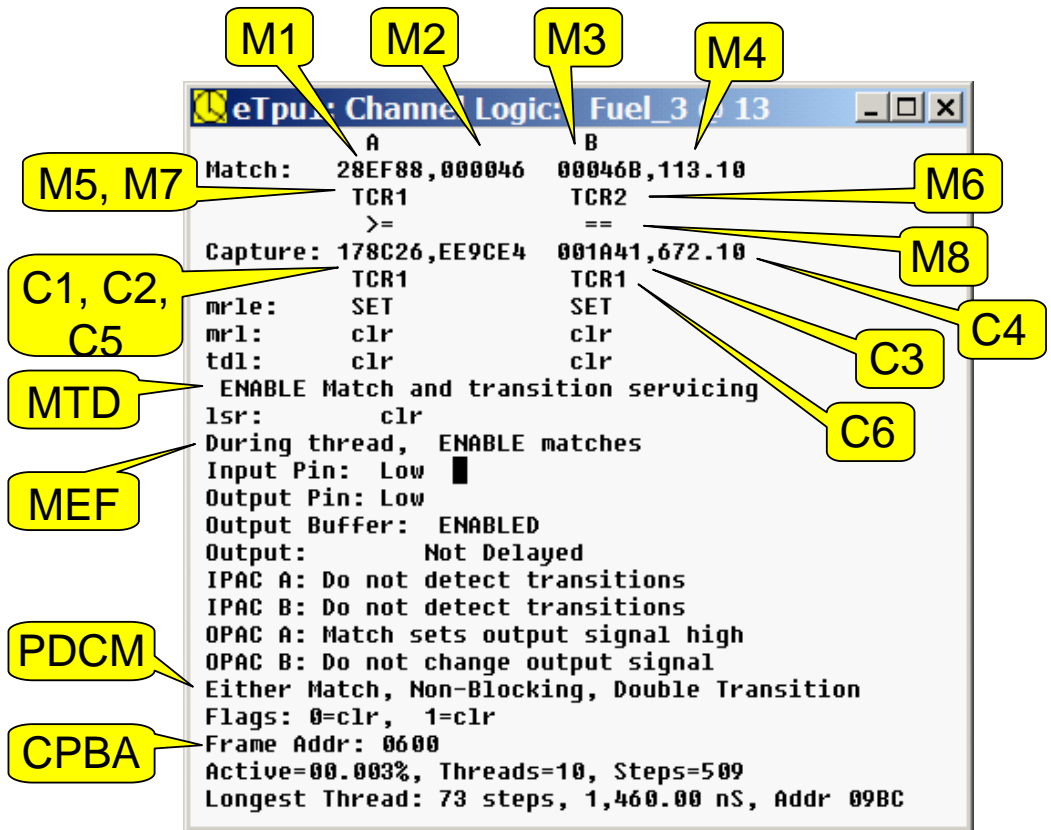
The Channel Data Transfer Request Status Register (CDTRSR) shows if this interrupt has been issued by the eTPU, the CDTRER indicates if the interrupt is enabled and the CDTROSRS indicates if an attempt to set the interrupt when it was already set occurred and therefore there was an overflow.

Note that in response to these enabled and asserted interrupts, special ISR script command files can execute as described in the Script ISR section.

### 15.2.5 eTPU Channel Hardware Window

There are two versions of the eTPU Channel window. The active channel version displays information on the active channel (or previously active channel, if none is currently active) while the fixed channel version displays information on a particular channel. Right click the mouse when inside this window to specify or change the window flavor.

When in fixed channel display mode, the window title contains the logical name for that channel. See the Test Vector "Node" Command section for assigning logical names.



M1 and M3 are the Match register values for action unit A and B, respectively.

M2 and M4 are the difference between the Match register and the active counter (TCR1 or TCR2) for which matches on that action unit are configured. When in angle mode M2 and M4 may display the angle at which the match is scheduled to occur. In order for Angle to be displayed, certain configuration must have occurred as covered in eTPU Time Base Configuration Script Commands section. Note that in the above diagram, M4 displays an angle of 113.10 degrees as a TCR2 value of 0x46B (see M3, above) corresponds to this angle.

M5 and M6 show the active counters (TCR1 or TCR2) which are used for the match comparison.

M7 and M9 show whether a match comparison has been configured for equals only (= =)

## 15. Operational Status Windows

---

or for greater than or equals ( $> = .$ )

C1 and C2 are the Capture register values for action units A and B, respectively.

C2 and C4 are the difference between the Capture register and the active counter (TCR1 or TCR2) for which the last capture into that register occurred. When in angle mode C2 and C4 may display the angle at which the capture occurred. In order for Angle to be displayed, certain configuration must have occurred as covered in eTPU Time Base Configuration Script Commands section. Note that in the above diagram, M4 displays an angle of 113.10 degrees as a TCR2 value of 0x46B (see M3, above) corresponds to this angle.

C5 and C6 show which global counter will be captured on the next MRL or TDL event.

The states of Match Recognition Latch Enable (MRLE,) Match Recognition Latch (MRL), and Transition Detection Latch (TDL) are displayed. These are the latches in the channel hardware and may not necessarily match the value in the execution unit window since those in the execution unit window are sampled at the beginning of each thread.

MTD (Match Transition Disable) indicates whether or not matches and transitions result in a service request.

The LSR field shows if there is a pending link into this channel.

MEF (Match Enable Flag) indicates if during a thread matches for the channel being serviced can be enabled or disabled. This will only be disabled if the active channel is active, and matches are disabled during the thread.

The input and output pin states are displayed as is the state of the output buffer. Although the simulator tracks the state of this buffer, there is no other affect in the simulation engine.

The input detection and output action fields show how the IPACs and OPACs have been configured.

The PDCM displays the Predefined Channel Mode for that channel.

Each channel has two flags and these are shown.

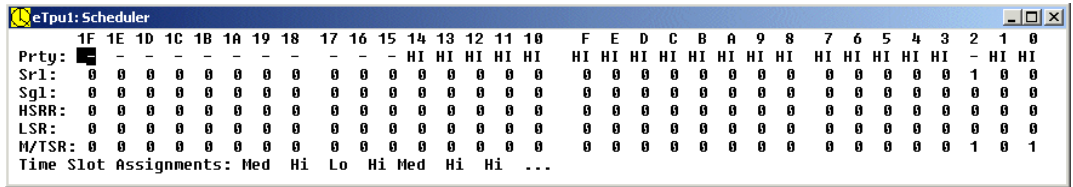
The CPBA (Channel Parameter Base Address) is shown in the Frame Address field. Note that the full address is shown rather than the CPBA value. This frame is where each channel stores its own private channel variable copies.

The channel's bandwidth is shown. Bandwidth is defined as the number of clocks in which either a thread or a Time Slot Transition (TST) for this channel was active divided by the

total number of clocks since the last reset. Additionally, the number of threads and the number of steps (instruction or NOPs) is displayed. The longest thread in both steps (instructions plus NOPs) plus the longest thread time is displayed.

Click on the longest thread address to display this location in the source code window and to show this time in the Logic Analyzer window.

### 15.2.6 eTPU Scheduler Window



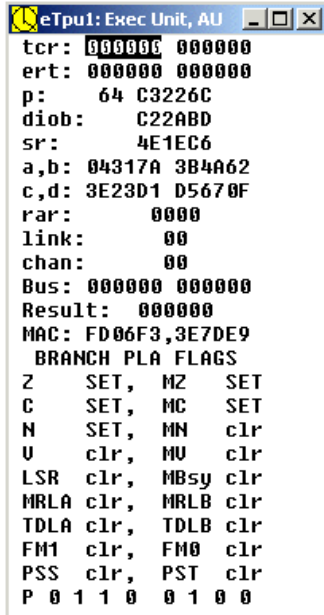
The eTPU’s microengine responds to the various channels based on a round robin scheduler. The scheduler bases its servicing decisions on the Channel Priority Register (CPR), the Service Request Latch (SRL), and the Service Grant Latch (SGL). These latches are affected by the Host Service Request Register (HSRR), the Link Service Requests (LSRs), and the Match or Transition Service Requests (M/TSRs), which are also displayed.

The M/TSR is generated from the Match Recognition Latch (MRL), Transition Detection Latch (TDL), and the Match/Transition Service Request Inhibit (SRI) latch. The M/TSR is formed from the following logical expression: [MRL and TDL] or SRI.

Each register is broken down by eTPU channel so that the value for each channel is easily found. Most of these registers can be modified only by the eTPU

## 15. Operational Status Windows

### 15.2.7 eTPU Execution Unit Registers Window



```
eTpu1: Exec Unit, AU
tcr: 000000 000000
ert: 000000 000000
p: 64 C3226C
diob: C22A8D
sr: 4E1EC6
a,b: 04317A 3B4A62
c,d: 3E23D1 D5670F
rar: 0000
link: 00
chan: 00
Bus: 000000 000000
Result: 000000
MAC: FD06F3,3E7DE9
BRANCH PLA FLAGS
Z SET, MZ SET
C SET, MC SET
N SET, MN clr
V clr, MV clr
LSR clr, MBSy clr
MRLA clr, MRLB clr
TDLA clr, TDLB clr
FM1 clr, FM0 clr
PSS clr, PST clr
P 0 1 1 0 0 1 0 0
```

The Timer Counter Global Registers (TCR1/TCR2) are displayed. The Event Register Timers for action units A and B are displayed (ERTA/ERTB.) The standard register, P, DIOB, SR, A, B, C, D are displayed. The Return Address Register (RAR), link, chan are displayed.

The execution unit's A Bus source, B Bus source, and result bus are shown. These are the buses internal to the execution unit.

The Multiply Accumulate register (MACH/MACL) are shown. The Zero, Carry, Negative, and overflow flags for both the execution unit and the MAC unit are shown (Z, C, N, V, MZ, MC, MN, M,) as is the Mac Busy flag (MBSY.)

Conditionals are displayed, as seen by the execution unit in that these are the versions of these flags that are sampled at the beginning of the thread. These include the Link Service Request (LSR), Match Recognition Latch, and Transition Detection Latch for both action units (MRLA/MRLB, TDLA/TDLB), the Function Mode Bits (FM1/FM0), the sampled and current input pin states (PSS, PST). The upper 8 bits of the P register which are treated as conditionals by the execution unit are also displayed.

# 16

## Dialog Boxes

Dialog boxes provide an interface for setting the various Simulator options.

### 16.1 File Open, Save, and Save As Dialog Boxes

The File Open, Save, and Save As dialog boxes support the opening and saving of a number of files. Each of these is described individually below.

#### **Load Executable Dialog Box**

The Load Executable dialog box controls the opening of the executable image (source code). This enables the user to change the source code, recompile the source code, and reread it into MtDt. It also allows a different executable image (source code) to be loaded into the target.

See the Source Code Files Windows section for information on viewing the executable image (source code) files.

#### **Open Primary Script File Dialog Box**

This dialog box specifies which primary script commands file is open. Only one primary

script commands file may be opened for each target at one time. The default search is \*.XXXCommand, where XXX denotes the type of target. For instance, for TPU Targets, the default search is \*.TpuCommand.

### **Save Primary Script Commands Report File Dialog Box**

This dialog box specifies the report file that is generated when the primary script commands file is parsed. Note that selection of a file name does not actually cause the report file to be written. Rather, the report file is written only when the primary script commands file is parsed.

### **Open Startup Script Commands File Dialog Box**

This dialog box specifies which startup script commands file is open. Only one startup script commands file may be opened for each target at one time. This file is very similar to the primary script file. Almost all script commands that are supported in the primary script file are also supported in the startup script file. The primary difference between primary and startup script command files is when they are executed. Startup script command files are executed only when MtDt resets a target. Primary script files execute as the target executes.

Startup script files do not support specification of a report file name. The report file name generated for the startup script is always the same as the startup script file name except the file suffix is changed to .report.

### **Open Test Vector File Dialog Box**

This dialog box is accessed via the Files menu by selecting the Vector, Open submenu. It is only available when a TPU simulation target is active.

This dialog box specifies a test vector file to be loaded. The entire file is loaded at once. Loading additional files causes existing test vectors to be removed. This dialog box allows full specification of drive, directory, and filename. The default filename is VECTORS.Vector, and the default search is \*.Vector.

### **Project Open and Project Save As Dialog Boxes**

The Project Open and Project Save As dialog boxes are opened from the Files menu by selecting the Project, Open submenu or the Project, Save submenu.

From the Project Open dialog box a Project filename is specified. MtDt loads a new

configuration from this file.

From the Project Save As dialog box a Project filename is specified. The current configuration is saved to this file.

### **Run MtDt Build Script Dialog Box**

The Run MtDt Build Script dialog box specifies which MtDt build script is to be run. Running an MtDt build script has a large impact. Before the new script is run, open windows are closed, and all target information is deleted from within MtDt. This effectively erases most of the Project information; so it is usually best to create a new project before opening a new MtDt build script.

### **Save Behavior Verification Dialog Box**

The Behavior Verification dialog box is opened from the Files menu by selecting the Behavior Verification Save submenu. This capability is only available for the eTPU and TPU simulation targets. The recorded behavior is saved to this file.

### **Save Coverage Statistics Dialog Box**

The Save Coverage Statistics dialog box is opened from the Files menu by selecting the Coverage Statistics, Save submenu. This dialog box is currently only available for TPU Simulation targets.

From the Save Coverage Statistics Dialog Box a coverage filename is specified. An overview of the coverage statistics since the last reset on both a project and a file-by-file basis is written to this file.

## **16.2 Auto Build Batch File Options Dialog Boxes**

The Auto-Build Batch File Options dialog box provides the capability of building the executable image file (source code) from within MtDt. This dialog box is opened from the Files menu by selecting the Auto-Build, Open submenu.

The auto-build batch file is a console window batch file. In this batch file the user puts the shell commands that build the executable image from the source code. MtDt executes this auto-build batch file.

The following describes the various capabilities accessed from within the Auto-Build Batch

## 16. Dialog Boxes

---

File Options dialog box.

### **Edit Window**

From within this window the user can edit the currently-selected auto-build batch file. Edits to this file are saved only if the user hits the OK button, the OK, Build button, or the Build button. If the Cancel button is hit, or if a new file is selected before one of these buttons is hit, then all edits are lost.

### **OK Button**

This saves edits, makes the currently-selected auto-build batch file the default, and closes the Auto-Build Batch File Options dialog box.

### **OK, Build Button**

This saves edits, makes the currently-selected auto-build batch file the default, and closes the Auto-Build Batch File Options dialog box. In addition, a build is performed.

### **Cancel Button**

This closes the Auto-Build Batch File Options dialog box. Edits are not saved. The default auto-build file is not set to be active for future builds. Instead, the default auto-build file reverts back to whatever the default was before the Auto-Build Batch File Options dialog box was opened.

### **Build Button**

This saves edits and performs an auto-build. The Auto-Build Batch File Options dialog box is not closed.

### **Help Button**

This accesses this help menu.

### **Change File Button**

This allows the user to select a new auto-build batch file. Beside this button is listed the name of the currently-selected auto-build file.

## 16.3 Goto Time Dialog Box

The Goto Time Dialog Box is opened via the Run menu by selecting the Goto Time submenu. It provides the capability to execute MtDt until a user-specified time.

There are two types of Goto time options, one of which must be selected.

### **Goto Until Time**

This sets MtDt to go to an absolute (simulation) time. The simulation time is initially set to zero. The simulation time is reset to zero via the Run menu by selecting the Reset submenu.

### **Goto Current Time, Plus**

This sets MtDt to go to the current (simulation) time plus some user-specified additional time.

User-specified time is entered as thousands of seconds (ksec), seconds (secs), milliseconds (ms), microseconds (us), and nanoseconds (ns). MtDt's resolution is two CPU clock cycles. For the 16.778 MHz (two to the 24th cycles per second) CPU clock, this equates to approximately 124 nanoseconds.

### **Help**

This accesses this help window.

### **Goto**

This closes the Goto Time dialog box and runs MtDt until the specified time.

### **OK, Save**

This closes the Goto Time dialog box and saves any changes. MtDt remains idle.

### **Cancel**

This closes the Goto Time dialog box without saving any changes.

### 16.4 Goto Angle Dialog Box

The Goto Angle Dialog Box is opened via the Run menu by selecting the Goto Angle submenu. It provides the capability to execute the active target until it gets at or beyond the specified angle. The eTPU must be in angle mode (AM=1) in order for this to function properly.

This dialog box uses the TCR2 counter, and user-defined angle indices to calculate the angle. See the eTPU Time Base Configuration Script Commands section for setting the angle indices.

The 'Cycles' field refers to the number of times the angle has rolled over. For example, in an automobile engine, two rotations constitutes one cycle. The angle therefore goes from 0 degrees, to 720 degrees, then rolls over back to zero degrees. 'Cycles' is the count of these rollovers.

The current angle can be seen in the status bar at the bottom of the IDE. See the IDE Options Dialog Box section for information on enabling this feature.

#### **Goto Until Angle**

This sets the active target to go to an absolute (simulation) angle. The simulation angle is initially set to zero. The simulation angle is reset to zero via the Run menu by selecting the Reset submenu.

Note that if the desired stop cycle and angle has already been traversed, then the Cycles field is ignored and the simulation is halted the next time the specified angle is traversed, in either the current or next cycle.

#### **Goto Current angle, Plus**

This sets the additional angle to which the active target will be run. The angle to which the active target will run is the current angle plus the specified delta angle.

#### **Help**

This accesses this help window.

#### **OK**

This closes the Goto Angle dialog box and runs the simulator until the simulator is at or beyond the specified angle in the active target.

#### **Cancel**

This closes the dialog box without saving any changes.

## 16.5 Occupy Workshop Dialog Box

The Occupy Workshop dialog box provides the capability of specifying for individual windows which workshop(s) the window will be visible.

### **OK**

This closes the dialog box and saves any changes.

### **Cancel**

This closes the dialog box and discards all changes.

### **Help**

This accesses this help window.

### **Occupy All**

This causes the window to be visible in all workshops.

### **Leave All**

This causes the window to not be visible in any workshop. This should be treated as a shortcut for clearing all selections. Note that this is not the same as closing the window as the window will still exist within MtDt.

### **Revert**

This causes any settings made since the dialog box was opened to be discarded.

### **Options**

This opens the Workshop Options dialog box.

## 16.6 IDE Options Dialog Box

The following describes the IDE Options dialog box.

### All Targets Settings

The following settings are for all targets.

### **Change the Application Wide Font**

Changes the fonts used by all the windows.

### **Update Windows While Target is Running**

This specifies whether the windows are continually updated as target runs. The window update generally takes well under 1% of the application's CPU time so this option is generally selected. But in certain cases deselection of this option can improve performance.

### **View Toolbar**

This specifies whether the toolbar and its associated buttons are visible. The toolbar is the gray area located at the top of MtDt's application window.

### **View Status bar**

This specifies whether the status bar and its associated indicators are visible. The status bar is the gray area located at the bottom of MtDt's application window.

### **Pop to Halted Target's Workshop**

Each target is generally assigned to a workshop. In a multiple target environment when a target halts the workshop associated with the target that caused the system to halt. This should generally be left selected.

### **Auto-Open Window for File of Active-Target's Program Counter When Target Halts**

When this is selected, if the target halts the source code file corresponding to the program counter of the active target is closed, it will automatically be opened. In addition, the current line is automatically scrolled into view.

### **Display Angle (Instead of Clocks) if Available**

In the status bar there is a box that shows either the angle or time of the active target. This setting specifies that angle is shown rather than time, if it is available. The display has the format of XC Y.Z, where X is the number of cycles and Y.Z is the current angle in degrees. Note that determination of the engine angle relies on certain timing settings specified using the `set_angle_indices()`; script command which is described in the eTPU Time Base Configuration Script Commands section.

### **Use the GNU CPP Preprocessor for Script Commands Files**

The GNU Preprocessor supports enhanced preprocessing and can be used to initialize global variables in the eTPU. See the GNU CPP Preprocessor section for more details.

#### **eTPU and TPU Simulator Target Only**

The following settings are only available for eTPU and TPU Simulator targets.

#### **View Coverage (TPU Targets Only)**

This specifies whether the code coverage indicators are visible within source code windows. These are the color coded boxes at the far left of each line of text that is associated with an instruction. These boxes indicate if the associated instruction has been executed, and, if it is a branch instruction, if the true and false cases have been traversed.

#### **Active Target Settings**

The following settings act only on the target that was active at the time that the dialog box was opened. Each target can be individually set.

#### **Source Code Spaces per Tab**

Specifies the number of spaces that correspond to each tab character.

## **16.7 Workshop Options Dialog Box**

The Workshops Options dialog box is used to associate targets with workshops, specify workshop names, and place workshop buttons in the toolbar.

When adding a new target association to a workshop you will be prompted to indicate whether you would like all windows belonging to that target to be made visible within that workshop. It is generally desirable to select the "yes" or "yes to all" option.

The very first workshop is special in that all windows initially visible within this workshop. This prevents windows from becoming lost. To avoid confusion, this workshop name is always "All."

This dialog box allows you to remove a target association from a workshop. In this case you will also be prompted to indicate whether you would like to remove visibility of all windows belonging to the removed target from the affected workshops.

### On Toolbar

Checking this option causes a button to appear on the toolbar that automatically switches to that workshop when depressed.

### Workshop Name

This is the name of the workshop. You can either type in a new name using the keyboard; alternatively a button to the left of this edit control allows you to assign the associated target's name to the workshop.

### Primary Association

This is the target that is associated with the workshop. A dropdown list allows selection of any target.

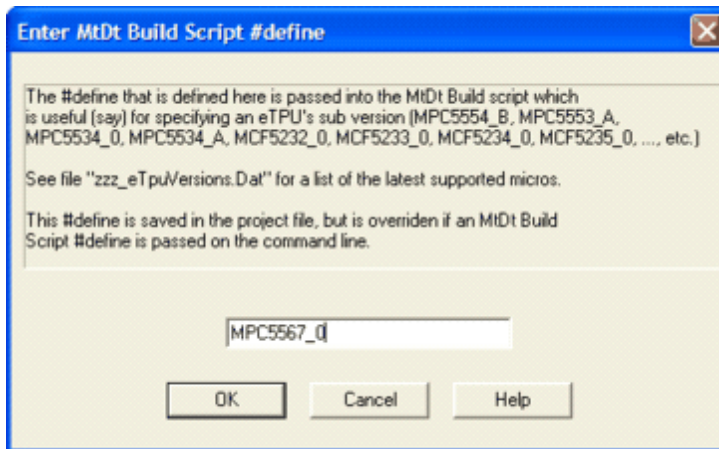
## 16.8 MtDt Build Options Dialog Box

This MtDt Build Options Dialog Box is opened from the Options Menu. It is used to specify options for the MtDt Build Script. The MtDt Build Script is run by the ASH WARE simulator or debugger when the application is initially launched to build the simulation or debugging environment as described in the System Simulation section.

An example usage of this would be to select an MCF5232 revision 0 eTPU (6K should 1.5K RAM) instead of the default MPC5554 revision B (16K code, 3K RAM.) This is shown below.

See file “zzz\_eTpuVersions.Dat” in the BuildScripts directory for definitions of all supported microcontroller versions that contain an eTPU. Note that Freescale constantly develops new microcontroller derivatives that contain eTPU's. The following is a list of all supported version at the time of this writing:

```
// JPC/SPC (eTPU-2's)
JPC563M60_1, SPC563M54_0, SPC563M60_0, SPC563M64_0,
// MPC56xx Family (eTPU-2's)
MPC5632M_0, MPC5633M_0, MPC5634M_0, MPC5674_0, MPC5674_2
// MPC55xx Family
MPC5534_0, MPC5553_A, MPC5554_B, MPC5534_A, MPC5565_0,
MPC5566_0, MPC5567_0
// MCF52xx Family
MCF5232_0, MCF5233_0, MCF5234_0, MCF5235_0, MCF5271_0
```



In order for any changes to take affect, the application must be closed and re-opened.

### **#define**

Used to specify a #define that is passed to the MtDt Build Script. Note that this setting is overridden if this same #define is passed on the command line when the application is launched. See the Command Line Parameters section.

### **Related Topics**

See the Building the MtDt Environment section

See the Command Line Options section.

## **16.9 Message Options Dialog Box**

The Message Options dialog box provides the capability to disable the display of various messages. These messages warn the users in a variety of situations such as a failed script verification command failure or when suspicious code is encountered.

When there is a check next to the described message, a message is generated when the described situation occurs. To disable the message, remove the check. The user might wish to execute through a failing or suspicious section of code without having to acknowledge each message and would therefore disable the associated message.

### **All Targets Messages**

#### **Behavior Verification Failure**

Behavior verification failure messages are generated when the current execution does not match that from a previously-recorded behavior verification file.

### **Script Verification Failure**

Script verification failure messages are generated when a user-defined script verification test fails.

### **Bad at\_time() Script Command**

Bad at\_time() script commands specify when a particular script command is to be executed. A message appears when this command specifies a time earlier than the current time.

### **Obsolete set\_cpu\_frequency Script Command**

The set\_cpu\_frequency script command is being deprecated and users should switch to the set\_period script command to achieve this functionality. The problem with the set\_cpu\_frequency command is that the simulation engine now calculates all times in periods rather than frequency. This command requires an inversion which can generate extremely small error, and this error over many billions of simulation cycles can become significant. Consider the case of two targets, where one target is running half the frequency of the other. If one clock is 333 MHz and the other is 666 MHz, after the inversion, the resulting clock periods may not be exactly double anymore.

## TPU Messages

### **An Instruction Accessed Unimplemented Parameter RAM**

The message that reports an access of un-implemented parameter RAM locations may be disabled. Some users have taken advantage of the un-documented feature that accesses to these un-implemented parameter RAM locations return zero. Taking advantage of this un-documented feature is dangerous because Freescale might choose to implement these locations, such as in the TPU2, or might change the values returned when these locations are accessed. In any case, these warning messages can be disabled.

### **WMER Instruction at N+1 after CHAN\_REG Change**

The TPU behavior in this situation is undefined so MtDt generates a message.

### **READ\_MER Instruction at N+2 after CHAN\_REG Change**

The TPU behavior in this situation is undefined so MtDt generates a message.

### **Subroutine Return from Outside a Subroutine**

When the TPU performs a subroutine call, the calling address is pushed unto the stack. It would be legal but highly suspicious to do multiple subroutine returns following a single subroutine call.

### **Sequential TCRx Write than Read**

This is actually legal, but it is unlikely that it does what you want it to. You see, the read finds the pre-written value having to do with some usual TPU craziness.

### **Entry Bank Register Accessed Multiple Times**

In the real TPU the register holding the entry bank number can be written only once. The TPU Simulator allows this register to be written multiple times but gives a warning if you choose to do so.

### **Address Rollover Within a Bank**

TPU2 and TPU3 support multiple banks. TPU behavior is not defined if a non-end and non-return last bank instruction is executed.

## **16.10 Source Code Search Dialog Box**

The Source Code Search Options dialog box is opened from the Options menu by selecting the Source Search submenu.

When the executable image is loaded, there are normally a number of source code files associated with the executable image that get loaded. MtDt needs to be able to find these files. This dialog box allows specification of source code directories to be searched when searching for these source code files.

The search locations can be specified for each individual target, and for all targets globally. Specifying global search options is useful in situations in which multiple targets are using the same directories for their library files.

When searching for a source code file, the following algorithm.

- If the path to the file is fully-specified (e.g. c:\SomeDir\SomeFile.c) use that if the file exists there.
- If the path is partially specified (e.g. ..\OneUpDir\SomeFile.c) resolve the path

## 16. Dialog Boxes

---

relative to the code image file. For instance, if the code image file is at c:  
\SomeDir\SubDir\CodeImage.Elf, then check for this file c:  
\SomeDir\OneUpDir\SomeFile.c

- If the source file is still not found, use the raw file name (strip any prepended directory information) and search in the directory where the executable image file is located.
- If the source file is still not found, use the raw file name (strip any prepended directory information) and search for the file in the directory(s) listed in the 'Selected Targets' directory search list, starting from the top-listed directory.
- If file is still not found, use the raw file name (strip any prepended directory information) and search for the file in the directory(s) listed in the 'All Targets' directory search list, starting from the top-listed directory.

### **Add**

This button inserts a new directory into the search list.

### **Modify**

This button modifies a previously entered search location.

### **Delete**

This button removes a location from the search list.

### **Cut**

This button removes the currently selected search location from the search list, and places it into the paste buffer.

### **Copy**

This button adds the currently selected search location to the paste buffer without removing it from the search list.

### **Copy**

This button creates a new search location using the paste buffer.

### **Move Up**

This button moves the currently selected search location higher in the search list such that

MtDt searches this location earlier.

### **Move Down**

This button moves the currently selected search location lower in the search list such that MtDt searches this location earlier.

## **16.11 Reset Options Dialog Box**

The Reset Options dialog box is accessed from the Options menu by selecting the Reset submenu. It specifies the actions taken when either the Reset submenu or the Reset and Go submenu is selected.

### **Reread Primary Script Commands File**

Selecting this option causes the primary script commands file to be reread every time MtDt resets the targets.

### **Rewrite Code Image**

Selecting this option causes a cached version of the executable image to be re-written every time MtDt resets the targets. The executable image file is not reread, instead, a cached version of the executable image is used. Since a cached version is used, source code files are also not changed. Note that this option should be selected with care as the executable image in the target should generally not be modified during target execution and reloading every reset could mask such a bug.

### **Reread Test Vector File (eTPU and TPU Targets Only)**

Selecting this option causes the test vector file to be reread every time MtDt resets the targets.

### **PRAM Variable Memory (eTPU and TPU Targets Only)**

This option specifies whether the PRAM memory is not changed, written to all zeroes, or is randomized on reset.

### **TCR1 and TCR2 Counters (eTPU and TPU Targets Only)**

This option specifies whether the TCR1 and TCR2 counters are not changed, written to zero, or are randomized on reset.

### **Help**

## 16. Dialog Boxes

---

This accesses this help menu.

### **Cancel**

This discards and changes and exits the Reset Options dialog box.

### **OK**

This saves any changes and exits the Reset Options dialog box.

## 16.12 Logic Analyzer Options Dialog Box

The Logic Analyzer Options Dialog box defines the settings associated with the Logic Analyzer Logic Analyzer Window.

### **Log TCRCLK (eTPU) and TCR2 (TPU) Pin Transition**

This controls whether the TCRCLK/TCR2 input pin transitions are logged to the data storage buffer. This pin can be used to clock and/or gate in the eTPU and TPU. Disabling this increases the effective data storage buffer size.

### **Log TCR1/TCR2 Counter Transitions**

The least significant bit of the TCR1 and TCR2 global counters can be logged, and thereby displayed as a waveform in the logic analyzer. Disabling this increases the effective data storage buffer size.

### **Log Angle Mode Transitions.**

Various aspect of angle mode can be displayed in the logic analyzer as a waveform. These include the mode (high-speed, Normal, Wait), angle ticks, synthesized PLL tooth ticks, etc. Disabling this increases the effective data storage buffer size.

### **Log Threads**

Threads can be grouped into thread groups, and the thread group activity can be displayed in the logic analyzer window. Disabling this increases the effective data storage buffer size.

### **Configure Thread Groups**

There are eight thread groups labeled from 'A' to 'H'. This opens the Thread Group Dialog Box `_CHAN_GROUP` allows configuration of which thread(s) are associated with each of

these groups.

### **Time Display**

The Logic Analyzer can base the timing display on either target clock count or Simulator time. Both of these are set to zero when MtDt is reset. This field allows the user to select between the two display options.

### **Target Clocks Display**

See the above description from the time display.

## **16.13 Channel Group Dialog Box**

The Channel Group Options Dialog box is used to select groups of one or more channels. It is accessed in different locations for different purposes.

This dialog box is accessed from the Logic Analyzer Options Dialog box to specify groups for monitoring of eTPU thread activity. Note that thread group activity is displayed as a waveform in the Logic Analyzer.

This dialog box also is used by the Complex Breakpoint Conditional Dialog to select which channels a complex breakpoint conditional acts upon.

## **16.14 Complex Breakpoint Conditional Dialog Box**

The complex breakpoint conditional dialog box allows the user to add conditionals to complex breakpoints. Each complex breakpoint must contain one or more conditionals. This dialog box is accessed from the complex breakpoints window.

## **16.15 Trace Options Dialog Box**

### **Instruction Execution**

Selecting this option causes each instruction execution to be logged to the trace buffer.

### **Instruction Boundary**

Selecting this option causes each instruction boundary to be logged in the trace buffer.

## 16. Dialog Boxes

---

While an instruction boundary contains no useful information, the resulting dividing line makes the trace window easier to read.

### **Memory Read**

Selecting this option causes each memory read to be logged to the trace buffer.

### **Memory Write**

Selecting this option causes each memory write to be logged to the trace buffer.

### **Exception**

Selecting this option causes each exception to be logged to the trace buffer. This is only meaningful in the context of CPU targets.

### **Time Slot Transition**

Selecting this option causes each time slot transition to be logged to the trace buffer. This is only meaningful in the context of TPU targets.

### **State End**

Selecting this option causes state end to be logged to the trace buffer. This is only meaningful in the context of TPU targets.

### **Pin Transition**

Selecting this option causes each pin transition to be logged to the trace buffer. This is only meaningful in the context of TPU targets.

### **SGL Negation NOP**

Selecting this option causes SGL negation NOP to be logged to the trace buffer. This is only meaningful in the context of TPU targets.

## Trace Window and Trace Buffer Considerations

The Trace Options Dialog Box specifies what is stored in the trace buffer. The Trace Window displays all trace information stored in the trace buffer regardless of whether or not this information is enabled for storage. If information has already been stored in the buffer and you disable tracing of this information the information will not disappear from the Trace Window until the buffer has been completely overwritten with new information, or until the buffer is flushed following a reset.

## Multiple Target Considerations

Individual trace settings are maintained for each target. In a multiple target environment it is possible to enable trace settings in one target and disable these same settings in another target. The target on which the Trace Options Dialog Box acts is listed at the top of the dialog box. This corresponds with either the active target at the time that the dialog box is opened or the target associated with the active window.

### 16.16 Local Variable Options Dialog Box

It is important to note that the Local Variables window automatically tracks the current function. When the target transitions from running to stopped, the active function's local variables are automatically displayed. But in addition, when you scroll through the Call Stack window, the Local Variable window automatically displays the stacked local variables associated with selected line in the Call Stack window. This automatic behavior is generally quite useful but in certain cases it can get in the way. By locking the local variable window you are able to disable this automation.

#### **Lock the local variables from <functionName> at stack frame <frameAddress>**

When this is checked, the Local Variable window displays a particular function's local variables. The normally automatic tracking of the current function's local variables is disabled. For example if you are parsing through text, a stacked function may point to the beginning of a buffer that is being traversed by called functions. Use of this option would allow you to lock onto display of the calling function's local variables thereby viewing a reference to the beginning of the buffer.

Note that the function name and frame address are determined by the window's current display state. The functionName is a symbolic name of a function, e.g. main(char argc, char \*argv[]). The frame address is the address such as 0x1000.

#### **Maximum Expansion Level for Structure Members, Pointers, etc**

It is common for structures to be members of other structures or to be referenced from within structures. The local variable window expands these contained and referenced structures. This setting specifies the number of levels to expand.

#### **Maximum Number of Array Elements to Display**

This specifies the number of local variable array members that are displayed. For example an integer array might consist of multiple members but the actual number of members is

## 16. Dialog Boxes

---

generally not available in the symbol table. This setting allows you to specify how many elements to display.

### 16.17 License Options Dialog Box

This dialog box allows you to enter additional information prior to sending a license file to ASH WARE Inc. A license file is generated whenever you install an ASH WARE product. Unfortunately, the license file generated at install time contains very little information other than a computer identifier. This dialog box allows you to add additional information such as your name, purchase order number, etc.

The license file has been a problem for ASH WARE in that users have sent in license files for purchased products but we were unable match the license files with the purchase. This dialog box is intended to reduce this confusion, thereby allowing us to serve you better.

All information is optional. Generally, it is best to include at least your company's purchase order number or the ASH WARE invoice number, if available.

### 16.18 Memory Tool Dialog Box

This tool supports specialized memory functions listed below. The dump file functions are also accessible from the `dump_file` script command listed in the File Script Commands section.

- [Fill memory with data or text](#)
- [Search for data or text](#)
- [Dump to disassembly file](#)
- [Dump to Motorola SRecord \(SREC\) file](#)
- [Dump to Intel Hexadecimal \(IHEX\) file](#)
- [Dump to image file](#)
- [Dump to "C" structure file](#)

For each function the address space and memory range can be specified. Earlier address ranges are stored in a buffer and can be retrieved using the recall button. For the file-dump options, selecting the change button can specify the file name.

In disassembly dump files inclusion of address, raw values, addressing mode information,

and symbolic information can be selected.

When creating an image file or a "C" data structure file, or when using the fill function, the data word size can be 8, 16, or 32-bits. For the fill function, verification after the fill can be selected.

The find capability allows a specific byte pattern to be located in memory. Options include the ability to search for between one and eight sequential bytes, as well as the ability to search for both a case sensitive or case-insensitive string.

"C" data files can be written with the address included within a comment. Output format can be either hexadecimal, which is the default, or decimal.

Selection of endian ordering is available for where the data size exceeds one byte. This option is available when dumping to an image or a "C" structure file or when using the find function. In cases where the selected endian ordering does not match the natural endian ordering of the target a warning is displayed.

## 16.19 Insert Watch Dialog Box

The Insert Watch dialog box is opened from within the Watch Window using the insert function keys.

This dialog box contains three lists.

The left-most list of all watches that are currently in use. This is helpful when a new watch is desired that is similarly spelled as an existing watch.

The middle list contains all recently used watches, even ones that are no longer in use.

The right-most list contains a list of the print action command and timing action commands. This is special tagged text in source code used to generate formatted output to trace files (think printf) and for verifying that minimum and maximum timing criteria are met for traversal of named timing regions.

### Related Information

- Naming timing regions in source code
- Verifying traversal times a script command file
- View named timing regions timing using the Watch Window
- List named timing regions in the Insert Watch Dialog Box

### 16.20 Watch Options Dialog Box

The Watch Options dialog box is currently featureless. This dialog box is provided to maintain forward-compatibility with future versions of this software.

### 16.21 The 'About' Dialog Box

The eTPU Simulator (C) 1994-2012 ASH WARE, Inc. All rights are reserved. Various national and international laws and treaties protect these rights. Any misuse or other violation of the copyright will be prosecuted to the full extent of the law.

MtDt may not be copied except for the purpose of creating a backup copy for archival purposes.

# 17

## Menus

The menus allow the user to access the various capabilities of the eTPU Simulator. The menus appear at the top of the eTPU Simulator. Pointing the arrow at the menu by moving the mouse and then clicking the left mouse button accesses them. They are also accessed by the <ALT-F10> hot keys.

The menu structure is organized as a series of submenus within each menu. When a menu item is clicked, the submenu structure pops up into view. Clicking on the corresponding items can then access a submenu selection.

### 17.1 Loading Files Menu

#### **The Executable, Open submenu**

This opens the Load Executable dialog box.

#### **The Executable, Fast Submenu**

This provides a fast repeat capability for the Load Executable dialog box. By selecting this submenu the actions set up in the Load Executable dialog box are quickly repeated without having to actually open the dialog box.

#### **The Primary Script, Open Submenu**

This opens the Open Primary Script File dialog box.

### **The Primary Script, Fast Submenu**

This provides a fast repeat capability for the Open Primary Script File dialog box. By selecting this submenu the actions set up in the Open Primary Script File dialog box are quickly repeated without having to actually open the dialog box.

### **The Primary Script, Report Submenu**

This opens the Save Primary Script Commands Report File dialog box. This report file is generated when the primary script commands file is parsed.

### **The Startup Script, Open Submenu**

This opens the Open Startup Script Commands File dialog box. This script commands file is executed after MtDt resets its targets.

### **The Vector, Open Submenu**

This opens the Open Test Vector File dialog box.

### **The Vector, Fast Submenu**

This provides a fast repeat load test vector file capability. By selecting this submenu the last loaded test vector file is reloaded.

### **The Project, Open Submenu**

The Project Open submenu opens the Project Open dialog box. Each dialog box provides a project session capability where MtDt settings are associated with Project files. This submenu allows the user to open a previously-saved project session.

### **The Project, Save As Submenu**

The Project Save As submenu opens the Project Save As Dialog Box. Each dialog box provides a project session capability where MtDt settings are associated with Project files. This submenu provides the capability to both create a new Project file and overwrite an existing project file with the currently-active configuration.

### **The MtDt Build Script, Run Submenu**

This opens the Run MtDt Build Script Commands File dialog box and runs the selected MtDt build script commands file. This action has a large impact on MtDt so care must be taken when selecting it.

### **The MtDt Build Script, Fast Submenu**

This provides a fast repeat run MtDt build script commands file. By selecting this submenu the last run MtDt build script file is re-run. This is useful when debugging a custom MtDt build script file.

### **The Auto-Build, Edit Submenu**

This opens the Auto-Build Batch File Options dialog box. for selection and editing of the auto-build batch file.

### **The Auto-Build, Fast Submenu**

This provides a fast repeat build capability. By selecting this submenu or using the <CTRL A> hot key the default auto-build batch file is executed.

### **The Behavior Verification, Open Submenu**

This opens the Open Behavior Verification dialog box. This is used to load previously saved behavior files. NOTE: This option only exists prior to version 5.00. With 5.00 and newer behavior verification files can only be loaded through scripting.

### **The Behavior Verification, Save Submenu**

This opens the Save Behavior Verification dialog box. This is used to save the recorded Simulator behavior into a behavior verification file. NOTE: This option only exists prior to version 5.00. With 5.00 and newer behavior verification files can only be saved through scripting.

### **The Coverage Statistics, Save Submenu**

This opens the Coverage Statistics dialog box. This is used to store a coverage statistics report to a file.

### **Help**

This accesses this help screen.

## **17.2 Stepping MtDt**

### **Into**

This runs the active target until one line of source code is executed. If a function is called,

## 17. Menus

---

MtDt halts on the first instruction within that function. If no instructions associated with source code lines occur, MtDt continues to run until stopped by selecting the Stop submenu in the Run menu.

### Over

This single steps the target by one line of source code, stepping over any function call. This is the same as the above "Into" function except the "Into" function will halt on a line of source code within the function that is called. If no instructions associated with source code lines occur, MtDt continues to run until stopped by selecting the Stop submenu in the Run menu.

### Out

This runs the active target until the current function returns. Execution is halted on the next line of source to be executed in the calling function. If no instructions associated with source code lines associated with a calling function occur, MtDt continues to run until stopped by selecting the Stop submenu in the Run menu.

### Anything

This causes one action to be performed. This action may be execution of a single instruction, execution of a script command, or simply a tick of the CPU clock. This is helpful for advancing execution by as small an amount of possible, allowing you to really zoom in on a problem.

### Script

This runs MtDt until one script command from the active target is executed. If no new script commands become available MtDt continues to run until stopped by selecting the Stop submenu in the Run menu.

### Thread Start (Time Slot)

Runs until the beginning of the next eTPU/TPU thread (time slot transition.) Execution stops just prior to execution of the first opcode in the thread. If no thread occurs, execution continues to run until stopped by selecting the Stop submenu in the Run menu.

### Thread End

This command is excellent for running from thread-to-thread in the same channel.

- If in a thread, run, then stop at the end of the thread.
- If at the end of a thread, run until the beginning of a thread on the same

channel.

- If no thread is active, run until the beginning of a thread on the last-active channel.

### **Angle Tick**

When the eTPU is in angle mode, this runs until the next angle tick occurs.

### **Angle Tooth**

When the eTPU is in angle mode, this runs until the angle tooth occurs. The angle tooth could be generated by a physical tooth, a tooth induced by asserting TPR.IPH, or by the EAC decrementing TPR.MSCNT.

### **Assembly**

This runs the active target until a single assembly instruction occurs. If instructions occur, MtDt continues to run until stopped by selecting the Stop submenu in the Run menu.

### **Assembly, N**

This runs the active target until a user-specified number of assembly instructions occur. A dialog box opens allowing specification of the desired number of assembly instructions. If no assembly instructions occur, MtDt continues to run until stopped by selecting the Stop submenu in the Run menu.

### **Help**

This accesses this help screen.

## **17.3 Running MtDt**

### **Goto Cursor**

This runs MtDt until an instruction associated with the current cursor location is about to be executed or a breakpoint is encountered. A source code window must be active for this command to work.

### **Goto Time**

This opens the Goto Time dialog box. Runs MtDt until the specified time or until a breakpoint is encountered.

### **Goto Time, Fast**

This behaves exactly like the Goto Time submenu, except the goto time dialog box is not opened. Instead, the previously specified Goto Time options are used.

### **Go**

This causes MtDt to execute indefinitely. MtDt executes until a breakpoint is encountered or until the user terminates execution by selecting the Stop submenu in the Run menu.

### **Stop**

This causes MtDt to stop executing. This is normally used to stop MtDt when the expected event (such as step, breakpoint, time slot transition, etc.) failed to occur.

### **Reset and Go**

This causes MtDt to reset and immediately begin execution. The reset actions are specified in the Reset Options submenu. MtDt will execute until a breakpoint is encountered or until interrupted by the user selecting the Stop submenu from the Run menu.

### **Reset**

This causes MtDt to reset. The reset actions are specified in the Options submenu in the Reset menu.

### **Help**

This accesses this help screen.

## 17.4 Breakpoints

The breakpoints menu controls the various functions associated with the breakpoint capabilities of the MtDt. These capabilities are accessed via the following submenus.

### **Set (Toggle)**

This toggles a breakpoint at the source code window's cursor. If the instruction already has a breakpoint, then the breakpoint is deleted. A source code window must be active for this to work. If the source code is in mixed assembly view mode, and the cursor is at a dis-assembly line, then the breakpoint will be generated for only that address.

**Disable All**

This disables all active breakpoints. Disabled breakpoints remain disabled. MtDt remembers which breakpoints have been disabled, such that the Enable Breakpoints submenu can reactivate all disabled breakpoints.

**Enable All**

This enables all disabled breakpoints.

**Delete All**

This deletes all active and all disabled breakpoints.

**At Address ...**

This opens a dialog box which allows placing a breakpoint any user-specified address. This is useful when debugging code for which there is no line number information, such as GNU assembly code.

**Delete All Address ...**

This deletes all breakpoints for which there is no associated source code.

**Delete All Script**

This deletes all breakpoints that are in the script commands file.

**All Targets Sub Menu**

This is only available in multiple target simulators. It performs the Disable, Enable, and Delete Breakpoint functions for all targets. The default is for just the active target.

**Help**

This accesses this help screen.

## 17.5 Workshops

The Activate menu allows you to specify which workshop and target are active. These capabilities are accessed via the following submenus.

**Workshop**

## 17. Menus

---

This displays a list of workshops that can be activated. Activation of a workshop causes all windows within that workshop to be displayed, and the target associated with that workshop, if any, to be activated. Activation of an associated target upon workshop activation is a feature that can be disabled within the Workshop Options dialog box.

### Target

This displays a list of targets that can be activated. The active target is the one used when target-specific functions such as single stepping are selected. This feature is useful only in a multiple target environment.

### Help

This accesses this help screen.

## 17.6 Opening View Windows

This menu opens the various Simulator windows for viewing. MtDt allows multiple instances of each window to be open simultaneously. When there are multiple targets a submenu for each target appears. Otherwise, all available windows are available directly within the view menu.

See the Operational Status Windows chapter for a listing of the available windows for each target.

### Help

This accesses this help screen.

## 17.7 Window Options Menu

The Window menu accesses the four standard windows functions: Cascade, Tile, Arrange Icons, and Close All. The standard Windows size toggle function switches the window between maximized and normal size. In addition there is a list of all open windows. Selecting an open window from this list causes that window to pop to the front.

### Occupy Workshop

This opens the Occupy Workshop dialog box. This allows display of the currently active window within the various workshops to be enabled and disabled.

### **Redraw All**

This causes all windows to be redrawn. In addition, all windows caches are invalidated, thereby forcing MtDt to go out to the hardware (for hardware targets) to refresh window data. This is helpful on hardware targets for updating the display of hardware registers that may be changing even while execution is halted.

### **Help**

This accesses this help screen.

## **17.8 Setting Simulator Options**

The Options menu provides the user with the capability of setting various Simulator options. These are listed below.

### **IDE**

This opens the IDE Options dialog box.

### **Workshop**

This opens the Workshops Options dialog box.

### **Messages**

This opens the Message Options dialog box.

### **Reset**

This opens the Reset Options dialog box. It specifies the actions taken when MtDt is reset from the Reset menu by selection of the Reset submenu.

### **Logic Analyzer**

This opens the Logic Analyzer Options dialog box. Note that a Logic Analyzer window must be active for this submenu item to be available.

### **Memory Tool**

This opens the Memory Tool dialog box. Note that if a Memory Dump Window is selected when this dialog box is opened, settings from the window are automatically loaded into the dialog box. This supports a variety of capabilities including dumping memory to files,

## 17. Menus

---

searching for patterns in memory, and filling memory.

### **Memory**

This lists a variety of settings available for a Memory Dump Window. Note that this type of window must be selected for these options to be available.

### **Toggle Mixed Assembly**

This toggles the visibility of assembly within Source Code File windows. Note that this type of window must be selected for these options to be available.

### **Verify Recorded Behavior**

This verifies the recorded behavior against the currently active master behavior verification file. See the Pin Transition Behavior Verification section for a description of this capability. NOTE: This option only exists prior to version 5.00. With 5.00 and newer behavior verification can only be controlled via scripting.

### **Enable Continuous Verification**

This enables continuous verification of behavior against the currently active master behavior verification file. This provides immediate feedback if the microcode behavior has changed. See the Pin Transition Behavior Verification section for a description of this capability. NOTE: This option only exists prior to version 5.00. With 5.00 and newer behavior verification can only be controlled via scripting.

### **Disables Continuous Verification**

This disables continuous verification of behavior against the currently active master behavior verification file. This prevents a large number of verification errors from being displayed if microcode behavior has changed significantly. See the Pin Transition Behavior Verification section for a description of this capability. NOTE: This option only exists prior to version 5.00. With 5.00 and newer behavior verification can only be controlled via scripting.

### **Timer**

This lists a variety of options available for timers. These options are only available if a Source Code File window is selected. Options include setting the timer's start or stop address to be equal to that associated with the cursor location within the Source Code File window. A timer can also be enabled or disabled.

### **Watch**

This lists a variety of options available for a watch. These options are available only if a Watches window is selected. Options include inserting and removing a watch, moving a watch up or down, and setting the options for a specific watch.

### **Help**

This accesses this help screen.

## **17.9 Help Menu**

The Help menu provides the following options.

### **Contents**

Accesses the contents screen of MtDt's on-line help program.

### **Using Help**

Accesses the standard Microsoft Windows "help on help" program.

### **Latest Enhancements**

This accesses information on the enhancements added in the various versions of this software.

### **Technical Support**

This accesses information on obtaining technical support for this product.

### **About**

This gives general information about MtDt.



# 18

## Hot Keys, Toolbar, Status Window

Pointing and clicking with the mouse activate Toolbar buttons. The toolbar is located toward the top of the main screen.

Toolbar buttons also allows you to quickly switch between workshops. These are the text buttons located at the top right of the main screen.

An execution status indicator appears at the top right of the main screen. This indicator appears as a moving error while the targets are executing. When the targets are stopped, a bitmap appears that depicts the cause of the halt.

An active target button appears at the top right of the main screen. This button lists the active target. Depressing this button causes a list of all targets to appear as a menu. Selection of a target from this menu causes that target to become activated.

The status window is located at the bottom of MtDt. Miscellaneous information is displayed in this window.

Both the toolbar and the status window can be hidden as explained in the Options Menu section.

At the bottom right of the main window is a menu help indicator. This indicator shows the function of all menus and toolbar buttons prior to selection.

## 18. Hot Keys, Toolbar, Status Window

---

To the right of the menu help indicator is the ticks/angle indicator. This displays either the number of ticks since the last target reset (if available) or the current angle (if available.) This indication may not be valid for hardware targets that have been free-run since the last reset as MtDt is not able to determine the number of clock ticks under this condition. The users selects between ticks and angle from the IDE Options Dialog Box.

To the right of the ticks indicator is the steps indicator. This indicates the number of opcodes that have been executed for the active target since the last reset. The limitations listed above for the ticks indicator also applies to the steps indicator.

To the right of the steps indicator is the failures indicator. The failures indicator lists the number of script commands and behavior failures, respectively.

To the right of the failures indicator is the target type indicator. This is a bitmap that depicts the type of target that is currently active.

To the right of the target type indicator is the current time indicator. This displays the amount of time that has occurred since the last target reset. The limitations listed above for the ticks indicator also applies to the current time indicator.

To the right of the current time indicator is a clock. Use this to determine if it is time to go home.

# 19

## Supported Targets and Available Products

Due to its layered design, MtDt supports a variety of both simulated and hardware targets. Customer requirements dictate that these capabilities to be offered as specific individual products.

### 19.1 eTPU/CPU System Simulator

Our eTPU/CPU System Simulator supports instantiation and simulation of an arbitrary number and combination of eTPUs and CPUs. A dedicated external system modeling CPU could be used, for instance, to model the behavior of an automobile engine. Executable code can be individually loaded into each of these targets. Synchronization between targets is fully retained as the full system simulation progresses.

All CPU engine targets can be used with this system simulation include the CPU32, CPU16, and soon-to-be-released, PPC simulation engines.

### 19.2 eTPU2 Stand-Alone Simulator

This product is a single-target version that uses only a single instance of our eTPU2 simulation engine. Because it is a stand-alone product the user must use script commands files to act as the host and test vector files to act as the external system.

The eTPU2 Stand-Alone Simulator is a superset of the eTPU Stand Alone Simulator in that purchase of the eTPU2 Stand Alone Simulator license allows installation of a fully-functional eTPU Stand-Alone Simulator product as well as an eTPU2 Stand-Alone Simulator product.

### 19.3 eTPU Stand-Alone Simulator

This product is a single-target version that uses only a single instance of our eTPU simulation engine. Because it is a stand-alone product the user must use script commands files to act as the host and test vector files to act as the external system.

This product simulates the original eTPU1 from Freescale. For simulation of the new eTPU2, see the eTPU2 Stand-Alone Simulator product.

### 19.4 eTPU2 Simulation Engine Target

The Enhanced Time Processing Unit Two, or ‘eTPU2’, is a microsequencer sold by STMicroelectronics and Freescale Semiconductor on a variety of microcontrollers including the SPC563Mxx. This is sold as a stand alone product and also as a system simulator in which it is co-simulated along with one or more CPU targets.

The eTPU2 simulation engine can be used both as a stand-alone device and in conjunction with other targets including multiple TPUs. When used in stand-alone mode, of primary importance are script commands files and test vector files.

### 19.5 eTPU Simulation Engine Target

The Enhanced Time Processing Unit, or eTPU, is a microsequencer sold by Freescale on a variety of microcontrollers.

# 20

## Building the Target Environment

The simulated or hardware development environment is specified in special MtDt build script files. ASH WARE provides standard build scripts for all its products. In the vast majority of cases these build scripts are sufficient and therefore effectively transparent to the user. But occasionally a user may desire to modify the simulation environment to unlock advanced capabilities. This chapter describes how to do so.

A complete system may consist of multiple CPUs, TPUs, peripherals, and non-electrical devices such as automobile engines. MtDt supports simulation of such advanced systems using MtDt build script files. MtDt build script files are used to create the targets and specify how they interact.

### Theory of Operation

MtDt is capable of instantiating and simulating multiple targets, allowing them to interact via shared memory while maintaining the correct synchronization and relative timing. In addition, a rich set of debugging capabilities normally associated with single target systems has been extended to this multiple target environment.

Each target loads and runs its own executable code image.

Each target can have primary and startup script commands files. TPU targets can also have ISR files that are associated with and activated by specific interrupts, and test vector file that can be used to wiggle the TPU's I/O pins.

## 20. Building the Target Environment

---

The relative timing of targets is maintained with one femto-second precision. Each target has its own atomic execution step size that must be a multiple of the femto-second precision. Negative numbers and zero are valid steps sizes. Since the currently-supported targets are all execution cycle simulators, the step size is equal to the amount of simulated time it takes to execute a single opcode.

As each target executes it is advanced by the amount of time the last opcode took to execute. It is then scheduled to execute again when the simulation time is equal to the target's next scheduled time. The current simulation time is defined as the time that the next scheduled target will execute.

Although all the currently-supported targets are execution-cycle simulation engines, this is not a fundamental restriction of MtDt. In fact, MtDt can support targets that have much finer execution granularities. This would allow, for instance, a VHDL target that properly models inter-target interaction down to the transistor level.

### Debugging Capabilities

All standard single-target debugging capabilities such as single stepping, breakpoints, goto cursor, etc., are available in the multiple target debugging environments. For instance, if breakpoints are injected and activated within multiple targets, the simulation halts on whichever breakpoint is encountered first.

A concept of an "active target" is employed to support specifically single-target capabilities such as single stepping. When the active target is single-stepped, the entire system simulation proceeds until the active target completes the commanded single step.

With an essentially limitless number of targets, and with the large number of possible windows per target, the vast number of windows can become unwieldy, to say the least. Actually, without some mechanism to bring order to the chaos of having way too many windows, MtDt becomes unusable. Workshops bring order to this chaos and are therefore a key enabling feature that makes MtDt usable. Each target can be associated with a specific workshop. Those target's windows are displayed only when the workshop associated with that target is activated. Individual windows can be overridden to appear in more than one target.

A target other than the active target can halt a simulation. In this situation the workshop is changed to one associated with the halting target. This can be caused, for instance, if a breakpoint is encountered in a non-active target. In this case, the simulation is halted, the halting target becomes the active target, and the workshop is switched to the one

associated with the newly-active target.

### **Building the MtDt Environment**

With MtDt an entire hardware or simulation environment can be built. This is done using a dedicated build script file that gets loaded when the project file is loaded. A detailed description of each command that can be used within MtDt build script files is found in the MtDt Build Script Commands File section. That section explains how a complete system is defined using these build script commands.

### **MtDt Simulated Memory**

Simulated targets require memory for executing code, for holding data, and for providing capabilities supported in a simulated environment. The following is a list of simulated memory characteristics supported by MtDt memory.

- Multiple address spaces
- Memory sizing
- Read only or read/write accesses
- Shared memory
- Byte, word, and long-word access widths
- Access speed based on even or odd access addresses
- Privilege violations
- Bus faults
- Address faults
- Banking
- Mirroring

The ASH WARE MtDt simulated memory model supports all of these characteristics though at the cost of increased complexity. The good news is that for many applications the standard memory models work just fine so a detailed understanding of memory modeling is not required. In the vast majority of other cases only a small percentage of these capabilities are required.

### **Memory Block**

## 20. Building the Target Environment

---

Whereas a simulated memory map can support a large variety of characteristics that might change from one memory range and address space to the next, a memory block is a range of memory that has a single uniform set of characteristics.

A target's address space comprises a finite number of memory blocks. For instance, a 3 wait state RAM could reside at address 0 to FFFF hexadecimal. A 0 wait state ROM could reside at address 1000 to 1FFFF. The rest of memory, from 1000 to FFFFFFFF hexadecimal, could be empty.

There are a number of rules associated with memory blocks. A build of MtDt simulated memory will succeed only if both of the following rules are met:

- Memory blocks must cover all memory.

- Memory blocks may not occupy both the same address and address space.

A report file for each build attempt provides a detailed listing of the memory map, including the information required to fix any problems.

The following is an example script commands sequence.

```
#define MEM_DEVICE_STOP 0xffff
#define BLANK_START MEM_DEVICE_STOP + 1
#define MEM_END 0xffffffff
instantiate_target(SIM16, "MySim16");
add_mem_block("MySim16", 0, MEM_DEVICE_STOP,
              "RAM", ALL_SPACES);
add_non_mem_block("MySim16", BLANK_START, MEM_END, "OFF",
                  ALL_SPACES);
```

In this example a single CPU16 CPU is instantiated. A 64K simulated memory device is added between addresses 0 and FFFF hexadecimal and is assigned the name "RAM." The device resides in all address spaces. A second memory block, also residing in all address spaces, fills the rest of memory between 1000 hexadecimal and FFFFFFFF hexadecimal and is assigned the name "OFF." This block is blank and as such takes up no physical memory on your computer.

### Address Spaces

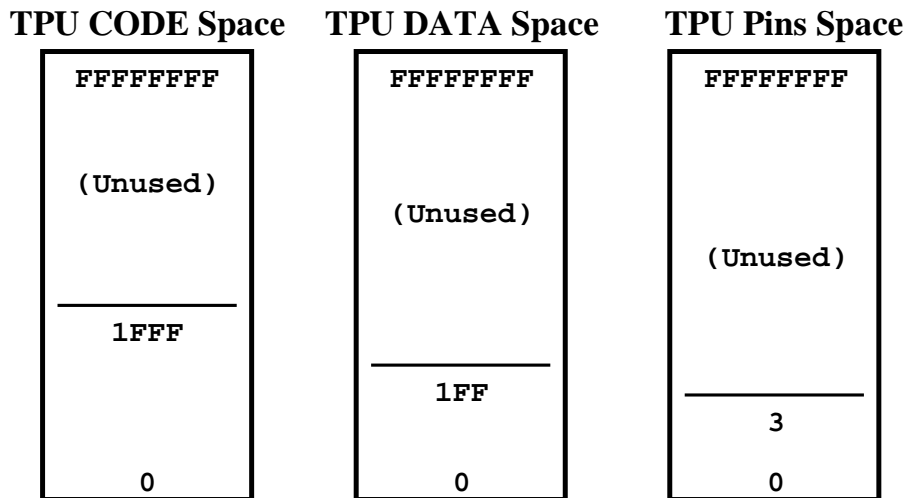
All MtDt simulated memory supports eight address spaces. The function of each address space depends entirely on the particular target and how it is specified in the build script file.

For instance, a simulated TPU target makes use of three address spaces: Code, Data, and Pins. By treating its I/O pins as shared memory the simulated TPU exposes its pins to other targets. This allows, for instance, a simulated engine model to read and modify the TPU's pins.

With the nearly universal acceptance of the superiority of a single, unified, large address space why does MtDt still support non-unified memory space architecture? The answer is twofold. First, the MtDt supports older but still popular architectures such as CPU16 in which a split code/data and space might actually be employed. Second, the multiple address space model provides the required mechanism for support of advanced simulation features. These mechanisms do not necessarily exist in the actual hardware. For example, an engine modeling CPU might be set up to query and modify TPU channel I/O pins. To support this, the TPU pins have been exposed in a purely theoretical "PINS" address space. MtDt can be configured so that a read or write in the engine modeling CPU's DATA\_SPACE occurs in the TPU's PINS space, thus allowing the engine modeling CPU to react to and drive the TPU's pins. This mechanism does not have a hardware corollary, but it provides the powerful capability of simulating the full system.

In many cases a uniform address model is desirable. This is achieved by mapping all address spaces to the same physical memory.

The following diagram depicts the address spaces accessed by the TPU simulation engine.



The TPU simulation model fetches code between 0 and 1FFF hexadecimal from its CODE space. It accesses its parameter RAM and host interface registers between 0 and 1FF hexadecimal of its DATA space. And it accesses its channel pins in the first four bytes of a simulated PINS space. Note that its code banks are "unrolled" and placed linearly in memory.

## 20. Building the Target Environment

---

In order for accesses to these spaces to behave properly, simulated memory devices must be placed into these address spaces. The following build script commands create the required memory for a stand-alone TPU Simulation engine.

```
// Create a target TPU
instantiate_target(TPU_SIM, "TpuSim");
// Create a simulated memory block
// for the TPU's code (microcode)
add_mem_block("TpuSim", 0, 0x1fff, "Code",
              TPU_CODE_SPACE);
add_non_mem_block("TpuSim", 0x2000, 0xFFFFFFFF,
                  "UnusedCode", TPU_CODE_SPACE);

// Create a simulated memory block
// for the TPU's data (host interface)
add_mem_block("TpuSim", 0, 0x1fff, "Data",
              TPU_DATA_SPACE);
add_non_mem_block("TpuSim", 0x200, 0xFFFFFFFF,
                  "UnusedData", TPU_DATA_SPACE);

// Create a simulated memory block for the TPU's pins
// (channel pins and TCR2 counter pin)
add_mem_block("TpuSim", 0x0, 0x3, "Pins",
              TPU_PINS_SPACE);
add_non_mem_block("TpuSim", 0x4, 0xFFFFFFFF,
                  "UnusedPins", TPU_PINS_SPACE);

// Be sure to provide a non_mem block
// for the unused address spaces
add_non_mem_block("TpuSim", 0x0, 0xffffffff, "B4",
                  TPU_UNUSED_SPACE);
```

In this example the TPU's code, data, and pins address spaces are filled with the memory devices required for proper operation. Unused space above and below the simulated devices is filled in with blank blocks. This is required, as all space must be filled in; even unused space must be provided with memory block(s).

### Memory Block Size

Each memory block has a specific size. The size is equal to the stop address minus the start address plus one. A common error is to overlap by one byte the end of one device with the start of the next device. MtDt cannot support multiple devices occupying the same address in the same address space so this causes an error. One method for avoiding

this error is to cascade ‘#define’ directives and thereby ensure that contiguous devices form the proper zero-byte seam.

```
#define FLASH_SIZE    0x10000    /* 64K FLASH device */
#define RAM_SIZE      0x8000     /* 32K RAM device */
#define FLASH_START   0
#define FLASH_END     FLASH_START + FLASH_SIZE - 1
#define RAM_START     FLASH_END + 1
#define RAM_END       RAM_START + RAM_SIZE - 1
#define BLANK_START   RAM_END + 1
#define BLANK_END     FFFFFFFF
instantiate_target(SIM32, "MyCpu");
add_mem_block("MyCpu", FLASH_START, FLASH_END,
              "Flash", ALL_SPACES);
add_mem_block("MyCpu", RAM_START, RAM_END, "RAM",
              ALL_SPACES);
add_non_mem_block("MyCpu", BLANK_START, BLANK_END,
                  "Empty", ALL_SPACES);
```

In this example, two devices are created in such a way that a zero-byte seams between them is guaranteed. All memory for every address spaces is covered. Notice that the FLASH and RAM sizes can be changed at a single location and that the devices will remain contiguous in memory.

## Memory Block Access Control

The purpose of the memory block access control is to make a simulated model match the behavior of real hardware. For instance you might want to make a ROM read-only, such that reads are simply ignored or perhaps cause a bus fault. An odd access may cause an address fault or an additional wait state. Memory block access control allows the required level of control to achieve this. This section serves as a high-level guide rather than a detailed description.

Each memory block supports the following access types.

```
8-bit read
8-bit write
16-bit read
16-bit write
24-bit read
24-bit write
32-bit read
32-bit write
64-bit read
```

## 20. Building the Target Environment

---

```
64-bit write
128-bit read
128-bit write
```

For each access type, the user can specify a number of parameters. The following parameters are available.

```
Clocks per even access
Clocks per odd access
Odd access causes bus fault yes/no?
Bus fault yes/no?
Blank access yes/no?
Dock offset (applicable docked accesses only!)
Dock function code (applicable to docked accesses only)
```

In addition, there is a block-wide default, "blank access value." This is the value that is returned on a read access to a memory block that has been marked as blank.

### Read/Write Control

It is possible to disable specific types of memory accesses.

In this example a memory device is configured as read only. A write to this memory device will not cause the values in memory to change. This read only behavior could be used to model a ROM device.

```
#define ROM_STOP 0xffff
#define BLANK_START ROM_STOP + 1
#define MEM_END 0xffffffff
instantiate_target(SIM32, "MyCpu");
add_mem_block("MyCpu", 0, ROM_STOP, "Rom", ALL_SPACES);
add_non_mem_block("MyCpu", BLANK_START, MEM_END, "Empty",
                  ALL_SPACES);

// Turn off READ access for the ROM device
#define WRITE_ALL RW_WRITE8 + RW_WRITE16 + RW_WRITE32
set_block_to_off("MyCpu", "Rom", ALL_SPACES, WRITE_ALL);
```

The command specifies that for all address spaces, all write accesses will be "empty." Despite this being an "empty" access, other parameters such as clocks per even cycle remain valid. The last two arguments specify the address spaces and access types affected by this script command. It is possible to indicate specific address spaces and a specific type of access. For instance, using this script command, one could specify 32-bit writes to user data space.

### Clocks per Access Control

For each memory block and type of access, the clocks per even access and the clocks per odd access can be specified. This capability effectively provides the capability of setting the number of wait states.

```
set_block_timing("MyCpu", "Rom", ALL_SPACES, RW_ALL, 2, 3);
```

In this example even accesses are set to two clocks per access and odd accesses are set to three clocks per access. In this example these settings apply to all address spaces and for all read and write accesses, though it is possible to indicate the specific set of address spaces and access types for which this applies.

### Address Fault Control

For each address space and access type an address fault can be set to occur on odd accesses. Note that this is generally not applicable to 8-bit accesses, though it is still available.

```
#define CODE_SPACE    CPU32_SUPV_CODE_SPACE \
                    + CPU32_USER_CODE_SPACE
set_block_to_addr_fault("MyCpu", "Rom", CODE_SPACE,
                       READ_ALL);
```

In this example, odd read accesses to the memory block's code space are configured to cause an address fault.

### Bus Fault Control

For each address space and access type a bus fault can be set to occur.

```
set_block_to_bus_fault("MyCpu", "Rom",
                      CPU32_USER_CODE_SPACE, RW_ALL);
```

In this example, any access while at the user privilege level result in a bus fault. This might be useful in a protected system in which a user process is prevented from accessing hardware.

### Sharing Memory

A fundamental aspect of multiple target simulation is the ability to share memory. MtDt employs "docking" to implement shared memory. If two targets are to share memory, one target must dock memory blocks to another target.

There is a fundamental and important lack of symmetry in that one target must provide the "dock-from" block and the other target must be the "dock-to." The "dock-to" target is relatively unaffected by being the recipient of a dock, other than that its physical memory

## 20. Building the Target Environment

---

might be modified on occasion.

On the other hand, there are numerous effects to the "dock-from" target. First and foremost, it must be able to support extrinsic, or externally mapped, memory. In other words, it must be able to project its memory access outside of itself and to a different target.

Note that this command must match exactly the memory bounds of an existing memory block for the docking device, but not for the "dock-to" target. In fact, the "dock-to" target could be any target such as a MC68332 across a BDM port. In fact, the "dock-to" target could be itself, and this is the recommended way of modeling multiple image memory. Although memory accesses are fully re-entrant, legal, and often necessary, it is possible to create an infinitely cyclic access that would, without guards, cause a stack overflow on your computer. To guard against this, MtDt has limited memory accesses re-entrance to a depth of 100.

The following build script command is presented in a later example.

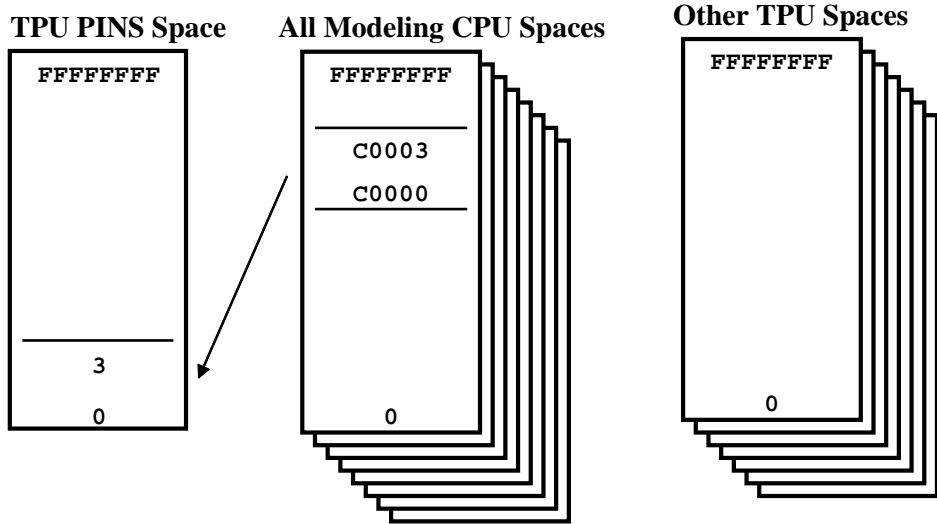
```
// ...
set_block_to_dock("HostCpu", "TpuDataDock", ALL_SPACES,
                  "Tpu", 0-0x80000);
// ...
```

In the above command a previously-added blank block is docked to memory contained in a target named "TPU." The last argument, ALL\_SPACES, is potentially problematic, as will be discussed later.

### Shared Memory Address Space Transformation

No assumptions should be made about address spaces between or among dissimilar targets. In other words, the code space of a CPU32 may not map to the code space of memory shared with a TPU. For memory docks between dissimilar target types it is critical to fully specify all address spaces from the docking memory block. Due to the lack of symmetry, this is not true for the dockee. The following script command should be used to fully define all docked address spaces between dissimilar targets.

When docking a CPU to a TPU the following address space transformation such as that shown in the following figure is often required.



The following build script commands generate the address space transformations shown in the above figure. If the modeling CPU does a read from its address C0000 hexadecimal, the read will actually access the shared memory with the TPU in its PINS space.

```
set_block_dock_space("ModelingCpu", "TpuPinsDock",
                    ALL_SPACES, RW_ALL, TPU_PINS_SPACE);
```

In this example all address spaces from a docked block of a modeling CPU are set to the TPU's PINS address space. This is an eight-to-one transformation in that an access in any of the CPU's address spaces becomes an access to the TPU's PINS space. For example if the CPU performs a data read within this block, the value of the TPU's channel pins will be what actually gets read.

## Shared Memory Address Offset

Shared memory need not appear in the same address from the perspective of each target. Indeed, shared memory usually appears at different addresses for each target that is sharing that memory. The following illustrates this.

```
set_block_to_dock("HostCpu", "TpuDataDock", ALL_SPACES,
                 "Tpu", TPU_DOCK_OFFSETT);
```

In this example an offset of TPU\_DOCK\_OFFSETT is applied to any HostCpu access within the docking block. For example if the docking block is at address FFE00

## 20. Building the Target Environment

---

hexadecimal and an offset of -FFE00 hexadecimal is applied by defining TPU\_DOCK\_OFFSETT to this value, a CPU access at address FFE20 occurs at address 20 within the TPU.

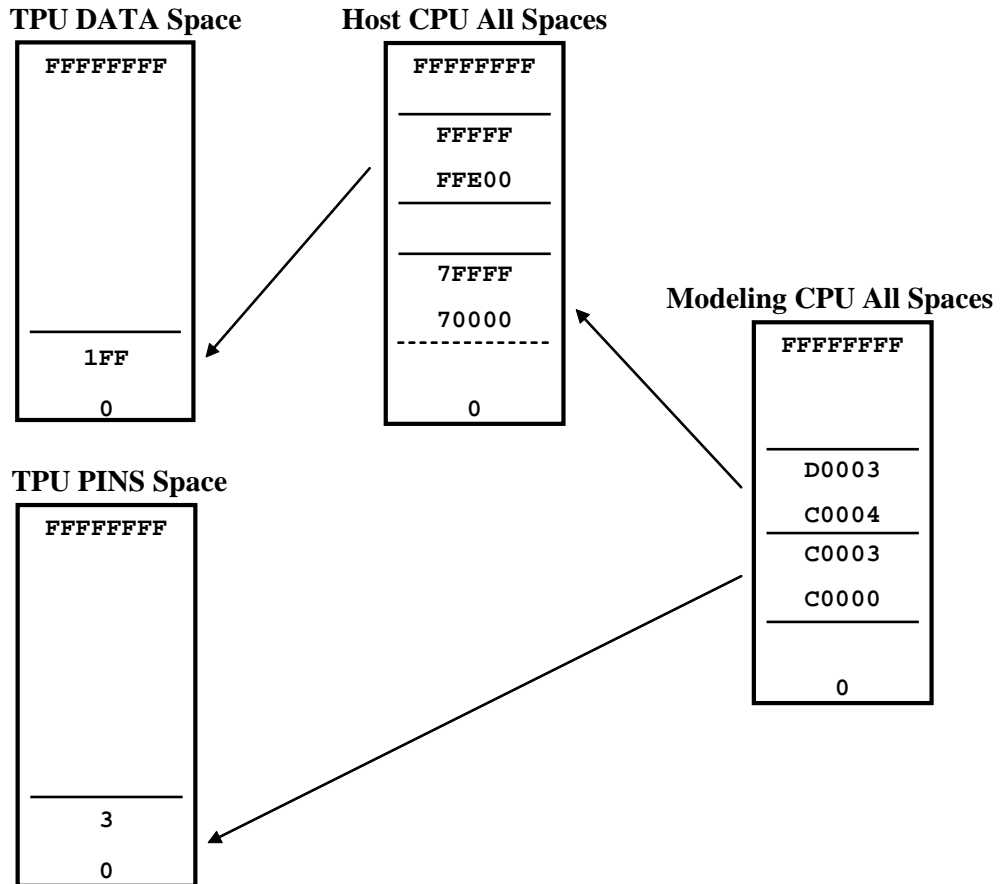
### **Shared Memory Timing**

Timing transformations allow the clock per access to be specified for the docking block. For instance a shared memory block between a TPU and a CPU might take the CPU two clocks to access, but might take the TPU only a single clock.

There is no special method for doing this. The timing parameters specified by each of the targets own dock block apply to the docking target.

### **A Complete Shared Memory Example**

The example in this section creates a TPU simulation engine, a host CPU simulation engine, and a modeling CPU simulation engine. The shared memory architecture from the following figure is generated.



The following build script commands instantiate the shared memory architecture found in the above figure.

```
#define MEM_END 0xffffffff

// Create a target TPU
instantiate_target(TPU_SIM, "Tpu");

// Create a simulated memory block
// for the TPU's code (microcode)
add_mem_block("Tpu", 0, 0x1fff, "Code", TPU_CODE_SPACE);
add_non_mem_block("Tpu", 0x2000, MEM_END, "B1",
```

## 20. Building the Target Environment

---

```
        TPU_CODE_SPACE);

// Create a simulated memory block
// for the TPU's data (host interface)
add_mem_block("Tpu", 0, 0x1fff, "Data", TPU_DATA_SPACE);
add_non_mem_block("Tpu", 0x200, MEM_END, "B2",
        TPU_DATA_SPACE);
// Create a simulated memory block for the TPU's pins
// (channel pins and TCR2 counter pin)
add_mem_block("Tpu", 0x0, 0x3, "Pins", TPU_PINS_SPACE);
add_non_mem_block("Tpu", 0x4, MEM_END, "B3",
        TPU_PINS_SPACE);
// Be sure to provide a non_mem block
// for the unused address spaces
add_non_mem_block("Tpu", 0x0, MEM_END, "B4",
        TPU_UNUSED_SPACE);

//***** END OF TPU *****

// Create a target host CPU
instantiate_target(SIM32, "HostCpu");

// Add a half-meg RAM
add_mem_block("HostCpu", 0, 0x7FFFF, "RAM", ALL_SPACES);

// Add three empty spaces
add_non_mem_block("HostCpu", 0x80000, 0xFFDFF, "B1",
        ALL_SPACES);
add_non_mem_block("HostCpu", 0xFFE00, 0xFFFFF,
        "TpuDataDock", ALL_SPACES);
add_non_mem_block("HostCpu", 0x100000, MEM_END, "B2",
        ALL_SPACES);

// Set the middle empty block to dock with the TPU target
set_block_to_dock("HostCpu", "TpuDataDock", ALL_SPACES,
        "Tpu", 0-0x80000);
// Make sure that no matter which space
// the TPU accesses within this dock
// the TPU's data space is always accessed
set_block_dock_space("HostCpu", "TpuDataDock",
        ALL_SPACES, RW_ALL, TPU_DATA_SPACE);

//***** END OF HOST CPU *****
```

```
// Create a CPU for modeling the external system
instantiate_target(SIM32, "ModelCpu");

// Add a half-meg RAM
add_mem_block("ModelCpu", 0, 0xBFFFF, "RAM", ALL_SPACES);

// Add three empty spaces
add_non_mem_block("ModelCpu", 0xC0000, 0xC0003,
                  "TpuPinsDock", ALL_SPACES);
add_non_mem_block("ModelCpu", 0xC0004, 0xD0003,
                  "CpuCpuShare", ALL_SPACES);
add_non_mem_block("ModelCpu", 0xD0004, MEM_END, "B2",
                  ALL_SPACES);

// Set the lowest empty block to dock with the TPU target
set_block_to_dock("ModelCpu", "TpuPinsDock", ALL_SPACES,
                  "Tpu", 0-0xC0000);

// Make sure that no matter which space
// the TPU accesses within this dock
// the TPU's pins space is always accessed
set_block_dock_space("ModelCpu", "TpuPinsDock",
                    ALL_SPACES, RW_ALL, TPU_PINS_SPACE);

// Set the middle empty block to dock with the HOST CPU
set_block_to_dock("ModelCpu", "CpuCpuShare", ALL_SPACES,
                  "HostCpu", 0x70000-0xC0000);

//***** END OF MODELING CPU *****
```

In this example the shared memory architecture from the above figure is generated.

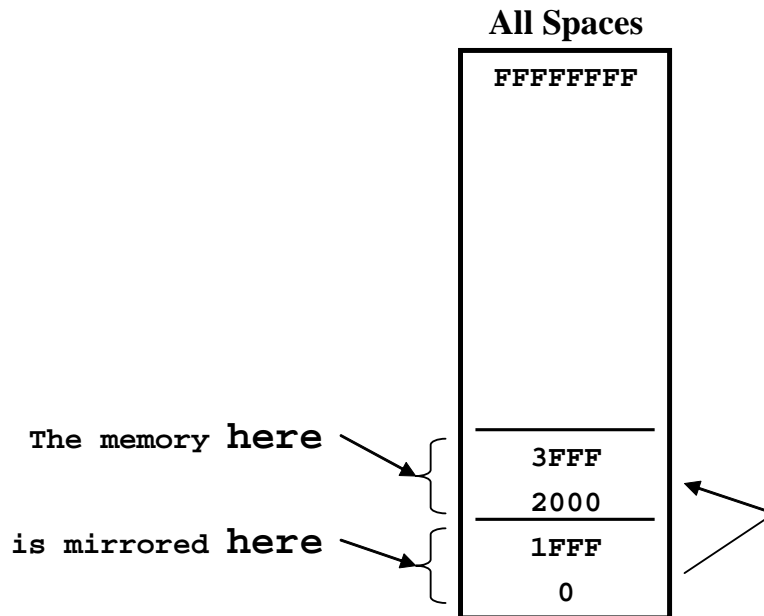
## Simulating Mirrored Memory

Mirrored memory is memory that is accessible at multiple address ranges within a memory map. This can occur, for instance, when not all address bits are decoded for a memory device. The following figure depicts an 8K memory device that resides in a 64K memory system. Assume it is an 8-bit wide device. Since the 8K device is on a byte wide bus, the device itself decodes the lower 13 address bits, A12 through A0. Assume that the memory controller decodes only the upper two address bits, A15 and A14, and enables the device when both are zero. This means that nothing decodes A13, and thus the memory device is

## 20. Building the Target Environment

---

activated when A15 and A14 are zero, regardless of the state of A13.



This mirrored memory architecture is created by implementing a dock from the address space to itself as follows.

```
instantiate_target(SIM16, "MyCpu");
add_non_mem_block("MyCpu", 0x0, 0x1FFF, "Mirror",
                  ALL_SPACES);
add_mem_block("MyCpu", 0x2000, 0x3FFF, "RAM",
              ALL_SPACES);
add_non_mem_block("MyCpu", 0x4000, 0xFFFFFFFF, "B1",
                  ALL_SPACES);

// Create a mirror at the lowest 8K of the next higher 8K
set_block_to_dock("MyCpu", "Mirror", ALL_SPACES, "MyCpu",
                  0x2000);
```

In this example the previously-described memory architecture with mirrored memory is implemented.

## Computer Memory Considerations

MtDt uses your computer's memory to model the memory devices belonging to your target.

There is roughly a one-to-one correspondence between the total amount of memory occupied by the simulated devices and the amount of your computer's memory that is required. In the examples shown in the previous sections, simulated memory totals a few hundred kilobytes. This is a trivial amount of memory for a modern computer. When many megabytes are required, you are limited to the amount of virtual memory available for the MtDt application that your computer can provide. If you attempt to simulate a 100-gigabyte memory device, for example, but have only 50-gigabytes of available virtual memory on your computer, the build script file will fail to execute.

Note that since modern computer systems employ virtual memory, the amount of simulated memory can exceed the amount of RAM actually in your computer. Adjusting the available amount of virtual memory on your computer can increase the total amount of memory devices that you can simulate. A description of how to increase the swap file size of your computer is beyond the scope of this manual.

