Multiple Target Development Tool

Version 3.41

User Manual

Developed by John Diener, Andrew Klumpp, Keith Kroeker, Michael Schwager, and Tony Zabinski.

Edited by Margaret Frances.

Stan Ostrum, Richard Soja, and Mike Pauwels of Freescale, Inc. contributed ideas for key features, as did Marc Peters from Mot-Consulting GmbH.

Walter Banks of Byte Craft Limited provided the critical companion technology in the eTPU "C" Compiler.

ASH WARE Inc.

2610 NW 147th Place

Beaverton, OR 97006

www.ashware.com

TABLE OF CONTENTS

TABLE OF CONTENTS 2

FOREWORD 10

ACKNOWLEDGEMENTS 11

ON-LINE HELP CONTENTS 12

OVERVIEW 13

- **Targets and Products** 13
- **Concurrently Developing Code for Multiple Targets** 13
- **The Freescale eTPU and TPU** 13
- **Script Commands Files** 14
- **Test Vector Files** 14
- **Sharing Memory between Multiple Targets** 14
- **Miscellaneous Capabilities** 14

SOFTWARE DEPLOYMENT, UPGRADES, AND TECHNICAL SUPPORT 15

- **World Wide Web Software Deployment** 15
- **World Wide Web Software Upgrades** 15
- **Hearing about Software Releases** 15
- **Technical Support Contact Information** 16
- **Overview of Version 3.41, 3.40, 3.20, 3.10, 3.00, 2.1, and 2.0 Enhancements** 16

PROJECT SESSIONS 19

SOURCE CODE FILES 20

- **Theory of Operation** 20
- **Source Code Search Rules** 20
  - Absolute and Relative Paths 21
- **Supported Compilers and Assemblers** 21

SCRIPT COMMANDS FILES 22

- **Overview** 22
  - Types of Script Commands Files 22

Similarities to the C Programming Language 22
The Primary Script Commands File 23
ISR Script Commands Files 23
Startup Script Commands Files 24
The MtDt Build Script Commands File 24
MtDt Build Script Commands File Naming Conventions 25

Script Commands File Format and Features 25

Script Commands Format 26
Multiple-Target Scripts 26
Directives 26
The #define directive 26
The #include <FileName.h> directive 27
The #ifdef, #ifndef, #else, #endif directives 27
Enumerated Data Types in Script Commands Files 27
Integer Data Types in Script Commands Files 27
Referenced Memory in Script Commands Files 28
Assignments in Script Commands Files 28
Operators and Expressions in Script Commands Files 29
Comments 29
Decimal, Hexadecimal, and Floating Point Notation 29
String Notation 30

Script Commands Groupings 30

All Target Types Script Commands 30
New 3.41 eTPU/TPU Script Commands 31
eTPU Script Commands 31
TPU Script Commands 31
Build Script Commands 32
Clock Control Script Commands 32
Timing Script Commands 32
Modify Memory Script Commands 33
Verify Memory Script Commands 33
Write Register Script Commands 34
Verify Register Script Commands 34
Write Symbol Value Script Command 34
Verify Symbol Value Script Commands 35
System Script Commands 36
File Script Commands 36
Trace Script Commands 37
New 3.41 Code Coverage Script Commands 38
RAM Test Script Commands 41
eTPU/TPU Channel Function Select Register Script Commands 42
eTPU/TPU Channel Priority Register Script Commands 42
eTPU/TPU Pin Control and Verification Script Commands 42
eTPU Topics 42
TPU Topics 43
eTPU/TPU Pin Transition Behavior Script Commands 43
New 3.41 eTPU/TPU Resizing the Pin Transition Buffer 44
New 3.41 Thread Script Commands 44
Disable Messages Script Commands 44
eTPU System Configuration Script Commands 45
eTPU Time Base Configuration Script Commands 45
New 3.41 eTPU Channel Data Script Commands 46
eTPU Channel Base Address Script Commands 46

eTPU Channel Function Mode (FM) Script Command 47
eTPU Event Vector Script Commands 47
eTPU Interrupt Script Commands 48
TPU Parameter RAM Script Commands 48
eTPU/TPU Host Service Request Register Script Commands 49
TPU Channel Interrupt Service Register Script Commands 49
TPU Host Sequence Request Register Script Commands 50
TPU Clock Control Script Commands 50
TPU3-Specific Script Commands 51
TPU Bank and Mode Control Script Commands 51
Setting the TPU Code Bank 52
TPU Match, Transition, and Link Script Commands 52
eTPU/TPU Interrupt Association Script Commands 52
eTPU/TPU External Logic Commands 53
eTPU Considerations 54
TPU Considerations 54
Build Script Commands 54

Automatic and Pre-Defined Define Directives 58
Target-Specific Scripts 58
Determining the Auto-Run Mode 58
Determining the Interrupt Number 59
Passing Defines from the Command Line 59
TPU Target Pre-Defined Define Directives 59

Pre-Defined Enumerated Data Types 60
Script FILE_TYPE Enumerated Data Type 60
Script DUMP_FILE_OPTIONS Enumerated Data Type 61
Save Trace File Enumerated Data Types 61
Base Time Enumerated Data Type 61
Script TARGET_TYPE Enumerated Data Type 62
Build Script ADDR_SPACE Enumerated Data Type 62
Build Script READ_WRITE Enumerated Data Type 62
Register Enumerated Data Type 63
eTPU Register Enumerated Data Types 63
TPU Register Enumerated Data Types 64
CPU32 Register Enumerated Data Types 64
CPU16 Register Enumerated Data Types 65

TRACE BUFFER AND FILES 66
Overview 66
Generating Viewable Files 66
Generating Parseable Files 66
Parsing the Trace File 66
Trace Buffer Size Considerations 66

TEST VECTOR FILES 68

Overview 68

Test Vector Generation Functional Model 68

Command Reference 69
Comments 69
Test Vector "Node" Command 69
Test Vector "Group" Command 70

Test Vector "State" Command 71
Master Test Vector Clock "Frequency" Command 71
Test Vector "Wave" Command 71
Test Vector Engine Waveform Example 72

FUNCTIONAL VERIFICATION 75

Overview 75

A Full Life-Cycle Verification Perspective 75

Data Flow Verification 76

Pin Transition Behavior Verification 76
Pin Transition Buffers 77

Code Coverage Analysis 78
Code Coverage Visual Interface 78
Code Coverage Verification Commands 78
Code Coverage Report Files 78

Regression Testing 80
New 3.41 Command Line Parameters 80
Test Termination 80
Regression Test Example 81
New 3.41 Cumulative Logged Regression Testing 81
File Location Considerations 82

EXTERNAL LOGIC SIMULATION 83
Example 1: Driving the TCR2 Pin 83
Example 2: Multi-Drop Communications 83

INTEGRATED TIMERS 85

Virtual Timers in Hardware Targets 85

WORKSHOPS 86

OPERATIONAL STATUS WINDOWS 87

Overview 87

Window Groups 87
Source Code File Windows 90
Script Commands File Windows 91
Watches Window 92
Local Variables Window 93
Call Stack Window 94
New 3.41 Thread Window 94
Trace Window 97
Complex Breakpoint Window 99
Memory Dump Window 99
Timers Window 100
eTPU Channel Function Frame Window 101
eTPU Configuration Window 101

eTPU Global Timer and Angle Counters Window 102
eTPU Host Interface Window 103
eTPU Channel Window 104
eTPU Scheduler Window 105
eTPU Execution Unit Registers Window 105
TPU Configuration Window 106
TPU Host Interface Window 108
TPU Scheduler Window 108
TPU Microsequencer Registers Window 109
TPU Execution Unit Window 109
TPU Channel Window 110
TPU Parameter RAM Window 112
683xx Hardware Debugger Configuration Window 112
SIM Main Window 113
SIM Ports Window 114
SIM Chip Selects Window 115
QSM Main Window 116
QSM Port Window 116
QSM QSPI Window 116
QSM SCI (UART) Window 117
Masked ROM Window 118
Standby RAM Submodule Window (68336/68376) 118
Static RAM Submodule Window (68338) 119
TPU Emulation RAM Window (68332, 68336, 68376) 119
TPU Main Window 119
TPU Host Interface Window 119
TPU Parameter RAM Window 120
GPT Main Window 120
GPT Input Captures Window 121
GPT Output Compares Window 121
GPT Pulse Accumulate Window 122
GPT Pulse Width Modulation Window 122
CTM4/CTM6 Bus Interface and Clocks Window 123
CTM4/CTM6 Free-Running Counter Submodule Window 123
CTM4/CTM6 Modulus Counter Submodule Window 123
CTM4 Double-Action Submodule Window 124
CTM4 Pulse Width Modulation Submodule Window 124
CTM6 Single-Action Submodule Window 124
CTM6 Double-Action Submodule - Modes Window 125
CTM6 Double-Action Submodule - Bits Window 125
Real Time Clock Window 126
Parallel Port I/O Submodule Window 126
TouCAN Main Window 126
TouCAN Buffers Window 127
QADC Main Window 128
QADC Ports Window 128
QADC Channels Window 129
683xx ASH WARE Hardware Window 130
CPU32 Simulator Configuration Window 130
CPU32 Simulator Busses Window 131
CPU32 Simulator Interrupt Window 131
CPU32 Registers Window 132
CPU32 Disassembly Dump Window 132
CPU16 Simulator Configuration Window 132
CPU16 Register Window 133
CPU16 Disassembly Dump Window 133

Unsupported Window 134

LOGIC ANALYZER 135

Overview 135
Executing to a Precise Time 135
View Waveform Selection 136
The Active Waveform 137
The Left and Right Cursors 137
#Left, Right, and Delta Cursor Time Indicators 138
Mouse Functionality 138
The Vertical Yellow Context Time Cursor 139
Context Time Indicator 139
Scroll Bars 139
Waveform Boundary Time Indicators 140
Buffer Data Start Indicator 140
Current Time Indicator 140
Auto-Scroll the Waveform Display as the Target Executes 140
Button Controls 140
Timing Display 141
Data Storage Buffer 141

DIALOG BOXES 142

File Open, Save, and Save As Dialog Boxes 142
Load Executable Dialog Box 142
Open Primary Script File Dialog Box 142
Save Primary Script Commands Report File Dialog Box 143
Open Startup Script Commands File Dialog Box 143
Open Test Vector File Dialog Box 143
Project Open and Project Save As Dialog Boxes 143
Run MtDt Build Script Dialog Box 143
Save Behavior Verification Dialog Box 144
Save Coverage Statistics Dialog Box 144
Auto-Build Batch File Options Dialog Box 144
Goto Time Dialog Box 145
Occupy Workshop Dialog Box 145
IDE Options Dialog Box 146
All Targets Settings 146
TPU Simulator Target Only 147
Active Target Settings 147
Workshops Options Dialog Box 147
Message Options Dialog Box 148
All Targets Messages 148
TPU Messages 148
Source Code Search Options Dialog Box 149
Reset Options Dialog Box 150
Logic Analyzer Options Dialog Box 150
Channel Group Options Dialog Box 151
Complex Breakpoint Conditional Dialog Box 151
Trace Options Dialog Box 152
Trace Window and Trace Buffer Considerations 152
Multiple Target Considerations 152
Local Variable Options Dialog Box 153
License Options Dialog Box 153
BDM Options Dialog Box 153
Memory Tool Dialog Box 154

Insert Watch Dialog Box 154
Watch Options Dialog Box 155
About ASH WARE's Multiple Target Development Tool 155
BDM Port Dialog Box 155

MENUS 156

Files Menu 156
Step Menu 158
Run Menu 159
Breakpoints Menu 159
Activate Menu 160
View Menu 160
Window Menu 160
Options Menu 161
Help Menu 162

HOT KEYS, TOOLBAR, AND STATUS INDICATORS 163

SUPPORTED TARGETS AND AVAILABLE PRODUCTS 164

eTPU/CPU System Simulator 164

TPU/CPU32 System Simulator 165

TPU/CPU16 System Simulator 165

eTPU Stand-Alone Simulator 165

TPU Stand-Alone Simulator 165

TPU Standard Mask Simulator 165

CPU32 Stand-Alone Simulator 166

CPU16 Stand-Alone Simulator 166

683xx Hardware Debugger 166

eTPU Simulation Engine 166

TPU Simulation Engine 166

The Time Processor Unit (TPU) 167
Support for TPU3 168

CPU32 Simulation Engine 169

CPU32 Hardware across a BDM Port 169

CPU16 Simulation Engine 169

FULL-SYSTEM SIMULATION 170

Theory of Operation 170

Debugging Capabilities 170

Building the MtDt Environment 171

MtDt Simulated Memory 171

Memory Block 171
Address Spaces 172
Memory Block Size 173
Memory Block Access Control 174
Read/Write Control 174
Clocks per Access Control 175
Address Fault Control 175
Bus Fault Control 175
Sharing Memory 175
Shared Memory Address Space Transformation 176
Shared Memory Address Offset 177
Shared Memory Timing 177
A Complete Shared Memory Example 177
Simulating Mirrored Memory 179
Computer Memory Considerations 180

FOREWORD

This User Manual is a superset of the On-line Help program. The On-line Help program is invoked within the IDE in various ways, such as when the <F1> function key is pressed. The differences between the User Manual and the On-line Help program are listed below.

- ? The On-line Help program is an electronic file. The Windows help program uses this file. The User Manual is a book.
- ? The On-line Help program contains special codes that allow the user to electronically browse through and jump from topic to topic. These codes are stripped from the User Manual.
- ? The User Manual has page numbers, while the On-line Help program does not.
- ? The User Manual contains a Table of Contents while the On-line Help program does not.
- ? The User Manual contains this Foreword while the On-line Help does not.

There are both advantages and disadvantages of the similarities between the User Manual and the On-line Help program. The most important advantage is that by becoming familiar with either medium, the user automatically becomes familiar with both media. The primary disadvantage is that the constraints imposed on the On-line Help program are also imposed on the User Manual and visa versa. For instance, the On-line Help program imposes the constraint that the different browse sequences may not be intermixed. This forces the organization of the manual to be by browse sequence rather than by subject, although the latter would be preferable.

ACKNOWLEDGEMENTS

The story of ASH WARE must necessarily begin with the TPU so thank you Vernon for your wonderful invention, and Celso & Co. for bringing it to the next level!

The last few years have been particularly rewarding with the development of the Enhanced Time Processing Unit (eTPU) and the emergence of "Los Tres Amigos," Mike, Walter and Me. You two have been a particularly dear part of my experience as your roles have extended well beyond the purely technical. Thank you!

To the entire team here at ASH WARE including Keith, Michael, and Tony, who make to happen, to Barbara, who makes us look good, to Margaret who makes us sound good and (hopefully) profitable, and to Amy who makes it clear!

Thank you users, too numerous to mention, but certainly Stan and Josef stand out from the rest. Thank you Freescale and Metrowerks, and Richard, the great idea man, Sharon, Munir, Robert, Dave, Clint, Samantha, and all the others too numerous to be mentioned individually, but do not be offended if you are not, because, yes, I do mean you, too!

Thank you John, the great brain behind our products: our best and most useful features are always yours! Back at Allied when you first became involved with the original TPU you disappeared and did not ask me for the help which I offered and which chagrined me at first since nobody can understand the TPU without benefit of the course, or the book, or a colleague. But of course YOU could!

And thank you Suzanne, who stands by me.

Andy Klumpp
Technical Director, ASH WARE Inc.

ON-LINE HELP CONTENTS

The following help topics are available.

- ? Overview
- ? Software Deployment, Upgrades, and Technical Support
- ? Version 3.41, 3.40, 3.20, 3.10, 3.00, 2.1, and 2.0 Enhancements
- ? Supported Targets and Available Products
- ? Full-System Simulation
- ? 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

OVERVIEW

Multiple Target Development Tool (MtDt) is a technology from which a variety of products are derived. 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 a 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 will soon 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 a 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 is 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 imperative 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. Extrinsically 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 extrinsically 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.

SOFTWARE DEPLOYMENT, UPGRADES, AND TECHNICAL SUPPORT

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 supported by MtDt 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 MtDt installation utility.
- ? Download and re-install MtDt 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 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 MtDt is available, download and re-install MtDt.

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

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.

MtDt 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 TPU 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
- ? support@ashware.com

Overview of Version 3.41, 3.40, 3.20, 3.10, 3.00, 2.1, and 2.0 Enhancements

New 3.41 The following is a list of the features that are new to the MtDt version 3.41.

- ? New 3.41 Extended eTPU code coverage to include event vectors
- ? New 3.41 Inferred event vector coverage
- ? New 3.41 Accumulate coverage over multiple tests
- ? New 3.41 View channel nodes
- ? New 3.41 Write and verify chan_data16(); commands
- ? New 3.41 View critical thread execution indices
- ? New 3.41 Clear Worst Case Thread Indices Commands
- ? New 3.41 Resizing the Pin Transition Buffer Testing
- ? New 3.41 Cumulative Logged Regression Testing

The following is a list of the features that are new to the MtDt version 3.40 and 3.30.

- ? Complex Breakpoint Window
- ? eTPU Stand-Alone Simulator Product
- ? eTPU/CPU System Simulator Product
- ? Multiple target scripting
- ? Added ability to view thread group activity in the logic analyzer window
- ? Improved Regression Testing section
- ? 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

The following is a list of the features that are new to the MtDt version 3.20.

- ? Testing automation via command line capabilities
- ? Verification and modification of symbolic data

- ? Superior Logic Analyzer Window including 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.

The following is a list of the features that are new to the MtDt version 3.10.

- ? 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
- ? Additional 683xx Hardware Debugger support for 68331, 68336, 68338, and 68376
- ? New TPU Standard Mask Simulator supports 683xx and MPC5xx
- ? New local variable options dialog box
- ? Additional IDE options
- ? Additional operation status windows

The following is a list of the features that are new to the MtDt version 3.00.

- ? 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
- ? Watch, local variable, and memory dump windows
- ? Workshops

The following is a list of the features that are new to MtDt version 2.1. Note that in this version MtDt was known as the TPU Simulator.

- ? TPU3 support
- ? Association of ISR script commands files with interrupts.
- ? Web-based software upgrades

The following is a list of the features that are new to MtDt version 2.0. Note that in this version MtDt was known as the TPU Simulator.

- ? TPU2 support
- ? Functional verification
- ? Pin transition behavior verification
- ? Code coverage analysis
- ? External logic simulation
- ? Auto-build

PROJECT SESSIONS

The Project Open and Project Save As dialog boxes allow association of the MtDt configuration with an MtDt project file. Various global settings are stored in this file, including the active microcode, test vector, script, and auto-build files.

To open an existing MtDt 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 MtDt project file. Various settings from the just-opened MtDt project file are loaded into MtDt. 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 MtDt 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 MtDt project file, assuming that no MtDt project file was passed on the command line. Otherwise, the MtDt project file passed on the command line is opened.

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

Whenever the MtDt application is exited, the configuration is automatically written to the active MtDt project file.

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, though this will soon be replaced with ASH WARE's own integrated assembler linker. 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. MtDt 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

MtDt 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.

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

In order for MtDt to read the executable code, the source code must be compiled, assembled, and linked. The resulting executable file can then loaded into MtDt. 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.

Source Code Search Rules

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

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

MtDt 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\Libary

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 MtDt 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

Supported Compilers and Assemblers

We will be adding support for compilers and assemblers based on customer demand. A list of these can be found on our Web site.

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 MtDt 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., MtDt executes the script commands in sequential order. This allows MtDt to know when to execute the commands. Timing commands cause MtDt to cease executing commands until a particular point in time. At that point in time, MtDt 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.

- ? 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

MtDt 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.

The Primary Script Commands File

MtDt 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 MtDt 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. MtDt 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.

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 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 interrupt service routine the ISR script commands file should clear the interrupt. This is accomplished using the clear_cisr(X), the clear_this_cisr(), 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_cisr() or similar script command.
- ? Do not use the clear_this_cisr() 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 a 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, eTPU1, or eTPU2), 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 named TPUSIM");
#else
print("this is not both 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();
```

Startup Script Commands Files

Startup scripts provide the capability to get the target into a known state following an MtDt-controlled reset. It is particularly useful for the case in which MtDt is configured to load an executable image when MtDt is 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 MtDt 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.

The MtDt Build Script Commands 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

Script Commands 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
- ? Assignment operators
- ? Operators and expressions
- ? Comments
- ? Numeric Notation
- ? String notation

Script Commands Format

The command is case insensitive 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.

Multiple-Target Scripts

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 acts 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.

```
etpu2.write_chan_hsrr (LAST_CHAN, 1);
wait_time(10);
verify_mem_u32(ETPU_DATA_SPACE, SLAVE_SIGNATURE_ADDR,  0xfffffF,
        DATA_TRANSFER_INTR_SIGNATURE);
etpu2.verify_data_intr(LAST_CHAN, 1);
etpu1.verify_data_intr(LAST_CHAN, 0);
```

In this example, a host service request is applied to target etpu1. Ten microseconds later script commands verify that the host service request generated a data interrupt on etpu1 but not etpu2.

Directives

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();
#else
print("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 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 simulator/debugger is launched in such a way that it runs without user input. In this case, upon reaching the end of the script file the application (simulator or debugger) is closed when it is part of an automated test suite and otherwise a message is issued to the user.

Enumerated Data Types in Script Commands Files

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.

Integer Data Types in Script Commands Files

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

Referenced Memory in Script Commands 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 address 0x20**
***((U32 *) 0x40 // Refers to a 32-bit word at address 0x40**

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 operators.

=	Assignment
+=, -=	Arithmetic assignment of addition and subtraction
***=, /=, %=**	Arithmetic assignment multiply, divide, and remainder
<<=, >>=	Bitwise assignment of shift right and shift left
&=, |=, ^=	Bitwise assignment of "and," "or," and "exclusive or"
<<=, >>=	Bitwise assignment of "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;
// Sets bits 31 and 15 of the 32-bit word at address 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;
```

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, EXCLUSIVE OR, and INVERSE.
- ? **<<, >>** 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_CHANNEL 16
#define SPARK_CHANNEL_BASE PARAMETER_RAM + BYTES_PER_CHANNEL * 5
// Write a 77 hexadecimal byte to address 150 hexadecimal
*((U8 *) SPARK_CHANNEL_BASE) = 0x22+0x55;
```

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

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

Comments

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);
```

Decimal, Hexadecimal, and Floating Point Notation

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

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 stings. The following exa mple 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

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"
```

Script Commands Groupings

Listed below are the available script command functional groups. Clicking on the desired command functional group will access the command listing for that group.

All Target Types Script Commands

? Clock control script commands

? Timing script 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
- RAM test script commands

New 3.41 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
- New 3.41 Pin transition buffer script commands
- Disable messages script commands
- New 3.41 Clear Worst Case Thread Indices Commands

eTPU Script Commands

- System configuration commands
- Timing configuration commands
- Channel data commands
- Channel base address commands
- Channel function mode (FM) commands
- New 3.41 Channel event vector entry commands
- Interrupt script commands

TPU Script Commands

- TPU parameter RAM script commands
- TPU channel interrupt service register script commands
- TPU host sequence request register script commands

- ? TPU clock control script commands
- ? TPU bank and mode control script commands
- ? TPU match, transition, and link script commands

Build Script Commands

- ? Build script commands

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 16.778 MHz.

Many Freescale MC683xx microcontrollers use an external low-frequency crystal to generate the CPU's clock source. This crystal generally has a frequency of 32.768 KHz or 2 to the 15th cycles per second. A crystal of this frequency is known as a "watch" crystal because a simple 15-bit ripple counter can be used to generate a one second clock tick. A phase lock loop within the 683xx microcontroller uses this to synthesize the main CPU clock.

The 683xx Hardware Debugger assumes that this external crystal oscillates at 32768 cycles per second and a number of calculations (such as TPU frequency, QSM frequency, etc.) are based on this assumption. Unfortunately, the 683xx Hardware Debugger has no way of verifying that this is, in fact, the correct crystal frequency so the following script command allows the user to override the default.

set_crystal_frequency(cylesPerSecond);

This code sets the assumed external clock frequency to cyclesPerSecond. Note that the cyclesPerSecond must be a decimal or hexadecimal number. A floating point number will not work.

set_crystal_frequency(34000);

The code in this example overrides the default external clock crystal value and sets it instead to 34 KHz.

Timing Script Commands

at_code_tag(TagString);

This command prevents subsequent commands from executing until the target hits the sourcecode 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.

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.

Modify Memory Script Commands

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

Verify Memory 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 bcation 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_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_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.

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

Verify Register 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.

Write Symbol Value Script Command

Write symbol value script commands provide the mechanism for writing the values of symbolic data for both variables as well as strings. Symbols are variables and strings defined within the user's

code.

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.

write_val(symbolExprString, exprString);
write_str(symbolExprString, valueString);

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 (prepended with 0x), or floating point numbers. A symbolic expression can be just a local/global symbol or a simple expression such as *V (de-reference), V[constant], V.member or V->member, where V is a symbol of the appropriate type. "SymbolExprString" must be a symbolic expression as described above, an "l-value" in compiler parlance. The write_str command is provided as a shorthand way to write a string into the memory pointed to by a symbolic expression.

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.

Symbol is defined within the user's code and is determined from the debugging information associated with the executable image. Value is the value to which the symbol will be set.

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_val("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.

Verify Symbol Value Script Commands

These verify symbol value commands have the same limitations as those commands described in the Write Symbol Value Script Command section.

verify_val("ExprString", "TestOpString", "ExprString");

The expression string (ExprString) can be numerical constants or simple symbolic expressions. Constants can be supplied as decimal signed integers, unsigned hexadecimal numbers (prepended with 0x), or floating point numbers. A symbolic expression can be just a local/global symbol, or a simple expression such as *V (de-reference), V[constant], V.member or V->member, where V is a symbol of the appropriate type.

"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.

verify_str(SymbolString, TestOpString, CheckString);

If a character pointer is resolved from "SymbolExprString", then the string at that location is compared to "CheckString" using the comparator specified in "TestOpString". If the outcome of this is true (non-zero), the verification test passes; otherwise a failure is reported. Supported comparator operators are ==, !=, >, and <. Greater-than and less-than operations follow the same rules as the strcmp() standard library function.

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.

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.

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 application 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 MtDt, 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.

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 TPU test vectors found in file filename.Vector are loaded by this script command. This command is available only for the TPU Simulator target. 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);
```

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.

Trace Script Commands

```
start_trace_stream("FileName.Trace", enum TRACE_ EVENT_OPTIONS,
            enum TRACE_FILE_OPTIONS, enum BASE_TIME,
            U32 numTrailingDigits, U32 isResetTime );
```

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, and the initial time can be set to the time at which the trace file is executed by setting the isResetTime field to non-zero.

Note that all events specified in the script command must be currently enabled within MtDt 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);
```

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.

New 3.41 Code Coverage Script Commands

An important index of test suite completeness is the coverage percentage that a test suite achieves. MtDt 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
New 3.41 verify_file_coverage_ex("MyFile.c",instPct,braPct,entPct);
New 3.41 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.

New 3.41 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.

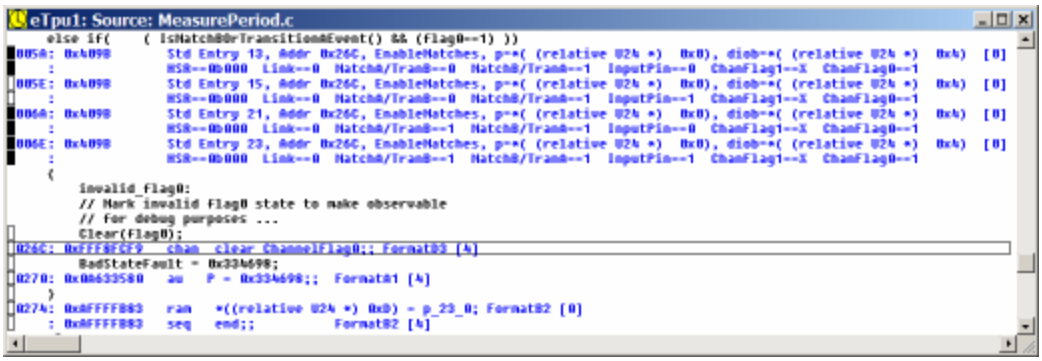
The first situation would be a valid and expected thread that is difficult to reproduce in a simulation environment. For example when measuring the time at which a rising edge occurs, it may be difficult to generate a test case for when the input pin is a zero, because a thread handler will normally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

The second situation in which it may be difficult to achieve complete event vector coverage is when there are multiple event vectors that handle an 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, the 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.

New 3.41 **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.



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 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 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.

New 3.41 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);
```

RAM Test Script Commands

RAM test script commands provide a method for verifying internal or external RAM. These tests provide both a verification and a diagnostic capability.

test_address_lines(startAddress, stopAddress, enum ADDR_SPACE, numIterations);

This command runs the address line RAM test between stopAddress and startAddress in the address space ADDR_SPACE, for the number of iterations specified by numIterations. The test attempts to provide diagnostics on any errors. For instance it will attempt to isolate specific address lines as stuck high or stuck low, and, if successful, will report the suspected problem. Note that this test verifies only those address lines that are dynamic within memory range being tested. For instance, address bit 16, which selects between 64K blocks, will not be tested unless the memory block straddles a 64K block boundary.

test_data_lines(startAddress, stopAddress, enum ADDR_SPACE, numIterations);

This command runs the data line RAM test between stopAddress and startAddress in the address space ADDR_SPACE, for the number of iterations specified by numIterations. The test attempts to provide diagnostics on any errors. For instance it will attempt to isolate specific data lines as stuck high or stuck low, and, if successful, will report the suspected problem.

test_random(startAddress, stopAddress, enum ADDR_SPACE, numIterations);

This command runs the data line RAM test between stopAddress and startAddress in the address space ADDR_SPACE, for the number of iterations specified by numIterations. This is the most robust test but it provides no diagnostic capabilities. It writes a pseudo-random number sequence to the entire memory device and then verifies that it reads back correctly. The most difficult-to-detect errors will generally be detected by this test.

test_increment(startAddress, stopAddress, startVal, enum ADDR_SPACE, numIterations);

This command runs the block increment RAM test between stopAddress and startAddress in the address space ADDR_SPACE, for the number of iterations specified by numIterations, using the start value startVal as the initial value from which to increment. This is the least robust test, but it is excellent for human-factors purposes. Folks like you and I are generally quite good at picking out breaks-in-patterns. For a typical memory error this test sets the memory to a convenient pattern so that we can easily spot errors by scrolling down through a memory dump window.

test_seam(address, enum ADDR_SPACE, numIterations);

This command runs the seam RAM test at address in the address space ADDR_SPACE, for the number of iterations specified by numIterations. This was developed to verify proper functionality of all read and write accesses at the boundary between two memory devices. All combinations of even, odd, and double-odd addresses and 8-, 16-, and 32-bit wide accesses are exercised. Note that the test assumes that all these access combinations are supported; the test will fail if this is not a valid assumption.

test_byte_order(address, enum ADDR_SPACE, numIterations);

This command runs the byte order RAM test at address <address> in the address space ADDR_SPACE, for the number of iterations specified by numIterations. It verifies the byte order of the target processor. This was developed to verify proper functionality of shared memory

between two targets that have different byte orders. A shared memory between two targets that do not have the same byte ordering can be tricky.

eTPU/TPU Channel Function Select Register Script 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 wri te h, ( #define MY_FUNC_NUM ::ETPUfunctionnumber(Pwm));
```

eTPU/TPU Channel Priority Register Script 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.

eTPU/TPU Pin Control and Verification Script Commands

eTPU input pins and TPU I/O pins configured as inputs are normally controlled using test vector files. eTPU output pins and TPU I/O pins configured as outputs are normally controlled by the eTPU/TPU 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.

eTPU Topics

```
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.)

TPU Topics

pin_hi(chanNum);
pin_lo(chanNum);
tcr2_hi();
tcr2_lo();

These commands sets either channel chanNums's (or the TCR2's) pin hi (logical 1) or lo (logical 0.)

pin_hi(0xC);
pin_lo(5);
tcr2_lo();

In this example channel 12's pin is set to a one, channel 5's and the TCR2 pin are set lo.

verify_pin(chanNumber, pinState);

This command verifies that channel chanNumbers's pin is equal to pinState (logical one or zero). If the verification fails a verification failure messages is printed to the screen. This verification failure message can be disabled from the Message Options dialog box.

verify_pin(3, 1);

In this example channel 3's pin is inspected and if it not a one a verification failure message is printed to the screen and the verification failure count is incremented.

eTPU/TPU 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 automate this second verification process. Script command capabilities include the ability to load pin transition behavior files, the ability to enable continuous verification against these models, and the ability to perform a complete verification of all recorded behavior at once.

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.

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 code.

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.

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.

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.

New 3.41 eTPU/TPU Resizing the Pin Transition Buffer

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 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 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 the *Logic Analyzer Options Dialog Box* section.

New 3.41 Thread Script Commands

The 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.

Disable Messages Script Commands

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.

eTPU System Configuration Script 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);
```

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

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;
```

eTPU Time Base Configuration Script Commands

```
write_angle_mode(Val);
write_tcr1_control(Val);
write_tcr2_control(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(4);      // System clock/8
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.)

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 is defined as a "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);
```

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

New 3.41 eTPU Channel Data Script Commands

```
write_chan_data32(ChanNum, AddrOffset, Val);
write_chan_data24(ChanNum, AddrOffset, Val);
write_chan_data16(ChanNum, AddrOffset, Val);   New 3.41
write_chan_data8 (ChanNum, AddrOffset, Val);
verify_chan_data32(ChanNum, AddrOffset, Val);
verify_chan_data24(ChanNum, AddrOffset, Val);
verify_chan_data16(ChanNum, AddrOffset, Val);   New 3.41
verify_chan_data8 (ChanNum, AddrOffset, Val);
```

This command writes channel ChanNum's data at address AddrOffset to 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 automatically generated using the following macro.

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

eTPU Channel Base 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 function variables and static local variables use this. Each channel should be given its own

In the Byte Craft eTPU "C" Compiler the amount of parameter RAM that needs to be allocated to

each channel for that eTPU function is determined as follows.

```
#pragma write h, ( #define PWM_RAM ::ETPUram(Pwm));
```

eTPU Channel Function Mode (FM) Script Command

write_chan_mode(ChanNum, ModeVal);

This command writes channel ChanNum's 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

eTPU Event Vector Script 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);
```

? System configuration commands

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 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 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.

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(ChanNum, Val);
verify_microcode_exception(ChanNum, 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.

TPU Parameter RAM Script Commands

write_par_ram(ChanNum,ParamNum,Val)

This command writes channel ChanNum's (16-bit) parameter ParamNum to value Val.

write_par_ram(3,1,0x2222);

In this example channel 3's parameter one RAM word is written to value 2222 (hex).

verify_ram_word(ChanNum,ParamNum,Val);

This command verifies that channel ChanNum's RAM word number ParamNum is equal to Val. If the verification fails a verification failure messages is printed to the screen. This verification failure message can be disabled from the Message Options dialog box. The accumulated count of verification failures is available in the Configuration Window.

verify_ram_word(0xa,3,0x1324);

In this example channel 10's parameter RAM word three is verified to be equal to 0x1324 (hex). If

the value is indeed equal to 0x1324 (hex) then the simulation continues. If the value is not equal to 0x1324 (hex) an error message is printed to the screen.

verify_ram_bit(ChanNum,ParamNum,BitNum,Val);

This command verifies that channel ChanNum's RAM word number ParamNum bit BitNum is equal to Val. Bit 15 is defined as the most significant while bit 0 is defined as the least significant. Val must be between zero and one inclusive. If the verification fails a verification failure messages is printed to the screen. This verification failure message can be disabled from the Message Options dialog box. The accumulated count of verification failures is available in the Configuration Window.

verify_ram_bit(0xa,3,14,1);

In this example channel 10's parameter RAM word 3 bit 14 is verified to be equal to 1. If the value is indeed equal to 1 then the simulation continues. If the bit is instead a zero an error message is printed to the screen. For instance if channel 10's RAM word three is equal to 4000 (hex) the verification would pass while if it were instead equal to zero the test would fail and an error message would be displayed.

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.

TPU Channel Interrupt Service Register Script Commands

clear_cisr(ChanNum);

This command clears a Channel Interrupt Service Request (CISR) from channel ChanNum. This command accepts a single argument.

clear_cisr(7);

In this example the CISR is cleared from channel 7.

clear_this_cisr();

This command clears the CISR of the active channel from within an ISR script commands file. Note that in the primary script commands file there is no channel context so this command cannot be used and use of this command generates a warning message. No arguments are required with this command.

clear_this_cisr();

Assume in this example that an ISR script commands file that is associated with TPU channel 3 executes this command. This file gets executed upon assertion of channel 3's interrupt. When this command is executed channel 3's interrupt request bit is cleared.

verify_cisr(ChanNum,Val);

This command verifies that the CISR bit from channel ChanNum is equal to Val. This command accepts a pair of arguments. Argument Val must be between zero and one inclusive. If the verification fails a verification failure message is printed to the screen. This verification failure message can be disabled from the Message Options dialog box. The count of verification failures is available in the Configuration Window.

verify_cisr(5,1);

In this example channel 5's CISR bit is verified to be a 1. If the bit is indeed a 1 then the simulation continues. If the bit is instead a 0 a verification failure message is printed to the screen. The accumulated count of verification failures is available in the Configuration Window.

TPU Host Sequence Request Register Script Commands

write_chan_hsqr(ChanNum,Val);

This command writes channel ChanNum's HSQR bits to value Val. The HSQR bits of all other registers remain unchanged.

write_chan_hsqr(8,0x2);

In this example two is written to channel 8's HSQR bits.

TPU Clock Control Script Commands

These commands control the TCR1 counter, the TCR2 input pin frequency, and the TCR2 counter.

See the Clock Control Script Command section for information on how to set the clock period.

The set_cpu_frequency(); script command has been deprecated. Instead, use the set_clk_period() script command described in the Clock Control Script Command section. A message is generated when this command is used. This message can be disabled from the Message Options Dialog Box.

write_psck(Val);

This command writes the prescaler clock source bit. Only 0 and 1 are allowed values. A 1 selects the system clock divided by 4. A zero selects the system clock divided by 32.

write_div2(Val);

This is available in TPU2 mode only. Writes the divide-by-2 control bit. Only 0 and 1 are allowed values. A 1 selects the system clock divided by 2 and bypasses the prescaler. A 0 selects the clock source specified in the PSCK bit.

write_tcr1_prescaler(Val);

This command writes the TCR1 prescaler. Valid Val values are 0, 1, 2, or 3. A 0 causes a division by 1 (no prescaler). A 1 causes a division by 2. A two causes a division by 4. A 3 causes a division by 8.

```
write_psck(1);
write_div2(0);     // Available in TPU2 mode only
write_tcr1_prescaler(2);
```

The two commands in this example set the TCR1 counter to increment at 1.0486 MHz, assuming the CPU clock is set to 16.778 MHz.

write_t2cg(Val);

This command writes the TCR2 configuration bit. Valid Val values are zero and one.

write_t2csl(Val);

This is available in TPU2 mode only. It writes the TCR2 counter clock source edge control bit. Only 0 and 1 are allowed values. This control bit along with the T2CG control bit specify the source for the TCR2 counter. With both control bits at 0, the TCR2 counter is clocked on a rising edge. With both control bits at one the TCR2 counter is clocked on both the rising and falling edges, thus effectively doubling the counter frequency. With T2CSL at 0 and T2CG at 1 the clock source is the gated system clock. With T2CSL at 1 and T2CG at 0 the TCR2 counter is clocked on the falling edge.

write_tcr2_prescaler(Val);

This command writes the TCR2 prescaler. Valid Val values are 0, 1, 2, or 3. A 0 causes a division by 1 (no prescaler). A 1 causes a division by 2. A 2 causes a division by 4. A 3 causes a division by 8.

```
write_t2cg(0);
write_t2csl(0)   // Available in TPU2 mode only
write_tcr2_prescaler(3);
```

The commands in this example set the external pin to be the TCR2 counter clock source and set the TCR2 prescaler to divide by 8. The TCR2 counter is setup to be clocked on the rising edge. These commands together set the TCR2 counter to increment on every eighth rising edge of the signal at the TCR2 input pin.

TPU3-Specific Script Commands

write_epscke(Val);

This command writes the TCR1 enhanced prescaler enable bit. Only 0 and 1 are allowed values. This control bit determines whether the TPU3's new and enhanced prescaler is used or the standard prescaler is used. With the enhanced prescaler the resolution of the clock period is controllable in 3% increments of the longest clock period.

write_epsckv(Val);

This command writes the TCR1 enhanced prescaler value. Only values between 1 and 31 are allowed. Assuming the EPSCKE bit is set, the clock frequency fed to the TCR1 prescaler is the system clock divided by the TCR1 enhanced prescaler where the enhanced prescaler is equal to EPSCKV plus 1 times 2.

```
write_epscke(1);
write_epsckv(20);
write_tcr1_prescaler(2);  // Available in all TPUs
```

The three commands in this example set the TCR1 counter to increment at 99.87 KHz, assuming the CPU clock is set to 16.778 MHz.

write_tcr2psck2(Val);

This command writes the TCR2 pre-divider prescaler. The only allowed values are 0 and 1.

```
// This, with the next command, select the system clock/8
write_t2cg(1);
write_t2csl(0)
// Set the TCR2 prescaler to divide by 8
write_tcr2_prescaler(3);
// Set the pre-divider prescaler to divide by 2
write_tcr2psck2(1);
```

The above script command sequence sets the TCR2 counter to be clocked at the system clock frequency divided by 128.

TPU Bank and Mode Control Script Commands

set_tpu_type(X,Y);

MtDt automatically determines the TPU type from the source TPU microcode, so this command is not normally required, except for the TPU3 in which TPU2 should be specified because bug in the Freescale TPU assembler prevents selection of TPU3 so this script command is required to specify a TPU3 target. In addition, there are subtle differences among the TPU1, TPU2, and TPU3 that the TPU assembler hides from the user. Use of this instruction can cause significant behavioral

deviations between MtDt and the actual TPU hardware.

This command sets the TPU type and the instruction space size. Valid types are 1 through 3, which correspond to TPU1 through TPU3, respectively. The instruction space size is expressed in 32-bit long-words. Supported combinations of types and instruction space sizes are as follows.

TPU1, 256
TPU1, 512
TPU2, 512
TPU2, 1024
TPU2, 2048
TPU3, 1024
TPU3, 2048

write_entry_bank(X);

This command writes the TPU entry bank. Valid values depend on the instruction space size. Each bank consists of 512 32-bit long-words, so the number of valid banks is equal to the instruction space size divided by 512. For the common size of 1024 there are two valid banks so the maximum valid entry bank is 1. Zero is always a valid entry bank.

```
set_tpu_type(1,1024);
write_entry_bank(1);
```

The commands in this example set the TPU type to TPU1, the instruction space size to 1024 32-bit long-words, and the entry bank to 1.

Setting the TPU Code Bank

The following code bank discussion is applicable only to TPU2 and TPU3.

There are no script commands for setting the code bank. At run-time the active code bank is determined during each time slot transition. Bits 10 and 9 from the entry table determine the code bank. The TPU assembler generally handles this field by determining the bank in which the code is located. The user has control over which bank the code is located from within the source TPU microcode. One method of specifying the bank in which the code will reside is the ORG statement. This and other methods are described in the TPU assembler literature.

TPU Match, Transition, and Link Script Commands

```
set_mrl(ChanNum);
set_tdl(ChanNum);
set_link(ChanNum);
```

These commands directly sets the Match Recognition Latch (MRL,) the Transition Detection Latch (TDL) or a Link Service Request (LSR) of channel ChanNum.

```
set_mrl(5);
set_tdl(3);
set_link(8);
```

In this example channel 5's MRL is set, channel 3's TDL is set, and a link is generated on channel 8.

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
load_isr("filename.TpuCommand", ChanNum);      // TPU-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
disable_data_intr( chanNum );      // eTPU Only
write_chan_cier(chanNum, isEnable); //  TPU 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
close_isr(ChanNum);        //  TPU only
```

These commands close the ISR script command that is associated with the channel ChanNum.

```
load_isr("ISR_5.TpuCommand", 5);
write_chan_cier(5, 1);  // Enable the TPU interrupt
wait_time(2000);
close_isr(5);
write_chan_cier(5, 0);  // Disable the TPU interrupt
```

This TPU 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.

eTPU/TPU External Logic Commands

Boolean logic that is external to the eTPU/TPU is instantiated in MtDt 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 whose output drives channel Z

eTPU Considerations

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.

TPU Considerations

Note that for the TPU, 16 is the index used for the TCR2 pin.

place_buffer(5,16);

This example instantiates a buffer follower with TPU channel 5 as the input and the TCR2 pin as the output.

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

instantiate_target_ex1(TPU_SIM, TPU_MASK, "MyTpu");
instantate_target(CPU32_SIM, "MyCpu");

This example instantiates two simulation models, a TPU and a CPU32. The name MyTpu is assigned to the TPU and the name MyCpu is assigned to the CPU32. A special "standard mask" style of TPU is generated that will load only the Standard Mask TPU microcode. 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 and/or write cycles must be subsets of those 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_data("TargetName", "BlockName", enum ADDR_SPACE,
OFF_DATA);

This 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 spaces must be a subset of the address spaces to which the referenced memory block applies.

add_non_mem_block("MyCpy", 0xFFFF, 0xFFFFFFFF, "Unused",
ALL_SPACES);
set_block_off_data("MyCpu", "Unused", 0x5A);

In this example the address space between FFFF and FFFFFFFF hexadecimal is specified to contain no memory. Read cycles to this block will return the specified off data, 5A hexadecimal, at every address. For instance, a 32-bit read in this block returns 5A5A5A5A hexadecimal.

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 and/or 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, "Protecte d",
          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);
```

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 on 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 a 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.

Automatic and Pre-Defined Define Directives

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 simulator.

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

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

Determining the Auto-Run Mode

The Simulator/Debugger 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 an 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

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.

eTPU Targets

- ? 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.

TPU Targets

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

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 simulator or debugger application 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 ADD NORMAL Tests ...
eTpuSim -pAutoRun.ETpuSysSimProject -AutoRun -IAcceptLicense -d_AUTO_TEST_AU_ALUOP_ADD_NORMAL_
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 _AUTO_TEST_AU_ALUOP_ADD_NORMAL_
load_executable("AuAluopAddNormal.gxs");
#endif // _AUTO_TEST_AU_ALUOPI_ADD_NORMAL_
```

TPU Target Pre-Defined Define Directives

The following define directives are automatically loaded and available for both TPU Simulator and TPU Debugger targets.

- ? #define CFSR0 ((U16 *) 0x0C)
- ? #define CFSR1 ((U16 *) 0x0E)
- ? #define CFSR2 ((U16 *) 0x10)
- ? #define CFSR3 ((U16 *) 0x12)
- ? #define HSQ ((U32 *) 0x014)
- ? #define HSQ0 ((U16 *) 0x014)
- ? #define HSQ1 ((U16 *) 0x016)
- ? #define HSRR ((U32 *) 0x018)
- ? #define HSRR0 ((U16 *) 0x018)
- ? #define HSRR1 ((U16 *) 0x01A)
- ? #define CPR ((U32 *) 0x01C)
- ? #define CPR0 ((U16 *) 0x01C)
- ? #define CPR1 ((U16 *) 0x01e)

Pre-Defined Enumerated Data Types

Listed below are the predefined enumerated data types used on a variety of script commands. Clicking on the desired enumerated data type will access a listing.

- ? Script FILE_TYPE enumerated data type
- ? Script FILE_OPTIONS enumerated data type
- ? Script BASE_TIME enumerated data type
- ? Script TRACE_OPTIONS enumerated data type
- ? Script TARGET_TYPE enumerated data type
- ? Script ADDR_SPACE enumerated data type
- ? Build Script READ_WRITE enumerated data type
- ? Build script Register enumerated data type
- ? eTPU Register enumerated data types
- ? TPU Register enumerated data types
- ? CPU32 Register enumerated data types
- ? CPU16 Register enumerated data types

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, }

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

Save Trace File 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 {
STEP, EXCEPTION, MEM_READ, MEM_WRITE, DIVIDER,
// TPU Targets, only
TPU_TIME_SLOT, TPU_NOP, TPU_PIN_TOGGLE,
TPU_STATE_END,
// Hardware debugger only options
FREE_RUN,
// 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
}
```

Base Time 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,
}
```

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, SIM32, BDM32, SIM16, SIM32,  }
```

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,
// CPU16 & CPU32
CPU16_CODE_SPACE, CPU16_DATA_SPACE,  CPU16_UNUSED_SPACE,
CPU32_USER_CODE_SPACE, CPU32_SUPV_CODE_SPACE,
CPU32_USER_DATA_SPACE, CPU32_SUPV_DATA_SPACE,
CPU32_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
```

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_READ32,
RW_WRITE8,  RW_WRITE16, RW_WRITE32,
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   ALL_READS   RW_READ8 + RW_READ16 + RW_READ32
#define   NON_ACCESS32S   -  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- and 16-bit read and write cycles.

Register Enumerated Data Type

Each target has its own enumerated data types for its own registers. The following is a list of the enumerated registers for each target.

- ? eTPU Enumerated Registers
- ? TPU Enumerated Registers
- ? CPU32 Enumerated Registers
- ? CPU16 Enumerated Registers

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, };

enum REGISTERS_U24 {
REG_A, REG_B, REG_C, 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, },

enum REGISTERS_U8 {
REG_LINK},

enum REGISTERS_U5 {
REG_CHAN, },

enum REGISTERS_U1 {
REG_Z, REG_C, 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   );
```

TPU Register Enumerated Data Types

The TPU register enumerated data types provide the mechanism for referencing some of the TPU 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.

There are no enumerated types for the TPU's 8-bit registers, and the TPU has no 32-bit registers.

The following enumeration provides the mechanism for referencing the TPU's 16-bit registers. Note that this covers only those TPU registers that are newly accessible with this software version. Future software versions will make greater use of this enumerated data type.

```
enum REGISTERS_U16 {
REG_TCR1, REG_TCR2,  //  Counter registers
};
```

CPU32 Register Enumerated Data Types

The CPU32 register enumerated data types provide the mechanism for referencing the CPU32 registers. These enumerated data types are used in commands that reference the CPU32 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 CPU32's 8-bit registers.

```
enum REGISTERS_U8 {
REG_CCR,  //  Condition Code Register
};
```

The following enumeration provides the mechanism for referencing the CPU32's 16-bit registers.

```
enum REGISTERS_U16 {
REG_SR  //  Status Register
};
```

The following enumeration provides the mechanism for referencing the CPU32's 32-bit registers.

```
enum REGISTERS_U32 {
REG_PC,  REG_VBR,    // Program Counter, Vector Base Register
REG_USP,  REG_SSP,   // User and Supervisor Stack Pointers
REG_SFC,  REG_DFC,   // Alternate Function Code Registers
REG_DO,  REG_D1,  REG_D2,  REG_D3,  //  Data register D0 to D3
REG_D4,  REG_D5,  REG_D6,  REG_D7,  //  Data register D4 to D7
REG_AO,  REG_A1,  REG_A2,  REG_A3,  //  Addr register A0 to A3
REG_A4,  REG_A5,  REG_A6,  REG_A7,  //  Addr register A4 to A7
};
```

CPU16 Register Enumerated Data Types

The CPU16 register enumerated data types provide the mechanisms for referencing the CPU16 registers. These enumerated data types are used in commands that reference the CPU16 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 CPU16's 4-bit registers.

```
enum REGISTERS_U4 {
REG_XK,  REG_YK,  REG_ZK, REG_SK ,  REG_PK,REG_EK
};
```

The following enumeration provides the mechanism for referencing the CPU16's 8-bit registers.

```
enum REGISTERS_U8 {
REG_A,  REG_B, REG_XMSK,  REG_YMSK,
};
```

The following enumeration provides the mechanism for referencing the CPU16's 16-bit registers.

```
enum REGISTERS_U16 {
        REG_D,  REG_E, REG_IX,  REG_IY,  REG_IZ,
        REG_SP,  REG_PC, REG_CCR,
        REG_HR, REG_IR, REG_AMLO,
};
```

The following enumeration provides the mechanism for referencing the CPU16's 32-bit register.

```
enum REGISTERS_U32 {
        REG_AMHI,  //  MAC Accumulator MSB, bits [35:16]
};
```

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 File 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 script commands described in the File 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 MtDt and the trace buffer has not overflowed when generating a trace file from the buffer.

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 free trace file parse source code available from ASH WARE. The public methods in the TraceParser class are fully documented and will remain stable for all future releases.

The trace file parser's source code is released to the public domain under the GNU licensing agreement. You should familiarize yourself with the restrictions imposed by the GNU license before using the source code.

ASH WARE intends to provide as a service the generation of parser file post processing capabilities. For instance, TPU latency calculations such as minimum, maximum, average, and standard deviation would be one excellent application. All such utilities will remain in the public domain.

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.

TEST VECTOR FILES

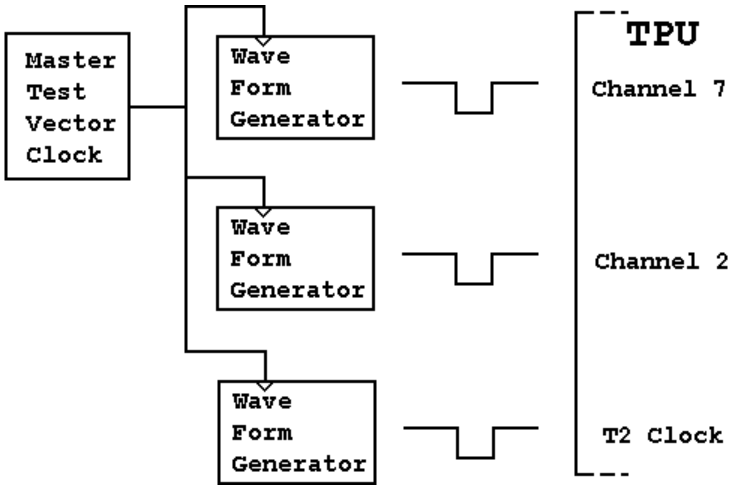
Overview

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.

The TPU simulation engine provides a complex test vector generation capability. This allows the user to exercise the simulated TPU with signals similar to those found in a typical TPU environment. Test vector scripts 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 TPU's I/O pins. Note that this capability is available only for the TPU 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 TPU, 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 TPU Simulator nodes. As MtDt executes, these test vectors are driven unto the specified nodes.

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.

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 backslash comments. A comment field is started with two sequential backslash characters, //. All text starting from the backslashes 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.
define UART CH5

Test Vector "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

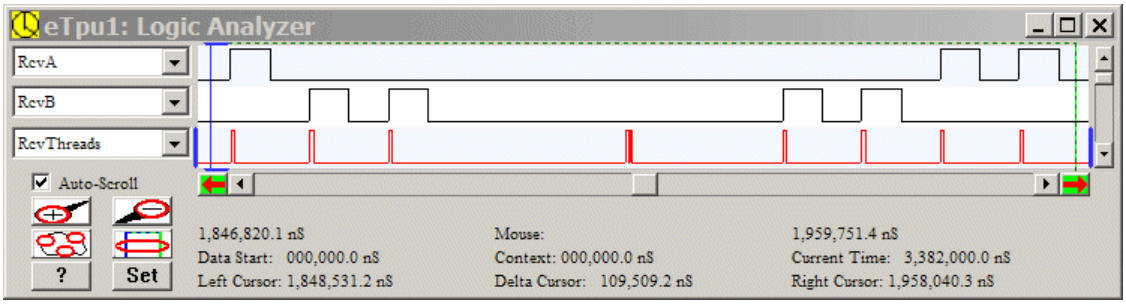
Thread Activity Nodes

Thread activity can be extremely useful in both understanding and debugging eTPU and TPU functions. The 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



Test Vector "Group" Command

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

The group command assigns multiple nodes NODEs 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 insensitive. Up to 30 nodes may be grouped.

define ADDRESS1 ch5
define ADDRES S2 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.

Test Vector "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 insensitive. 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.

Master Test Vector Clock "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.

Test Vector "Wave" Command

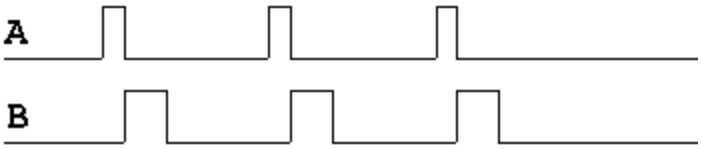
The wave command creates and defines a new wave form generator. There is no limit to the number of wave commands that may be used in a test vector file. The number of wave form generators is equal to the number of wave commands found in the test vector file.

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.

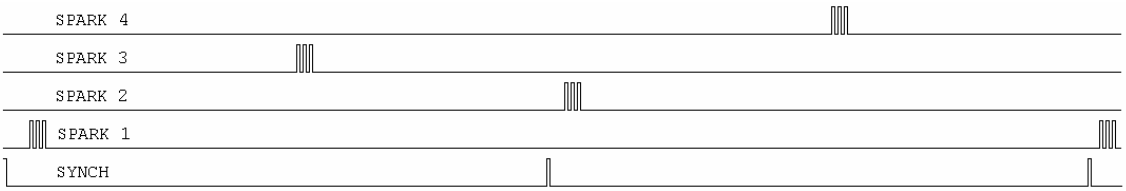


Test Vector Engine Waveform Example

```
// 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 MtDt'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
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 or 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.

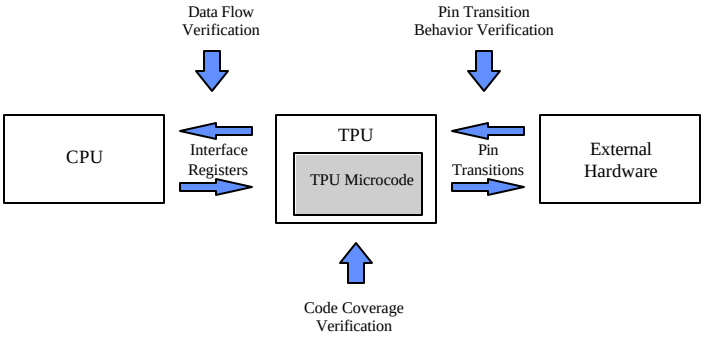

SPARK 4
SPARK 3
SPARK 2
SPARK 1
SYNCH

FUNCTIONAL VERIFICATION

Overview

The functional verification capabilities are currently intended solely for the eTPU/TPU Stand-Alone Simulators. Future versions of this software will extend these capabilities so they are useful for other targets and in a multiple target environment.

Version 2.0 of MtDt has an increased emphasis on functional verification and especially the automation of these functional verification capabilities using script commands. 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.



Data flows between the TPU and the CPU via interface registers. Data flow verification provides the capability of verifying this data flow.

Pin transitions are generated by the TPU 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.

While version 1.0 of MtDt was primarily a development tool, version 2.0 has an increased emphasis

on capabilities that address the verification and maintenance stages. As such, while version 1.0 answers the development question of "What is the TPU doing?" version 2.0 answers the verification question of "Are the tests complete?" and the maintenance question of "What has changed?"

Version 2.0 also emphasizes the automation of these tests and the generalization of results in terms of simple pass/fail criteria. In fact, two pass/fail criteria can be defined for an entire test suite.

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 parameter RAM data flow is verified using the verify_ram_word(X,Y,Z) and verify_ram_bit(X,Y,Z,V) script commands described in the *TPU Parameter RAM Script Commands* section.

The data flow across the CISR register is verified using the verify_cisr(X,Y) script command described in the *TPU Channel Interrupt Service Register 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 until MtDt has 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 until MtDt runs 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);
```

Pin Transition Behavior Verification

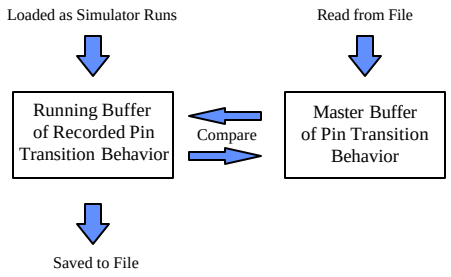
Pin transition behavior verification is one of the verification capabilities for which an overview is given in the *Functional Verification* chapter. Pin transition behavior verification capabilities include the ability to save recorded pin transition behavior to pin transition behavior (.bv) files, the ability to load saved pin transition behavior files into MtDt, and the ability to verify that the most current pin

transition behavior matches the saved behavior.

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 MtDt is on the automation of these modeling capabilities through script commands, although the capabilities are also available directly from the menus.

Pin Transition Buffers

There are two pin transition storage buffers in MtDt as shown in the following diagram.



The running buffer is filled as MtDt runs. Whenever a pin transition occurs information regarding this transition is stored in this buffer. This buffer can then be saved by selecting the Behavior Verification, Save submenu from the File menu.

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. These files are generated by selecting the Behavior Verification, Open submenu from the File menu, or by 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 either by the enable_continuous_behavior() script command or from the Enable Continuous Verification submenu from the Options menu. 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 or from the Options menu.

A count of failures is displayed in the Configuration window. This count is incremented whenever a behavior verification failure occurs.

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. A warning is generated if the user attempts to save a master behavior verification file when the buffer has rolled over.

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.

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 non-branch instruction that has not been executed.
- A blue box indicates a branch instruction that has not been executed.
- A green box indicates a branch instruction where the branch path has been traversed.
- A red box indicates a branch instruction where the non-branch path has been traversed.
- A white box indicates a fully covered instruction.

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 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 1996-1997 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
```

Regression Testing

Regression Testing supports the ability to launch MtDt 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 MtDt once all the tests have been run.

Upon termination, MtDt 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 MtDt multiple times, each time with a different set of tests specified, and by checking the error level each time MtDt terminates, multiple tests can be run automatically and a single pass (meaning all 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.

New 3.41 Command Line Parameters

When launching any MtDt application, such as a debugger or simulator, the following command line options support Regression Testing.

- ? -h Prints a list of all the command line options
- ? -p<ProjectFileName> Loads the specified project file
- ? -s<ScriptFileName> Loads the specified script file (single target)
- ? -s<TargetName>@@<ScriptFileName> Loads a script file into a specific target
- ? -d<DefinedText> In script commands file, #define DefinedText
- ? -lf5<LogFileName.log> Logs messages to end of this file.
- ? -tn<TestName> Assigns a name to a test (used in log file.)
- ? -Quite Disables display of dialog boxes
- ? -IAcceptLicense Skips the startup license agreement dialog box
- ? -AutoRun Sets the target system to running

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 have 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.

If the user closes the application manually, for instance by selecting "close" from the system menu, then the error level is set to non-zero.

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

path=%path%;"C:\Program Files\ASH WARE\TPU Simulator\;"

TpuSimulator.exe –pTest1.TpuSimProject –sTest1.TpuCommand -AutoRun -IAcceptLicense
if %ERRORLEVEL% NEQ 0 ( goto errors )

TpuSimulator.exe –pTest2.TpuSimProject –sTest2.TpuCommand -AutoRun -IAcceptLicense
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

New 3.41 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 do 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 envoked using a combination of two command line options, shown below.

? -LF5MyLogFile.log -Quite

The first command, **-L5MyLogFile.log** specifies that message are ***appended to the end*** of a log file named, "MyLogFile.log."

The second command, **-Quite**, 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 **–Quite** 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"

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.

EXTERNAL LOGIC SIMULATION

External logic simulation is TPU centric since it may be used solely on the I/O pins of TPU targets and only on an intra-TPU 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.

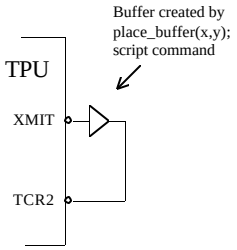
- ? There are only two logic states, one and zero.
- ? The logic is simulated with a single pass per step (each step is two CPU clocks).
- ? All output states are calculated before they are written, and therefore all calculations are based on the pre-calculated states. Thus it takes multiple passes for 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 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);

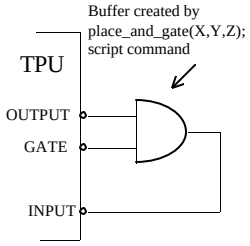


Example 2: Multi-Drop Communications

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);



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. 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.

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 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.

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.

OPERATIONAL STATUS WINDOWS

Overview

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 resized, scrolled, iconized, minimized, and maximized. Multiple instances of each window may be opened. Below is a listing of all operational status windows.

Window Groups

Common Windows

The following windows are available on multiple target.

- ? New 3.41 Threads Window
- ? Source Code File Windows
- ? Script Commands File Windows
- ? Watch Window
- ? Local Variables Window
- ? Call Stack Window
- ? Trace Window
- ? Complex Breakpoint
- ? Memory Dump Window
- ? Timers Window
- ? Logic Analyzer

eTPU Specific Windows

The following windows are available on eTPU Simulator targets.

- ? eTPU Channel Function Frame
- ? eTPU Configuration Window
- ? eTPU Global Timer and Angle Counters Window
- ? eTPU Host Interface Window
- ? eTPU Channel Window
- ? eTPU Scheduler Window
- ? eTPU Execution Unit Registers Window

TPU Simulator Specific Windows

The following windows are available on TPU Simulator targets.

- ? TPU Configuration Window
- ? TPU Host Interface Window
- ? TPU Scheduler Window
- ? TPU Microsequencer Registers Window
- ? TPU Execution Unit Registers Window
- ? TPU Channel Window
- ? TPU Parameter RAM Window
- ? Logic Analyzer

683xx Hardware Debugger Windows

The following windows are available for 683xx Hardware Debugger targets.

- ? 683xx Hardware Debugger Configuration Window
- ? 683xx ASH WARE Hardware Window

System Integration Module Windows

- ? SIM Main Window
- ? SIM PortsWindow
- ? SIM Chip SelectsWindow

Queued Serial Module Windows

- ? QSM Main Window
- ? QSM Port Window
- ? QSM QSPI Window
- ? QSM SCI (UART) Window

RAM and ROM Windows

- ? Masked ROM Submodule Window (68336/68376)
- ? Standby RAM Submodule Window (68336/68376)
- ? Static RAM Submodule Window (68338)
- ? TPU Emulation RAM Window (68332, 68336, 68376)

Timer Processor Unit Windows

- ? TPU Main Window
- ? TPU Host Interface Window
- ? TPU Parameter RAM Window

General-Purpose Timer Windows

- ? GPT Main Window
- ? GPT Input Captures Window
- ? GPT Output Compares Window
- ? GPT Pulse Accumulation Window

? GPT Pulse Width Modulation Window

Counter Timer Module 4 and 6 Windows

? CTM4/CTM6 Bus Interface and Clocks Window

? CTM4/CTM6 Free-Running Counter Submodule Window

? CTM4/CTM6 Modulus Counter Submodule Window

? CTM4 Double-Action Submodule Window

? CTM4 Pulse Width Modulation Submodule Window

? CTM6 Single-Action Submodule Window

? CTM6 Double-Action Submodule – Modes Window

? CTM6 Double-Action Submodule – Bits Window

Miscellaneous Windows

? Real Time Clock Window

? Parallel Port I/O Submodule Window

TouCAN Windows

? TouCAN Main Window

? TouCAN Buffers Window

Queued Analog to Digital Converter Windows

? QADC Main Window

? QADC Ports Window

? QADC Channels Window

CPU32 Simulator Specific Windows

The following window is available for CPU32 Simulator targets.

? CPU32 Simulator Configuration Window

? CPU32 Simulator Busses Window

? CPU32 Simulator Interrupt Window

Universal CPU32 Windows

The following windows are available for both the CPU32 hardware and the CPU32 Simulator targets.

? CPU32 Registers Window

? CPU32 Disassembly Dump Window

CPU16 Simulator Windows

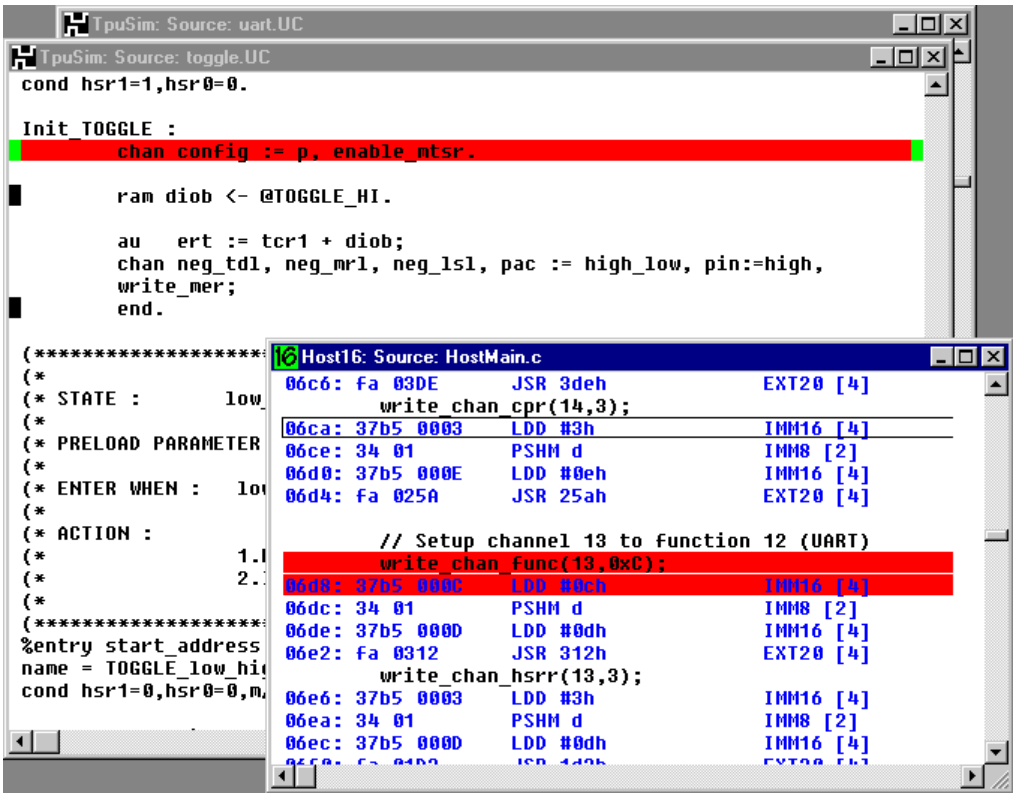
The following windows are available for CPU16 Simulator targets.

? CPU16 Simulator Configuration Window

? CPU16 Register Window

? CPU16 Disassembly Dump Window

Source Code File 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

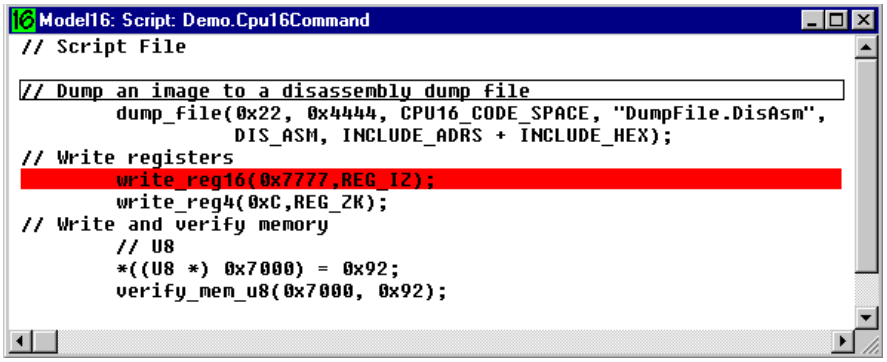
To make visible the dis-assembled instructions associated with each line of source code, select the Toggle, Assembly Mixed submenu from the Options menu. For TPU targets, the 32-bit

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.

Script Commands File Windows



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

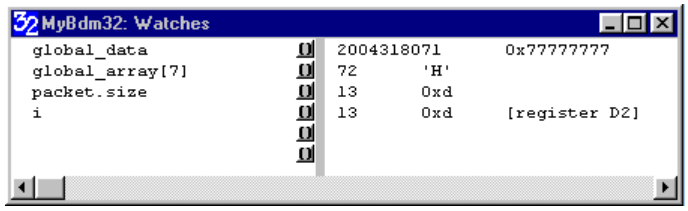
These capabilities are available starting in version 3.20. Script commands file windows support breakpoints. Similar to source code windows, breakpoints can be set, cleared, enabled, and

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.

Watches Window



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 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 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:

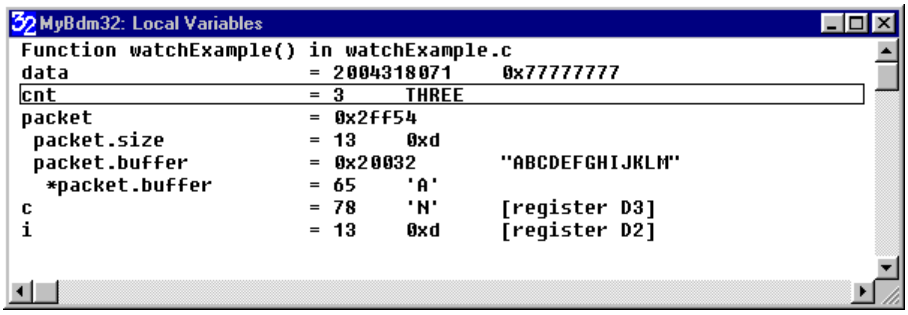
- ? Pointers can be dereferenced with the * operator.

? 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.

Local Variables Window



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.

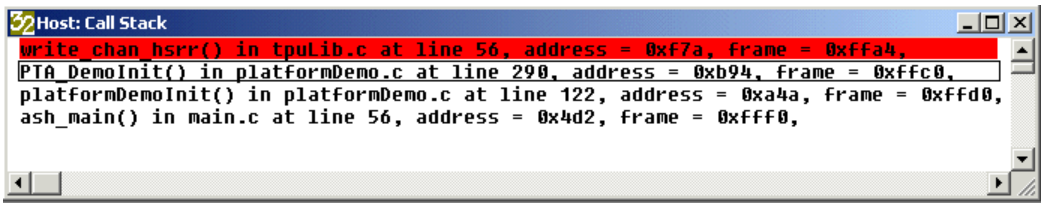
Global variables are not displayed within this window. See the description of the Watches window for information on how to display global variables.

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 unto 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.

Call Stack Window

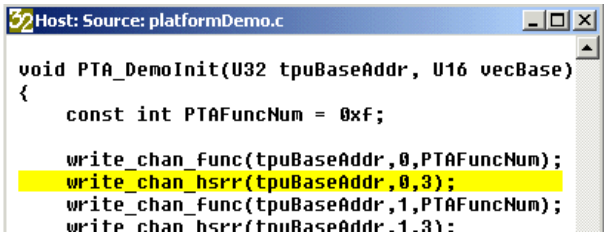


The Call Stack window displays the function, source code file name and line number, address, and stack frame pointer associated with stacked functions as shown above.

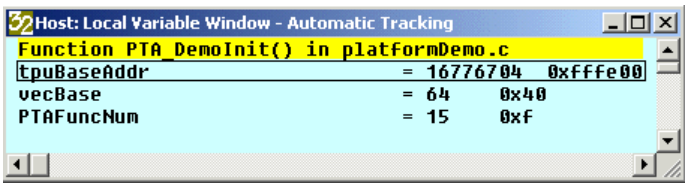
Call Stack, Local Variables, and Source Window Automation

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.



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.



New 3.41 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.

eTpu1: ThreadStats: ARINC_RX [func=1]

	THREAD Addr	Cov	Cnt	STEPS WC	Tot	WC Time	
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.

eTpu1: Source: ..\eTPU\etpuc_arinc_rx.c

```
//*****************************************************************************
//  Received an expected data bit, bits 24...31
    else if (IsTransitionAEvent() && (flag0==0) && (flag1==1))
    {
        RECEIVED_DATA_BIT_24_31:
        ClearTransLatch();

        shift_reg.dataA.data31_24>>=1;
        shift_reg.dataB.data31_31 = chan_reg.dataC.data0_0;

        if(
        {
```

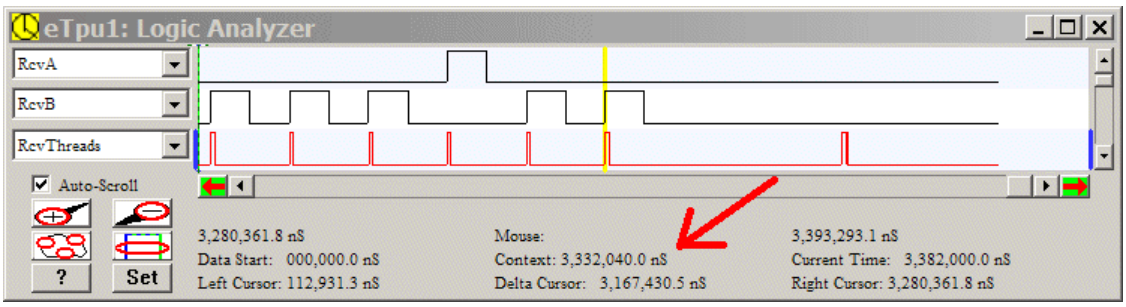
eTpu1: ThreadStats: ARINC_RX [func=1]

	THREAD Addr	Cov	Cnt	STEPS WC	Tot	WC Time	
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

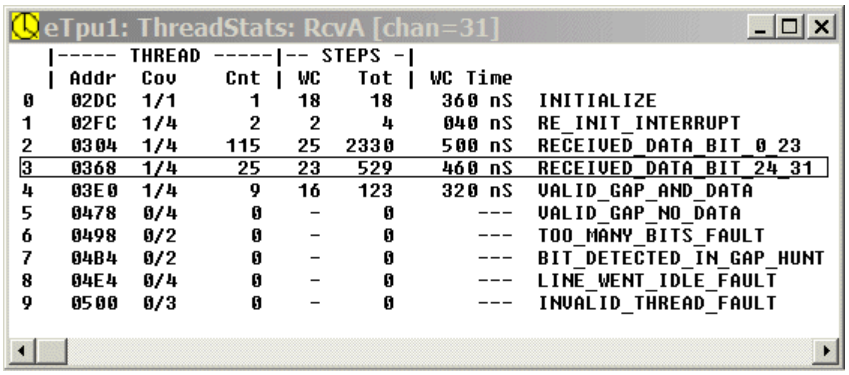
Re-Executing the Worst Case Thread

If the thread has executed at least once, such that there is a worst case thread time 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.



	THREAD			STEPS			
	Addr	Cov	Cnt	WC	Tot	WC Time	
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 that 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 ¼ 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.

eTpu1: ThreadStats: RcvA [chan=31]

	Table Addr	|-- THREAD -- Addr	Cnt	|-- STEPS - 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 response combinations it becomes clear that the simulation run on this window is not fully excersizing the function, and as such the event vector coverage is poor.

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 00002D9BA
  32-Bit Read using Supervisor Data Space at address 2500,   0x590 (1424) 007,472.5 00002D9BD
//*************************************************
  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,   0xfffe (-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 FFBC,   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,   0xfffe1e (16776734) 007,474.2 00002D9E7
  32-Bit Write using Supervisor Data Space at address FFAC,   0xfffe1e (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,   0x6c0a (27658) 007,110.0 00002B656
 Time Slot      0  007,110.0  0001  0001    00002B656
  32-Bit Read using Program Space at address 00A8,   0xac1fffff (-1407188993) 007,110.4 00002B660
  Step          0  007,110.4                00002B660 00A8
  32-Bit Read using Program Space at address 00AC,   0xa40dffff (-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.

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

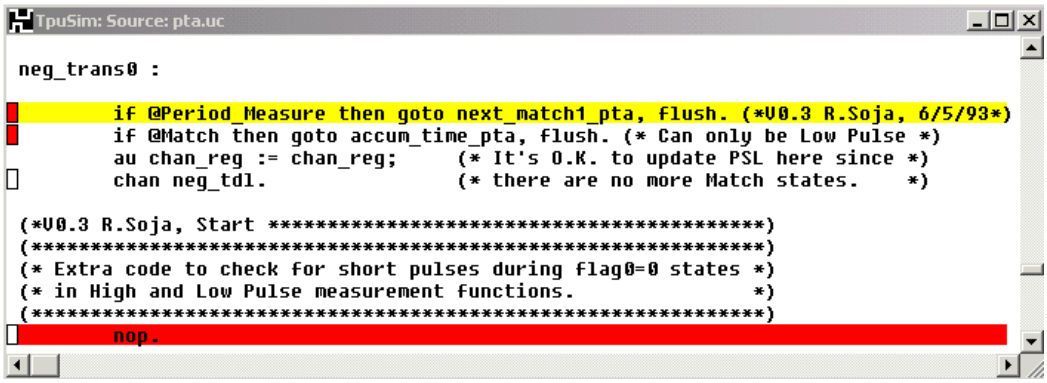
The TPU simulation model provides several additional events that can be viewed in the Trace window. These are in addition to the normal data flow between TPU and memory and opcode execution, etc., that are available for most targets. These events include execution steps, time state transitions, active 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.

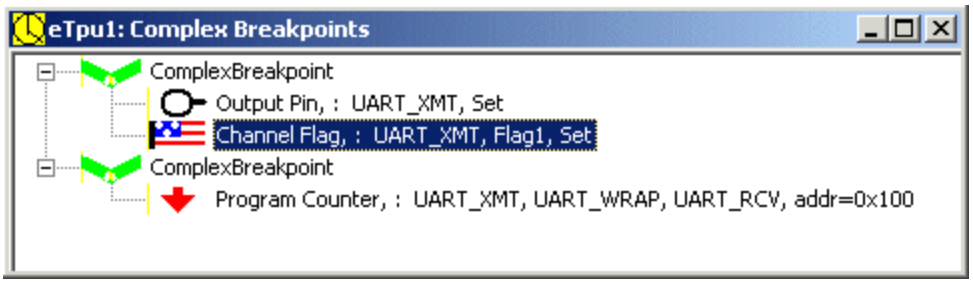
When all pending service requests from a particular priority level have been serviced, the TPU negates the service grant bits associated with those channels. This requires four CPU clocks. For this event MtDt displays the timestamp and the states of the SRL and the SGL.

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.



Complex Breakpoint Window

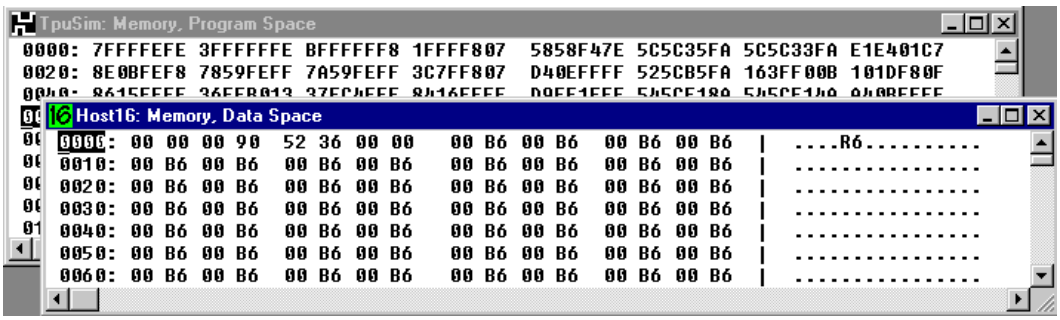


Complex breakpoints are added, removed, and modified from within the complex breakpoint window. Complex breakpoints support the ability to halt the target on the occurrence of one or more combinations of conditions. Each complex breakpoint operates independently of all other complex breakpoints.

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.

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.

Timers Window

Model16: Timers

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.

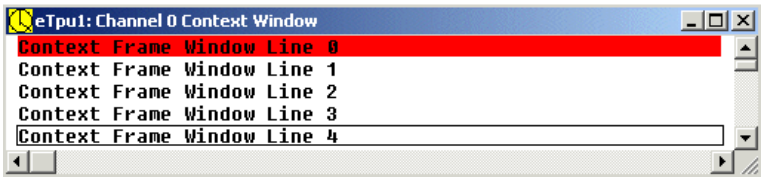
The Start Address and Stop Address fields are to the right of the State field. These fields contain the addresses that, when accessed by the target, cause the state to change. For instance, a timer in the armed state changes to the started state when the start address is executed. Then, when the stopped 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.

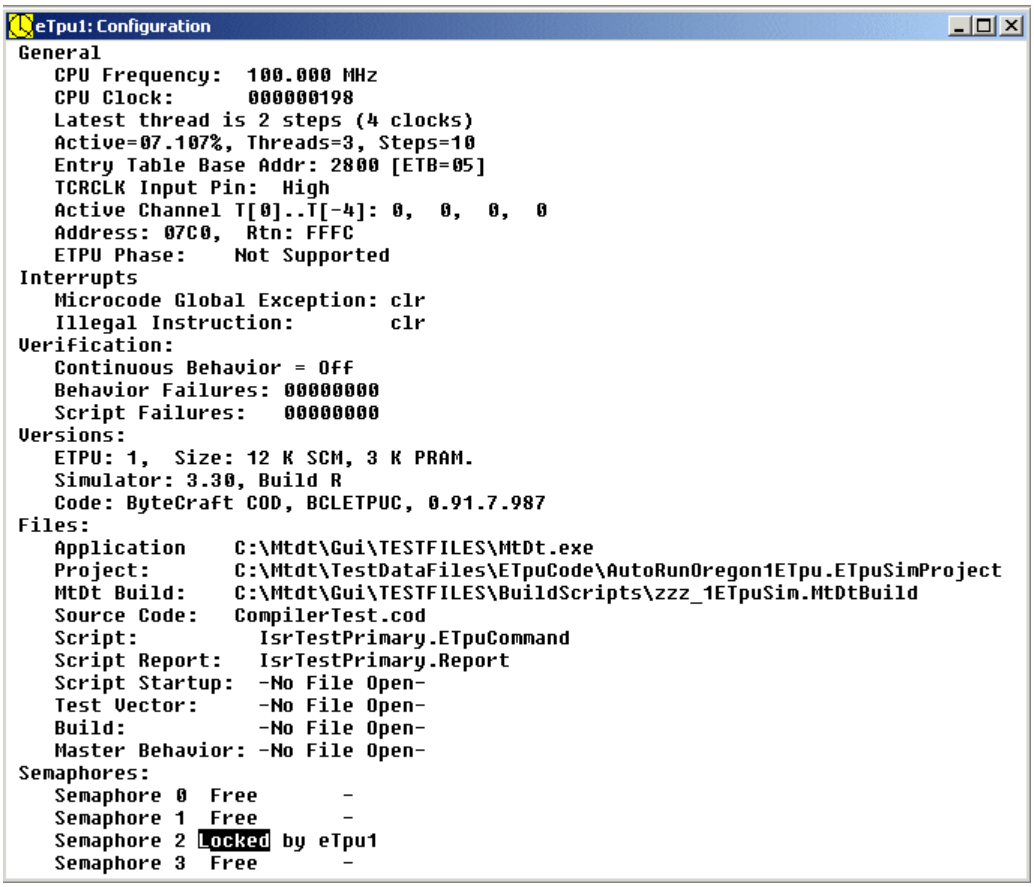
Note that the Last Ticks and This Ticks fields use the target's current clock period in the calculation. Only the start time and stop time are retained by the timers. This means that if the clock period changes during the course of a calculation, these clock tick calculations will not be correct. Note 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.

eTPU Channel Function 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.

eTPU Configuration Window



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 shows 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.

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.

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.

The eTPU version, size of its Shared Code Memory (SCM), and parameter RAM are shown. The eTPU simulation engine build are shown. The code generation (generally the Byte Craft eTPU "C" compiler and version numbers are shown, when available.

The simulator executable with full path, project file, and MtDt build file, eTPU source code file are displayed. The primary script file along with its 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.

The state (locked or free) of each of the four semaphores, along with an indicator of which eTPU is locking the semaphore, is displayed.

eTPU Global Timer and Angle Counters Window

```
eTpu1: Counters
-------------- Module Configuration Register (MCR) -----
MCR  00050001 | GTBE     0..0  |  Time bases in both engines are ENABLED
-------------- Time Base Configuration (TBCR)-----------
TBCR 42018004 | 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   5 [TCR1P=04]
              |                |  TCR1 Period: 100.0 nS
-------------- Tooth Program Register (TPR) ------------
TPR      003B | 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  |  03B + 1 Angle ticks in the current physical tooth
              |                |  60 (dec) angle ticks in each tooth tick
-------------- Tick Rate Register (TRR) ----------------
TRR    05F780 | INT     17..9  |  02FB Integer part of TCR1 clocks in one angle tick
              | FRAC     8..0  |  180   Fractional part of TCR1 clocks in one angle tick
-------------- Timer Counter 2 (TCR2) ------------------
TCR2   00028F | Rotations      |  0  Current Rotations/Tooth Count Resets
              | Tooth Ticks    |  34 Current tooth tick counts
              | Angle Ticks    |  55 Current angle tick counts
              | Angle          |  349.17  Current Angle
-------------- Analyses --------------------------------
Mode:  Normal    (PLL operating, next physical tooth not yet detected)

TCRCLK Physical Tooth Detection Window:
       Open, a physical pin transition can be detected
TCRCLK Last Tooth
       Open, a physical pin transition can be 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 displays 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 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 TCR2 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 rotation, which is every 720 degrees in a typical car. The number of such rotations 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 720 degrees per rotation.

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.

eTPU Host Interface Window

eTpu1: Host Interface

	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	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	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	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
CDTROSR:	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

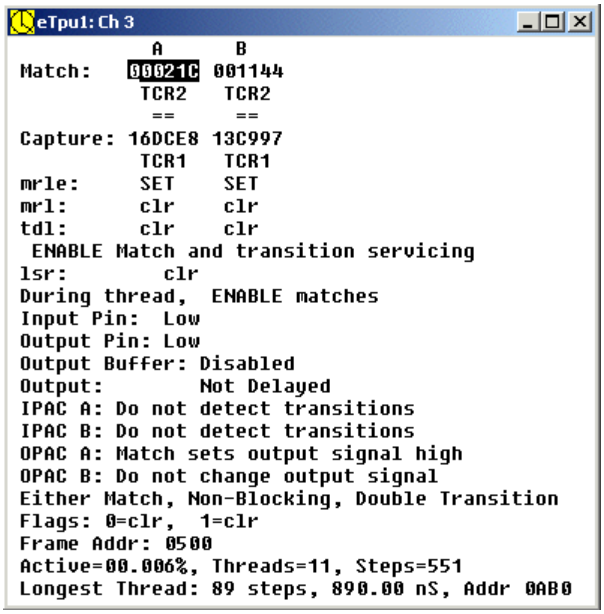
For each channel the following information is shown. The Channel Function Select Register (CFSR) shows the eTPU Function, the Channel Priority Register (CPR) shows the priority for that channel, the Host Service Request Register (HSRR) shows the state of the service request. The Function Mode (FM) shows 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 CDTROSR 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.

eTPU Channel 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. The following information is displayed. Right click the mouse when inside this window to specify or change the window flavor.

The value of the 24-bit Match and Capture registers for both action units are displayed. The global counter (either TCR1 or TCR2) that is used for the match comparison and also for the capture are displayed. The match comparators condition (either equals only or greater than and equals) is displayed.

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. Whether or not matches and transitions result in a service request is displayed.

The LSR field shows if there is a pending link into this channel. During a thread matches for the channel being serviced can be enabled or disabled and this is displayed.

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 Predefined Channel Mode (PDCM) is shown. Each channel has two flags and these are shown. Each channel can have its own private variables within the Channel Function Frame and the Channel Parameter Base Address (CPBA) of the channel within this frame is shown.

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.

eTPU Scheduler Window

```
eTpu1: Scheduler
       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
Prty:  -  -  -  -  -  -  -  -   -  -  - HI HI HI HI HI  HI HI HI HI HI HI HI HI  HI HI HI HI HI  - HI HI
Srl:   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  0  0
Sgl:   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
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  0
LSR:   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
M/TSR: 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  0  1
Time Slot Assignments: Med  Hi  Lo  Hi Med  Hi  Hi  ...
```

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

eTPU Execution Unit Registers Window

```
eTpu1: Exec Unit, AU
tcr: 000000 000000
ert: 000000 000000
p:     64 C3226C
diob:     C22ABD
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.

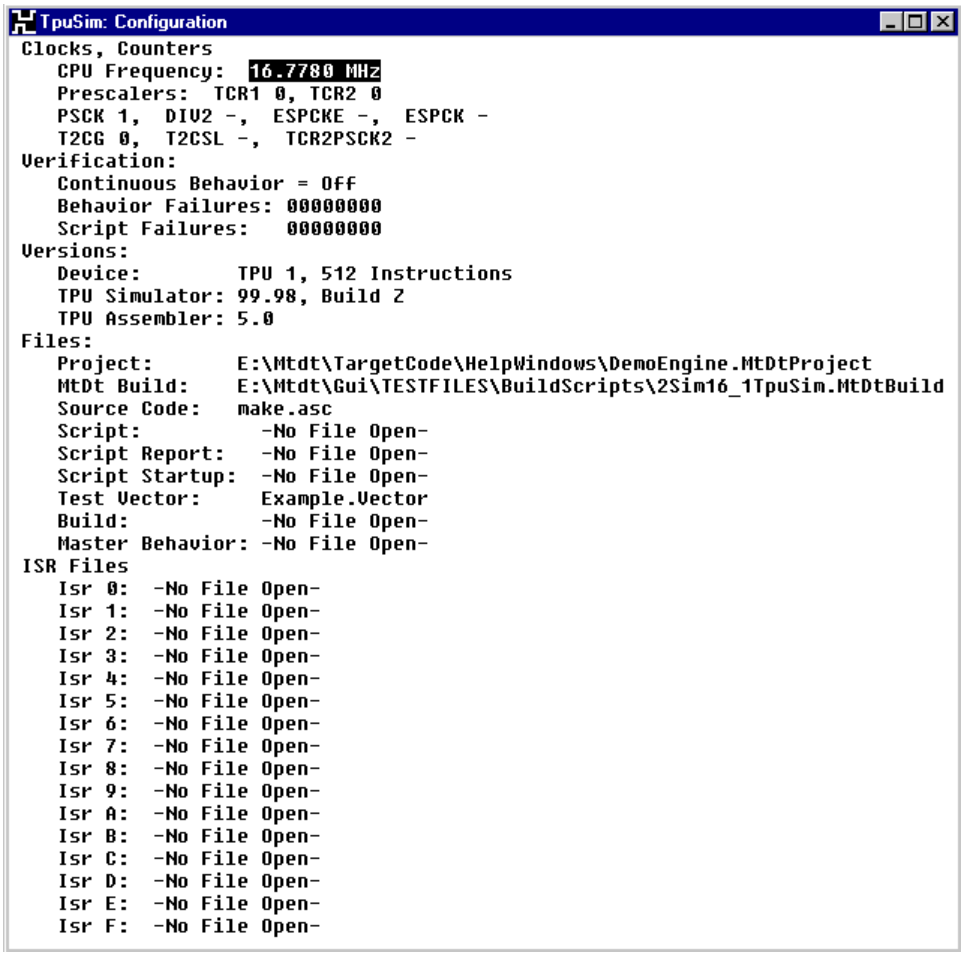
The execution unit's A Bus source, B Bus source, and result bus are shown. These are the buses internal to the execution unit.

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.

TPU Configuration Window

Availability: TPU Simulation Target



Clocks, Counters

The (simulated) CPU clock frequency is displayed. The TCR1 prescaler, TCR2 prescaler, PSCK control bit, and T2CG control bit are displayed.

When MtDt is in TPU2 mode the DIV2 and T2CSL control bits are displayed. These control bits,

which are new to the TPU2, add capabilities to the TCR1 and TCR2 counters. These control bits are automatically hidden when in TPU1 mode since they apply only to the TPU2. These bits are controlled using script commands described in the *TPU Clock Control Script Commands* section.

When MtDt is in TPU3 mode the enhanced prescaler enable bit (ESPCKE) the enhanced prescaler bit (ESPCK), and the TCR2 pre-divider prescaler enable bit (TCR2PSCK2) are displayed. These control bits, which are new to the TPU3, add capabilities to the TCR1 and TCR2 counters. These control bits are automatically hidden when in TPU1 or TPU2 mode since they apply only to the TPU3. These bits are controlled using script commands described in the *TPU Clock Control Script Commands* section.

Verification

An indication of whether behavior verification is enabled or disabled is displayed. The count of the number of behavior verification errors that have occurred is displayed. See the *Pin Transition Behavior Verification* section.

The count of the number of user-defined verification tests that have failed since the last MtDt reset is displayed. User-defined tests are part of script commands files. See the *Script Commands Groupings* section for a description of the various script commands that support user-defined tests

Versions

Version information on the device being simulated (TPU1, TPU2, TPU3), the Simulator, and the microcode assembler is listed.

Files

The name of the project file is displayed. See the *Project Sessions* chapter for a detailed explanation of this capability.

The name of the MtDt build script file is displayed. See the *Full-System Simulation* chapter and *the MtDt Build Script Commands File* chapter for detailed explanations of the available capabilities and format of this file.

The names of the open source microcode, primary script, primary script report, startup script, and test vector files are displayed. These files are listed relative to the path of the open project file.

The auto-build file name is displayed. The auto-build file allows direct building of the executable image from within MtDt and is covered in the *Auto-Build Batch File Options Dialog Box* section.

The pin transition behavior verification file name is displayed. This new feature supports the verification of TPU behavior against previously-saved pin transition behavior files. This allows the user to make changes to a test suite and automatically determine which pin transition behaviors have changed and which behaviors have stayed the same. See the *Pin Transition Behavior Verification* section.

ISR Files

The currently loaded script commands ISR files associated with each TPU interrupt are listed. Script commands files can be associated with TPU interrupts. When the interrupt associated with a particular TPU channel becomes asserted the ISR script commands file associated with that channel gets executed. Script commands ISR files are loaded using script commands. See the *ISR Script Commands Files* section for a description.

TPU Host Interface Window

Availability: TPU Simulation Target

```
MyBdm32: TPU - Host I/F
        F  E  D  C  B  A  9  8  7  6  5  4  3  2  1  0
CFSR:   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0     0000,0000,0000,0000
CPR:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     0000,0000
HSRR:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     0000,0000
HSQR:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     0000,0000
CISR:   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0     0000
CIER:   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0     0000
```

This window displays common TPU/CPU registers whose function is primarily to control. These registers include the Channel Function Select Registers (CFSRs), Channel Priority Registers (CPRs), the Host Service Request Register (HSRR), the Host Sequence Registers (HSQRs), and the Channel Interrupt Service Request Register (CISR).

Each register is broken down by TPU channel so that the value for each channel is easily found. All register values are displayed in both binary and hexadecimal formats.

The user can modify the value of each of these registers in a repeatable fashion via the script commands file. Alternatively, these registers can be modified directly using the keyboard and mouse. See the *Script Commands File Window* section for an explanation of this capability.

The TPU affects the states of the registers in the normal execution of its microcode. For example, the HSRR bits for a TPU channel are cleared when the microengine services that channel.

TPU Scheduler Window

Availability: TPU Simulation Target

```
TpuSim: Scheduler
        F  E  D  C  B  A  9  8  7  6  5  4  3  2  1  0
Prty:   H  H  H  -  -  -  -  -  -  -  -  H  H  H  H  H
Srl:    0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0
Sgl:    0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0
HSRR:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
LSR:    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
M/TSR:  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0
Time Slot Assignments: Med  Hi  Hi Med  Hi  Lo  Hi  ...
```

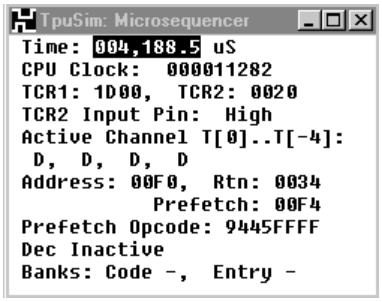
The TPU's microengine responds to the various channels based on a round robin scheduler. The scheduler bases its servicing decisions on the Channel Priority Register (CPR), the Service Request Latch (SRL), and the Service Grant Latch (SGL). These latches are affected by the Host Service Request Register (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 TPU channel such that the value for each channel is easily found. Most of these registers can be modified only by the TPU.

TPU Microsequencer Registers Window

Availability: TPU Simulation Target



The TPU Microsequencer Registers window displays miscellaneous information generally applying to the microsequencer. This window is something of a catch all.

The current Simulator time is displayed. The time is expressed in microseconds. The count of CPU clocks since the last Simulator reset is also displayed.

The TCR1 and TCR2 counter values are displayed. These 16-bit counters are displayed in hexadecimal. The TCR2 input pin's value is also displayed.

The active channel number is displayed. Since the active channel can be changed via the microcode when the CHAN register is modified, the active channel numbers corresponding to the current, previous, and second-most-previous microcycles are displayed. The last channel number is that of the channel that was initially being serviced after the time slot transition. This allows the user to calculate which channel a particular parameter is associated with for those parameters that take multiple microcycles to adjust after the CHAN register is changed.

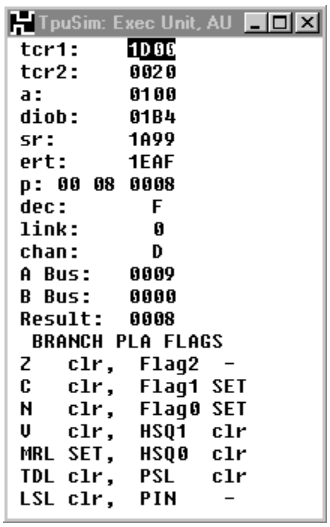
The current opcode and prefetch opcode are displayed in this window. Sequencing microinstructions that affect program flow may flush the prefetch opcode causing a NOP to be executed. Loading a 0xFFFFFFFF as the current opcode simulates this situation.

The DEC register is used for various purposes such as loop indexing and subroutine termination. The current DEC usage is displayed.

When MtDt is in TPU2 or TPU3 mode the two bank fields are displayed. The code bank is the bank in which code is being executed. The entry bank is the bank from which the active entry table is stored. In the real TPU the entry bank is specified via the TPUMCR2 register at bit positions 5 and 6. In MtDt this entry bank register is modified by the write_entry_bank() script command.

TPU Execution Unit Window

Availability: TPU Simulation Target



This window displays the information associated with the microengine registers and the branch PLA flags.

The A bus source, B bus source, and result (before any shifting) are also displayed.

The ERT register is that of the active channel.

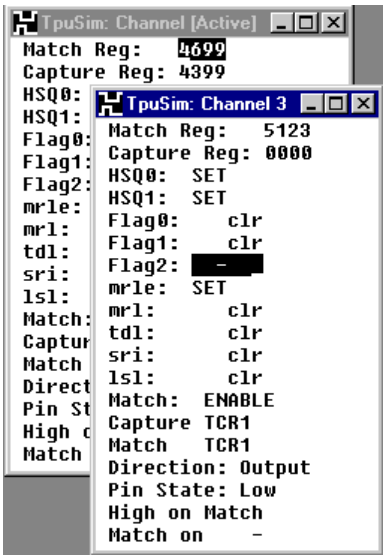
An important consideration is that the branch PLA flags are, for the most part, latched on a time slot transition. Therefore, they do not necessarily match the values displayed in the channel windows. The following list shows those branch PLA flags that may change during a channel service routine.

- ? The four execution unit flags, Z, C, N, and V, are latched whenever the CCL is specified on an AU subcommand.
- ? The Pin State Latch (PSL) gets latched two microinstructions after the CHAN_REG is written.
- ? FLAG2, FLAG1, and FLAG0 are always input directly to the branch PLA. (They are not latched at all.) Two microinstructions after the CHAN_REG is changed these flags reflect the new channel.

When MtDt is in TPU2 or TPU3 mode the FLAG2 flag and PIN branch conditional are displayed. FLAG2 is an additional TPU2 flag similar to the existing FLAG1 and FLAG0 flags. PIN branch conditional is a new branch condition available in the TPU2 that is similar to the PSL latch. But whereas the PSL reflects the pin state on the last time slot transition or channel register change, the PIN reflects the current pin state. Both the FLAG2 flag and the PIN branch conditional are features of the TPU2 that do not exist in the TPU1 and therefore are not displayed when in TPU1 mode.

TPU Channel Window

Availability: TPU Simulation Target



There are two versions of the TPU 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. The following information is displayed.

- ? The match and capture registers;
- ? The channel's bits from the Host Sequence Request (HSQR) register;
- ? Flag2 (TPU2 and TPU3 mode only), Flag1, and Flag0;
- ? The states of the Match Request Latch Enable (MRLE), Match Recognition Latch (MRL), Transition Detection Latch (TDL), Match/Transition Service Request Inhibit (SRI), and Link Service Latch (LSL);
- ? The disable match bit from the entry point. This bit controls whether a match is inhibited during channel servicing;
- ? The active counter for capture and match (TCR1 or TCR2);
- ? The pin direction (input or output);
- ? The current pin state (high or low);
- ? The PAC setting, which determines the pin action on a match if the pin is an output or the transition to be detected if the pin is an input;
- ? The effective match condition, greater-than-or-equal-to, or greater-than. Since this field applies to TPU2 and TPU3 only, when in TPU1 mode this field is not displayed;

Unlike previous versions of this software, the active channel is not specified directly from the menu when the window is opened. Instead, the active channel is specified from within the window by accessing the popup menu. To do this, right-click the mouse and select the desired channel number from the menu. This makes it easier to view the various channel settings without having to have lots of channel windows open.

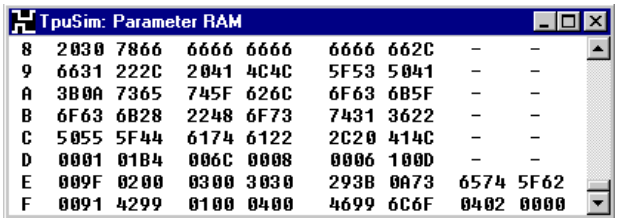
During channel servicing the active channel can be changed by the microcode. The microcode changes the active channel by writing to the CHAN register (found in the TPU Execution Unit window). When the active channel is changed by the microcode the effective active channel is somewhat murky. Some channel parameters migrate, after a number of instructions, to the new channel, while other parameters remain attached to the channel that originally caused the channel

servicing. How does MtDt handle this situation since the active channel version of the channel window is intended to show channel information of the active channel? This is answered in the following paragraph.

The active channel version of the channel window shows the channel information corresponding to the active parameters. For instance, the effective flag changes from the old channel to the new channel two microcycles after the CHAN register is changed. MtDt also switches the displayed flag two microcycles later. In this way, the user always views the affective channel.

TPU Parameter RAM Window

Availability: TPU Simulation Target



The parameter RAM is accessible by both the CPU and the TPU. It typically is used to pass information between the CPU and TPU and is used by the TPU for scratch and storage. For instance, the CPU might store the number of signal edges the TPU should measure in one parameter RAM location. The TPU would then keep a running count of the current number of edges measured in another parameter RAM location.

MtDt displays all existing RAM parameters in this window. The window is arranged in rows and columns. Each row displays the six or eight parameter words (each word is 16-bits wide) corresponding to a particular channel. Across each row are the six or eight parameters belonging to that channel.

In the TPU1 design, Freescale apparently ran out of room to provide the full eight parameters for all 16 channels. In fact, of the 16 channels, the lower 14 contain six parameters while only the final two channels contain the full complement of eight parameters. While MtDt is in TPU1 mode the un-implemented parameter RAM locations are not displayed.

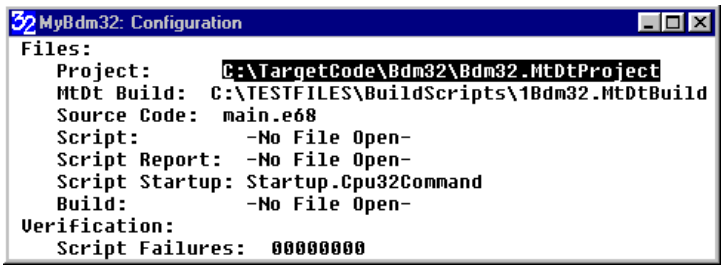
In TPU2 and TPU3 (and in MtDt while in TPU2 and TPU3 mode) all parameter locations are implemented. While in TPU2 and TPU3 mode MtDt displays all parameter locations.

How do the TPU and MtDt deal with accesses to the missing locations in TPU1 mode? It behooves the TPU microcoder to assume that new TPU versions will most likely have a full complement of parameter RAM. For both the TPU and MtDt, writes to these locations are ignored while the data returned fromreads is always zero.

During reset it has been observed that the parameter RAM of the actual TPU remains unchanged as long as power is not lost, though this behavior is apparently not documented. MtDt allows the user to specify whether the parameter RAM is cleared to all zeros, or remains unchanged after a reset. This is specified in the Reset Options dialog box.

683xx Hardware Debugger Configuration Window

Availability: 683xx Hardware Debugger Target



Files

The name of the project file is displayed. See the *Project Sessions* chapter for a detailed explanation of this capability.

The name of the MtDt build script file is displayed. See the *Full-System Simulation* chapter and *the MtDt Build Script Commands File* section for detailed explanations of the available capabilities and format of this file.

The names of the open source code (executable image), primary script, primary script report, and startup script files are displayed. These files are listed relative to the path of the open project file.

The auto-build file name is displayed. The auto-build file allows direct building of the executable image from within MtDt and is covered in the *Auto-Build Batch File Options Dialog Box* section.

Verification

The count of the number of user-defined verification tests that have failed since the last MtDt reset is displayed. User-defined tests are part of script commands files. See the *Script Commands Groupings* section for a description of the various script commands that support user-defined tests

SIM Main Window

Availability: 683xx Hardware Debugger

```
MyBdm32: SIM - Main
SIMCR   60CF  | EXOFF  F..F | CLKOUT Pin  On
              | FRZSW  E..E | On FREEZE the watchdog & periodic interrupt timer are Off
              | FRZBM  D..D | On FREEZE the bus monitor is Off
              | SLVEN  B..B | The IMB is Not Available to external bus master
              | SHEN   9..8 | Show Cycles Enable is 00
              | SUPV   7..7 | SIM S/U registers access at Supv priviledge level
              | MM     6..6 | Internal modules at FFF000-FFFFFF
              | IARB   3..0 | SIM interrupt arbitration priority F
SYNCR   5F08  | W      F..F | W-freq ctrl: VCO multiplied by 1
              | X      E..E | X-freq ctrl: VCO divided by 2
              | Y      D..8 | Y-freq ctrl: VCO divided by [1F+1]
              |             | Clock frequency is 8.388608 MHz (assuming 32.768 KHz crystal)
              | EDIV   7..7 | ECLK pin (addr 23) divide rate is system clock  divided by 8
              | SLIMP  4..4 | Crystal reference is ok
              | SLOCK  3..3 | VCO is locked, or external
              | RSTEN  2..2 | Loss of reference causes limp mode
              | STSIM  1..1 | During low-power stop, SIM clock is driven by an external source
              | STEXT  0..0 | During low-power stop, the CLKOUT pin is held low
RSR       00  | EXT    7..7 | Reset caused by an external signal          No
              | POW    6..6 | Reset caused by the power-up reset circuit  No
              | SW     5..5 | Reset caused by a software watchdog timeout No
              | HLT    4..4 | Reset caused by the halt monitor            No
              | LOC    2..2 | Reset caused by a loss of clock             No
              | SYS    1..1 | Reset caused by a RESET instruction         No
              | TST    0..0 | Reset caused by the test submodule          No
SYPCR     80  | SWE    7..7 | The software watchdog is  ENABLED
              | SWP    6..6 | First watchdog prescaler is divide by 1
              | SWT    5..4 | Additional watchdog prescaler is 2YX9
              |             | Watchdog period is 015,625.0 micro-seconds (assuming 32.768 KHz c
              | HME    3..3 | Halt monitor function Disabled
              | BME    2..2 | Bus monitor is Disabled for external bus cycles
              | BMT    1..0 | Bus monitor timeout is 64 system clocks
PICR    000F  | PIRQL  A..8 | Interrupt priority disabled
              | PIV    7..0 | Periodic interrupt vector 0F
PITR    0000  | PTP    8..8 | First periodic interrupt timer prescaler is divide by 1
              | PTM    7..0 | Additional periodic interrupt timer prescaler is [00+1]
              |             | Periodic interrupt timer period is 000,122.1 micro-seconds (assum
```

The SIM Main window displays the Module Configuration Register (SIMCR), the Clock Synthesizer Control Register (SYNCR), the Reset Status Register (RSR), the System Protection Control Register (SYPCR), the Periodic Interrupt Control Register (PICR), and the Periodic Interrupt Timer Register (PITR).

SIM Ports Window

Availability: 683xx Hardware Debugger

```
MyBdm32: SIM - Ports
***** PORT C DATA AND CHIP SELECT PIN ASSIGNMENT REGISTER 1 *****
             | PRIMARY ALTERNATE DISCRETE BITS |
PORTC     7F |  CS10    ADDR23     ECLK    9..8 | 16-Bit CS   -
CSPAR1 0355  |  CS9     ADDR22     PC.6    7..6 | ADDR22      -
             |  CS8     ADDR21     PC.5    5..4 | ADDR21      -
             |  CS7     ADDR20     PC.4    3..2 | ADDR20      -
             |  CS6     ADDR19     PC.3    1..0 | ADDR19      -
***** PORT C DATA AND CHIP SELECT PIN ASSIGNMENT REGISTER 0 *****
PORTC     7F |  CS5     FC2        PC.2    D..C | 16-Bit CS   -
CSPAR0 3FFF  |  CS4     FC1        PC.1    B..A | 16-Bit CS   -
             |  CS3     FC0        PC.0    9..8 | 16-Bit CS   -
             |  CS2     BGACK       -      7..6 | 16-Bit CS   -
             |  CS1     BG          -      5..4 | 16-Bit CS   -
             |  CS0     BR          -      3..2 | 16-Bit CS   -
             |  CSBOOT  -           -      1..0 | 16-Bit CS   -
***** PORT E PIN ASSIGNMENT, DATA DIRECTION, AND DATA REGISTERS *****
PEPAR     FF |  SIZ1    7..7 |  SIZ1      -    -
DDRE      00 |  SIZ0    6..6 |  SIZ0      -    -
PORTE     FF |  AS      5..5 |  AS        -    -
             |  DS      4..4 |  DS        -    -
             |  RMC     3..3 |  RMC       -    -
             |  AVEC    2..2 |  AVEC      -    -
             |  DSACK1  1..1 |  DSACK1    -    -
             |  DSACK0  0..0 |  DSACK0    -    -
***** PORT F PIN ASSIGNMENT, DATA DIRECTION, AND DATA REGISTERS *****
PFPAR     00 |  IRQ7    7..7 |  PF.7     Input 0
DDRF      00 |  IRQ6    6..6 |  PF.6     Input 0
PORTF     01 |  IRQ5    5..5 |  PF.5     Input 0
             |  IRQ4    4..4 |  PF.4     Input 0
             |  IRQ3    3..3 |  PF.3     Input 0
             |  IRQ2    2..2 |  PF.2     Input 0
             |  IRQ1    1..1 |  PF.1     Input 0
             |  MODCLK  0..0 |  PF.0     Input 1
```

The SIM Ports window contains the registers that determine the port pin usage, pin direction (input or output) and value. These registers are the Port C, Port E, and Port F Data Registers (PORTC, PORTE, PORTF), the Chip Select Pin Assignment Registers one and zero (CSPAR1, CSPAR0), Port E and Port F Pin Assignment Registers (PEPAR, PFPAR) and the Port E and Port F Data Direction Registers (DDRE, DDRF).

To the right of the bit position field are a variable number of fields. Depending on the specific capability of each port the usage, direction and value can be selected.

SIM Chip Selects Window

Availability: 683xx Hardware Debugger

```
MyBdm32: SIM - Chip Selects
CS#  CSBAR CSOR  ADDR   SIZE  MODE   BYTE    R/W  STRB WAITS SPACE IPL AVEC PIN/WIDTH
Boot 0007  7B70 000000  1M  Asynch  Both    Both  AS   0xD   S/U   All  Off 16-Bit CS
CS0  0000  0000 000000  2K  Asynch Disable Rsvd  AS    0    Cpu   All  Off 16-Bit CS
CS1  0000  7B70 000000  2K  Asynch  Both    Both  AS   0xD   S/U   All  Off 16-Bit CS
CS2  0000  7B70 000000  2K  Asynch  Both    Both  AS   0xD   S/U   All  Off 16-Bit CS
CS3  0000  0000 000000  2K  Asynch Disable Rsvd  AS    0    Cpu   All  Off 16-Bit CS
CS4  0000  0000 000000  2K  Asynch Disable Rsvd  AS    0    Cpu   All  Off 16-Bit CS
CS5  0000  0000 000000  2K  Asynch Disable Rsvd  AS    0    Cpu   All  Off 16-Bit CS
CS6  0000  0000 000000  2K  Asynch Disable Rsvd  AS    0    Cpu   All  Off ADDR19
CS7  0000  0000 000000  2K  Asynch Disable Rsvd  AS    0    Cpu   All  Off ADDR20
CS8  0000  0000 000000  2K  Asynch Disable Rsvd  AS    0    Cpu   All  Off ADDR21
CS9  0000  0000 000000  2K  Asynch Disable Rsvd  AS    0    Cpu   All  Off ADDR22
CS10 0000  0000 000000  2K  Asynch Disable Rsvd  AS    0    Cpu   All  Off 16-Bit CS
```

The SIM Chip Selects window displays information for each of the SIMS chip select. The second and third fields from the left are the Chip Select Base Address (CSBARX) and the Chip Select Options register (CSORX), where X denotes the selects one of the 11 chip selects.

Starting from the left and going to the right are the Base Address, Block Size, Asynchronous/Synchronous E-Clock Mode, Upper/Lower Byte Option, Read/Write, Address/Data Strobe, Wait States, Address Space Select, Interrupt Priority Level, Autovector Enable, and Pin Widths fields. All except the Pin/Width field are bit groupings derived from the CSBARX and the CSORX registers. The Pin/Width field is derived from the Chip Select Pin Assignment registers one and zero (CSPAR1, CSPAR0) (not shown in this window).

QSM Main Window

Availability: 683xx Hardware Debugger

```
MyBdm32: QSM - Main
QSMMCR   0086 | STOP   F..F | The QSM Module is operating normally
              | FRZ1   E..E | When FREEZE on IMB, ignore it
              | FRZ0   D..D | [Reserved for future use]
              | SUPV   7..7 | QSM S/U registers access at Supv priviledge level
              | IARB   3..0 | QSM interrupt arbitration priority 0
QILR/VR 000F  | ILQSPI D..B | QSPI interrupt priority is 0
              | ILSCI  A..8 | SCI  interrupt priority is 0
              | INTV   7..0 | QSM interrupt vecor number is 0F
```

The QSM Main window displays the QSM Configuration Register (QSMMCR) and the QSM Interrupt Level and Vector Register (QILR/QIVR).

QSM Port Window

Availability: 683xx Hardware Debugger

```
MyBdm32: QSM - Port
PQSPAR    00 | TXD     7..7 |  PQS.7    Input 0
DDRQS     00 | PCS3    6..6 |  PQS.6    Input 0
PORTQS    00 | PCS2    5..5 |  PQS.5    Input 0
             | PCS1    4..4 |  PQS.4    Input 0
             | PCS0/SS 3..3 |  PQS.3    Input 0
             | SCK     2..2 |  PQS.2    Input 0
             | MOSI    1..1 |  PQS.1    Input 0
             | MISO    0..0 |  PQS.0    Input 0
```

The QSM Port window displays the Port QS Pin Assignment Register (PQSPAR), the Port QS Data Direction Register (DDRQS), and the Port QS Data Register (PORTQS). Using the bit port bit fields, located to the right of the bit position fields, the use of each port pin can be specified. Each pin can be specified as a generic I/O or assigned to the QSM. In addition, both the data direction and register value can be specified.

QSM QSPI Window

Availability: 683xx Hardware Debugger

```
MyBdm32: QSM - QSPI
SPCR0  0104 | MSTR   F..F | QSPI is a slave device
            | WOMQ   E..E | Outputs have normal MOS drivers
            | BITS   D..A | Bits field is 0 resulting in 16   bits per transfer
            | CPOL   9..9 | The inactive state value of the SCK pin is lo
            | CPHA   8..8 | Data CHANGED  on SCK's leading edge and captured on the trailing edge
            | SPBR   7..0 | SPBR is 04
            |             | for a clock period of 000,001.0 microseconds
SPCR1  0404 | SPE    F..F | QSPI is a slave device
            | DSCKL  E..8 | Delay from chip-select assertion to leading edge of SCK
            |             | is equal to DSCKL field, 04, system clocks.
            |             | for a delay of 000,000.5 micro-seconds
            |             | for each channel with CMD.RAM's DSCK bit asserted
            | DTL    7..0 | Delay from chip-select negation to next operation
            |             | is equal to DTL field, 04, multipied by 32 system clocks
            |             | for a delay of 000,015.3 micro-seconds
            |             | for each channel with CMD.RAM's DT bit asserted
SPCR2  0000 | SPIFIE F..F | QSPI interrupts are disabled
            | WREN   E..E | Wraparound mode is disabled
            | WRTO   D..D | Wrap to  pointer address 0
            | ENDQP  B..7 | The last QSPI channel to transfer is 0
            | NEQQP  3..0 | The first QSPI channel to transfer is 0
SPCR3  0000 | LOOPQ  A..A | QSPI interrupts are disabled
            | HMIE   9..9 | HALTA and MODF interrupts are disabled
            | HALT   8..8 | Halt is not enabled
            | SPIF   7..7 | QSPI is not finished
            | MODF   6..6 | Another SPI node requested to become master? no
            | HALTA  5..5 | QSPI is not halted
            | CPTQP  3..0 | The last channel executed is 0
QSPI RAM  ==================================================================================
             0    1    2    3     4    5    6    7      8    9    A    B     C    D    E    F
Receive   0000 81E7 7FFB 7C9F  EFB3 66DB B8F2 E6FF   6747 97F3 EAAF 25AA  65FD F8F1 9F7A F8B7
Transmit  BBF8 7BFB 1A8E 94FF  7872 96BA FEDB 0DBF   15EB E4C7 BF6F F987  575F F77E F77B 41FD
Command     7F   EF   42   6D    ED   DC   FD   FF     1B   FD   DA   8F    CB   FF   D2   7A
CONT.7    PORT DEAS PORT PORT  DEAS DEAS DEAS DEAS   PORT DEAS DEAS DEAS  DEAS DEAS DEAS PORT
BITSE.6   BITS BITS BITS BITS  BITS BITS BITS BITS      8 BITS BITS    8  BITS BITS BITS BITS
DT.5       DTL  DTL  std  DTL   DTL  std  DTL  DTL    std  DTL  std  std   std  DTL  std  DTL
DSCK.4,   DSCK CK/2 CK/2 CK/2  CK/2 DSCK DSCK DSCK   DSCK DSCK DSCK CK/2  CK/2 DSCK DSCK DSCK
PCS.3,      HI   HI   lo   HI    HI   HI   HI   HI     HI   HI   HI   HI    HI   HI   lo   HI
PCS.2,      HI   HI   lo   HI    HI   HI   HI   HI     lo   HI   lo   HI    lo   HI   lo   lo
PCS.1,      HI   HI   HI   lo    lo   lo   lo   HI     HI   lo   HI   HI    HI   HI   HI   HI
PCS.0,      HI   HI   lo   HI    HI   lo   HI   HI     HI   HI   lo   HI    HI   HI   lo   lo
```

The QSM QSPI window displays the QSPI Control registers 0, 1, 2 and 3 (SPCR0, SPCR1, SPCR2, and SPCR3). In addition the QSPI's Receive, Transmit and Command RAM (RR[0:F], TR[0:F], and CR[0:F]) are displayed for each of the 16 locations. Below the Command RAM field, each of the 16 command RAM locations is broken down by bit groupings. These fields are the Continue (CONT), the Bits per Transfer Enable (BITSE), the Delay After Transfer (DT) and the PCS to SCK Delay (DSCK).

QSM SCI (UART) Window

Availability: 68332 Hardware Debugger

```
MyBdm32: QSM - SCI
SCCR0   0004 | SCBR   C..0 | SCI clock period 000,015.3 micro-seconds
             |             | (assuming 32.768 KHz crystal)
SCCR1   0000 | LOOP   E..E | Normal SCI operation, no looping, feedback path disabled
             | WOMS   D..D | If configured as an output, TXD is a normal CMOS output
             | ILT    C..C | short idle line detect
             | PT     B..B | even parity
             | PE     A..A | SCI parity is disabled
             | M      9..9 | 10-bit SCI frame
             | WAKE   8..8 | SCI receiver awakened by idle-line detection
             | TIE    7..7 | SCI transmit TDRE interrupts are disabled
             | TCIE   6..6 | SCI transmit complete interrupts are disabled
             | RIE    5..5 | SCI receive "recieve data register flag" (RDRF)
             |             | and "overrun error" (OR) interrupts are disabled
             | ILIE   4..4 | Idle line interrupts are disabled
             | TE     3..3 | SCI receiver is disabled (TXD pin can be used as I/O)
             | RE     2..2 | SCI receiver is disabled (status bits inhibited)
             | RWU    1..1 | Normal receiver operation (received data recognized)
             | SBK    0..0 | Normal operation
SCSR    0180 | TDRE   8..8 | A new character can now be written to register TDR
             | TC     7..7 | SCI transmitter is idle
             | RDRF   6..6 | Receive data register (RDR) is empty or contains previously read data
             | RAF    5..5 | SCI receiver is idle
             | IDLE   4..4 | SCI receiver did not detect an idle line condition
             | OR     3..3 | Receive data register (RDRF) is     cleared before new data arrives
             | NF     2..2 | Noise was not detected on the received data
             | FE     1..1 | A framing error or break was not detected on the received data
             | PE     0..0 | A parity error did not occur on the received data
SCDR    0050 | RDR    8..0 | Receive data register 0050 (read only)
             | TDR    8..0 | Transmit data register 0055 (write only - read value is meaningless)
```

The QSM SCI window displays the SCI Control registers 0 and 1 (SCCR0 and SCCR1), the SCI Status Register (SCSR), and the SCI Data Register (SCDR).

Masked ROM Window

Availability: 683xx Hardware Debugger

```
MySim32: Masked ROM
MRMCR   98C6 | STOP   F..F | ROM array is is in low-power stop mode
             | BOOT   C..C | ROM does not responds to bootstrap word locations during reset vector fetch
             | LOCK   B..B | ROM Write lock enabled . Protected registers and fields cannot be written
             | ASPC   9..8 | ROM array is accessible at the unrestricted program and data priviledge level
             | WAIT   7..6 | ROM array accesses take 2 wait states
ROMBAH 0456  | AD-HI  7..0 |
ROMBAL 4000  | AD-LO  F..C | Base address is 04564000
RSIGHI 1234  | RS-HI  7..0 | ROM signature high register is 1234
RSIGLO 5678  | RS-LO  F..0 | ROM signature low  register is 5678
ROMBS0 1001  | BS-0   F..0 | ROM bootstrap word 0 is 1001
ROMBS1 2222  | BS-1   F..0 | ROM bootstrap word 1 is 2222
ROMBS2 3333  | BS-2   F..0 | ROM bootstrap word 2 is 3333
ROMBS3 12AC  | BS-3   F..0 | ROM bootstrap word 3 is 12AC
```

The Masked ROM window displays the Masked ROM Module Configuration Register (MRMCR), the ROM Array Base Address High Register (ROMBAH), the ROM Base Address Low Register (ROMBAL), the ROM Signature High Register (RSIGHI), the ROM Signature Low Register (RSIGLO), and the ROM Bootstrap Words 0 through 3 Registers (ROMBS0, ROMBS1, ROMBS2, and ROMBS3).

Standby RAM Submodule Window (68336/68376)

Availability: 683xx Hardware Debugger

```
My68376: Standby RAM
RAMMCR  0800 | STOP   F..F | Standby RAM (SRAM) is operating normally
             | RLCK   B..B | SRAM base address registers are locked
             | RASP   9..8 | SRAM is accessible at the unrestricted program and data priviledge level
RAMBAH  0050 | AD-HI  7..0 |
RAMBAL  2000 | AD-LO  F..C | Base address is 00502000
```

The Standby RAM Submodule window (68336/68376) displays the RAM Module Configuration Register (RAMMCR), the RAM Array Base Address High Register (RAMBAH), and the RAM

Array Base Address Low Register (RAMBAL).

Static RAM Submodule Window (68338)

Availability: 683xx Hardware Debugger

```
MySim32: Static RAM Submodule
FFF500:  0000 0000 0000 0000   0000 0000 0000 0000
FFF510:  1111 0000 0000 0000   0000 0000 0000 0000
FFF520:  0000 0000 0000 0000   0000 0000 0000 0000
FFF530:  0000 0000 0000 0000   0000 0000 0000 0000
```

The Static RAM Submodule Window (68338) consists of 16 contiguous 16-bit words of RAM.

TPU Emulation RAM Window (68332, 68336, 68376)

Availability: 683xx Hardware Debugger

```
MyBdm32: Standby RAM
TRAMMCR 0100  | STOP   F..F | TPURAM Module is operating normally
              | RASP   8..8 | TPURAM array is accessible at the Supv priviledge level
TRAMBAR 0001  | ADDR   F..3 | TPURAM array base address is 00000000
              | RAMDS  0..0 | TPURAM array is Disabled
```

The TPU Emulation RAM window displays the TPURAM Module Configuration Regis ter (TRAMMCR) and the TPURAM Base Address Register (TRAMBAR).

TPU Main Window

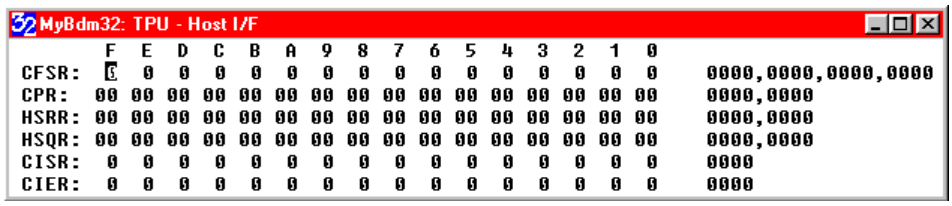
Availability: 683xx Hardware Debugger

```
MyBdm32: TPU - Main
TPUMCR  0086  | STOP   F..F | TPU Module is operating normally
              | TCR1P  E..D | TCR1 prescaler divides by 1
              | TCR2P  C..B | TCR2 prescaler divides by 1
              | EMU    A..A | TPU executes out of standard mask ROM
              | T2CG   9..9 | TCR2 pin used as clock source for TCR2
              | STF    8..8 | TPU is operating normally
              | SUPV   7..7 | TPU S/U registers access at Supv priviledge level
              | PSCK   6..6 | Clock input to the TCR1 input is the system clock/32
              | IARB   3..0 | TPU interrupt arbitration priority 0
              |             | TCR1 counter clock period is 000,003.8 micro-seconds
              |             | TCR2 counter clock period is     -     micro-seconds
              |             | The TCR2 counter clock source is the TCR2 input pin divided by 1
TICR    0000  | CIRL   A..8 | Channel interrupt request level is 0
              | CIBV   7..4 | Channel interrupt base vector is 0
              |             | (interrupt vector is 0x00 plus channel number)
```

The TPU Main window displays the TPU Module Configuration Register (TPUMCR) and the TPU Interrupt Configuration Register (TICR).

TPU Host Interface Window

Availability: 683xx Hardware Debugger

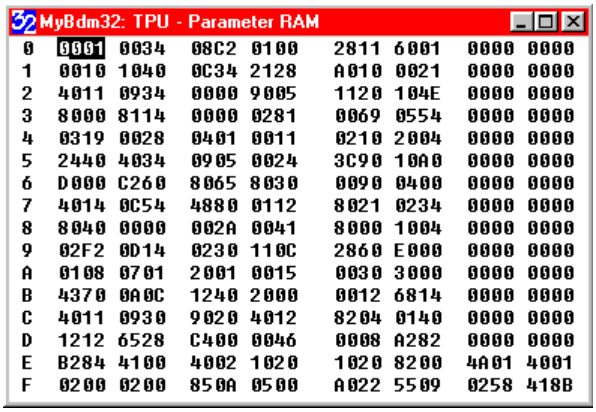


```
MyBdm32: TPU - Host I/F
        F  E  D  C  B  A  9  8  7  6  5  4  3  2  1  0
CFSR:   C  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0     0000,0000,0000,0000
CPR:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     0000,0000
HSRR:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     0000,0000
HSQR:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     0000,0000
CISR:   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0     0000
CIER:   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0     0000
```

The TPU Host Interface window displays the Channel Function Select Register (CFSR), the Channel Priority Register (CPR), the Host Service Request Register (HSRR), the Host Sequence Register (HSQR), the Channel Interrupt Status Register (CISR), and the Channel Interrupt Enable Register (CIER).

TPU Parameter RAM Window

Availability: 683xx Hardware Debugger



```
MyBdm32: TPU - Parameter RAM
0   0001 0034  08C2 0100   2811 6001   0000 0000
1   0010 1040  0C34 2128   A010 0021   0000 0000
2   4011 0934  0000 9005   1120 104E   0000 0000
3   8000 8114  0000 0281   0069 0554   0000 0000
4   0319 0028  0401 0011   0210 2004   0000 0000
5   2440 4034  0905 0024   3C90 10A0   0000 0000
6   D000 C260  8065 8030   0090 0400   0000 0000
7   4014 0C54  4880 0112   8021 0234   0000 0000
8   8040 0000  002A 0041   8000 1004   0000 0000
9   02F2 0D14  0230 110C   2860 E000   0000 0000
A   0108 0701  2001 0015   0030 3000   0000 0000
B   4370 0A0C  1240 2000   0012 6814   0000 0000
C   4011 0930  9020 4012   8204 0140   0000 0000
D   1212 6528  C400 0046   0008 A282   0000 0000
E   B284 4100  4002 1020   1020 8200   4A01 4001
F   0200 0200  850A 0500   A022 5509   0258 418B
```

The TPU Parameter RAM window displays the TPU Parameter RAM. Each row in the window displays the parameter RAM belonging to the row number's TPU channel.

GPT Main Window

Availability: 683xx Hardware Debugger

```
MySim32: GPT - Main
GPTMCR        3881 | STOP   F..F | GPT clocks are not shut down
                   | FRZ1   E..E | FRZ1 is not used
                   | FRZ0   D..D | When FREEZE on IMB, halt the GPT
                   | STOPP  C..C | Stop prescaler and pulse accumulator, ignore input pins
                   | INCP   B..B | On STOPP, increment prescaler & input synchronizers once
                   | SUPV   7..7 | S/U registers access at Supv priviledge level
                   | IARB   3..0 | Interrupt arbitration priority 1
GPTICR        2300 | IPA    F..C | Make Input Capture 2                 highest priority
                   | IPL    A..8 | Interrupt priority is level 3
                   | IVBA   7..4 | Interrupt vector base address is 0X
DDGRP/PORTGP 2FF2  | DDGRP  F..8 | Pin Direction is 2F  (see other windows)
                   | PORTGP 7..0 | Pin Data is      F2  (see other windows)
OC1M/OC1D     F8E0 | OC1M   F..B | OC1M Action Mask is 1F (see Output Compare window)
                   | IPA    7..3 | OC1D Action Data is 1C (see Output Compare window)
TCNT          1344 |        F..0 | Timer counter register is 1344
PACTL/PACNT   0400 | I4/O5  A..A | Output compare 5 Disable,  Input capture 4  ENABLE
                   |             | (See Pulse Accumulator window for most settings)
TMSK1/TMSK2   82B8 |             | (See other windows for most settings)
                   | TOI    7..7 | Interrupt requested when TOF flag is set
                   | CPROUT 3..3 | TCNT clock driven out OC1 pin
                   | CPR    2..0 | CPR=0; TCNT Capture/Compare clock period is 000,250.1 ns or Fsys/4
                   |             |                                  -
TFLG1/TFLG2   1030 | TOF    7..7 | Timer Overflow flag is clr
CFORC/PWMC    2B84 |             | (See Output Compare and PWM windows for most settings)
                   | PPR    6..4 | PPR=0; PWM CNT input is  000,125.1 ns or Fsys/2
                   |             |                        -
PRESCL        0033 |        F..0 | 9-Bit prescaler is 0033 (Bits[F..9] always zero)
```

The GPT Main window displays the GPT Module Configuration Register (GPTMCR), the GPT Interrupt Configuration Register (GPTICR), the Port G Data Direction Register (DDGRP), the Port G Data Register (PORTGP), the OC1 Action Mask Register (OC1M), the OC1 Action Data Register (OC1D), the Timer Counter Register (TCNT), the Pulse Accumulator Control Register (PACTL), the Pulse Accumulate Counter (PACNT), the Timer Interrupt Mask Registers 1 and 2 (TMSK1/TMSK2), the Timer Interrupt Flag Registers 1 and 2 (TFLG1/TFLG2), the Compare Force Register(CFORC), the PWM Control Register C (PWMC), and the GPT Prescaler (PRESCL.)

GPT Input Captures Window

Availability: 683xx Hardware Debugger

```
MySim32: GPT IC - Input Captures
      Capture  Reg                Input/   Pin
 #     Edge    Val    Interrupt   Output?  Is
IC4    both    0000    ENABLED     Input   HI
IC3   rising   0000   Disabled    Output   lo
IC2  falling   0000    ENABLED    Output   HI
IC1  disabled  0000   Disabled    Output   lo
-----------------------------------------------
TCTL1/TCTL2=1ED8,  TMSK1/TMSK2=82B8
DDGRP=2F,          PORTGP=F2
```

The GPT Input Captures window displays the Timer Control Registers 1 and 2 (TCTL1/TCTL2), the Timer Interrupt Mask Registers 1 and 2 (TMSK1/TMSK2), the Port G Data Direction Register (DDGRP), and the Port G Data Register (PORTGP).

GPT Output Compares Window

Availability: 683xx Hardware Debugger

```
MySim32: GPT OC - Output Compares
        On       On OC1   On OC1    Reg                TCNT      Force      Input/    Pin
 #     Match     Match     pin      Val    Interrupt   Match?   Compare**   Output?   Is
OC5     -         -        -        -       -          -         -          -        -
OC4   toggle      op       SET      1233   Disabled    No       clr         Input    HI
OC3    set        op       SET      3300   Disabled    No       SET        Output    HI
OC2   clear       op       clr      0AB0   Disabled    Yes      clr         Input    HI
OC1     -         op       clr      CDEE   Disabled    No       SET        Output    lo
-------------------------------------------------------------------------------------
TCTL1/TCTL2=1ED8,  OC1M/OC1D=F8E0,  TMSK1/TMSK2=82B8,  TFLAG1/TFLAG2=1030
CFPRC/PWMC=2B84,   DDGRP=2F,        PORTGP=F2
** Note: Force Compare always reads zero
```

The GPT Output Captures window displays the Timer Control Registers 1 and 2 (TCTL1/TCTL2), the OC1 Action Mask Register (OC1M), the OC1 Action Data Register (OC1D), the Timer Interrupt Mask Registers 1 and 2 (TMSK1/TMSK2), the Timer Interrupt Flag Registers 1 and 2 (TFLG1/TFLG2), the Compare Force Register (CFORC), the PWM Force Register (PWMC), the Port G Data Direction Register (DDGRP), and the Port G Data Register (PORTGP).

GPT Pulse Accumulate Window

Availability: 683xx Hardware Debugger

```
MySim32: GPT PA - Pulse Accumulate
PACTL/PACNT   0400 | PAIS   F..F | Pulse accumulator input pin state is lo
                   | PAEN   E..E | Pulse accumulator is Disabled
                   | PAMOD  D..D | External event counting
                   | PEDGE  C..C | PAI falling edge increments counter
                   | PCLKS  B..B | PCKLK pin state is lo
                   | I4/O5  A..A | Output compare 5 Disable,  Input capture 4  ENABLE
                   | PACLK  9..8 | Clock source: System clock divided by 512
                   |             | Clock period is 000,032.0 ns
                   | PACNT  7..0 | Pulse accumulator count is 00
TMSK1/TMSK2   82B8 | PAOVI  5..5 | Counter overflow Interrupt  ENABLED
                   | PAII   4..4 | Input pulse Interrupt  ENABLED
TFLG1/TFLG2   1030 | PAOVF  5..5 | Counter overflow flag SET
                   | PAIF   4..4 | Input pulse flag is SET
```

The GPT Pulse Accumulate window displays the Pulse Accumulate Control Register (PACTL), the Pulse Accumulate Counter Register (PACNT), the Timer Interrupt Mask Registers 1 and 2 (TMSK1/TMSK2), and the Timer Interrupt Flag Registers 1 and 2 (TFLG1/TFLG2).

GPT Pulse Width Modulation Window

Availability: 683xx Hardware Debugger

```
MySim32: GPT PWM - Pulse Width Modulation
         Force    Force    Pin    Fast/  Period  Period    Width  ~Width
        Output?   State    Use    Slow?  (CNTs)   (us)      Val   Percent
PWM-A     Yes      lo     TCNT    FAST    256    000,032.0  00      0
PWM-B     Yes      lo       -     slow   32768   004,098.4  00      0
-------------------------------------------------------------
PPR=0; PWM CNT input is 000,125.1 ns or Fsys/2
                        -
-------------------------------------------------------------
CFPRC/PWMC=2B84,
```

The GPT Pulse Accumulate Width Modulation window displays the Compare Force Register (CFORC), and the PWM Control Register C Register (PWMC).

CTM4/CTM6 Bus Interface and Clocks Window

Availability: 683xx Hardware Debugger

```
My68376: CTM4 - Bus I/F and Clocks
BIUMCR   1A01 | STOP   F..F | Configirable Timer Module 4 clocks are  ENABLED
              | FRZ1   E..E | When FREEZE on IMB, ignore it
              | VECT   C..B | Interrupt vector base is $C0 plus submodule offset
              | IARB   A..8 | CTM4 interrupt arbitration priority 2
              | TBRS1  5..5 | Time Base Select Bit 1 is 0
              | TBRS0  0..0 | Time Base Select Bit 0 is 1
              |             | Use Time Base Bus 2 (TBB2)
BIUTBR   0000 |        F..0 | The value of Time Base Bus 2 is 0000
CPCR     0002 | PRUN   3..3 | Prescaler divider is held in reset and not running
              | DIV23  2..2 | First prescaler stage divides by two
              | PSEL   1..0 | PCLK6 post DIV23 prescaler is 128
                      PCLK1      PCLK2      PCLK3      PCLK4      PCLK5      PCLK6
Clock Period (ns)  015,258.8  001,907.3  000,953.7  000,476.8  000,238.4  000,119.2
Total Prescaler        256         32         16          8          4          2
```

The CTM4/CTM6 Bus Interface and Clocks window displays the BIU Module Configuration Register (BIUMCR), the BIU Time Base Register (BIUTBR), and the CPSM Control Register (CPCR). In addition, the prescaler clock period is calculated and listed in both nano-seconds and division ratio.

CTM4/CTM6 Free-Running Counter Submodule Window

Availability: 683xx Hardware Debugger

```
My68376: CTM4 - Free-Running Counters (FCSM)
FCSMSIC 1800 | COF    F..F | A free-running counter overflow has not occured
             | IL     E..C | Interrupt is level 1
             | IARB3  B..B | Interrupt arbitration bit 3 is SET
             |             | (Concatenated with BIUMCR:IARB[2..0])
             | DRV    9..8 | Time base bus is not driven
             | IN     7..7 | The CTM2C clock input pin is Low
             | CLK    2..0 | The counter clock source is PCLK1
FCSMCNT 23D2 |        F..0 | The value of the FCSM counter register is 23D2
```

The CTM4/CTM6 Free-Running Counter Submodule window displays the FCSM Status/Interrupt/Control Register (FCSMSIC) and the FCSM Counter Register (FCSMCNT).

CTM4/CTM6 Modulus Counter Submodule Window

Availability: 683xx Hardware Debugger

```
My68376: CTM4 - Modulus Counters (MCSM)
***********    MCSM02     ****************************************
SIC   2921 | COF    F..F | A overflow has not occured
           | IL     E..C | Interrupt level is level 2
           | IARB3  B..B | Interrupt arbitration bit 3 is SET
           |             | (Concatenated with BIUMCR:IARB[2..0])
           | DRV    9..8 | Time base bus A is driven
           | IN2    7..7 | The CTM2C clock input pin is Low
           | IN1    6..6 | The CTD9 modulus load input pin is Low
           | EDGE   5..4 | CTD9 edge detection: negative edge only
           | CLK    2..0 | The counter clock source is PCLK2
CNT   1232 |        F..0 | Counter register value is 1232
ML    1232 |        F..0 | Modulus latch value is 1232
***********    MCSM11     ****************************************
SIC   0000 | COF    F..F | A overflow has not occured
```

The CTM4/CTM6 Modulus Counters Submodule window displays the MCSM 2 and 11 Status/Interrupt/Control Registers (MCSM2SIC, MCSM11SIC), the MCSM 2 and 11 Counter Registers (MCSM2CNT, MCSM11CNT), and the MCSM 2 and 11 Modulus Latch Registers (MCSM2ML, MCSM11ML).

CTM4 Double-Action Submodule Window

Availability: 683xx Hardware Debugger

```
My68376: CTM4 - Double Actions (DASM)
-----------------  DASM #3   ---------------------------------------------------
SIC     2B88 | FLAG   F..F | An input capture or output compare has not occured
             | IL     E..C | Interrupt is level 2
             | IARB3  B..B | Int. arbitration bit 3 is SET + BIUMCR:IARB[2..0]
             | WOR    9..9 | If output then buffer operates in wired-or mode
             | BSL    8..8 | Connected to time base bus B
             | IN     7..7 | If input, then the pin is High
             | FORCA  6..6 | Set to force channel A compare, always reads 0
             | FORCB  5..5 | Set to force channel B compare, always reads 0
             | EDPOL  4..4 | 0 is A compare on channel A sets   the ouptut pin to logic 1
             |             |      A compare on channel B clears the ouptut pin to logic 0
             | MODE   3..0 | OPWM - Output pulse width modulation   16 bit resolution
DATA-A 0BB5  |        F..0 | The DATA-A register value is 0BB5
DATA-B E97B  |        F..0 | The DATA-B register value is E97B
-----------------  DASM #4   ---------------------------------------------------
SIC     0000 | FLAG   F..F | An input capture or output compare has not occured
```

The CTM4 Double-Action Submodule window displays the DASM 3, 4, 9, and 10 Status/Interrupt/Control Registers (DASM3SIC, DASM4SIC, DASM9SIC, DASM10SIC), and the DASM 3, 4, 9, and 10 Data Register A and B registers (DASM3A, DASM4A, DASM9A, DASM10A, DASM3B, DASM4B, DASM9B, and DASM10B).

CTM4 Pulse Width Modulation Submodule Window

Availability: 683xx Hardware Debugger

```
My68376: CTM4 - Pulse Width Modulators (PWM)
-------------  PWM #5  ------------------------------------------------
SIC   2800 | FLAG   F..F | A PWM period is not complete
           | IL     E..C | PWM interrupt is level 2
           | IARB3  B..B | PWM interrupt arbitration bit 3 is SET
           |             | (Concatenated with BIUMCR:IARB[2..0])
           | PIN    7..7 | The PWM output pin is Low
           | LOAD   5..5 | SET to load pulse width (always reads 0)
           | POL/EN 4..3 | Always low
           | POL/EN 4..3 | -
           | CLK    2..0 | CLK is 0 for a period of 000,119.2 [Fsys/2]
PER   F463 |        F..0 | The PERIOD is      F463
PW    9D1E |        F..0 | The PULSE WIDTH is 9D1E
CNT   0001 |        F..0 | The COUNTER is     0001
-------------  PWM #6  ------------------------------------------------
SIC   0000 | FLAG   F..F | A PWM period is not complete
```

The CTM4 Pulse Width Modulation Submodule window displays the DASM 5, 6, 7, and 8 Status/Interrupt/Control Registers (PWM5SIC, PWM6SIC, PWM7SIC, PWM8SIC), the DASM 5, 6, 7, and 8 Period Registers (PWM5PER, PWM6PER, PWM7PER, PWM8PER), the DASM 5, 6, 7, and 8 Pulse Width Registers (PWM5PW, PWM6PW, PWM7PW, PWM8PW), and the DASM 5, 6, 7, and 8 Count Registers (PWM5CNT, PWM6CNT, PWM7CNT, PWM8CNT).

CTM6 Single-Action Submodule Window

Availability: 683xx Hardware Debugger

```
MySim32: CTM6 - Single Actions (SASM)
                        Input capture or output compare has occured??
                         |    Interrupt level is (Shared by A and B)
                         |     |   Interrupt arbitration bit 3.  {IARB += BIUMCR:IARB[2..0]}
                         |     |    |    Interrupts are disabled/enabled?
                         |     |    |     |   Use time base bus A or time base bus B?
                         |     |    |     |    |   Pin state when input or flip flop state when output
                         |     |    |     |    |    |   Set to force output compare (always reads 0)
                         |     |    |     |    |    |    |    Capture edge, match action or output pin state
                         |     |    |     |    |    |    |     |      Operating mode
          SIC   DAT  |  FLAG  IL   IARB  IEN  BSL  IN  FORCE  EDOUT  MODE
SASM12A   0001  0000 |  No   DIS  clr   dis   A   lo    0     clr   OP  ==> Output Port, pin set high
SASM12B   E364  0000 |  Yes  DIS  clr   dis   B   lo    1     clr   IC  ==> Input Capture on falling edge
SASM14A   1801  0000 |  No    1   SET   dis   A   lo    0     clr   OP  ==> Output Port, pin set high
SASM14B   0045  0000 |  No    1   SET   dis   A   lo    0     clr   OP  ==> Output Port, pin set high
SASM18A   0000  0000 |  No   DIS  clr   dis   A   lo    0     clr   IC  ==> Input Capture on falling edge
SASM18B   0191  8D0A |  No   DIS  clr   dis   B   HI    0     SET   OP  ==> Output Port, pin set low
SASM24A   0520  0000 |  No   DIS  clr   EN    B   lo    1     clr   IC  ==> Input Capture on falling edge
SASM24B   0080  0000 |  No   DIS  clr   dis   A   HI    0     clr   IC  ==> Input Capture on falling edge
```

The CTM6 Single-Action Submodule window displays the SASM 12A, 12B, 14A, 14B, 18A, 18B, 24A, 24B Status/Interrupt/Control Registers (SIC12A, SIC12B, SIC14A, etc.), and the SASM 12A, 12B, 14A, 14B, 18A, 18B, 24A, 24B Registers (S12DATA, S12DATB, S14DATA, etc.)

CTM6 Double-Action Submodule - Modes Window

Availability: 683xx Hardware Debugger

```
MySim32: CTM6 - Double Actions Modes (DASM)
DASM04: OPWM - Output pulse width modulation
        A compare on channel A sets   the ouptut pin to logic 1
        A compare on channel B clears the ouptut pin to logic 0
DASM05: IPWM - Input pulse width measurement
        Channel A captures on a rising  edge
        Channel B captures on a falling edge
DASM06: IC   - Input capture
```

The CTM6 Double-Action - Modes Submodule window displays the DASM 4, 5, 6, 7, 8, 9, 10, 26, 27, 28, and 29 Status/Interrupt/Control Registers (DASM4SIC, DASM5SIC, DASM6SIC, DASM7SIC, DASM8SIC, DASM9SIC, DASM10SIC, DASM26SIC, DASM27SIC, DASM28SIC, DASM29SIC).

See the Bits form of the CTM6 Double-Action Submodule window for display of additional information.

CTM6 Double-Action Submodule - Bits Window

Availability: 683xx Hardware Debugger

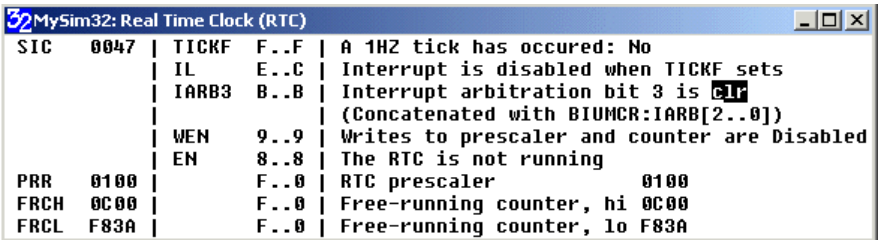
```
MySim32: CTM6 - Double Actions Bits (DASM)
                        An input capture or output compare has occured?
                         |   Interrupt level is
                         |    |   Interrupt arbitration bit 3.  {IARB += BIUMCR:IARB[2..0]}
                         |    |    |     If output then buffer operates mode is mode
                         |    |    |      |     Connected to time base bus
                         |    |    |      |      |  Pin function when input
                         |    |    |      |      |   |    Set to force chan-A compare, always reads 0
                         |    |    |      |      |   |     |     Set to force chan-B compare, always reads 0
                         |    |    |      |      |   |     |      |
                         |    |    |      |      |   |     |      |     Edge Polarity
                         |    |    |      |      |   |     |      |      |    Mode selecte string
                         |    |    |      |      |   |     |      |      |     |     Bits of resolution
          SIC  D-A  D-B   FLAG  IL  IARB3   WOR    BSL  IN  FORCA  FORCB  EDPLO  MODE    |
DASM04   938C 9200 988A   Yes   1  clr  wired-or  B  High   0      0      0     C      12   OPWM
DASM05   0224 0000 0000   No   DIS  clr  wired-or  A  Low    0      1      0     4      16   OCB
DASM06   8003 1900 0000   Yes  DIS  clr   normal   A  Low    0      0      0     3      16    IC
DASM07   0004 0101 0000   No   DIS  clr   normal   A  Low    0      0      0     4      16   OCB
DASM08   0326 0000 693A   No   DIS  clr  wired-or  B  Low    0      1      0     6      -   unused
DASM09   198D 0000 B003   No    1   SET   normal   B  High   0      0      0     D      11   OPWM
DASM10   B005 0701 0000   Yes   3  clr   normal   A  Low    0      0      0     5      16   OCAB
DASM26   0222 0000 0000   No   DIS  clr  wired-or  A  Low    0      1      0     2      16   IPM
DASM27   0041 0000 0000   No   DIS  clr   normal   A  Low    1      0      0     1      16   IPWM
DASM28   0000 0000 0000   No   DIS  clr   normal   A  Low    0      0      0     0      -    DIS
DASM29   0006 0000 0000   No   DIS  clr   normal   A  Low    0      0      0     6      -   unused
```

The CTM6 Double-Action Bits Submodule window displays the DASM 4, 5, 6, 7, 8, 9, 10, 26, 27, 28, and 29 Status/Interrupt/Control Registers (DASM4SIC, DASM5SIC, etc.), and the DASM Data A and B registers (DASM4A, DASM4B, DASM5A, etc.)

See the Modes form of the CTM6 Double-Action Submodule window for display of additional information.

Real Time Clock Window

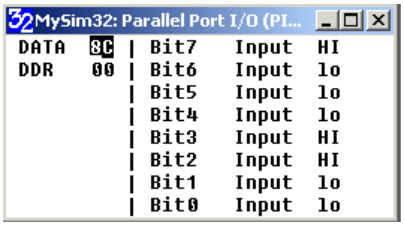
Availability: 683xx Hardware Debugger



The Real Time Clock window displays the RTC Status, Interrupt, Control Register (RT16SIC), the RTCSM Prescaler Register (RTCSM), and the RTCSM Free-Running Counter High and Low Registers (R16FRCH, R16FRCL).

Parallel Port I/O Submodule Window

Availability: 683xx Hardware Debugger



The Parallel Port I/O Submodule window displays the PIOSM Control Register (PIO17A) which controls data direction and either reads or writes the port data.

TouCAN Main Window

Availability: 683xx Hardware Debugger

```
My68376: TouCAN - Main
CANMCR    59E0 | STOP    F..F | TouCAN clocks are  ENABLED
               | FRZ     E..E | TouCAN is allowed to enter debug mode
               | HALT    C..C | On FRZ=1, TouCan goes to debug mode
               | NotRdy  B..B | TouCAN is in      low-power stop or debug mode
               | WakeMsk A..A | Wake up interrupt is  ENABLED
               | SoftRst 9..9 | Soft reset cycle completed
               | FrzAck  8..8 | TouCAN has entered debug mode and the prescaler is Disabled
               | SUPV    7..7 | S/U registers access at Supv priviledge level
               | SlfWake 6..6 | Self wake is  ENABLED
               | APS     5..5 | Auto power save  ENABLED; clocks stop and restart as needed
               | StopAck 4..4 | TouCAN is     in low-power stop mode
               | IARB    3..0 | Interrupt arbitration priority 0
CANICR    004F | ILCAN   A..8 | Interrupt request is disabled
               | IVBA    7..5 | Interrupt vector base upper 3 bits are 2
CANCTRL0 4480  | BoffMsk F..F | BUS OFF interrupt is Disabled
               | ErrMsk  E..E | ERROR   interrupt is  ENABLED
               | Rx1Mode B..B | On the CANRX1 pin, a logic 0 is dominant, 1 is recessive
               | Rx0Mode A..A | On the CANRX0 pin, a logic 1 is dominant, 0 is recessive
               | TxMode  9..8 | Full CMOS; positive polarity (CANTX0=0,CANTX1=1 is dominant)
               | SAMP    7..7 | Sample each receive bit three times
               | LOOP    6..6 | Internal loopback is Disabled
               | TSYNC   5..5 | Timer synchrnization Disabled
               | LBUF    4..4 | Message buffer with lowest ID is transmitted first
               | PROPSEG 2..0 | PROPSEG=0  -->     Propogation Segment Time: 000,059.6 (ns)
 PRESDIV/CTRL  | PRESDIV 7..0 | PRESDIV=60 -->                     S-Clock: 000,059.6 (ns)
         6000  | RJW     7..6 |     RJW=0  --> Resynchronization Jump Width: 000,059.6 (ns)
               | PSEG1   5..3 |   PSEG1=0  -->       Phase Buffer Segment 1: 000,059.6 (ns)
               | PSEG2   2..0 |   PSEG2=0  -->       Phase Buffer Segment 1: 000,059.6 (ns)
ESTAT     XXXX |              | *** THE ESTAT STATES ARE NOT DISPLAYED ***
               |              | *** BECAUSE ESTAT READS ARE INTRUSIVE  ***
RXGMSKHI FFEF  |              | Receive global mask High is    FFEF
RXGMSKLO FFFE  |              | Receive global mask Low  is    FFFE
RX14MsHI FFEF  |              | Receive buffer 14 mask High is FFEF
RX14MsLO FFFE  |              | Receive buffer 14 mask Low  is FFFE
RX15MsHI FFEF  |              | Receive buffer 15 mask High is FFEF
RX15MsLO FFFE  |              | Receive buffer 15 mask Low  is FFFE
IMASK     454E |              | See buffer window
IFLAG     0000 |              | See buffer window
RXECTR      00 |              | Receive error counter is  00
TXECTR      00 |              | Transmit error counter is 00
```

The TouCAN Main window displays the TouCAN Module Configuration Register (CANMCR), the TouCAN Interrupt Configuration Register (CANICR), the TouCAN Control Registers 0, 1 and 2 (CANCTRL0, CANCTRL1 and CANCTRL2), the touCAN Prescaler Divide Register (PRESDIV), the Receive Global Mask High and Low Registers (RXGMSKHI, RXGMSKLO), the Receive Buffer 14 and 15 High and Low Registers (RX14MSKHI, RX14MSKLO, RX15MSKHI, RX15MSKLO), the Interrupt Mask and Flag Registers (IMASK, IFLAG), the Receive and Transmit Error Counters (RXECTR, TXECTR).

TouCAN Buffers Window

Availability: 683xx Hardware Debugger

```
My68376: TouCAN - Buffers
Buf  Ctrl | ID   ID   |        DATA BYTES          |    INTERRUPTS
 #   /Sts | HIGH LOW  | D0,D1 D2,D3  D4,D5 D6,D7 |  Enabled?  Flag
15   N/A  | DEDF E645 | FB,5F FF,EE  4D,AF FD,F3 |  Disabled  clr
14   N/A  | 2BF7 FFFF | F7,FF FA,36  F7,F7 D7,F5 |   ENABLED  clr
13   N/A  | 1FEF 7C5F | 3E,BE 7F,B7  B6,FE AF,75 |  Disabled  clr
12   N/A  | FFD7 75EB | 7B,F4 FF,FF  6F,EF 8F,9B |  Disabled  clr
11   N/A  | BBFB FFDF | A4,5F 5B,67  CF,CD 7D,E7 |  Disabled  clr
10   N/A  | FEFF B7F7 | 7F,C9 FC,EF  0F,76 9A,A3 |   ENABLED  clr
 9   N/A  | AEA7 E59D | FD,CF 57,72  FF,EF DD,F7 |  Disabled  clr
 8   N/A  | DDF6 7DFF | FE,FF 56,EF  DE,3F CF,D7 |   ENABLED  clr
 7   N/A  | 5DBF EBDF | FD,3F D9,27  CD,B6 B7,BF |  Disabled  clr
 6   N/A  | FEC7 FFDF | EF,DE FB,F7  DF,DC FF,FF |   ENABLED  clr
 5   N/A  | 73F9 EF3A | FF,CF 7F,EB  1F,B4 71,FA |  Disabled  clr
 4   N/A  | 57FE F7AF | 7F,D3 F2,FF  9D,7A DF,FF |  Disabled  clr
 3   N/A  | 2AF7 EAEB | F7,FF DF,BF  F7,F7 7D,F7 |   ENABLED  clr
 2   N/A  | FD7D F67F | 7D,B5 D9,FB  2F,FF B5,E7 |   ENABLED  clr
 1   N/A  | FF7B FFFF | EE,F9 FF,EB  FF,AE FD,7D |   ENABLED  clr
 0   N/A  | FFEA F79E | AF,DF FF,7F  B7,5D FC,BE |  Disabled  clr
```

The TouCAN Buffers window displays information for each of the 16 buffers. For each buffer the status, identification (high and low), eight data bytes, interrupt enable/disable, and interrupt set/clear state is displayed

QADC Main Window

Availability: 683xx Hardware Debugger

```
My68376: QADC - Main
QADCMCR 4084 | STOP   F..F | The QADC clock is enabled
             | FRZ    E..E | QADC finishes any current conversion, then freezes
             | SUPV   7..7 |          All tables and        registers are supervisor only
             | IARB   3..0 | QADC interrupt arbitration priority 4
QADCINT 313C | IRLQ1  E..C | Queue 1 interrupt is level 3  (0=disabled, 7=highest)
             | IRLQ2  A..8 | Queue 2 interrupt is level 1  (0=disabled, 7=highest)
             | IVB    7..0 | Interrupt vector base is 3C
             |             |    [2 lsb's cannot be written and always read 0]
QACR0   0133 | MUX    F..F | Internally multiplexed, 16 possible channels
             | PSH    8..4 | PSH=13 --> Prescaler clock hi time is 001,192.1 ns
             | PSA    3..3 | QCLK high and low times are not modified
             | PSL    2..0 | PSL=3  --> Prescaler clock lo time is 000,238.4 ns
QACR1   0000 | CIE1   F..F | Queue 1 completion interrupt is disabled
             | PIE1   E..E | Queue 1 pause interrupts are disabled
             | SSE1   D..D | Enables a single-scan after trigger event
             |             | [This bit always reads as a zero]
             | MQ1    A..8 | Disabled mode, conversions do not occur
QACR2   0027 | CIE2   F..F | Queue 2 completion interrupt is disabled
             | PIE2   E..E | Queue 2 pause interrupts are disabled
             | SSE2   D..D | Enables a single-scan after trigger event
             |             | [This bit always reads as a zero]
             | MQ2    C..8 | Single-scan mode,     [3..0]=0, Interval is     -     micro-seconds
             |             | Disabled mode, conversions do not occur
QASR    0000 | CF1    F..F | Queue 1 scan is not complete
             | PF1    E..E | Queue 1 has not reached a pause
             | CF2    D..D | Queue 2 scan is not complete
             | PF2    C..C | Queue 2 has not reached a pause
             | TOR1   B..B | No          queue 1 trigger events have occured
             | TOR2   B..B | No          queue 2 trigger events have occured
             | QS     9..6 | Queue 1 idle,   Queue 2 idle
             | CWP    5..0 | Conversion Command Word 00 is currently or last executed
```

The QADC Main window displays the QADC Module Configuration Register (QADCMCR), the QADC Interrupt Register (QADCINT), the QADC Control Registers 0, 1, and 2 (QACR0, QACR1, QACR2), and the QADC Status Register (QASR).

QADC Ports Window

Availability: 683xx Hardware Debugger

My68376: QADC - Ports

DDRQ 7100	AN59	F..F	PQA.7/AN59	Input	0
PORTQ 4100	AN58	E..E	PQA.6	Output	1
	AN57	D..D	PQA.5	Output	0
	AN56/ETRIG2	C..C	PQA.4	Output	0
	AN55/ETRIG1	B..B	PQA.3/AN55	Input	0
	AN54/MA2	A..A	PQA.2/AN54	Input	0
	AN53/MA1	9..9	PQA.2/AN54	Input	0
	AN52/MA0	8..8	PQA.2	Output	1
	AN51	7..7	PQB.7/A51	(always)	0
	AN50	6..6	PQB.6/A50	(always)	0
	AN49	5..5	PQB.5/A49	(always)	0
	AN48	4..4	PQB.4/A48	(always)	0
	AN[3]/ANz	3..3	PQA.3/AN3		0
	AN[2]/ANy	2..2	PQA.2/AN2		0
	AN[1]/ANx	1..1	PQA.1/AN1		0
	AN[0]/ANw	0..0	PQA.0/AN0		0

The QADC Ports window displays the QADC Port Data Direction Register (DDRQA) and the QADC Port A and B registers (PORTQA, PORTQB).

QADC Channels Window

Availability: 683xx Hardware Debugger

My68376: QADC - Channels

P --- PAUSE after executing current CCW?
BYP --- Use BYPASS amplifier?
IST --- Input sample time, ns

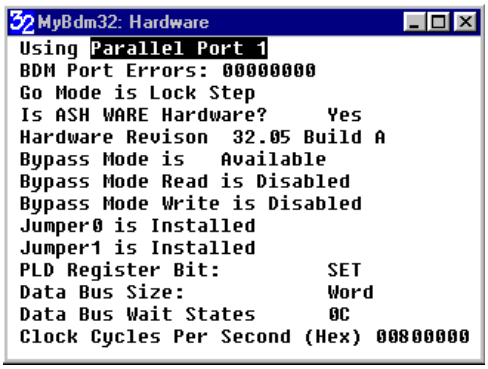
CCW[X]	VAL	9..9	8..8	7..6	CHAN 5..0	RESULTS RHT JST UNS	LFT JST SND	LFT JST UNS
0	00FF	..	no	16-QCLK's=022,888.2	3F==>PQA.0/AN0	02FF	3FC0	BFC0
1	03FF	PAUSE	YES	16-QCLK's=022,888.2	3F==>PQA.0/AN0	0357	55C0	D5C0
2	03FF	PAUSE	YES	16-QCLK's=022,888.2	3F==>PQA.0/AN0	013F	CFC0	4FC0
3	02D9	PAUSE	no	16-QCLK's=022,888.2	19==>PQA.0/AN0	03C8	7200	F200
4	021F	PAUSE	no	2-QCLK's=002,861.0	1F==>PQA.0/AN0	00FB	BEC0	3EC0
5	02FF	PAUSE	no	16-QCLK's=022,888.2	3F==>PQA.0/AN0	03EF	7BC0	FBC0
6	02FF	PAUSE	no	16-QCLK's=022,888.2	3F==>PQA.0/AN0	03BE	6F80	EF80
7	02D1	PAUSE	no	16-QCLK's=022,888.2	11==>PQA.0/AN0	03EB	7AC0	FAC0
8	01FF	..	YES	16-QCLK's=022,888.2	3F==>PQA.0/AN0	03FA	7E80	FE80
9	03FF	PAUSE	YES	16-QCLK's=022,888.2	3F==>PQA.0/AN0	01F5	FD40	7D40
A	03FF	PAUSE	YES	16-QCLK's=022,888.2	3F==>PQA.0/AN0	006F	9BC0	1BC0
B	03D8	PAUSE	YES	16-QCLK's=022,888.2	18==>PQA.0/AN0	03DE	7780	F780
C	007F	..	no	4-QCLK's=005,722.0	3F==>PQA.0/AN0	00FF	BFC0	3FC0
D	03BE	PAUSE	YES	8-QCLK's=011,444.1	3E==>PQA.0/AN0	03FF	7FC0	FFC0
E	03FE	PAUSE	YES	16-QCLK's=022,888.2	3E==>PQA.0/AN0	03FF	7FC0	FFC0
F	015F	..	YES	4-QCLK's=005,722.0	1F==>PQA.0/AN0	03D9	7640	F640
10	01BE	..	YES	8-QCLK's=011,444.1	3E==>PQA.0/AN0	00FF	BFC0	3FC0
11	005F	..	no	4-QCLK's=005,722.0	1F==>PQA.0/AN0	03FF	7FC0	FFC0
12	01D8	..	YES	16-QCLK's=022,888.2	18==>PQA.0/AN0	03FF	7FC0	FFC0
13	03DD	PAUSE	YES	16-QCLK's=022,888.2	1D==>PQA.0/AN0	03D9	7640	F640
14	03EA	PAUSE	YES	16-QCLK's=022,888.2	2A==>PQA.0/AN0	00FF	BFC0	3FC0
15	00FE	..	no	16-QCLK's=022,888.2	3E==>PQA.0/AN0	03FF	7FC0	FFC0
16	00EF	..	no	16-QCLK's=022,888.2	2F==>PQA.0/AN0	03FF	7FC0	FFC0
17	01FF	..	YES	16-QCLK's=022,888.2	3F==>PQA.0/AN0	03D9	7640	F640
18	00FF	..	no	16-QCLK's=022,888.2	3F==>PQA.0/AN0	00FF	BFC0	3FC0
19	03FF	PAUSE	YES	16-QCLK's=022,888.2	3F==>PQA.0/AN0	03FF	7FC0	FFC0
1A	03FF	PAUSE	YES	16-QCLK's=022,888.2	3F==>PQA.0/AN0	03FF	7FC0	FFC0
1B	03D9	PAUSE	YES	16-QCLK's=022,888.2	19==>PQA.0/AN0	03D9	7640	F640
1C	00FF	..	no	16-QCLK's=022,888.2	3F==>PQA.0/AN0	00FF	BFC0	3FC0
1D	03FF	PAUSE	YES	16-QCLK's=022,888.2	3F==>PQA.0/AN0	03FF	7FC0	FFC0
1E	03FF	PAUSE	YES	16-QCLK's=022,888.2	3F==>PQA.0/AN0	03FF	7FC0	FFC0
1F	03D9	PAUSE	YES	16-QCLK's=022,888.2	19==>PQA.0/AN0	03D9	7640	F640
28	03D9	PAUSE	YES	16-QCLK's=022,888.2	19==>PQA.0/AN0	00FF	BFC0	3FC0
29	01FD	..	YES	16-QCLK's=022,888.2	3D==>PQA.0/AN0	03FF	7FC0	FFC0
2A	02DB	PAUSE	no	16-QCLK's=022,888.2	1B==>PQA.0/AN0	03FF	7FC0	FFC0
2B	0391	PAUSE	YES	8-QCLK's=011,444.1	11==>PQA.0/AN0	03D9	7640	F640
2C	00FF	..	no	16-QCLK's=022,888.2	3F==>PQA.0/AN0	00FF	BFC0	3FC0
2D	03FF	PAUSE	YES	16-QCLK's=022,888.2	3F==>PQA.0/AN0	03FF	7FC0	FFC0
2E	03FF	PAUSE	YES	16-QCLK's=022,888.2	3F==>PQA.0/AN0	03FF	7FC0	FFC0
2F	03D9	PAUSE	YES	16-QCLK's=022,888.2	19==>PQA.0/AN0	03D9	7640	F640

The QADC Channels window displays the Conversion Command Word Table Registers (CCW0 to CCW2F), the Right Justified Unsigned Result Registers (RJURR0 to RJURR2F), the Left Justified Signed Result Registers (LJSRR0 to LJSRR63), and the Left Justified Unsigned Result Registers (LJURR0 to LJURR27).

Note that the QADC normally has up to 16 analog input channels but can directly support up to 40 analog channels with QADC-controlled external multiplexing.

683xx ASH WARE Hardware Window

Availability: 683xx Hardware Debugger



The first field indicates to which parallel port the BDM cable is attached. The BDM Port errors field gives the total number of low-level communications errors encountered. This field should never be greater than zero when the port communications are working properly. The Lock Step field indicates if the hardware is free running or is in lock step. Lock step is when the hardware is only single stepped. This greatly reduces the target execution speed and generally should not be selected.

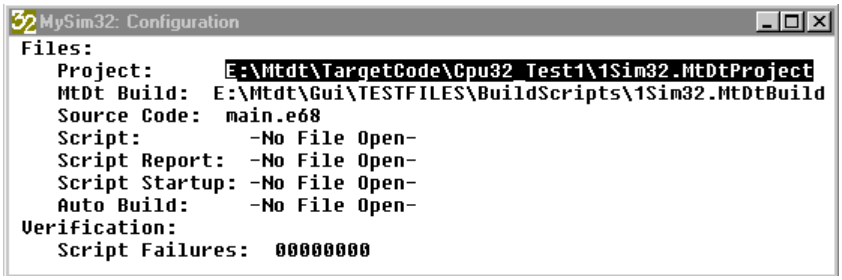
The Hardware Revision field gives the release number of the ASH WARE hardware.

Bypass mode is a special accelerated communications mode in which the normal BDM path is bypassed to speed up code download and upload. The three fields indicate if the mode is available, if bypass mode is enabled for reads, and if bypass mode is available for writes. Enabling bypass mode significantly improves code uploads/downloads and window display update rates.

The PLD Register bit field indicates if a bit within the ASH WARE development board's PLD is set or cleared. See the Development Board documentation for a detailed explanation of this bit.

CPU32 Simulator Configuration Window

Availability: CPU32 Simulator



Files

The name of the project file is displayed. See the *Project Sessions* chapter for a detailed explanation of this capability.

The name of the MtDt build script file is displayed. See the *Full-System Simulation* chapter and *the MtDt Build Commands File* section for detailed explanations of the available capabilities and format of this file.

The names of the open source code (executable image), primary script, primary script report, and startup script files are displayed. These files are listed relative to the path of the open project file.

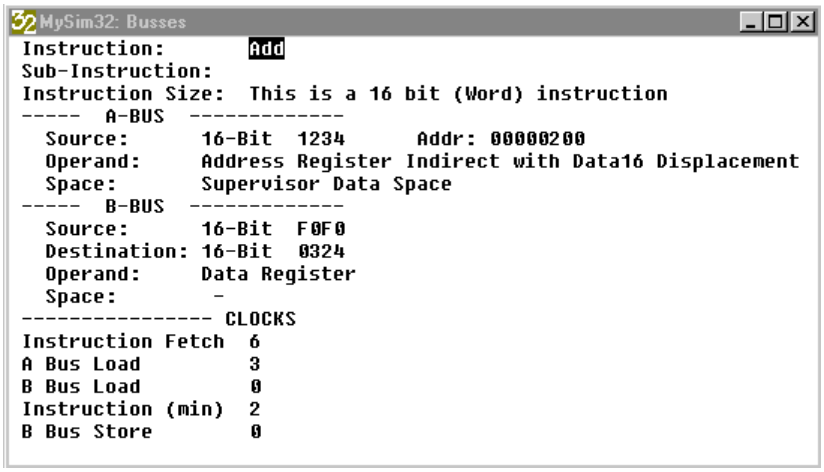
The auto-build file name is displayed. The auto-build file allows direct building of the executable image from within MtDt and is covered in the *Auto-Build Batch File Options Dialog Box* section.

Verification

The count of the number of user-defined verification tests that have failed since the last MtDt reset is displayed. User-defined tests are part of script commands files. See the Script Commands Groupings section for a description of the various script commands that support user-defined tests.

CPU32 Simulator Busses Window

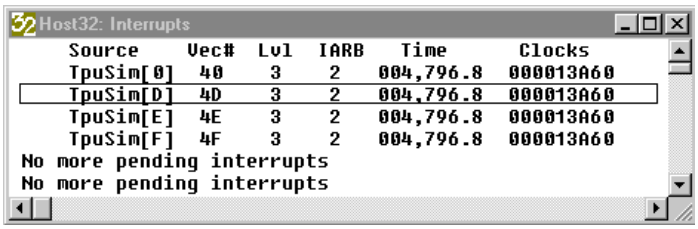
Availability: CPU32 Simulator



The CPU32 Simulator Busses window displays detailed information regarding the just executed instruction. This includes the data flow within the simulated CPU32, the data flow between the simulated CPU32 and external memory, and timing information.

CPU32 Simulator Interrupt Window

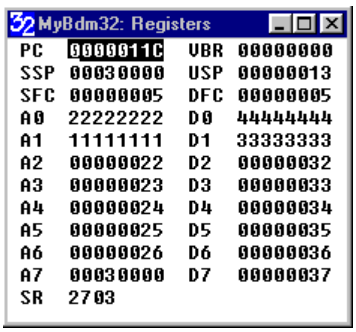
Availability: CPU32 Simulator



The CPU32 Simulator Interrupt window displays detailed information regarding the currently pending CPU32 interrupts. Towards the top are the highest priority (first to be serviced) interrupts and towards the bottom are the lowest priority (last to be serviced) interrupts. Once interrupts are serviced they are removed from this window.

CPU32 Registers Window

Availability: CPU32 Simulator and 683xx Hardware Debugger



The CPU32 Registers window displays the CPU32's registers. These are the Program Counter (PC), the Vector Base Register (VBR), the Supervisor and User Stack Pointers (SSP and USP), the Alternate Function Code registers (SFC and DFC), the Address registers (A0 through A7), the Data registers (D0 through D7), and the Status Register (SR).

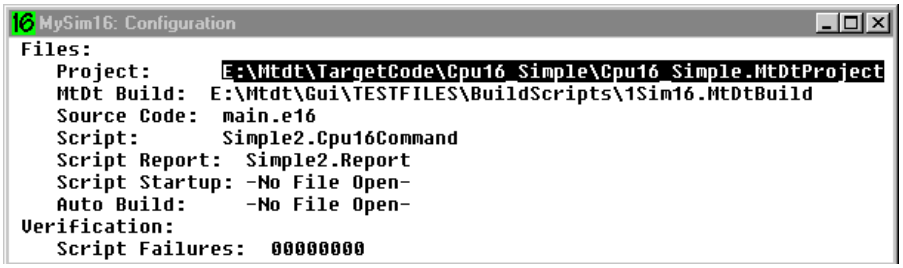
CPU32 Disassembly Dump Window

Availability: CPU32 Simulator and 683xx Hardware Debugger

The CPU32 Disassembly Dump window displays disassembled code starting at the target's program counter.

CPU16 Simulator Configuration Window

Availability: CPU16 Simulator



Files

The name of the project file is displayed. See the *Project Sessions* chapter for a detailed explanation of this capability.

The name of the MtDt build script file is displayed. See the *Full-System Simulation* chapter and *the MtDt Build Commands File* section for detailed explanations of the available capabilities and format of this file.

The names of the open source code (executable image), primary script, primary script report, and startup script files are displayed. These files are listed relative to the path of the open project file.

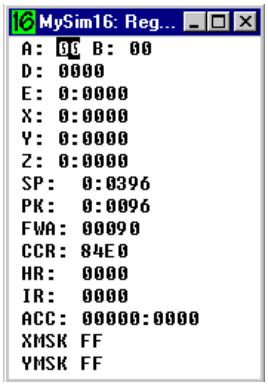
The auto-build file name is displayed. The auto-build file allows direct building of the executable image from within MtDt and is covered in the *Auto-Build Batch File Options Dialog Box* section.

Verification

The count of the number of user-defined verification tests that have failed since the last MtDt reset is displayed. User-defined tests are part of script commands files. See the *Script Commands Groupings* section for a description of the various script commands that support user-defined tests.

CPU16 Register Window

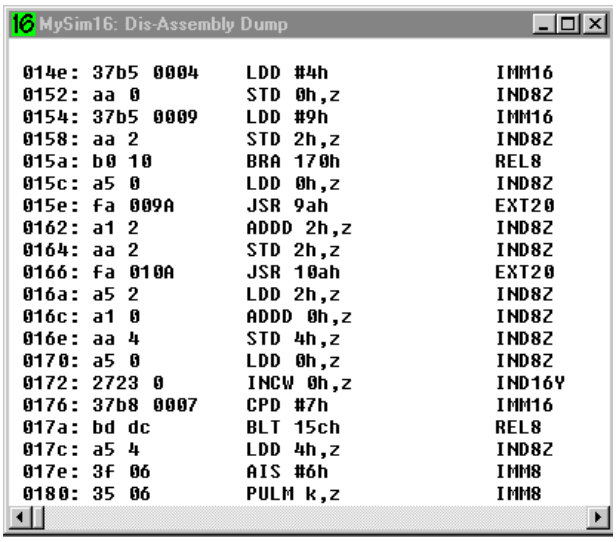
Availability: CPU16 Simulator



The CPU16 Register window displays the various CPU16 registers.

CPU16 Disassembly Dump Window

Availability: CPU16 Simulator



The CPU16 Disassembly Dump window displays instructions in memory starting with the program counter. The instructions address, the hexadecimal value of the instruction, the actual disassembled instruction, and the parse mode are displayed. Note that the disassembled instructions are viewable in a generally more useful fashion by selecting mixed assembly viewing mode from within source code windows.

Unsupported Window

This window for diagnostic purposes only and is not supported for customer use.

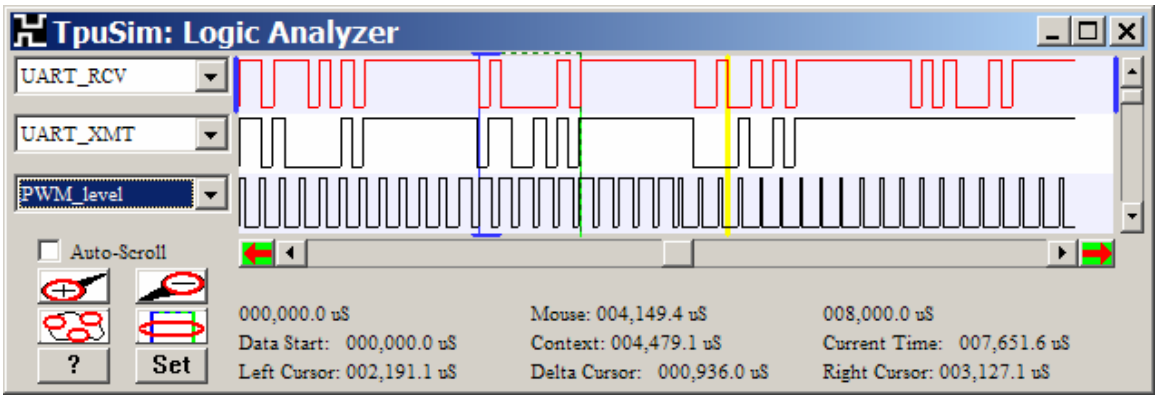
LOGIC ANALYZER

Overview

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.

The following topics are available regarding the Logic Analyzer.

- ? Executing to a precise time
- ? View node selection
- ? The active waveform
- ? The left and right cursors
- ? Bringing an out-of-view cursor into view
- ? Using the mouse
- ? The vertical yellow context cursor
- ? Scroll bars
- ? Waveform boundary time indicators
- ? Buffer data start indicator
- ? Context time indicator
- ? Current time indicator
- ? Left, right and delta cursor time indicators
- ? Auto scroll the waveform display as the simulator executes
- ? Button controls
- ? Data storage buffer



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.

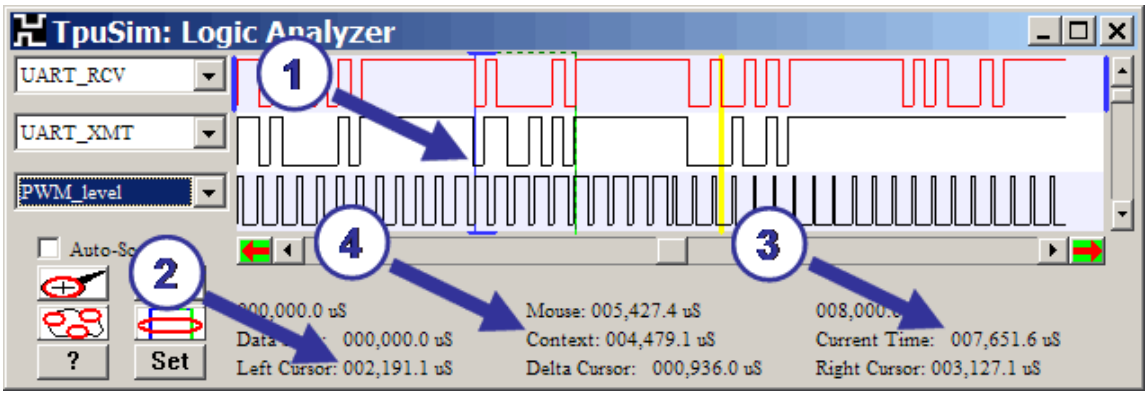
? Place the cursor over the field showing the time of the left cursor indicated by the second arrow. An open hand will appear indicating that this "time" can be copied.

? Begin dragging the time by holding the left mouse button down to form a closed hand.

? Drag the time (closed hand) over the "Current Time" field shown by arrow 3.

? Drop the time into the "current time" field by releasing the left mouse button.

? The simulator will reset and execute to precisely 2,191.1 microseconds.



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.

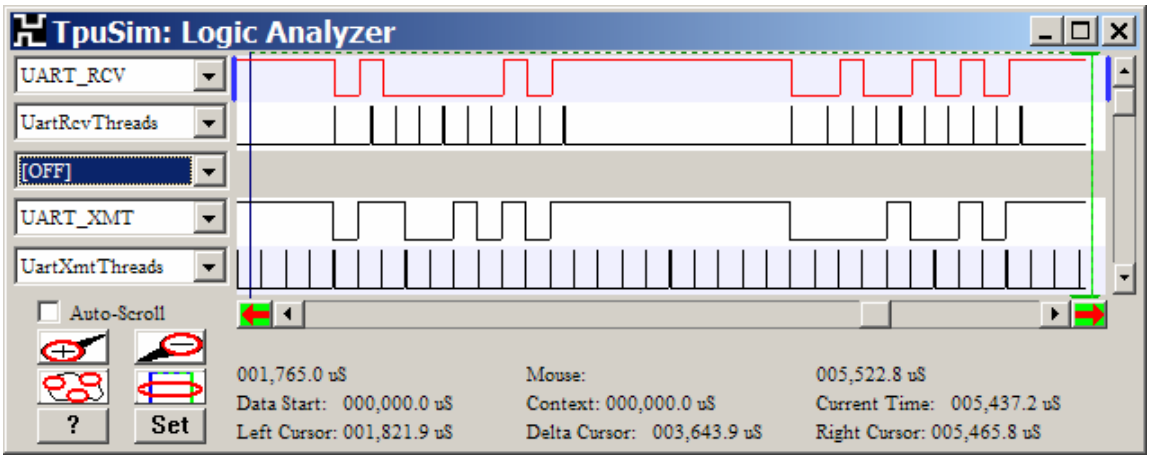
View Waveform Selection

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.

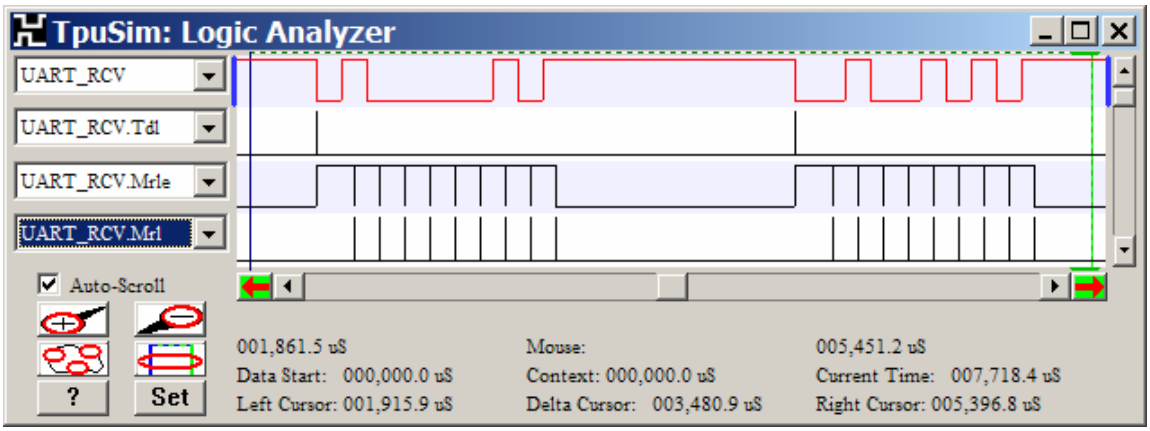
New 3.41 Viewing Thread Activity Nodes (eTPU and TPU Only)

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.



New 3.41 Viewing Channel Nodes (eTPU and TPU Only)

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.



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.

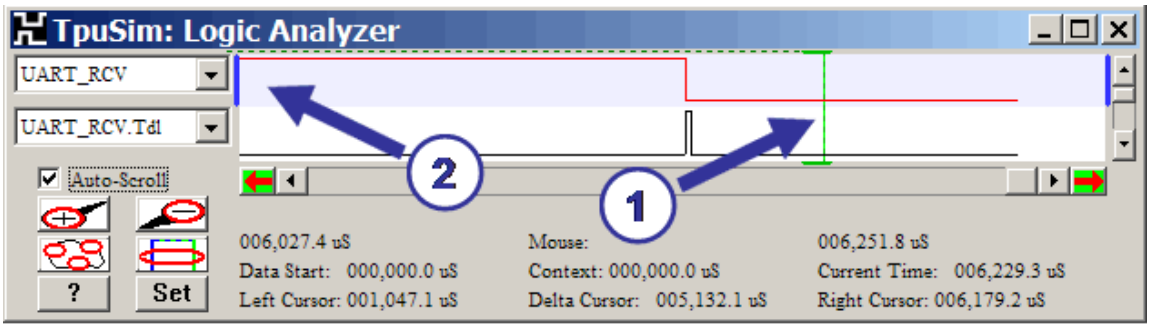
The Left and Right Cursors

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.

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

LOGIC_ANALYZER_CURSOR_INDICATORS

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.

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.

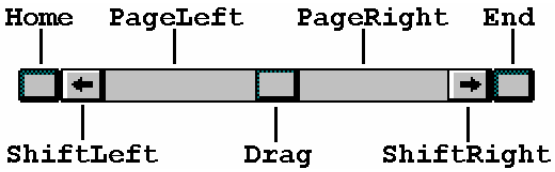
This context capability is a powerful debugging tool. The source code associated with the trace data also pops into view. This allows you to line up previously executed source code, the trace data, and in some cases the call stack data. If you want to re-execute to that context time, use the mouse to 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

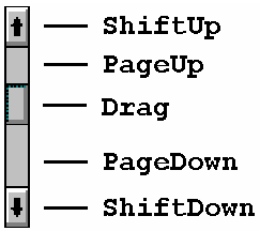
The context time indicator displays the time with the IDE-wide context time. For example, select a past-event in a trace window. The time associated with this event becomes the context time. This context time appears as the vertical yellow context cursor in the waveform display. Drag and drop this time into the current time, as explained in the, Executing to a Precise Time section.

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.



Waveform Boundary Time Indicators

The left and right time indicators show the time associated with the left-most and right-most visible portions of the waveform pane.

The waveform pane can be changed so that a different portion 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, the release the left mouse button.

Buffer Data 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.

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.

Auto-Scroll the Waveform Display as the Target Executes

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.

Button Controls

Zoom In

Selecting the Zoom In button narrows the view to the transition data at the center of the 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.

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.

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.

DIALOG BOXES

Dialog boxes provide an interface for setting the various Simulator options. This version of MtDt has the dialog boxes listed below:

- ? File Open, Save, and Save As Dialog Boxes
- ? Auto-Build Batch File Options Dialog Box
- ? Goto Time Dialog Box
- ? Occupy Workshop Dialog Box
- ? IDE Options Dialog Box
- ? Workshop Options Dialog Box
- ? Message Options Dialog Box
- ? Source Code Search Options Dialog Box
- ? Reset Options Dialog Box
- ? Logic Analyzer Options Dialog Box
- ? Thread Options Dialog Box
- ? Complex Breakpoint Dialog Box
- ? Trace Options Dialog Box
- ? Local Variable Options Dialog Box
- ? License Options Dialog Box
- ? BDM Options Dialog Box
- ? Memory Tool Dialog Box
- ? Insert Watch Dialog Box
- ? Watch Options Dialog Box

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 MtDt project filename is specified. MtDt loads a new configuration from this file.

From the Project Save As dialog box a MtDt 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 a 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.

Auto-Build Batch File Options Dialog Box

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 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.

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.

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.

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.

TPU Simulator Target Only

The following settings are only available for 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.

Workshops 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.

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 a two targets, one 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.

Source Code Search Options 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.

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.

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

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.

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 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.

Channel Group Options 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.

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.

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. 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.

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 generally not available in the symbol table. This setting allows you to specify how many elements to display.

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.

BDM Options Dialog Box

The following describes the Background Debug Mode (BDM) Options dialog box.

Halt Target for Periodic Update

When this is selected, the free-running hardware is halted when MtDt redraws its windows. The advantage of selecting this is that the windows can be updated as the target runs. The disadvantages are that it effectively disables the target's ability to service interrupts and in some cases the target will execute significantly slower.

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, data word size can of eight-bits, 16-bits, or 32-bit. 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 is 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.

Insert Watch Dialog Box

The Insert Watch dialog box is used whenever a new watch is added to the watch window using the insert function.

Two lists support the ability to either re-select (and presumably modify) a currently active watch, and the ability to re-select a previously discarded watch string. You can also edit a brand new string

or modify any string selected from the aforementioned lists.

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.

About ASH WARE's Multiple Target Development Tool

MtDt is copyrighted in 1994 through 2000 by 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.

BDM Port Dialog Box

The BDM Port dialog box is used only in the rare event that MtDt finds BDM target hardware on multiple parallel ports. In this case you are asked to select target hardware from a list found during BDM auto-detection.

MENUS

The menus allow the user to access the various capabilities of MtDt. The menus appear at the top of MtDt. 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 <F10> and the <ALT> function 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.

MtDt provides the following menus.

- ? Files Menu
- ? Step Menu
- ? Run Menu
- ? Breakpoints Menu
- ? Activate Menu
- ? View Menu
- ? Window Menu
- ? Options Menu
- ? Help Menu

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 MtDt 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 MtDt project files. This submenu provides the capability to both create a new MtDt 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.

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.

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.

Step Menu

Into

This runs the active target until one line of source code is executed. If a function is called, 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.

Time Slot

This runs MtDt until a TPU time slot transition occurs. MtDt stops just before execution of the first microinstruction of the TPU channel service routine. If no time slot transitions occur, MtDt continues to run until stopped by selecting the Stop submenu in the Run menu.

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.

Run Menu

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.

Breakpoints Menu

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 instruction 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 cursor is beyond the last instruction associated with the file, then this will not work.

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 disabled breakpoints. Selecting this submenu causes MtDt to "forget" which instructions had breakpoints.

Help

This accesses this help screen.

Activate Menu

The Activate menu allows you to specify which workshop and target are active. These capabilities are accessed via the following submenus.

Workshop

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.

View Menu

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.

Window 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.

Options Menu

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.

BDM

This opens the BDM Options dialog box. Note that a 683xx Hardware Debugger target 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, 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.

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.

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.

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.

Help Menu

The Help menu provides the following options.

Contents

This accesses the contents screen of MtDt's on-line help program.

Using Help

This 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.

HOT KEYS, TOOLBAR, AND STATUS INDICATORS

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.

To the right of the menu help indicator is the ticks indicator. This displays the number of ticks since the last target reset, 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.

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 indictor is a clock. A clock is a clock. Use this to determine if it is time to go home.

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.

List of Available System Simulators

The currently available products are listed below and are explained in greater detail later in this section.

- ? eTPU/CPU System Simulator
- ? TPU/CPU32 System Simulator
- ? TPU/CPU16 System Simulator

List of Available Single-Target Simulators and Debuggers

- ?
- ? eTPU Stand-Alone Simulator
- ? TPU Stand-Alone Simulator
- ? TPU Standard Mask Simulator
- ? CPU32 Stand-Alone Simulator
- ? CPU16 Stand-Alone Simulator
- ? 683xx Hardware Debugger

List of Supported Targets

The currently supported targets are listed below and are explained in greater detail later in this section.

- ? eTPU simulation engine
- ? TPU simulation engine
- ? CPU32 simulation engine
- ? CPU32 hardware across a BDM port
- ? CPU16 simulation engine

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.

TPU/CPU32 System Simulator

Our TPU/CPU32 System Simulator supports instantiation and simulation of an arbitrary number and combination of TPUs and CPU32s. A dedicated external system modeling CPU32 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 and among targets is fully retained as the full system simulation progresses.

Standard single-target debugging capabilities such as single stepping, breakpoints, goto cursors, etc., are fully supported within this full system simulation environment. For instance, breakpoints can be inserted and activated within several targets simultaneously. The full system simulation is halted as soon as a breakpoint is encountered in any target.

TPU/CPU16 System Simulator

Our TPU/CPU16 System Simulator supports instantiation and simulation of an arbitrary number and combination of TPUs and CPU16s. A dedicated external system modeling CPU16 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.

Standard single-target debugging capabilities such as single stepping, breakpoints, goto cursors, etc., are fully supported within this full system simulation environment. For instance, breakpoints can be inserted and activated within several targets simultaneously. The full system simulation is halted as soon as a breakpoint is encountered in any target.

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.

TPU Stand-Alone Simulator

This product is a single-target version that uses only a single instance of our TPU Simulation engine. Because it is a stand-alone product the user must use script commands files to act as the host and TPU test vector files to act as the external system. This is the original ASH WARE product.

TPU Standard Mask Simulator

Our TPU Standard Mask Simulator allows you to develop your CPU code used in conjunction with any standard Freescale TPU microcode. Currently ASH WARE supports the following three TPU Microcode builds.

- ? MASK A
- ? MASK G

? MASK MPC5xx

This product supports all the capabilities of our system simulator products with the single exception that you cannot load microcode that you have modified. Instead, the only TPU microcode that can be loaded is the one of the aforementioned standard TPU microcode masks.

Although you can tailor your configuration to any specific microcontroller mapping the following three configurations are the defaults. The default configurations are usually sufficient and intricacies of custom configuration generation need not be explored.

? MASK A: 683xx with one TPU mapped at 0xFFFE00

? MASK G: 683xx with one TPU mapped at 0xFFFE00

? MASK MPC5xx: MPC555 with two TPUs mapped at 0x304000 and 0x304400.

CPU32 Stand-Alone Simulator

Our CPU32 Stand-Alone Simulator allows you to load code into a single simulated CPU32. You can also load the standard mask TPU microcode into a simulated TPU.

CPU16 Stand-Alone Simulator

Our CPU16 Stand-Alone Simulator allows you to load code into a single simulated CPU16. You can also load the standard mask TPU microcode into a simulated TPU.

683xx Hardware Debugger

Our 683xx Hardware Debugger allows you to control any 683xx microcontroller across a BDM port. You can load and execute code. Standard debugging capabilities such as single step, breakpoints, goto cursor, etc., are supported.

Operational status windows support viewing of CPU32 registers, memory, peripherals, etc.

eTPU Simulation Engine

The Enhanced Time Processing Unit, or eTPU, is a microsequencer sold by Freescale on a variety of microcontrollers. 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 eTPU 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.

TPU Simulation Engine

The Time Processing Unit, or TPU, is a microsequencer sold by Freescale on a variety of microcontrollers. There are currently three varieties, known as TPU1, TPU2, and TPU3. MtDt

supports all three varieties.

The TPU 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.

The TPU simulation engine is used for developing, debugging, and verifying operation of TPU microcode. User microcode is loaded from files currently generated by Freescale's TPU Microcode Assembler. These files are loaded into the TPU simulation target's microcode space. The storage format is the binary equivalent of the microcode actually executed by the TPU. MtDt also displays user microcode source files in source code windows, highlighting the line associated with the actual microinstruction being executed.

Additional TPU operational status information is displayed in various context windows. The windows include information about channel control registers, host CPU interface registers, microengine control registers, the scheduler, script commands files, source microcode files, etc.

The Time Processor Unit (TPU)

The TPU is a microsequencer developed by Freescale for measurement and generation of time-critical waveforms. It is a single silicon target that is offered on several microcontroller products, such as the MC68332 and the HC16.

The TPU has two operational modes. It can execute microcode out of its internal ROM, or it may execute custom microcode out of its internal RAM. In order to execute custom microcode, the microcontroller's operational software must first copy the microcode into its internal RAM and then configure the TPU to execute out of RAM.

Support for TPU1

MtDt supports the TPU1. This is the original TPU.

Support for TPU2

MtDt supports the TPU2. The following is a list of new and supported features of the TPU2, not available in TPU1.

- ? Entry table bank (ETBANK)
- ? Code bank
- ? Full parameter RAM
- ? Load zero using a RAM sub-instruction
- ? Additional TCR2 and TCR1 clock options
- ? Flag2 set, clear, and conditional execution
- ? Branch on current pin state
- ? Negation of MRL and TDL using TBS sub-instruction
- ? Match on equal

The entry bank is controlled in the TPU simulation engine using a write_entry_bank() script command as described in *the TPU Bank and Mode Control Script Commands* section. The current state of this field is displayed in the microsequencer window.

The code bank is set during a time slot transition using bits 10 and 9 of the entry table. The user

modifies this directly within the source microcode. The current state of this field is displayed in the microsequencer window.

The TPU1 does not implement parameters six and seven of the parameter RAM for channels zero through 13. The TPU2 supports a full complement of parameter RAM for all channels.

The TPU2 supports a loading of zero using the parameter RAM sub-instruction. In the TPU1 one could make use of the undocumented feature that reading from the un-implemented parameter RAM returned a zero. In this case a zero was able to be effectively loaded. This capability is supported in the TPU2 by a direct clear or write of zero using the RAM sub-instruction.

New counter options support a higher resolution (faster count) for TCR1 and improved edge control (clock on rising pin edge, falling pin edge, or both pin edges) for TCR2. These include a DIV2 control bit for TCR1 and a T2CSL control bit for the TCR2. These fields are displayed in the configuration window. Script commands for controlling these two bits are described in the *TPU Clock Control Script Commands* section.

A Flag2, which is quite similar to Flag1 and Flag0, has been added.

A conditional branch can now be made on the current pin state using the PIN conditional. This field is displayed in the Execution Unit window. Previously only the PSL conditional could be used, and the PSL is updated only on a time slot transition or a write to the CHAN_REG register. The PIN conditional allows a branch based on the most current pin state.

MRL and TDL can be negated using the TBS field of instruction format three.

A match can be set to occur on an equal condition. Previously matches always occurred based on a greater than or equal logic. This match-on-equal allows the user to schedule a match further out in the future because, using this feature, the concept of past events is dropped. All events are effectively considered to occur in the future. This field is displayed in the Channel window.

Setting the TPU Mode

MtDt automatically sets the TPU mode when the source microcode files are parsed. The %type command is located and the appropriate TPU mode (TPU1, TPU2, or TPU3) is set based on the values of this field.

MtDt also supports a script command described in the *TPU Bank and Mode Control Script Commands* section for setting the TPU mode. Use of this command is highly discouraged since there are subtle differences among the TPU1, TPU2, and TPU3 that the TPU assembler hides from the user.

Support for TPU3

Version 2.1 of MtDt supports the TPU3. The following is a list of new and supported features of the TPU3.

- ? Enhanced TCR1 prescaler
- ? TCR2 pre-divider prescaler
- ? TCR2 controllable clock edge (rising, falling, or both)

The prescaler options listed above are accessible using script commands described in the *TPU Clock Control Script Commands* section.

CPU32 Simulation Engine

The CPU32 is sold on a number of Freescale microcontrollers, including the very popular 68332. The CPU32 Simulation Engine can be used both in stand-alone mode and in conjunction with other CPUs and peripherals including multiple CPU32s.

CPU32 Hardware across a BDM Port

The 683xx line of microcontrollers can debugged using this BDM Debugger.

CPU16 Simulation Engine

The CPU16 is sold on a number of Freescale microcontrollers, including the HC16 series. The CPU16 Simulation Engine can be used both in stand-alone mode and in conjunction with other CPUs and peripherals including multiple CPU16s.

FULL-SYSTEM SIMULATION

This section covers how an entire system consisting of multiple CPUs, peripherals, and models of the external system can be simulated. Of special interest are MtDt build script files because these are used to create the targets and specify how they interact.

In many cases, such as when using a TPU Stand-Alone Simulator or a 683xx Hardware Debugger, the standard build batch files are sufficient and little or no knowledge of MtDt build batch files is required. But when implementing complex multiple-target systems the detailed description of MtDt build batch files contained in this section is required.

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.

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., have been extended to the multiple target environment. 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

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.

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_bl ock("TpuSim", 0,  0x1ff, "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
- ? 32-bit read
- ? 32-bit write
- ? 64-bit read (not yet implemented)
- ? 64-bit write (not yet implemented)

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_addrs_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 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.

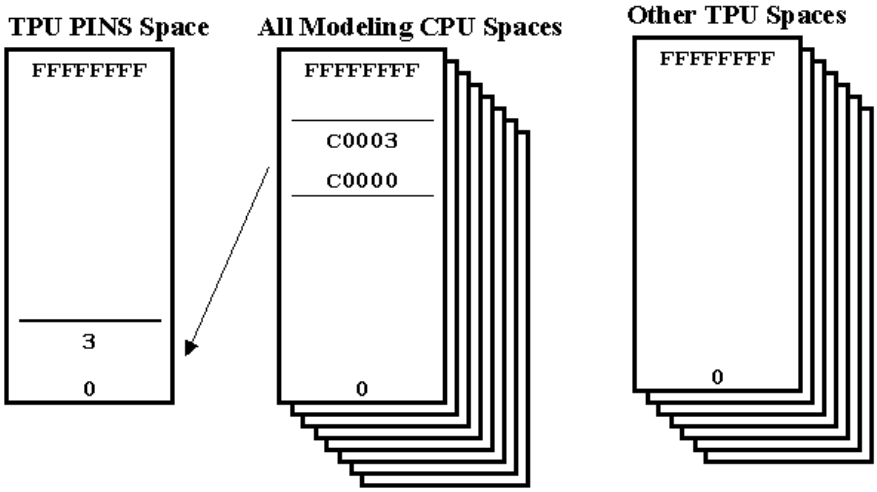
```
// ...
set_block_to_dock("HostCpu", "TpuDataDock", ALL_SPACES, "Tpu", 0-0x80000);
// ...
```

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 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.

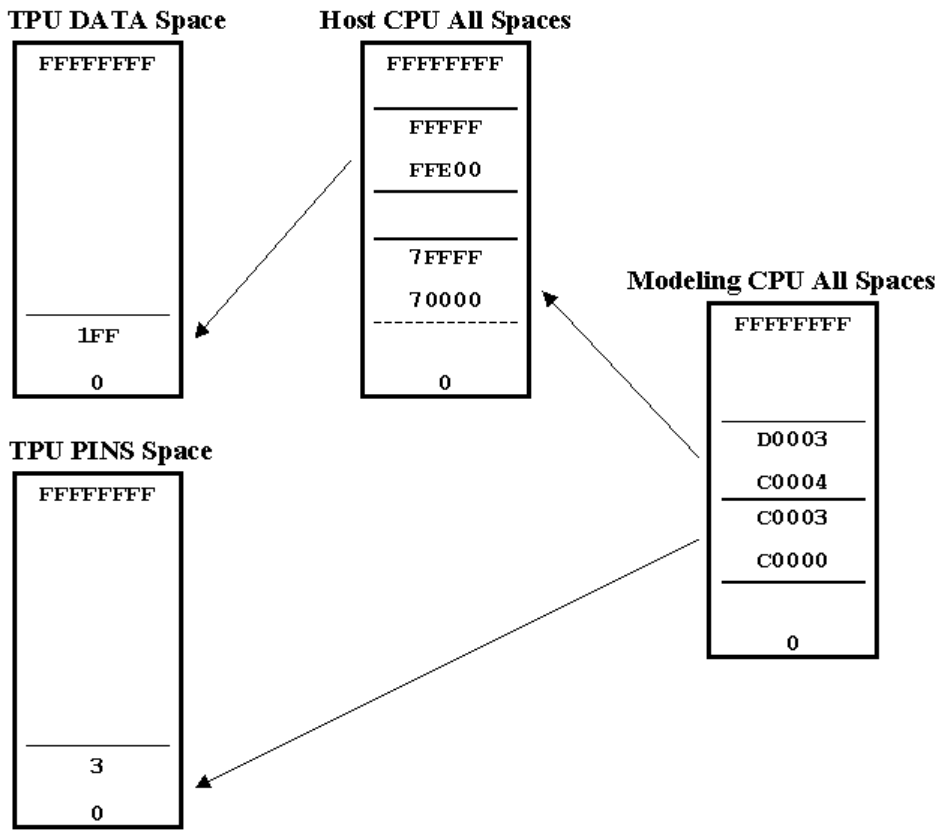
Shared Memory Timing

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.

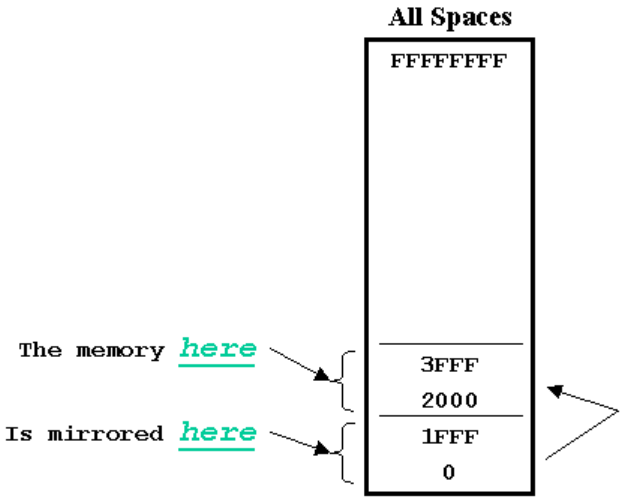
```
#define MEM_END 0xffffffff
// Create a target TPU
instantiate_target(TPU_SIM, "Tpu");
// Create a simulated memory block for the TPU's code  (microcode)
add_mem_block("Tpu", 0, 0x1fff, "Code", TPU_CODE_SPACE);
add_non_mem_block("Tpu", 0x2000, MEM_END, "B1", TPU_CODE_SPACE);
// Create a simulated memory block for the TPU's data (host interface)
add_mem_block("Tpu", 0,  0x1ff, "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);
```

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 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.