eTPU DevTool IDE

Reference Manual

by

John Diener and Andy Klumpp

ASH WARE, Inc.

Version 2.75

7/23/2023

ASH WARE Inc.

Table of Contents

1

Overview

This introductory section provides a high level overview of some of the key aspects of the eTPU Development Tool. More detail is provided in the remainder of this manual.

Source Code Files

Source code files are built using a compiler or an assembler and the resulting executable image is loaded into the target/core code space. When executing code, it is actually the executable image that gets executed. Source level debugging information cross references the source code files' line numbers and variables with the underlying executable image thereby supporting key debugging capabilities such as breakpoints, single stepping, viewing/modifying variables in watch windows, etc. Source code files can be edited directly within the IDE. Once edited, the source code is considered 'dirty' and will be rebuilt prior to resuming execution. The process of rebuilding and loading of the executable image is done automatically and largely transparently to the user.

Script Commands Files

Script commands files have several purposes. The 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.

Test Vector Files

Test vector files provide the user with one means of exercising input and I/O pins with complex test patterns. While a script commands file functionally represents the CPU or Host-MCU 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.

Concurrently Developing Code for Multiple Targets/Cores

Code for multiple targets/cores can be developed, and debugged, concurrently. Interactions between and among multiple targets/cores 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/core. The IDE supports instantaneous target/core switching such that it is possible in the system simulator, for example, to run to a CPU breakpoint, and then switch to a eTPU or MC33816 core and single step it. All the while, all targets/cores are kept fully synchronized.

The NXP eTPU

Of special interest are the Enhanced Time Processor Unit (eTPU) from NXP. For this wishing to develop their own eTPU Code, it is highly recommended that you obtain the book *"eTPU Programming Made Easy"* from AMT Publishing. Do not be misled by the title. This book is essential for beginners and experts alike. The eTPU training seminars are also highly recommended.

Miscellaneous Capabilities

The eTPU Development Tool has a number of additional features not yet mentioned. These include project sessions, source code files, functional verification, external logic simulation, multiple workshops, a rich set of dialog boxes, a menu system, and an IDE with hot keys, a toolbar, and target status indicators.

1.1 On-Line Help Contents

The following help topics are available.

2

Demo Descriptions

Note that YouTube videos covering several of these demos as well as feature tutorial demos are available on our website at www.ashware.com/product_videos.htm.

The following eTPU demos install by default with eTPU Development Tool. Several of these demos will also build using the ETEC compiler/linker toolset run from the command line.

NXP Set 1 - UART Demo

- Use of the NXP's Set12 UART function..
- Use of header file, 'etec_to_etpuc_uart_conv.h' to convert between the automatically generated 'MyCode_defines.h' and the standard NXP API interface file.

NXP Set 2 - Engine Demo - AN4907 (NEW)

- Use of the NXP's latest Set2, Cam, Crank, Fuel, Spark, Knock functions.
- Note that this is the NEW SET2 function, often referred as 'AN4907'.

NXP Set 2 - Engine Demo (OLD)

- Use of the NXP's Set2, Cam, Crank, Fuel, Spark, Knock functions.
- Note that this is the original SET2 function, a more recent version is available.

NXP Set 3 - ASDC Demo

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

NXP Set 4 - ASAC Demo

- Use of the NXP's Set12 ASAC, PWMF, and PWMMAC functions.
- Use of ASH WARE's auto-defines file, 'MyCode_defines.h'.

SD32 Evaluation Board Demo

- A basic S32DS eTPU project that runs on an MPC5777C Evaluation Board (EVB.)
- Configured to use a PEmicro Multilink Universal JTAG debugger
- TWO projects: one is built for the eTPU-AB module, the other is built for the eTPU-C module.
- ETPU Engine A, Channel 1: continuous "SOS" QOM output with dot time unit of 0.333s
- ETPU Engine C, Channel 2: 10Hz active low PWM output at 50% duty

Data Types Demo

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

Auto-Defines Demo

- Use of file the ETEC Compiler auto-generated file, 'MyCode_defines.h', to write & verify eTPU settings such as channel settings, variables, etc.

Auto-Struct Demo

- Each channel's variables are accessed using an automatically-generated structure. The structure is automatically generated by the ETEC Compiler
- Use of a generic CPU to build and simulate generic host-side code.
- CPU code build is controlled by DevTool ('internal' build'.)

Templates Demo

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

Worst Case Latency Demo

- An (optional) 'System Configuration' file sets the system parameters such as clock frequency, processor family, which functions run on which channels, channel priority, etc.
- The maximum allowed worst case latency (WCL) for each channel is specified in the System Configuration file
- Build fails if WCL requirements are not met.
- Analysis file shows resulting system behaviors such as WCL and WCTL for each channel.

Stepper Motor System Simulator Demo

- System simulator demo (both CPU and dual-eTPU are simulated)
- NXP's host-side API on a simulated CPU.
- The ASH WARE <>_defines file used in the host-side API.
- NXP's Set 1 Stepper Motor (SM) function.
- CPU code is built by calling an external batch file ('external' build.)

UART ETEC Mode System Simulator Demo

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

Byte Craft ETPU_C

- External Build using a Console Build Batch file (.BAT)
- Building code using the Byte Craft compiler
- Using a channel's output pin in the entry table.

Dual-eTPU STAC Bus Demo

- Simulating both eTPU Engines (64 channels total)
- TCR1/TCR2 export/import using the STAC bus.

eTPU/GTM, Four-Injectors, Two-Banks Demo

- A demo is available at the following link. www.ashware.com/product_videos.htm
- Co-simulation of the eTPU and the GTM.
- Banked injection, four injectors in organized into two banks. Each bank controlled by an GTM core.
- eTPU responding to an input 'CAM' signal to generate four injection pulses at the GTM's 'START1'-START4' pins.
- GTM responding to four START events to generate injection pulses on four injectors.
- eTPU channel output pins are drive to the GTM 'start' pins thereby generating precisely-timed injection pulses.
- Script command variables used to programmatically control and verify injection firing timing via the CAM signal.

3

Software Upgrades

World Wide Web Software Deployment

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

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

World Wide Web Software Upgrades

All versions since 2.1 can be upgraded directly through the World Wide Web. The following procedure is required when performing this upgrade. Note that versions prior to 2.1 cannot be upgraded via the World Wide Web.

- Upon receiving notification from ASH WARE that a new version of the eTPU Development Tool is available, download and re-install.

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

Network Floating Licenses

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

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

Dongle Licensing

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

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

Hearing about Software Releases

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

The eTPU Development Tool automatically displays an informational message when your software subscription is close to expiration. Note that the software license has no expiration so it is legal to use the eTPU Development Tool 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

3.1 Handling Multiple Versions

Each software version is installed (by default) into a unique installation directory by concatenating the program name with the version, as follows:

```
c:\Program Files (x86)\ASH WARE\eTPU2p DevTool IDE V1_01C
c:\Program Files (x86)\ASH WARE\eTPU Compiler V2_30C
```

Note that this differs from the way earlier ASH WARE Simulators and Debuggers were installed.

Note that the default installation directory can be overridden during the installation.

Although it is great to be able to have multiple software versions installed at once, this can lead to problems when trying to run regression tests and the version changes. How can the regression test 'find' the latest version? To solve this problem, during installation an environment variable is updated to reflect the last-installed version, as follows.

```
DEV_TOOL_ETPU_BIN="C:\Program Files (x86)\ASH WARE\eTPU2p DevTool IDE
V1_01C\"
ETEC_BIN="C:\Program Files (x86)\ASH WARE\eTPU Compiler V2_30C\"
```

Automated regression tests can find the software version using the following Console .BAT command sequence. Note that this is from the AutoStruct DevTool IDE Demo.

```
set DEV_TOOL_ETPU_EXE=%DEV_TOOL_ETPU_BIN%\ETpuDevTool.exe

echo Running "eTPU IDE Auto-Struct Demo" Test ...
%DEV_TOOL_ETPU_EXE% -p=AutoStructDemo.FullSysIdeProj -AutoBuild -AutoRun
if  %ERRORLEVEL% NEQ 0 ( goto errors )
goto end
:errors
echo ************************************************
echo        YIKES, WE GOT ERRORS!!
echo ************************************************
exit /b 1
:end
```

Note that the ETEC compiler has a similar mechanism for finding the compiler/assembler/linker, etc., executables in their (uniquely named by version) installation directory. See below for an example.

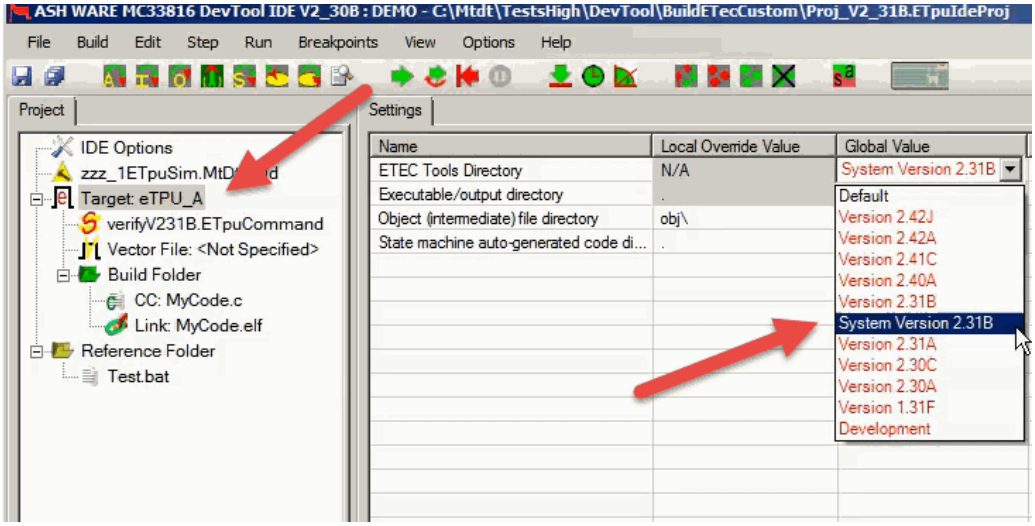
```
set CC=%ETEC_BIN%"\ETEC_cc.exe"
%CC% -version
%CC% AutoStruct.c
```

3.2 Using Non-Installed ETEC Versions

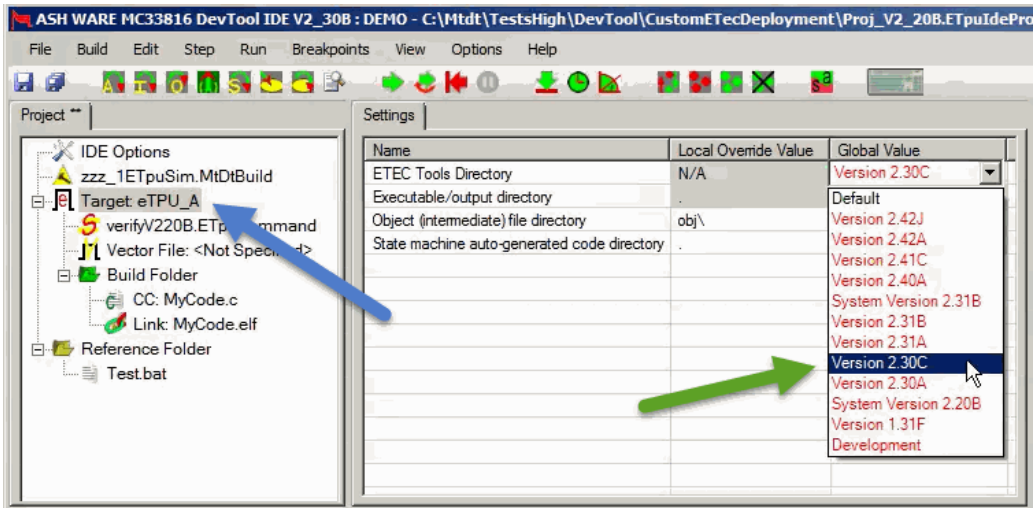
Certain customers may desire to deploy the ETEC Compiler Toolset on multiple users computers (often in conjunction with the network license capability) and in this type of situation it can be onerous to install ETEC on every computer. The following describes the process for accessing ETEC from the eTPU Development Tool when ETEC has not actually been installed.

1. Copy the ETEC directory on to the users machine at the same location on every machine. For example, to c:\Tools\eTPU Compiler V2_31B\.

2. Set the System Environment Variable as shown below. The Variable Name's format must follow this pattern: COMPILER_ASH_<VersionHigh><VersionLow><BuildLetter>.



3. That's it except you might need to reboot the computer before the environment variable takes effect. Open the settings window and then click on a target as shown below, In the options for the ETEC compilers you will now see the ETEC version as an option but with the word 'System' prepended as shown below.



4

IDE and Editors

This section covers some of the basic features of the eTPU Development Tool and editor windows.

4.1 IDE Options

The IDE Options are accessed through the Project window - right-click on the "IDE Options" tree node to open the settings. Settings allow control over how the editor handles tabs and spaces, the font type and size, and several global simulation options.

4.2 Panel Layout Options

The eTPU Development Tool window framework is highly configurable. By default there are 5 panels - three side-by-side above a horizontal divider, and two side-by-side below. All dividers can be adjusted as needed. Right-clicking in the tab area of these panels gives users the options of dividing the panel vertically or horizontally into sub-panels - this can be done to any depth desired. Right-clicking in the tab area of a child panel provides the option of closing the panel. The default 5 panels cannot be closed.

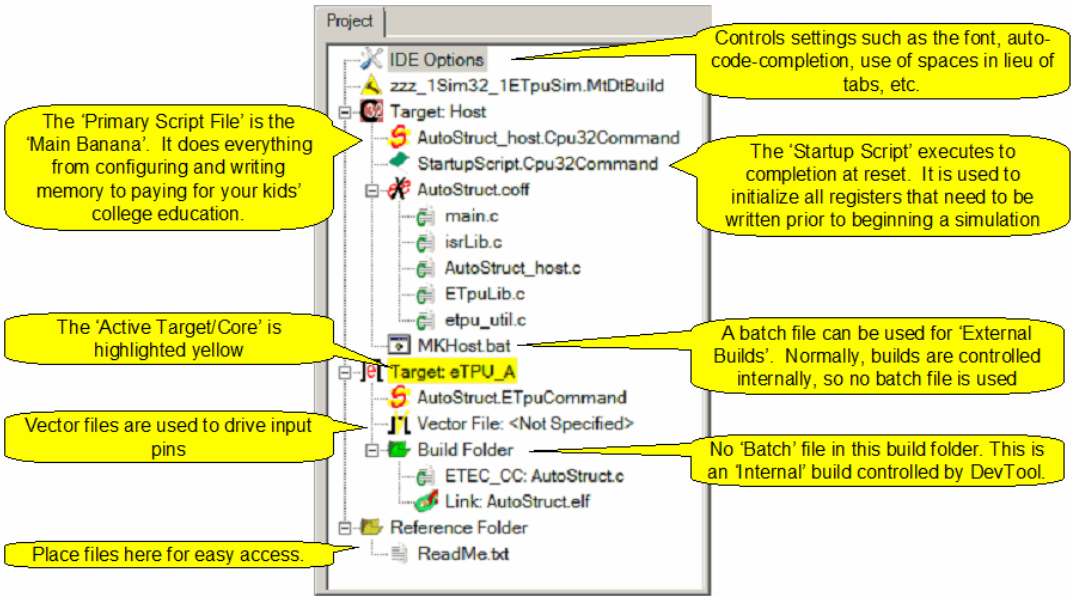
In addition to the panel layout controls, individual windows (non-editor windows only) can be "floated". This is accomplished by right-clicking on the desired window and selecting "Float This Tab". Doing so moves the tab into its own modeless dialog, which can be

minimized or moved wherever desired. This is a particularly powerful feature for users with multiple displays. Tabs that have been floated can be re-docked with right-clicking on the window and selecting "Dock This Tab".

5

The Project

The project is the glue that holds the eTPU Development Tool together.



5.1 Ide Settings

To modify the IDE settings, right-click on the 'IDE Options project node and select 'Settings'. The following will appear. From this settings window an number of settings can be modified such as the user of tabs (or spaces) the font, enabling of auto-completions, and various other options.

Settings

Name	Local Override Value	Global Value
Editor's Use of Tabs	N/A	Use spaces in place of tabs
Spaces per Tab	N/A	4
Font for Editor, Output, Sim View Windows	N/A	Courier New, Regular, 10
Auto-Complete Enable	N/A	Enable code/script auto-complete
View code coverage in source code files	N/A	Show Coverage
List the 'Build' Options in the Output Window	N/A	Don't list Build Options
List the 'Build Time' in the Output Window	N/A	Don't list the Build Time
On Reset, Initialize Global Variables per 'Elf' file	<Use Global Value>	No, don't initialize
On Reset, Initialize Data Memory	<Use Global Value>	Set to Zero
On Reset, Initialize ALU Registers (eTPU Only)	<Use Global Value>	Set to Zero
On Reset, Initialize TCR1 and TCR2 Counters (eTPU Only)	<Use Global Value>	Set to Zero

5.2 The Project Files

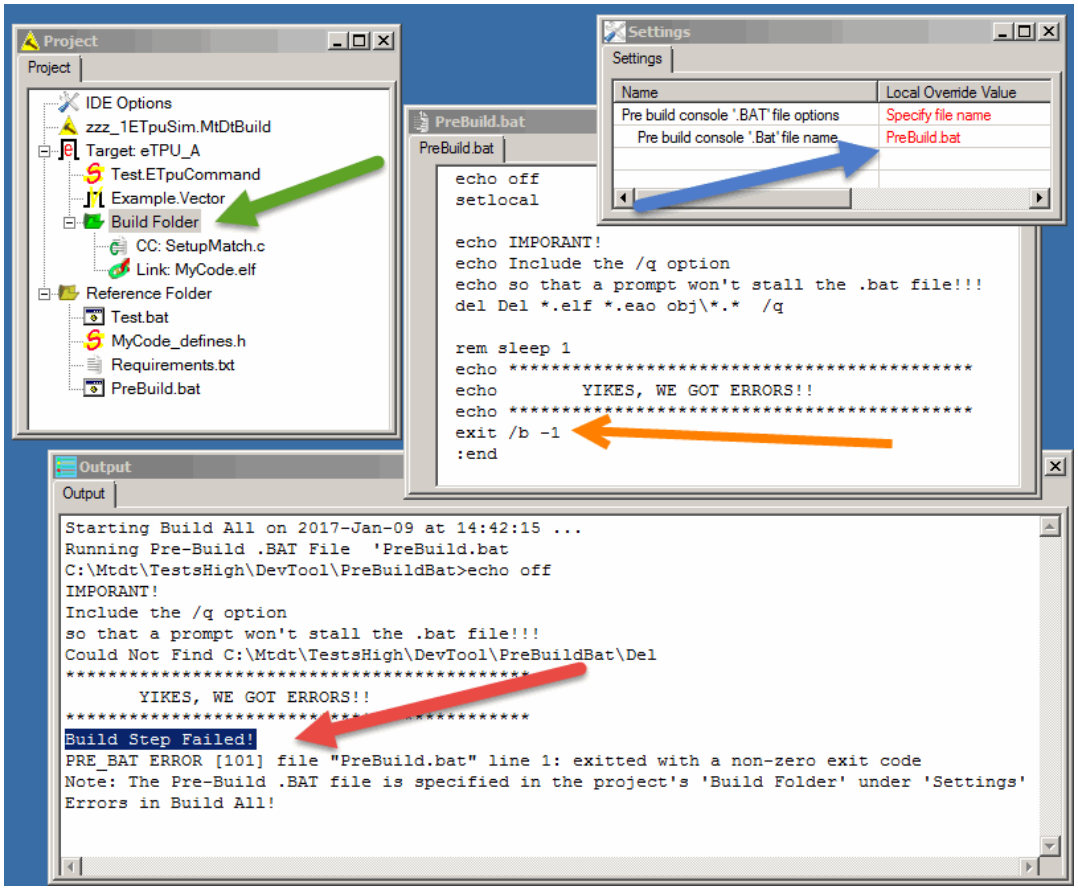
The project settings are saved in two files; the 'Project File' and he 'Environment File'.

The 'Project File' contains the key settings that are generally put under a versioning system such as CVS. It contains key filenames such as the primary script filename, source code filenames (if the build is controlled by the eTPU Development Tool) build options such as the memory model, etc.

The 'Environment file' contains settings that are generally specific to the user such as window sizes and positions, the font, etc. Most users will NOT keep this file in a versioning systems such as CVS. The environment file has the same name as the project file except that the user-name is appended to the base file name and has a different file suffix.

5.3 The Pre-Build Windows' Console '.BAT' File

A pre-build Windows' console '.BAT' file can be specified from within the project window by right-clicking on the Build Folder (pointed to in the picture seen below by the green arrow) and selecting 'Settings. The Settings windows then becomes visible and the '.BAT' file can be entered as seen by the blue arrow. Text which is echoed from the '.Bat' file is displayed in DevTool's Output Window as shown by the lower red arrow. Note that the '.BAT' file can return a pass by exiting with a '0' exit code thereby allowing the build to proceed. However in the example shown below, the '.BAT' file's exit code is '-1' thereby preventing the build or make from proceeding.



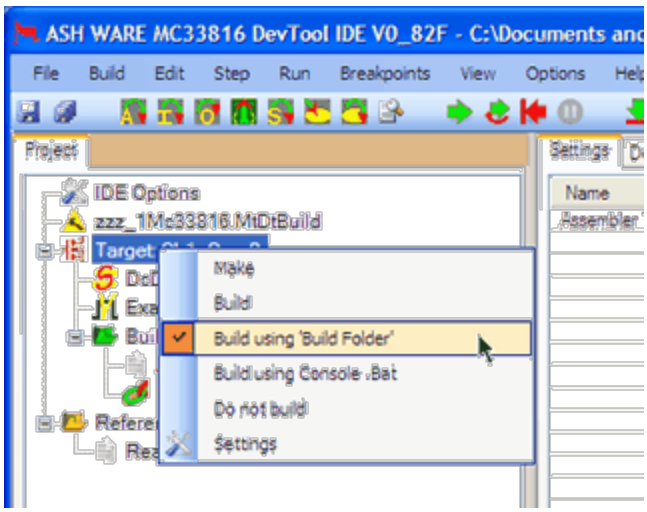
6

Integrated Build

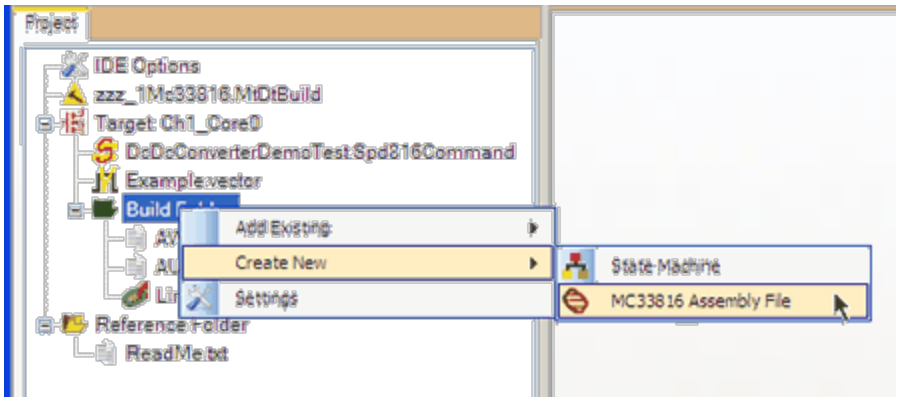
The Development Tool supports an internal and an external integrated build.

6.1 Internal Build

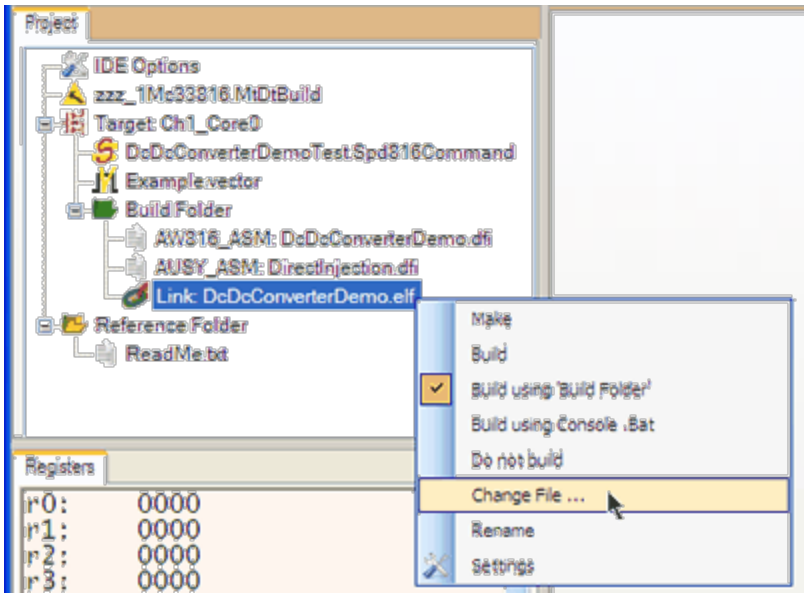
An 'Internal Build' is controlled by the eTPU Development Tool. A target is set to perform an 'Internal Build' within the Project by right-clicking the target's on the target node.



The source code files to be compiled or assembled are added to the target's 'Build Folder' as shown below.



The executable image filename is set in the target's 'Link' node, as shown below.



6.1.1 Host Target Build

The "generic" host target available in System DevTool now supports internal builds of C code. Startup code and a few basic libraries are automatically linked. The user code must contain an entry point function called "user_main" that has a return type of "int" and takes no parameters.

Users can enable interrupt handling from other targets (e.g. eTPU) during system simulation by taking the following steps:

- run "isrLibInit()" at the start of the entry function, and call "isrEnableAllInterrupts()" to enable interrupt handling. Then use "isrConnect" to set up handlers for the expected (eTPU) interrupts of the application. See isrLib.h in the "Include" directory under the DevTool installation directory.

- from the script environment of the host target, run the "enable_target_interrupts()" script command for each target from which to enable interrupts.

The host target code can also access the scripting environment through a provided API. See "scriptLib.h" in the "Include" directory for more details on what is available. The "at_time", "wait_time" and "read_time" calls can be particularly useful for synchronizing with .

6.2 External Build

Normally, a software build is done using an 'Internal Build.' However, there are occasions when it is preferable to call an external batch file (.BAT) to perform the build process. For instance, this is the way to build software using the Byte Craft ETPU_C compiler.

To specify an external build, right-click on the project's Target node and select 'Build using Console Bat.'

One significant issue is retrieving the exit code from the batch file. The batch file must be terminated using the following console batch command

```
exit <ID>
```

The 'ID' indicates to the eTPU Development Tool if there was an error or not. **If the build was successful then the ID should be zero, otherwise the ID should be non-zero**.

One common mistake is to terminate just the batch command using the /b option.

```
exit /b <ID>  DO NOT DO THIS!!!
```

So this can be a problem because if the batch file is also used as a part of a regression test suite then the /b might be needed to prevent the console window from closing. To address this conundrum, the eTPU Development Tool passes the console batch file the following parameter.

```
EXIT_CMD_PROCESSOR
```

This can be used to differentiate between a eTPU Development Tool external build and a regression test suite, as follows.

```
set EXIT_CMD=%1
if not defined EXIT_CMD goto do_exit_batch
if %EXIT_CMD% EQU EXIT_CMD_PROCESSOR  exit -1

:do_exit_batch
exit /b -1
:end
```

It is also a good idea to delete the executable image file before doing the build. This prevents a stale executable image file from being deleted and improves the eTPU Development Tool's ability to detect external build errors.

```
del HiLo.cod
%ETPUC_EXE%  HiLo.c  +l +e +q n=..\..\DemosMpc55xx\Lib
```

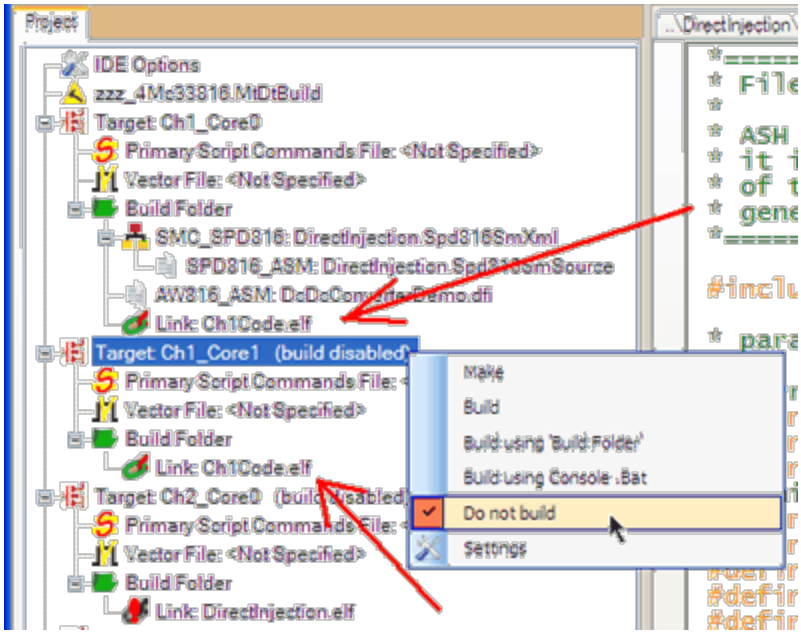
Here is a complete listing of a build batch file that can be used in the eTPU Development Tool's external build and as part of a separate regression test.

```
echo off
setlocal

if exist %ETPUC_EXE% goto DoneCheckBC_EXE
echo .
echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!
echo !!!  Set the environment variable, ETPUC_EXE, to the
eTPU_C Compiler's location !!!
echo !!!  Such as:
                          !!!
echo !!!            SET ETPUC_EXE="c:\program files\byte
craft\etpuc\etpu_c.exe"     !!!
echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!
echo .
goto :errors
```

```
:DoneCheckBC_EXE

if not exist  HiLo.cod  goto no_pre_delete
del HiLo.cod
:no_pre_delete

%ETPUC_EXE%  HiLo.c  +l +e +q n=..\..\DemosMpc55xx\Lib
if  %ERRORLEVEL% NEQ 0 ( goto errors )

echo COMPILATION PASSES
goto end

:errors
type HiLo.err
echo ************************************************
echo        YIKES, WE GOT ERRORS!!
echo ************************************************

set EXIT_CMD=%1
if not defined EXIT_CMD goto do_exit_batch
if %EXIT_CMD% EQU EXIT_CMD_PROCESSOR  exit -1

:do_exit_batch
exit /b -1
:end
```

6.3 Disabled Build

It is often desirable to disable a build such that the eTPU Development Tool does not rebuild the executable image file. This is particularly helpful when two cores share a code memory such that a single executable code image file is used by both cores.


Project
IDE Options
zzz_4Mc33816MtDtBuild
Target: Ch1_Core0
Primary Script Commands File: <Not Specified>
Vector File: <Not Specified>
Build Folder
SMC_SPD816: DirectInjection.Spd816SmXml
SPD816_ASM: DirectInjection.Spd816SmSource
AW816_ASM: DcDcConverter.Demo.dfi
Link: Ch1Code.elf
Target: Ch1_Core1 (build disabled)
Primary Script Commands File:
Vector File: <Not Specified>
Build Folder
Link: Ch1Code.elf
Target: Ch2_Core0 (build disabled)
Primary Script Commands File:
Vector File: <Not Specified>
Build Folder
Link: DirectInjection.elf
Make
Build
Build using 'Build Folder'
Build using Console .Bat
Do not build
Settings
..\DirectInjection\

7

Source Code Files

Source code files is built using code build tools. For the eTPU the ASH WARE ETEC compiler or the Byte Craft compiler is used to build the code.

These executable files are loaded into the target/core's memory space. The eTPU Development Tool 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.

Source code files can be edited directly within the IDE. Once edited, the source code is considered 'dirty' and will be rebuilt prior to resuming execution. The process of rebuilding and loading of the executable image is done automatically and largely transparently to the user.

Editing of source code files outside the IDE is also supported. When file is saved by the external editor, eTPU Development Tool automatically recognizes that the file is 'dirty' and will both reload the file and rebuild the code prior to resuming execution.

7.1 Source Code Search Rules

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

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

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

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

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

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

Absolute and Relative Paths

The eTPU Development Tool 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 eTPU Development Tool 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\
```

8

Script Commands Files

Overview

Script commands files provide a number of important capabilities. Script commands files provide a mechanism whereby actions available within the GUI can be automated. In the eTPU Simulator, Script commands can be used in place of the host CPU.

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

- Enhanced Scripting capabilities (variables, loops, etc)

Types of Script Commands Files

The different types of script commands files are as follows.

- The Primary Script Commands Files
- ISR Script Commands Files

It is best to just have a single primary script command associated with just one target/core. Alternatively, in a multi-target/core simulation, each target/core can have it's own primary script commands file. 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. The eTPU Development Tool can have only a single active MtDt build batch file.

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.

8.1 The Primary Script Command Files

The eTPU Development Tool automatically executes a primary script commands file if one is open. A new or alternate script commands file must be opened before it is available to the eTPU Development Tool for execution. The script command file can be changed within the project window. Right click on the script commands file node and select 'Change Script Commands File. The eTPU Development Tool 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.

8.2 ISR Script Commands Files

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

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

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

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

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

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

#define _ASH_WARE_<TargetName>_ISR_ X

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

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

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

Critical Change in Version 3.70 and Later!

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

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

Is:

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

Was:

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

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

8.3 The "ETEC_cpp.exe" Preprocessor

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

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

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

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

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

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

```
#define __etpu_globalinit32(address, value) \
        write_global_data32(address, value);
```

8.4 Enhanced Scripting Capabilities

The ASH WARE scripting syntax has always been based upon the C language, and enhanced scripting continues that tradition. In some cases it may be possible to share the same code between the scripting environment and host code. Enhanced scripting support includes

- ability to declare enumerated types and use enumeration literals in place of constants in expressions

- variables (called "script variables") can be declared, assigned to, used in expressions and used as inputs to script commands in places wherein previously a numeric constant was required.

- most C statements are supported: if-else, switch, for loops, while and do-while loops, and goto/labels.

More information on each of the above is found in the ensuing sections. A script file is treated like the contents of a C function from a scoping perspective, with the exception that variables defined in the outermost scope are treated as "global" and can thus be seen from ISR scripts. Inner scopes can be created with pairs of '{' and '}' tokens.

8.4.1 Enumeration Declarations

C Enumeration declarations are supported in the scripting language. The enumeration literals may be used in place of constants in expressions or as numerical arguments to script commands. Enumeration literals are treated as S32 type - signed 32-bit integers. Below is an example of supported syntax.

```
enum I2C_SLAVE_MODE
{
     I2C_SLAVE_MODE_FIND_IDLE,
     I2C_SLAVE_MODE_IDLE,
     I2C_SLAVE_MODE_START_SDA_LOW,
     I2C_SLAVE_MODE_WRITE_HEADER,
     I2C_SLAVE_MODE_WRITE_BYTE,
     I2C_SLAVE_MODE_WRITE_BYTE_CHECK_STOP,
     I2C_SLAVE_MODE_READ_BYTE,
     I2C_SLAVE_MODE_READ_FIND_STOP,
     I2C_SLAVE_MODE_ACK_OUT,
     I2C_SLAVE_MODE_ACK_IN,
     I2C_SLAVE_MODE_ACK_COMPLETE = I2C_SLAVE_MODE_ACK_IN +
1,
     I2C_SLAVE_MODE_IGNORE = -1,
};
```

If the script file has been parsed, when hovering the mouse over an enumeration literal its value will be displayed in the tooltip. Currently declaration of script variables of enumerated type is not supported.

8.4.2 Script Variables

Use of script variables in the scripting environment allows for true closed-loop feedback control, allowing scripts to do essentially anything a host CPU or MCU would do. Script variables follow C scoping rules with a few small exceptions, (1) you cannot re-use the

same name in an inner scope, hiding the outer scope variable, and (2) variables defined at the outermost scope of the first target's primary script file are treated as "global" and can be referenced in ISR script files. Three basic variable types are supported - S32 (32-bit signed int), U32 (32-bit unsigned int), and F64 (64-bit floating point). Declaration lists and initializers are supported.

```
S32 a;
S32 b = -33, c, d = 0;
U32 e = 0xfebc, f = ENUM_LITERAL;
F64 g = 1.25e9, h = 0.0, i = 0x11;
```

Multi-dimensional arrays of the basic types are supported, including initializers.

```
S32 x[20];
U32 y[10][3] = { {1,2,3}, {4,5}, {6} };
F64 z[4][4][2] = { { {1.11, 2.22, } } };
```

Array element access is supported, but not incomplete de-referencing as pointers are not supported in the scripting environment.

```
x[5] = (S32)(y[c][d] * z[3][2][1]); // valid
y[3] = read_dim_length("z", 1); // error, y not fully de-
referenced
```

Script variables lifetime ends when their scope ends. Note that C99-style late declarations are allowed.

```
{
     // code...
     S32 myVar;
     myVar = read_chan_data_u24(5, 0x11);
     if (myVar == 0x34)
     {
           // do something...
     }
     // code...
}
myVar += 10; // ERROR!!
```

Script variable values can be viewed in the tooltip by hovering the mouse over the variable name, or all in-scope script variables can be viewed in the Script Variable window. Script variables can additionally be configured as discrete nodes to be viewed in the Waveform window.

8.4.3 Expression Statements

All C operators are supported except the following:

- array and pointer operators (unary *, unary &, [], ->, '.' struct member de-reference)

- expression comma operator

From an implicit type conversion precedence standpoint, type precedence is S32 -> U32 -> F64. Note that for historical reasons constant expressions are computed in U32.

```
while (1)
{
     F64 rpm;
     F64 tooth_period;
     U32 tooth_per_sync;
     tooth_period = read_chan_data_u24( CRANK_CHAN,
FS_ETPU_CRANK_OFFSET_LAST_TOOTH_PERIOD );
     tooth_per_sync = read_chan_data_u8 (CRANK_CHAN,
FS_ETPU_CRANK_OFFSET_TEETH_PER_SYNC );
     rpm = (1.0 / (tooth_period * tooth_per_sync /
TCR1_FREQ_HZ)) * 60;
           // done if RPM has gone above 2000
     if (rpm > 2000.0)
           break;
     wait_time(1);
}
```

Explicit typecasts are supported.

```
write_chan_data_u24( PWM_CHAN, PERIOD_OFFSET, (U24)(1333.33
* SOME_SCALAR));
write_chan_data_u24( CRANK_CHAN, WINDOW_RATIO_OFFSET,
(UFRACT24)0.45);
```

Along with the 3 basic script types, the S32 and U32 base types have several supported sub-types available just for cast:

S32 subtypes (result of all these casts are stored as an S32)

- S24
- S16
- S8
- SFRACT24
- SFRACT16
- SFRACT8

U32 subtypes (result of all these casts are stored as an U32)

- U24
- U16
- U8
- UFRACT24
- UFRACT16
- UFRACT8

8.4.4 Selection Statements

The C if-else syntax is fully supported by the eTPU Development Tool script parser/interpreter.

```
if (<expression>)
     c_statement
[else
     c_statement]
```

Where c_statement can be any single C statement or script command (but not declaration), or it can be a compound statement enclosed by scope '{' '}' tokens.

The C switch statement is also available in the script environment, along with the associated case and default statements. The syntax for these statements is:

```
switch (<expression>)
     c_statement

case <constant expression>:

default:
```

The only difference with the C spec is that case constant expressions with floating point are allowed.

8.4.5 Loop and Jump Statements

All types of C looping are supported by the enhanced scripting environment. For loops:

```
for ([expression];[expression];[expression])
c_statement
```

Where c_statement can be any single C statement or script command (but not declaration), or it can be a compound statement enclosed by scope '{' '}' tokens. Each of the [expression] terms in the for statement are optional. At the current time variable declarations are not allowed in the first term.

While and do-while loops:

```
while (<expression>)
c_statement
```

and

```
do
c_statement
while (<expression>) ;
```

In all of theses loop types the 'break' and 'continue' statements are fully supported.

C labels and the goto statement are supported by the scripting environment.

```
label:
// code...
goto label;
```

Since the script is like the contents of a C function, any label in the script is accessible to a goto statement.

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

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

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

8.5.1 Multiple-Target Scripts

In a multiple target environment it is generally best to have just a single script file in the first target/core found in the project.

Script commands executed in this target/core affect specifically that core. For example, the following script command will affect just this first target/core.

```
write_chan_hsrr(TEST_CHAN, 7);
```

So how can scripts control other targets/cores besides the (default) one with which the script file is associated? For a script to operate on a specific target/core the target name is prepended to the script as follows.

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

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

```
eTPU_B.write_chan_hsrr (LAST_CHAN, 1);
wait_time(10);
verify_global_data32(SLAVE_SIGNATURE_ADDR,
                     DATA_TRANSFER_INTR_SIGNATURE);
eTPU_B.verify_data_intr(LAST_CHAN, 1);
eTPU_A.verify_data_intr(LAST_CHAN, 0);
```

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

8.5.2 Script Directives, Define, Ifdef, Include

The #define directive

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

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

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

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

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

The #include <FileName.h> directive

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

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

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

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

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

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

The above directive is commonly found at the very end of a script commands file that is part of an automated test suite. It allows behavior dependent on the test conditions. Note that _ASH_WARE_AUTO_RUN_ is automatically defined when the eTPU Development Tool is launched in such a way that it runs without user input. In this case, upon reaching the end of the script file the eTPU Development Tool is closed when it is part of an automated test suite and otherwise a message is issued to the user.

8.5.3 Script Enumerated Data Types

Many defined functions have arguments that require specific enumerated data as arguments. 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. The following is an example of an internally defined enumeration.

```
enum TARGET_TYPE    {
ETPU_SIM,
};
```

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.

8.5.4 Script Integer Data Types

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

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

- **U1** valid range is 0 to 1
- **U2** valid range is 0 to 3

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

8.5.5 Referencing Memory in Script Files

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

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

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

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

8.5.6 Assignments in Script Commands Files - DEPRECATED

Note: it is now possible to declare and manipulate variables in script commands. Therefore use of variables such as the following should now be used instead of those listed farther down in this section.

```
S32 val;
val = read_mem_u32(ETPU_DATA_SPACE,0x200);
val |= ((1<<12) | (1<<28));
```

```
write_global_data32(0x200, val);
```

DEPRECATED

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

The following examples perform assignments on memory.

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

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

```
// Sets the eTPU I/O pins for channel 15 and channel 3
*((ETPU_PINS_SPACE U32 *) 0x0) |= ((1<<15) + (1<<3));
// Injects a new opcode into the eTPU's code space
*((ETPU_CODE_SPACE U32 *) 0x20) = 0x12345678;
```

NOTE: Because the use of the ADDR_SPACE Enumerated Data Type as in the above expressions is not legal C syntax, the use of such in a script commands file disables the code reference capability. It is recommended that the write_global_data[8/16/24/32]() and write_global_bits[8/16/24/32]() script commands be used instead whenever possible.

8.5.7 Operators and expressions in Script Commands Files

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

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

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

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

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

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

8.5.8 Syntax for global access of eTPU Function Variables

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

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

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

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

8.5.9 Syntax for eTPU Channel Hardware Access

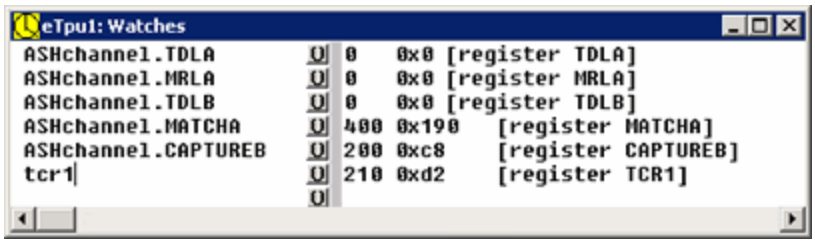
eTPU-C provides a built-in type called chan_struct that allows access to such channel-specific settings as IPAC, OPAC, PDCM, TBS, etc. This symbolic access to channel settings has now been exposed in the eTPU Simulator. Typically, a chan_struct variable is defined in a standard header file (etpuc.h),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

```
chan_struct channel;
```

This variable channel can then be used to access channel settings in the watch window, or script command such as print_to_trace() and verify_val(), with the syntax:

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

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



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

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

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

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

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

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

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

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

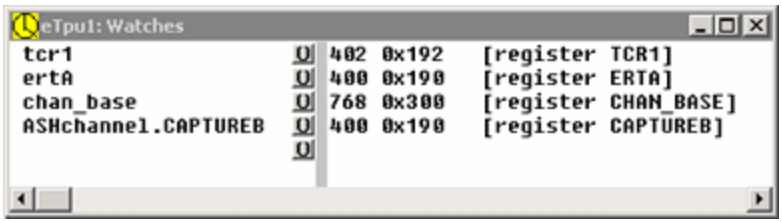
```
IPACA
IPACB
LSR
MRLA
MRLB
MTD
OPACA
OPACB
PDCM
TBSA
TBSB
FLAG0
FLAG1
FM0
FM1
PRSS
PSS
FM0_CHAN
FM1_CHAN
PRSS_CHAN
PSS_CHAN
PSTI
PSTO
TDLA
TDLB
MATCHA
MATCHB
CAPTUREA
CAPTUREB
```

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

when the chan register is written. To access the per-channel values for these fields, use the corresponding <>_CHAN field names. Note that PRSS_CHAN and PSS_CHAN reference the last sampled values for a specified channel.

8.5.10 Syntax for eTPU ALU Register Access

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



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

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

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

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

```
register_diob diob;
register_mach mach ;
register_p15_0 p15_0; // lower 16 bits of p register
```

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

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

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

In addition, the ALU and MDU condition codes are exposed to the symbolic script commands via the CC symbol:

```
struct {
     unsigned int V : 1; // ALU overflow condition code
     unsigned int N : 1; // ALU negative condition code
     unsigned int C : 1; // ALU carry condition code
     unsigned int Z : 1; // ALU zero condition code
     unsigned int MV : 1; // MDU overflow condition code
     unsigned int MN : 1; // MDU negative condition code
     unsigned int MC : 1; // MDU carry condition code
     unsigned int MZ : 1; // MDU zero condition code
     unsigned int MB : 1; // MDU busy flag
     unsigned int SMLCK : 1; // semaphore locked flag
} CC;
```

This allows for script access such as the examples below:

```
print_to_trace("CC.N = %d", CC.N);
verify_val("CC.N", "==", "1");
verify_val("CC.Z", "==", "0");
verify_val("CC.SMLCK", "==", "0");
print_to_trace("CC.MZ = %d", CC.MZ);
verify_val("CC.MN", "==", "0");
verify_val("CC.MZ", "==", "1");
```

8.5.11 String within a string supports formatted symbolic information

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

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

```
"      C=%d\",A,B,C");
```

8.5.12 Comments in Script Commands Files

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

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

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

8.5.13 Decimal, Hexadecimal, and Floating Point Notation in Script Files

Decimal and Hexadecimal notation are interchangeable.

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

In certain cases floating point notation is also supported.

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

8.5.14 String Notation

The following is the accepted string notation.

```
"STRING"
```

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

```
"File.dat"
```

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

Concatenation

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

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

C-Style Escape Sequences

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"
```

8.6 Script Commands

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

eTPU 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
Clear Worst Case Thread Indices Commands
System configuration commands
Timing configuration commands
STAC Bus commands
Global Data Commands
Channel data commands
Channel base address commands
Engine data commands
Channel function mode (FM) commands
Channel event vector entry commands
Interrupt script commands
Shared Subsystem Script Commands
Link Injection Script Command

Build Script Commands

Build script commands

8.6.1 Timing

Script commands in this group provide capabilities such as the ability to set the system frequency, control when (other) script commands execute, clear the worst case threads (used following the init threads,) and to find worst case information for a time region.

8.6.1.1 Timing Script Commands

```
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);
wait_time(100.0);
set_link(2);
```

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.**at_code_tag(TagString);**

```
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.

The eTPU Development Tool's enhanced scripting provides a script command to read the current time in microseconds into a script variable.

```
F64 time_us;
time_us = read_time();
```

See the Enhanced Scripting Capabilities section for more information on the use of variables within the scripting environment.

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

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

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

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

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

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

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

8.6.1.2 Verify Timing Script Commands

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

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

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

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

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

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

In the following example, the last full traversal of the above code is verified to have taken between 10 and 20 system clocks. A verification error occurs if the code has never been fully traversed, if the last traversal took 9 or fewer system clocks, or if the last traversal took 21 or more system clocks.

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

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

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

Related Information

Naming timing regions in source code
Verifying traversal times a script command file
View named timing regions timing using the Watch Window

8.6.1.3 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.

8.6.1.4 Thread Script Commands

The eTPU Development Tool 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 provides for 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.

Another script command can be used to verify that a particular meets performance requirements.

```
verify_wctl(<thread name>, <max steps>, <max RAM
accesses>);
```

If the thread is part of an ETEC class, the name should be specified in Class::ThreadName form. This script command will through a script verification error if the worst case steps for the specified thread exceeds the number provided, or worst case RAM accesses exceeds the number passed in the script command.

To output WCTL data to a file, in an easy-to-read format, the following command can be used:

```
write_wctl(<output file name>, <channel or function
number>, <channel for function mode>, <grouping on or off>,
<output style>);
```

So, for example, if the following is placed at the end of the Hello World demo script file:

```
write_wctl("pwm_wctl.txt", PWM_CHAN, WCTL_TYPE_CHANNEL,
WCTL_GROUP_ON, WCTL_OUTPUT_PRETTY);
```

The following text gets output to the pwm_wctl.txt file when the simulation is executed:

```
  |----- THREAD -----|-- STEPS -| RAMs |
  | Addr  Cov    Cnt | WC   Tot |  WC  | WC Time
0   0858  1/1      1    9     9     5    180 nS  Init
1   087C  1/4    133   11  1463     5    220 nS
HandleFallingEdge
2   08AC  0/27     0    -     0     -       ---  Unused
```

Note that `clear_worst_case_threads()` clears the thread data collection that is output by `write_wctl()` or verifed with `verify_wctl()`.

8.6.2 MCU Configuration

These include a host of MCU-specific configuration script commands such as setting the entry table base address (ETBA) in the eTPU or setting the DC load in the MC339816.

8.6.2.1 eTPU System Configuration Commands

```
write_entry_table_base_addr(Addr);
```

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

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

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

```
write_engine_relative_base_addr(Addr);
```

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

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

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

```
write_scheduler_priority_passing_disable(Val);
```

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

```
write_scheduler_priority_passing_disable(1);
```

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

```
write_global_time_base_enable(Enable);
```

The command enables the time bases for all the eTPUs.

```
write_entry_table_base_addr(Addr);
```

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

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

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

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

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

```
write_scm_off_data( Val );
```

8.6.2.2 eTPU Timing Configuration Commands

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

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

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

IMPORTANT NOTE: These two script commands write actual value of the 'system clock divider' **NOT THE VALUE OF THE TCR1P or the TCR2P registers!!!** So, for example, the script 'write_tcr1_prescaler(6);' gives a divide by of '6' but the TCR1P register gets written to a '5'!

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 (default TCR1 clock
source)
write_tcr2_control(2);        // TCR2’s clk is falling TCRCLK Pin
write_tcr1_prescaler(1);      // Fastest ... divide by 1
write_tcr2_prescaler(64);     // Slowest ... divide by 64
```

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

```
write_tcr1_control(2);        // use TCR1 clock source
write_tcr1_source(1);         // System Clock/1 (eTPU2 Only!)
write_tcr1_prescaler(1);      // Fastest ... divide by 1
```

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

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

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

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

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

8.6.2.3 eTPU 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.

8.6.2.4 eTPU Channel Address Script Commands

```
write_chan_base_addr(ChanNum, Addr);
```

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

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

In this example channel 3's data will start at address 0x300. Note channel variables and static local variables use this.

8.6.2.5 eTPU Channel Function Select Register Commands

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

This command sets channel CHANNUM to function Val.

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

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

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

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

8.6.2.6 eTPU Event Vector Entry Condition (Standard/Alternate) Commands

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

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

```
#define UART_STANDARD_ENTRY_VAL    0
#define PWM_ALTERNATE_ENTRY_VAL    1

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

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

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

The ETEC compiler automatically outputs entry table type information into the auto-defines file.

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 commandsSYSTEM_CFG_CMDS section.

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

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

```
#define ETPD_PIN_DIRECTION_INPUT      0
#define ETPD_PIN_DIRECTION_OUTPUT     1

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

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

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

8.6.2.7 eTPU Channel Function Mode Script Command

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

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

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

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

8.6.2.8 eTPU Channel Priority Register Commands

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

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

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

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

8.6.2.9 eTPU Shared Subsystem Script Commands

```
config_shared_sub_system(subSystemId, engineId, chanNum);
```

This command configures the subsystem 'subSystemId' to generated a link on engine 'engineId' channel 'ChanNum'. SubSystemId is the ID for the sub-system. The parameter 'engineId' programs the ETPUSSSnIR register's 'LENQ' field and is set to 'DISABLED', 'ENGINE_1', or 'ENGINE_2'. The parameter chanNum programs the ETPUSSSnIR register's 'LCHAN' field.

```
issue_sub_system_link(subSystemId);
```

This command issues a link that originates from the sub-system specified by the subSystemId field.

```
config_shared_sub_system(0, ENGINE_1, 17);
issue_sub_system_link(0);
```

The above command sequence configures the shared memory subsystem to interrupt channel 17 of engine 1 and then issues a link.

```
write_chan_shared_subsystem_access_enable(chanNum,
enableVal); // (eTPU2 only)
```

This command enables or disables shared subsystem accesses. This is done on a per channel bases in that some channels can have this enabled while others have it disabled. The 'chanNum' specifies the channel to be enabled or disabled. The 'enableVal' can either be '1' (enabled) or '0' (disabled.) Note that this command is only available for eTPU2.

```
write_chan_shared_subsystem_access_enable(4, 1);
```

In this example, channel 4 is enabled for memory accesses to the shared subsystem.

8.6.2.10 eTPU STAC Bus Script Commands

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

```
write_stac_tcr1_enable();
write_stac_tcr1_assignment(Mode);
write_stac_tcr1_server(ServerID);
```

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

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

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

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

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

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

8.6.2.11 eTPU Link Script Command

```
set_link(ChanNum);
clear_link(ChanNum);
verify_link(ChanNum, Val);
```

This 'set_link' script command directly sets a Link Service Request (LSR) to channel ChanNum and the 'clear_link' clears the link. The verify_link() is used to verify that a link is either set or cleared (Val is 0 ir 1).

```
set_link(8);
wait_time(3);
verify_link(8, 1);
clear_link(8);
verify_link(8, 0);
```

In this example a link is generated on channel 8.

> Important note. There is no corresponding capability in the actual host interface relating to the 'set_link' and 'clear_link' script commands. These commands are entirely a figment of the rather distorted imagination of the simulation designer. As such there is no related capability in the host interface, so please use these script commands with caution.

8.6.2.12 eTPU Interrupt Script Commands

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

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

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

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

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

```
clear_global_exception();
```

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

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

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

```
clear_this_intr();
```

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

8.6.2.13 eTPU Interrupt Association Script Commands

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

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

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

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

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

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

8.6.3 Variable, Memory, and Register Modification and Verification

These are the 'main banana' script commands as far as data flow goes. Values in variables, (more generic) memory, registers, etc., can be both modified and verified.

8.6.3.1 Memory Read Script Commands

Read memory script commands provide the mechanism for reading data from the target memory into the scripting environment (variables). The required syntax is to assign a script variable to the return value of the script command. The first argument of the script command is an address space-enumerated type. The second argument is the address from which the value should be read.

```
x = read_mem_u8(enum ADDR_SPACE, U32 address);
x = read_mem_u16(enum ADDR_SPACE, U32 address);
x = read_mem_u24(enum ADDR_SPACE, U32 address);
x = read_mem_u32(enum ADDR_SPACE, U32 address);
```

```
U32 x;
x = read_mem_u8(ETPU_DATA_SPACE, 0x7);
```

In the above script command example, the byte found at address 0x7is read into script variable 'x'.

```
S32 y;
y = read_mem_u24(ETPU_DATA_SPACE, 0x101);
```

In the above script command example, a 24-bit (three-byte) memory at address 101h is read into 'y'. Note that the 24-bit value is signed-extended when it is converted to S32.

Note: Under the hood the variable to which the read value is assigned is actually treated as the first argument. This means a statement like this:

```
x = read_mem_u16(ETPU_DATA_SPACE, _AW816DA_IMM_MinCurrent_);
```

Could also be written like below.

```
read_mem_u16("x", ETPU_DATA_SPACE, _AW816DA_IMM_MinCurrent_);
```

Related Topics

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

8.6.3.2 Memory Modify Script Commands

Modify memory script command provide a way to read-modify-write any data in any target memory. The first argument is an address space-enumerated type. The second argument is the address at which the value should be written or read-modified-written. The third argument is an assignment operation type. The fourth argument is the value to be used for the assignment operation.

```
modify_mem_u8(enum ADDR_SPACE, U32 address, enum ASSIGNMENT_TYPE op,
U8 val);
modify_mem_u16(enum ADDR_SPACE, U32 address, enum ASSIGNMENT_TYPE
op, U16 val);
modify_mem_u24(enum ADDR_SPACE, U32 address, enum ASSIGNMENT_TYPE
op, U24 val);
modify_mem_u32(enum ADDR_SPACE, U32 address, enum ASSIGNMENT_TYPE
op, U32 val);
```

Memory can also be modified within script commands using the assignment operator. See the Assignments in Script Commands Files section for a description. Note that since the assignment syntax is not actually C-compatible, it is recommended that memory modification script commands be used instead, as the non-C assignment syntax causes script file code references not to be available.

8.6.3.3 Memory Verify Script Commands

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

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

This command uses the following algorithm.

- Read the memory location in the specified address space and address.
- Perform a logical "and" of the mask with the value that was read from memory.
- Compare the result to the expected value.

- If the expected value is not equal to the masked value, generate a verification error.

The following example verifies the value of an 8-bit (byte) memory location.

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

The example above verifies that the two most significant bits found at address 0x7 are equal to 10b. The lower 6 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(ETPU_DATA_SPACE, 0x10, 0xffff, 0x55aa);
```

In the example above, a 16-bit (two-byte) memory at address 10h is verified to equal 0x55aa. By using a mask of FFFFh, the entire word is verified.

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

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

```
verify_mem_u32(ETPU_DATA_SPACE, 0x20, 1<<27, 1<<27);
```

In the example above, bit 27 of a 32-bit (four-byte) memory location at address 20h is verified to be set. All other bits except bit 27 are ignored.

Related Topics

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

8.6.3.4 Register Write Script Commands

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

```
write_reg1(U1, enum REGISTERS_U1);
write_reg5(U5, enum REGISTERS_U5);
write_reg8(U8, enum REGISTERS_U8);
write_reg16(U16, enum REGISTERS_U16);
write_reg24(U24, enum REGISTERS_U24);
write_reg32(U32, enum REGISTERS_U32);


write_reg16(0x5557, REG_DIOB);
```

In the above script command example 5557 hexadecimal is written to register DIOB.

8.6.3.5 Register Verify Script Commands

Verify register script commands provide the mechanism for verifying the values of the target registers. The first argument is aeTPU 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_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_reg16(REG_DIOB, 0x5557);
```

In the script command example above, register DIOB is verified to be 5557 hexadecimal. If not, a script failure message is generated and the script failure count is incremented.

8.6.3.6 Symbol Write Script Commands

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

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

The expression string (exprString) can be a numerical constant or a simple symbolic expression. Constants can be supplied as decimal signed integers, unsigned hexadecimal numbers (prepended with ‘0x’), floating point numbers, or as a character (e.g. ‘A’). A symbolic expression can be just a local/global symbol or a simple expression such as *V (de-reference), &V (address of V), V[constant], V.member or V->member, where V is a symbol of the appropriate type. The special @ channel variable (eTPU only) reference syntax is also supported. "symbolExprString" must be a symbolic expression as described above, an "l-value" in compiler parlance. The type of the symbolic expression must be a basic type – char, int, float, etc. If the types of the two sides differ then C type conversion rules are followed before writing the data to memory.

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

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

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

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

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

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

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

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

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

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

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

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

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

8.6.3.7 Verify Symbol Value Script Commands

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

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

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

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

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

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

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

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

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

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

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

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

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

The extended version of this command, verify_str_ex(); also support a length specifier, len. Think strncmp where the comparison acts only on the first "len" characters and the remainder are ignored.

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

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

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

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

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

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

8.6.3.8 eTPU Engine Data Script Commands

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

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

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

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

The address is formed by adding the engines base address (see ERBA.) with the address formed in the by the Addroffset field. See the System Configuration Commands section

for information on how the Engine Relative Base Address (ECR.ERBA) field is writtenwrite_engine_data32 (0x20, 0xC6E2024A);

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

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

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

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

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

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

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

```
// set _Bool at LSB of 8-bit unit to 1
write_engine_bits8( 0x10, 7, 1, 1 );
```

8.6.3.9 eTPU Channel Data Script Commands

```
write_chan_data32(ChanNum, AddrOffset, Val);
write_chan_data24(ChanNum, AddrOffset, Val);
write_chan_data16(ChanNum, AddrOffset, Val);
write_chan_data8 (ChanNum, AddrOffset, Val);
```

```
verify_chan_data32(ChanNum, AddrOffset, Val);
verify_chan_data24(ChanNum, AddrOffset, Val);
verify_chan_data16(ChanNum, AddrOffset, Val);
verify_chan_data8 (ChanNum, AddrOffset, Val);
```

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

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

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

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

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

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

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

```
write_chan_bits32(ChanNum, AddrOffset, BitOffsetFromMSB, BitSize, Val);
write_chan_bits24(ChanNum, AddrOffset, BitOffsetFromMSB, BitSize, Val);
write_chan_bits16(ChanNum, AddrOffset, BitOffsetFromMSB, BitSize, Val);
write_chan_bits8 (ChanNum, AddrOffset, BitOffsetFromMSB, BitSize, Val);
verify_chan_bits32(ChanNum, AddrOffset, BitOffsetFromMSB, BitSize, Val);
verify_chan_bits24(ChanNum, AddrOffset, BitOffsetFromMSB, BitSize, Val);
verify_chan_bits16(ChanNum, AddrOffset, BitOffsetFromMSB, BitSize, Val);
verify_chan_bits8 (ChanNum, AddrOffset, BitOffsetFromMSB, BitSize, Val);
```

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

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

With the eTPU Development Tool's enhanced scripting capabilities, it is possible to read from channel frame memory into script variables, as a shortcut from using the read_mem_u<>() script commands.

```
U32 x;
x = read_chan_data_u32(ChanNum, AddrOffset);
x = read_chan_data_u24(ChanNum, AddrOffset);
x = read_chan_data_u16(ChanNum, AddrOffset);
x = read_chan_data_u8(ChanNum, AddrOffset);
```

Depending upon the variable type, sign-extension is performed as necessary. The alternative form for reads is:

```
read_chan_data_u32("x", ChanNum, AddrOffset);
```

8.6.3.10 eTPU Global Data Write/Verify Commands

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

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

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

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

```
write_global_data32 (0x20, 0xC6E2024A );
verify_global_data32(0x20, 0xC6E2024A );
verify_global_data24(0x21,   0xE2024A );
verify_global_data16(0x20, 0xC6E2     );
verify_global_data16(0x22,     0x024A );
```

```
verify_global_data8 (0x20, 0xC6       );
verify_global_data8 (0x21,   0xE2     );
verify_global_data8 (0x22,     0x02   );
verify_global_data8 (0x23,       0x4A );
```

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

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

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

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

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

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

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

8.6.4 Pin and Node Modification and Verification

This section includes script commands for controlling, modifying, and verifying pins and nodes. For all devices, input pins are controlled and their values are verified. Verification can be over a region of time (e.g., a 3 micro-second pulse should occur somewhere between 100 and 200 microseconds) or at a specific snapshot in time (e.g., the pin must be high at exactly the time when the script executes.) For analog devices such as the MC33816, the various voltages and currents can be both controlled and verified.

8.6.4.1 Pin Window Verification Commands

These command have the purpose of verifying the state of the pin (or node) within a time-window. Within this time-window an entire pulse can be verified, a single transition can be verified, or the absence of any transitions at all can be verified. The commands in this section differ from the commands like, 'verify_chan_output_pin();' (documented in section Pin Verification and Control Script Commands) which verify the pin state state at a snapshot in time and is therefore subject to 'false pass' scenarios in which the pin pulses occur (and are not detected) between adjacent snapshots. These script commands described in this section are not subject to this limitation because the pin state is verified over an entire time-windw.

Script verify_pulse();

The verify_pulse(); script command verifies that exactly one pulse (two transitions) occurs on a pin (or node) within a time-window.

Parameter 'Node' specifies the Node or Pin type. See the Pin and Node Enumerated Type section for a listing of the pin and node names. For the eTPU, only the CHAN_OUTPUT node is valid.

Parameter 'Index' is the index of the Node. For example, in the eTPU, this might signify the channel number.

Parameters 'StartTime' and 'EndTime' specify the beginning and end of the time-window in which the pulse must occur.

Parameter 'PulseType' can be either LO_HI_LO (a positive pulse) or HI_LO_HI (a negative pulse.)

Parameters 'FirstTransitionTime' and 'SecondTransitionTime' specify the times of the two transitions in micro-seconds.

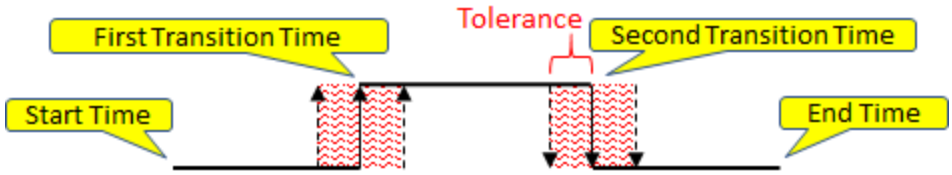
Parameter 'Index' is the index of the pin. For instance, in the eTPU there are 32-channels, so a Node of 'CHAN_OUTPUT' and an index of '4' refers the the eTPU's channel 4 output pin.

```
verify_pulse(Node, Index, StartTime, PulseType,
FirstTransitionTime, SecondTransitionTime, EndTime);
```

The default 'tolerance' is one system clock minus one nanosecond. This can be overridden with the 'set_verify_region_tolerance(); script command described below.

This script command verifies the following aspects of this waveform. Note that the script must execute after the pulse and as such verifies that the pin transitions that have already occurred.

- At the 'Start Time' the pin is at State1 (low in this diagram)
- The First Transition occurs within +/- the allowed tolerance
- The Second Transition occurs within +/- the allowed tolerance
- Exactly two transitions occur between the Start Time and the End Time



Script verify_transition();

The verify_transition(); script command verifies that exactly one transitions occurs on a pin (or node) within a time-window.

Parameter 'Node' specifies the Node or Pin type. See the Pin and Node Enumerated Type section for a listing of the pin and node names. For the eTPU, only the CHAN_OUTPUT node is valid.

Parameter 'Index' is the index of the Node. For example, in the eTPU, this might signify the channel number.

Parameters 'StartTime' and 'EndTime' specify the beginning and end of the time-window in which the transition must occur.

Parameter 'TransitionType' can be either RISING (a low to high transition) or FALLING (a high to low transition.)

Parameters 'TransitionTime' specify the time of the transition in micro-seconds.

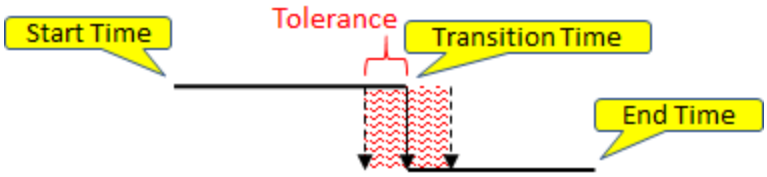
Parameter 'Index' is the index of the pin. For instance, in the eTPU there are 32-channels, so a Node of 'CHAN_OUTPUT' and an index of '4' refers the the eTPU's channel 4 output pin.

```
verify_transition(Node, Index, StartTime, PulseType,
TransitionTime, EndTime);
```

The default 'tolerance' is one system clock minus one nanosecond. This can be overridden with the 'set_verify_region_tolerance(); script command described below.

This script command verifies the following aspects of this waveform. Note that the script must execute after the transition and as such verifies that the pin transition have already occurred.

- At the 'Start Time' the pin is either low (if verifying a rising edge) or high (if specifying a falling edge.)
- The Transition occurs within +/- the allowed tolerance
- Exactly one transition occurs between the Start Time and the End Time



Script verify_pin_region();

The verify_pin_region(); enhanced script command verifies the pin (or node) over a time-window. The intent of the command is to verify that pin (or node) stays at a particular state (high or low) over an entire time-window, and specifically that no pin transitions at occur occur within the time-window. Note that there is no 'tolerance' band.

Parameter 'Node' specifies the Node or Pin type. See the Pin and Node Enumerated Type section for a listing of the pin and node names. For the eTPU, only the CHAN_OUTPUT node is valid.

Parameter 'Index' is the index of the Node. For example, in the eTPU, this might signify the channel number.

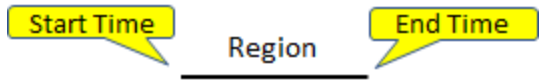
Parameters 'StartTime' and 'EndTime' specify the beginning and end of the time-window in which the pin must be at 'State' (HI or LO).

Parameter 'State', is the pin state in the time-window. Allowed values are HI and LO.

```
verify_pin_region(Node, Index, StartTime, State, EndTime);
```

Note that there is no implied tolerance. I suppose you could think of the tolerance as being zero in the same way as I think of myself as being a billionaire with zero billions of dollars.

- Between the Start Time and the Script Execution Time the pin is at State
- No transitions occur between the Start Time and the End Time



Script set_verify_region_tolerance();

```
set_verify_region_tolerance(MicroSecondsTolerance);
```

The set_verify_region_tolerance() script command sets the tolerance for the verify_pulse(); and verify_transition(); script commands to 'MicroSecondsTolerance'.

Note that only this tolerance only applies to the time of the transitions. It does not apply to either time-window boundaries (set by 'StartTime and EndTime'.) This does not apply to the 'verify_pin_region(); script command because it does not have a pin-transition to which this script command would apply.

Note that once set, this value applies for future script commands. However, if the simulator is reset, then the tolerance is restored to the default.

8.6.4.2 Pin Verification and Control Script Commands

Note that there are superior 'region verification' scripts defined in section "Pin Region Verification Commands" that you may want to use instead of the 'verify_chan_output_pin();' documented in this section. The 'verify_chan_output_pin();' described below verifies the pin state state at a specific snapshot in time and is therefore subject to 'false pass' scenarios in which the pin pulses occur (and are not detected) between adjacent snapshots. The pin region verification script commands described in the "Pin Region Verification Commands" section are not subject to this limitation because the pin state is verified over an entire region.

eTPU input pins are normally controlled using test vector files. eTPU output pins configured as outputs are normally controlled by the eTPU and are verified using master

behavior verification test files described in the Functional Verification section. Therefore, these commands are not the primary method for controlling and verifying pin states. Instead, these commands serve as a secondary capability for pin state control and verification.

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

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.

```
#define TEST_CHAN 4
write_chan_input_pin(TEST_CHAN, 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.

```
#define TEST_CHAN 4
verify_chan_output_pin(TEST_CHAN, 1);
verify_ouput_buffer_disabled(TEST_CHAN);
wait_time(23.4);
verify_chan_output_pin(TEST_CHAN, 0);
verify_ouput_buffer_enabled(TEST_CHAN);
```

In this example channel 4's output pin is verified to have a falling transition within a 23.4-micro-second window. It is also verifying that the pin is acting like an open-drain (active low, passive high.)

8.6.4.3 Pin Transition Behavior Script Commands

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

NEW : Enhanced behavior verification starting with release 5.00 provides much more flexible and useable pin transition verification capabilities. With enhanced behavior

verification pin transition data is saved in a new format that is .csv (comma separated value) format for compatibility with many other tools. However, the enhanced behavior verification can read original style .bv files as well as the new .ebv files. Existing scripting commands apply to original .bv files only where noted.

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

Enhanced Behavior Verification Script Commands

The test tolerance commands apply whether an old-style .bv file or new .ebv file is being used as the master, however, the file manipulation script commands only apply to .ebv files.

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

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

- only one can be created

- cannot have the same file name as a running (under verification) .ebv file

- must occur at simulation time 0 before any wait_time() or at_time() script commands

- must occur after any external gate instantiation or after any vector file commands

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

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

```
close_ebehavior_file();
```

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

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

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

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

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

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

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

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

Pins can be individually disabled from behavior verification with this script command. Note that this command and set_ebehavior_pin_tolerance work sequentially when determining the final set of pins of interest.

```
stop_ebehavior_file();
```

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

Deprecated Behavior Verification Script Commands

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

This command loads the pin transition behavior file into the master pin transition behavior buffer. This buffer forms a behavioral model of the pin transition behavior of the source code. Only old-style .bv files can be read with this command.

```
verify_all_behavior();
```

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

```
enable_continuous_behavior();
```

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

```
disable_continuous_behavior();
```

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

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

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

```
resize_pin_buffer(500000);
```

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

Note that resizing the pin transition buffer can have serious affects on performance. For instance it can cause a long delay when the eTPU Development Tool is reset. It can also significantly slow down the logic analyzer redraw rate, such that the simulation speed is bound by the redraw rate. Simulation speed reductions can be obviated by hiding or minimizing the logic analyzer while the eTPU Development Tool 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 Waveform Options Dialog Box section.

8.6.4.4 External Logic Commands

Boolean logic that is external to the eTPU is instantiated through the use of **place_xyz();** 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(InPin, OutPin); Instantiates a buffer follower
- place_inverter(InPin, OutPin); Instantiates an inverter
- place_and_gate(In1Pin, In2Pin, OutPin); Instantiates an 'AND' gate
- place_or_gate(In1Pin, In2Pin, OutPin); Instantiates an 'OR' gate
- place_xor_gate(In1Pin, In2Pin, OutPin); Instantiates an 'XOR' gate
- place_nand_gate(In1Pin, In2Pin, OutPin); Instantiates a 'NAND' gate
- place_nor_gate(In1Pin, In2Pin, OutPin); Instantiates a 'NOR' gate
- place_nxor_gate(In1Pin, In2Pin, OutPin); Instantiates an 'INVERTING XOR' gate
- remove_gate(Out); Removes the gate that drives channel 'Out'

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. Indexes are defined as follows for the eTPU.

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

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

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

Two eTPU Engine Configurations

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

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

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

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

Each of the **place_xyz();** script command has an extended version that supports cross target/core gates, as shown below.

- place_buffer_ex(InTarget, InPin, OutTarget, OutPin);
  Instantiates a buffer follower
- place_inverter_ex(InTarget, InPin, OutTarget, OutPin);
  Instantiates an inverter
- place_and_gate_ex(In1Target, In1Pin, In2Target, In2Pin, OutTarget, OutPin);
  Instantiates an 'AND' gate
- place_or_gate_ex(In1Target, In1Pin, In2Target, In2Pin, OutTarget, OutPin);
  Instantiates an 'OR' gate
- place_xor_gate_ex(In1Target, In1Pin, In2Target, In2Pin, OutTarget, OutPin);
  Instantiates an 'XOR' gate
- place_nand_gate_ex(In1Target, In1Pin, In2Target, In2Pin, OutTarget, OutPin);
  Instantiates a 'NAND' gate
- place_nor_gate_ex(In1Target, In1Pin, In2Target, In2Pin, OutTarget, OutPin);
  Instantiates a 'NOR' gate
- place_nxor_gate_ex(In1Target, In1Pin, In2Target, In2Pin, OutTarget, OutPin);
  Instantiates an 'INVERTING XOR' gate

The InTarget, In1Target, In2Target and OutTarget are the names of the target and are expressed as strings such as "eTPU_A" or "Ch1_Core0".

```
place_or_gate_ex("eTPU_A", 32+16, "Ch1_Core0", 10,
"eTPU_A",11);
```

In the above example an 'OR' gate is placed with its two inputs coming from the eTPU A's channel 16 output pin and the MC33816's OA_1 pin. The output of the OR gate drives eTPU A's channel 11's input pin.

8.6.5 Code

Code coverage, which is a critical aspect of structured testing, is done with the script commands found in this section. It is also possible to arm the simulator to create a warning when a specific (and presumably unexpected) section of code executes.

8.6.5.1 Code Coverage Script Commands

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

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

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

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

The verify_file_coverage(…); and verify_file_coverage_ex(…); commands are used as part of automation testing of a specific source file. The instPct and braPct parameters are the minimum required branch and coverage percentages in order for the test to pass. The

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

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

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

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

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

Inferred Event Vector Coverage

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

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

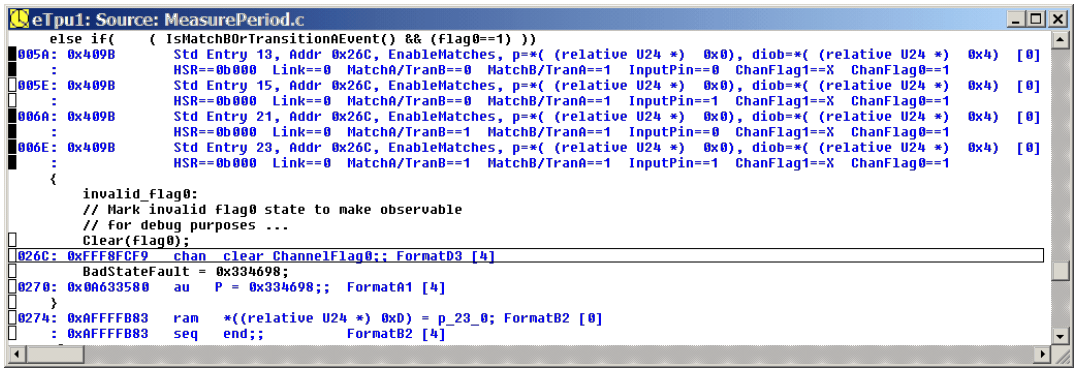
But an event vector handling the case of a rising edge and a low pin is valid. For instance, a rising edge followed by a falling edge could occur before a thread executing in another channel completes. Now the thread handling this rising edge executes with a low pin state. It is therefore important to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

serviced sequentially, so if you keep the pulse width shorter than the thread length than the second thread will execute with the input pin low.

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

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

Consider the following thread labeled, "invalid_flag0". This thread is never expected to occur because the function clears flag0 at initialization, and flag0 is never set. So thus state which handles a match or transition event in which the flag0 condition is set should never execute.



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

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

Although there are a number of restrictions listed below that are enforced by the eTPU Development Tool, the most important restriction is not enforced. Namely, that this coverage by inference should only used for invalid cases where the thread exists purely as

Built In Test (BIT) and would not in normal operation be expected to execute. In fact, when testing to the very highest software testing standards, 100 percent event vector coverage should be achieved without the use of this script command.

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

Note: when developing eTPU code using the ETEC eTPU class programming paradigm, unused/unexpected entries are often directed to the built-in global error handler. These entries thus do not show up in the user source module, but rather are associated with the global error handler source code (_global_error_handler.sta).

Cumulative Coverage

To produce the highest quality software it is imperative that testing cover 100% of instructions and 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);
```

Code Coverage Annotated Listing Files

It can be useful to generate reports files that provide coverage information at the source and opcode level. The write_coverage_listing_file() has been provided to generate such reports.

```
write_coverage_listing_file("ModuleName.c",
"ListingFileName", enum COVERAGE_LISTING_OPTIONS);
```

For a specified module (source file name), a code coverage annotated listing file with the name specified by the "ListingFileName" argument is created. Any previous file of the same name is overwritten. The output form is very much like a regular listing file as generated by the ETEC compiler, with the additions of

- source file line number preprended to each line

- opcode/disassembly lines include a coverage field that indicates to what level the opcode has been executed by tests (none, full execution, true branch, false branch, inferred coverage)

The COVERAGE_LISTING_OPTIONS argument supports two mode options

```
ALL_LINES : Output all lines of the specified module/source
file with listing information.
NON_COVERED_ONLY_LINES : Output only line associated with
opcodes not fully executed.
```

Independent flags that modify the modes listed above may be added to the COVERAGE_LISTING_OPTIONS argument. Currently one modifier flag is supported:

```
FILTER_ETPU_ENTRIES : Ignore entry table code for
disassembly purposes, and consider it fully covered so it
is ignored in non-covered only mode.
```

Example:

```
write_coverage_listing_file("CoverageListing.c", "..\\temp\
\CL_100percent_nc_filter.lst",
     NON_COVERED_ONLY_LINES + FILTER_ETPU_ENTRIES);
```

8.6.5.2 Code Warning Script Commands

A warning can be generated when the first opcode following a code label is executed. Note that the warning is generated on execution of the code, not when the program counter gets to the code. This means that the warning is executed a bit later than might be expected.

```
set_code_label_warning("PWMF_Error");
```

In the above script command a warning is assigned to the code label 'PWMF_Error'. If the code following this label is executed a warning is generated.

This can be used (say) when code that is not expected to execute actually executes. One example of this is in the eTPU

```
<...snip...>
/*-------------------------------------------------------
----------------+
| Catch all unspecified entry table conditions
                |
+-------------------------------------------------------
---------------*/
   else
   {
      PWMF_Error:
      ClearAllLatches();
   }
}
```

Note that a similar result can also be achieved using a Print action tag described more fully in the 'Print Action Tag' section. This capability is illustrated below.

```
<...snip...>
/*-------------------------------------------------------
----------------+
| Catch all unspecified entry table conditions
                |
+-------------------------------------------------------
---------------*/
   else
   {
      ClearAllLatches();  // @ASH@print("The 'dangling
else' thread is executing");
   }
}
```

8.6.6 Files

These script commands control creation of files. For instance, trace data can be piped to an output file and 'data dump' files can be generated from regions of data or code memory.

8.6.6.1 Trace Buffer and Files

Overview

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

Generating Viewable Files

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

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

Generating Parseable Files

Parseable files can be generated only by using the trace commands described in the Trace Script Commands section. To ensure generation of deterministic parseable trace files, these files can be generated only if the selected trace events are enabled within the eTPU Development Tool 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 trace file parse source code available from ASH WARE. The public methods in the TraceParser class are documented and will remain stable for future releases.

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

8.6.6.2 File Script Commands

These commands support loading and saving files via script commands.

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

This command creates the file filename.dump of type FILE_TYPE, using the options specified by DUMP_FILE_OPTIONS. The file is created from the image located between startAddress and stopAddress, out of the address space ADDR_SPACE.

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

The downside of the dump_file command is that it does a one-time dump of the entire file overwriting any previous file. An alternative to this is to continuously write the trace data to a file as it is generated. See the Trace Script Commands section.

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

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

This script command verifies two file named fileName1.dat and fileName2.dat. In addition to checking for matching or mismatching, this script command also can verify non-

existence of either file or both files. Expected results are specified with the VERIFY_FILES_RESULT enumeration.

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

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

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

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

Related Topics:

Trace Script Commands

8.6.6.3 CSV Data Import/Export

```
read_csv_1d("<data_1d_array_var>", "<csv format file>",
SCRIPT_CSV_CONTROL);
read_csv_2d("<data_2d_array_var>", "<csv format file>",
SCRIPT_CSV_CONTROL);
```

The above script commands read a data file in CSV format into script array variables. The one-dimensional version will read the first column of a CSV file into the specified array, while the two-dimensional version will read the entire CSV file into the specified script array variable. Any necessary type conversions will be performed. IMPORTANT NOTE: the script array variable dimensions will be enlarged IF NEEDED to hold the data.

```
S32 a[20];
F64 b[1][1];
S32 a_len;
S32 b_len[2];
```

```
read_csv_1d("a", "csv_data_file_10rows.csv",
CSV_READ_FIRST_LINE); // first line of file is data, not
header
a_len = read_dim_length("a", 0); // a_len gets value of 20
read_csv_2d("b", "csv_data_file_15rows_3cols.csv",
CSV_IGNORE_FIRST_LINE); // first line in file is header;
ignore
b_len[0] = read_dim_length("b", 0); // returns 15
b_len[1] = read_dim_length("b", 1); // returns 3
```

Since the read of a CSV file may change the dimensions of an array, there is a script command to retrieve the length of a script array variable dimension.

```
read_dim_length("<array_var>", <dimension index>);
```

Array data can also be exported to a CSV-format file.

```
write_csv("<source array var>", "<output file>", "<header
text>");
```

The source array variable can be one or two dimensional. It's entire contents are written to the specified file in CSV format. If the header text parameter is an empty string, no header line is written to the file and data starts on the first line, otherwise the header text is written out as the first line.

```
write_csv("capture", "pin_data.csv", "TIME, PIN STATE");
```

For enhanced flexibility, data can be appended to existing files. The file must exist before executing the command, whose format is as follows:

```
append_csv("<source var>", "<output file>");
```

The source variable may be non-array or one dimensional (appends one line), or two dimensional, in which case the number lines appended is equal to the number of rows in the array variable.

8.6.7 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 eTPU Development Tool and sets the error level to non-zero if any verification tests failed. If all tests pass, the error level is set to zero. The error level can be examined by a batch file that launched the eTPU Development Tool, 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);
print_pass(messageString);
// NOTE: does not affect exit code in auto-run mode
```

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

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

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

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

The print_pass() print command is identical to the print() script command except print_pass() leaves the error level of the simulation unaffected. The print() script command always sets the simulation error level to 1.

The print command supports using __FILE__ as an argument that gets replaced by the name (and absolute path) of the file in which it is found.

```
print("\"%s : TEST COMPLETE\", __FILE__"); // outputs
"<filename> : TEST COMPLETE"
```

In the case of print_sfn(), the error level of the simulation run is unaffected. These script commands are available to be used via action tags as well.

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

```
verify_version_ex(Tool, verHi, verLo, verBuildChar,
messageString);
```

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

```
verify_version_ex(TOOL_DEV_TOOL, 0,82,'A',
   "This demo application that illustrates GLOBAL INITIALIZATION\n"
   "will not work in this early version of the eTPU Development
Tool."  );
```

In the example above the message, "this demo …" appears if run on a eTPU Development Tool version earlier than version 3.50, build B. We recommend this script command be placed first in the script file so that it gets processed before any parse errors or unsupported command errors can occur.

Note that the following script command is deprecated in the eTPU Development Tool! However, if it is run in the eTPU Development Tool it checks the equivalent Mtdt Version. This prevents false-passes because the eTPU Development Tool version are restarted at version 1.00.

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

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

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

In the example above the message, "this demo …" appears if run on a eTPU Development Tool version earlier than version 3.50, build B. We recommend this script command be placed first in the script file so that it gets processed before any parse errors or unsupported command errors can occur.

8.6.8 Trace Script Commands

```
print_to_trace(Message)
```

This command prints the string 'Message' to the Trace Window assuming that the last executed channel at the time the script was executed is enabled in the Trace Options ... dialog box (or to a current trace stream if so configured).

```
print_to_trace_ex1(ChanNum, Message)
```

This command prints the string Message to the Trace Window assuming 'ChanNum' is enabled in the Trace Options ... dialog box (or to a current trace stream if so configured). Note that this works only in the eTPU.

```
start_trace_stream_ex("FileName.Trace", enum TRACE_EVENT_OPTIONS,
                      enum TRACE_FILE_OPTIONS, enum BASE_TIME,
                      U32 numTrailingDigits, int isZeroTraceTime);

start_trace_stream_ex("FileName.Trace", enum TRACE_EVENT_OPTIONS,
                      enum TRACE_FILE_OPTIONS, enum BASE_TIME,
                      U32 numTrailingDigits, int isZeroTraceTime);
```

These commands saves the target's trace buffer to file FileName.trace with the options set by TRACE_EVENT_OPTIONS., TRACE_FILE_OPTION, BASE_TIME_OPTIONS, and numTrailingDigits.

The numTrailingDigits field defines the number of digits to display after the decimal point. For example, if you specify numTrailingDigits to be 2 and the time options to use micro-seconds then the representation of 1.333 microseconds is '1.33'.

The 'isZeroTraceTime' parameter specifies if the time reference used by the tracing capability gets referenced to the time when the 'start_trace_stream' script command is executed. For instance, if the start_trace_stream were to be issued at time 100 microseconds and 22 microseconds an 'Pin Transition' occurred, then the time of the 'Pin Transition' event time would be listed at 122 microseconds if 'isZeroTraceTime' is a '0.' Conversely, if 'isZeroTraceTime is a '1' the time would be listed as 22 microseconds.

Note that the start_trace_stream_ex() and the start_trace_stream() script commands take identical parameters. The difference between the two command is that start_trace_stream_ex() allows OPCODE_FETCH to be disabled while MEM_READ is enabled. In the legacy start_trace_stream() if MEM_READ is enabled then the OPCODE_FETCH is also enabled.

> Note that this command has been improved starting with Version 4.8. The enabled events specified by the script command all get logged to the trace file, even if not enabled in the GUI. Any and all events specified in the script command get sent to the trace log file. Previously, when saving in 'VIEWABLE' mode, any and all events in displayed in the trace window were sent to the log file, and the parameters passed in the script file were ignored. This VIEWABLE mode behavior of previous versions was, well, goofy. And the limitation of not being able to continuously log events to the trace file if those events were not enabled in the GUI was beyond goofy ... it was flat out weak.

```
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.

```
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.

Formatted symbolic information can be generated using as described in the "String within a String" section. The following is an example of this format.

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

In the above example, variables A, B, and C must be in scope at the time the script command is executed.

However, in it is possible to print eTPU Channel Variables using the '@<ChanNum>' syntax as shown below.

The print_to_trace command supports using __FILE__ and/or __LINE__ as arguments that get replaced by the name (and absolute path) of the file in which it is found, or the file line number of the text.

```
print_to_trace("\"%s(%d) : TEST COMPLETE\", __FILE__,
__LINE__"); // outputs "<filename>(<linenum>) : TEST
COMPLETE"
```

Save Trace Buffer Script Command (DEPRECATED!)

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

WARNING: this save_trace_buffer() script command is deprecated.

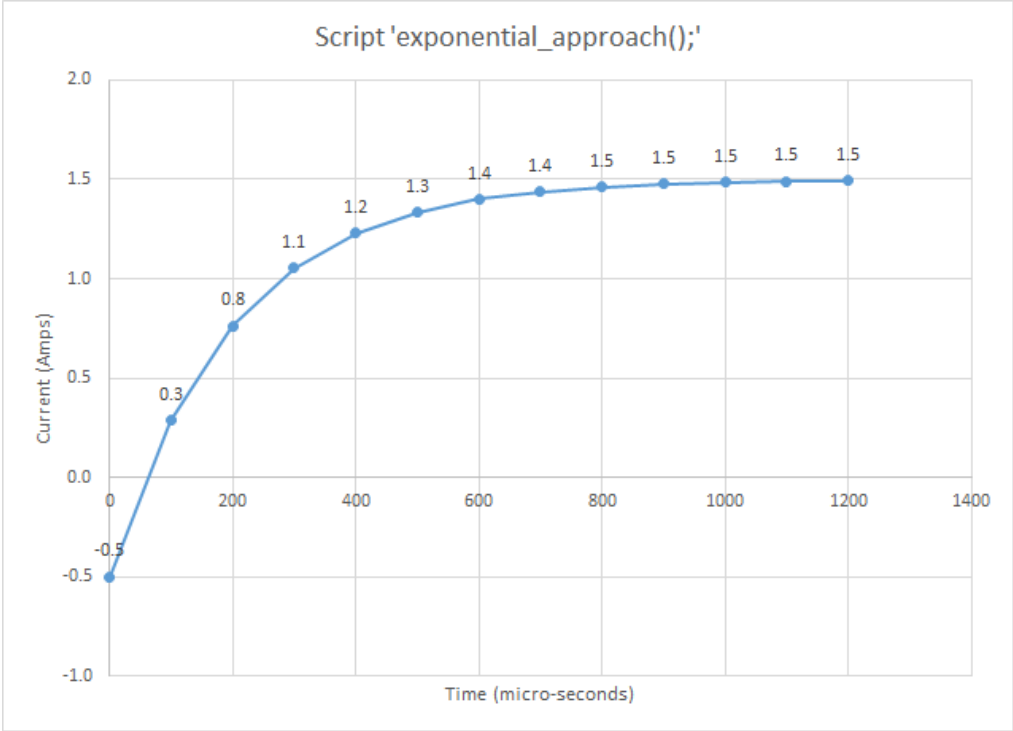
WARNING: This command won't work if the '-NoEnvFile' is used as part of a command line because the default behavior is for traces to NOT be saved to a trace buffer.

8.6.9 Math Script Commands

Exponential Approach

Say an R/L circuit starts at -0.5 amps and exponentially approaches 1.5 amps. How long does it take to cross 1.4 amps? This script command calculates the amount of time for this threshold to be reached.

```
#define INDUCTOR_MICRO_HENRIES  2000.0
#define LOAD_OHMS                 10.0
F64 time_constant_us = INDUCTOR_MICRO_HENRIES / LOAD_OHMS;
F64 start_amps     = -0.500;
F64 threshold_amps =  1.3;
F64 terminal_amps =   1.5;
F64 crossover_time_us;
// The crossover time gets set to 460 micro-seconds.
crossover_time_us = exponential_approach(start_amps,
threshold_amps, terminal_amps, time_constant_us);
```



Natural Log

```
F64 result;
result = natural_log( val );
```

This command returns the natural log of number 'Val'.

```
// Calculate the (negative) number of time constants
// for a 90% decay. Note: returns -2.3.
F64 neg_time_constants;
neg_time_constants = natural_log( 0.1 );
```

8.6.10 Simulation Configuration

```
enable_target_interrupts(InterruptSourceTarget)
```

This command enables cross-target interrupts in a multi-target environment that support such behavior. By default such interrupts are disabled from invoking interrupt handlers in

the destination target, but do signal via interrupt registers. In a System DevTool simulation, this is intended to allow eTPU interrupts (channel and global exception) to activate interrupt handlers in the simulated Host target.

```
enable_target_interrupts("eTPU_A"); // enable ISRs from
eTPU_A (run in Host script environment)
```

For eTPU targets, by default interrupts are mapped to the Host vector table as follows:

eTPU-A : channel interrupts 0-31, global exception 32

eTPU-B : channel interrupts 64-95, global exception 96

eTPU-C : channel interrupts 128-159, global exception 160

Note: this script command is currently only available on the Host target.

8.7 Automatic and Pre-Defined Define Directives

ASH WARE Specific Script

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

```
#define _ASH_WARE_SCRIPT_ 1
```

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

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

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

Tool-Specific Scripts

With the introduction of the new Development Tool in some cases it may be helpful to have tool-specific behavior. This is achieved using the 'Development Tool' define as follows.

```
#define _ASH_WARE_DEV_TOOL_ 1
```

This #define is prepended prior to parsing every script file in the 'Development Tool' but NOT in the legacy 'Mtdt' tool.

When running as a hardware debugger (rather than a simulator) the following define allows differentiation between the simulator and debugger.

```
#define _ASH_WARE_DEV_TOOL_ 1
```

The following is an example usage of simulator/debugger specific scripting.

```
#ifdef _ASH_WARE_HARDWARE_DEBUGGER_
// These scripts will execute if it is a debugger
verify_spi_data16( _AW816DA_IMM_StartPins_, 0x08);
verify_spi_data16( _AW816DA_IMM_StartPins45_, 0x00);
#else
// These scripts will execute if it is a simulator
verify_spi_data16( _AW816DA_IMM_StartPins_, 0x00);
verify_spi_data16( _AW816DA_IMM_StartPins45_, 0x01);
#endif
```

Target-Specific Scripts

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

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

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

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

Note the the target name as embedded in the macro name always has all letters capitalized. In this example the set_crystal_frequency(); script command executes only if the script command is running under a target named DBG32.

The build define is also injected as a macro into the script environment.

```
#define <BuildDefine> 1
```

A script file shared among projects testing on different targets can then have target-specific sections:

```
#ifdef MPC5554_B
// do eTPU1 stuff
#elif defined(MPC5674F_2)
```

```
// do eTPU2 stuff
#endif
```

Determining the Tool Versions

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

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

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

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

Determining the Auto-Run Mode

The eTPU Development Tool 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 eTPU Development Tool is launched in auto-run mode.

```
_ASH_WARE_AUTO_RUN_
```

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

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

Determining the Interrupt Number

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

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

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

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

```
_ASH_WARE_<TargetName>_ISR_
```

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

```
#define _ASH_WARE_eTPU_A_ISR_   25
```

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

```
#define   _ASH_WARE_ETPU_ISR_   57
```

An example of how this can be used is as follows

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

Passing Defines from the Command Line

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

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

```
echo Running ALUOP B6 Tests ...
eTpuSimulator.exe -pAutoRun.ETpuSysSimProject -d_TEST_ALUOP_B6_
if  %ERRORLEVEL% NEQ 0 ( goto errors )
```

eTPU Target Pre-Defined Define Directives

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

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

8.8 Listing of Script Enumerated Data Types

This section describes the pre-defined enumerated types used by various script commands.

8.8.1 Script FILE_TYPE Enumerated Data Type

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

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

8.8.2 Script VERIFY_FILES Enumerated Data Type

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

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

8.8.3 Script FILE_OPTIONS Enumerated Data Type

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

```
enum DUMP_FILE_OPTIONS {

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

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

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

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

```
YES_BLANK_LINES,

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

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

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

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

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

8.8.4 Trace Options Enumerated Data Types

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

```
enum TRACE_EVENT_OPTIONS {
// All targets
STEP, EXCEPTION, OPCODE_FETCH, MEM_READ, MEM_WRITE,
DIVIDER, PRINT, ERROR,

// eTPU Target only
TPU_TIME_SLOT, TPU_NOP, TPU_PIN_TOGGLE,
TPU_STATE_END, TPU_MATCH_CAPTURE,
TPU_TCR1_COUNTER, TPU_TCR2_COUNTER,
// NOTE: Same as (TPU_TCR1_COUNTER|TPU_TCR2_COUNTER)
TPU_TCR_COUNTER,

// All options
ALL,
}
```

> Note the 'OPCODE_FETCH' option has been added to the eTPU Development Tool in version 2.3 and can be used with the start_trace_stream_ex() script command. It was added because it is generally desired to see memory reads, while opcode fetches are generally not desired and are so numerous as to get in the way. Note that when using the start_trace_stream() script command, the OPCODE_FETCH is also enabled by MEM_READ.

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
}
```

8.8.5 Code Coverage Listing Options Enumerated Data Type

The following enumerated data type is used to specify the listing file options when writing a code coverage annotated listing file (write_coverage_listing_file() script command).

```
enum COVERAGE_LISTING_OPTIONS {
// listing file modes
ALL_LINES, // output all source module lines and
disassembly
NON_COVERED_ONLY_LINES, // output only non-fully-covered
source module lines
// flag items below can be added to alter main modes above
FILTER_ETPU_ENTRIES,   // do not show disassembly, and do
not show entry
                       // table source/disassembly at all
in non-covered mode
};
```

8.8.6 Base Time Options Enumerated Data Type

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

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

8.8.7 Build Script TARGET_TYPE Enumerated Data Type

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

```
enum TARGET_TYPE    {
ETPU_SIM,
};
```

8.8.8 Build Script TARGET_SUB_TYPE Enumerated Data Type

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

```
enum TARGET_SUB_TYPE   {
// eTPU2 SIM – Single Engine
MPC5632M_0_A
MPC5633M_0_A
MPC5634M_0_A
SPC563M54_0_A
SPC563M60_0_A
SPC563M64_0_A

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

// eTPU SIM – Dual Engine
MPC5566_0_1,  // Rev-0, Engine A
MPC5566_0_2,  // Rev-0, Engine B
MPC5554_B_1,  // Rev-B, Engine A
MPC5554_B_2,  // Rev-B, Engine B

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

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

8.8.9 Build Script ADDR_SPACE Enumerated Data Type

This enumerated data type is used when specifying the applicable address spaces for various build script commands. It is also used in the generic modify_mem_uXX() and verify_mem_uXX() script commands.

```
enum ADDR_SPACE {

// eTPU
ETPU_CODE_SPACE, ETPU_CTRL_SPACE, ETPU_DATA_SPACE,
ETPU_DATA_24_SPACE, ETPU_NODE_SPACE, ETPU_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!!   Target types (GTM and MC33816) cannot be
intermixed
```

```
 GTM_RAM_SPACE + MC33816_DATA_SPACE
// !!INVALID!!
// The same value cannot be added or subtracted to itself
ETPU_DATA_SPACE + ETPU_DATA_SPACE
```

8.8.10 Build Script READ_WRITE Enumerated Data Type

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

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

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

```
#define   SOME_READS     RW_READ8 + RW_READ16 + RW_READ32
#define   NON_ACCESS32S   RW_ALL -  RW_WRITE32 - RW_READ32
```

In this example, ALL_READS is defined as any read access, be it an 8-, 16-, or a 32-bit read cycle. NON_ACCESS32S is defined as all 8-, 16-, 24-, 64-, and 128-bit read and write cycles.

8.8.11 Assignment Operation Enumerated Data Type

This enumerated data type is used in the modify_mem_uXX() script commands.

```
enum ASSIGNMENT_TYPE {
     ASSIGN_EQUAL,      // =
     AND_EQUAL,         // &=
     OR_EQUAL,          // |=
     XOR_EQUAL,         // ^=
     BSL_EQUAL,         // <<=
     BSR_EQUAL,         // >>=
     PLUS_EQUAL,        // +=
     MINUS_EQUAL,       // -=
     TIMES_EQUAL,       // *=
     DIVIDE_EQUAL,      // /=
```

```
        REMAINDER_EQUAL,   // %=
    };
```

8.8.12 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 Commandssection.

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

```
enum REGISTERS_U32 {  REG_P_31_0, };

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

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

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

enum REGISTERS_U5 { REG_CHAN, };

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

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

```
write_reg32( 0x12345678, REG_P );
verify_reg32( REG_P, 0x12345678 );
write_reg24(  0x123456, REG_A );
verify_reg24( REG_A, 0x123456 );
write_reg16(  0x1234,   REG_RETURN_ADDR );
```

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

8.8.13 Pin and Node Enumerated Type

This enumerated type is used to specify external pins and various simulation nodes.

```
enum PIN_AND_NODE {

// eTPU Channel Output Pin
CHAN_OUTPUT,

};
```

8.8.14 Script CSV_CONTROL Enumeated Data Type

This is used in the CSV import script commands to control how the file is read and processed.

```
enum SCRIPT_CSV_CONTROL { CSV_IGNORE_FIRST_LINE,
CSV_READ_FIRST_LINE, }
```

9

Test Vector Files

Overview

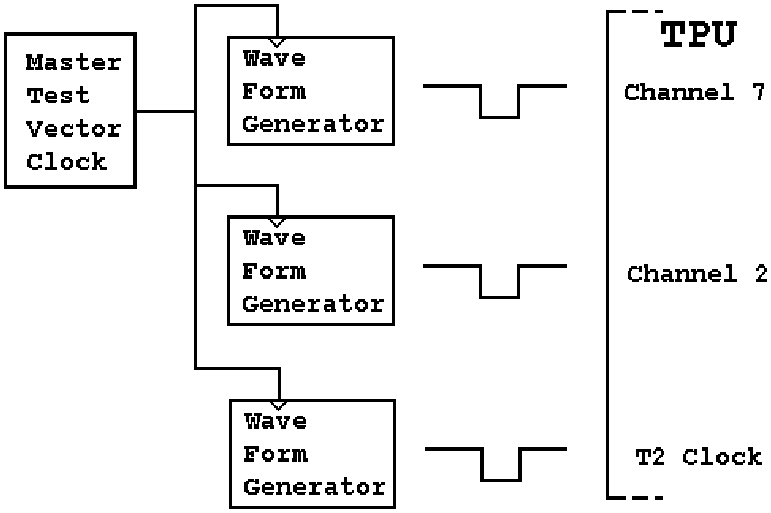
Test vector provides a complex test vector generation capability. Simulated signals similar to those found in a typical environment can be generated. Test vector files are used for forcing of "high" or "low" states at the device's input or I/O pins.

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

Loading and Editing the Test Vector File

Test vector files are opened (and created ... if appropriate) by double clicking on the 'Vector' node in the project window. Once created, the test vector file will always be reloaded prior to beginning or restarting execution.

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 drive device input pins (nodes).
- Multiple nodes are grouped (i.e., group COMM consists of UART_RCV1 and UART_RCV2).
- Complex Boolean states are defined.

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

Command Reference

The following test vector commands are available.

Node
Group
State
Frequency
Wave

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

Comments

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

```
// This is a comment.
```

9.1 Node Command

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

The node commands assign the user defined name, "Name" to a node, "Node". Note that depending on the microcontroller, the input and output from each channel may (or may not) be brought to external pins. Please refer to the NXP 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
```

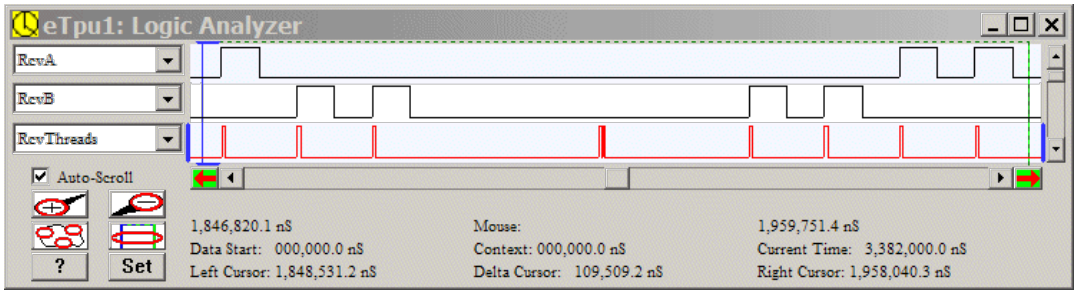
Thread Activity Nodes

Thread activity can be extremely useful in both understanding and debugging eTPU Development Tool functions. The eTPU Development Tool 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 Waveform Options Dialog Box section for more information on how to specify groups for monitoring of TPU and eTPU thread activity.

node RcvThreads ThreadsGroupB



9.2 Group Command

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

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

```
define  ADDRESS1  ch5
define  ADDRESS2  ch7
define  DATA  ch3
group  PORT1  ADDRESS1  ADDRESS2  DATA
```

In this example a group with the name PORT1 is associated with eTPU channel pins 5, 7, and 3.

9.3 State Command

```
state  <STATE_NAME>  <BIT_VALUE>
```

The state command assigns a bit value BIT_VALUE to a user-defined state name STATE_NAME. State names may contain any ASCII printable text. These state names are case sensitive (though this is currently not enforced.) Bit values must consist of a sequence zeros and ones. The total number of zeros and ones must be between one and 30.

```
state NULL  0110
```

In this example a user-defined state NULL is associated with the bit pattern 0110.

9.4 Frequency Command

```
frequency  <FREQUENCY>
```

The frequency command sets the master test vector clock to FREQUENCY which is a floating point number whose terms are million cycles per second (MHz). All test vectors are set by this frequency. Since the entire test vector file is loaded at once, if a test vector file contains multiple frequency commands, only the last frequency command is used and all previous frequency commands are ignored.

```
frequency 1
```

In this example the master test vector generation frequency is set to one MHz. This is a convenient test vector frequency because its period of one microsecond makes timing easy to calculate.

9.5 Wave Command

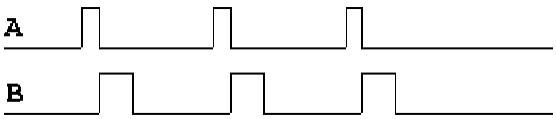
The wave command creates and defines a new wave form generator. There is no limit to the number of wave commands that may be used in a test vector file. The number of wave form generators is equal to the number of wave commands found in the test vector file.

```
wave  <GROUP>  <STATE REPEAT>  <(STATE REPEAT <...>)
REPEAT> end
```

The wave command causes the nodes of GROUP to be stimulated by the state and repeat count pairs STATE REPEAT. Multiple states and repeat counts are allowed. A special infinite repeat count, *, generates an infinite repeat count. This command may span multiple lines. An end statement must terminate the command.

```
wave OUTPUTS
     (OFF  5    DRIVE_A  1    DRIVE_B  2) 3
     OFF *
end
```

The resulting wave form from this example is shown below. The wave form begins with signal A and signal B being off for five master test vector clock periods. Signal A is then driven for one period. Then signal B is driven for two periods. These three states constitute a loop which executes three times. After the loop has executed three times the OFF state is driven forever. Note that the NODE, GROUP, STATE, and FREQUENCY commands are omitted from this example for simplification purposes.



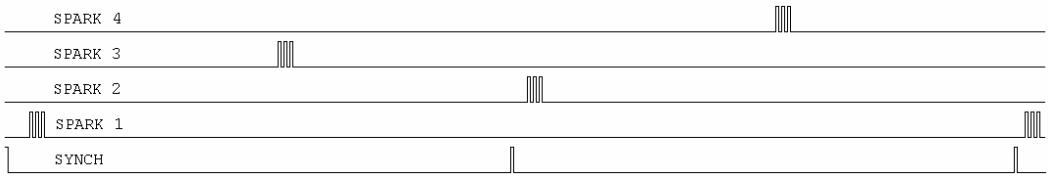
9.6 Engine Example, eTPU

```
// File:     ENGINE.Vector
// AUTHOR:   Andrew M. Klumpp, ASH WARE.
// DATE:     950404
//
```

```
// DESCRIPTION:
//     This generates the test vectors associated with a four cylinder
// car.  The four spark plugs fire in the order 1,3,2,4.  For convenience
// an engine frequency is chosen such that one degree corresponds to
// 10 microseconds (10 microseconds will be written as 10us).
// A test vector frequency is chosen such that one degree corresponds
// to one time-step.
// The test vector frequency is the eTPU Development Tool'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 eTPU channel as a member
group  SYNCH  Synch

// Make a group named SPARKS that consists of the four spark plug signals
group  SPARKS  Spark4 Spark3 Spark2 Spark1

// DEFINE THE SYNCH STATES
// The synch signal can be either pulsing (1) or waiting (0).
state  synch_pulse 1
state  synch_wait  0

// DEFINE THE SPARK FIRE STATES
// There are five states:
//    There is one state for each of the four spark plugs firing.
//    There is one state for none of the spark plugs firing.
state  FIRE4    1000
state  FIRE3    0100
state  FIRE2    0010
state  FIRE1    0001
state  NO_FIRE  0000

// SET THE TEST VECTOR BASE FREQUENCY
//    In order to have a convenient relationship between time and degrees
// an engine revolution is made to be 3600us such that one degree
// corresponds to 10us.
// Thus a convenient test vector time-step period of 10us is chosen.
// frequency = 1/period = 1/10us = 0.1MhZ
// (Frequency is expressed in MHz; this is a modification of a
//  previous version of the User Manual.)
```

```
frequency 0.1

// CREATE/DEFINE THE SYNCH WAVE FORM
// This generates a one degree (10us) pulse every 360 degrees (3600us).
wave  SYNCH  (synch_pulse 1   synch_wait 364) * end

// CREATE/DEFINE THE SPARK WAVE FORM
// Each spark is equally spaced every 1/2 revolution
// which is 180 degrees (1800us).
// Each spark plug triple fires and each fire lasts one degree (10us).
//
// In addition there is a 17 degree (170us) lag of this wave form
// relative to the synch wave form.
wave  SPARKS
      no_fire 17   // This creates a 17 degree lag

      (         // Enclose the following in a bracket to generate a loop/

             // The first plug fire cycle lasts five degrees (50us).
             // The spark plug fires three times.
             fire1 1   no_fire 1  fire1 1   no_fire 1  fire1 1

             // The delay between fire cycles is 180 degrees
             // less the five degree fire cycle.
             // 180-5=175 degrees
             no_fire  175

             // The third plug fires next.
             fire3 1   no_fire 1  fire3 1   no_fire 1  fire3 1

             // Give another 175 degree delay.
             no_fire  175

             // The second plug fires next.
             fire2 1   no_fire 1  fire2 1   no_fire 1  fire2 1

             // Give another 175 degree delay.
             no_fire  175

             // The fourth plug fires next.
             fire4 1   no_fire 1  fire4 1   no_fire 1  fire4 1

             // Give another 175 degree delay.
             no_fire  175

      ) *  // Enclose the loop and put the infinity character, *.
end
```

The following wave form is generated from the above example.

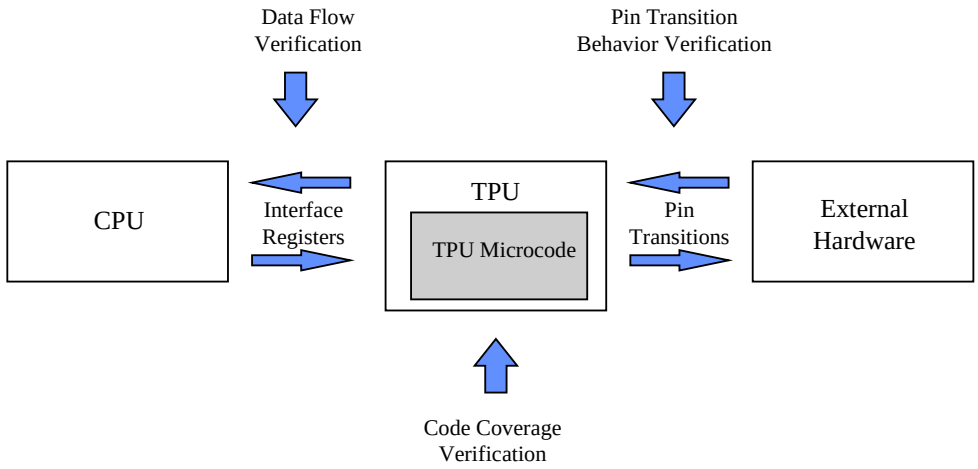

SPARK 4
SPARK 3
SPARK 2
SPARK 1
SYNCH

10

Functional Verification

Functional verification supports an automated method for verifying that behavioral requirements are met by the code. These capabilities can be grouped as data flow verification, pin transition behavior verification, and code coverage verification.

The following diagram shows a hardware perspective of functional verification. Pin transition verification is applicable only to eTPU Simulation.



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

Pin transitions are generated by the eTPU or by external hardware. Pin transition verification capabilities allow the user to verify this pin transition behavior.

Code coverage provides the capability to determine the thoroughness of a test suite. Code coverage verification allows the user to verify that a test suite thoroughly exercises the code.

A Full Life-Cycle Verification Perspective

The following describes a typical software life-cycle. Initially, a set of requirements is established. Then the code is designed to meet these requirements. Following design, the code is written and debugged. A set of formal tests is then developed to verify that the software meets the requirements. The software is then released.

Now the software enters a maintenance stage. In the maintenance stage, changes must be made to support new features and perhaps to fix bugs. Along with this, the formal tests must be modified and rerun. Then the software must be re-released.

This life-cycle can be described as having three stages: development, verification, and maintenance. All of these three stages are supported. The IDE and GUI are primarily of interest in the initial development phase. Verification involves developing a set of repeatable and automated tests that that show that the requirements are being met. In the maintenance stage these previously-developed automated tests are rerun to prove that requirements are still being met when bug fixes and enhancements cause the code base to change.

There is a significant emphasis on automation such that a complete test suite can be run, sometimes spanning hours, and a series of tests results in a single 'pass' or 'fail' result.

10.1 Data Flow Verification

Data flow verification is one of the verification capabilities for which an overview is given in the Functional Verification chapter.

Data flows between the eTPU and the CPU primarily across the Channel Interrupt Service Request (CISR) register and the parameter RAM. The data flow verification capabilities address data flow across these registers.

The eTPU parameter RAM data flow is verified using the verify_chan_data24(X,Y,Z) and verify_chan_bits24(X,Y,Z,V) script commands described in the eTPU Parameter RAM Script Commands section.

The data flow across the CISR register is verified using the verify_intr(X,Y) script command described in the eTPU Interrupt Script Commands Script Commands section.

In the following example data flow across channel 10's CISR and parameter RAM is verified at a simulated time of 100 microseconds and again at 250 microseconds.

```
// Wait for the eTPU Development Tool to run 100 micro-
seconds.
at_time(100);
// Verify that channel 10's CISR is set.
verify_cisr(0xa,1);
// Verify that channel 10's parameter 2 bit 14 is set.
verify_ram_bit(0xa,2,14,1);
// Verify that channel 10's parameter 3 is 1000 hex.
verify_ram_word(0xa,3,0x1000);
// Verify that channel 10's parameter 5 is 1500 hex.
verify_ram_word(0xa,5,0x1500);
// Clear channel 10's CISR.
clear_cisr(0xa);
// Wait for the eTPU Development Tool
// to run an additional 150 microseconds.
wait_time(150);
// Verify that channel 10's CISR is set.
verify_cisr(0xa,1);
// Verify that channel 10's parameter 2 bit 14 is cleared.
verify_ram_bit(0xa,2,14,0);
// Verify that channel 10's parameter 3 is 3000 hex.
verify_ram_word(0xa,3,0x3000);
// Verify that channel 10's parameter 5 is 3500 hex.
verify_ram_word(0xa,5,0x3500);
```

10.2 Pin Transition Behavior Verification

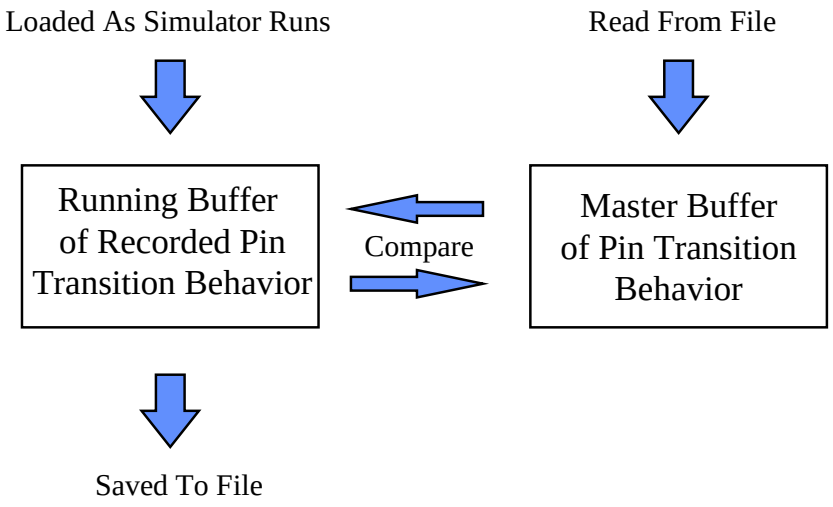
Pin transition behavior verification is one of the verification capabilities for which an overview is given in the Functional Verification chapter. Pin transition behavior verification capabilities include the ability to save recorded pin transition behavior to enhanced pin transition behavior (.ebv)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

behavior matches the saved behavior. Old-style behavior verification files (.bv) can still be read and used for verification, but can no longer be created. All pin transition behavior verification must now be managed via scripting. GUI menu options are no longer available.

From a behavioral model perspective these capabilities correlate to the ability to create behavioral models and the ability to compare source microcode against these behavioral models. The emphasis in the eTPU Development Tool is on the automation of these modeling capabilities through script commands, as the capabilities are no longer available from the menus.

Pin Transition Buffers

There are two pin transition storage buffers in the eTPU Development Tool as shown in the following diagram.



The running buffer is filled as the eTPU Development Tool runs. Whenever a pin transition occurs, information regarding this transition is stored in this buffer, which is also used to draw the signals in the logic analyzer window. This pin transition information can be selectively recorded to file using the create_ebehavior_file("filename.ebv"), add_ebehavior_pin("<pin name>") and close_ebehavior_file() script commands; also see Enhanced Pin Transition Behavior Verification below.

Enhanced Pin Transition Behavior Verification

The new enhanced pin transition behavior verification makes use of a user-friendly data file format and provides more options on how master file pin transitions are compared to transitions in the current simulation. With the original behavior verification capability, anything other than an exact match, clock cycle for clock cycle, was considered a failure. The reality is that often times even very small code changes can result in a signal being shifted by a clock cycle or two, which previously would result in failure. Enhanced behavior verification provides options for controlling the tolerance used in comparisons, so that small changes that are functionally correct can pass verification. Additionally, master file pin transition data can now be viewed in the Waveform Window making it easier to track down problems when a failure does occur.

The first step in making use of pin transition behavior verification is to create a master file, also called a .ebv file for its default extension. These enhanced behavior verification files are text and in comma-separated value (CSV, or .csv) format. All behavior verification is controlled from the scripting environment - to create a master file use the create_ebehavior_file("filename.ebv") script command. It must execute at simulation time zero, but after any vector file load or pin buffer placements.

```
// save off all pins
create_ebehavior_file("Engine.EBV");
```

By default all pins are saved to the .ebv file. On the eTPU, this is all 32 channel input pins, all 32 output pins, and the TCRCLK pin. In many cases, only a small subset of pins are of interest. The add_ebehavior_pin("<pin name>") script command allows the user to save data only on those pins of interest. The pin name comes from any node naming in the vector file, or is the underlying default pin name (e.g. "ch3" is the channel 3 pin on the TPU, or "_ch5.out" is the channel 5 output pin on the eTPU).

```
// save off only the fuel injection output signals
create_ebehavior_file("Engine_Injection.EBV");
// add_ebehavior_pin commands must immediately follow the
create command
add_ebehavior_pin("Injector1");
add_ebehavior_pin("Injector2");
add_ebehavior_pin("Injector3");
add_ebehavior_pin("Injector4");
```

After the creation setup commands are complete, and the script file exercises the software, the .ebv file needs to be closed and any unsaved data flushed out. Note that data may be saved in several steps as the simulation runs - enhanced behavior verification is not subject

to buffer size limitations due to its continuous file management. This is done with the close_ebehavior_file() script command.

```
wait_time(20000000);    // Let the tests run to completion
close_ebehavior_file(); // close EBV being saved
```

As mentioned, the .ebv file is in comma-separated value format. The first line contains the column header information, which is time units for the first column, followed by pin names of all saved pin data. After that, each line contains a simulation time value in microseconds followed by the value of each pin. Below is an example of the first few lines of an .ebv file which contains data on just two pins.

```
TIME (US), PWM_A, PWM_B,
000000000000.000000, 0, 1,
000000000019.980000, 1, 1,
000000000039.980000, 0, 1,
000000000049.980000, 0, 0,
```

The comma-separated value allows the file to be easily parsed by other existing tools, such as spreadsheet applications.

Once a master .ebv file has been created, it can be used to verify pin behavior stays within an expected tolerance range on subsequent simulation runs. Typically this is done after a software change has been made that is not supposed to affect pin transition behavior. With enhanced behavior verification, only a continuous verification model is supported. As with .ebv file creation, all control is via script commands. First, which .ebv file to be used for the verification run must be specified - this is done with run_ebehavior_file("filename.ebv"). By default, all pins found in the .ebv file are verified with a default transition error tolerance equivalent to two system clocks.

```
// test all pins
run_ebehavior_file("PWM_gold.EBV");
```

The above command should be issued at simulation time zero, at the same place in the script file as the .ebv file was generated with the create_ebehavior_file() command. The simulation should run, and at the end, at the same time as when the .ebv file was saved and closed, the verification should be stopped with the stop_ebehavior_file() script command.

```
stop_ebehavior_file(); // stop EBV verification
```

Tolerances can be adjusted, and the pins being verified can be specified, using the enhanced behavior verification pin tolerance script commands. With the set_ebehavior_global_tolerance() script command, all pins under test can have their allowed error tolerance set.

```
run_ebehavior_file("Engine_gold.EBV");
// verify all pins in the .ebv file to a 2us tolerance
set_ebehavior_global_tolerance(EBV_ABSOLUTE, 0.0, 2.0);
```

The set_ebehavior_pin_tolerance() script command serves a dual purpose. Once one of these commands is specified, only pins specified with set_ebehavior_pin_tolerance() script commands will be verified.

```
run_ebehavior_file("PWM_gold.EBV");
// test only PWM_A and PWM_C pins w/ appropriate tolerance
set_ebehavior_pin_tolerance("PWM_A", EBV_ABSOLUTE, 0.0,
1.1);
set_ebehavior_pin_tolerance("PWM_C", EBV_ABSOLUTE, 0.0,
1.1);
```

Although the tolerances can be adjusted at any time during simulation, if using set_ebehavior_pin_tolerance() to configure a subset of pins to test, these commands should directly follow the run_ebehavior_file() command. Tolerances can then be adjusted later, either by individual pin or globally.

Currently there is one tolerance type - "EBV_ABSOLUTE". This means the absolute time of each transition in the master file is compared to the absolute time of the matching transition in the current simulation run. In the comparison process, first the offset is is applied to the master file transition time, and then the difference between the two times is calculated. If the absolute value of this result is less than the configured tolerance, the behavior is considered valid and simulation continues. If it is greater than the tolerance, a behavior error is thrown - this may or may not trigger a pop-up error dialog depending upon the IDE messages configuration.

Last, individual pins can be disabled from behavior verification with the disable_ebehavior_pin() script command.

```
disable_ebehavior_pin("PWM_C");
```

With enhanced behavior verification, an .ebv file can be created while simultaneously using another one for verification. This could be handy in the sense that after running a simulation session the user has a new current pin transition file - it could be used as an input into other tools, or if verification is successful, it could be copied as the new "gold", or master file.

```
// create and verify simultaneously
create_ebehavior_file("Powertrain_current.EBV");
run_ebehavior_file("Powertrain_gold.EBV");

// ... simulation ...

// finish behavior verification
close_ebehavior_file(); // close EBV being saved
stop_ebehavior_file(); // stop EBV verification
```

When enhanced behavior verification is used with a dual-eTPU simulation model, an .ebv file and the associated commands apply to a single eTPU engine. In other words, if pin transition behavior verification is to be done on each eTPU, there must be a separate .ebv file for each.

10.2.1 Deprecated Pin Transition Behavior Verification

Legacy .bv pin transition data files can still be used by the new enhanced behavior verification; .bv files can no longer be generated. The master buffer can be loaded only with pin transition behavior data from a previously saved file. This file forms a behavioral model of the source microcode. Changes can be made in the source microcode and the changed microcode can be verified against these behavioral models. This file is loaded using the read_behavior_file("filename.bv") script command.

There are two options for verifying pin transition behavior against the previously-generated behavioral model. The first option is to continuously check the running pin transition behavior buffer against the master pin transition behavior buffer. This is selected by the enable_continuous_behavior() script command that follow a behavior file load. The second option is to perform a complete check of the running buffer against the master buffer all at once. This is selected using the verify_all_behavior() script command at the end of a simulation run. Time tolerances for pin transitions default to 2 system clocks, but can be adjusted using the new enhanced behavior verification set_ebehavior_pin_tolerance() and set_ebehavior_global_tolerance() script commands.

A count of failures is displayed in the Configuration Window. This count is incremented whenever a behavior verification failure occurs. By default, each failure will generate an error dialog, but this can be disabled by de-selecting Options -> Messages... -> Behavior verification failure.

A consideration regarding these behavioral models is the 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 second consideration is TCR2 pin recording. Normally TCR2 pin transitions are not written in these buffers. This is because the recording of TCR2 pin transitions very quickly fills up the buffer and causes the buffer to quickly roll over. When the buffer rolls over verification is not possible with legacy .bv files.

10.3 Pin Transition Verification Example

The following is an example eTPU Commands Script that both generates, and verifies pin transition data. The #define '_CREATE_EBV_FILE_' determines if the .EBV file will be generated (create the gold file) or tested (test the current run against a previously-generated gold file.

```
// Linker-generated header file
#include "MyCode_defines.h"

#define TEST_CHAN 4

// Set the clock to 100 MHz (10 ns/clock -->1e7
FemtoSeconds/clock)
set_clk_period(10000000);
write_entry_table_base_addr( _ENTRY_TABLE_BASE_ADDR_ );
write_tcr1_control(2);        // System clock/2,  NOT gated
by TCRCLK
write_tcr1_prescaler(1);
write_global_time_base_enable(1);

// Channel Configuration Registers Functions (CxCR, CxSCR,
CxHSRR)
write_chan_base_addr(       TEST_CHAN, 0x300);
write_chan_func(            TEST_CHAN, _FUNCTION_NUM_Pwm_);
write_chan_entry_condition( TEST_CHAN,
_ENTRY_TABLE_TYPE_Pwm_);

//
*************************************************************
***************
write_chan_cpr  ( TEST_CHAN, 3);
write_chan_hsrr ( TEST_CHAN, 7);
write_chan_data24 (TEST_CHAN, _CPBA24_Pwm_Period_,
0x000400);
write_chan_data24 (TEST_CHAN, _CPBA24_Pwm_HighTime_,
0x000050);

//#define _CREATE_EBV_FILE_

#ifdef _CREATE_EBV_FILE_
create_ebehavior_file("MyGoldSim.ebv");
add_ebehavior_pin("PWM");
```

```
#else
run_ebehavior_file("MyGoldSim.ebv");
#endif // _CREATE_EBV_FILE_

//
*************************************************************
****************
// Run the simulator for awhile
// Compare recorded pin-transitions with known-good gold
file
wait_time(1000);

#ifdef _CREATE_EBV_FILE_
close_ebehavior_file();
#else
stop_ebehavior_file();
#endif  // #ifdef _CREATE_EBV_FILE_

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

10.4 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 the project's IDE Options settings. When this is enabled, black boxes appear as the first character of each source code line that is associated with a microinstruction. As the code executes, and the instruction coverage changes, these black boxes change to reflect the change in the coverage. This is summarized below.

- A black box indicates a 'C' source line or assembly instruction that has not been executed.
- An orange box indicates 'C' source line that has been partially executed.
- A blue box indicates an assembly branch instruction in which neither branch path has been traversed.
- A green box indicates an assembly branch instruction where the branch path has been traversed.
- A red box indicates an assembly branch instruction where the non-branch path has been traversed.
- A white box indicates a 'C' source line or assembly instruction that has been executed.

Code Coverage Verification Commands

The code coverage verification commands, described in the Code Coverage Script Commands section provide the capability to verify both instruction and branch coverage percentages on both an individual file basis and a complete microcode build. If the required coverage has not been achieved then a verification error message is generated and the count of script failures found in the Configuration Window is incremented.

Code Coverage Report Files

Code coverage report files can be generated using the write_coverage_file("filename.Coverage") script command described in the Code Coverage Script Commands section. Code coverage report files can also be generated directly from the File menu by selecting the Coverage submenu. This is described in the Files Menu section.

The top of the code coverage report file contains a title line, a copyright declaration line, and a time stamp line.

Following this generic information is a series of sections. The first section provides coverage data on the entire microcode build. Succeeding sections provide coverage 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 (eTPU target) follows.

```
// Code coverage analysis file.
// Copyright 1996 ASH WARE, Inc.
// Wed Feb  6 09:08:29 2019

Total
        Instructions:          43
        Instruction Hits:      14
        Instruction Coverage Percent:   32.56
        Branches:              18
        Branch Hits:           4
        Branch Coverage Percent:        22.22
        Entries:           32
        Entry Hits:        5
        Entry Coverage Percent:     15.63
C:\Mtdt\TestsHigh\ETpu\Coverage\Pwm.c
        Instructions:          14
        Instruction Hits:      14
        Instruction Coverage Percent:   100.00
        Branches:              4
        Branch Hits:           4
        Branch Coverage Percent:        100.00
        Entries:           32
        Entry Hits:        5
        Entry Coverage Percent:     15.63
C:\Mtdt\Gui\testfiles\Lib\_global_error_handler.sta
        Instructions:          29
        Instruction Hits:      0
        Instruction Coverage Percent:   0.00
        Branches:              14
```

```
Branch Hits:            0
Branch Coverage Percent:        0.00
Entries:            0
Entry Hits:         0
Entry Coverage Percent:     0.00
```

Code Coverage Annotated Listing Files

Listing files (source file lines with the associated disassembled opcodes shown below each line that generates code) annotated with code coverage information can be generated with the write_coverage_listing_file() script command described in the Code Coverage Script Commands section. An option parameter allows users to select between outputting a complete annotated listing file for a specified module, or output just the source lines and object code not executed (covered). In both cases every line is prepended with the original source line number. When the non-covered listing option is specified, a line gap is placed between annotated source/assembly wherever source code lines have been filtered. A small annotated sample is shown below.

```
[ 536]:              else if ( IsMatchAEvent() && (flag1==1)  &&
(flag0==1) )
[ 536]: [X]: 0016: 0x4250       Alt Entry 11, Addr 0x940, EnableMatches,
p_31_0=*( (channel U32 *)  0x0), diob=*( (channel U24 *)  0x5)  [0]
[ 536]:    :     :              HSR==0b000  Link==X  matchA/TranB==1
matchB/TranA==0  InputPin==0  ChanFlag1==1  ChanFlag0==1  [2]
[ 536]: [ ]: 001E: 0x4250       Alt Entry 15, Addr 0x940, EnableMatches,
p_31_0=*( (channel U32 *)  0x0), diob=*( (channel U24 *)  0x5)  [0]
[ 536]:    :     :              HSR==0b000  Link==X  matchA/TranB==1
matchB/TranA==0  InputPin==1  ChanFlag1==1  ChanFlag0==1  [2]
[ 537]:              {
[ 538]:                     // All 32 bits were received & half a gap has
been detected
[ 539]:                     // filtering (if any) is done now, while parity
check is done
[ 540]:                     // when the full gap is validated
[ 541]:            AW_A429R_ValidHalfGapFilter:
[ 542]:
[ 543]:                     ClrFlag0();
[ 543]: [X]: 0940: 0xDFE87A8C   ram   p_23_0 = *((channel int24 *) 0x31);
FormatD6 [0]
[ 543]:    :     : 0xDFE87A8C   chan  clear ChannelFlag0, clear
MatchRecognitionLatchA;; FormatD6 [4]
[ 544]:                     ClearMatchAEvent();
[ 545]:
[ 546]:                     if (RxFilter.Label)
[ 546]: [X]: 0944: 0x0008F019   alu    nil = p+0x0, SampleFlags;; FormatA2
[4]
```

```
[ 546]: [T]: 0948: 0xF0D84BC7   seq  if z==true then goto addr_0x978,
flush;; FormatE1 [4]
[ 547]:                     {
[ 548]:                           unsigned int24 bitIndex;
[ 549]:                           unsigned int24 filterWord;
[ 550]:            #ifdef __ETEC__
[ 551]:                           bitIndex = __bit_n_update(0,
Data.B.Bits23_0 & 15, 1, 0);
[ 551]: [ ]: 094C: 0xBFEFFB80   ram   p_23_0 = *((channel int24 *) 0x1);;
FormatB2 [4]
[ 551]: [ ]: 0950: 0x0C380BFA   alu    a = p & 0xF;; FormatA3 [4]
[ 551]: [ ]: 0954: 0x3F3F1FFD   alu    sr = ((u24) 0) | (1<<a);; FormatB6
[4]
```

After the prepended source code line number, each opcode line has coverage status shown inside brackets [], followed by code address, opcode (or entry point), and disassembly. The coverage status can be one of the following:

```
[ ] - not executed at all
[X] - fully executed (both paths taken if branch)
[T] - only branch true path executed
[F] - only branch false path executed
[I] - inferred entry point coverage
```

Note, if the non-covered listing option is specified, and the module has been fully executed, the result is an empty output file.

For eTPU targets, an additional flag option can be applied that causes entry table items to be ignored (no disassembly for "all" mode, all entry source and disassembly filtered for non-covered mode).

10.5 Regression Testing (Automation)

Regression Testing supports the ability to launch the eTPU Development Tool from a DOS command line shell. Command line parameters allow specific tests to be run. From the command line the project file that is run and the primary script file(s) that are loaded into each target are specified. Command line parameters also are used to specify that the target system automatically start running with no user intervention, and to accept the license agreement thereby bypassing the dialog box that would otherwise open up and require user intervention. A script command is used to terminate the eTPU Development Tool once all the tests have been run.

Upon termination, the eTPU Development Tool 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 eTPU

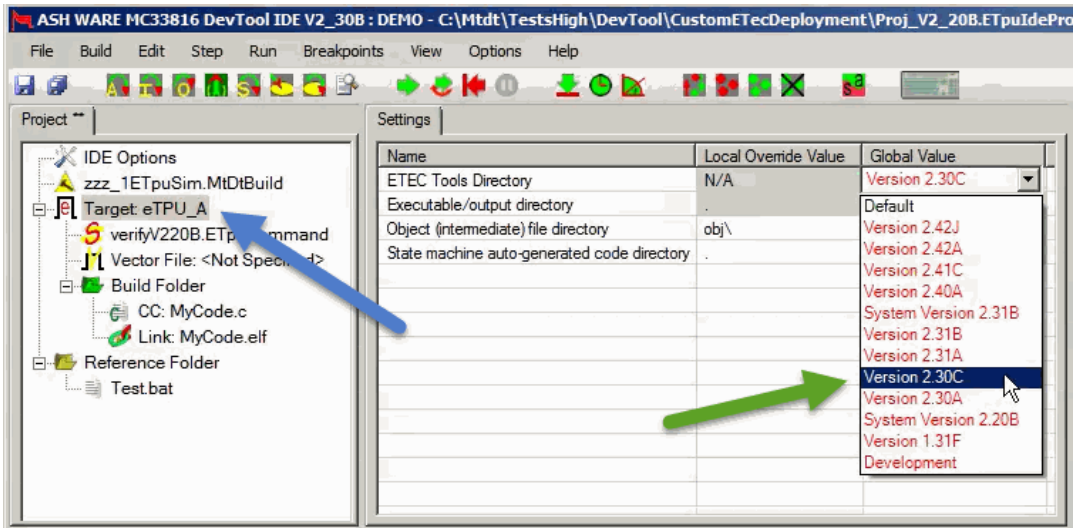
Development Tool's termination code and can be queried within a batch file running under the operating system's DOS shell. By launching the eTPU Development Tool multiple times, each time with a different set of tests specified, and by checking the error level each time the eTPU Development Tool terminates, 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 7, Windows XP Professional, Windows 2000 Professional, and Windows NT 4.0. do support automation.

10.6 Testing with a Specific Compiler Version

A production software release will generally be tied to a specific ETEC Compiler version. However as ASH WARE continues to release new software versions, the once-default (latest installed) software version will at some point longer be the default. It is therefore important to be able to tie regression tests to a specific ETEC compiler version. This is done by selecting the build target as show by the blue arrow, below. Right-click to bring up the popup menu and select 'settings' to bring up the settings window shown below. Within the settings window, select the version of the compiler you wish to tie the tests to. The green arrow shows the tests being tied to ETEC compiler version 230C. Note that if the tests were to be run on a compiler that does not have version 2.30C installed, then the tests would fail. This setting is stored in the project file and not the environment file!

Also, see related section 'Using Non-Installed ETEC Versions'.



10.7 Command Line Options

When launching the eTPU Development Tool the last-loaded project file is automatically loaded and the project file contains most of the settings such as the name of the script file to load, which messages to suppress, etc. However, it can be useful to specify the project file to load or even to override settings contained in the project file. This is by specifying settings at the command line when launching the eTPU Development Tool. This is especially useful when building regression tests.

Setting	Option	Example
Display Help Prints a list of all the Command Line Parameters	-h	-h
Project File Loads the specified project file. Note that if a relative path is used to specify the project file, it is relative to	-p=<ProjectFileName>	-p=Proj.ETpuIdeProj Loads project file "Proj.ETpuIdeProj"

Setting	Option	Example
the current working directory at app launch. Also note that all other relative paths is command line options are relative to the project file directory, NOT the current working directory (although they are often the same).		
Executable Image File Loads the specified executable image file (e.g., MyCode.elf)	-exe=<ExeImageFile>	-exe=<MyCode.elf> Loads "MyCode.elf"
Script File Loads the specified script file (single target). Path relative to project file directory.	-s=<ScriptFileName>	-s=RarChunkTest.Cmd Loads script file "RarChunkTest.Cmd"
Script File Loads the specified script file into the specified target (multiple targets). Path relative to project file directory.	-s=<ScriptFileName>@@<TargetName>	-s=MyScript.Cmd@@Cpu32 Loads script "MyScript.Cmd" into the CPU32 simulation model named "Cpu32"
Vector File Loads the specified vector file (single target) path relative to project file directory.	-v=<VectorFileName>	-v=MyVector.vector

Setting	Option	Example
Vector File Loads the specified vector file into the specified target (multiple targets). Path relative to project file directory.	-v=<VectorFileName>@@<TargetName>	-v=MyVector.vector@@Ch1_Core0
Vector Define In vector file, define the DefinedText as Val. Note that the '=<Val>' portion can be omitted and 'VAL' will be assigned a default value of '1'.	-vd=<DefinedText>=<Val>	-vd=CONTROL_CASE Passes the #define CONTROL_CASE 1 to the vector file -vd=CMD_LINE_DEFINE_A=01 -vd=CMD_LINE_DEFINE_B=10 Passes two defines with values '01' and '10' on the command line.
Define as true In script commands file, #define DefinedText 1	-d=<DefinedText>	-d=CODE_BASELINE Passes the #define CODE_BASELINE 1 to the script file
Define as value In script commands file, #define DefTxt Val	-d=<DefTxt>=<Val>	-d=MY_VAR=55 Passes the #define MY_VAR 55 to the script file
Build Define In MtDt build script, define the DefinedText as true	-bd=<DefinedText>	-bd=MPC5674_2 Defines MPC5674 as 'true' in the build define such that

Setting	Option	Example
		the MPC5674 rev 2 model is loaded.
Log File Logs messages to end of file, "FileName.log". Path relative to project file directory.	-lf5=<FileName.log>	-lf5=Error.log Logs messages to file "Error.log"
Log File Test Suite Mode Outputs formatted messages and extra verification information into the log file, when specified. Useful when using the tool for verification purposes.	-lfmTestSuite	-lfmTestSuite Sets logging to test suite mode (only applies if logging enabled via -lf5).
Test Name Used in conjunction with the log file to append a test result to the end of the log file.	-tn=<TestName>	"-tn=Pulse Width Test" Appends 'Pulse Width Test Passes' (or 'fails') to the end of the log file.
Suppress Warning #1 Suppress the "Source Code Missing" warning	-ws=1	-ws=1
No Popup Dialogs Disables display of dialog boxes	-q	-q
Suppress Environment File Load	-NoEnvFile	-NoEnvFile

Setting	Option	Example
Normally an environment file is loaded that specified window positions, file scroll locations, etc. This suppresses loading of the environment file and can improve the repeatability of certain tests.		
Run Minimized Runs the eTPU Development Tool 'Minimized' such that it is possible to continue on ones computer while running regression tests.	-Minimize	-Minimize
Automatically Build Forces the target system to rebuild the code, even when not out of date, when the eTPU Development Tool is launched. Note that if a target's build is disabled, this will NOT cause the code to build. Instead, use the '-EnableCodeBuild=<Target>' option.	-AutoBuild	-AutoBuild
Override a Disabled Build For a target with it's build disabled in the project file, this overrides the 'Disable' such that the code will be built.	-EnableCodeBuild=<Target>	-EnableCodeBuild=eTPU_A

Setting	Option	Example
Prevent Build Prevents code rebuild even if the code is out of date.	-NoBuild	-NoBuild
Automatically Run Sets the target system to running when the eTPU Development Tool is launched	-AutoRun	-AutoRun
Network Retry Time Time to wait (in seconds) for a network license if none is available	-NetworkRetry=<Sec>	-NetworkRetry=30 Retries for 30 seconds if unable to make a network connection
Network License Checkout Perform a network license checkout, providing success/fail status, and then exiting.	-NetworkCheckout=<y:m:d:h:m>	-networkcheckout=2035:5:6:16:00 Attempts to check out a license until 4pm of May 6, 2035.
Network License Checkin Check back in a checked-out license, which frees the license for other users (after the linger time has completed)	-NetworkCheckin	-NetworkCheckin

10.7.1 Using the –d (define) Option and Escape Characters

There are issues with passing a quote character into eTPU Development Tool in Windows because Windows uses the quote character to bunch multiple pieces of text into a single command line parameter. Consider the case where a filename is to be passed into the eTPU Development Tool. A good way of doing this is to define a string, then use the dump_file(); script command as follows.

```
#define FILE_TO_DUMP "n:\\MyDataFile.dat"
dump_file(0, 0x28, ETPU_DATA_SPACE, FILE_TO_DUMP, IMAGE, DATA8);
```

It is possible to pass the FILE_TO_DUMP #define into the eTPU Development Tool from the command line (instead of having it in the script commands file) using the following command line parameter.

```
"-dFILE_TO_DUMP=\"MyDataFile.dat\""
```

There are four quote characters in the command line parameter shown above. The first and last quote characters are used by Windows to bunch everything between them together into a single command line parameter which would (for example) allow spaces to appear within the single parameter.

But the filename, MyDataFile.dat, is a string, and strings must be surrounded by quote characters in the script commands file. This is accomplished by preceding the quote character with the backslash character. Windows interprets this backslash-quote combination literally and the quote character is thereby passed along with the filename into eTPU Development Tool.

10.7.2 Warning Suppresion Command Line Options

Warning suppression command line options are used to suppress warnings from being generated.

This is useful for regression testing in which a warning can cause a regression test to fail.

Although warning can also be suppressed in the 'Messages Options' dialog box, doing so does not work well in conjunction with regression tests, and here is why. Messages disabled in the 'Messages Options' dialog are stored in the environment file. Each user has their own environment file. So a regression test that might pass when one user runs it might fail when the other user runs it because one user has a message enabled that the other user has disabled. So by disabling options on the command line, all users running the same regression test will get the same behavior.

Note that this is significantly different in DevTool relative to MtDt because MtDt did not have environment files and therefore MtDt stored these Messages Options in the (generally common) project file.

The following list shows the value and meaning of each command line warning.

Warning Or Message	ID
Behavior Verification Failure	101
Script Command Verification Failure	102
Bad at_time(); script command	103
Divide by zero	104
Obsolete set_cpu_frequency script command	105
A 32-bit script not at double-even address boundary	106
A 24-bit script not at single-odd address boundary	107
A 16-bit script not at an even address boundary	108
A memory-write script failed to write	109
Non-ASCII character in script command file	110
'print_pass()' script command	111
Missing source code	112
New HSR issues with prior HSR still set	200
New LSR issues with prior LSR still set	201
Un-Initialized User-Defined Chan Mode (UDCM) Selected	202
eTPU2 code loading into eTPU1 model	203
AU's B bus active, but ignored	204
Semaphore Locked too long	205
Reserved field	206
Invalid Entry	207
Entry Standard/Alternate Mismatch	208
Invalid PRAM Access	209

Invalid OPCODE fetch	210
Invalid call return	211
Un-flushed call followed by another call	212
MRLE set when MRL==1	213
Race condition detected	214
Invalid TBCR.TCR2CTL Value	215
Invalid TBCR.TCR1CTL Value	216
Event vectoring on output pin unsupported	217
Write of unsupported ETPUSCMOFFDATAR<Not Applicable>	218
Invalid Memory Size	219
Bad CIN field assertion, CIN is ignored in this operation	220
MACH or MACL access prior to operation completion	230
SampleFlags active for a MAC operation	231
Bad combination of BINV and CIN for this operation	232
TPR's LAST, TICKS, and HOLD asserted all at once.	240
TPR.TICK changed before or on TPR.IPH assertion	241
TPR's IPH asserted, then TICKS changed on next cycle	242
TPR's IPH asserted, then LAST asserted on next cycle	243
Setting of TPR.LAST on two contiguous teeth	244
Angle Overflow TCR2 > (TPR.TICKS+1) * TeethPerCycle	245
TBCR.AM changed while MCR.GTBE is enabled	246
Detection Edge (TBCR.TCR2CTL) mismatches CH0 IPAC	247
Bus error	300
Address error	301
Breakpoint instruction encountered	302
Privililedge violation	303

Stack frame error exception	304
Bad Binary Coded Decimal (BCD) number	305
Interrupt with IARB equal to zero	306
Interrupt with duplicate priority & IARB	307
Double Bus Fault	308
Chk/Chk2 exception	309
Ill-formed CHK2 or CMP2 bounds par (b1>b2)	310
Legal illegal instruction	311
Illegal illegal instruction	312
A and F line emulator exceptions	313
Un-initialized jump register	400
Un-initialized subroutine return address register	401
Un-initialized indexed access base address register	402
Out of bounds RAM access	403
Out of bounds CODE access	404
Invalid opcode	405
Set control register bit 'stcrb' instruction when control register is read-only due to 'Control_register_split.cr_shared_ucX' being set	410
Writing the 'Condition Register' (ar) as an ALU result. This register cannot be written as an ALU result.	411
Executing a new SHIFT or MULTIPLY operation before the not-yet-finished SHIFT or MULTIPLY operation is complete.	412
Accessing the 'mh' or 'ml' register before the not-yet-finished MULTIPLY or SHIFT operation is still underway.	413
Shift is out of the valid range which is between 1 and 15 bit positions,	414
ALU Write of the same register being written by the SHIFTER (note: alu result is overwritten, shifter 'wins')	415
Bad Shortcut	420

Bad Core Clock Configuration. Set 'ck_per' to 3 (SysClk/4) in dual-core mode, or 1 (SysClk/2) in single core mode. See also 'flash_enable.'	421
Core execution not enabled. To enable, set 'Flash_enable.pre_flash_enable'	440
Core execution on second core not enabled. To enable, set 'Flash_enable.en_dual_uc'	441
Code Width too small. Note: 'Flash_enable.Code_width' must be >= 3	442
Checksum failure. See 'Flash_enable.checksum_h' and 'Flash_enable.checksum_l'	443
VBOOST Overvoltage, the VBOOST circuitry has exceeded the maximum allowed voltage	450
Waiting on un-initialized wait table row	470
Waiting on un-initialized timer	471
Waiting on un-enabled 'Start Event' (register 'Start_config_reg', start6_sens_ucX through start1_sens_ucX all 0's)	472
Waiting on un-written DAC	473
Waiting on un-settled DAC	474
Waiting on un-settled OP AMP	475
Bad wait table row (row>5)	476
A 'reqi' instruction has been executed while the Software Interrupt Service Routine is already active or pending	480
A 'iret' instruction has been executed but no Software Interrupt Service Routine is active	481
A second START interrupt occured prior to completion of an existing SOFTWARE interrupt	482
Interrupt Service Routine (ISR) has begun but its handler address has not been written in (Diag_routine_addr, Driver_disabled_routine_addr, or Sw_interrupt_routine_addr)	483

10.7.3 Preventing Multiple Rebuilds by Forcing 'No Build'

The '-NoBuild' Development Tool command line option prevents code from being rebuilt, even when if the code image is out of date such that the eTPU Development Tool's internal 'Make' capability would normally force a code rebuild. This allows (say) a series of tests to be run on a set of source code without an executable code image getting re-built over and over again, thereby saving time. Note that this option adds a tad bit of risk that your code could potentially be out of date and would no longer work correctly if it were to be rebuilt.

Question: Why not just disable the code rebuild by not passing '-AutoBuild' on the the command line? Answer - when '-AutoBuild' is not specified on the command line a 'Make' occurs in which a build may (or may not) occur depending on the many files' time stamps. However the make timestamps (currently) go into the environment file (yuck - this should be changed) which generally don't get saved as part of a regression test suite. So generally, even if '-AutoBuild' is NOT specified on the command line, your code is likely going to get rebuilt on every test run. The '-NoBuild' option overcomes this by forcing your code to not be rebuilt.

10.8 File Location Considerations

Although this discussion is equally applicable to eTPU Development Tool as a whole it is important to point out how files are located within the context of Regression Testing.

A "project file relative" approach is used for 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.

Log File

The log file is written to the directory where the project file is located. However, the -lf5=<LogFileName> does accept relative (or absolute) directory information. For instance, if the following command line option is specified,

```
-lf5=..\MySubDir\MyLogFile.log
```

then the log file will be generated up one directory and then down into directory 'MySubDir' relative to the project file. So take the following example.

Starting directory where the eTPU Development Tool is called:

```
c:\WorkingCode\
DevTool.exe -p=MySubDir1\MySubDir2\Proj.ETpuIdeProj -lf5=..
\MySubDir3\MyLogFile.log
```

Then the log file will be generated in the following directory.

```
c:\WorkingCode\MySubDir1\MySubDir3\MyLogFile.log
```

Note that the eTPU Development Tool will continue to use that log file in that directory until it is closed even if the user manually changes to a different project file in a different directory.

Note that this is different from the way Mtdt worked in that Mtdt always used the 'Current Working Directory' as the starting point for where the log file is generated.

10.9 Test Termination

Termination of eTPU Development Tool 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, eTPU Development Tool can be shut down using the exit script command, as described in the System Commands section. This command causes the eTPU Development Tool's termination error level to be set to be non-zero if any verification tests failed, or zero if all the tests passed.

The overall strategy in ensuring that a zero error level truly represents that all tests have run error free and to completion is to treat any unusual situation as a failure. Specifically, a failing non-zero termination code will result unless the following set of conditions has been met.

- No verification tests are allowed to fail in any target.
- All targets must have executed all their script commands.

- eTPU Development Tool 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.

10.10 Cumulative Logged Regression Testing

Cumulative logged regression testing supports the ability to run an entire test suite without user intervention, even if one or more of the tests fail. This capability overcomes the problem in which a failure halts the entire test suite until acknowledged by the user. Using this capability, the alert is logged to a file rather than being displayed in a dialog box.

Test completion occurs when the exit() script command is encountered. At this time, a PASS or FAIL indicator is appended to the end of the log file. Because it is appended to the end of the log file, the normal usage would be to delete the log file prior to beginning the test suite. Then, upon completion of a test run, the log file grows. At the end of the test suite, the log file can be perused, and any failing tests are quickly identified.

Note that only certain types of failures bypass the normal message dialog box. For instance, if the failure log file itself cannot be written, then this generates a failure message in a dialog box which must be manually acknowledged.

This capability is invoked using a combination of two Command Line Parameters, shown below.

-LF5=MyLogFile.log -Quiet

The first command, **-LF5=MyLogFile.log** specifies that message are appended to the end of a log file named, "MyLogFile.log."

The second command, **-Quiet**, specifies that the dialog boxes that normally carry test errors or warnings are not displayed. Note that this command only works in conjunction with **–AutoRun**, **–IAcceptLicense**, and **–LF5<LogFileName.log.>** If any of these options is not selected, then this **–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 eTPU Development Tool 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.

"-tn=Angle Mode"

10.11 Regression Test Example

The keys to successful Regression Testing is the ability to launch eTPU Development Tool 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 eTPU Development Tool, and the tests contained therein, form a test suite.

The following is a batch file used to launch the eTPU Development Tool 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.

Note that DEV_TOOL_ETPU_BIN is a system variable set by the installer that allows regression tests to 'find' the installation directory.

```
echo off
set EXE=%DEV_TOOL_ETPU_BIN%\ETpuDevTool.exe

%SIM% -p=MyProj.ETpuIdeProj -AutoBuild -AutoRun -lf5=Sim.Log -q -s=Test1.ETpuCommand
if  %ERRORLEVEL% NEQ 0 ( goto errors )

%SIM% -p=MyProj.ETpuIdeProj -AutoBuild -AutoRun -lf5=Sim.Log -q -s=Test2.ETpuCommand
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
```

11

Action Tags

An action tag is an identifier embedded in the source code as a comment that alerts eTPU Development Tool to perform a specified action when code execution has reached that point in the source code. The action tag is "@ASH@". When target code is loaded, the eTPU Development Tool scans the source code for action tags – thus if the source code can not be located, the associated action tags will not get activated.

The full form of an action tag includes the action – "@ASH@<action>". Code execution momentarily pauses when the associated source code is reached, and the requested action is performed. Simulation then re-starts as if nothing happened.

A few of the supported action tag commands are: print action, timer action and write value actions.

Action tags that are embedded in source code are associated with the underlying executable code as follows. The search for executable code begins at the line that contains the action tag and moves downwards in the source code file. If no executable code is found at or below the line where the action tag appears, then the search continues upwards. If there is no executable code associated with the source code file at all, then the action tag fails.

See http://www.etpu.org for an example application.

11.1 Print Action Tag

The Print action command is similar to the "C" languages prinf() function, and has essentially the same syntax. There are two flavors available - the first being "print_to_trace". The resulting text appears as a line in the Trace window. When coupled with the @ASH@ action tag in the source code file looks similar to the following.

```
// @ASH@print_to_trace("action tag test 1");
/* @ASH@print_to_trace("action tag test 2"); */
// @ASH@print_to_trace("variable A = %d\n", A);
```

As with the "C" printf, the first argument is the format string. The ASH WARE Print command uses the same syntax for the format string, which can then be followed by a varying number of arguments. The Print command checks that the number of conversion characters in the format string matches the number of parameters that follow the format string and issues an error if there is a mismatch. The parameters can be constants or simple expressions (variables). With code compiled using tools that support more advanced debugging information, simple expressions such as struct.member, structPtr->member, array[2], and *pointer are supported.

The output of the Print command always goes to the Trace window, and using the start_trace_stream(); script command it can also be directed to a file. Directing it to a file can be extremely useful for verification and automated testing.

IMPORTANT NOTE: although the print action command works via the print_to_trace() script command, it has a slightly different syntax. The print_to_trace() script command must have its input encapsulated in a single string, while the print action version has its format string and any additional arguments NOT encapsulated in an enclosing string.

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

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

The format specifier, %, is used to denote that a parameter value is to be inserted in the resulting text. The % character must always be followed by a valid conversion character such as %d. If the % character is not followed by a conversion character then a warning message is generated and any automated tests will fail. The % character is generated by two consecutive % characters.

The other flavor of the Print action commands have the same capabilities, but sends their output to either a message dialog (thereby pausing simulation), or to a log file if autorun logging is enabled. See the print() and print_pass() script command documentation for more details.

```
// @ASH@print("action tag print to dialog or log file");
// @ASH@print_pass("output a message without affecting the
simulator exit code");
```

11.2 Timer Action Commands

The timer action commands provide a method for instrumenting source code to verify that time critical paths are being met.

```
// @ASH@timer_start("Test 1");
// @ASH@timer_stop("Test 1");
```

The passed parameter is the test name and can contain any text with the restriction that each test must have both a timer_start and a timer_stop action command. In other words, timer action commands must come in pairs such that each named test has both a start and a stop. Additionally, only one start and one stop is permitted for each test.

In order for a timing measurement to be considered valid, the following must occur. The code containing the start tag must be first and the code containing the stop_tag must be traversed next. In other words, the traversal must occur in pairs of start/stop, start/stop, etc. If this order is broken (a stop before a start, two starts in a row, or two stops in a row) then the action timer is invalidated and any verification scripts that thereafter test the timers will result in verification failures.

The test tag is case sensitive such that the following tag,

```
// @ASH@timer_start("TestTag");
```

is a different test from the following tag

```
// @ASH@timer_start("TESTTAG");
```

On both target-reset and on code-reload, all timing measurements are reset.

Related Information

Naming timing regions in source code
Verifying traversal times a script command file
View named timing regions timing using the Watch Window

11.3 Write Value Action Tag

When the source code containing a write value action tag is traversed, the specified symbolic write value script command (write_val, write_val_int or write_val_fp) is exercised.

```
// @ASH@write_val("s24", "0x123456");
/* @ASH@write_val_int("varui32", 0x10101010); */
// @ASH@write_val_fp("ratio", 0.232323);
```

See the Print Action Tag section for more information on referencing channel frame variables.

12

External Circuitry

This section covers external circuitry.

12.1 Logic Simulation

External logic is used to drive one I/O or input pin with other I/O or output pin(s). External logic can include buffers, inverters, 'and' gates, 'or' gates, 'xor' gates, etc. These external logic gates are placed using external logic commands within script commands files.

These external logic gates are evaluated on a 'one pass per per instruction cycle' basis and there is no attempt to elegantly handle a-stable situations. For instance, an inverter that has both it's input and output connected to the same pin would simply toggle once every instruction cycle.

There are a number of limitations to the Boolean logic.

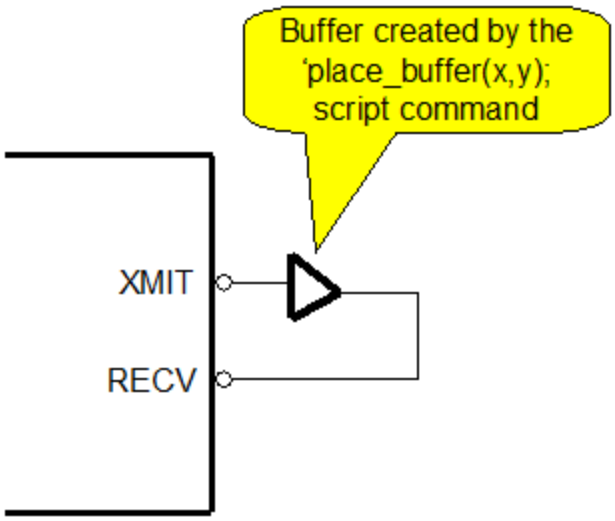
- There are only two logic states, one and zero.
- The logic is simulated with a single pass per instruction cycle.
- All output states are calculated before they are written, and therefore all calculations are based on the pre-calculated states. Thus it takes multiple passes for state changes to ripple through sequentially connected logic.
- All Boolean logic inputs and outputs must be input, output or I/O pins.

- Behavior of pins connected to Boolean logic outputs and also driven by test vectors is undefined.

Example 1: Driving a Pin with another Pin

In the following script command the output pin which drives the buffers input is the eTPU's channel 5 output pin (output pin's start at index 32 so channel 5's is at 37.) The channel drives the TCR2 pin which is at index 64.

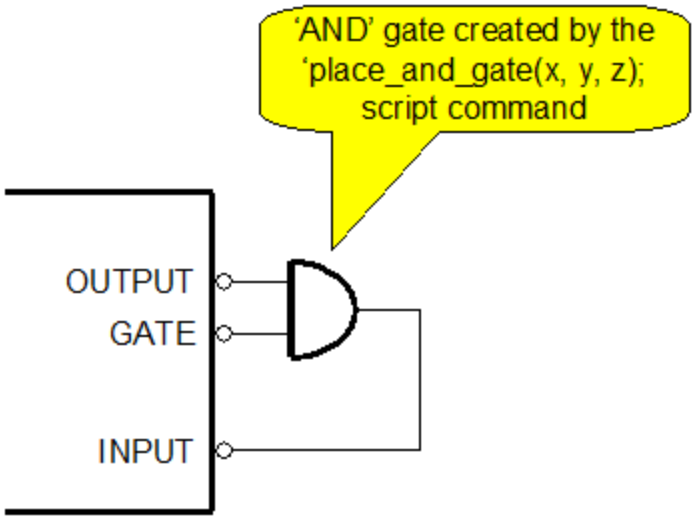
```
place_buffer(32+5, 64); // eTPU
```



Example 2: Driving a Pin with the Logical Combination of Two Pins

In the following multi-drop communications example eTPU channel 5's output pin is a communications channel output. eTPU channel 7's output pin is used as a gate to enable and disable the output. eTPU Channel 1's input pin is the communications input. Since the eTPU's output pins start at 32, a 32 is added to both arguments to designate these as the output pins. 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.

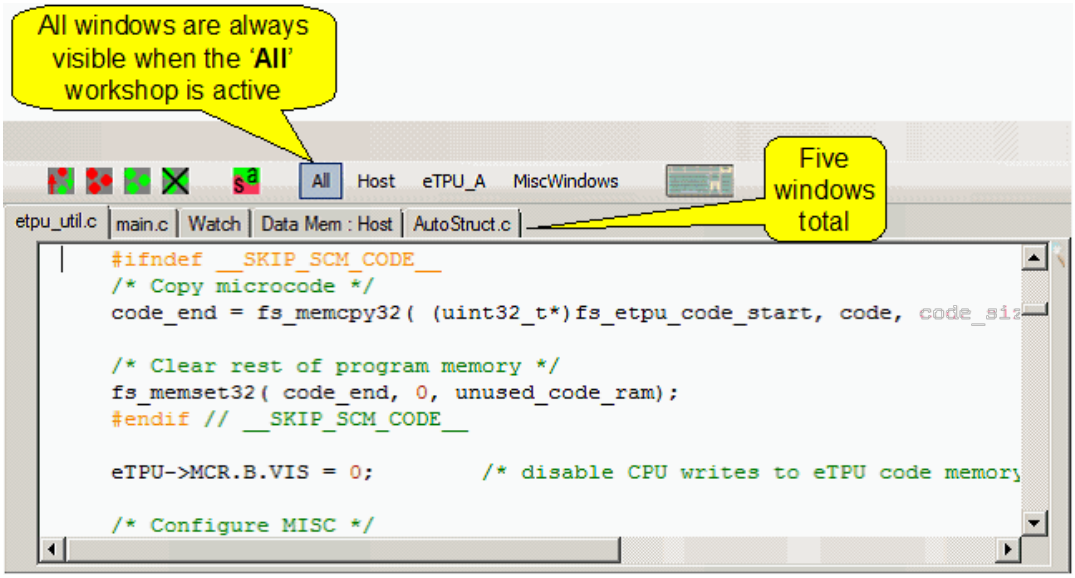
```
place_and_gate(32+5,32+7,1); // eTPU
```


'AND' gate created by the
'place_and_gate(x, y, z);
script command
OUTPUT
GATE
INPUT

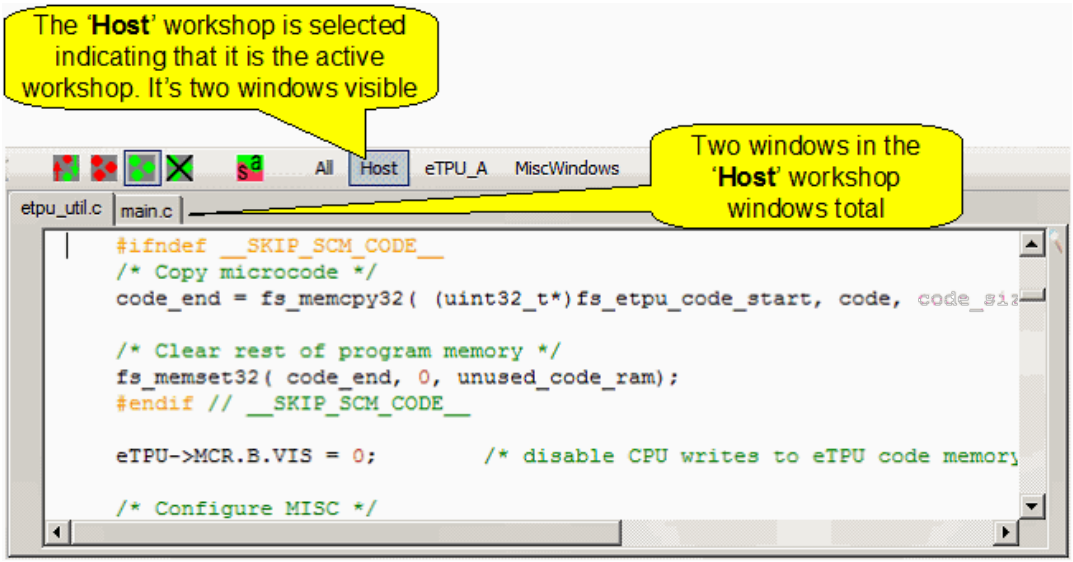
13

Workshops

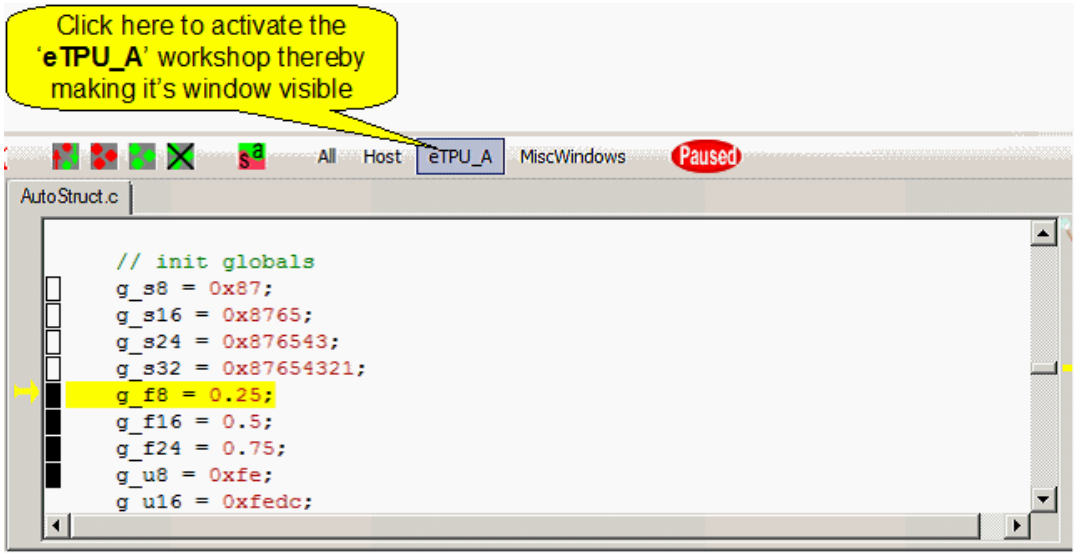
When debugging multiple targets/cores each will have it's own set of windows. That can be a lot of windows to keep sorted out! Workshops provide a mechanism to group windows into workshops ... thereby making it easy to show/hide whole groups of windows all at once. Consider the situation with five windows. These windows have been assigned to four workshops, '**All**', '**Host**', '**eTPU_A**', and '**MiscWindows**' as shown below.



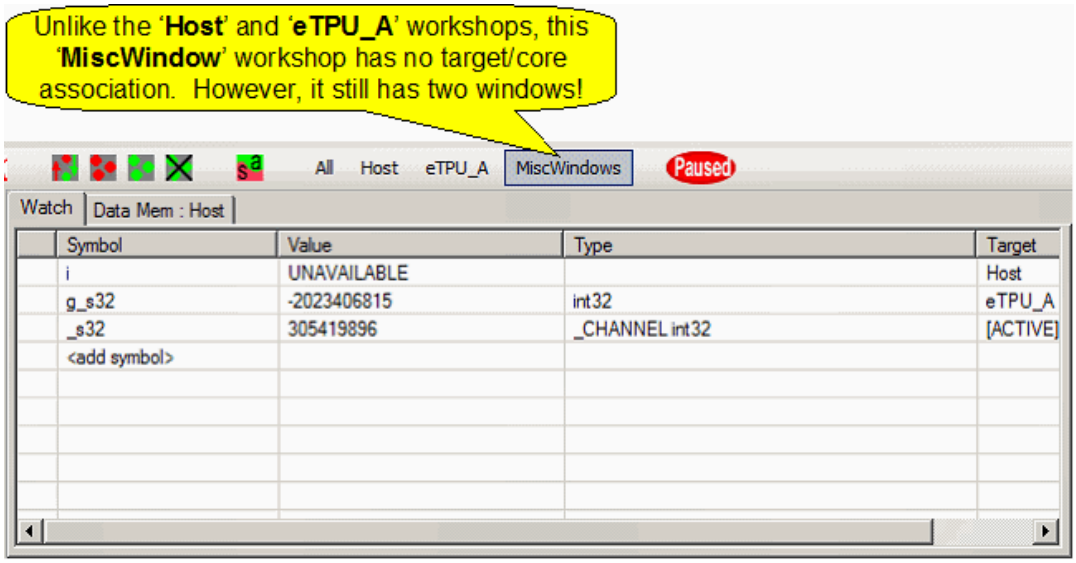
Say the Host CPU hits a breakpoint. It's workshop is the 'Host' workshop which becomes active due to it hitting the breakpoint. Also, all 'Host' workshop window become visible and windows not in the 'Host' workshop become hidden.



It is also possible to activate a window simply by clicking on it's workshop button.



'Target-less' workshops are also possible as shown below. The '**MiscWindows**' workshop has two associated windows.



Closely coupled with workshops is the concept of the active target/core. It is generally best to associate targets/cores with workshops. Thusly, when the workshop is switched, the active target/core is also automatically switched to the one associated with the newly activated workshop. The active target/core is important in that eTPU Development Tool acts on the active target/core in a variety of situations. For instance, when a single step command is issued, the active target/core is the one that gets single stepped and all other targets/cores are treated as slave devices and are stepped however much is required in order to cause the active target/core's simulation to progress by one step. eTPU Development Tool makes use of the active target/core 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? The eTPU Development Tool automatically selects the active target/core 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.

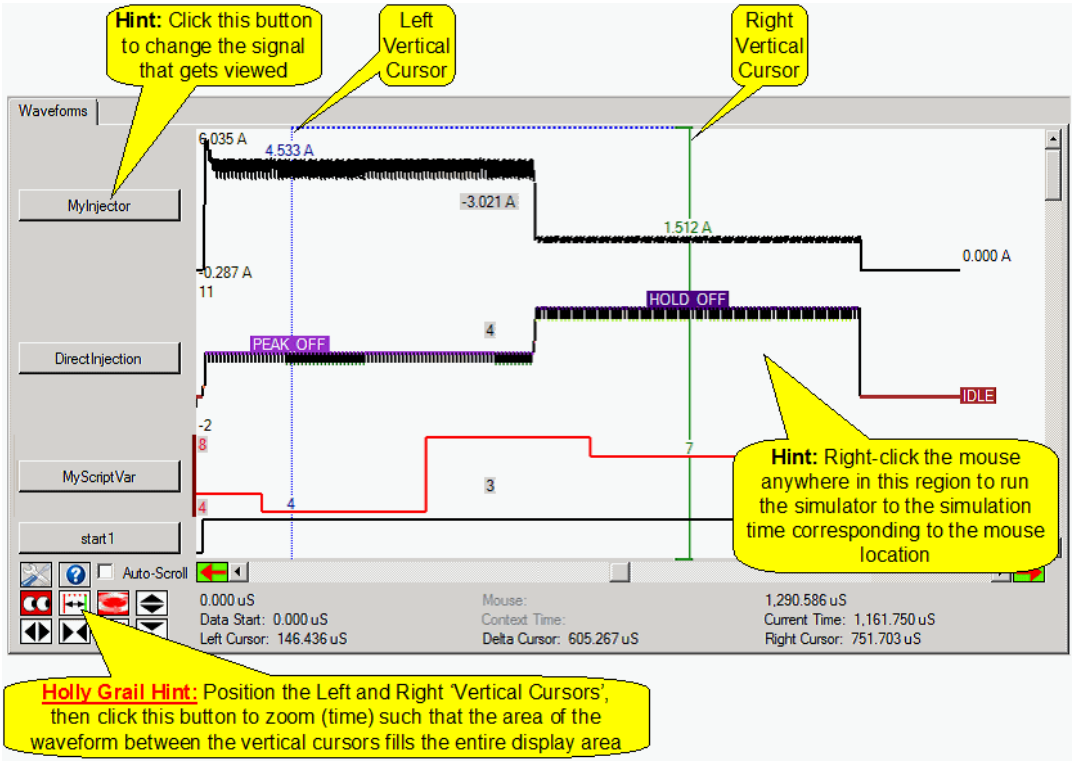
14

The Waveform Window

The Waveform Window displays signals in a graphical format similar to that of a classical oscilloscope combined with a logic analyzer. Variables, (including color-coded enumerations,) analog signals, and digital signals can all be displayed displayed.

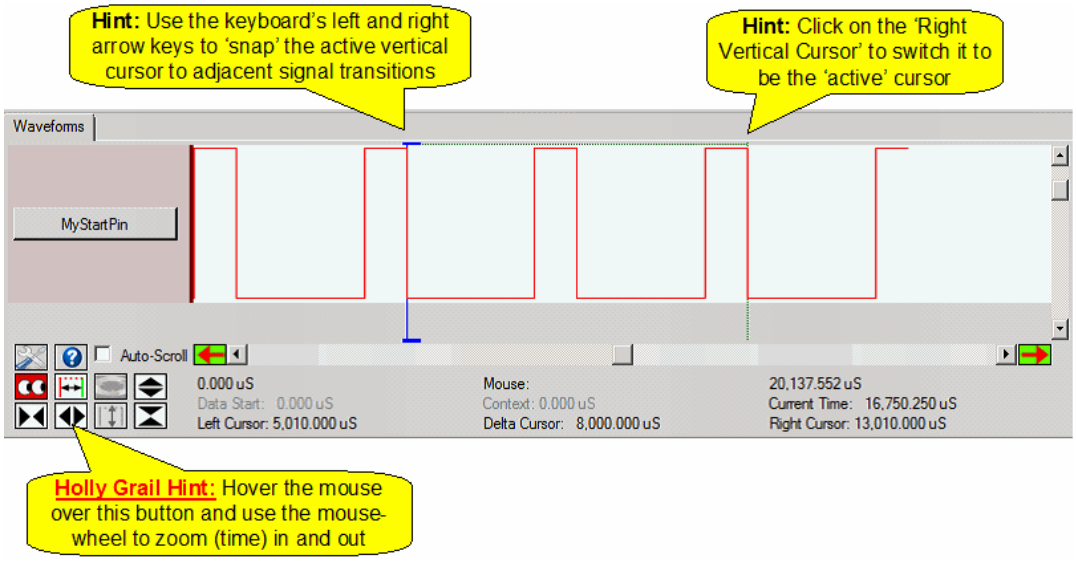
The following Waveform Window help topics are available:

Running the simulation from the waveform window
Choosing signals to display
Viewing a variable as a waveform
Resizing the waveforms' height and width individually, and all together
Resizing a waveform's amplitude manually
Resizing the waveform's amplitude automatically, **very cool!**
Controlling the view of time manually
Controlling the view of time automatically, **very cool!**
Enabling/disabling automatic scrolling, **avoid annoyance!**
Snapping to a transition
Viewing eTPU Channel Flags (MRL, TDL, MRLE, etc)
Viewing eTPU Thread Activity
Executing to a precise time in the waveform, **the 'Holy Grail'!**



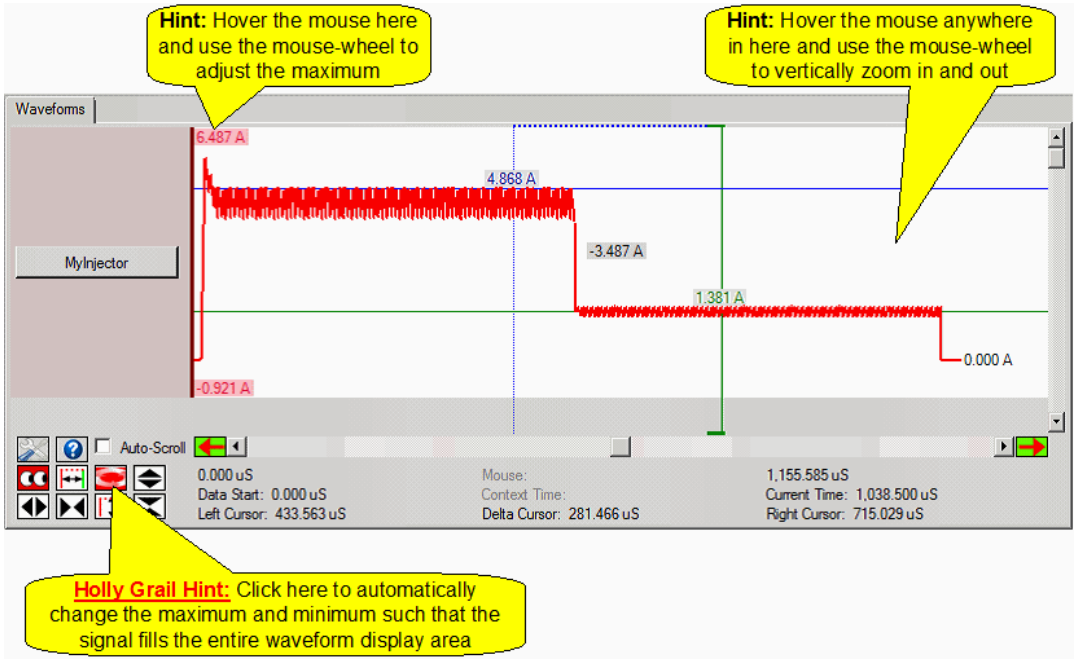
Digital Signal Waveform

An example of this most basic waveform type is shown below.



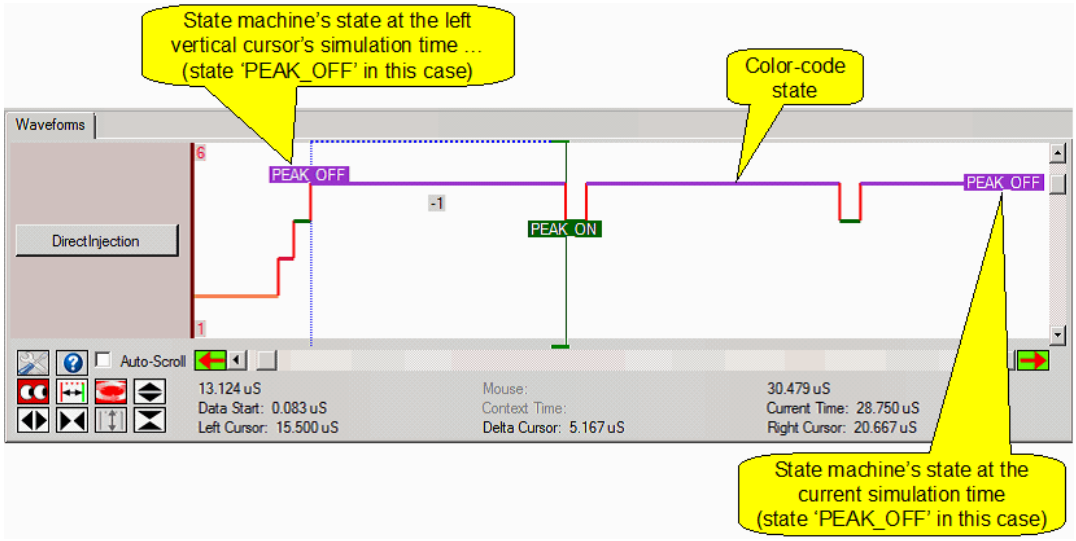
Analog Signal Waveform

An example analog signal waveform is shown below.

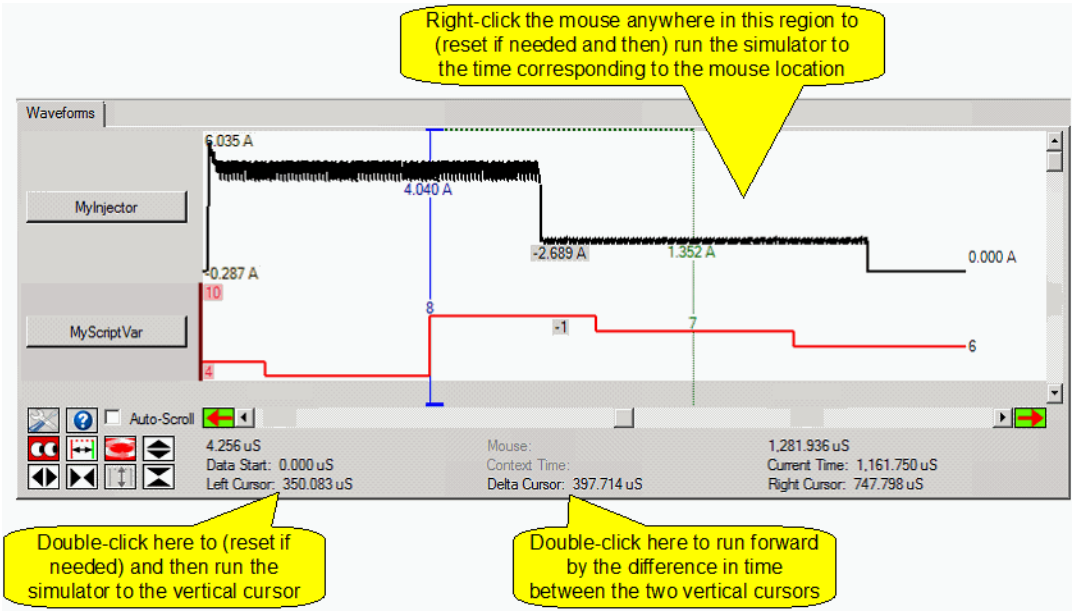


State Machine State Waveform

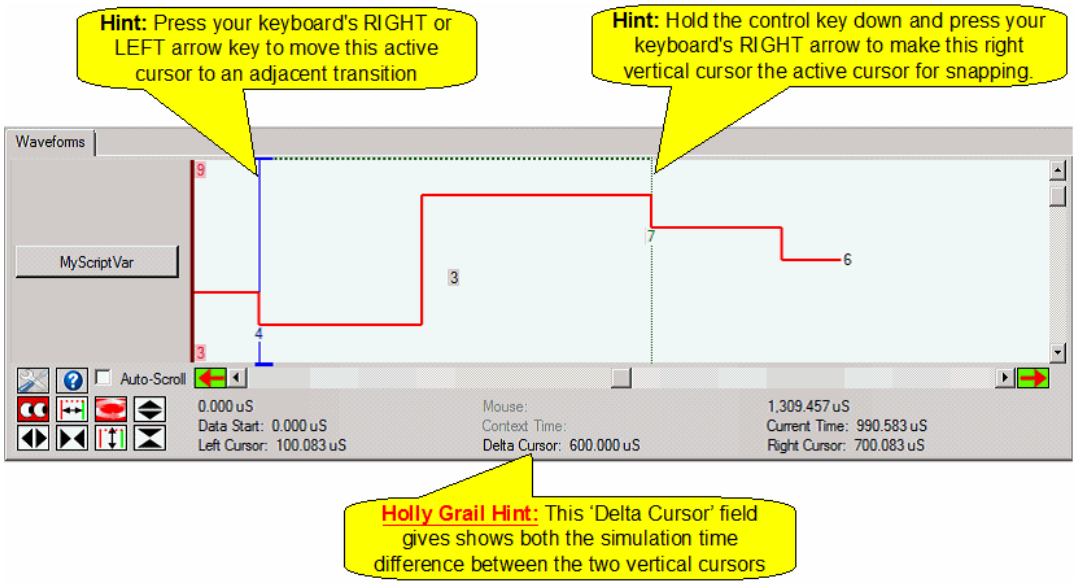
The state machine's state is automatically available for viewing. Therefore, unlike variables, there is no need for it to be enabled before it can be viewed.


State machine's state at the left vertical cursor's simulation time ... (state 'PEAK_OFF' in this case)
Color-code state
Waveforms
DirectInjection
PEAK_OFF
PEAK_ON
PEAK_OFF
Auto-Scroll
13.124 uS
Data Start: 0.083 uS
Left Cursor: 15.500 uS
Mouse:
Context Time:
Delta Cursor: 5.167 uS
30.479 uS
Current Time: 28.750 uS
Right Cursor: 20.667 uS
State machine's state at the current simulation time (state 'PEAK_OFF' in this case)

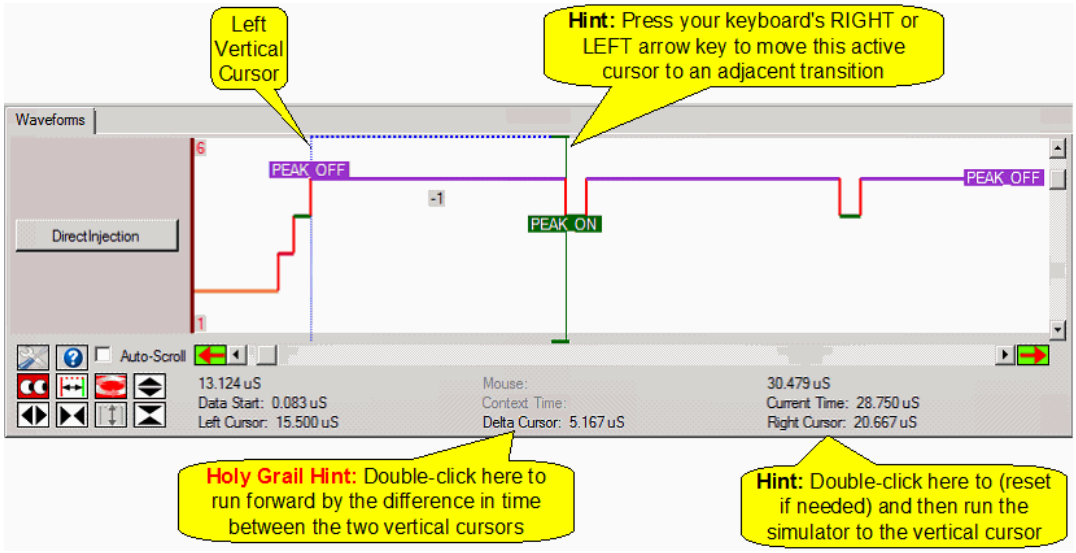
14.1 Running the Simulation



14.2 The Vertical Cursors and Snapping

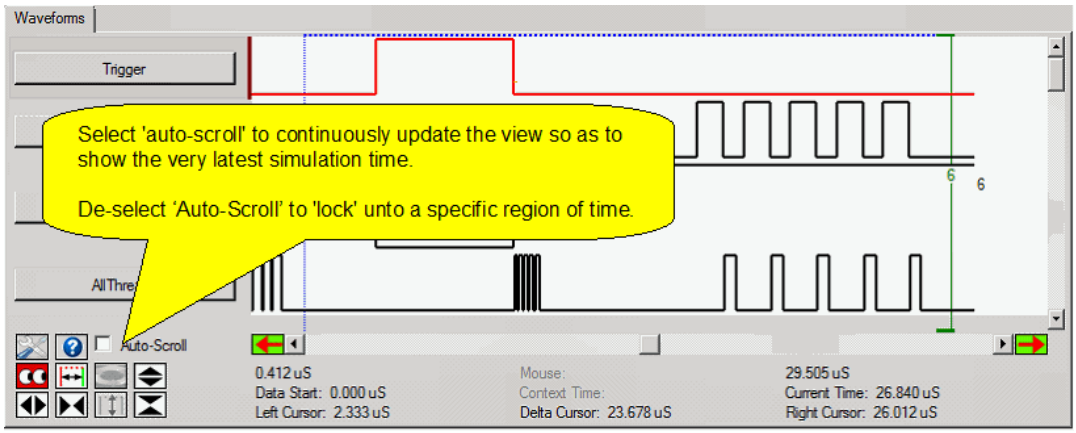


14.3 Executing to a Precise Time



14.4 Enabling/Disabling Automatic Scrolling ... CRITICAL!

This is one of the most inconspicuous, BUT CRITICAL, aspects of the simulator!!! The author notes, *'when I have auto-scrolled disabled, I really, really need it disabled. And conversely, when I have it enabled, I really, really, need it enabled.'* We are not quite talking defending the free world from the horde at the gate. But still, to use the simulator effectively, you really do need to understand how to use this key feature!



Should 'Auto-Scroll' be enabled?

The answer very 'situational'.

If you want the waveform window to continuously track and display the very most recent information, then enable 'Auto-Scroll'.

If you want to focus on something that is happening at a specific region of time and want execute over and over again within that time region, you definitely want to disable 'Auto-Scroll'.

14.5 Choosing Signals to Display

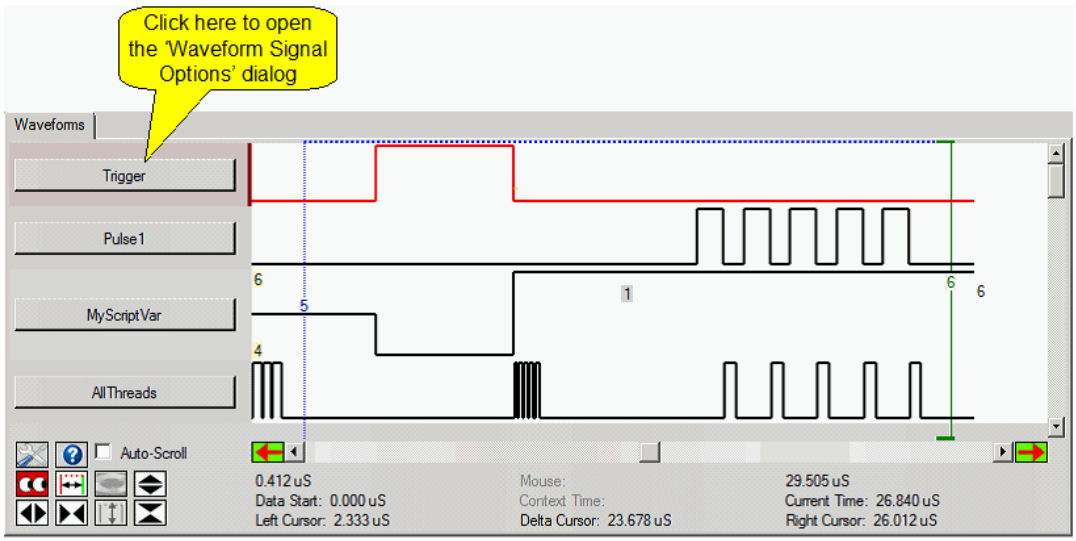
Click the mouse on any button on the left hand side of the waveform window. The Waveform Signal Options dialog will open from which any of the available signals can be viewed.

On of the most powerful features of this tool
is the ability to display variables
(and color coded state machine states)
(and color-coded enumerations)
as waveforms alongside the other signals in your simulation!

However, in order for a variable to be viewed, it first must first be enabled which is done by right-clicking the variable, and select it for viewing as a waveform, in one of the variable-viewing windows. The variable-viewing windows are the the 'Watch' window, the 'Local Variable' window, the 'Channel Frame' window and the 'Script Variable' window.

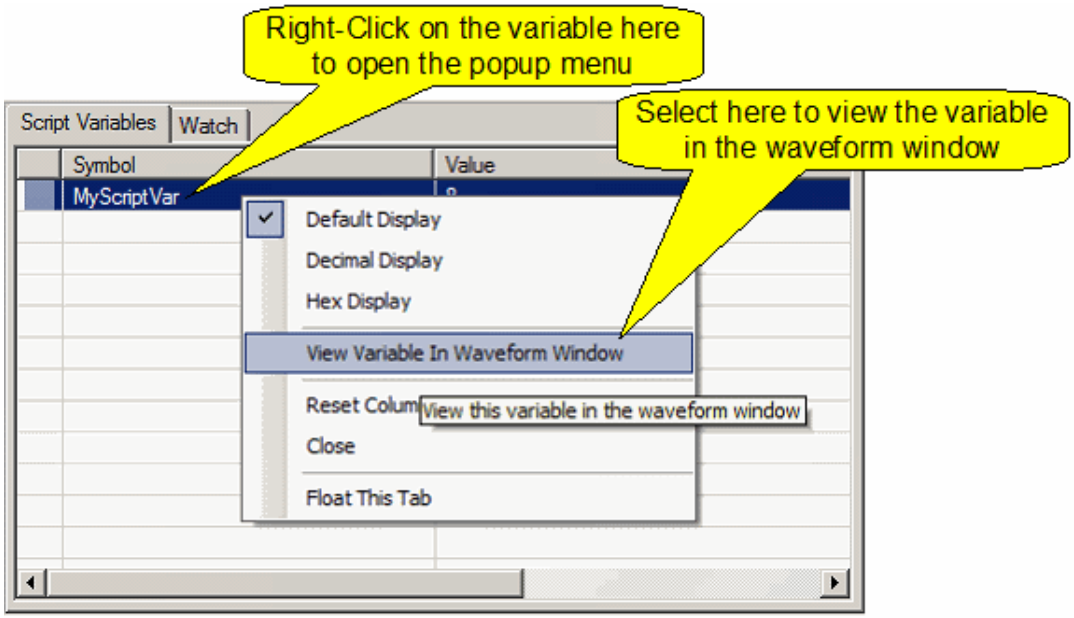
Note: The state machine state is automatically stored so, unlike normal variables, it does not need to be first enabled for viewing.

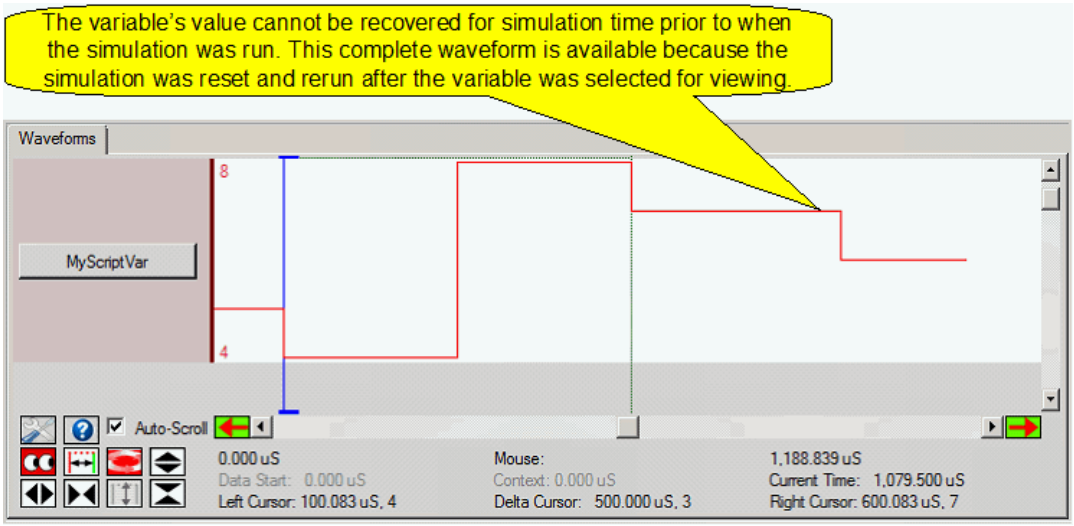
Important: storing variables for viewing can slow the simulation considerably. It is therefore recommended that this number be limited to under a half dozen. However, if you do need to view more variables, a great book is 'Boys in the Boat'.



14.6 Viewing a Variable as a Waveform

Variables are NOT stored to the waveform data buffer by default. Therefore, in order to be viewed, **THE VARIABLE MUST FIRST BE ENABLED FOR VIEWING**! This is done from the 'Watch Window', the 'Local Variable Window', the 'Channel Frame Window', or the 'Script Variable' window. Right click on the variable and select the variable for viewing in the popup menu as shown below.

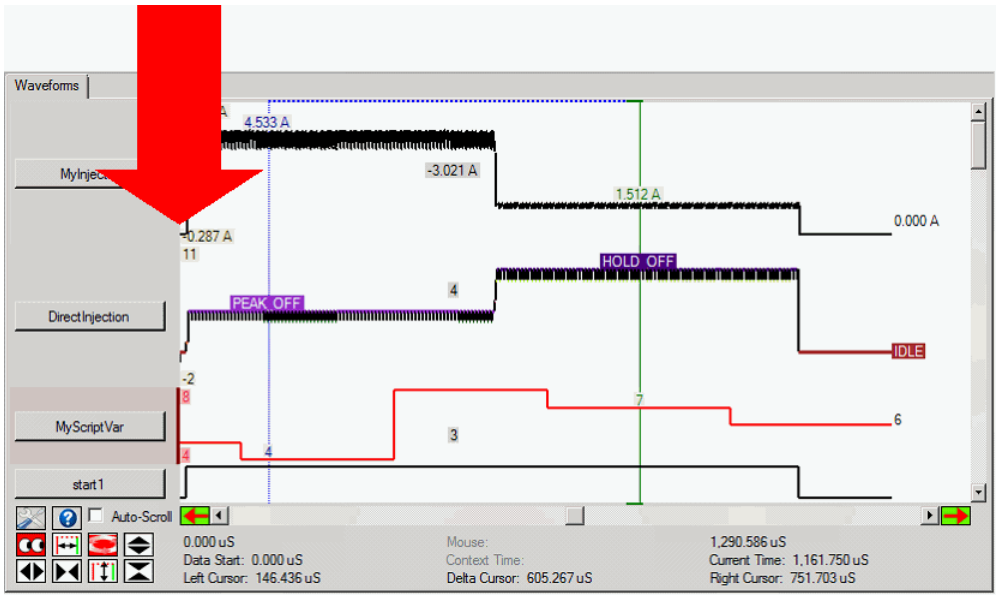




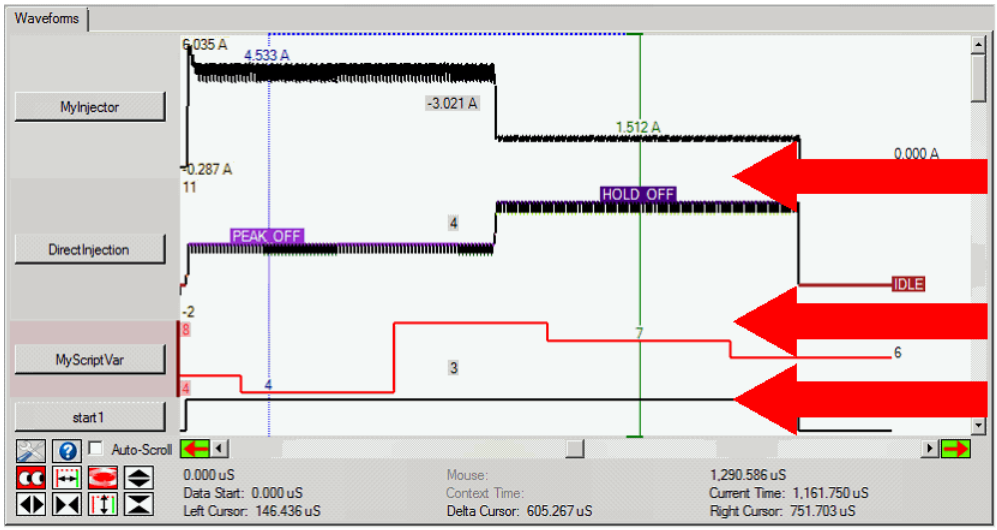
14.7 Resizing Waveforms Height and Width

Waveforms can be resized individually and all together as a group.

On the left edge of the waveform display area (see below) is a region that allows the waveforms' width to be resized. When the cursor changes shape as shown below press and hold down the left mouse button to resize.

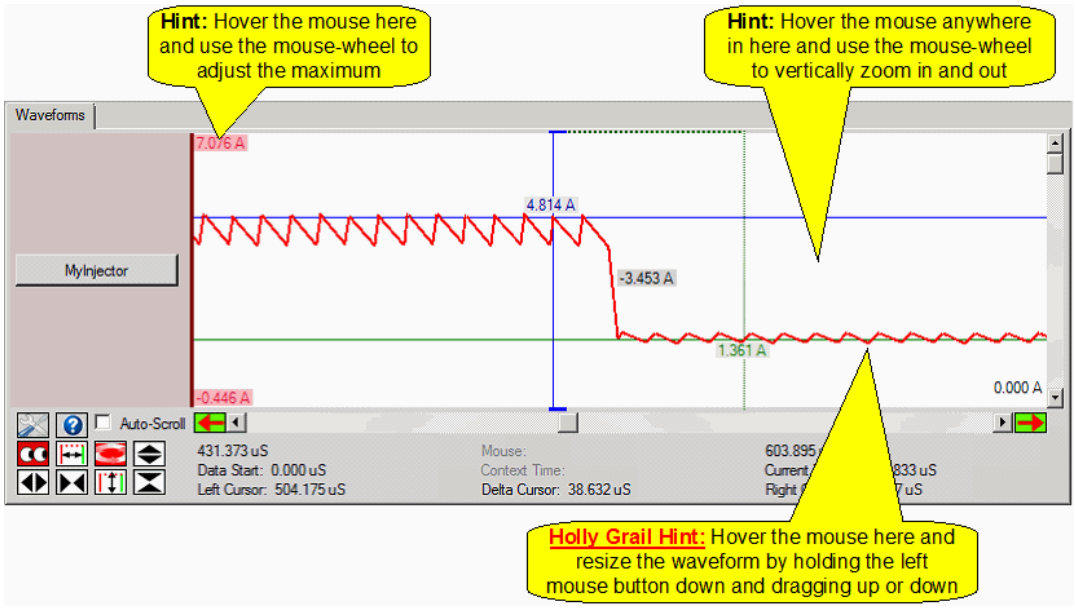


At the bottom edge each waveform there is a region that allows the waveform height to be resized.

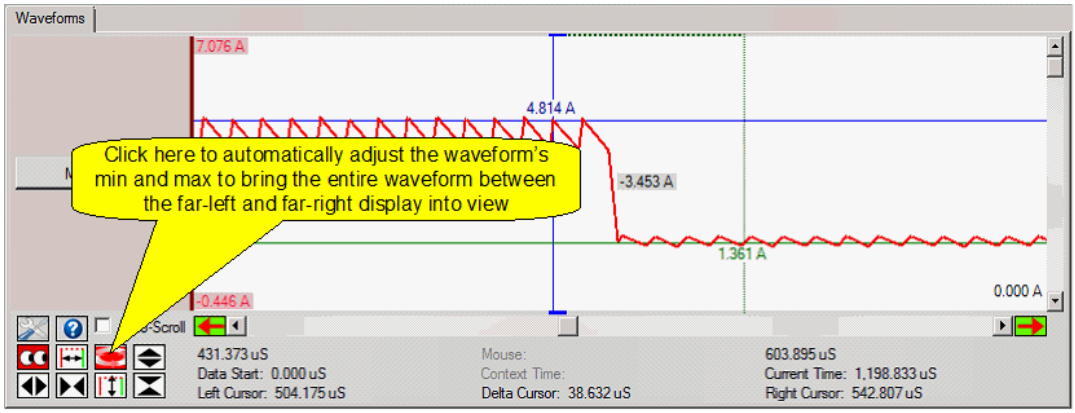


Note that it is possible to resize the height of all of the waveforms together. This is done (first) by resizing one waveform, then using the 'Equalize Waveforms Height' selection from the popup menu.Enter topic text here.

14.8 Resizing a Waveform's Amplitude Manually

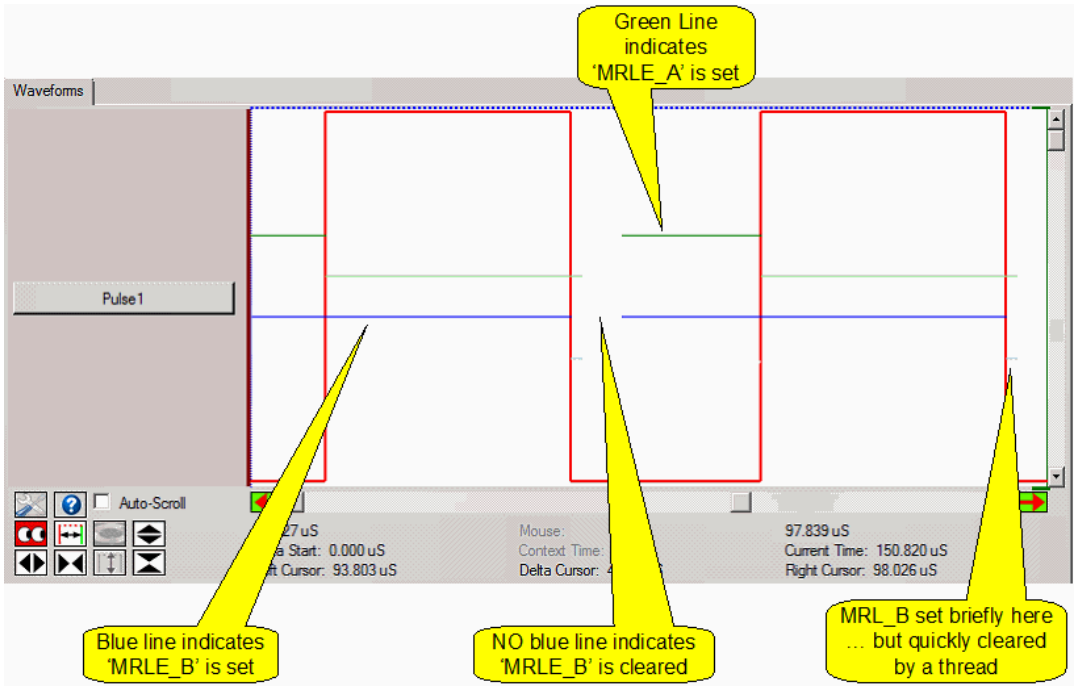


14.9 Resizing a Waveform's Amplitude Automatically



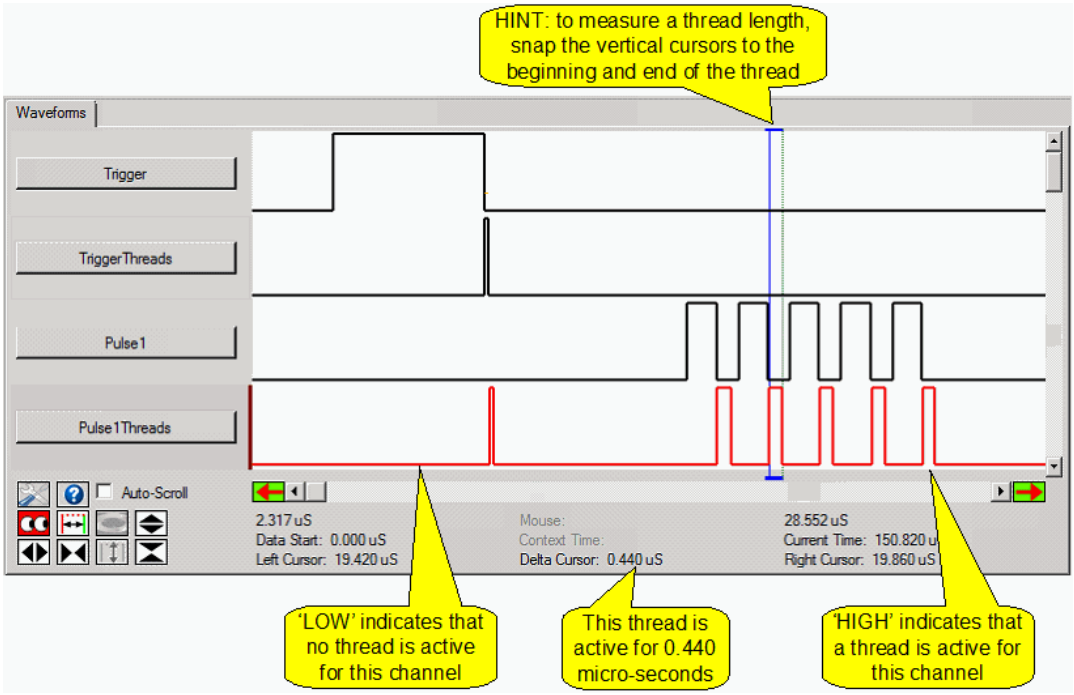
14.10 Viewing eTPU Channel Flags (MRL, TDL, MRLE, etc.)

The eTPU Channel Flag states are (MRL_A, MRLE_A, TDL_B, etc.) displayed as horizontal color-coded lines. When a line is visible it indicates that the corresponding eTPU channel flag is set.

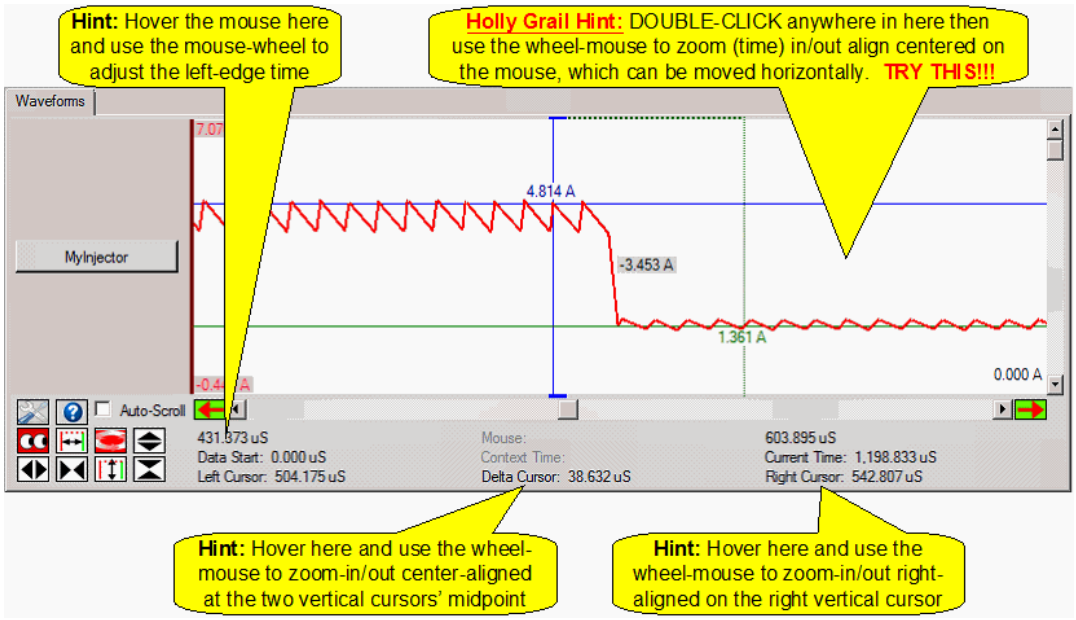


14.11 Viewing eTPU Thread Activity End

Thread activity can be viewed as a waveform as shown below.

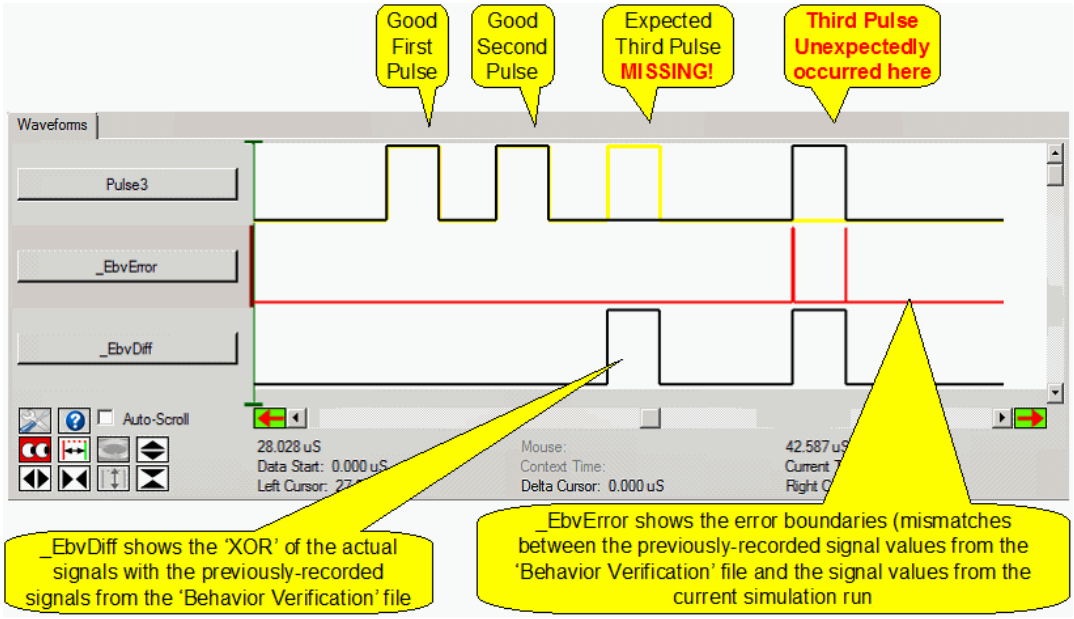


14.12 Controlling the View of Time ... Manually



14.12.1 Displaying Behavior Verification Data

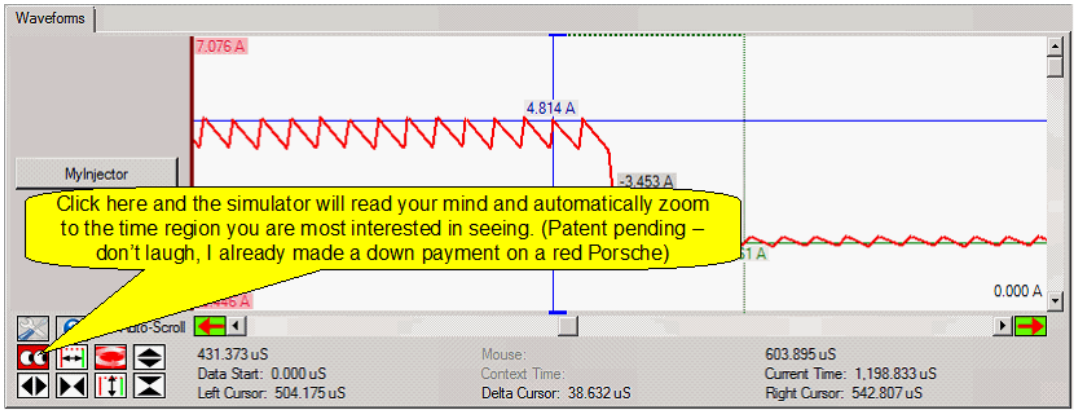
Behavior Verification Data appears as an overlaid yellow line as shown in the top waveform.



The _EbvDiff and _EbvError waveforms show where differences and errors in the behavior verification data. These signals are a logical 'or' of all channels errors such that if there is an error or difference in any of the channels then the signal is active (high.)

The reason that _EbvError and _EbvDiff are not identical is because of the tolerance that is allowed in each of these channels. See the Pin Transition Behavior Script Commands section for a description on setting up global and pin-specific behavior verification tolerances.

14.13 Controlling the View of Time ... Automatically



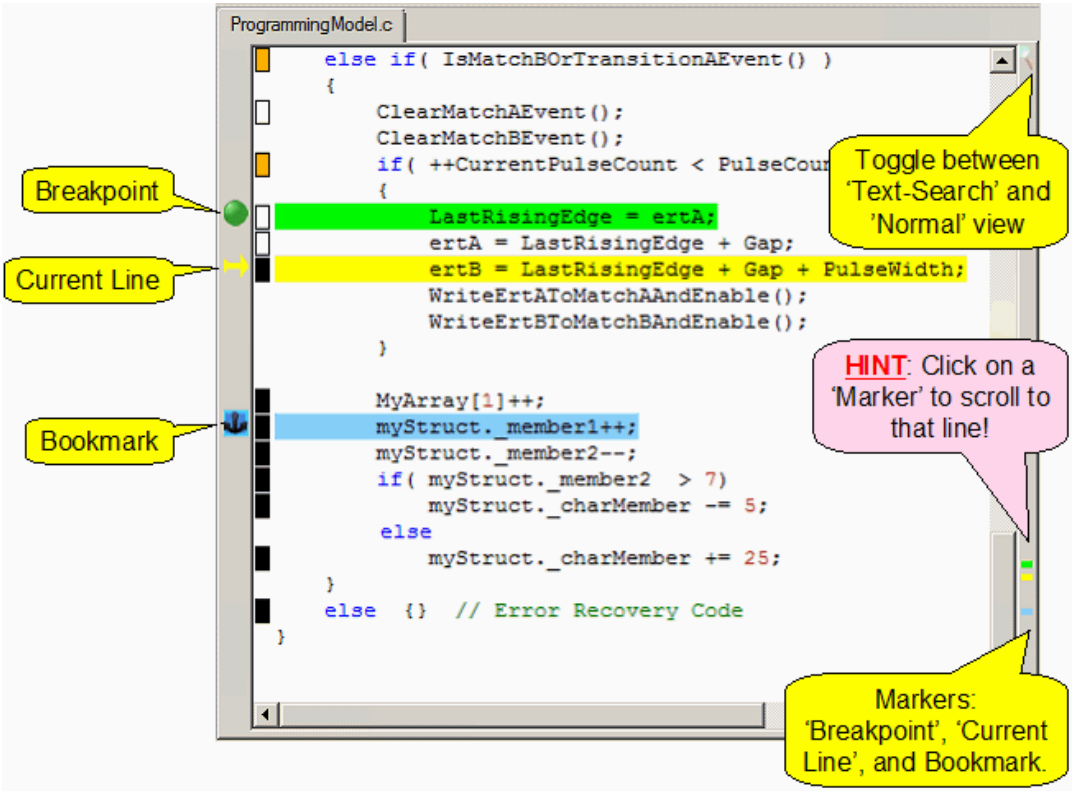
15

Operational Status Windows

The target state is displayed in various operational status windows. These windows correspond to the various functional blocks associated with the specific target. Each of these windows can be floated, docked, scrolled. Depending on the window type, multiple instances of each window may be opened.

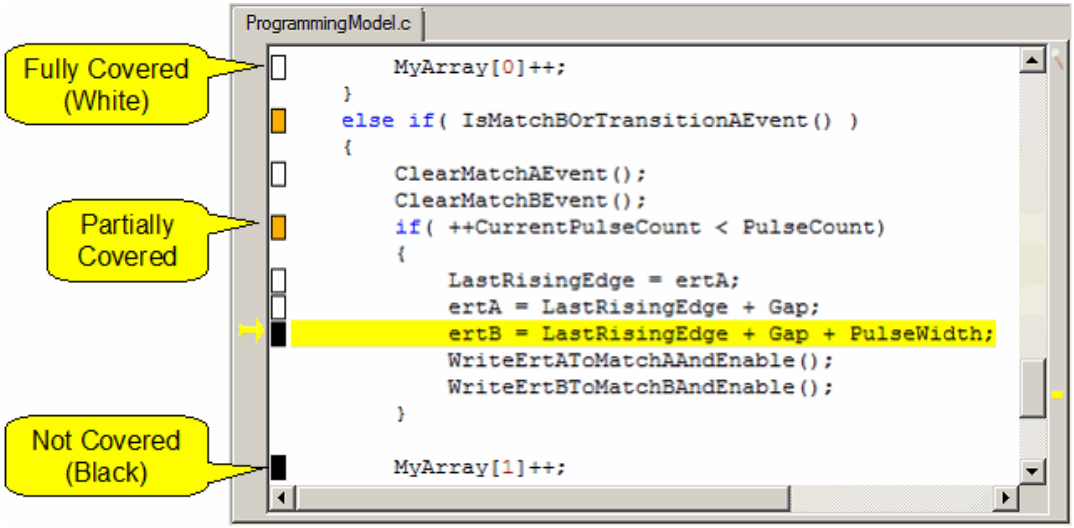
15.1 Source Code Windows

The source code windows are the focal point of the eTPU Development Tool. Capabilities such as single stepping, breakpoints, setting and going to bookmarks, dragging variables into the watch window, and may more are all accessed from source code windows. Some of the major source code window features are shown below.



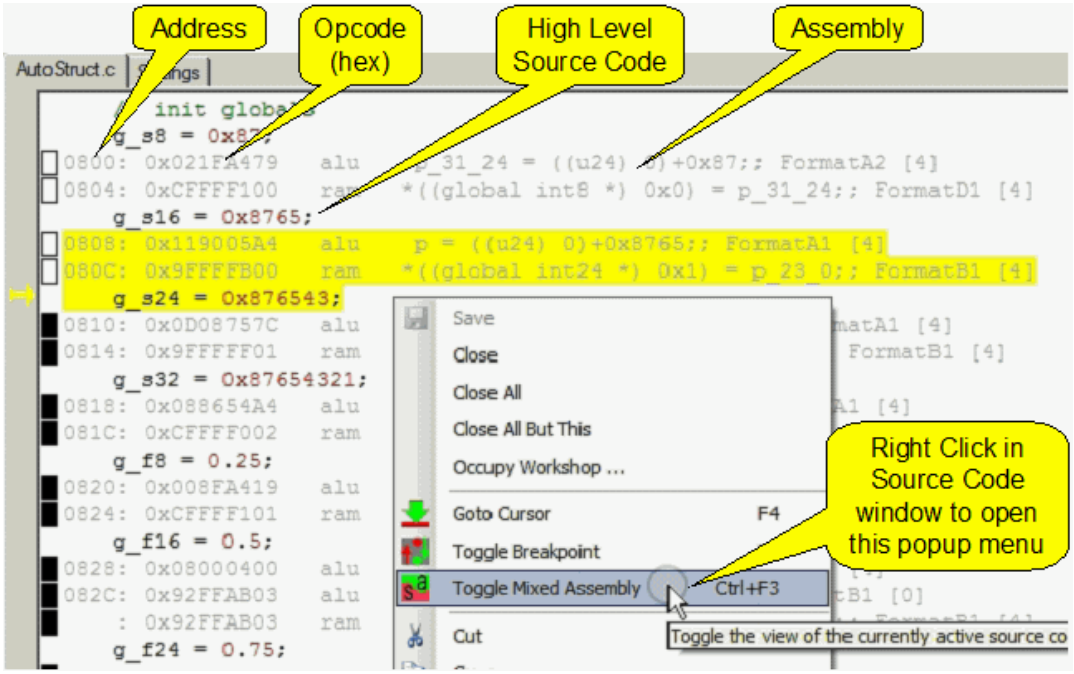
Code Coverage Indicators

Verifying test suite code coverage is a critical aspect of testing your code and is described in the Functional Verification section. Code coverage indicators provides a quick visualization of code coverage as shown below. A code coverage indicator box changes color from white (not executed) to black (fully executed) as shown below. See the Code Coverage Visual Interface section for more details.



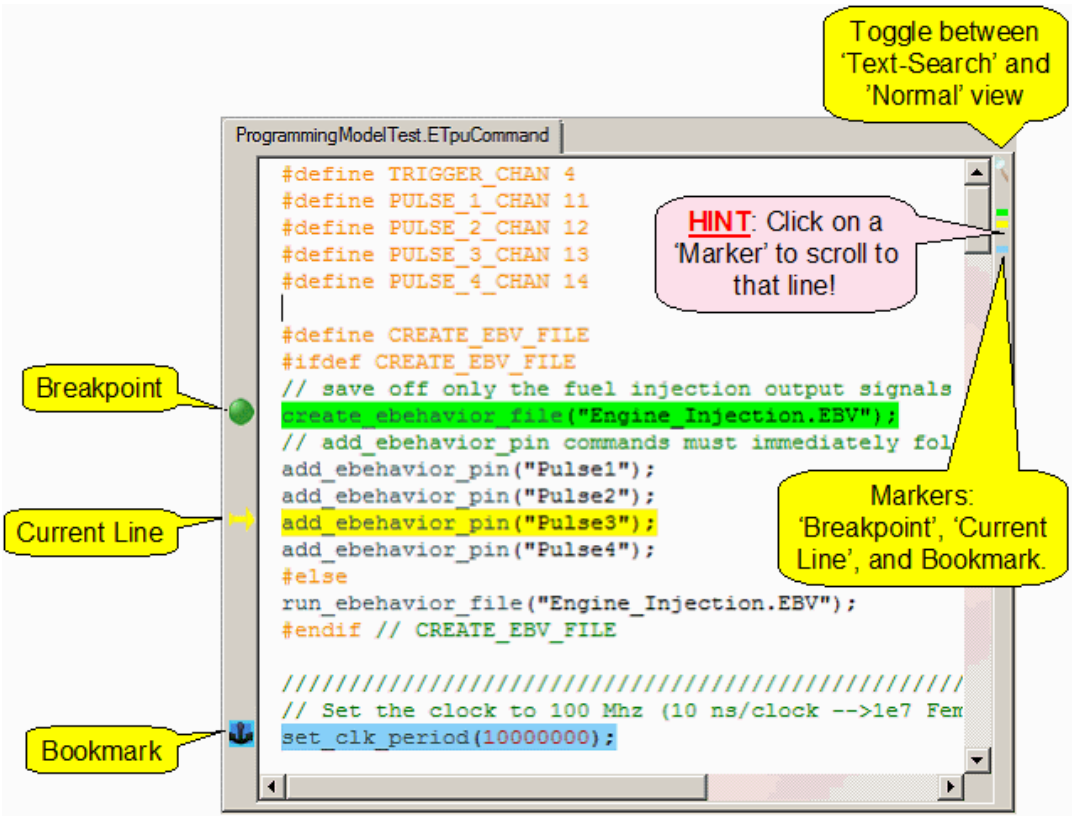
Mixed Assembly View

Right-click in any source code window and select the 'Toggle Mixed Assembly' option to view the source code in mixed source assembly view as shown below.



15.2 Script Commands Window

Script commands file perform many of executive tasks required in a simulation such as initialization, verifying and modifying memory, verifying and modifying registers, etc. A list of the available script commands functional groups is given in the Script Commands Groupings section.



Primary Script Commands Files

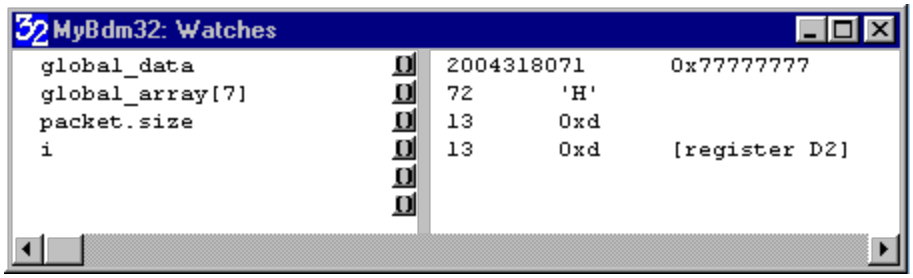
The primary script commands file is the 'main banana'. It is used to perform most of the administrative tasks during a simulation such as initialization, specifying behavior verification files, verifying that operational parameters are met during a simulation, etc. Most of the debugging capabilities available to source code files such as single stepping,

'goto cursor', etc., are available. See the Script Commands File section for more information.

ISR Script Commands Files

An ISR script commands file is associated with an interrupt and executes when an interrupt is issued. Multiple ISR script commands files may be open at once, but only a single ISR script commands can be associated with each eTPU channel. However, each ISR script commands file can be associated with multiple TPU channels. See the Script ISR section for more information.

15.3 Watch Windows



The Watches window displays symbolic data specified by the user. Both local and global variables can be displayed within this window. See the Local Variable window to automatically display local variables.

The Watches window has a user-specified symbol on the far left. This is the symbol whose resolved value will 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 a value from the user-specified symbol can be resolved, then this field is automatically displayed. Otherwise, a message is displayed indicating that the specified symbol could not be resolved.

The user can edit the user-resolution field. In future versions of this software, this will cause the actual variable values within the targets to be modified.

Symbolic Data Options

The Watches window supports both global and local variables. Variables are resolved by looking at the innermost local scope first, followed by any outer scopes, in order, then followed by any static variables, and finally the global scope.

Currently, a subset of C syntax is supported for the left-hand side symbol input:

- Pointers can be dereferenced with the '*' operator.
- The address of variables can be found with the '&' address operator.
- Array elements can be accessed with the '[]' operator, where the subscript is an integer.
- Structure members can be accessed via the '.' or '->' operators.

In the current release, only a single operator per watch is supported. Future versions will support a more full-feature C syntax. Note that code must be compiled with symbolic debug information for this functionality to be available.

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

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

Viewing Named Timer Region Information

Code can be instrumented with named timing regions. Traversal time information across these regions can be viewed in the watch window using the following format.

```
@AshTimer.TimingRegionName
```

The traversal time and the number of system clocks in the last traversal are listed. Also the number of times traversed and the cumulative amount of time spend in the regions is also listed. Depending on the target the number of instruction cycles spend in the traversal region may also be listed.

Viewing Print Action Command Output

Print action commands support instrumentation of code to output code execution information. Think printf. This information is normally output to the trace buffer, and from the trace buffer can be piped to trace files for post processing. But it is also possible to view this information in the watch window using the following command.

```
@AshTimer.FormatString
```

Where the format string matches exactly a Print Action Command's format string from within the source code.

See the Print Action Command section for more information on how to instrument your code.

15.4 eTPU Channel Frame Window

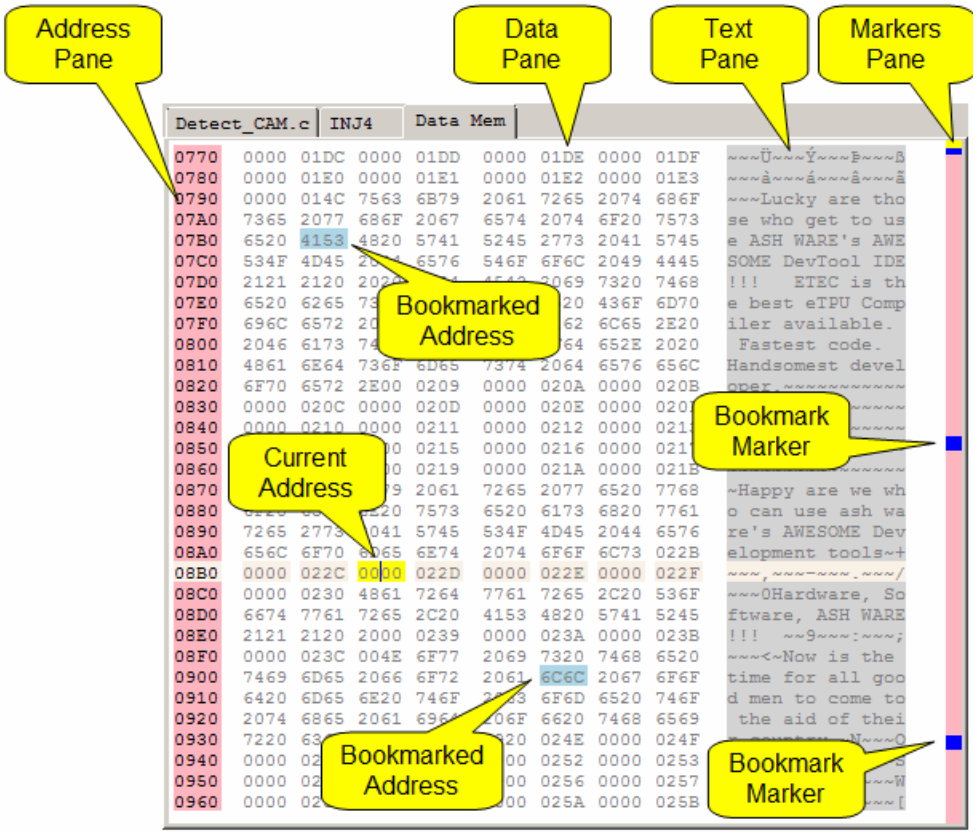
Channel Frame : Pulse3

Channel Pulse3

	Symbol	Value	Type
	Delay	1024	_CHANNEL int24
	PulseWidth	48	_CHANNEL int24
	Gap	101	_CHANNEL int24
	PulseCount	3	_CHANNEL int24
⊟	MyArray	0x491	_CHANNEL int24[]
	MyArray[0]	258	_CHANNEL int24
	MyArray[1]	279	_CHANNEL int24
	MyArray[2]	546	_CHANNEL int24
⊟	myStruct	0x49c	_CHANNEL struct MyStruct
	myStruct._member1	9	_CHANNEL int24
	myStruct._member2	6	_CHANNEL int24
	myStruct._charMember	60 '<'	_CHANNEL unsigned char
	CurrentPulseCount	3	_CHANNEL int24

The Channel Frame window shows the channel function and static local variables belonging to a particular channel. Select the channel from the dropdown box.

15.5 Memory Dump Window



Miscellaneous capabilities

To access the 'Goto Address' dialog, double click an address box.

General navigation - use the keyboard's up/down, page-up/page-down, Home, End keys. Also, use the mouse's scroll bar or click anywhere in the markers pane, also, click on a marker to view it.

To go to the first data in the row - hit the 'Home' key twice.

To go to address zero - hit the 'Home' key three times.

To go to the last address in the row - hit the 'End' key twice.

To go to the last address in memory - hit the 'End' key three times.

Editing - Edit any data box to change it's value (only alphanumeric values are accepted). Edit in the text pane (most keystrokes are supported).

Bookmarks

Bookmarks are used to quickly restore different views into the data memory. Stored view information includes selected address, the first row in the window, and active row within the window. The intent is to restore all aspects of the window layout. Bookmarks appear as little blue boxes in the 'Markers' box.

To toggle a bookmark - double click on any 'Data Box'

To goto a bookmark - click on the marker in the markers window.

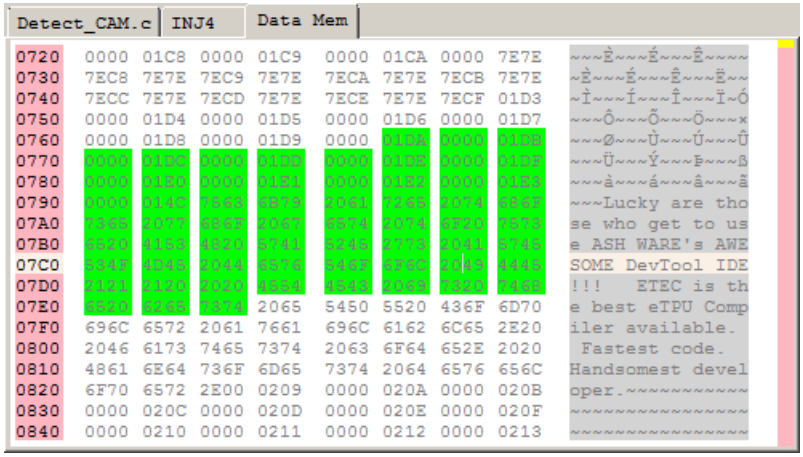
Additional bookmark navigation and hot-keys. Right click to bring up the popup menu the various capabilities and associated hot keys are listed.

Copy/Paste

Multi-Select - Hold the left mouse key down and select desired test. Note that the address and text fields can also be selected.

Multi-Paste - Multiple data's can be pasted from the window's clipboard. Data must be white-space or comma separated. Numbers are interpreted as decimal unless either a leading '0x' is detected or the number contains [a-zA-Z] characters. Out of range numbers are masked. The paste start is the cursor's address or the first address of the multi-select (if any).

```
Example clipboard:  0x1234, 0x4567 abcd
```



Pasting text is supported in the 'Text Pane'

Views

Address field can be 1, 2, or 4 bytes per address LSB thereby supporting 'Indexed' addressing.

Data field can be 1, 2, or 4 bytes wide.

Text Field can be enabled or disabled.

15.6 Local Variable Windows

Local Variables

	Symbol	Value	Type	
	Delay	336	_CHANNEL int24	
	PulseWidth	48	_CHANNEL int24	
	Gap	85	_CHANNEL int24	
	PulseCount	5	_CHANNEL int24	
	LastRisingEdge	836	int24 [register sr]	
⊟	MyArray	0x411	_CHANNEL int24[]	
	MyArray[0]	257	_CHANNEL int24	
	MyArray[1]	274	_CHANNEL int24	
	MyArray[2]	546	_CHANNEL int24	
⊟	myStruct	0x41c	_CHANNEL struct MyStruct	
	myStruct._member1	6	_CHANNEL int24	
	myStruct._member2	19	_CHANNEL int24	
	myStruct._charMember	45 '-'	_CHANNEL unsigned char	
	CurrentPulseCount	1	_CHANNEL int24	

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.

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.

15.7 Breakpoint Window

	File Name	Line	Hit Cnt	Hit Trg	Hit Lmt	Condition	Action
	MeasurePulse.c (C:\GIT\ASH...	66	0	%==0	10		
	GatedPwm.c (C:\GIT\ASHWARE...	82	0	always	1	errorState == INPUT_PULSE_TOO_SHORT	print_to_trace("edge time = 0x%x", ertA);
X	GatedPwmTest.ETpuCommand (...	34	0	always	1		

The Breakpoint Window displays all the source and script file breakpoints, and provides access to enhanced breakpoint capabilities. Breakpoints must initially be toggled on (created) from within a code/script editor window, by clicking on the left margin at the desired line. Once created, a breakpoint can be toggled off by left-clicking on the breakpoint icon. All other breakpoint attributes and capabilities are available within the Breakpoint Window, once a breakpoint is created.

All breakpoints from all targets and files are listed in the Breakpoint Window. It contains 8 columns. The first column indicates the breakpoint active status - clicking on this item toggles a breakpoint between enabled and disabled. Disabled breakpoints are ignored when simulation is running. Note that breakpoints can be completely deleted from within the window by selecting a breakpoint line (or multiple lines) and hitting the Delete key.

The second and third columns provide the file and line of the breakpoint. Left-clicking on either of these fields will cause the file to open/pop to the front of a tab, and scroll to the breakpoint.

The seventh column allows for a conditional expression to be entered for the breakpoint (source code breakpoints only). When execution reaches the breakpoint, the expression is evaluated. If it evaluates to a non-zero value, then the breakpoint is considered a "hit", whereas a zero result means execution continues and the breakpoint is ignored.

The fourth to sixth columns provide a hit count capability for breakpoints. This capability comes into play after a conditional expression is evaluated, if any. The fourth column displays the number of times the breakpoint has been encountered since simulation reset. The user can manually edit this field. The sixth column contains the hit limit value for the breakpoint (default of 1). This value can be used in conjunction with the hit count and the hit trigger type to determine whether the breakpoint should activate or be skipped. The hit trigger type is configured in the fifth column. The default is "always", which means the breakpoint should always activate, regardless of hit count or limit. The other trigger types are ">=", "==", "<", and "%=0" (modulo). For example, for the first, if hit count >= hit limit, then activate the breakpoint. The modulo trigger type activates when (hit count % hit limit) == 0.

The last column allows for the entry of a breakpoint action. This is essentially the same as the action tag capability, but linked to a breakpoint rather than an @ASH@ tag in the source code text. If a breakpoint passes its other tests and is set to activate, the logic checks to see if there is an action to process. If no action, the breakpoint halts execution as per normal. If there is, the action is processed (see the action tag section for the available commands), and if it succeeds, the breakpoint does not actually activate and execution proceeds.

16

Dialog Boxes

This section covers the various dialog boxes used throughout the eTPU Development Tool.

16.1 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 the eTPU Development Tool until a user-specified time.

There are two types of Goto time options, one of which must be selected.

Goto Until Time

This sets the eTPU Development Tool to execute until an absolute (simulation) time is reached. 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 the eTPU Development Tool 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). Note that the eTPU Development Tool resolution is one instruction cycle.

Help

This accesses this help window.

Goto

This closes the Goto Time dialog box and runs the eTPU Development Tool until the specified time.

OK, Save

This closes the Goto Time dialog box and saves any changes. The eTPU Development Tool remains idle.

Cancel

This closes the Goto Time dialog box without saving any changes.

16.2 Goto Angle Dialog Box

The Goto Angle Dialog Box is opened via the Run menu by selecting the Goto Angle submenu. It provides the capability to execute the active target until it gets at or beyond the specified angle. The eTPU must be in angle mode (AM=1) in order for this to function properly.

This dialog box uses the TCR2 counter, and user-defined angle indices to calculate the angle. See the eTPU Time Base Configuration Script Commands section for setting the angle indices.

The '**Cycles**' field refers to the number of times the angle has rolled over. For example, in an automobile engine, two rotations constitutes one cycle. The angle therefore goes from 0 degrees, to 720 degrees, then rolls over back to zero degrees. '**Cycles**' is the count of these rollovers.

The current angle can be seen in the status bar at the bottom of the IDE.

Goto Until Angle

This sets the active target to go to an absolute (simulation) angle. The simulation angle is initially set to zero. The simulation angle is reset to zero via the Run menu by selecting the Reset submenu.

Note that if the desired stop cycle and angle has already been traversed, then the Cycles field is ignored and the simulation is halted the next time the specified angle is traversed, in either the current or next cycle.

Goto Current angle, Plus

This sets the additional angle to which the active target will be run. The angle to which the active target will run is the current angle plus the specified delta angle.

Help

This accesses this help window.

OK

This closes the Goto Angle dialog box and runs the simulator until the simulator is at or beyond the specified angle in the active target.

Cancel

This closes the dialog box without saving any changes.

16.3 Workshop Options Dialog Box

Each window is assigned to one or more Workshops.

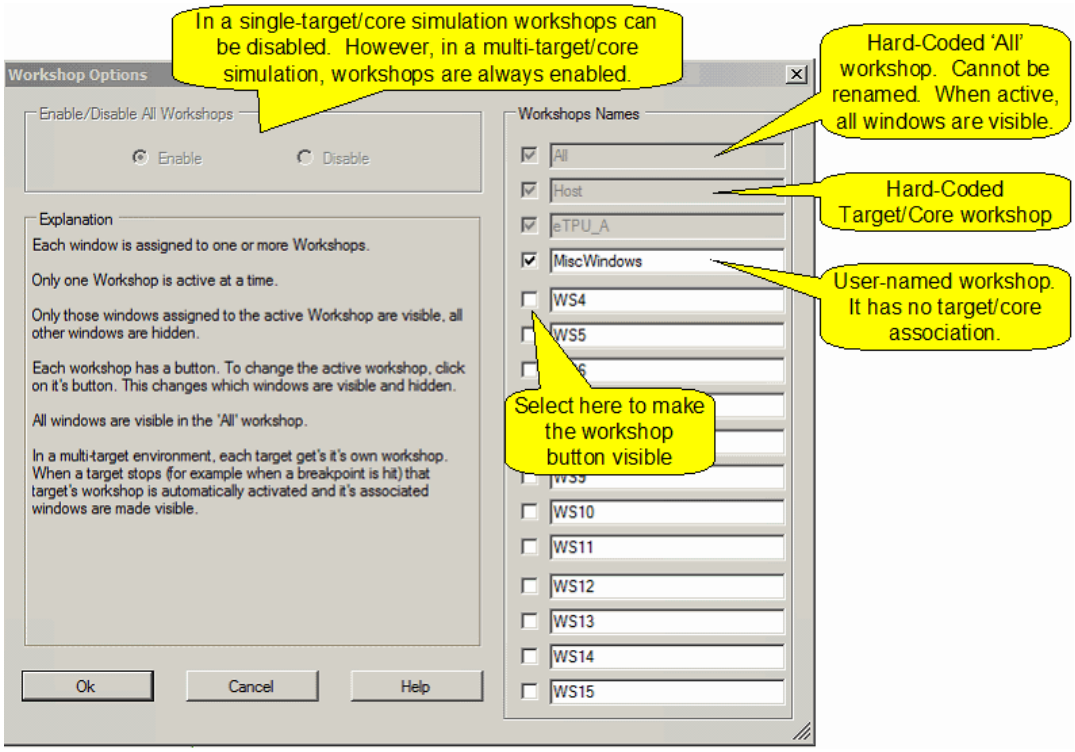
Only one Workshop is active at a time.

Only those windows assigned to the active Workshop are visible, all other windows are hidden.

Each workshop has a button. To change the active workshop, click on it's button. This changes which windows are visible and hidden.

All windows are visible in the 'All' workshop.

In a multi-target environment, each target get's it's own workshop. When a target stops (for example when a breakpoint is hit) that target's workshop is automatically activated and it's associated windows are made visible.



Related Information

The 'Workshops' section
The 'Workshops Occupy' section on how to assign windows to workshops

16.4 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 theeTPU Development Tool .

Revert

This causes any settings made since the dialog box was opened to be discarded.

Options

This opens the Workshop Options dialog box.

16.5 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.

16.6 Source Code Search Dialog Box

The Source Code Search Options dialog box is opened from the Options menu by selecting the Source Search submenu.

When the executable image is loaded, there are normally a number of source code files associated with the executable image that get loaded. The eTPU Development Tool needs to be able to find these files. This dialog box allows specification of source code directories to be searched when searching for these source code files.

The search locations can be specified for each individual target, and for all targets globally. Specifying global search options is useful in situations in which multiple targets are using the same directories for their library files.

When searching for a source code file, the following algorithm.

- If the path to the file is fully-specified (e.g. c:\SomeDir\SomeFile.c) use that if the file exists there.
- If the path is partially specified (e.g. ..\OneUpDir\SomeFile.c) resolve the path relative to the code image file. For instance, if the code image file is at c:\SomeDir\SubDir\CodeImage.Elf, then check for this file c:\SomeDir\OneUpDir\SomeFile.c
- If the source file is still not found, use the raw file name (strip any prepended directory information) and search in the directory where the executable image file is located.
- If the source file is still not found, use the raw file name (strip any prepended directory information) and search for the file in the directory(s) listed in the 'Selected Targets' directory search list, starting from the top-listed directory.
- If file is still not found, use the raw file name (strip any prepended directory information) and search for the file in the directory(s) listed in the 'All Targets' directory search list, starting from the top-listed directory.

Add

This button inserts a new directory into the search list.

Modify

This button modifies a previously entered search location.

Delete

This button removes a location from the search list.

Cut

This button removes the currently selected search location from the search list, and places it into the paste buffer.

Copy

This button adds the currently selected search location to the paste buffer without removing it from the search list.

Copy

This button creates a new search location using the paste buffer.

Move Up

This button moves the currently selected search location higher in the search list such that this location is searched earlier.

Move Down

This button moves the currently selected search location lower in the search list such that this location is searched later.

16.7 Waveform Window Options Dialog Box

The Waveform Window Options Dialog box defines the settings associated with the Waveform Window.

Clocks or Time

Determines if the the Waveform Window should reference target clocks or time.

Display of Time

Determines the timebase and leading/trailing digits for the waveform window's display of time.

Configure Thread Groups

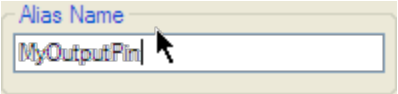
There are eight thread groups labeled from 'A' to 'H'. This opens the Thread Group Dialog Box allows configuration of which thread(s) are associated with each of these groups.

16.8 Waveform Signal Options Dialog Box

The waveform dialog box allows a selection of a variety of signals for display in the waveform window. It also provides a options for viewing.

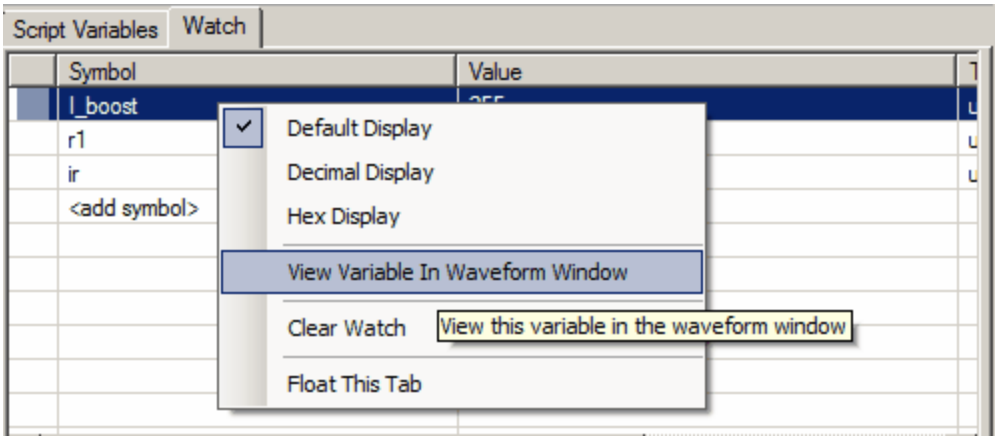
Alias Name

The alias name allows the symbol name to be given a more intuitive and application-relevant name. For example, the underlying '_ch10.out' signal could be renamed to be 'Spark 3'.



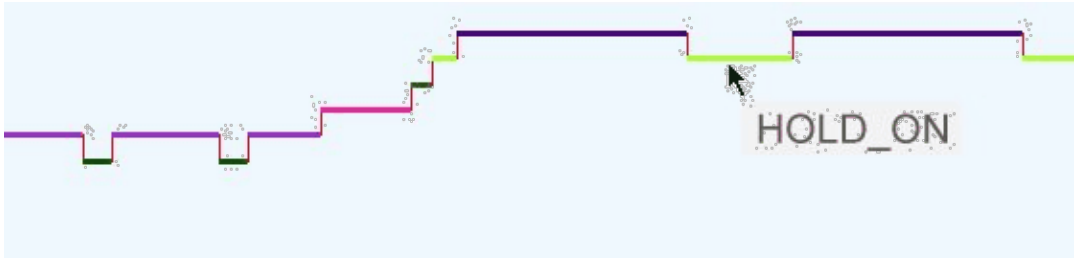
Code Variables

In order to display code variables as discrete nodes they first must be configured to be stored in the waveform data buffer. This is done from the watch window by right clicking on a symbol and selecting 'Make Waveform a Discrete Node' as shown below.



State Machine State

The state machine's state is automatically added to the waveform data buffer and is therefore available in the 'variables section'. Note that the State Name will appear if the mouse is hovered over the waveform as shown below in which the green portion of the waveform indicates that the 'HOLD_ON' state was active over that period.



Thread Groups

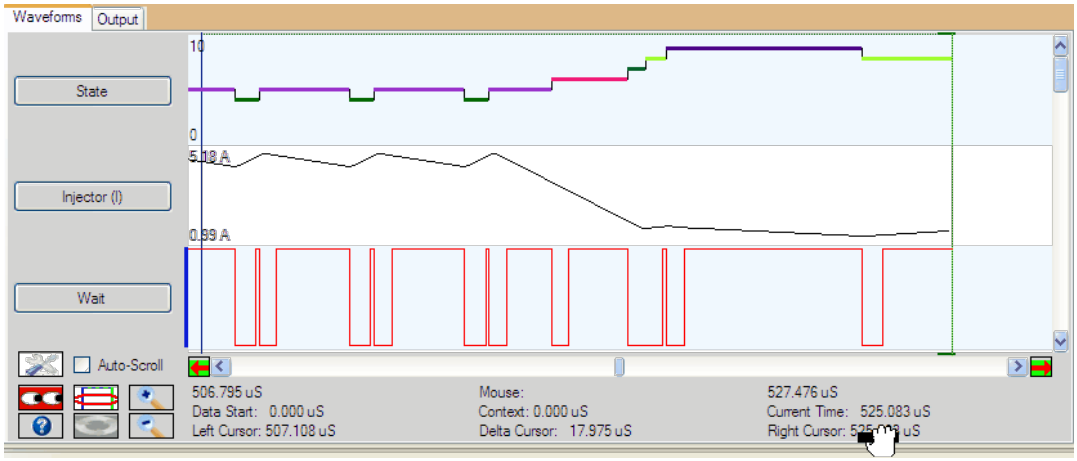
Thread activity is one of the most important and useful features of the waveform window. In the eTPU, before thread groups are meaningful, they must be configured in the Waveform Window Dialog.

Both the eTPU and the MC33816 are event response machines such that the Execution Unit goes idle after servicing an event and become active when an event requiring servicing occurs. The Thread Groups shows when these event-handling threads occur.

However, the very most powerful and useful feature is the ability to snap the vertical cursors unto the start of the thread. This takes several steps as listed below.

- Use the up/down keyboard keys to make the thread group the active (red) waveform
- Use the left and right keyboard keys to snap the vertical cursor unto the exact point in time when the thread servicing begins
- Using the mouse, drag and drop the time listed in the 'Right Cursor' or 'Left Cursor' unto the waveform.

Wow, ..., what happened? Answer: the simulator was reset, then ran until the point time listed in the 'Right Cursor' or 'Left Cursor' waveform.



Analog and Discrete Signals

Analog and discrete signals can modify their vertical range.

The 'Auto Range View' button configures the waveform display visible portion of the waveform is fully visible in the display.

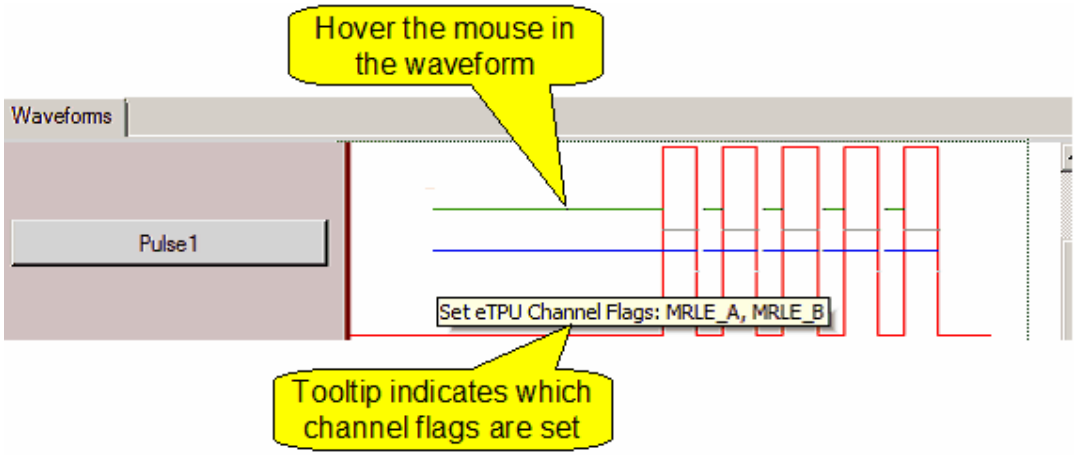
The 'Auto Range Buffer' button is exactly the same as the 'Auto Range View' button except that the entire waveform (including that in the buffer which may not be visible) will fit, vertically, into the display.

Digital Signals

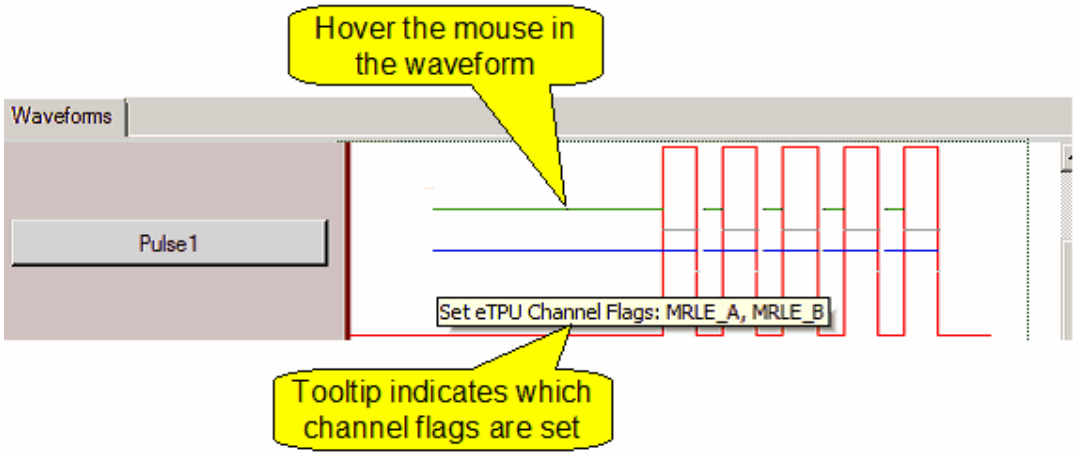
A variety of digital signals can be selected for viewing from the 'Node Selection' tree.

Channel State Overlay Enables

Channel State Overlay Enables allow the channel states to be visible in the waveform. Note that these are only available in eTPU input and output pins.



The waveform below shows the channel State Overlays. The green horizontal line indicates when MRLE_A is set. The blue horizontal line indicates when MRLE_B is set.

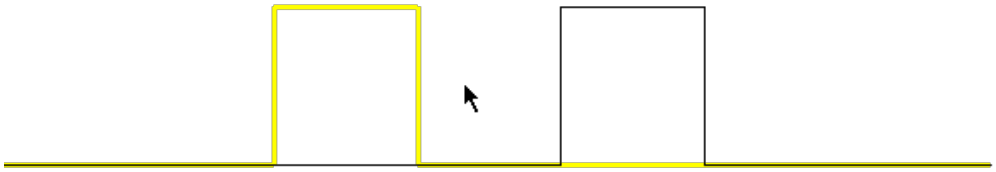


Behavior Verification Overlay

Behavior verification allows waveform data to be stored into a 'Gold File'. The data in the 'Gold File' can be replayed in the simulator and compared against the current simulation value. When 'Behavior Verification' is checked, any previously stored data is overlayed unto the current waveform, thereby making highlighting differences in the current simulation versus the previous simulation as shown below.

- The black waveform is the same signal in the current simulation run.

- The yellow waveform is the previously-saved waveform restored from the gold file. Note that in the waveform shown below, the pulse occurred earlier than in the latest run.



In order to view previously-saved simulation data the following steps are required

- Save waveform data into gold files by adding script commands create_ebehavior_file(), add_ebehavior_pin(), and close_ebehavior_file() to the the script commands file, then run a simulation to generate a gold file.
- Replay a previously-saved simulation waveforms from a gold file using the close_ebehavior_file() script command.
- Enable viewing of the previously saved 'Behavior Verification' waveform by selecting the 'Enable Behavior Verification Overlay' checkbox.

Note that behavior verification is a big topic. For a complete description, see the Pin Transition Behavior Verification section of the Functional Verification chapter.

Vector Files

A list of all nodes defined in the vector file is available along with the aliases for those named nodes. This allows waveforms to be quickly associated with aliases matching those in the vector file.

Note that unlike previous products, the waveform alias is not tied to the vector file. When the vector file reloads and the eTPU Development Tool detects that the pin/signal of a waveform alias matching a vector file alias mismatch, the user will be asked if they want their waveform alias to update to the new pin/signal. This situation can happen if a user changes pin assignment in the vector file - the eTPU Development Tool allows the user to easily update their Waveform configuration to match.

16.9 Channel Group Dialog Box

The Channel Group Options Dialog box is used to select groups of one or more channels. It is accessed in different locations for different purposes.

This dialog box is accessed from the Waveform 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.

16.10 Trace Options Dialog Box

The Trace Options Dialog Box specifies which types of trace data are stored in the buffer and which are displayed in the trace window. Once a simulation is run, only data stored in the trace buffer can be displayed. However, trace data that is configured for storage in the buffer but not display can later be displayed without re-running the simulation. Data not configured for storage in the trace buffer cannot be viewed without re-running the simulation.

A computer's memory is finite and trace data can therefore sometimes overflow the trace buffer. In this case the oldest trace data is discarded to make room for the most recent trace data. One way to increase the effective buffer size is to disable certain types of data, or data from specific targets, from being stored in the trace buffer.

Note that in a multi-target environment 'Target Display' check boxes appear at the top of the trace window. These check boxes allow the enabling and disabling of all trace data for the entire target all at once. However, in order to be displayed, a trace type must be enabled in both the dialog box and in the checkbox at the top of the trace window.

Note that unlike previous simulation, all trace data from all targets is stored in a single trace buffer. So disabling storage of trace data from one target will increase the effective storage size of other targets.

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 eTPU 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.

16.11 License Options Dialog Box

This dialog box allows you to enter additional information prior to sending a license file to ASH WARE Inc. A license file is generated whenever you install an ASH WARE product. Unfortunately, the license file generated at install time contains very little information other than a computer identifier. This dialog box allows you to add additional information such as your name, purchase order number, etc.

The license file has been a problem for ASH WARE in that users have sent in license files for purchased products but we were unable match the license files with the purchase. This dialog box is intended to reduce this confusion, thereby allowing us to serve you better.

All information is optional. Generally, it is best to include at least your company's purchase order number or the ASH WARE invoice number, if available.

16.12 Memory Tool Dialog Box

This tool supports specialized memory functions listed below. The dump file functions are also accessible from the dump_file script command listed in the File Script Commands section.

- Fill memory with data or text
- Search for data or text
- Dump to disassembly file
- Dump to Motorola SRecord (SREC) file
- Dump to Intel Hexadecimal (IHEX) file
- Dump to image file
- Dump to "C" structure file

For each function the address space and memory range can be specified. Earlier address ranges are stored in a buffer and can be retrieved using the recall button. For the file-dump options, selecting the change button can specify the file name.

In disassembly dump files inclusion of address, raw values, addressing mode information, and symbolic information can be selected.

When creating an image file or a "C" data structure file, or when using the fill function, the data word size can of 8, 16, or 32-bits. For the fill function, verification after the fill can be selected.

The find capability allows a specific byte pattern to be located in memory. Options include the ability to search for between one and eight sequential bytes, as well as the ability to search for both a case sensitive or case-insensitive string.

"C" data files can be written with the address included within a comment. Output format can be either hexadecimal, which is the default, or decimal.

Selection of endian ordering is available for where the data size 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.

16.13 The 'About' Dialog Box

17

Menus

This section covers the various application menus in the eTPU Development Tool.

17.1 Files Menu

The Recent, ... submenu's

The Recent Project, Source, Script, Vector, and Other submenu's allow recently-opened files to be readily re-opened.

Open Existing, ... submenu's

Opens a dialog for selection of existing files.

Create New

Creates a brand new file. A dialog opens that allows selection of the new file's filename. When creating a new Source code files, State Machine, or Primary Script file the new file can either be blank or created from a template.

Save File

Saves the currently-active file.

Save All Files

Saves all open files that have been modified (dirty).

Close

Closes the currently-active file.

Project, Save

Saves the current settings to the project file and the environment file. The project file contains all the key settings and is typically placed in version control. The environment file contains all the user settings such as window positions, open files, etc., and is typically specific to each user and not placed in version control.

Write

A number of files can be generated

The Code Coverage Statics file lists which code has been executed.

The Code Load report gives statics on the currently-loaded code.

The Trace Buffer file shows detailed trace information from the last simulation run.

The Symbol Table file gives detailed symbolic (variable) information on the loaded code.

The Window Snapshot takes a snapshot of the currently active window and saves it to a file. This is great for documentation.

17.2 Build Menu

Build All

Force a re-build of all code, regardless of whether the source files have been modified or not

Make

Conditionally re-build code, based on whether or not the source code files and project settings have changed since the last build.

Compile

Compile (or assemble) the active source code file

Re-Parse Script

Re parse just the script file. *Note that the hot-key for this action when editing is 'Ctrl+R' which is the same as 'Reset' when simulating.*

Re-Parse Vector

Re parse just the vector file. *Note that the hot-key for this action when editing is 'Ctrl+R' which is the same as 'Reset' when simulating.*

Help

This accesses this help screen.

17.3 Edit Menu

Goto Error

In the Output Window, errors and warnings from the last build, script parse, and vector parse are listed. These menu items activates the referenced file and scrolls to the referenced file line.

Goto Line ...

Opens a dialog to move the cursor to a specific line in a source code file.

Find

Finds text in files. Options are as follows.

- Just a single file or multiple files can be searched
- Included files (using #include) can be searched
- The last search option can be repeated.
- All files in a specific directory can be searched
- Files in sub-directories can be included in the search

Bookmarks

Bookmarks can be dropped on to lines of code. These lines of code can be quickly brought back into view using 'Goto Next Bookkmark'

Select All

Selects all text in an editable file.

Cut, Copy, Paste

Uses the standard windows clipboard to perform 'Cut, Copy and Paste' functions.

Undo, Redo

Undo edits. Redo the undo edits.

Block Indent, Block Un-Indent

Adds or removed indentation from multiple text lines.

Help

This accesses this help screen.

17.4 Step Menu

Into

This runs the active target until one line of source code is executed. If a function is called, eTPU Development Tool halts on the first instruction within that function. If no instructions associated with source code lines occur, eTPU Development Tool 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, eTPU Development Tool 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, eTPU Development Tool 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 eTPU Development Tool until one script command from the active target is executed. If no new script commands become available eTPU Development Tool continues to run until stopped by selecting the Stop submenu in the Run menu.

Thread Start (Time Slot)

Runs until the beginning of the next eTPU thread (time slot transition.) Execution stops just prior to execution of the first opcode in the thread. If no thread occurs, execution continues to run until stopped by selecting the Stop submenu in the Run menu.

Thread End

This command is excellent for running from thread-to-thread in the same channel.

- If in a thread, run, then stop at the end of the thread.
- If at the end of a thread, run until the beginning of a thread on the same channel.
- If no thread is active, run until the beginning of a thread on the last-active channel.

Angle Tick

When the eTPU is in angle mode, this runs until the next angle tick occurs.

Angle Tooth

When the eTPU is in angle mode, this runs until the angle tooth occurs. The angle tooth could be generated by a physical tooth, a tooth induced by asserting TPR.IPH, or by the EAC decrementing TPR.MSCNT.

Assembly

This runs the active target until a single assembly instruction occurs. If instructions occur, DevTool 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, eTPU Development Tool continues to run until stopped by selecting the Stop submenu in the Run menu.

Help

This accesses this help screen.

17.5 Run Menu

Goto Cursor

This runs eTPU Development Tool 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 eTPU Development Tool 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 eTPU Development Tool to execute indefinitely. eTPU Development Tool executes until a breakpoint is encountered or until the user terminates execution by selecting the Stop submenu in the Run menu.

Stop

This causes eTPU Development Tool to stop executing. This is normally used to stop eTPU Development Tool when the expected event (such as step, breakpoint, time slot transition, etc.) failed to occur.

Reset and Go

This causes eTPU Development Tool to reset and immediately begin execution. The reset actions are specified in the Reset Options submenu. eTPU Development Tool will execute until a breakpoint is encountered or until interrupted by the user selecting the Stop submenu from the Run menu.

Reset

This causes eTPU Development Tool to reset. The reset actions are specified in the Options submenu in the Reset menu.

Help

This accesses this help screen.

17.6 Breakpoints Menu

The breakpoints menu controls the various functions associated with the breakpoint capabilities of the eTPU Development Tool. These capabilities are accessed via the following submenu's.

Set (Toggle)

This toggles a breakpoint at the source code window’s cursor. If the instruction already has a breakpoint, then the breakpoint is deleted. A source code window must be active for this to work. If the source code is in mixed assembly view mode, and the cursor is at a disassembly line, then the breakpoint will be generated for only that address.

Delete All

This deletes all active and all disabled breakpoints.

At Address …

This opens a dialog box which allows placing a breakpoint any user-specified address. This is useful when debugging code for which there is no line number information, such as GNU assembly code.

Delete All Script

This deletes all breakpoints that are in the script commands file.

Help

This accesses this help screen.

17.7 View Menu

This menu opens the various Simulator windows for viewing. The eTPU Development Tool allows multiple instances of each window to be open simultaneously. When there are multiple targets a submenu for each target appears. Otherwise, all available windows are available directly within the view menu.

See the Operational Status Windows chapter for a listing of the available windows for each target.

Help

This accesses this help screen.

17.8 Options Menu

The Options menu provides the user with the capability of setting various Simulator options. These are listed below.

Workshop

This opens the Workshops Options dialog box.

Messages

This opens the Message Options dialog box.

Waveform, ...

This opens the Waveform Window Options dialog box

Waveform

This accesses a number of sub-menus used for setting the currently-active waveform in the waveform window.

Help

This accesses this help screen.

17.9 Help Menu

The Help menu provides the following options.

Contents

Accesses the contents screen of eTPU Development Tool's on-line help program.

Technical Support

This accesses information on obtaining technical support for this product.

About

This gives general information about eTPU Development Tool.

18

Supported Targets and Available Products

Due to its layered design, eTPU Development Tool supports a variety of both simulated and hardware targets. Customer requirements dictate that these capabilities to be offered as specific individual products.

18.1 eTPU/CPU System Simulator

Our eTPU/CPU System Simulator supports instantiation and simulation of an arbitrary number and combination of eTPUs and CPUs. A dedicated external system modeling CPU could be used, for instance, to model the behavior of an automobile engine. Executable code can be individually loaded into each of these targets. Synchronization between targets is fully retained as the full system simulation progresses.

All CPU engine targets can be used with this system simulation include the CPU32, and soon-to-be-released, PPC simulation engines.

18.2 MC33816 Stand-Alone Simulator

This product is a single-target version that uses only the MC33816 simulation engine. Because it is a stand-alone product the user must use script commands files to act as the host and test vector files to act as the external system.

The MC33816 Stand-Alone Simulator is a superset of the MC33816/CPU System Simulator in that purchase of the Stand Alone Simulator license allows installation of a fully-functional Stand-Alone Simulator product as well as the full System Simulator configuration.

18.3 eTPU2 Stand-Alone Simulator

This product is a single-target version that uses only a single instance of our eTPU2 simulation engine. Because it is a stand-alone product the user must use script commands files to act as the host and test vector files to act as the external system.

The eTPU2 Stand-Alone Simulator is a superset of the eTPU Stand Alone Simulator in that purchase of the eTPU2 Stand Alone Simulator license allows installation of a fully-functional eTPU Stand-Alone Simulator product as well as an eTPU2 Stand-Alone Simulator product.

18.4 eTPU Stand-Alone Simulator

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

This product simulates the original eTPU1 from NXP. For simulation of the new eTPU2, see the eTPU2 Stand-Alone Simulator product.

18.5 eTPU2 Simulation Engine Target

The Enhanced Time Processing Unit Two, or ‘eTPU2’, is a microsequencer sold by STMicroelectronics and NXP Semiconductor on a variety of microcontrollers including the SPC563Mxx. This is sold as a stand alone product and also as a system simulator in which it is co-simulated along with one or more CPU targets.

The eTPU2 simulation engine can be used both as a stand-alone device and in conjunction with other targets including multiple TPUs. When used in stand-alone mode, of primary importance are script commands files and test vector files.

18.6 eTPU Simulation Engine Target

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

19

Building the Target Environment

The simulated or hardware development environment is specified in special MtDt build script files. ASH WARE provides standard build scripts for all its products. In the vast majority of cases these build scripts are sufficient and therefore effectively transparent to the user. But occasionally a user may desire to modify the simulation environment to unlock advanced capabilities. This chapter describes how to do so.

A complete system may consist of multiple CPUs, eTPUs, peripherals, and non-electrical devices such as automobile engines. The eTPU Development Tool supports simulation of such advanced systems using MtDt build script files. MtDt build script files are used to create the targets and specify how they interact.

Theory of Operation

eTPU Development Tool 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., are available in the multiple target debugging environments. For instance, if breakpoints are injected and activated within multiple targets, the simulation halts on whichever breakpoint is encountered first.

A concept of an "active target" is employed to support specifically single-target capabilities such as single stepping. When the active target is single-stepped, the entire system simulation proceeds until the active target completes the commanded single step.

With an essentially limitless number of targets, and with the large number of possible windows per target, the vast number of windows can become unwieldy, to say the least. Actually, without some mechanism to bring order to the chaos of having way too many windows, MtDt becomes unusable. Workshops bring order to this chaos and are therefore a key enabling feature that makes MtDt usable. Each target can be associated with a specific workshop. Those target's windows are displayed only when the workshop associated with that target is activated. Individual windows can be overridden to appear in more than one target.

A target other than the active target can halt a simulation. In this situation the workshop is changed to one associated with the halting target. This can be caused, for instance, if a breakpoint is encountered in a non-active target. In this case, the simulation is halted, the halting target becomes the active target, and the workshop is switched to the one associated with the newly-active target.

Building the MtDt Environment

With MtDt an entire hardware or simulation environment can be built. This is done using a dedicated build script file that gets loaded when the project file is loaded. A detailed description of each command that can be used within MtDt build script files is found in the MtDt Build Script Commands File section. That section explains how a complete system is defined using these build script commands.

MtDt Simulated Memory

Simulated targets require memory for executing code, for holding data, and for providing capabilities supported in a simulated environment. The following is a list of simulated memory characteristics supported by MtDt memory.

- Multiple address spaces
- Memory sizing
- Read only or read/write accesses
- Shared memory
- Byte, word, and long-word access widths
- Access speed based on even or odd access addresses
- Privilege violations
- Bus faults
- Address faults
- Banking
- Mirroring

The ASH WARE MtDt simulated memory model supports all of these characteristics though at the cost of increased complexity. The good news is that for many applications the standard memory models work just fine so a detailed understanding of memory modeling is not required. In the vast majority of other cases only a small percentage of these capabilities are required.

Memory Block

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(SIM32, "MySim32");
add_mem_block("MySim32", 0, MEM_DEVICE_STOP,
              "RAM", ALL_SPACES);
add_non_mem_block("MySim32", BLANK_START, MEM_END,"OFF",
                  ALL_SPACES);
```

In this example a single CPU32 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.

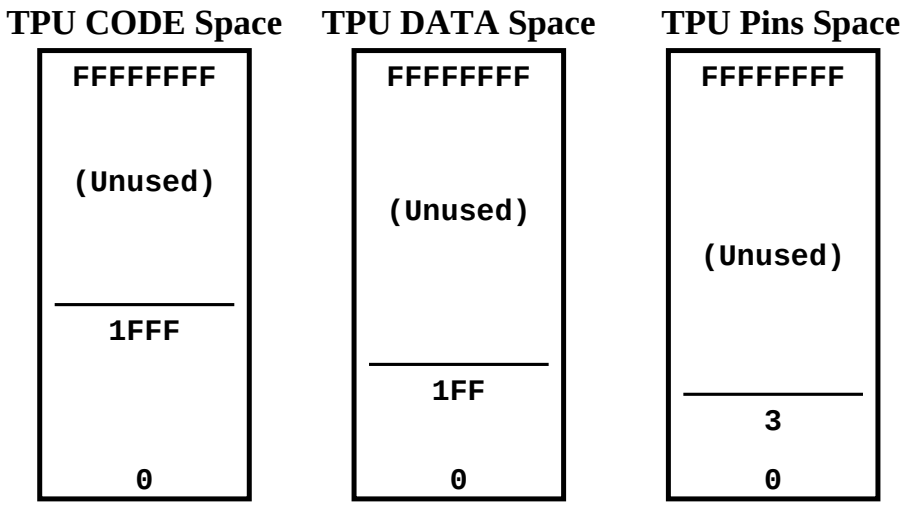
Address Spaces

All eTPU Development Tool 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 CPU32 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. The eTPU Development Tool can be configured so that a read or write in the engine modeling CPU's DATA_SPACE occurs in the eTPU'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_block("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
24-bit read
24-bit write
32-bit read
32-bit write
64-bit read
```

```
64-bit write
128-bit read
128-bit write
```

For each access type, the user can specify a number of parameters. The following parameters are available.

```
Clocks per even access
Clocks per odd access
Odd access causes bus fault yes/no?
Bus fault yes/no?
Blank access yes/no?
Dock offset (applicable docked accesses only!)
Dock function code (applicable to docked accesses only)
```

In addition, there is a block-wide default, "blank access value." This is the value that is returned on a read access to a memory block that has been marked as blank.

Read/Write Control

It is possible to disable specific types of memory accesses.

In this example a memory device is configured as read only. A write to this memory device will not cause the values in memory to change. This read only behavior could be used to model a ROM device.

```
#define ROM_STOP 0xffff
#define BLANK_START ROM_STOP + 1
#define MEM_END 0xffffffff
instantiate_target(SIM32, "MyCpu");
add_mem_block("MyCpu", 0, ROM_STOP, "Rom", ALL_SPACES);
add_non_mem_block("MyCpu", BLANK_START, MEM_END, "Empty",
                                    ALL_SPACES);
// Turn off READ access for the ROM device
#define   WRITE_ALL   RW_WRITE8 + RW_WRITE16 + RW_WRITE32
set_block_to_off("MyCpu", "Rom", ALL_SPACES, WRITE_ALL);
```

The command specifies that for all address spaces, all write accesses will be "empty." Despite this being an "empty" access, other parameters such as clocks per even cycle remain valid. The last two arguments specify the address spaces and access types affected by this script command. It is possible to indicate specific address spaces and a specific type of access. For instance, using this script command, one could specify 32-bit writes to user data space.

Clocks per Access Control

For each memory block and type of access, the clocks per even access and the clocks per odd access can be specified. This capability effectively provides the capability of setting the number of wait states.

```
set_block_timing("MyCpu", "Rom", ALL_SPACES, RW_ALL, 2, 3);
```

In this example even accesses are set to two clocks per access and odd accesses are set to three clocks per access. In this example these settings apply to all address spaces and for all read and write accesses, though it is possible to indicate the specific set of address spaces and access types for which this applies.

Address Fault Control

For each address space and access type an address fault can be set to occur on odd accesses. Note that this is generally not applicable to 8-bit accesses, though it is still available.

```
#define CODE_SPACE    CPU32_SUPV_CODE_SPACE  \
                    + CPU32_USER_CODE_SPACE
set_block_to_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.

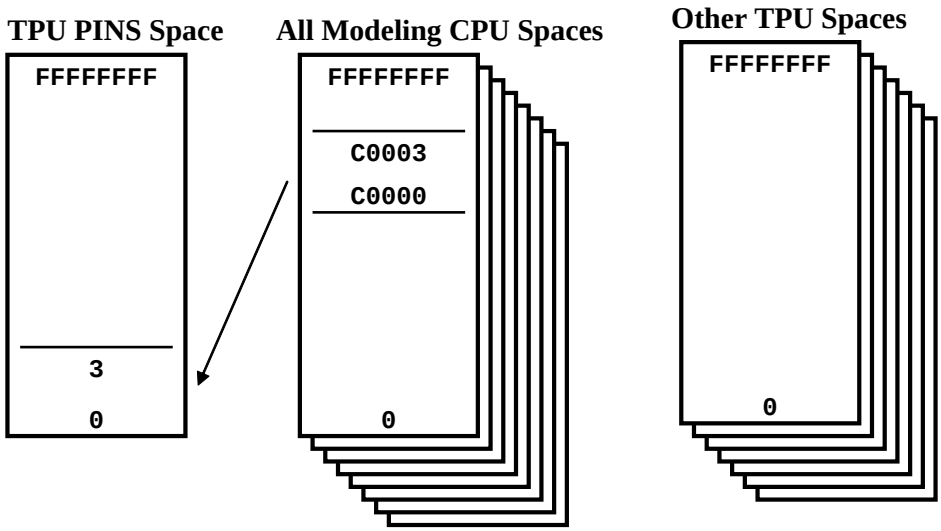
```
// ...
set_block_to_dock("HostCpu", "TpuDataDock", ALL_SPACES,
                  "Tpu", 0-0x80000);
// ...
```

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.

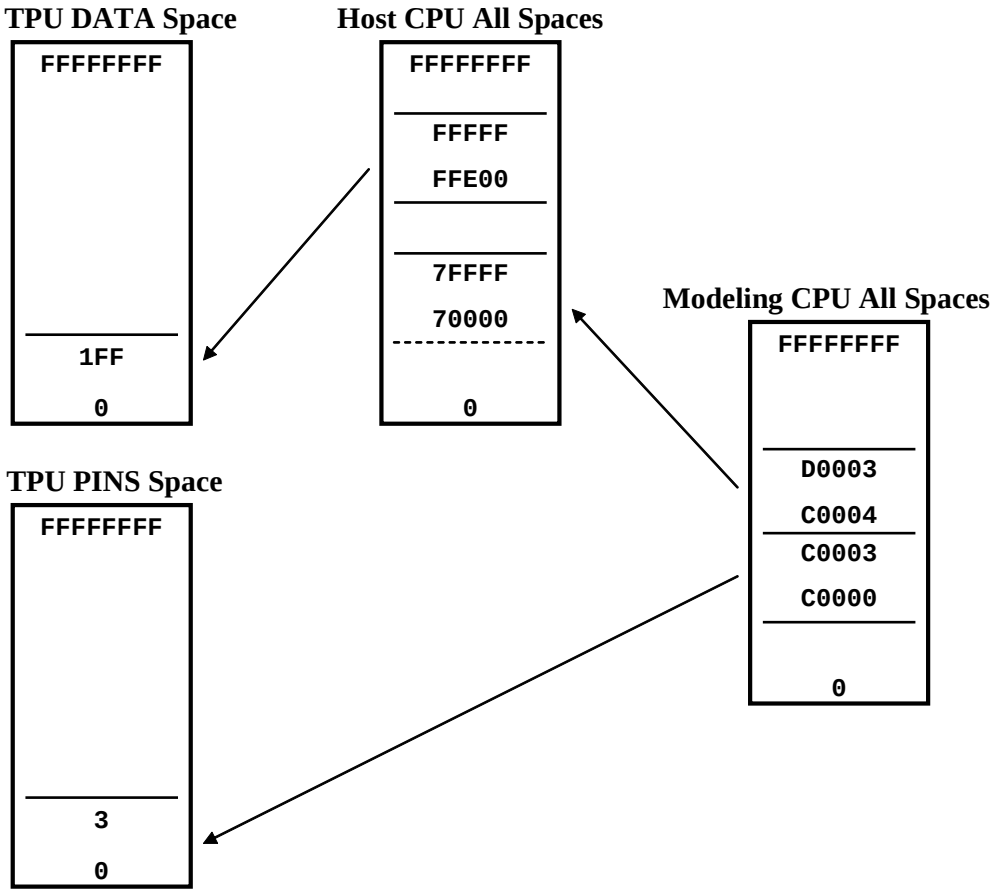
Shared Memory Timing

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 ********************
```

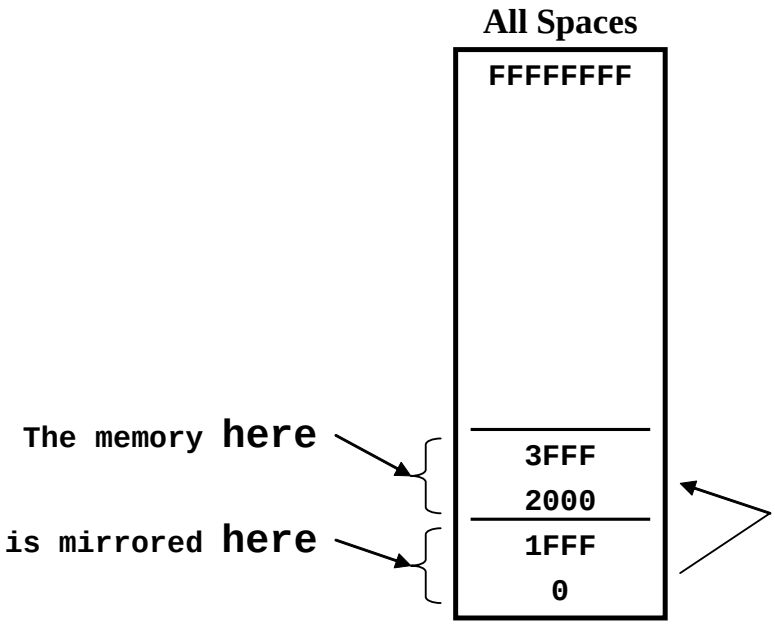
```
// Create a CPU for modeling the external system
instantiate_target(SIM32, "ModelCpu");

// 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);

//***************** END OF MODELING CPU *****************
```

In this example the shared memory architecture from the above figure is generated.

Simulating Mirrored Memory

Mirrored memory is memory that is accessible at multiple address ranges within a memory map. This can occur, for instance, when not all address bits are decoded for a memory device. The following figure depicts an 8K memory device that resides in a 64K memory system. Assume it is an 8-bit wide device. Since the 8K device is on a byte wide bus, the device itself decodes the lower 13 address bits, A12 through A0. Assume that the memory controller decodes only the upper two address bits, A15 and A14, and enables the device

when both are zero. This means that nothing decodes A13, and thus the memory device is 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(SIM32, "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.

19.1 The Build Script File

eTPU Development Tool supports both debugging and simulation of a variety of targets. Since eTPU Development Tool can both simulate and debug a variety of simulation and hardware targets, how does eTPU Development Tool know what to do?

Build batch files provide the instructions that eTPU Development Tool 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 eTPU Development Tool installation will suffice.

Typical build script files might instantiate a CPU and a eTPU, then instantiate RAM and ROM for the CPU, and then cause the eTPU'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_1Gtm.MtDtBuild

zzz_1ETpuSim.MtDtBuild

zzz_1Mc33816.MtDtBuild

zzz_1Sim32.MtDtBuild

zzz_1Sim32_1ETpuSim.MtDtBuild

zzz_1Sim32_1GTM.MtDtBuild

zzz_1Sim32_1Mc33816.MtDtBuild

zzz_1Sim32_3ETpuSim_MPC5676.MtDtBuild

zzz_1Sim32_3ETpuSim_MPC5777.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

19.2 Custom Build Script File Pathing

When creating a Custom Build Script the question arises of, "where should the custom build script file be placed? There are three options discussed here.

Option 1, Installation Location. Place your Custom Build Script in the Build Scripts directory in the installation directory where the standard build scripts are located.

```
C:\Program Files (x86)\ASH WARE\Full System DevTool IDE
V2_20A\BuildScripts
```

Advantage. Works seamlessly with the toolset.

Disadvantages. Every user needs to place the file into their installation directory and does not work well with CVS.

Option 2, Project-File Relative. If the custom build script is placed in the same directory tree as the project file in which it is referenced then only the 'relative path' to the project file is used.

```
Project File:  c:\SomeDir\MyProjectDir\MyProject.GtmIdeProj
Custom Build Script File:  c:
\SomeDir\MyCustomBuildScripts\MyCustomSimCfg.MtDtBuild
```

Then in the project file the following pathing will be used to find the build script.

```
..\MyCustomBuildScripts\MyCustomSimCfg.MtDtBuild
```

Advantage. Works well with CVS and projects are easily relocatable as long as the Build Script is retained and it relative pathing spot.

Option 3, Independent Directory. If the custom build script is placed in a separate directory tree as the project file in which it is referenced then only the complete fully-qualified path is stored in the project file.

```
Project File:  c:\SomeDir\MyProjectDir\MyProject.GtmIdeProj
Custom Build Script File:  c:\Tools\MyCustomSimCfg.MtDtBuild
```

Then in the project file the following pathing will be used to find the build script.

```
c:\Tools\MyCustomSimCfg.MtDtBuild
```

Advantage. Eliminates confusion on the location of the custom build script.

Disadvantages. Difficult to maintain.

Pathing Search Rules

If the path is fully qualified, file must be at the fully-qualified location. Otherwise, first the 'BuildScripts' directory is used. If the file is not found, then the directory of the project file is searched.

When saving, if the file is in the BuildScripts directory then the path is fully removed. Otherwise, if the file is in the same directory tree as the BuildScripts directory then the file is saved relative to the BuildScripts directory. Otherwise, if the file is in the same directory tree as the Project file then the file is saved relative to the Project File. Otherwise, the file along with it's fully-qualified directory is saved.

GTM Register Definitions File.

The GTM Register Definitions file allows each vendor's memory mapping to match that of the actual target hardware.Since each vendor can have a different memory mapping, the GTM Register Definition file allows the mapping of each vendors device within the Host CPU's address space. The pathing rules described above also apply to this file.

19.3 Build Script Commands

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

This command instantiates a target enum TARGET_TYPE and assigns it the name TargetName. The enum TARGET_SUB_TYPE specifies target/core-specific settings. Subsequent references to this target use the target name specified in this command.

Of particular interest to the eTPU is file "zzz_eTpuVersions.Dat," which is found in the build directory. This file supports selection of the eTPU version (and associated errata and memory sizes) via a #define. The #define is specified in the project.

```
instantiate_target_ex1(SIM32, MC33816, "Host");
instantiate_target_ex1(ETPU_SIM, MPC5554_B_1, "eTPU_A");
```

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

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

This command adds a memory block to a range of simulated memory. The memory appears between stopAddress and startAddress in the memory space ADDR_SPACE. The name BlockName is assigned to this block and is used by other script commands when referencing this block. The block name must be unique within its target, but other targets can have a block with this same block name. The size of the memory block is equal to one

plus stopAddress minus startAddress. Only a single copy of this memory is created, regardless of how many address spaces the block occupies.

```
add_mem_block("Host", 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("Host", 0x1000, 0xffffffff,
                  "Empty", DATA_SPACE);
```

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

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

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

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

```
set_block_to_off("Host", "ROM", ALL_SPACES, ALL_WRITES);
```

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

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

These commands specify 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. The first command sets an eight bit value, the second sets a 32-bit value.

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

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

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

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

```
add_mem_block("Host", 0x10000,0xFFFFFFFF, "Unused",
              ALL_SPACES);
set_block_to_bus_fault("Host", "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_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 eTPU 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("Host", 0x1000, 0x1003, "ShareRange",
```

```
                    ALL_SPACES);
set_block_to_dock("Host", "ShareRange",  ALL_SPACES,
                  "eTPU_A", 0-0x1000);
set_block_dock_space("Host", "ShareRange", ALL_SPACES,
                     RW_ALL, TPU_PINS_SPACE);
```

In this example a target Host is docked to target eTPU_A. 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("Host", "C:\\Temp\\CpuBuildReport.txt");
check("eTPU_A", "TpuBuildReport.txt");
```

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