

Verification Continuum™

# **HAPS® Prototyping**

## **Debugger User Guide**

---

April 2020

**SYNOPSYS®**

[solvnetplus.synopsys.com](https://solvnetplus.synopsys.com)

Synopsys Confidential Information

## Copyright Notice and Proprietary Information

© 2020 Synopsys, Inc. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

## Free and Open-Source Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

## Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

## Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

## Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at <http://www.synopsys.com/Company/Pages/Trademarks.aspx>. All other product or company names may be trademarks of their respective owners.

## Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.  
690 East Middlefield Road  
Mountain View, CA 94043  
[www.synopsys.com](http://www.synopsys.com)

April 2020



# Contents

---

## **Chapter 1: Design Setup**

Instrumenting and Saving a Design .....	9
Exporting Your Design .....	10

## **Chapter 2: HAPS Deep Trace Debug**

DTD Memory Configurations .....	12
Running Deep Trace Debug .....	13
Running Deep Trace Debug with DDR3 Memory .....	13
Running Deep Trace Debug with Multi-FPGA DTD Module .....	14
Viewing Captured Deep Trace Debug Samples .....	14
Verifying the Hardware Configuration .....	15

## **Chapter 3: Using the Debugger**

Invoking the Debugger .....	18
Debugger Windows and Panels .....	19
Setup Panel .....	20
Run Panel and Run Button .....	20
Search Panel .....	21
RTL Panel .....	23
Watchpoints Panel .....	24
Breakpoints Panel .....	24
Sampled Signals Panel .....	25
Hierarchy Browser .....	25
Tcl Script Window .....	26
Debugger GUI Operations .....	28
Debugger Main Menu .....	28
Basic Debugger Operations .....	32
Executing a Script File .....	32
Activating/Deactivating an Instrumentation .....	32
Selecting Multiplexed Instrumentation Sets .....	36

Activating/Deactivating Folded Instrumentation . . . . .	37
Run Command . . . . .	39
Pre-Configured Triggers . . . . .	41
Sampled Data Compression . . . . .	41
Sample Buffer Trigger Position . . . . .	42
Sampled Data Display Controls . . . . .	43
Displaying Data for Partial Instrumentation . . . . .	47
Saving and Loading Activations . . . . .	47
Cross Triggering . . . . .	49
Listing Watchpoints and Signals . . . . .	51
Show Watchpoint/Breakpoint Icons . . . . .	52
Debugging on a Different Machine . . . . .	52
Waveform Display . . . . .	54
Generating the Fast Signal Database . . . . .	55
Logic Analyzer Interface Parameters . . . . .	57
Logic Analyzer Scan Tab . . . . .	57
Logic Analyzer Properties Tab . . . . .	58
Logic Analyzer Submit Tab . . . . .	59
IICE Assignments Report Tab . . . . .	59

## Chapter 4: Board Bring-up

Running the Board Bring-up Utility . . . . .	62
Common Parameters for Bring-up Utilities . . . . .	63
Board Configuration Tests . . . . .	64
Running Board Configuration Tests . . . . .	64
clock_check Test . . . . .	65
con_check Test . . . . .	65
con_speed Test . . . . .	66
umr_check Test . . . . .	67
spi_flash_memory_check Test . . . . .	68
GPIO (LEDs, Buttons) Check Test . . . . .	68
Utility Commands . . . . .	68
haps boardstatus Command . . . . .	69
haps prog Command . . . . .	70
haps setvcc Command . . . . .	70
haps setclk Command . . . . .	71
haps clear Command . . . . .	71
haps confscr Command . . . . .	72
haps tssgen Command . . . . .	72
haps hstdm_report Command . . . . .	72

HAPS SPI Flash Memory Check Command .....	76
HAPS LEDs, Buttons, DIP Check Command .....	76

## Chapter 5: IICE Hardware Description

Breakpoint and Watchpoint Blocks .....	80
Breakpoints .....	80
Watchpoints .....	81
Multiple Activated Breakpoints and Watchpoints .....	81
Sampling Block .....	82
Complex Counter .....	83
Creating a Complex Counter .....	83
Debugging with the Complex Counter .....	83
Disabling the Counter .....	85
State Machine Triggering .....	86
Simple or Advanced Triggering .....	86
Advanced Triggering Mode .....	87
State-Machine Editor .....	95
State-Machine Examples .....	98

## Chapter 6: Connecting to the Target System

Basic Communication Connection .....	104
Debugger Communications Settings .....	104
Debugger Configuration .....	105
UMRBus Communications Interface .....	111
UMRBus Communication Debugging .....	112





## CHAPTER 1

# Design Setup

---

After your HDL is successfully created, the instrumentor is used to define the specific signals to be monitored. Saving the instrumented design generates an *instrumentation design constraints* (idc) file and adds constraint files to the HDL source for the instrumented signals and break points. The design is synthesized and then run through the remainder of the process. After the device is programmed with the debuggable design, the debugger is launched to debug the design while it is running in the target system. For information on using the debugger, see the *HAPS® Prototyping Debugger User Guide*.

The information required to instrument a design includes references to the HDL design source, the user-selected instrumentation, the settings used to create the Intelligent In-Circuit Emulator (IICE), and other system settings. Additionally, you can save the original design, in either an encrypted or non-encrypted format. This information is used to reproduce the exact state of the design.

## Instrumenting and Saving a Design

After setting up the IICE as described in [Chapter 2, IICE Configuration](#) and after defining the instrumentation (selecting the signals for sampling, and setting breakpoints) as described in [Chapter 3, Using the Instrumentor](#), the design is instrumented and saved.

Saving a design generates an *instrumentation design constraints* (idc) file and adds compiler pragmas in the form of constraint files to the design RTL for the instrumented signals and break points. This information is then used to incorporate the instrumentation logic (IICE and COMM blocks) into the synthesized netlist.

## Exporting Your Design

After your design is instrumented and saved, synthesized, and the bit file is generated, the entire file set is exported from the HAPS ProtoCompiler database to an external location where it is accessed by the debugger.

## CHAPTER 2

# HAPS Deep Trace Debug

---

The HAPS Deep Trace Debug feature uses the expanded on-board DDR3 memory of the HAPS-DX7, HAPS-70 series, or HAPS-80 systems to run deep trace debug. The expanded memory capacity allows for deeper debug.

- [DTD Memory Configurations](#), on page 12
- [Running Deep Trace Debug](#), on page 13

# DTD Memory Configurations

You can use any of the following memory configurations for deep trace debug:

- DDR3\_SODIMM\_HT3 (4GB single-rank daughter board)
- DDR3\_SODIMM2R\_HT3 (8GB dual-rank daughter board)
- Multi-FPGA DTD Module

Using any of these types of added memory provides a significantly deeper, signal-trace buffer.

With the HAPS deep trace debug modes, the flow remains unchanged. The only difference is in the configuration of the additional memory as the sample buffer using IICE parameters in the instrumentor (see *Chapter 4, HAPS Deep Trace Debug* in the *HAPS ProtoCompiler Instrumentor User Guide*).

When you later debug the design, the tool automatically calculates the sample depth and source clock based on the configuration settings supplied in the instrumentor. The configured sample depth can be varied dynamically from the minimum depth to the maximum configured depth.

# Running Deep Trace Debug

- [Running Deep Trace Debug with DDR3 Memory](#), on page 13
- [Running Deep Trace Debug with Multi-FPGA DTD Module](#), on page 14
- [Viewing Captured Deep Trace Debug Samples](#), on page 14
- [Verifying the Hardware Configuration](#), on page 15

## Running Deep Trace Debug with DDR3 Memory

To maximize performance when using DDR3 memory, refer to the guidelines in the table below to determine the sample frequency based on the number of sample bits being instrumented. The maximum number of instrumented bits that can be sampled with DDR3 is 2042.

Instrumented Bits	Max Sample Frequency	Max Sample Depth (4GB single rank)	Max Sample Depth (8GB dual rank)
1 to 250	140 MHz	134,217,727	268,435,455
251 to 506	70 MHz	67,108,863	134,217,727
507 to 1018	35 MHz	33,554,431	67,108,863
1019 to 2042	17.5 MHz	16,777,215	33,554,431

Based on the number of signals instrumented, the tool automatically calculates the maximum buffer depth. The configured sample depth can be varied dynamically from the minimum depth to the maximum depth.

When the sample depth is set to a large value, the captured samples are first downloaded block-by-block. Once all of the blocks are downloaded, viewing of large samples in the waveform viewer is very time consuming and also the size of the VCD/FSDB dumps becomes extremely large (for a full buffer depth, the time to download all the sample blocks can be between 30 and 40 minutes and a full VCD dump can require several hours).

To reduce these times, use the waveform writer in the debugger to dump a specific range or *slice* of the VCD/FSDB waveform (see [Viewing Captured Deep Trace Debug Samples](#), on page 14). In the debugger, click on the waveform display icon to bring up the pop-up window where you can specify

the cycle range over which to view the waveform. The configured sample depth can be varied in the debugger, but cannot be greater than the depth set in the instrumentor.

Also, using deep-trace debug on a Windows-based system with minimal resources can be extremely slow, especially when downloading large captured samples or when writing the corresponding VCD/FSDB waveform dumps. Increasing the memory capacity and processor speed of the host can significantly improve performance.

## Running Deep Trace Debug with Multi-FPGA DTD Module

The guidelines in the table below determine the maximum sample frequency based on the number of signals per MGB (multi-gigabit) link. The maximum number of signals per link is 256.

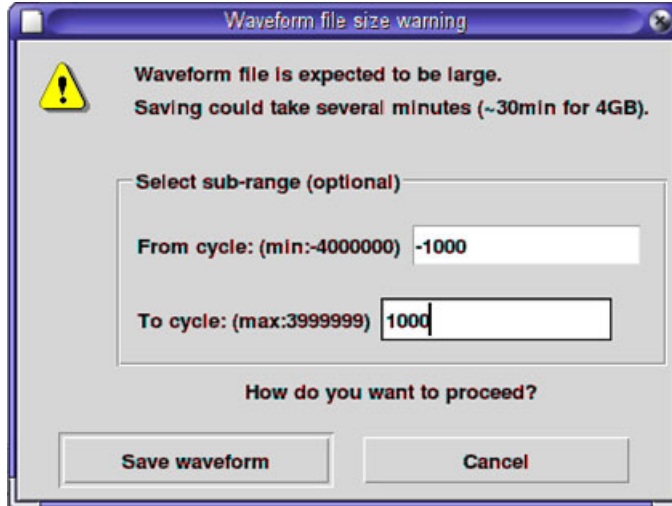
Signals Per MGB Link	MGB Connection Mux Ratio	Maximum Frequency	Max Sample Depth
1 to 32	1	70 MHz	268,435,455
33 to 64	2	35 MHz	134,217,727
65 to 128	4	17.5 MHz	67,108,863
129 to 256	8	8.75 MHz	33,554,431

## Viewing Captured Deep Trace Debug Samples

A large sample depth translates to large VCD/FSDB dump files. For these cases, the debugger includes the option of viewing or writing out a slice of the FSDB or VCD waveform based the number of captured cycles.

To write out a slice of the waveform:

1. Launch the debugger with the exported runtime environment from the operating system (see [Invoking the Debugger, on page 18](#)).
2. In the debugger GUI, open the design definition file (debug.prj).
3. Click the Waveform Display icon. If the sample depth is set to more than 8000000, the tool displays a popup window.



4. In the pop-up window:
  - Specify the cycle range to view on either side of the trigger position. The following example shows a sub-range of -1000 to 1000 specified, although the complete VCD/FSDB extends from -4000000 to 3999999 on either side of the trigger position.
  - Click Save waveform at the bottom of the dialog box to save and view the specified sub-range. If you click the button without specifying a sub-range, the tool saves the entire waveform to IICE.vcd or IICE.fsdb. This could take some time, as it downloads the full buffer depth and all the sample blocks. A full VCD dump can take hours.
5. Alternatively, write out vcd or fsdb using the `-range` argument with the appropriate TCL command:

```
write vcd -range {fromCycle toCycle} filename.vcd
write fsdb -range {fromCycle toCycle} filename.fsdb
```

## Verifying the Hardware Configuration

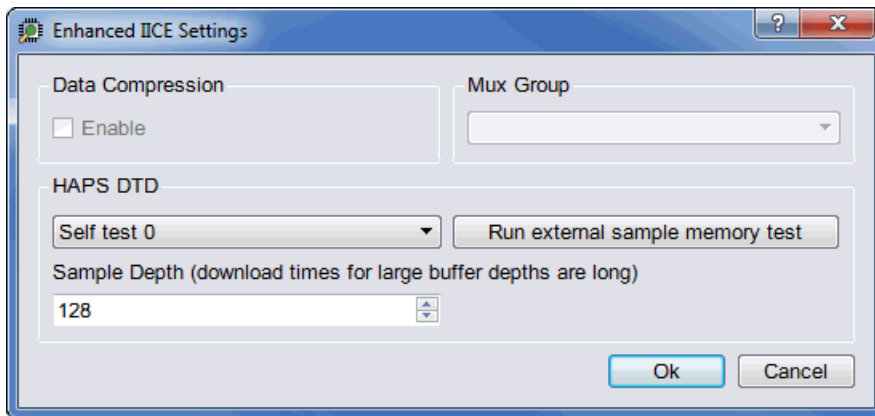
A self-test is available for verifying the deep trace debug hardware configuration. The self-test writes data patterns to the memory and reads back the data pattern written to detect configuration errors and connectivity problems.

The self test is normally executed:

- following the initial setup to verify the hardware configuration against the instrumentation
- during routine operations whenever a hardware problem is suspect
- whenever the physical configuration is modified (changing any of the IICE Sampler dialog box configuration settings)

To run the self-test from the debugger GUI:

1. Click the IICE icon.
2. Select one of the two patterns (pattern 0 or pattern 1) from the Self-test drop-down menu.
3. Click the Run external sample memory test button.



Selecting 0 uses one test pattern, and selecting 1 uses another pattern. To ensure adequate testing, repeat the command using alternate pattern settings.

The self-test can also be run from the command line using the following syntax:

**iice sampler -runselftest 1|0**



## CHAPTER 3

# Using the Debugger

---

The debugger enables HDL designs to be analyzed by interacting with the instrumented HDL design implemented in the target hardware system. You can activate breakpoints and watchpoints to cause trigger events within the IICE on the target device. These triggers cause signal data to be captured in the IICE. The data is then transferred to the debugger through a communications port where it can be displayed in a variety of formats.

This chapter describes:

- [Invoking the Debugger](#), on page 18
- [Debugger Windows and Panels](#), on page 19
- [Debugger GUI Operations](#), on page 28
- [Debugging on a Different Machine](#), on page 52
- [Waveform Display](#), on page 54
- [Logic Analyzer Interface Parameters](#), on page 57

## Invoking the Debugger

The debugger is launched directly from the Linux system by entering one of the following commands at the system prompt according to the product you are using and if you intend to use the Beta evaluation version of the debugger described in this document:

```
protocompiler_runtime_new debug -in pathToProjectFile
```

```
protocompiler_s_runtime_new debug -in pathToProjectFile
```

```
protocompiler_dx_runtime_new debug -in pathToProjectFile
```

In the above commands, the `-in` argument specifies the path to the project file in the exported runtime environment. If you do not use the `-in` argument, you must navigate to the *runtimeDirectory/designName/debug* directory and open the `debug.prj` file by either:

- Clicking the Open icon in the menu bar and, in the Open File dialog box, navigating to the export-runtime directory and clicking on the `debug.prj` file.
- Selecting File->Open from the main menu and, in the Open File dialog box, navigating to the export-runtime directory and opening the `debug.prj` file.

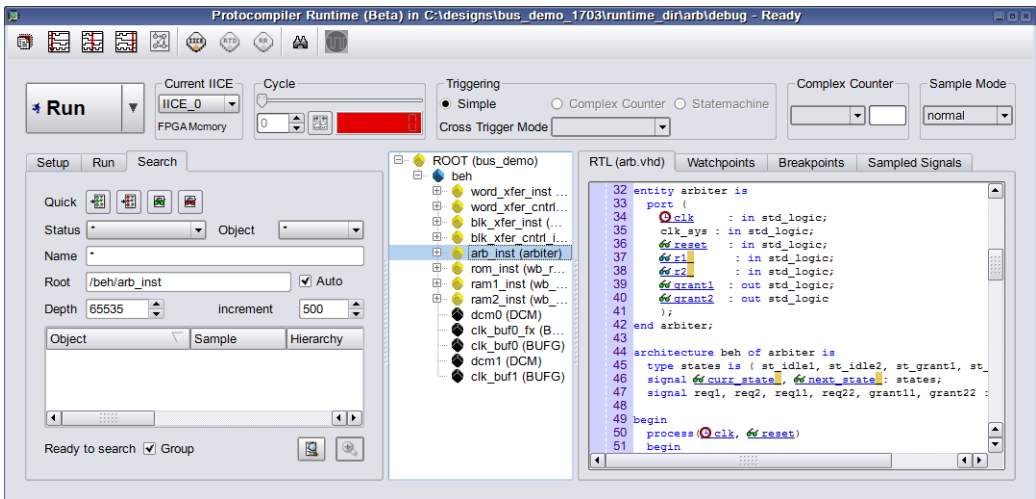
Selecting the `debug.prj` file opens the debugger instrumentation window (see [Debugger Windows and Panels, on page 19](#)) with the corresponding design HDL displayed (see [RTL Panel, on page 23](#)).

A Windows-based debugger executable can also be launched from a Windows command prompt or desktop shortcut.

# Debugger Windows and Panels

The Graphical User Interface for the debugger includes all of the required user controls, a hierarchy browser in the center, and a set of panels for various functions.

- [Setup Panel](#), on page 20
- [Run Panel and Run Button](#), on page 20
- [Search Panel](#), on page 21
- [RTL Panel](#), on page 23
- [Watchpoints Panel](#), on page 24
- [Breakpoints Panel](#), on page 24
- [Sampled Signals Panel](#), on page 25



In this section, each of these areas and their uses are described. The following discussions assume that:

- an HDL design has been loaded into the instrumentor and instrumented
- the design has been compiled with the instrumented data and the combined output netlist has been placed and routed by the place and route tool

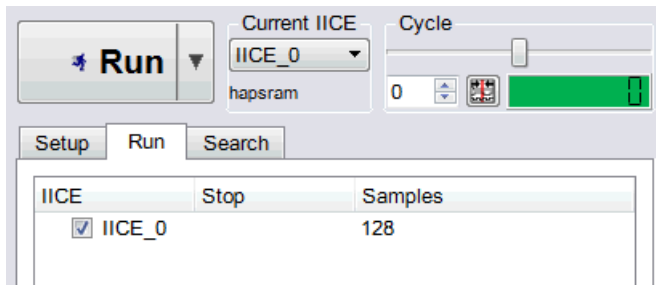
- the resultant bit file has been used to program the FPGA with the instrumented design
- the board containing the programmed FPGA is cabled to your host for analysis by the debugger

## Setup Panel

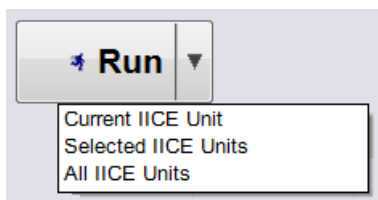
The Setup panel on the left side of the display provides access to three setup configuration menus that define the cable settings, show the CAPIM chaining, and run a hardware communications check (see [Configuring the Debugger, on page 12](#)). The Instrumentation section at the bottom of the panel shows the inherited settings that define the IICE; these settings can only be changed from within the instrumentor.

## Run Panel and Run Button

The Run panel reports the results of a run according to the watchpoints and breakpoints set.



A run is initiated by either clicking the Run button directly above the Run panel or by selecting Debugger->Run from the main ProtoCompiler Runtime menu. The Run button drop-down menu determines the unit or units selected and initiates the run.



When initiated, the command sends watchpoint and breakpoint activations to the IICE, waits for the trigger to occur, receives data back from the IICE when the trigger occurs, and then updates the watchpoint/breakpoint data in the RTL panel.

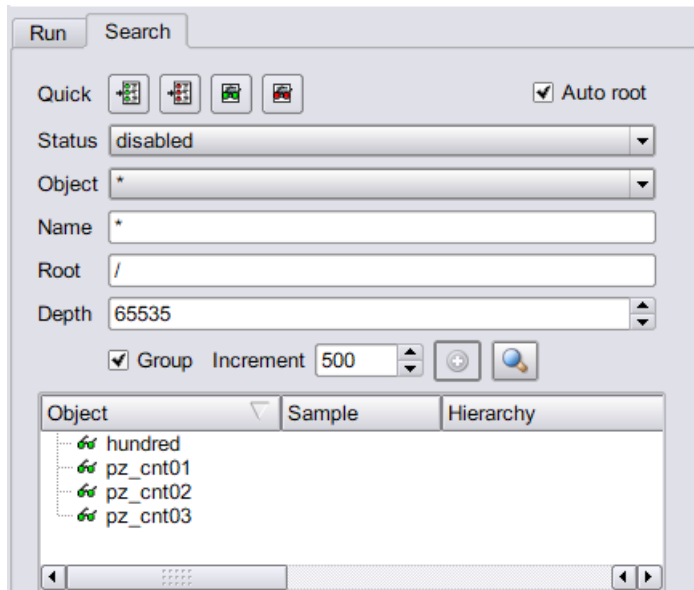
The Run panel includes three fields:

- IICE – displays the IICE unit or units selected for the run. Clicking an adjacent checkbox enables the IICE unit
- Stop – displays a Stop sign icon while the run is in progress; clicking the icon prematurely stops the run.
- Samples – displays the number of samples collected when the run is stopped.

If data compression is to be used on the sample data, refer to [Sampled Data Compression, on page 41](#). After the Run command is executed, the sample of signal values at the trigger position is annotated to the RTL code in the RTL panel. This data can be displayed in a waveform viewer or written out to a file (see the debugger waveform and write vcd command descriptions in the *Debug Environment Reference Manual*).

## Search Panel

The Search panel is a general utility to search for signals, breakpoints, conditions and/or instances. The panel includes an area for specifying the objects to find and an area for displaying the results of the search.



The search criteria in the upper section of the panel includes these options:

- **Quick** – icons that preset the conditions for instrumented or non-instrumented watchpoints or breakpoints.
- **Status** – specifies the status of the object to be found from the drop-down menu. The values can be enabled, disabled, sample\_only, readback, or “\*” (any). The default is “\*” (any).
- **Object** – specifies the type of object to search for from the drop-down menu: breakpoint, watchpoint, instance, condition or “\*” (any). The default is “\*” (any).
- **Auto** – when checked, fills in the root hierarchy.
- **Name** – specifies a name, or partial name to search for in the design. Wild cards are allowed in the name. The default is “\*” (any).
- **Root** – specifies the location in the design hierarchy to begin the recursive search. Root (“/”) is the default setting.
- **Depth** – specifies the depth of the sample buffer to be searched.
- **Results increment** – adds *results increment* items to the displayed list from the value set. Click the View more search results button to the right of the Execute the search button to add more results to the list.

- Group – indicates to expand search to all groups when checked.

The search results in the lower section of the panel show each object found along with its location. In addition, for breakpoints and watchpoints, the results section includes the corresponding icon (watchpoint or breakpoint) that indicates the instrumentation status of the qualified signal or breakpoint.

## RTL Panel

The RTL panel displays the HDL source code annotated with signals and breakpoints that were previously instrumented.

---

**Note:** Signals and breakpoints that were not enabled in the instructor are not displayed in the debugger.

---

Signals that can be selected for setting watchpoints are underlined, colored in blue text, and have small watchpoint (or “P”) icons next to them. Breakpoints that can be activated have small green circular icons in the left margin to the left of the line number.

```

32 entity arbiter is
33   port (
34     Ⓢ clk      : in std_logic;
35     clk_sys   : in std_logic;
36     Ⓢ reset'1' : in std_logic;
37     r1        : in std_logic;
38     r2        : in std_logic;
39     Ⓢ grant1   : out std_logic;
40     Ⓢ grant2   : out std_logic
41   );
42 end arbiter;
43
44 architecture beh of arbiter is
45   type states is ( st_idle1, st_idle2, st_grant1, st_grant2);
46   signal Ⓢ curr_state st_idle1, Ⓢ next_state st_idle2 : states;
47   signal req1, req2, req11, req22, grant11, grant22 : std_logic;
48
49 begin
50   process(Ⓢ clk, Ⓢ reset'1')

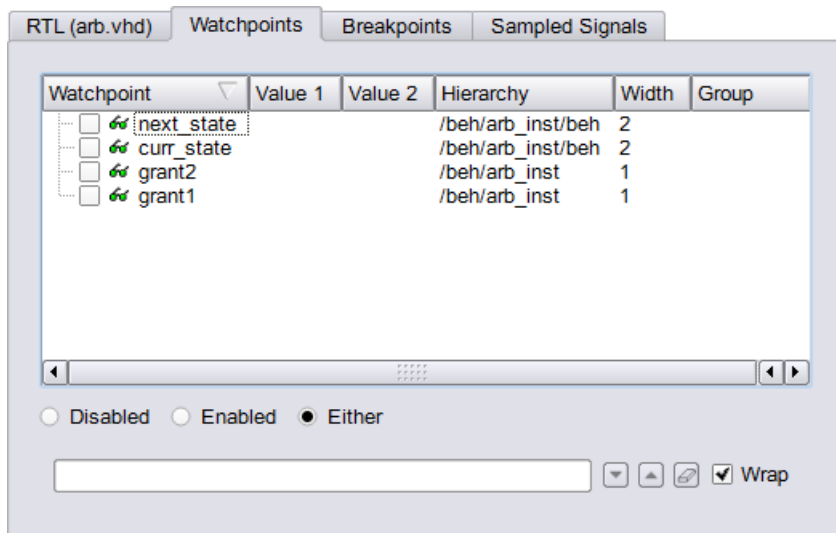
```

Selecting a watchpoint or “P” icon next to a signal (or the signal itself) allows you to select the Watchpoint Setup dialog box from the popup menu. This dialog box specifies a watchpoint expression for the signal. See [Setting a Watchpoint Expression](#), on page 33.

Selecting the green breakpoint icon to the left of the source line number causes that breakpoint to become armed when the run command is executed. See [Run Command](#), on page 39.

## Watchpoints Panel

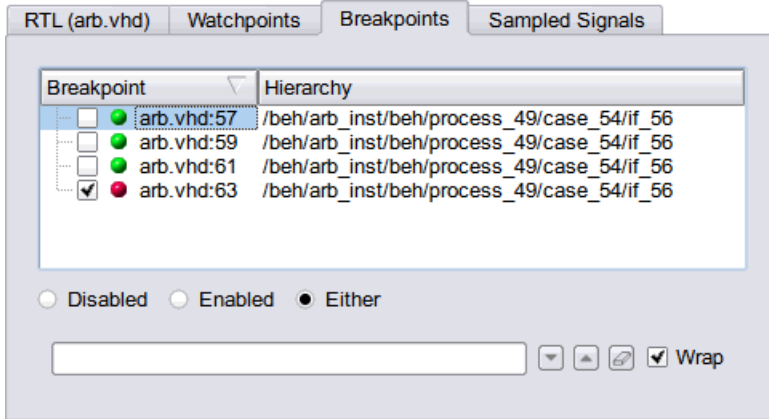
The watchpoint panel lists all of the active and inactive watchpoints in the design. The icons of active watchpoints are highlighted in red with an adjacent marked check box. Inactive watch points are green regardless of their type. Values for the active watchpoints are listed.



## Breakpoints Panel

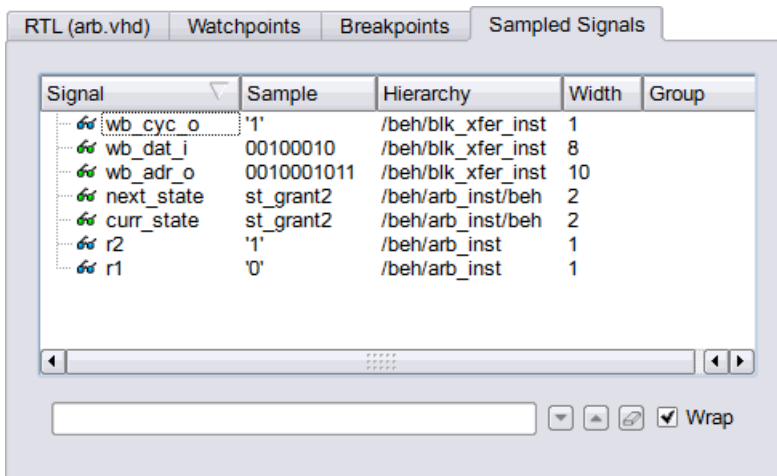
The breakpoints panel lists all of the active and inactive breakpoints in the design. The icons of active breakpoints are highlighted in red with an adjacent marked check box. Inactive watch points are green regardless of their type. Values for the active watchpoints are listed.





## Sampled Signals Panel

The Sampled Signals panel lists all of the sampled signals and their values when the sample trigger occurred.

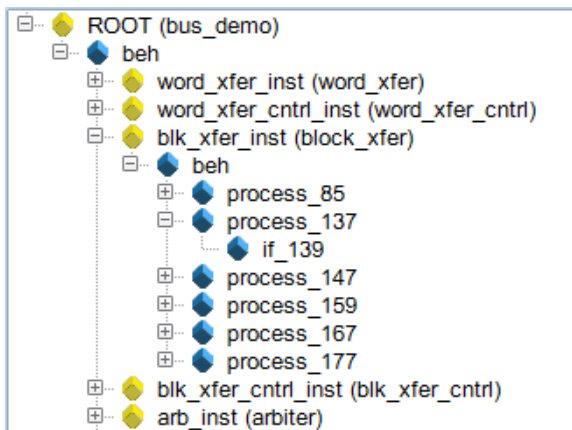


## Hierarchy Browser

The hierarchy browser shows a graphical representation of the design's hierarchy. At the top of the browser is the ROOT node. The ROOT node represents the top-level entity or module of your design. For VHDL designs,

the first level below the ROOT is the architecture of the top-level entity. The level below the top-level architecture for VHDL designs, or below the ROOT for Verilog designs, shows the entities or modules instantiated at the top level.

Clicking on a + sign opens the entity/module instance so that the hierarchy below that instance can be viewed. Lower levels of the browser represent instantiations, case statements, if statements, functional operators, and other statements.



Single clicking on any element in the hierarchy browser causes the associated HDL code to be displayed in the adjacent source code window.

## Tcl Script Window

The Tcl Script window displays commands that have been executed, including those executed by menu selections and button clicks. The debugger console window also allows you to type debugger commands and to view the results of command execution.

```

). Is Intel FPGA Quartus-II 7.0 or later installed on this system? This error m
Quartus tool path is not available in the PATH environment variable. Please incl
location to PATH variable in the
machine. The path should be set to the directory which contains the sld_hapi.dll
bin directory). For example:
With environmental variable QUARTUS_ROOTDIR set to "C:\altera\91\quartus", Ensur
includes "QUARTUS_ROOTDIR\bin64"
  
```

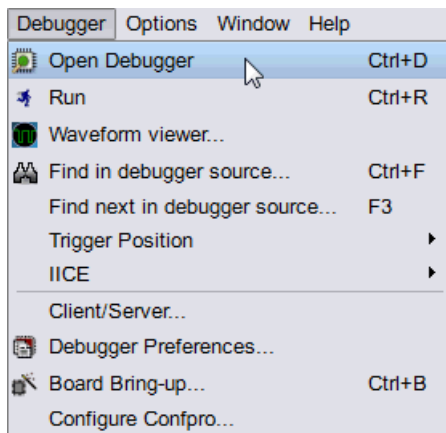
To capture all text written to the Tcl Script window, use the `log console` command, and to capture all commands executed in the Tcl Script window, use the `transcript` command. To clear the text from the console window, use the `clear` command (see the *Debug Environment Reference Manual* for command descriptions and syntax).

# Debugger GUI Operations

In addition to the panels previously described, the graphical interface includes functions and icons as well as the top-level menu bar.

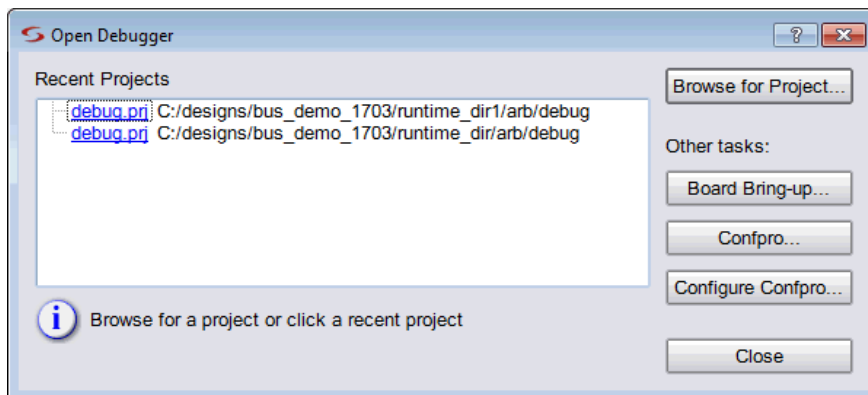
## Debugger Main Menu

Clicking Debugger in the top menu bar displays a drop-down menu with the following selections:



## Open Debugger

Selecting Open Debugger (or Ctrl-d) brings up the Open Debugger dialog box.



Function	Description
Recent Projects	Displays a list of recent projects; clicking a debug.prj link opens the project.
Browse for Project	Opens a browser to select the location of a different debug.prj file.
Board Bring-up	Opens the board bring-up utility for validating HAPS-based board systems; see <a href="#">Chapter 4, Board Bring-up</a> .
Confpro	Opens the HAPS Configuration Tool (Confpro); see the <i>System Configuration Software</i> manual on HAPS SupportNet.
Configure Confpro	Displays the Configure Confpro dialog box to specify the location of the Confpro installation.

## Run

Selecting Run initiates a run sequence by sending watchpoint and breakpoint activations to the IICE and waiting for a trigger. See [Run Panel and Run Button, on page 20](#) for more information.

## Waveform Viewer

The Waveform selection is enabled following a run sequence and displays the resulting waveform data in the specified waveform viewer. Clicking the Waveform viewer icon is equivalent to this selection. For more information, see [Waveform Display, on page 54](#).

## Find in debugger source

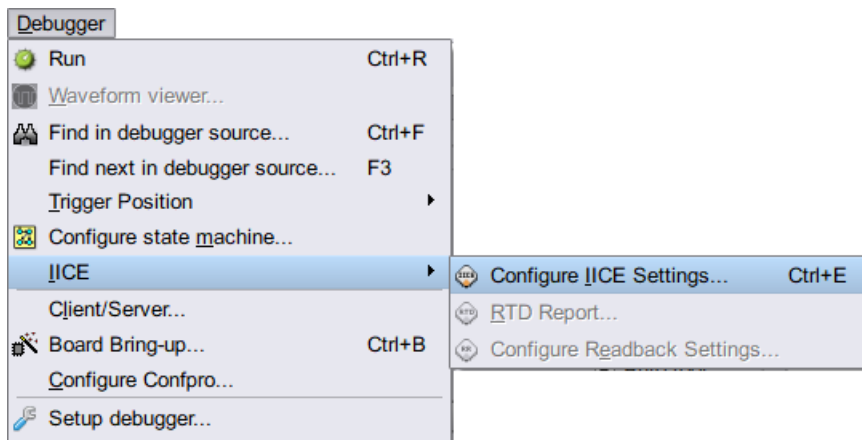
The Find dialog box locates text strings in the RTL source code. The dialog box is only valid for the RTL panel. Selecting Find next in debugger source locates the next occurrence of the search pattern.

## Trigger Position

Sets the position of the trigger within the sample buffer. The trigger position selection also includes a Configure state machine selection which is equivalent to the icon. For more information on trigger position, see [Sample Buffer Trigger Position, on page 42](#).

## IICE

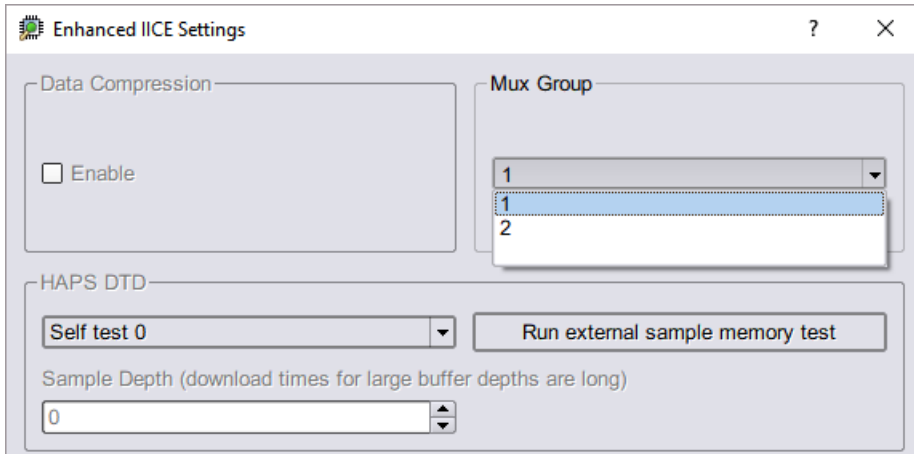
Brings up a submenu to select the IICE configuration.



Configure IICE Settings brings up the Enhanced IICE Settings dialog box for enabling data compression (see [Sampled Data Compression, on page 41](#)), selecting a multiplexed group (see [Selecting Multiplexed Instrumentation Sets, on page 36](#)), and running a DTD self test (equivalent to clicking the Configure IICE Settings icon).

RTD Report displays the signal/breakpoint interface assignments to the Mictor connector (see [IICE Assignments Report Tab, on page 59](#)).

The Configure Readback Settings selection is not supported in the beta evaluation version of the debugger.



## Client/Server

Brings up the Configure Client/Server dialog box to set up the connection between the debugger and the instrumented device through the cable. For detailed information, see [Debugger Configuration, on page 105](#).

## Debugger Preferences

Brings up the Debugger Preferences dialog box (equivalent to clicking the Preferences icon). The General tab includes activation controls and run settings; the Waveform tab sets up the waveform viewer (see [Waveform Display, on page 54](#)).

## Board Bring-up

Launches a special debugger window where you can either use the debugger graphical interface or enter a series of haps utility and board-test commands to set board parameters and run individual board tests to verify the board configuration for HAPS-60 and HAPS-70 series systems.

## Configure Confpro

Sets up the Confpro installation required by the board bring-up utility. For more information, see *System Configuration Software Handbook* in your installation directory.

# Basic Debugger Operations

Basic debugger operations include:

- [Executing a Script File](#), on page 32
- [Activating/Deactivating an Instrumentation](#), on page 32
- [Selecting Multiplexed Instrumentation Sets](#), on page 36
- [Activating/Deactivating Folded Instrumentation](#), on page 37
- [Run Command](#), on page 39
- [Sampled Data Compression](#), on page 41
- [Sample Buffer Trigger Position](#), on page 42
- [Sampled Data Display Controls](#), on page 43
- [Saving and Loading Activations](#), on page 47
- [Cross Triggering](#), on page 49
- [Listing Watchpoints and Signals](#), on page 51

## Executing a Script File

A script file contains Tcl commands and is a convenient way to capture a command sequence that you would like to repeat. To execute a script file, select the File->Execute Script menu selection and navigate to your script file location or use the source command (see *source* command in the *Debug Environment Reference Manual*).

## Activating/Deactivating an Instrumentation

The trigger conditions used to control the sampling buffer comprise breakpoints, watchpoints, and counter settings (see [Chapter 5, IICE Hardware Description](#)). Activation and deactivation of breakpoints and watchpoints are discussed in this section.



## Setting a Watchpoint Expression

Any signal that has been instrumented for triggering can be activated as a watchpoint in the debugger. A watchpoint is defined by assigning it one or two HDL constant expressions. When a watched signal changes to the value of its watchpoint expression, a trigger event occurs.



A watchpoint is set on a signal by clicking-and-holding on the signal or the watchpoint icon next to the signal and then selecting the Set trigger expressions menu item to bring up the Watchpoint Setup dialog box.

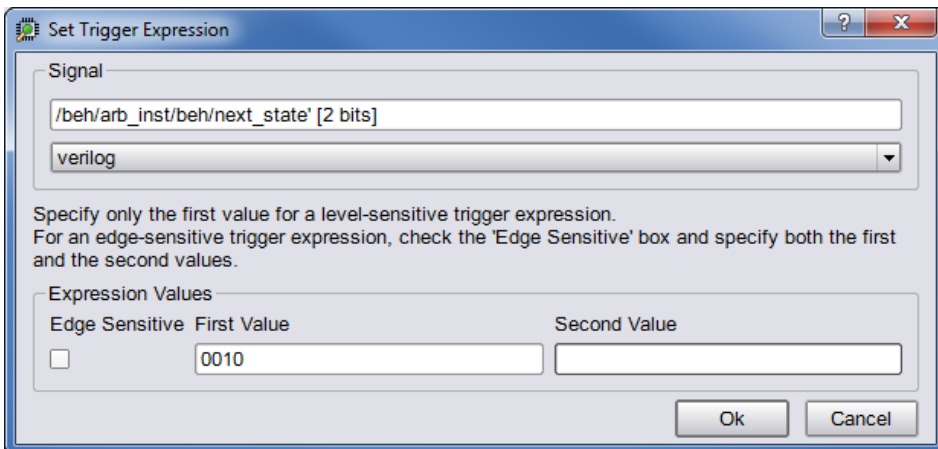


A watchpoint is set on a partial bus signal by clicking-and-holding on the signal or the “P” icon next to the signal, selecting the partial bus group from the list displayed, and then selecting the Set Trigger Expressions menu item to bring up the Watchpoint Setup dialog box.

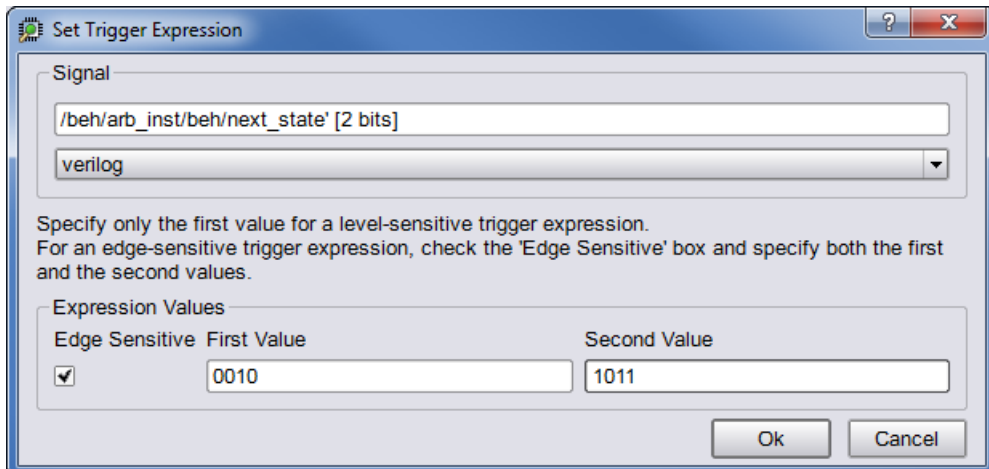
There are two forms of watchpoints: value and transition.

- A *value* watchpoint triggers when the watched signal attains a specific value.
- A *transition* watchpoint triggers when the watched signal has a specific value transition.

To create a value watchpoint, assign a single, constant expression to the watchpoint. A value watchpoint triggers when the watched signal value equals the expression. In the example below, the signal is a 4-bit signal, and the watchpoint expression is set to “0010” (binary). Any legal VHDL or Verilog (as appropriate) constant expression is accepted.



To create a transition watchpoint, enable the Edge Sensitive check box and assign two constant expressions to the watchpoint. A transition watchpoint triggers when the watched signal value is equal to the first expression during a clock period and the value is equal to the second expression during the next clock period. In the example below, the transition being defined is a transition from “0010” to “1011.”



The VHDL or Verilog expressions that are entered in the Watchpoint Setup dialog box can also contain “X” values. The “X” values allow the value of some bits of the watched signal to be ignored (effectively, “X” values are don’t-care values). For example, the above value watchpoint expression can be specified as “X010” which causes the watchpoint to trigger only on the values of the three right-most bits.

Hexadecimal values can additionally be entered as watchpoint values using the following syntax:

**x"hexValue"**

As shown, a hexadecimal value is introduced with an x character and the value must be enclosed in quotation marks. Similarly, you can include a hexadecimal entry in an equivalent Tcl command by literalizing the quote marks with back slashes as shown in the following example:

```
watch enable -iice IICE -condition 0 /structural/reg_fout x\"aa\"
```

Clicking OK on the Watchpoint Setup dialog box activates the watchpoint (the watchpoint or “P” icon changes to red) which is then armed in the hardware the next time the Run button is pressed.

## Deactivating a Watchpoint

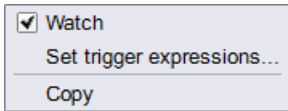
By default, a watchpoint that does not have a watchpoint expression is inactive. A watchpoint that has a watchpoint expression can be temporarily deactivated. A deactivated watchpoint retains the expression entered, but is not armed in the hardware and does not result in a trigger.



To deactivate a watchpoint, click-and-hold on the signal or the associated watchpoint icon. The watchpoint popup menu appears.



To deactivate a partial-bus watchpoint, click-and-hold on the signal or the associated “P” icon and select the bus segment from the list of segments displayed. The watchpoint popup menu appears.



The Watchpoints menu selection will have a check mark to indicate that the watchpoint is activated. Click on the Watchpoints menu selection to toggle the check mark and deactivate the watchpoint.

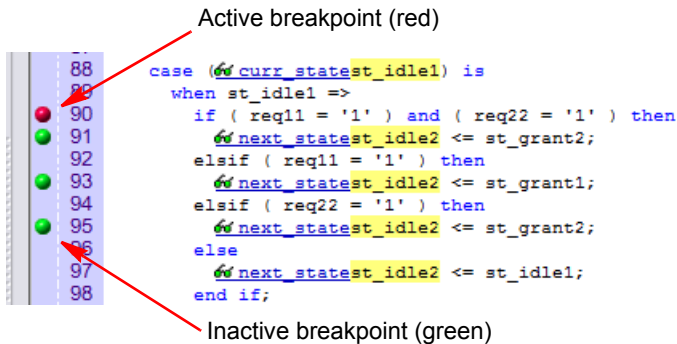
Watchpoint	Value 1	Value 2	Hierarchy
<input type="checkbox"/> next_state			/beh/arb_inst/beh
<input type="checkbox"/> curr_state			/beh/arb_inst/beh
<input type="checkbox"/> grant2			/beh/arb_inst
<input checked="" type="checkbox"/> grant1	1'bx		/beh/arb_inst

## Reactivating a Watchpoint

To reactivate an inactive watchpoint, click-and-hold on the signal or the associated watchpoint or “P” icon. Clicking the watchpoint icon redisplay the watchpoint popup menu: clicking the “P” icon, lists the partial bus segments; select the bus segment from the list displayed to display the watchpoint popup menu. Click on the Watch menu selection to toggle the check mark and reactivate the watchpoint.

## Activating a Breakpoint

Instrumented breakpoints are shown in the debugger as green icons in the left margin adjacent to the source-code line numbers. Green breakpoint icons are inactive breakpoints, and red breakpoint icons are active breakpoints. To activate a breakpoint, click on the icon to toggle it from green to red; to deactivate an active breakpoint, click on the breakpoint icon to toggle it from red to green.



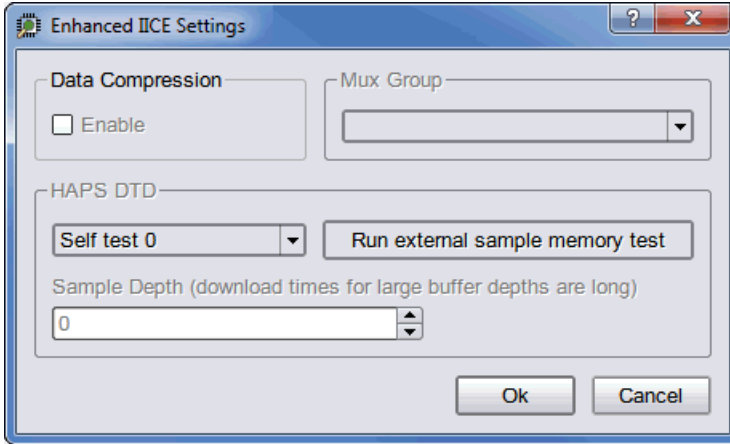
## Selecting Multiplexed Instrumentation Sets

Multiplexed groups of instrumented signals defined in the instrumentor can be individually selected for activation in the debugger (for information on defining a multiplexed group in the instrumentor, see *Multiplexed Groups* in the *HAPS®ProtoCompiler Instrumentor User Guide*).

Using multiplexed groups can substantially reduce the amount of pattern memory required during debugging when all of the originally instrumented signals are not required to be loaded into memory at the same time.

To activate a predefined multiplexed group in the debugger:

1. Select Debugger->IICE->Configure IICE Settings or click the Configure IICE Settings icon to display the dialog box.



2. Use the drop-down menu in the Mux Group section to select the group number to be active for the debug session.
3. The signals group command can be used to assign groups from the console window (see the *signals* command in the *Debug Environment Reference Manual*).

## Activating/Deactivating Folded Instrumentation

If your design contains entities or modules that are instantiated more than once, the design is termed to have a “folded” hierarchy (folded hierarchies also occur when multiple instances are created within a generate loop). By definition, there will be more than one instance of every signal and breakpoint in a folded entity or module. During instrumentation, it is possible to instrument more than one instance of a signal or breakpoint.

When debugging an instrumented design with multiple instrumented instances of a breakpoint or signal, the debugger allows you to activate/deactivate each of these instrumented instances independently. Independent selection is accomplished by displaying a list of the instrumented instances when the breakpoint or signal is selected for activation/deactivation.

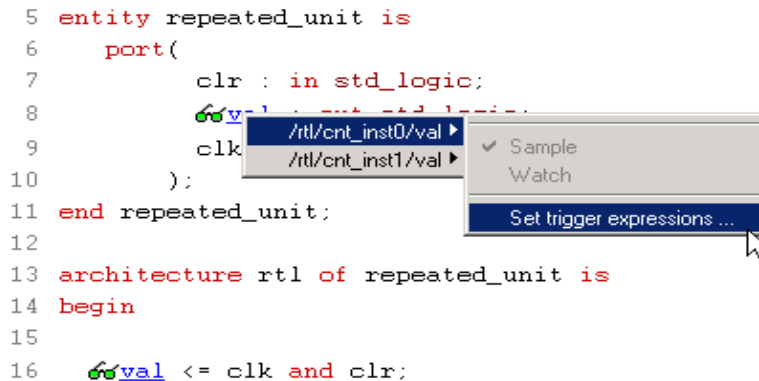
## Activating/Deactivating a Folded Watchpoint

The following example consists of a top-level entity called `folded2` and two instances of the `repeated_unit` entity. The source code of `repeated_unit` is displayed. In this folded entity, multiple instances of the signal `val` and the breakpoint at line 24 (not shown) are instrumented.

To activate/deactivate instances of the `val` signal, select the watchpoint icon next to the signal. A list will pop up with the two instrumented instances of the signal `val` available for activation/deactivation:

```
/rtl/cnt_inst0/val
/rtl/cnt_inst1/val
```

Either of these instances is activated/deactivated by clicking on the appropriate line in the list box to bring up the watchpoint menu shown in the following figure.



The color of the watchpoint icon is determined as follows:

- If no instances of the signal are activated, the watchpoint icon is green in color.
- If some, but not all, instances of the signal are activated, the watchpoint icon is yellow in color.
- If all instances are activated, the watchpoint icon is red in color.

For related information on folded hierarchies, see *Sampling Signal in a Folded Hierarchy* in the *HAPS ProtoCompiler Instrumentor User Guide* and [Displaying Data from Folded Signals](#), on page 45.

## Activating/Deactivating a Folded Breakpoint

To activate/deactivate instances of the breakpoint on line 24, select the icon next to line number 24. A list will pop up with the two instrumented instances of the breakpoint available for activation/deactivation:

```
/rtl/inst0/rtl/process_18/if_20/if_23/repeated_unit.vhd:24  
/rtl/inst1/rtl/process_18/if_20/if_23/repeated_unit.vhd:24
```

Either of these instances can be activated/deactivated by clicking on the appropriate line in the list box.

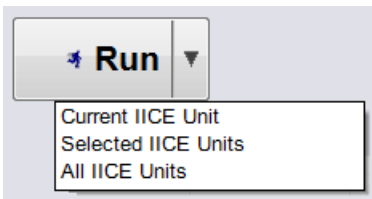
The color of the breakpoint icon is determined as follows:

- If no instances of the breakpoint are activated, the breakpoint icon is green.
- If some, but not all, instances of the breakpoint are activated, the breakpoint icon is yellow.
- If all instances are activated, the breakpoint icon is red.

## Run Command

The Run command sends watchpoint and breakpoint activations to the IICE, waits for the trigger to occur, receives data back from the IICE when the trigger occurs, and then displays the data in the source window.

To execute the Run command for the active IICE (or a single IICE), select Debugger->Run from the menu or click the Run button. If data compression is to be used on the sample data, see [Sampled Data Compression, on page 41](#). To execute the Run command for multiple IICE units, first enable the individual IICE units in the Run panel by checking their corresponding boxes, and then click either the large Run button or select Selected IICE Units from the Run menu.



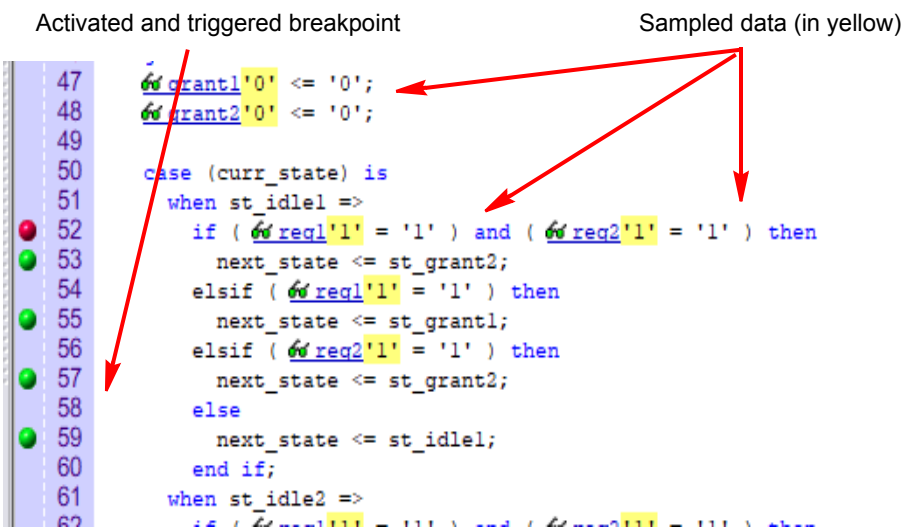
After the Run command is executed, the sample of signal values at the trigger position is annotated to the HDL code in the source code window. This data can be displayed in a waveform viewer with the debugger waveform command or written out to a file with the *write vcd* command (see the corresponding command descriptions in the *Debug Environment Reference Manual*).

---

**Note:** In a multi-IICE environment, you can edit and run other IICEs while an IICE is running.

---

The following example shows a design with one breakpoint activated, the breakpoint triggered, and the sample data displayed. The small green arrow next to the activated breakpoint in the example indicates that this breakpoint was the actual breakpoint that triggered. Note that the green arrow is only present with simple triggering.



## Stop Command

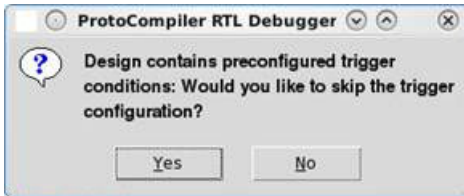


The Stop command on the Run panel sends control back to the debugger after you have armed the trigger, but before the trigger occurs.



## Pre-Configured Triggers

The pre-configure feature allows specific trigger conditions to be set in advance or to be *preset* into the debugger. Pre-configure triggers arm your IICE to run immediately after configuration in order to trap error conditions that can occur during the first several cycles. Defining a pre-configure trigger is controller by Tcl command options to the breakpoint, iice, or signals command. When running the debugger from the graphical interface, clicking the Run button displays the following prompt to indicate that a pre-configure trigger has been set in the instrumentor:



To download the sample data with the pre-configured trigger, select Yes. To ignore the pre-configured trigger and enable normal sampling, select No.

## Sampled Data Compression

A data compression mechanism is available to compress the sampled data to effectively increase the depth of the sample buffer without requiring any additional hardware resources. When enabled, data compression is applied to the sampled data to temporarily remove any data that remains unchanged between cycles (a sample is automatically taken after 64 unchanging cycles).

Data compression is enabled by clicking the Configure IICE Settings icon to display the Enhanced IICE Settings dialog box and clicking the Enable check box in the Data Compression section or from the Tcl command prompt by entering the following command:

```
iice sampler -datacompression 1
```

Data compression must be set prior to executing the Run command and applies to all enabled IICE units. Data compression is not available when using state-machine triggering, or qualified or always-armed sampling.

## Sample Data Masking

A masking option is available with data compression to selectively mask individual bits or buses from being considered as changing values within the sample data. This option is only available through the Tcl interface using the following syntax:

```
iice sampler -enablemask 0|1 [-msb integer -lsb integer] signalName
```

For example, the following command masks bits 0 through 3 of vector signal mybus[7:0] from consideration by the data compression mechanism:

```
iice sampler -enablemask 1 -msb 3 -lsb 0 mybus
```

Similarly, to reinstate the masked signals in the above example, use the command:

```
iice sampler -enablemask 0 -msb 3 -lsb 0 mybus
```

## Sample Buffer Trigger Position

The purpose of the activated watchpoints and breakpoints is to cause a trigger event to occur. The trigger event causes sampling to terminate in a controlled fashion. Once sampling terminates, the data in the sample buffer is communicated to the debugger and then displayed in the GUI.

The sample buffer is continuously sampling the design signals. Consequently, the exact relationship between the trigger event and the termination of the sampling can be controlled by the user. Currently, the debugger supports the following trigger positions relative to the sample buffer:

- Early
- Middle
- Late

Determining the correct setting for the trigger position is up to the user. For example, if the design behavior of interest usually occurs after a particular trigger event, set the trigger position to “early.”

The trigger position can be changed without requiring the design to be re-instrumented or recompiled. A new trigger position setting takes effect the next time the Run command is executed.

## Early Position



The sample buffer trigger position can be set to “early” so that the majority of the samples occurs after the trigger event. To set the trigger position to “early,” use the Debug->Trigger Position->early menu selection or click on the Set trigger position to early in the sample buffer icon.

## Middle Position



The sample buffer trigger position defaults to “middle” so that there is an equal number of samples before and after the trigger event. To set the trigger position to “middle,” use the Debug->Trigger Position->middle menu selection or click on the Set trigger position to the middle of the sample buffer icon.

## Late Position



The sample buffer trigger position can be set to “late” so that the majority of the samples occurs before the trigger event. To set the trigger position to “late,” use the Debug->Trigger Position->late menu selection or click on the Set trigger position to late in the sample buffer icon.

## Sampled Data Display Controls

The sampled data display controls are used to navigate through the data values captured by the sample buffer. All sample buffer data is tagged with a cycle number based on when the data item was stored in the sample buffer relative to the trigger event. The data item stored at the trigger event time has cycle number 0, the data item stored one sample clock cycle *after* the trigger has cycle number 1, and the data item stored one sample clock cycle *before* the trigger has cycle number -1. The data display procedures allow you to retrieve data values for a specific cycle number.

The sampled data displayed in the debugger is controlled by the Cycle numeric field. You can manually change the cycle number by typing a numeric value in the field. Also, the up and down arrows to the right of the cycle number increment or decrement the cycle number for each click.

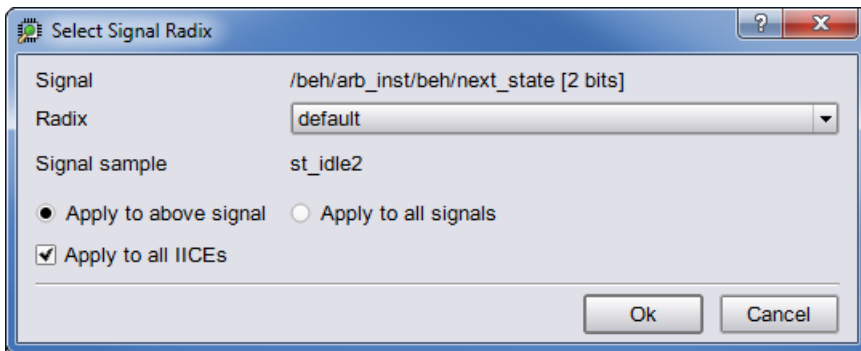


To reset the cycle number to the default position (the zero time position), click on the Goto trigger event in sample history icon.

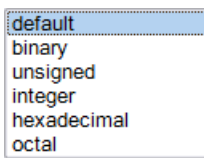
## Radix

The radix of the sampled data displayed can be set to any of a number of different numeric bases. To change the radix of a sampled signal:

1. Right click on the signal name or the watchpoint or “P” icon and select Change signal radix to display the following dialog box:



2. Select the desired radix from the Radix drop-down menu.



3. Click OK.

---

**Note:** You can change the radix before the data is sampled. The watch-point signal value will appear in the specified radix when the sampled data is displayed.

---

Specifying **default** resets the radix to its initial intended value. Note that the radix value is maintained in the “activation database” and that this information will be lost if you fail to save or reload your activation. Also, the radix set on a signal is local to the debugger and is not propagated to any of the waveform viewers.

---

**Note:** Changing the radix of a partial bus changes the radix for all bus segments.

---

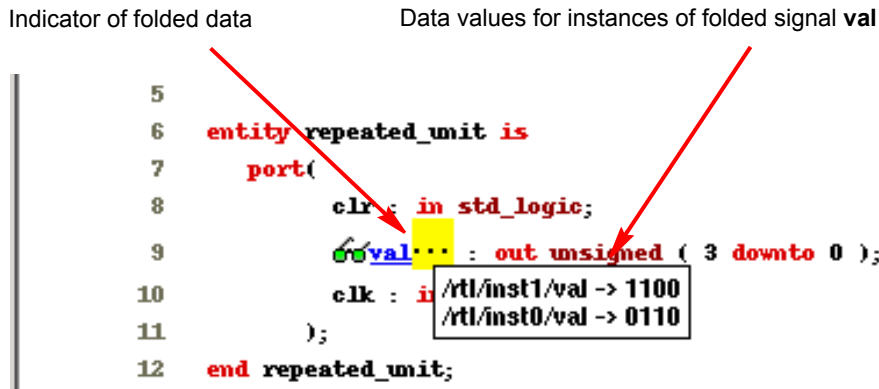
## Displaying Data from Folded Signals

If your design contains entities or modules that are instantiated more than once, it is termed to have a “folded” hierarchy (folded hierarchies also occur when multiple instances are created within a generate loop). By definition, there will be more than one instance of every signal in a folded entity or module. During instrumentation, it is possible to instrument more than one instance of a signal.

When debugging an instrumented design with multiple instrumented instances of a signal, the debugger allows you to display the data values of each of these instrumented signals.

Because multiple data values cannot be displayed at the same location, a single data value is always displayed. For multiply instrumented signals, the debugger displays an ellipsis (...) to indicate that there are multiple values present. To display all of the instrumented values, click-and-hold on the ellipsis indicator.

The example below consists of a top-level entity called `top` and two instances of the `repeated_unit` entity. In the example, the source code of `repeated_unit` is displayed, and both of the lists of instances of the signal `val` have been instrumented. The two instances are `/rtl/inst0/val` and `/rtl/inst1/val`, and their data values are displayed in the pop-up window as shown in the following figure:

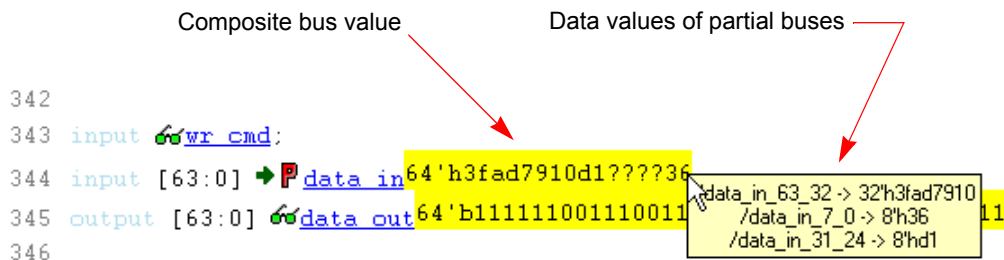


For related information on folded hierarchies, see *Sampling Signals in a Folded Hierarchy* in the *HAPS ProtoCompiler Instrumentor User Guide* and *Activating/Deactivating Folded Instrumentation*, on page 37.

## Displaying Data for Partial Buses

When debugging designs with partially instrumented buses, the debugger displays the data values of each of the instrumented segments.

To display the instrumented values for the individual bus segments, position the cursor over the composite bus value. The individual partial bus values are displayed in a tooltip in the specified radix as shown in the following figure.









Note that in the above figure, the question marks (?) in the composite bus value (64'h3fad7910d1????36) indicate that the corresponding segment (data\_in [23:8]) has not been instrumented.

## Displaying Data for Partial Instrumentation

In the debugger, the value for a fully instrumented record or structure is shown with a value for each field, in field order. The following figure shows instrumented signal `sig_iport_P_Struc_instr`. When displaying a partially instrumented bus, the value U is used for the uninstrumented slices. This same notation is used to show the data values for a partially instrumented record or structure (the value for each instrumented field is listed in field order, and an uninstrumented field value is shown as a U).

```

10 module uddt_P_Struc_tbttop (
11   input   clk_ip,
12   output type_Unsigned_P_Struc_data  sig_oport_P_Struc_data
13 );
14
15 logic           tb_rst 1'b1;
16 shortint      unsigned  rst_cnt 65535;
17
18 type_P_Struc_instr  sig_iport_P_Struc_instr CMP {{4'b0000} {4'b0010}}
19
20 always @ (posedge  clk_ip) //rst generation

```

The Find dialog in the debugger shows a partially instrumented signal with the P icon. You can set the trigger expressions on the fields instrumented for triggering in the same manner as if the signal was fully instrumented (that is, select the signal, right click to bring up the dialog, and select the option to set the trigger expression).

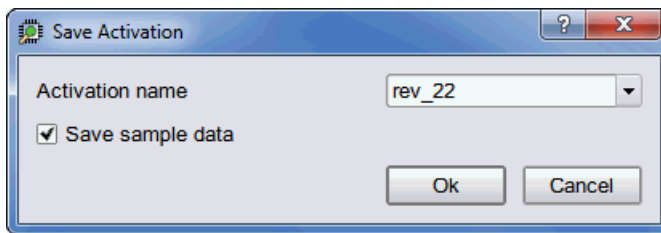
## Saving and Loading Activations

The debugger includes a “capture and replay” function that allows you to save and load a set of enabled watchpoints and breakpoints referred to collectively as an “activation.” Each activation can additionally include the sample data set that was captured for a given trigger condition. Activations are stored internally in files for each design.

## Saving an Activation

An activation can be explicitly saved or saved on exit. To explicitly save an activation:

1. Enable the set of watchpoints and breakpoints for the activation.
2. If the sample data set is to be included, run the debugger to collect the sample data.
3. Select File->Save Activation in the menu bar to bring up the following dialog box.



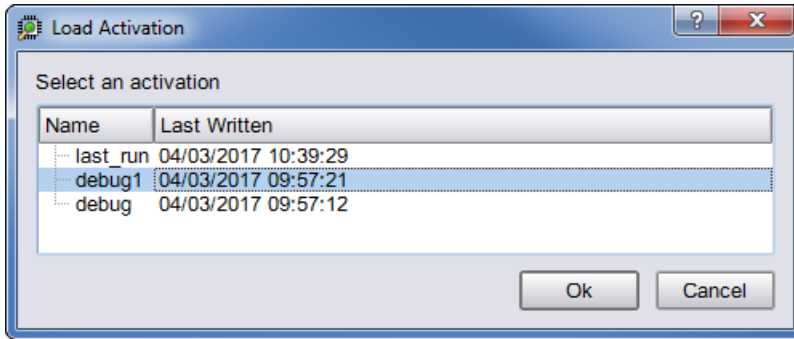
4. Enter (or select) an activation name in the adjacent field. Selecting an existing activation from the drop-down menu overwrites the selected activation.
5. To include the sample data set with the activation, enable the Save current sample data check box.
6. Click Yes to save the activation.

## Loading an Activation

To load an existing activation:

1. Select File->Load Activation from the main menu to display the Load Activation dialog box.





2. Highlight the activation to be loaded and click OK.

## Autosaving Current Activation

By default, when you exit the debugger without explicitly saving an activation, the active activation is automatically saved to the `last_run` file. This file is automatically loaded the next time you open the design.

---

**Note:** To save a specific activation, always use **Save current activations** to explicitly name the file and prevent the data from overwriting the `last_run` file.

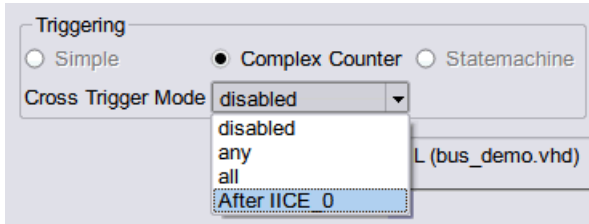
---

To disable the auto-save feature, uncheck the **Auto-save trigger settings** and **sample results check box** on the **Debugger Preferences** dialog box (select **Debugger->Debugger Preferences**).

## Cross Triggering

Cross triggering allows the trigger from one IICE unit to be used to qualify a trigger on another IICE unit, even when the two IICE units are in different time domains. Cross triggering is available in both the simple triggering and complex counter triggering modes (state-machine triggering supports cross triggering by allowing the IICE unit IDs to be included in the state-machine equations).

Cross triggering for an IICE unit is enabled in the instrumentor by selecting the Allow cross-triggering in IICE check box on the IICE Controller tab for the local IICE unit. The cross-trigger mode is selected from the drop-down menu in the debugger as shown below.



The drop-down menu selections are as follows:

The drop-down menu selections are as follows:

Menu Selection	Function
disabled	No triggers accepted from external IICE units (event trigger can only originate from local IICE unit)
any	Event trigger from local IICE unit occurs when an event at any IICE unit, including the local IICE unit, occurs
all	Event trigger from local IICE unit occurs when all events, irrespective of order, occur at all IICE units including the local IICE unit
after- <i>iiceName</i>	Event trigger from local IICE unit occurs only after the event at selected external IICE unit <i>iiceName</i> has occurred (external IICE units are individually listed)

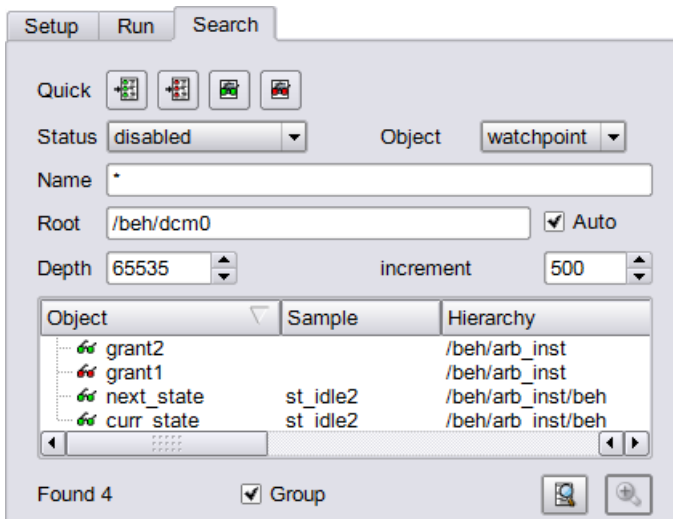
---

**Note:** If the drop-down menu does not display, make sure that Allow cross-triggering in IICE is enabled on the IICE Options tab of the instrumentor and that you have defined more than one IICE unit.

---

## Listing Watchpoints and Signals

To list watchpoints and signals in the debugger, use the Search panel.



The results are displayed in the lower portion of the dialog box. The Quick icons at the top of the dialog box display their corresponding values as follows:

## Show Watchpoint/Breakpoint Icons

The show watchpoint and breakpoint icons in the menu bar display their corresponding values in the Find Design Elements dialog box as follows:

### Search for disabled breakpoints



To display the disabled (inactive) breakpoints, click the Search for disabled breakpoints icon.

### Search for enabled breakpoints



To display the enabled (active) breakpoints, click the Search for enabled breakpoints icon.

### Search for disabled watchpoints



To display the disabled (inactive) watchpoints, click the Search for disabled watchpoints icon.

### Search for enabled watchpoints



To display the enabled (active) watchpoints, click the Search for enabled watchpoints icon.

## Debugging on a Different Machine

It is not unusual for the instrumentation phase and the debugging phase to be performed on different machines. For example, the debug machine is often located in a hardware lab. When a different machine is used for debugging, you must copy or transfer the exported runtime directory from the database to the lab machine.

Because the tool set allows you to debug your design in the HDL, the debugger must have access to the original source files. Depending on the type of your network, the debugger may be able to access the original sources files directly from the lab machine. If this is not possible or if the two computers are not networked, you must also copy the original sources to the debug machine. If the debugger cannot locate the original source files, it will open the design, but an error will be generated for each missing file, and the corresponding source code will not be visible in the source viewer.

Copying the source files to the debug machine can be done in two ways:

- The instrumentor can automatically include the original source files in the exported runtime directory so that when you transfer the directory to the lab machine, the original sources files (in the `orig_sources` subdirectory) are included. The debugger automatically looks in this directory for any missing source files. This preference is set before compiling the instrumented design by selecting Instrumentor->Instrumentation preference and making sure that Save original source in instrumentation directory is checked.
- The original source files can be manually copied to the lab machine or may already exist in a different location on this machine. In this case, it may be necessary to help locate the design files using the `searchpath` command. Simply call this command from the command line before loading the design file (`debug.prj`). The argument is a semi-colon-separated (Windows) or colon-separated (Linux) list of directories in which to find the original source files.

```
searchpath {d:/temp;c:/Documents and Settings/me/my_design/}
```

The debugger only displays files that match the CRC generated at the time of instrumentation.

---

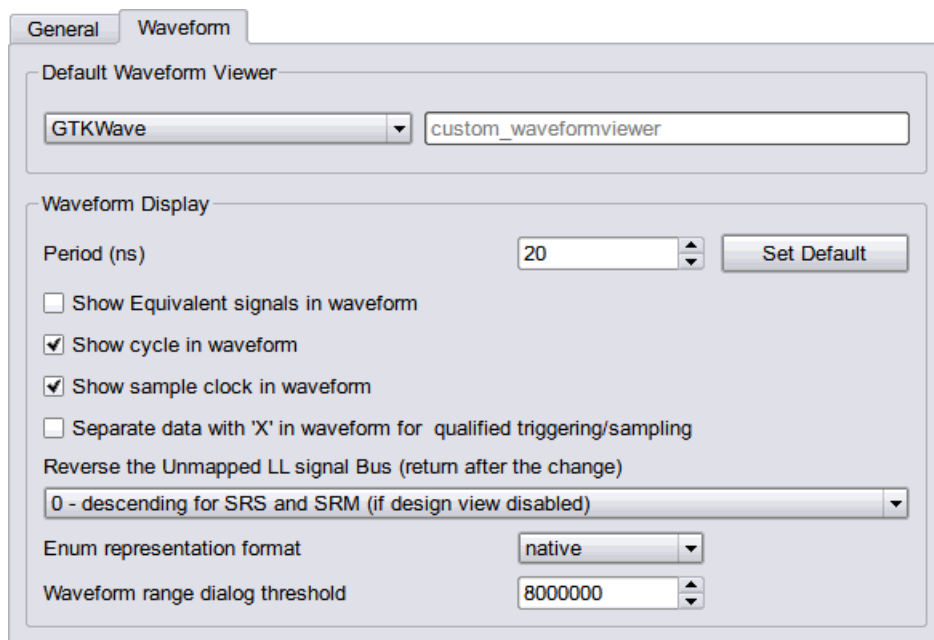
**Note:** If there are security issues with having the original source files on the lab machine, the instrumentor can password-protect the original sources on the development machine for use with the debugger (for information on file encryption, see *Including Original HDL Source* in the *HAPS ProtoCompiler Instrumentor User Guide*).

---

# Waveform Display

The waveform display control displays the sampled data in a waveform style. By default, this feature uses the Synopsys DVE waveform viewer. Provision for using other popular waveform viewers that support VCD data is included. Alternately, you can interface your own waveform viewer by writing a customized script to access your waveform viewer from the debugger. For details, see the application note, “Interfacing Your Waveform Viewer with the Debugger” on the SolvNetPlus website.

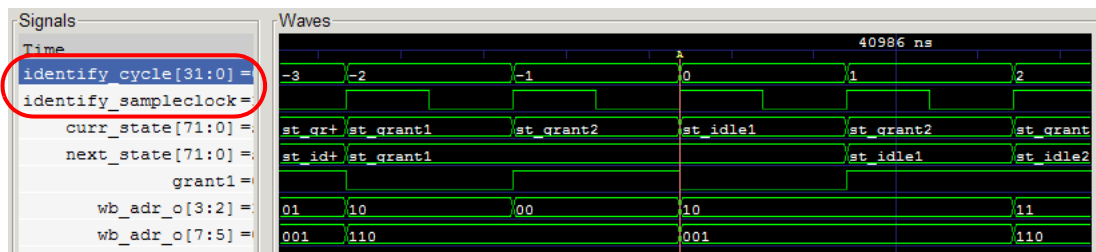
Viewer selection and setup are controlled by the Waveform Viewer Preferences dialog box. Selecting Debugger->Debugger Preferences from the menu bar brings up the dialog box shown below.



The Synopsys DVE waveform viewer is only available on Linux platforms. To use the included GTKWave viewer, click the GTKWave radio button in the Default Waveform Viewer section. When using the GTKWave viewer, Fast Signal Trace (FST) is a dumpfile format used by GTKWave which loads much faster than VCD. When GTKWave is invoked, the VCD file is first converted to FST format and then loaded into the waveform viewer. The Period field sets the period for the waveform display and is independent of the design speed.



After running the debugger, the selected waveform viewer is displayed by selecting Debugger->Waveform viewer from the menu or by clicking the Waveform viewer icon in the menu bar. All sampled signals in the design are included in the waveform display. Two additional signals are added to the top of the display when enabled by their corresponding check boxes. The first signal, `identify_cycle`, reflects the trigger location in the sample buffer. The second signal, `identify_sampleclock`, is a reference that shows every clock edge. The following figure shows a typical waveform view with the `identify_cycle` and `identify_sampleclock` signals enabled (highlighted in the figure).



If you select a waveform viewer from the Waveform preference dialog box that is not installed, an error message is displayed when you attempt to invoke the viewer. To install the waveform viewer:

1. Open the Debugger Preferences dialog box (select Options->Debugger preferences).
2. Select the desired waveform viewer from the drop-down menu and click OK.
3. Make sure that the selected simulator is installed on your machine and that the path to the executable is set by your \$PATH environment variable.

To invoke the viewer after running the debugger, select Window->Waveform or click on the Open Waveform Display icon.

## Generating the Fast Signal Database

The debugger is used to generate the fast signal database (FSDB) for the Verdi platform and for display by the Verdi nWave viewer. To generate this database:

1. Instrument the design with the essential signal list (see *Instrumenting the Verdi Signal Database* in the *HAPS ProtoCompiler Instrumentor User Guide*).
2. Run the instrumented design in the synthesis tool and load the design into the debugger.
3. Use the Debugger Preferences dialog box and make sure that Synopsys Verdi nWave is selected as the default waveform viewer.
4. Setup the trigger conditions and click the Run button to download the sample buffer.
5. Generate the fast signal database using the following command syntax:

```
write fsdb -iice iiceID -showequiv fsdbFilename
```

6. Click the Open Waveform Display icon to view the samples in the nWave viewer.

The fast signal database file (*fsdbFilename*) can be imported directly back to the Verdi platform.



# Logic Analyzer Interface Parameters



The logic analyzer interface parameters for the real-time debug feature in the debugger are defined on the tabs of the RTD type IICE information dialog box. To display this dialog box, select the RTD IICE as the current IICE (to enable the RTD icon) and then click on the icon (RTD type IICE Information/Settings) in the top menu.

## Logic Analyzer Scan Tab

The Logic Analyzer Scan tab defines:

- the logic analyzer type
- the TLA script program
- user name
- host name/IP address
- if pods are automatically assigned to Mictor connectors

Logic Analyzer Scan

Type of Logic Analyzer: tla

TLA Script Program: c:\Program Files\TLA 700\System\tlascript

User Name: user

Host Name/ IP Address: 10.9.148.51

☐ Assign Pods automatically to mictor conenctors

Scan Logic Analyzer

## Type of Logic Analyzer

Selects the type of logic analyzer from a drop-down menu. Current supported types are Agilent 16700 and 16900 series and Tektronix TLA series analyzers. The logic analyzer must be accessible on the local network.

## TLA Script Program

Specifies the full path to the `tlascript` script file on the Tektronix logic analyzer. The default path is `C:\Program Files\TLA 700\System\tlascript`. If this location does not match the location expected by the Tektronix logic analyzer, change the location setting. The logic analyzer requires an `rsh` daemon to access the script file. To download and install the `rsh` daemon on the logic analyzer, see the web-site at <http://rshd.sourceforge.net>.

## User Name

Identifies the user name on the analyzer (Tektronix only).

## Host Name/IP Address

Specifies the host name or IP address for the debugger host.

## Assign Pods automatically to Mictor connectors

When checked, automatically assigns pods to the Mictor connectors.

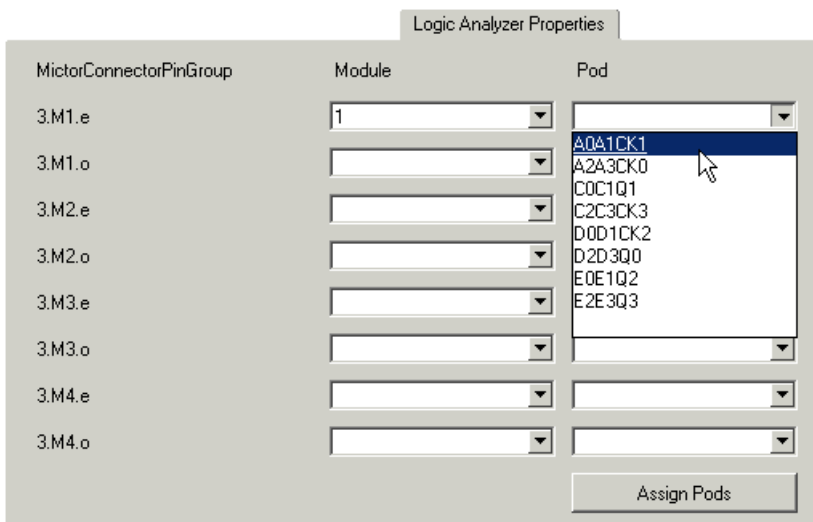
## Scan Logic Analyzer

Clicking the Scan Logic Analyzer button scans the specified IP address and, if scanned successfully:

- opens a network connection with the given parameters
- retrieves the modules and pods information
- displays Logic Analyzer Properties and Logic Analyzer Submit tabs

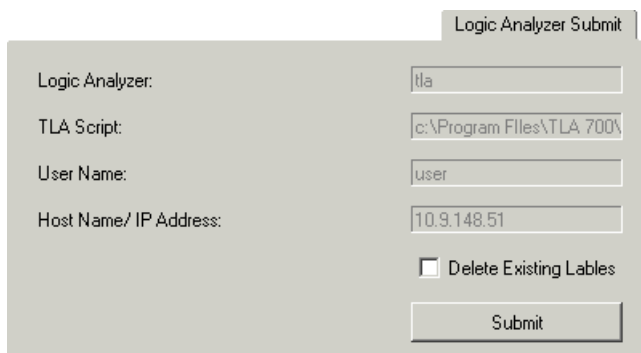
## Logic Analyzer Properties Tab

The Logic Analyzer Properties tab allows Mictor pin groups to be manually assigned to modules and pods using corresponding drop-down menus. Clicking the Assign Pods button updates the assignments.



## Logic Analyzer Submit Tab

The Logic Analyzer Submit tab submits signal/breakpoint names to the logic analyzer.



## IICE Assignments Report Tab

When using the real-time debugging feature in the instrumentor (see *RealTime Debugging* in the *HAPS ProtoCompiler Instrumentor User Guide*), the signal/breakpoint interface assignments to the Mictor connector are reported

in the debugger on the IICE Assignments Report tab. Clicking the tab before assigning logic analyzer pods to the Mictor pin groups reports only the signal/breakpoint assignments. Clicking the tab after assigning logic analyzer pods to the Mictor pin groups includes the pods assignments in the report.

By default, the report is displayed on the screen (standard out). The report can be redirected to a file using the `iice assignmentsreport` Tcl command (see the `iice` command in the *Debug Environment Reference Manual*).

IICE Assignments Report		
RTD type IICE 'IICE_1' Assignments Report		
Signal/breakpoint Assignments		
Mictor Pin	Fpga Pin	Signal/Breakpoint
mictor_clock_pinloc	Unknown	mapped_nothing
10_11_12.M1.D3e	AP11	/beh/blk_xfer_inst/beh/cntrl_adr_tmp[7]
10_11_12.M1.D4e	AP13	/beh/blk_xfer_inst/beh/cntrl_adr_tmp[6]
10_11_12.M1.D5e	AN13	/beh/blk_xfer_inst/beh/cntrl_adr_tmp[5]
10_11_12.M1.D6e	AN11	/beh/blk_xfer_inst/beh/cntrl_adr_tmp[4]
10_11_12.M1.D7e	AN12	/beh/blk_xfer_inst/beh/cntrl_adr_tmp[3]

## CHAPTER 4

# Board Bring-up

---

A set of board bring-up utilities are available to validate the HAPS<sup>®</sup> board configuration. See the following for details:

- [Running the Board Bring-up Utility](#), on page 62
- [Board Configuration Tests](#), on page 64
- [Utility Commands](#), on page 68

## Running the Board Bring-up Utility

To assist in system board validation, the Confpro tool set is used to perform verification of the newly-defined board including HAPS hardware checks, UMRBus® bus access verification, clock and reset checks, HSTDM performance, and a self-test. Twelve individual checks are available from the Tcl Script window; four checks are additionally available directly from the Board Bring-up dialog box. Before running the board bring-up utility:

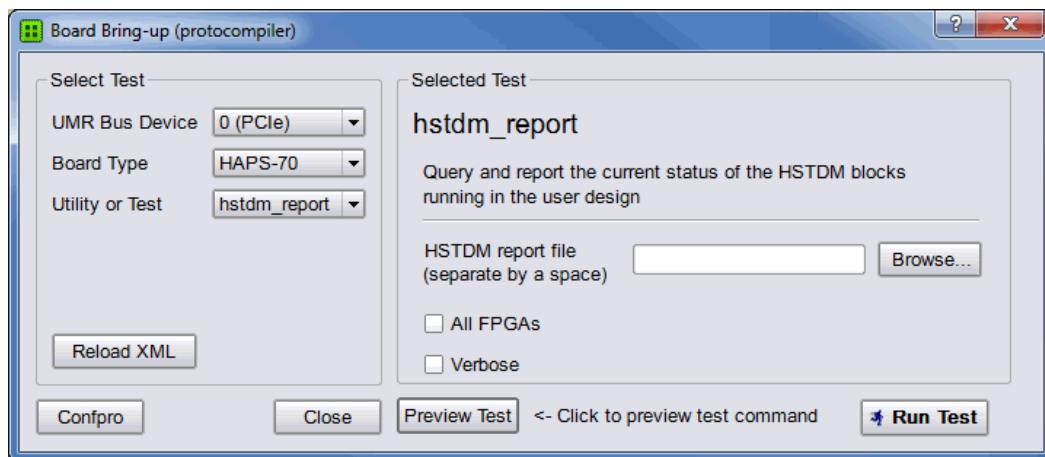
1. Install the following software, which is needed to run the HAPS bring-up utilities:
  - Current version of the HAPS ProtoCompiler Runtime tool set
  - Confpro tool set

For current versions and installation information, see the release notes.

2. Launch the board bring-up utility by running the following command from the bin directory where the debugger is installed:

**protocompiler\_runtime\_new board\_bringup [-shell] [-tcl *tclScript*]**

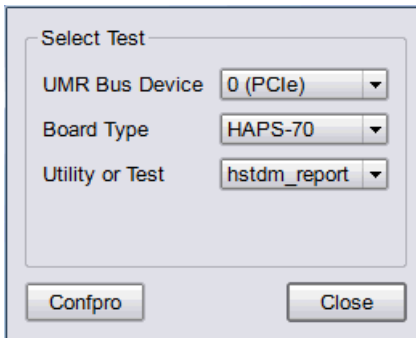
This command launches a special debugger window with the Board Bring-up dialog box displayed where you can enter individual haps utility and board-test commands at the Tcl script prompt to set board parameters and to run individual board tests to verify the board configuration.



The complete haps command syntax is described in the *HAPS ProtoCompiler Debug Environment Reference Manual*. The individual board tests are described in [Board Configuration Tests, on page 64](#).

## Common Parameters for Bring-up Utilities

The common bring-up utility parameters are available from drop-down menus on the left side of the Board Bring-up dialog box.



To configure the bring-up utility:

- The UMRBus Device field selects the type and location of the UMRBus device. Four PCIe and eight USB devices can be selected from the drop-down menu.
- The Board Type field selects the HAPS board type.
- To Confpro button invokes the Confpro GUI. The location of the Confpro installation is specified by selecting Debugger->Configure Confpro from the debugger main menu. Confpro GUI documentation is available by selecting Help->Contents from the top-level menu, and complete documentation is available from the *System Configuration Software Handbook* available on HAPS SupportNet.
- The four GUI-based utility commands are available from the Utility or Test drop-down menu. Selecting a test displays a description of the test and available parameter settings in the right side of the dialog box, and selecting the Run Test button executes the test. See [Utility Commands, on page 68](#) for information on the individual utility commands.

## Board Configuration Tests

The board tests are available only by individual Tcl commands. For individual test details, see the following:

- [clock\\_check Test](#), on page 65
- [con\\_check Test](#), on page 65
- [con\\_speed Test](#), on page 66
- [umr\\_check Test](#), on page 67
- [spi\\_flash\\_memory\\_check Test](#), on page 68
- [GPIO \(LEDs, Buttons\) Check Test](#), on page 68

## Running Board Configuration Tests

This section outlines how to run any of the board configuration tests.

Before you run any of the tests, make sure these prerequisites are met:

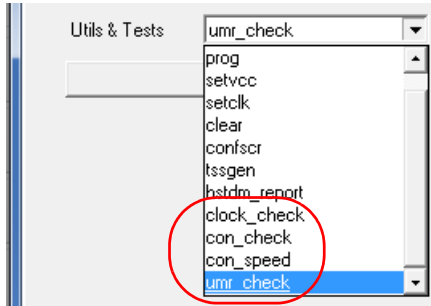
- Confpro has been installed and configured
- a UMRBus PCI/USB device is selected
- the appropriate board type is specified

The corresponding Tcl command for device and board selection is:

**haps settings {PORT\_NAME *portName* DEV\_ID *device* BUS\_NUM *bus*}**

The command specifies the port, device, and bus settings. For detailed information on the haps Tcl commands, see [haps](#), on page 33 in the *Debugging Environment Reference Manual*.





## clock\_check Test

The `clock_check` test reports the clock frequency of each GCLK output to allow all of the GCLK frequencies to be verified, and also reports board status in the `hapstest.log` file created in the current working directory. To run the `clock_check` test, use the following Tcl command syntax:

**haps run clock\_check**

The `clock_check` test configures all FPGAs in the system with a dedicated bit-stream (the clock-check design), which *measures* the frequencies of all global clocks entering each FPGA. When running `clock_check`, it is possible that an existing FPGA configuration can be cleared (existing clock configurations are always maintained).

## con\_check Test

The `con_check` test verifies that the cabling between HapsTrak<sup>®</sup> connectors is consistent with the defined connectivity file. The test is intended to be run on an already configured system and assumes that all I/O voltages for the board regions are defined (see [haps setvcc Command](#), on page 70). To run the `con_check` test, use the following Tcl command syntax:

**haps run con\_check [connectivityFileScript] [logFileName]**

If the `connectivityFileScript` does not reside in the current directory, navigate to the corresponding directory. Also, if you use a non-default log file, you must explicitly specify the connectivity file even if you intend to use the default; for example:

```
haps run con_check {} ./logfiles/hapstest2.log
```

## con\_speed Test

The `con_speed` test verifies the connectivity between HapsTrak connectors as well as the speed at which data transfers can run (duplicating HSTDM transfer rates). To run the `speed` test using the Tcl syntax:

```
haps run con_speed speed fast|sweep [connectivityFileScript] [logFileName]
```

To run the test:

- Specify a *speed* setting to set the HSTDM verification rate (in Mbps). The acceptable values are 840, 960, 1080, or ALL (specifying ALL scans a range of speeds to determine the highest rate possible).
- Specify *fast* or *sweep* for the run mode. The *fast* mode (the default) verifies each channel at the specified speed; the *sweep* mode tests (sweeps) all channels at each speed and can require several hours to complete.

- Specify the Tcl script (*connectivityFileScript*) that describes the connections between HapsTrak connectors in your current system set-up. The default script in the current working directory is named `connectivity.tcl`. Unless you are using this script in the current working directory, include the path to the location of the target script.
- Unless you are using the default `hapstest.log` file in the current working directory, you must explicitly specify the connectivity file even if you intend to use the default connectivity file; for example:

```
haps run con_speed 960 fast {} ./logfiles/hapstest2.log
```

The resultant log file reports:

- FPGA connectivity and banks
- Voltage regions
- FPGA configuration including clock and reset
- Transfer speed and results
- Training results for the banks and minimum eye width
- Hardware/firmware version and test results including any failed channels

The HapsTrak® 3 connectors at board locations 16, 17, 19, and 20 on a HAPS-70 series system are special connectors in that each of these connectors has only a single clock pair (normal connectors have four clock pairs). When testing the connectivity between any of these special connectors and a standard connector, the special connector is automatically reported last in the `connectPorts` statement (for example, `connectPorts fb1.B13 fb1.A16`).

## umr\_check Test

The `umr_check` test verifies the basic functionality of the UMRBus. The `umr_check` test:

- Downloads bin files through the UMRBus
- Reads status registers
- Writes control registers

To run the `umr_check` test using the Tcl syntax:

```
haps run umr_check fpgaID
```

Specify the FPGA to be tested (an ALL selection is available to test the UMRBus on all FPGAs). The default is 1 which is the first FPGA device on the first board. The test passes if the basic UMRBus operation succeeds.

## **spi\_flash\_memory\_check Test**

The `spi_flash_memory_check` test performs functional validation of the onboard SPI flash memory of HAPS-80D. In particular, this test performs write and read transactions to the memory and data comparison check based on which the test status is reported as PASS or FAIL.

Command to execute the test:

```
haps run spi_flash_memory_check <transfer_size> <pattern_type>  
<logfile>
```

## **GPIO (LEDs, Buttons) Check Test**

The `leds_buttons_dip_check` test helps user to verify the functionality of LEDs, buttons, and DIP switches connected to the front panel of the HAPS-80D system.

Command to execute the test

```
haps run leds_buttons_dip_check [<LogFile>]
```

<LogFile> optional parameter to specify the name of the log file.

Default: `hapstest.log` (in the current working directory).

# **Utility Commands**

Utility commands, like the board test commands, are entered from the Tcl shell command prompt. You can run any of the utility commands listed below:

- [haps boardstatus Command](#), on page 69
- [haps prog Command](#), on page 70
- [haps setvcc Command](#), on page 70
- [haps setclk Command](#), on page 71
- [haps clear Command](#), on page 71
- [haps confscr Command](#), on page 72
- [haps tssgen Command](#), on page 72
- [haps hstmd\\_report Command](#), on page 72

In addition, the following four commands can be run directly from the Board Bring-up dialog box in the GUI.

- [haps setvcc Command](#), on page 70
- [haps setclk Command](#), on page 71
- [haps tssgen Command](#), on page 72
- [haps hstmd\\_report Command](#), on page 72
- [HAPS SPI Flash Memory Check Command](#), on page 76
- [HAPS LEDs, Buttons, DIP Check Command](#), on page 76

The above commands are individually selected from the Utility or Test drop-down menu. Selecting a command displays a description of the selected command in the Selected Test area on the right side of the dialog box. After filling in any required fields, clicking the Run button executes the command.

## haps boardstatus Command

Displays the board status to the console window. Status includes clock and voltage settings, reset condition, daughter board connections, firmware version, and board serial number. The board is identified by the Board ID/No. field drop-down menu. The equivalent Tcl command syntax is:

**haps boardstatus [boardID]**

## haps prog Command

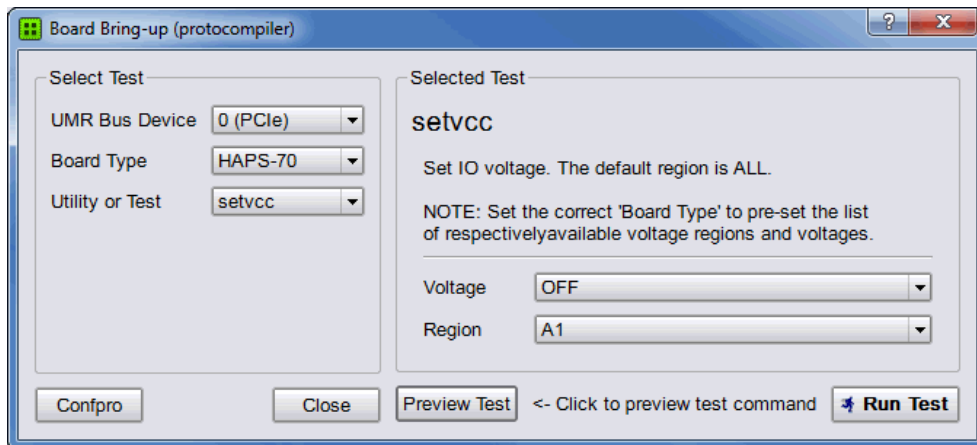
Programs the FPGA specified with the contents of the selected bin file. The *devID* selection ranges from 1 to 32.

The equivalent Tcl command syntax is:

```
haps prog binFile devID
```

## haps setvcc Command

Sets the I/O voltage for the board regions. The voltage value and region are selected from the corresponding drop-down menus and differ with the board/system selected. After selection, click the Preview Test button to display the complete command to be executed.



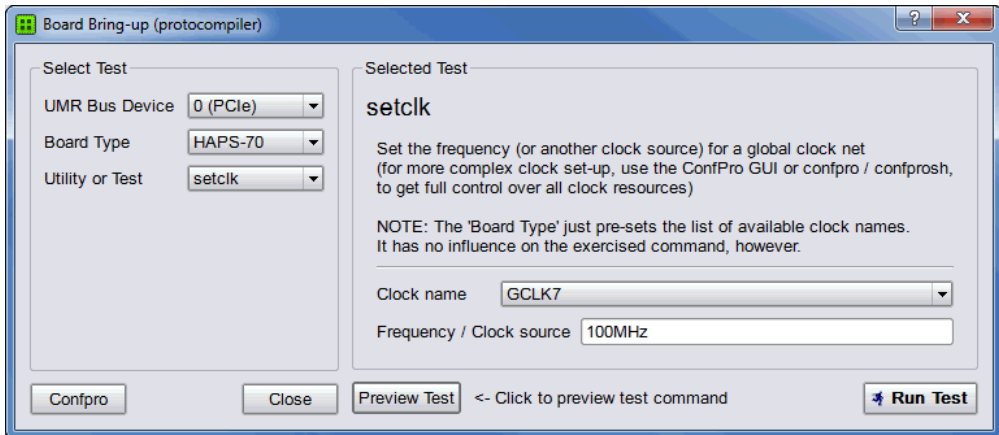
The equivalent Tcl command syntax is:

**haps setvcc** *voltage* [*region*]

In the syntax, *voltage* sets the I/O voltage for the board regions. The *voltage* and *region* values differ with the board/system selected. If *region* is omitted, all regions are set to *voltage*.

## haps setclk Command

Sets the frequency for the global input clock identified by the Clock name field entry with the frequency specified in the Frequency field. The frequency value is in kHz unless specified otherwise.



The equivalent Tcl command syntax is:

**haps setclk** *clockName* *frequency*

In the syntax, the Tcl version of the command also accepts a clock source parameter such as left or right. For additional information on clock sources, see the *System Configuration Software Handbook* on the HAPS SupportNet.

## haps clear Command

Clears the entire board/system configuration including the FPGA configuration, voltage, and clock settings. The equivalent Tcl command syntax is:

**haps clear**

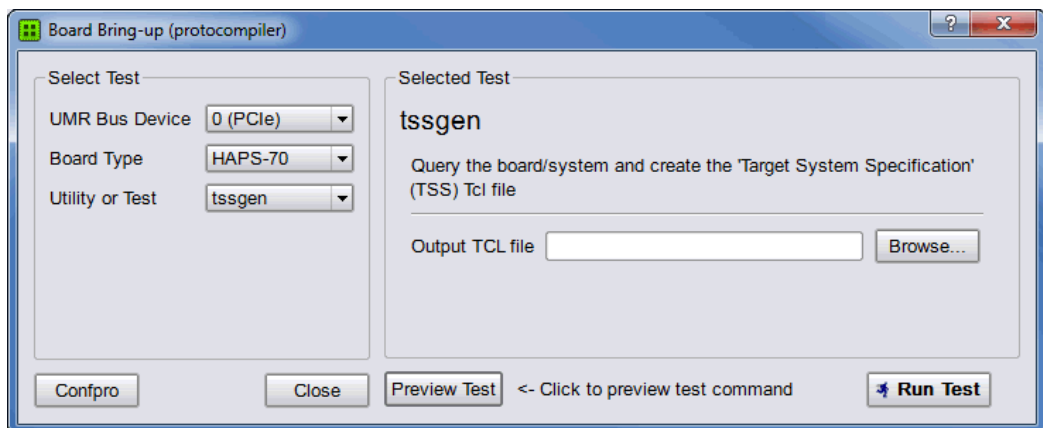
## haps confscr Command

Runs confprosh Tcl scripts. For example, the confprosh command can be used to source a HAPS clock and voltage-region configuration script; the user could then run clock checks to verify the on-board clock configuration. The name of the script is entered as a command argument. The Tcl command syntax is:

```
haps confscr scriptFileName
```

## haps tssgen Command

Queries the HAPS system and generates a file with the hardware description that can be used to check the hardware configuration. The output Tcl file is written to the file name specified in the Output TCL file field. Clicking the Save button prompts for an alternate location to save the Tcl file (by default, the Tcl file is saved to the current working directory).



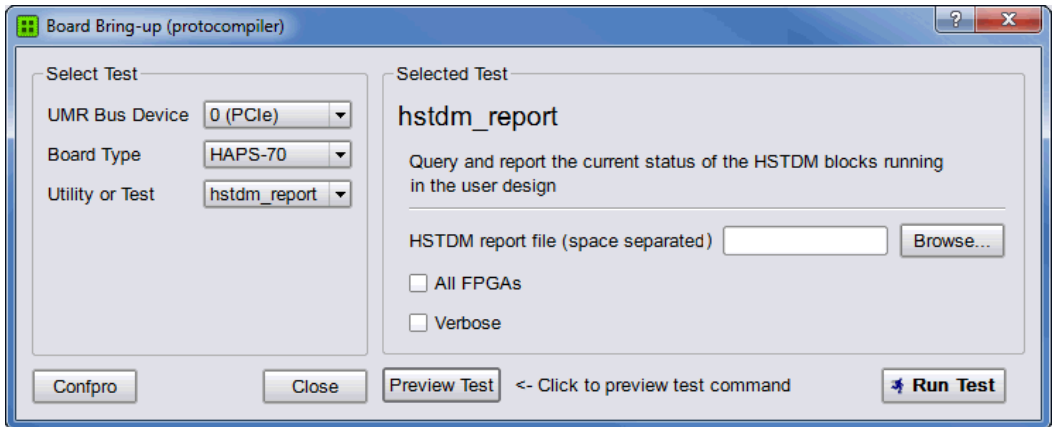
The equivalent Tcl command syntax is:

```
haps tssgen tclFile
```

## haps hstdm\_report Command

The hstdm\_report utility queries and reports the current training status of the HSTDm blocks running in the user design.





In the above dialog box:

- HSTDM report file – a Tcl list of training report files to query.
- All FPGAs – query the training report files from all FPGAs.
- Verbose – include additional content in the training report.

The equivalent Tcl command syntax is:

```
haps hstmd_report [{hstmdReportList}] ALLFPGAS [poll [{interval}]]  
[verbose] [device {index}]
```

In the Tcl command:

- *hstmdReportList* – a Tcl list of training report files to query.
- **ALLFPGAS** – query the training report files from all FPGAs.
- **poll** [*interval*] – poll the FPGAs for the specified interval in seconds; the poll argument is not supported from the GUI.
- **device** *index* – list training status for the FPGAs on the indexed board device.

## Examples

To report training status:

```
haps hstmd_report fb74_uD_hstmdreport.txt
```

To report training status with extra content in the report:

```
haps hstddm_report fb74_uD_hstddmreport.txt verbose
```

To report training status for all FPGAs:

```
haps hstddm_report ALLFPGAS
```

To report training status for FPGAs on device 8 only

```
haps hstddm_report ALLFPGAS device 8
```

To poll HSTDM error status for all FPGAs for an interval to 20 seconds:

```
haps hstddm_report poll 20
```

## Single Channel Table Legend

The following table defines the single FPGA channel table legends.

Table Legend	Description
Connector	Connector name of the channel on the HAPS system.
IO_Bank	I/O bank of the channel on the FPGA.
Trace	Trace name of the HSTDM channel.
Ratio	HSTDM user multiplex ratio.
Training	HSTDM training status.
Lost_sync	Indicates any error with the source sync clock.
ERD	HSTDM_ERD found errors in the channel.
Eye_w	Eye width of the channel.
Start	Start position of an open eye.
End	End position of an open eye.

## Summary Table Legend

The following table defines the summary table legends.

Table Legend	Description
ChipID	The unique ID for each FPGA in the system.
Chip_Name	FPGA name.
Training	Indicates an error during HSTDM training.
User_reset	Indicates an error when releasing user reset.
Lost_sync	Indicates an error with the source sync clock.
ERD	Indicates an HSTDM_ERD detected error.
CAPIM_Address	HSTDM CAPIM location on the UMRBus.

## HSTDM Training Status

The following table defines the possible training status results.

Status Reported	Explanation/Indication
Good	Training successful
Not_trained	Not yet trained
No_center	Uncommon error message. Found the start and end of stable data, but data at the middle of the eye was incorrect or no longer stable.
No_unstable_data	Indicates the absence of a signal coming from the trainer usually as a result of a pin assignment error or an unconnected clock on the trainer.
No_stable_data	Indicates possible noise or the use of an incompatible I/O standard.
Eye_too_small	The start and end of the stable data are too close together which makes it unlikely that data can be sent reliably. Try lowering the HSTDM bit rate or use CON cables for connections rather than CON boards.
Unknown_status	Unspecified status.
Empty_status	No data received on the channel. Usually indicates a lost cable connection.

## HAPS SPI Flash Memory Check Command

From the board bring-up window, enter the following to perform the SPI flash memory check test.

The screenshot shows the 'Select Test' and 'Selected Test' panels of the HAPS Prototyping Debugger. In the 'Select Test' panel, 'UMRBus Device' is set to '10 (USB)', 'Board Type' is 'HAPS-80D', and 'Utility or Test' is 'spi\_flash\_memc'. The 'Run in background' checkbox is checked. In the 'Selected Test' panel, the test is 'spi\_flash\_memory\_check', which performs functional validation of the onboard SPI flash memory of HAPS-80D. The 'Size of Transfer(kb)' is set to '2', and the 'Pattern Type' is 'fixed'. The 'Test log file' is 'spi\_flash\_memory\_check.log'. There are buttons for 'Preview Test', 'Run Test', 'Configure', and 'Configure Configro...'.

1. Select the test from the Utility or Test drop-down list.
2. Enter the Size of Transfer and select the Pattern Type (fixed or random).
3. Click the Run Test button.

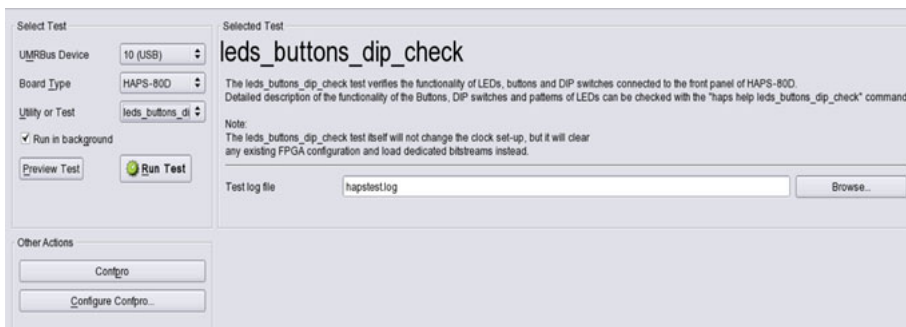
The default test log file will be `spi_flash_memory_check.log`.

Equivalent command:

```
haps run spi_flash_memory_check <transfer_size> <pattern_type>
<logfile>
```

## HAPS LEDs, Buttons, DIP Check Command

From the Board Bring-up window, enter the following to perform the LEDs, buttons, DIP check test.



1. Select the Utility or Test from the drop-down list.
2. Click the Run Test button.

The default test log file will be `hapstest.log`.

Equivalent Command:

```
haps run leds_buttons_dip_check [<LogFile>]
```



## CHAPTER 5

# IICE Hardware Description

---

The instrumentor adds instrumentation logic to your HDL design that allows you to understand and debug design operation. There are some aspects of the instrumentation logic that are important to understand in order to use the debug environment tool set in the most effective way. In this chapter, the overall instrumentation logic is described briefly followed by descriptions of some of the more important features. A simplified functional breakdown of the instrumentation logic consists of:

- [Breakpoint and Watchpoint Blocks](#)
- [Sampling Block](#)
- [Complex Counter](#)
- [State Machine Triggering](#)

# Breakpoint and Watchpoint Blocks

The following topics are described in this section:

- [Breakpoints](#)
- [Watchpoints](#), on page 81
- [Multiple Activated Breakpoints and Watchpoints](#), on page 81

## Breakpoints

Breakpoints are a way to easily create a trigger that is determined by the flow of control in the design.

In both Verilog and VHDL, the flow of control in a design is primarily determined by `if`, `else`, and `case` statements. The control state of these statements is determined by their controlling HDL conditional expressions. Breakpoints provide a simple way to trigger when the conditional expressions of one or more `if`, `else`, or `case` statements have particular values.

The example below shows a VHDL code fragment and its associated breakpoints.

```
99 process(op_code, cc, result) begin
100   case op_code is
101     when "0100" =>
102       result <= part_res;
103       if cc = '1' then
104         c_flag <= carry;
105         if result = zero then
106           z_flag <= '1';
107         else
108           z_flag <= '0';
109         end if;
110       end if;
```



The four breakpoints correspond to these control flow equations:

- Breakpoint at line number 102:

```
(op_code = "0100")
```

- Breakpoint at line number 104:

```
(op_code = "0100") and (cc = '1')
```

- Breakpoint at line number 106:

```
(op_code = "0100") and (cc = '1') and (result = zero)
```

- Breakpoint at line number 108:

```
(op_code = "0100") and (cc = '1') and (result != zero)
```

## Watchpoints

A watchpoint creates a trigger that is determined by the state of a signal in the design. The watchpoint can trigger either on the value of a signal or on a transition of a signal from one value to another.

## Multiple Activated Breakpoints and Watchpoints

How breakpoints and watchpoints operate individually is described in the *HAPS® ProtoCompiler Instrumentor User Guide*. Activated breakpoints and watchpoints also interact with each other in a very specific way.

### Multiple Activated Breakpoints

Each breakpoint is implemented as logic that watches for a particular event in the design. When an instrumented design has more than one activated breakpoint, the breakpoint events are ORed together. This effectively allows the breakpoints to operate independently – only one activated breakpoint must trigger in order to cause the sampling buffer to acquire its sample.

## Multiple Activated Watchpoints

Each watchpoint is implemented as logic that watches for a specific event consisting of a bit pattern or transition on a specific set of signals. When an instrumented design has more than one activated watchpoint, the watchpoint events are ANDed together. This effectively causes the watchpoints to be dependent on each other – all activated watchpoint events must occur concurrently to cause the sampling buffer to acquire its sample.

For example, if watchpoint 1 implements (count == 23) and watchpoint 2 implements (ack == '1'), then activating these watchpoints together effectively creates a new watchpoint: (count == 23) && (ack == '1').

## Combining Activated Breakpoints and Activated Watchpoints

When an instrumented design has one or more activated breakpoints and one or more activated watchpoints, the result of the OR of the breakpoint events and the result of the AND of the watchpoint events is ANDed together. The result of this AND operation is called the Master Trigger Signal. This ANDing effectively causes the breakpoints and watchpoints to be dependent on each other – one activated breakpoint and all activated watchpoint events must occur concurrently to cause the sampling buffer to acquire its sample.

As a result, a Master Trigger Signal event can be constructed that operates like a conditional breakpoint. For example, activating a breakpoint and the two watchpoints from the previous example produces a conditional breakpoint: (breakpoint event) && (count== 23) && (ack == '1').

# Sampling Block

The sampling block is basically a large memory used to store all the sampled signals. During an active debugging session, the sampled signals are continually being stored in the sample block. When the sample block receives an event from the Master Trigger Signal event logic or the complex counter logic, the sampling block stops writing new data to the buffer and holds its contents. Eventually, the contents of the sample block are uploaded to the debugger for display and formatting.

Whenever possible, the sample block should use the built-in RAM blocks that are available in most programmable chips. Otherwise, implementing the sample buffer using individual storage elements will consume large amounts of the logic capacity of the chip. If you have no choice but to use individual storage elements, analyze how much logic you have available on your chip and adjust how many signals you sample and the depth of the sample buffer.

## Complex Counter

The complex counter connects the output of the breakpoint and watchpoint event logic to the sampling block and allows the user to implement complex triggering behavior.

### Creating a Complex Counter

The counter is created, configured, and inserted into the HDL design during instrumentation using the instrumentor IICE Controller tab of the IICE Configuration dialog box or using the instrumentor `iice controller` command.

During configuration, the size of the counter is specified. For example, a 16-bit counter is the default. This default value produces a counter that ranges from 0 to 65535.

Setting the counter size to zero during instrumentation configuration disables counter insertion.

### Debugging with the Complex Counter

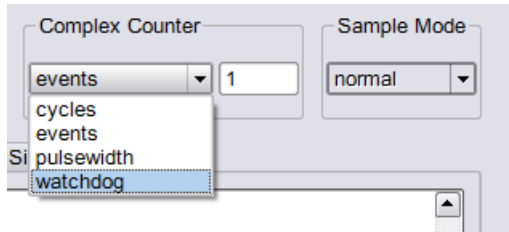
The complex counter is used to produce complex triggering behavior. During the debugging of the design, the complex counter is set to zero on invocation of the debugger run command. Then, it counts events from the Master Trigger Signal event logic in a specific way depending on the counter mode.

Finally, the counter sends a trigger event to the sample block when a termination condition occurs. The form of the termination condition depends on the mode of operation of the counter and on the target value of the counter:

- The counter target value can be set to any value in the counter's range.

- The counter has four modes: events, cycles, watchdog, and pulsewidth.

The counter target value and the counter mode can be set directly from the main menu.



The following table provides a general description of the trigger behavior for the various complex counter modes. Each mode is described in more detail in individual subsections, and examples are included showing how the modes are used. In both the table and subsection descriptions, the counter target value setting is represented by the symbol  $n$ .

Counter mode	Target value = 0	Target value $n > 0$
events	illegal	stop sampling on the $n$ th trigger event.
cycles	stop sampling on 1st trigger event	stop sampling $n$ cycles after the 1st trigger event.
watchdog	illegal	stop sampling if the trigger condition is not met for $n$ consecutive cycles.
pulsewidth	illegal	stop sampling the first time the trigger condition is met for $n$ consecutive cycles.

## events Mode

In the events mode, the number of times the Master Trigger Signal logic produces an event is counted. When the  $n$ th Master Trigger Signal event occurs, the complex counter sends a trigger event to the sample block. For example, this mode could be used to trigger on the 12278th time a collision was detected in a bus arbiter.

### **cycles Mode**

In the cycles mode, the complex counter sends a trigger event to the sample block on the  $n$ th cycle after the first Master Trigger Signal event is received. The clock cycles counted are from the clock defined for sampling. For example, this mode could be used to observe the behavior of a design 2,000,000 cycles after it is reset.

### **watchdog Mode**

In the watchdog mode, the counter sends a trigger event to the sample block if no Master Trigger Signal events have been received for  $n$  cycles. For example, if an event is expected to occur regularly, such as a memory refresh cycle, this mode triggers when the expected event fails to occur.

### **pulsewidth Mode**

In the pulsewidth mode, the complex counter sends a trigger event to the sample block if the Master Trigger Signal logic has produced an event during each of the most recent  $n$  consecutive cycles. For example, this mode can be used to detect when a request signal is held high for more than  $n$  cycles thereby detecting when the request has not been serviced within a specified interval.

## **Disabling the Counter**

According to the previous table, the counter can be disabled simply by setting its target value to 1 and its mode to events. Then, the complex counter will pass any received event from the Master Trigger Signal logic on to the sample block with no additional delay.

# State Machine Triggering

This section describes the different methods of triggering available in the debugger. It explains the different choices available during instrumentation and the functionality these choices provide in the debugger as well as discussing the cost effects of the various types of instrumentation.

## Simple or Advanced Triggering

There are two triggering modes available, the simple mode and the advanced mode. The simple mode allows comparing signals to values (including don't cares) and then triggering when the signals match those values. This scheme can be enhanced by using breakpoints to denote branches in control logic. If a breakpoint is enabled, this particular branch must be active at the same time that the signals match their respective values. The overall trigger logic involves signals and breakpoints in the following way:

- **Signals:** All signals must match their respective comparison values in order to trigger.
- **Breakpoints:** All breakpoints are OR connected, meaning that any one enabled breakpoint is enough to trigger.
- **Signals and breakpoints are combined using AND,** such that all signals must match their values AND at least one enabled breakpoint must occur.

The logic that implements breakpoint and signal triggering is referred to as trigger condition in the following text.

In the advanced trigger mode, multiple such trigger conditions are instrumented, and a runtime-programmable state machine is also instrumented to allow you to specify the temporal and logical behavior that combines these trigger conditions into a complex trigger function. For instance, this state machine enables you to trigger on a certain sequence of events like “trigger if pattern A occurs exactly five cycles after pattern B, but only if pattern C does not intervene.”

By default, the instrumentor instruments the design according to the simple trigger mode. See the following for more information on how to select the advanced trigger mode.

## Advanced Triggering Mode

Setting up an instrumented design to enable advanced triggering is extremely easy. There are two iice controller command options available in the instrumentor that control the extent and cost of the instrumentation:

- **-triggerconditions** *integer* – The *integer* argument to this option defines how many trigger conditions are created. The range is from 1 to 16. All these trigger conditions are identical in terms of signals and breakpoints connected to them, but they can be programmed separately in the debugger.
- **-triggerstates** *integer* – The *integer* argument to this option defines how many states the trigger state machine will have. The range is 2 to 10.

Similar to the simple-triggering mode, a counter can be instrumented to augment the functionality of the state machine. To instrument a counter, enter an iice controller -counterwidth option with an argument greater than 0 in the instrumentor console window.

Please refer to the following text to determine cost and consequences of these settings in the instrumentor.

### Cost Estimation

The most critical setting with respect to cost is the number of trigger conditions, as each trigger condition results in an additional address bit on the RAM, and thus doubles the size of the RAM table with each bit. Next in importance is the counter width as this factor contributes directly to RAM table width and is especially significant in the context of FPGA RAM primitives that allow a trade-off of width for depth.

The actual instrumentation, however, is not limited to the use of a single block RAM (for example, it may be advantageous in a particular situation to trade away states for additional trigger conditions or for additional counter width). Any configuration can be automatically implemented, as long as it fits on the device with the remainder of the design.

Although RAM parameters are automatically determined by the instrumentor, this information should be monitored to make sure that no resources are wasted.

## Using State Machine Triggering in the Debugger

Perform the following steps in the debugger console window to setup a trigger in advanced triggering mode. These steps can be done in any order.

- setup the values for the trigger conditions using the debugger watch and stop commands.
- setup the trigger state machine behavior using the debugger statemachine command.

The watch command takes an additional parameter, `-condition`, specifying the trigger conditions that the given condition is intended for. This argument is available in simple mode as well, but as there is only one trigger condition in this case, the argument is redundant.

- **watch enable -condition** (*triggerCondition*|all) *signalName* *value1* [*value2* ...]
- **watch disable -condition** (*triggerCondition*|all) *signalName*
- **watch info** [-raw] *signalName*

The parameter *triggerCondition* is a list value conforming to the Tcl language. Examples are: 1, "1 2 3", {2 3}, or [list 1 2 3], quotes, braces, and brackets included, respectively. Alternatively, the keyword `all` can be specified to apply the setting to all trigger conditions.

The debugger watch info command reports status information about the signal. This information is returned in machine-processible form if the optional parameter `-raw` is specified.

Similarly for the debugger stop command:

- **stop enable -condition** (*triggerCondition*|all) *breakpoint*
- **stop disable -condition** (*triggerCondition*|all) *breakpoint*
- **stop info** [-raw] *breakpoint*

The semantics of the parameters are identical to the above descriptions.

## The statemachine Command

During instrumentation, the number of states was previously defined using the `-triggerstates` option of the `instrumentor iice controller` command. Now, at debug time, you can define what happens in each state and transition depending on the pattern matches computed by the trigger conditions. The



debugger **statemachine** command is used to configure the trigger state machine with the desired behavior. This is very similar to the “advanced” trigger mode offered by many logic analyzers. As it is very easy to introduce errors in the process of specifying the state machine, special caution is appropriate. Also, a state-machine editor is available in the debugger user interface to facilitate state-machine development and understanding (see [State-Machine Editor, on page 95](#)). It is also important to note that the initial state for each run is always state 0 and that not all of the available states need to be defined.

The syntax forms of the debugger **statemachine** command are:

- **statemachine addtrans** **-from** *state* [**-to** *state*] [**-cond** "*equation|triggerInID*"] [**-cntval** *integer*] [**-cnten**] [**-trigger**]
- **statemachine clear** (**-all**|*state* [*state* ...])
- **statemachine info** [**-raw**] (**-all**|*state* [*state* ...])

## Subcommand **statemachine addtrans**

The debugger **addtrans** subcommand defines the transitions between the states. The options are as follows:

- **-from** *state* – specifies the state this transition is exiting from.
- **-to** *state* – specifies the state this transition goes to. If this is not given, it defaults to the state given in the **-from** option.
- **-cond** "*equation|triggerInID*" – specifies the condition or external trigger input under which the transition is to be taken. The default is “true” (i.e., the transition is taken regardless of input data; see below for more details).
- **-cntval** *integer* – specifies that if this transition is taken, the counter is loaded with the given value. Only valid when a counter is instrumented.
- **-cnten** – when this flag is given, the counter is decremented by 1 during this transition. Only valid when a counter is instrumented.
- **-trigger** – when this flag is given, a trigger occurs during this transition.

The order in which the transitions are added is important. In each state, the first transition condition that matches the current data is taken and any subsequent transitions in the list that match the current data are ignored.

## Conditions

The conditions are specified using Boolean expressions comprised of variables and operators. The available variables are:

- **c0, . . . cn**, where *n* is the number of trigger conditions instrumented. These variables represent the output bit of the respective trigger condition.
- **tiTriggerInID** – the ID (0 thru 7) of an external trigger input.
- **cntnull** – true whenever the counter is equal to 0 (only available when a counter is instrumented).
- **iiceID** – variable used with cross triggering to define the source IICE units to be included in the equation for the destination IICE trigger.

Operators are:

- Negation: `not`, `!`, `~`
- AND operators: `and`, `&&`, `&`
- OR operators: `or`, `||`, `|`
- XOR operators: `xor`, `^`
- NOR operators: `nor`, `~|`
- NAND operators: `nand`, `~&`
- XNOR operators: `xnor`, `~^`
- Equivalence operators: `==`, `=`
- Constants: `0`, `false`, `OFF`, `1`, `true`, `ON`

Parentheses ‘(’, ‘)’ are recommended whenever the operator precedence is in question. Use the debugger `statemachine info` command to verify the conditions specified.

For example, valid expression examples are:

```
"c0 and c1"
"! (c1 or c2) and c3"
"c0 or ti4" (condition c0 or external trigger ID ti4)
```

## Other Subcommands

The debugger `statemachine clear` command deletes all transitions from the states given in the argument, or from all states if the argument `-all` is specified.

The debugger `statemachine info` command prints the current state machine settings for the states given in the argument, or for the entire state machine, if the option `-all` is specified. If the option `-raw` is given, the information is returned in a machine-processible form.

## State Machine Examples

To implement a trigger behavior that triggers when the pattern on condition 1 or condition 2 (c1 or c2) becomes true for the 10th time (a setting identical to counter mode events in the simple mode triggering), the following state machine can be used:

```
statemachine addtrans -from 0 -to 1 -cntval 9
statemachine addtrans -from 1 -cond "(c1 | c2) & cntnull" -trigger
statemachine addtrans -from 1 -cond "c1 or c2" -cnten
```

A trigger condition requiring pattern c2 to occur 10 times after pattern c1 has occurred, without pattern c3 occurring in between (commonly available in logic analyzers as “Pattern 1 followed by Pattern 2 before Pattern 3”) can be achieved with the following state machine:

```
statemachine addtrans -from 0 -to 1 -cond c1 -cntval 9
statemachine addtrans -from 1 -cond "c2 & cntnull" -trigger
statemachine addtrans -from 1 -to 0 -cond c3
statemachine addtrans -from 1 -cond "c2" -cnten
```

These behaviors can be cascaded by moving on to the next behavior instead of triggering in the transition that has `-trigger` specified, as long as there are trigger conditions and states available.

## Convenience Functions

There are a number of convenience functions to set up complex triggers available in the file *InstallDir/share/contrib/syn\_trigger\_utils.tcl* which is loaded into the debugger at startup:

- **st\_events** *condition integer* – Sets up the state machine to mimic counter mode events of the simple triggering mode as described above. The argument *condition* is a boolean equation setting up the condition, and *integer* is the counter value.
- **st\_watchdog** *condition integer* – Same as **st\_events** for watchdog mode.
- **st\_cycles** *condition integer* – Same as above for cycles mode.
- **st\_pulsewidth** *condition integer* – Same as above for pulsewidth mode.
- **st\_B\_after\_A** *conditionA conditionB [integer:=1]* – Sets up a trigger mode to trigger if *conditionB* becomes true anytime after *conditionA* became true. The optional *integer* argument defaults to 1 and denotes how many times *conditionB* must become true in order to trigger.
- **st\_B\_after\_A\_before\_C** *conditionA conditionB conditionC [integer:=1]* – Sets up a trigger mode to trigger if *conditionB* becomes true after *conditionA* becomes true, but without an intervening *conditionC* becoming true (same as the second example above). The optional *integer* argument defaults to 1 and denotes how many times *conditionB* must become true without seeing *conditionC* in order to trigger.
- **st\_snapshot\_fill** *condition [integer]* – Uses qualified sampling to sample data until sample buffer is full. The argument *condition* is a boolean equation defining the trigger condition, and *integer* is the number of samples to take with each occurrence of the trigger (default 1).
- **st\_snapshot\_intr** *condition [integer]* – Uses qualified sampling to sample data until manually interrupted by an debugger stop command. The argument *condition* is a boolean equation defining the trigger condition and *integer* is the number of samples to take with each occurrence of the trigger (default 1).

Please refer to the file *syn\_trigger\_utils.tcl* mentioned above for implementing these trigger modes using the debugger *statemachine* command. Users can add their own convenience functions by following the examples in this file.

## Cross Triggering with State Machines

Cross triggering allows a specific IICE unit to be triggered by one or more IICE units in combination with its own internal trigger conditions. The IICE being triggered is referred to as the “destination” IICE; the other IICE units are referred to as the “source” IICE units.

Multiple IICE designs allow triggering and sampling of signals from different clock domains. With an asynchronous design, a separate IICE unit can be assigned to each clock domain, triggers can be set on signals within each IICE unit, and then the IICE units scheduled to trigger each other on a user-defined sequence using cross triggering. In this configuration, each IICE unit is independent and can have unique IICE parameter settings including sample depth, sample/trigger options, and sample clock and clock edges.

Cross triggering is supported in all three IICE controller configurations (simple, complex counter, and state-machine triggering) and all three configurations make use of state machines.

Cross triggering is enabled in the instrumentor (cross triggering can be selectively disabled in the debugger). To enable a destination IICE unit to accept a trigger from a source IICE unit, enter the following command in the instrumentor console window (by default, cross triggering is disabled):

```
iice controller -crosstrigger 1
```

For cross triggering to function correctly, the destination and the contributing source IICE units must be instrumented by selecting breakpoints and watchpoints. Concurrently run these units either by selecting the individual IICE units and clicking the RUN button in the debugger design view or by entering one of the following commands in the debugger console window:

```
run -iice all
```

```
run -iice {iiceID1 iiceID2 ... iiceIDn}
```

When simple- or complex-counter triggering is selected in the destination IICE controller, the following debugger cross-trigger commands are available:

- The following debugger command causes the destination IICE to trigger normally (the triggers from source IICE units are ignored).

```
iice controller -crosstriggermode DISABLED
```

- The following debugger command causes the destination IICE to trigger when any source IICE triggers or on its own internal trigger.

```
iice controller -crosstriggermode ANY
```

- The following debugger command causes the destination IICE to trigger when all source IICE units and the destination IICE unit have triggered in any order.

```
iice controller -crosstriggermode ALL
```

- The following debugger commands cause the destination IICE to trigger after the source IICE unit triggers coincident with the next destination IICE internal trigger.

```
iice controller -crosstriggermode after -crosstriggeriice iiceID
iice controller -crosstriggermode after -crosstriggeriice all
```

The first debugger command uses a single source IICE unit (*iiceID*), and the second debugger command requires all source IICE units to trigger.

When state-machine triggering is selected, the state machine must be specified with at least three states (three states are required for certain triggering conditions, for example, when the destination IICE is in Cycles mode and you want to configure the destination IICE to trigger after another (source) IICE.

With state-machine triggering, the following debugger `statemachine` command sequences are available in the debugger console window:

- The following debugger command sequence is equivalent to disabling cross triggering. The destination IICE triggers on its own internal trigger condition (c0).

```
statemachine clear -all
statemachine addtrans -from 0 -cond "c0" -trigger
```

- In the following debugger command sequence, the destination IICE waits for *iiceID* to trigger and then triggers on its own internal trigger condition (c0). This sequence implements the “after *iiceID*” functionality of the simple- and complex-counter triggering modes.

```
statemachine clear -all
statemachine addtrans -from 0 -to 1 -cond "iiceID"
statemachine addtrans -from 1 -to 0 -cond "c0" -trigger
```

- In the following debugger command sequence, the destination IICE triggers when the last running IICE triggers.

```
statemachine clear -all
statemachine addtrans -from 0 -cond "c0 and iiceID and iiceID1
    and iiceID2" -trigger
statemachine addtrans -from 0 -to 1 -cond "c0"
statemachine addtrans -from 1 -to 0 -cond "iiceID and iiceID1
    and iiceID2" -trigger
```

- In the following debugger command sequence, the destination IICE waits for all the other running source IICE units to trigger and then triggers on its own internal trigger condition (c0).

```
statemachine clear -all
statemachine addtrans -from 0 -to 1 -cond "iiceID and iiceID1
    and iiceID2"
statemachine addtrans -from 1 -cond "c0" -trigger"
```

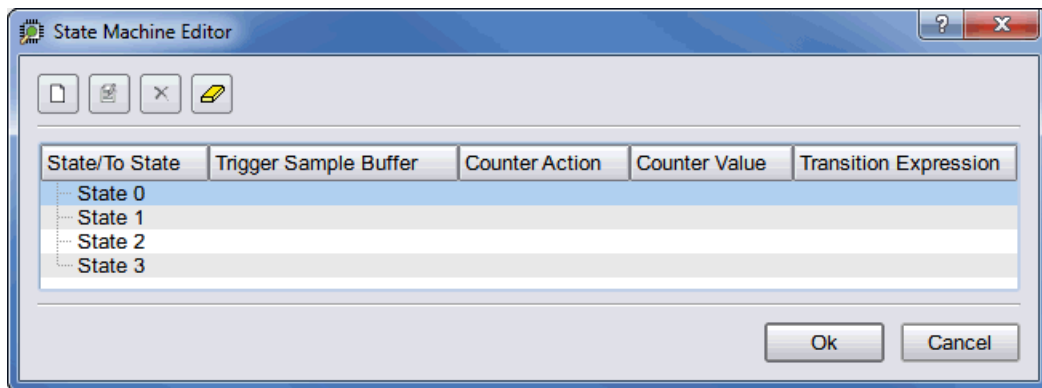
The incorporation of a counter in the state-machine configuration is similar to the use of a counter in non-cross trigger mode for a state machine.

## State-Machine Editor

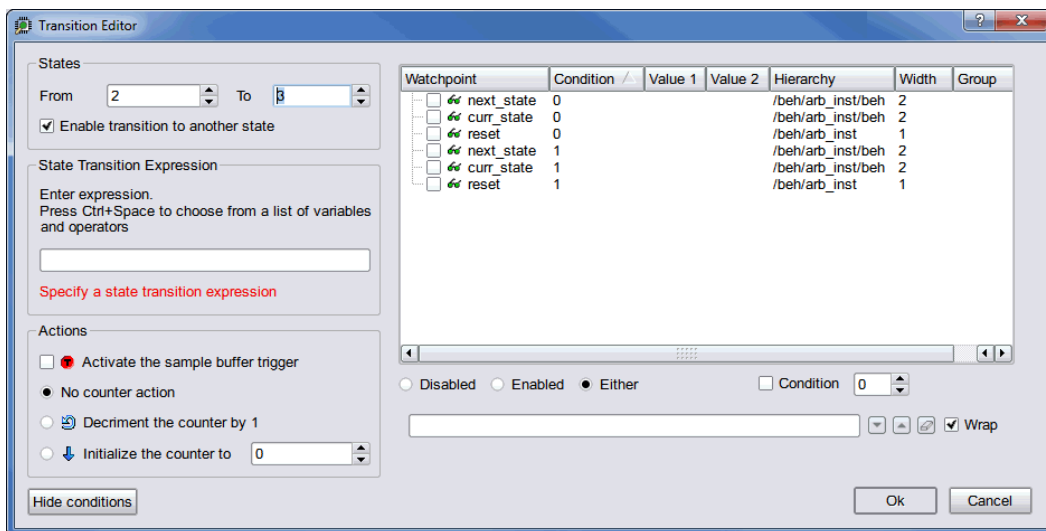
The debugger includes a graphical state-machine editor that is available when state-machine triggering is enabled for the active IICE unit on the IICE Controller tab in the instrumentor.



To bring up the state-machine editor in the debugger, click the Configure state machine icon in the debugger toolbar. Note that the icon will be grayed out if state-machine triggering was not enabled in the instrumentor when the design was instrumented and that an error message will be generated if more than 10 states are defined. Clicking the icon displays the State Machine Editor dialog box for the selected IICE.



Each state is defined in an individual entry field based on the number of states defined in the instrumentor. For each entry, you can add, edit or remove transitions from that state using the transition editing icons in the upper left corner of the dialog box. The Add a new transition and Edit the selected transition icons bring up the Transition Editor dialog box which allows transitions to be defined or redefined. Each transition includes either one or two actions and a condition.



From the dialog box, enter the state-transition expression in the corresponding field. The conditions in the following table are available for defining state transition expressions.



Condition	Description
<b>c0 ... cN</b>	References trigger event in active IICE unit
<b>cntnull</b>	True when counter is equal to 0 (available only when counter is instrumented)
<b>iiceID</b>	References trigger event from a second IICE unit for cross triggering (cross triggering must have been enabled when the design was instrumented)
<b>titriggerInID</b>	References external trigger originating from an IICE module in another FPGA or on-board external logic
<b>Boolean</b>	Boolean operators used to define state-machine events (see <a href="#">Conditions</a> , on page 90)

Note that you can view the corresponding state-machine commands in the Tcl Script window using the `statemachine info -all` command.

```
C:/tools/ident211_078R/bin$ statemachine info -all
State 0:
  if "c0" goto 1 -cntval 4
State 1:
  if "(c1 and cntnull)" goto 0 -trigger
  if "c1" goto 1 -cnten
State 2:
State 3:
C:/tools/ident211_078R/bin$
```

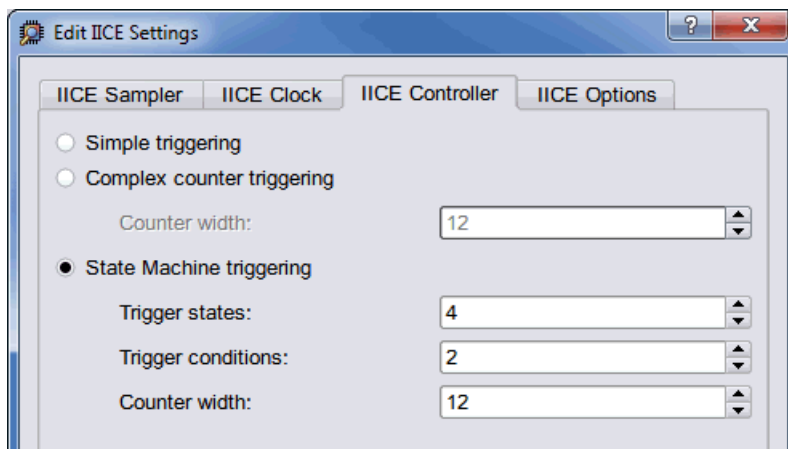
## State-Machine Examples

The state-machine triggering feature allows the creation of counter-based state machines from sequences of trigger conditions to create very effective triggers. You can set up a state-machine trigger during instrumentation and then program the state machine dynamically during debug to create a complex, design-specific trigger.

### Building a Complex State-machine Trigger

When building a complex, state-machine trigger, you specify the number of trigger states, the trigger conditions (which can be set dynamically in the debugger), and the counter width. A common design configuration is to trigger when a specific sequence of events occurs which, in turn, causes data collection to stop and the sample data to be downloaded by the corresponding debugger executable from the FPGA. You can enable state-machine triggering and specify the states through the user interface as outlined in the following steps:

1. Make sure that the following prerequisites are done:
  - In the instrumentor graphical user interface, select Instrumentor->IICE->Edit IICE from the top menu bar or click the Edit IICE icon.
  - From the instrumentor Edit IICE Settings dialog box, select the IICE Controller tab, click the State Machine triggering radio button, and specify the number of trigger states, trigger conditions, and the counter width in the corresponding fields.



2. Build the state machine trigger from the debugger console window. The following debugger command sequence is an example.

```
statemachine addtrans -from 0 -to 1 -cond c0 -cntval 7 -trigger
statemachine addtrans -from 1 -to 0 -cond "cntnull"
statemachine addtrans -from 1 -to 1 -cnten -trigger
```

Note that in the last debugger `statemachine` command, the `-to 1` can be omitted (unnecessary because there is no change in state) and that because the `-from` states are the same in the second and third commands, execution falls through to the third command when the second condition is not true.

3. Once the state-machine trigger is created, use the debugger `statemachine info -all` command to display and review the state-machine transitions.

The debugger state-machine and state-machine transition editors allow:

- Graphical entry of state machines
- Editing of state transitions and trigger events
- Conditions to be combined with each other or with a counter
- Counter mode selection of up, down, or initialized to any value

## State-machine Triggering with Tcl Commands

The IICE can be configured using Tcl commands entered from both the instrumentor and debugger console windows. Some of the example commands are as follows:

- To delete the state transitions from each IICE, use the following debugger command:

```
statemachine clear -iice all
```

- To enable complex counter triggering, use the following instrumentor command:

```
iice controller complex
```

- To set the counter width, use the following instrumentor command:

```
iice controller -counterwidth 8
```

- To configure an IICE for state-machine triggering, use the following instrumentor command sequence:

```
iice controller -iice IICE statemachine
iice controller -iice IICE -counterwidth 4
iice controller -iice IICE -triggerconditions 2
iice controller -iice IICE -triggerstates 2
```

In addition to state-machine triggering, the above instrumentor commands set the number of trigger conditions to 2 and the number of trigger states to 2.

- To enable cross triggering, use the following instrumentor command:

```
iice controller -crosstrigger 1
```

- Similarly, to configure the sample depth, use the following instrumentor command:

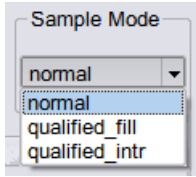
```
iice sampler -depth 2048
```

Note that the only option for buffer type is `internal_memory`.

## Qualified Sampling

During qualified sampling, a single sample of all sampled signals is collected each time the trigger condition is true. When a trigger condition occurs, instead of filling the entire buffer, the IICE collects the single sample and then waits for the next trigger to acquire the next sample. The following example uses qualified sampling to examine the data for a given number of clock cycles. To create a complex trigger event to perform qualified sampling:

1. As a prerequisite in the instrumentor GUI:
  - From the Edit IICE Settings dialog box, select the IICE Controller tab, click the State Machine triggering radio button, and enter a value in the Counter width field to define the width of the sample buffer.
  - Select the IICE Sampler tab and enable the Allow qualified sampling check box.
2. From the debugger GUI, select `qualified_fill` or `qualified_int` from the Sample Mode drop-down menu. For more information, see *-qualified\_sampling* under the `iice` command description in the *Debug Environment Reference Manual*.



3. From the debugger GUI, click on the adjacent Configure state machine icon and define the state-machine trigger event.

When you click Run, the sample buffer begins accumulating data when the trigger event occurs and stops accumulating data after the specified number of samples is reached.

You can also perform qualified sampling using equivalent debugger Tcl commands. The following debugger example command sequence samples the data every *N* cycles beginning with the first trigger event.

```
iice sampler -samplemode qualified_fill
statemachine clear -iice IICE -all
statemachine addtrans -iice IICE -from 0 -to 1
    -cond "true" -cntval 0
statemachine addtrans -iice IICE -from 1 -to 2
    -cond "c0" -cntval 15 -trigger
statemachine addtrans -iice IICE -from 2 -to 2
    -cond "! cntnull" -cnten
statemachine addtrans -iice IICE -from 2 -to 2
    -cond "cntnull" -cntval 15 -trigger
```

## Remote Triggering

Remote triggering allows one debugger executable to send a software trigger event to terminate data collection in the other debugger executables, effectively creating a remote stop button.

You can selectively set the remote trigger to:

- trigger all IICEs in all debugger executables
- trigger all IICEs in a specific debugger executable
- trigger a specific IICE in a specific debugger executable

A common design configuration is to trigger all FPGAs on a single board-level event; when that event occurs, data collection is stopped and the sample data is downloaded by the corresponding debugger executables for all FPGAs.

Remote triggering is a scripting application. The IICE/debugger targets are defined by the debugger `remote_trigger` command (see the command description in the *Debug Environment Reference Manual*).

As an example, the debugger scripting sequence

```
run ; remote_trigger -pid 12
```

waits for the trigger condition in the active IICE and then sends a trigger to all IICE units in the debugger executable identified by process ID 12.

## Importing External Triggers

An import external trigger capability can be used with trigger signals originating from on-board logic external to the FPGA or from an IICE module in a second FPGA. For information on using this feature with state-machine triggering, see the *Importing External Triggers* application note available on SolvNetPlus.

## CHAPTER 6

# Connecting to the Target System

---

This chapter describes methods to connect the debugger to the target hardware system. The programmable device in the target system that contains the design to be debugged is usually placed on a printed circuit board along with a number of other support devices. The difficulty is that the boards differ greatly in the connections between their programmable devices, the other components, and the external connections of the boards.

This chapter outlines how to connect the debugger to most of the common board configurations and addresses the following topics:

- [Basic Communication Connection](#)
- [UMRBus Communications Interface](#)

Note that the debugger can only be connected to a HAPS-70 or HAPS-80 target system through the UMRBus communications protocol.

## Basic Communication Connection

The components that make up the debugging system are:

- The host machine running the debug environment with a loaded design.
- The communication cable connecting the host machine to the programmable device.
- The programmable device loaded with the instrumented version of the design to be debugged.

The following topics are outlined in this section:

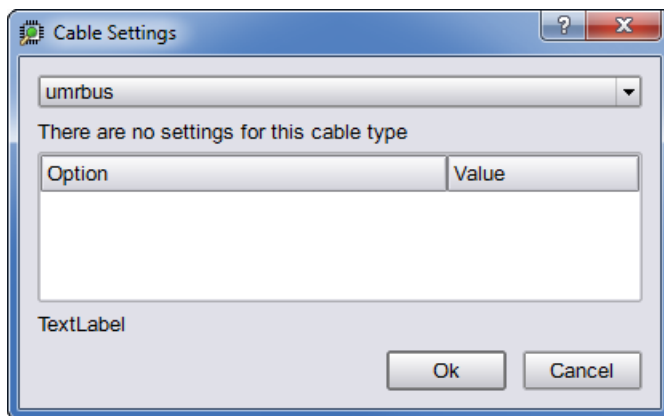
- [Debugger Communications Settings](#), on page 104
- [Debugger Configuration](#), on page 105

### Debugger Communications Settings

Debugger communications settings are defined on the Setup panel and include selecting the cable type and port parameters for the selected cable.

#### Cable Type

The cable type is selected from the Cable Settings dialog box on the Setup panel of the debugger.





If you are using the command interface, set the `com` command's `cabletype` option to `umrbus`. The `umrbus` setting establishes UMRBus communications between the HAPS hardware and the debugger (see [UMRBus Communications Interface, on page 111](#)) and does not require any additional port settings.

## Demo Cable Settings

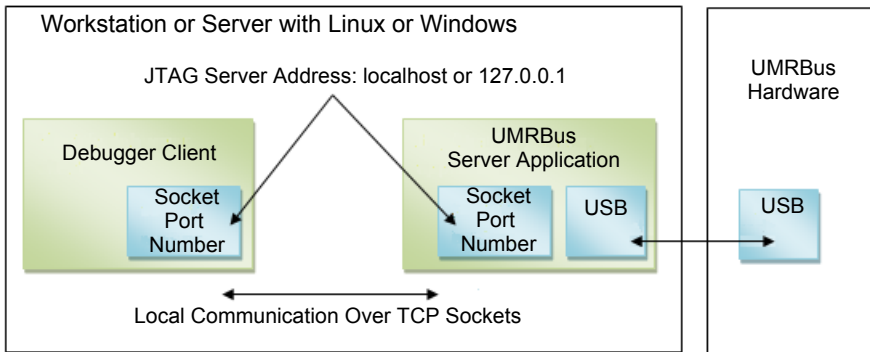
The Port Settings button is disabled when the demo cable is selected.

## Debugger Configuration

All parts of the debugging system must be configured correctly to make a successful connection between the debugger and the instrumented device through the cable. In addition to selecting the cable type and, when required, the port parameters described in [Debugger Communications Settings, on page 104](#), the following additional requirements must be met to ensure proper communications.

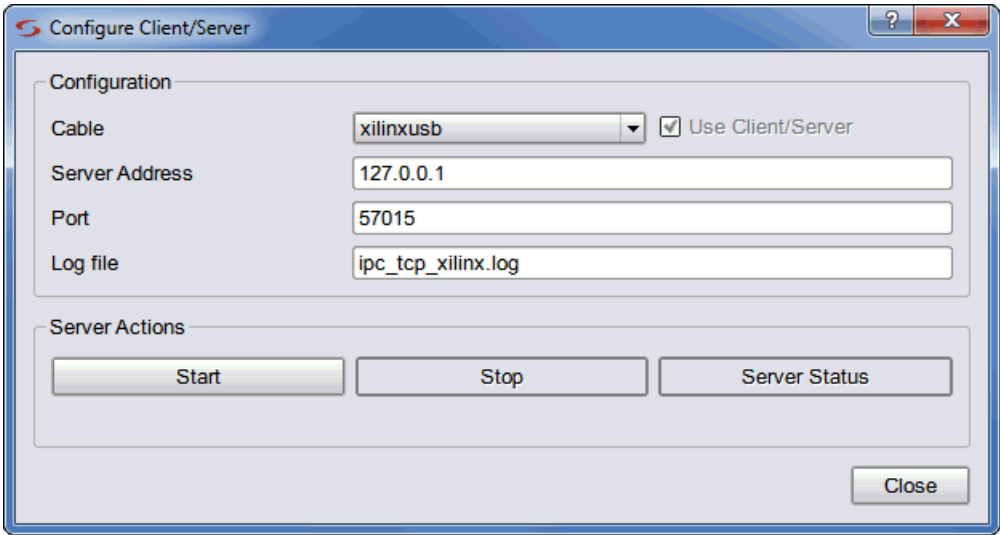
## Local Client-Server Configuration

The following figure shows a typical local server configuration.



The client-server configuration is set from a dialog box available by selecting Client/Server from the Debugger popup menu in the Identify debugger. The default settings are usually correct for most configurations and require changing only when the default server port address is already in use or when

the debugger is being run from a remote machine that is not the same machine connected to the FPGA board/device (see [Remote Client-Server Configuration, on page 107](#)).



The available configure client-server settings in the dialog box are defined in the following table:

Setting	Function
Cable	The type of interface cable (see <a href="#">Cable Type , on page 104</a> ).
Use Client/Server	Check box for enabling client-server communications when the cable type is USB-based UMRBus (limited to HAPS-70 systems).
Server Address	The address of the server. The address localhost (or 127.0.0.1) is used when the debugger is run on the same machine connected to the FPGA device. The server address is set to the name or tcp/ip4 address of the machine connected to the FPGA device/board when the debugger is run from a different machine.
Port	The port number of the server. For all Xilinx cable types, the default port number is 57015; for the UMRBus cable, the default port number is 59015. Change the server port setting when there is a conflict with another tool on the machine.
Log file	The name of the log file.

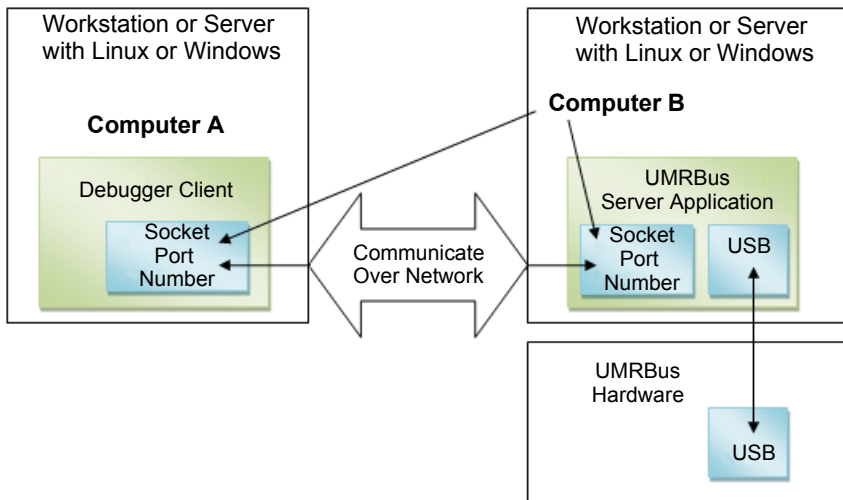
Setting	Function
Start	Server control button for starting the server in stand-alone mode. Activating the button adds a start entry to the log file.
Stop	Server control button for stopping the server in stand-alone mode. Activating the button adds a stop entry to the log file.
Server Status	Adds a start/stop entry to the log file.

To establish a local client-server connection:

1. Start the debugger and open the Configure Client/Server settings dialog box. Select the cable type from the Cable drop-down menu and make sure that the server address is either 127.0.0.1 or localhost.
2. Use the default client-server port (59015) if available. If this port is already in use, list the port status with the `netstat` command and select an unused port. Known ports for system components range from 0 through 1023, and registered ports for software components range from 1024 through 49151. The dynamic and/or private ports range from 49152 through 65535. If possible, use a port address within this range where there is usually ample room.
3. For a local client-server connection, click the OK button when satisfied with the server address and port values. Do not use the Start button as this creates a *standalone* server which must then be manually stopped with the Stop button.
4. Start the debugger client-server session with a `run` or `com check` command after loading the project. The local client-server application ends automatically when the debugger session ends.

## Remote Client-Server Configuration

The following figure shows a client-server configuration for remote debugging.



The debugger uses a client-server architecture to communicate with the device. Client-server architecture lets you work remotely with the debugger using Ethernet as the backbone for the client-server communication.

In the client-server architecture, the machine connected to the target device hardware (Computer B in the diagram) is termed the *server* and any machine on the same network that is used to launch the debugger and connect to the server is termed the *client* (Computer A). You use the Configure Client/Server dialog box described in the previous section to set up both the client and server machines so that you can remotely debug the design. Client-server communication uses the TCP/IP communication protocol over the network.

To establish a server connection for remote debugging:

1. Configure the target device with the design to be debugged.
2. To start the server on the machine connected to the target device, launch the debugger, and then configure the server-side debugger as described below:
  - Load the design project file (debug.prj) of the design to be debugged.
  - In the debugger GUI, select Client/Server from the Debugger popup menu to display the Client/Server dialog box.

- Specify the cable type, server address, port number, and log file name in the corresponding fields. Set the client/server port according to the selected cable type and enable the Use Client/Server check box. Configuring the client-server parameters does not start the server.
  - Start the server in stand-alone mode by clicking the Start button in the dialog box. Once started, close the dialog box by clicking OK to save any changed settings or simply click Cancel to close. With the server running, you can exit the debugger, but you must manually stop the server (click the Stop button) after your session ends.
  - If the server starts successfully, you see the umrbussrv process running in the task manager. If the server cannot be started on the host machine, an error message is displayed.
3. To debug the design from a remote machine (client), launch the debugger on the client machine and load the design to be debugged. Then configure the client-side debugger as described below:
- In the debugger GUI, select Client/Server from the Debugger popup menu.
  - Specify the server address, port number, and log file name in the Configure Client/Server dialog box. Use the ipconfig (Windows) or /sbin/ifconfig (Linux) command to verify the name or tcp/ip4 address of the client. The port number must be the same as the port number used to configure the server.
  - If you are using the UMRBus, enable the Use Client/Server check box.

Once started, close the dialog box by clicking OK to save any changed settings or simply click Cancel to close.

The following syntax shows the equivalent TCL command to configure the server:

```
umrbus_server set -addr {hostName/IP_address} -port {serverPort}  
-logf {logFileName}
```

To view the existing server configuration settings, use the `umrbus_server get Tcl` command.

Check the client-server communication by running the `com check` command (click the Comm check button in the Setup panel). If the client-server communication cannot be established, an error message is displayed in the debugger.

The client-server architecture may not always work within a WLAN. Also, firewall restrictions as well as security software such as anti-virus or anti-spyware can also impact client-server communications.

Once the client-server communication is running properly, you can debug the design remotely.

## License Consumption

If you start a debugger session on the server machine, then load an instrumented project, and run a communications check, the server does not start in standalone mode. With this method, you cannot terminate the debugger session, and two licenses are consumed.

You can start the `umrbussrv` process in stand-alone mode on the server/host machine that interfaces to the HAPS hardware system either from the debugger GUI or from the command line. Both methods are described below.

1. Start the debugger on the HAPS system host.
2. Configure the client/server:
  - Select Debugger Client/Server.
  - In the dialog box, specify the port number
  - Set the cable type
  - Click Use Client/server
  - Set the server address to the hostname of the machine (localhost or 127.0.0.1)
  - Click the Start button to start the `umrbussrv` process, according to the cable type selected

3. Close the debugger session.

The server (`umrbussrv`) continues to run in standalone mode, without consuming a debugger license.

4. Verify that the `umrbussrv` process is running, using systems tools such as Task Manager or Process Explorer on Windows or `ps`, `top`, or `htop` on Linux.
5. As an alternative to the previous steps, start the process by running the command from the shell or command prompt.

```
umrbussrv -p portNum -l logfile
```

Use the `-` option with either of the commands to verify that the process is running. For example: `umrbus -`. For usage information about these commands, specify the `-?` option.

## Communication Cable Connection

There are two connections: cable-to-board and cable-to-host. The latest cable types use a USB connector to interface with the host and require a USB driver to be installed (see the installation procedures in the release notes).

The cable-to-board connection requires a type of mating connector or interface pod to connect to the board containing the device. The most recent cable types use a Xilinx USB platform ribbon cable on all currently supported HAPS platforms.

## Device Programming

Make sure that you program the device with the instrumented version of your design, NOT the original version.

# UMRBus Communications Interface

The UMRBus is available as a communication interface between the HAPS hardware and the host machine running the debugger. With the UMRBus, all communications are performed over the UMRBus communication system. During instrumentation, the top level of the user design is automatically extended with additional top-level ports for the UMRBus. The tool automatically constrains these UMRBus ports for pin location and timing.

The UMRBus communication interface is supported for all buffer types that target the HAPS platform. This UMRBus interface can also be used with user-defined CAPIMs. The interface is also used by the board bring-up utility. The UMRBus communication interface can also be used for client-server based debugging

To enable the use of the UMRBus in the debugger, do the following:

1. In the instrumentor, set the device `jtagport` option to `umrbus` in the Tcl Script window.
2. In the debugger, set the `com cabletype` option to `umrbus` in the Tcl Script window.

## UMRBus Communication Debugging

The debugger performs a number of diagnostic communication tests every time the “run” function is executed either by clicking the Run button or executing the run command.

Below is a list of communication related problems associated with UMRBus communications and some additional explanations.

### Local Client-Server Communications

To eliminate as many unknowns as possible, terminate any debugger and server applications such as `umrbussrv` and make sure that you are the only user working on the system.

- For a local test, start the debugger and open the Client/Server dialog box. Select the cable type, set the server address to `127.0.0.1` or `localhost`, and set the port number to `57015`. Click the Start button and check for a connection startup message. If not received, make sure that the cable type, server address, and port number entered are correct. Again click the Start button. If the server starts, the problem is resolved; press the Stop button.
- If the startup test still fails, either use another port or search the security software options installed on the machine. Check the rules from firewall (for the LAN adapter) and for all other components such as anti-virus or anti-spyware software. In most cases, the problem can be located through the rules, logs, or messages.
- If the problem persists, shutdown the server completely (and restart the computer) and create a new test with the client-server as the highest priority.

If the server now starts, the options from the firewall or from the security software are suspect.



## Remote Client-Server Communications

To eliminate as many unknowns as possible, clear memory from the debugger and server applications such as umrbussrv. Make sure that you are the only user working on the system.

WLAN is not supported directly. An administrator is required to set up the WLAN router with the appropriate rules on how the port is mapped from one network to another.

For an initial check, use the ping command:

```
ping computerName
```

Try the command from each machine with the appropriate computer name or address. If you can ping in both directions, it is safe to assume that the addresses were located and the responses received. This test is not conclusive, but it does indicate that the client server can work.

Repeat, step-by-step, the section [Local Client-Server Configuration, on page 105](#):

- Verify that the same cable type is specified on both the client and server sides.
- Make sure that the server address on the server side is either localhost or 127.0.0.1.
- Make sure that the server address on the client side is correct. Use the ipconfig (Windows) or /sbin/ifconfig (Linux) command to verify the name or tcp/ip4 address of the client interfacing the UMRBus hardware.
- Verify that the same port number is specified for both the client and server sides.

If all of the above are correct and client-server communication is not running properly, individually start a local test on each computer host using only the Start and Stop buttons. If both computers can be started and stopped locally (but not over the network), a problem with network configuration and/or security software is indicated.

## User Preferences File Impact on Remote Debugging

Communication options and settings are saved in the `userprefs.cfg` file which is located in the user profile. When debugging a design remotely, a problem can occur if the same `userprefs.cfg` file is used when logging in to both the client and server (if the user profile is defined as global, the specified configuration applies to all logins by that user which would include the client host for the debugger).

To avoid conflicts with dissimilar `userprefs.cfg` files, start the server (the host connected to the device) only after checking and changing any parameters in the dialog box or on the command line. Start the client with `run/com check`. Check the parameters in dialog box and change if needed. Save the settings by clicking OK and repeat this procedure each time you begin a new remote debug session.

# Index

---

## A

- activations
  - auto-saving 49
  - loading 48
  - saving 48
- asynchronous clocks 93

## B

- blocks
  - sampling 82
- board bringup 62
- board configuration tests 64
- breakpoints
  - activating 36
  - combined with watchpoints 82
  - folded 39
  - multiple 81

## C

- cable type 104
- cables
  - connection 111
- client-server configuration 105
- clocks
  - asynchronous 93
- communications settings 104
- complex counter 83
  - cycles mode 85
  - disabling 85
  - events mode 84
  - modes 83
  - pulsewidth mode 85
  - size 83
  - watchdog mode 85
- con\_speed test 66
- condition operators 90
- Configure IICE dialog box 57

- console window 26
  - operations 27
- convenience functions 92
- cross triggering 49, 93
  - commands 93
  - enabling 93
  - state machine commands 94
- cycles mode
  - complex counter 85

## D

- data compression 41
  - masking 42
- DDR3 performance 13
- debug sample data
  - viewing 14
- Debugger tool
  - invoking 18
- debugging
  - on separate machines 52
- deep trace debug configurations 11
- dialog boxes
  - Configure IICE 57

## E

- events mode
  - complex counter 84

## F

- fast signal database 55
- files
  - last\_run 49
  - script 32
  - syn\_trigger\_utils.tcl 92
- folded breakpoints 39
- folded signals 45

folded watchpoints [38](#)

## I

IICE

cross triggering [93](#)

IICE parameters

individual [57](#)

IICE units

cross triggering [49](#)

## J

JTAG

debugging [112](#)

## L

last\_run file [49](#)

## M

macros

st\_snapshot\_fill [101](#)

st\_snapshot\_intr [101](#)

modes

cross triggering [50, 51](#)

Multi-FPGA DTD performance [14](#)

multi-IICE

tabs [57](#)

multiple signal values [45, 46](#)

multiplexed groups

selecting [36](#)

## O

operators

condition [90](#)

original source files

searchpath [53](#)

original sources [53](#)

## P

pre-configured triggers [41](#)

pulsewidth mode

complex counter [85](#)

## Q

qualified sampling [100](#)

## R

radix

sampled data [44](#)

RAM resources [83](#)

remote triggering [101](#)

run command [20, 39](#)

## S

sample buffer [43](#)

trigger position [42](#)

sample data

viewing [14](#)

sample modes [100](#)

sampled data

changing radix [44](#)

compressing [41](#)

display controls [43](#)

masking [42](#)

sampling block [82](#)

sampling signals [23](#)

script files [32](#)

signal values

displaying multiple [45, 46](#)

signals

folded [45](#)

listing available [23](#)

listing instrumented [23](#)

multiply instrumented [45, 46](#)

partially instrumented [47](#)

sampling selection [23](#)

status [88](#)

source files

copying [53](#)

st\_snapshot\_fill macro [101](#)

st\_snapshot\_intr macro [101](#)

state machines

transitions [89](#)

triggering [88](#)

statemachine command [88](#)

state-machine editor [95](#)  
status reporting [88](#)  
stop command [40](#), [88](#)  
syn\_trigger\_utils.tcl file [92](#)

## T

tools  
  invoking Debugger [18](#)  
transition watchpoint [34](#)  
trigger conditions [86](#)  
triggering  
  advance mode [87](#)  
  between IICEs [93](#)  
  modes [86](#)  
  remote [101](#)  
  state machine [88](#)  
triggers  
  complex [83](#)  
  pre-configured [41](#)

## U

UMRBus [111](#)  
UMRBus test [67](#)

## V

value watchpoint [33](#)  
Verdi nWave viewer [55](#)

## W

watch command [88](#)  
watchdog mode  
  complex counter [85](#)  
watchpoints [81](#)  
  activating [33](#), [35](#)  
  combined with breakpoints [82](#)  
  deactivating [35](#)  
  folded [38](#)  
  hexadecimal values [34](#)  
  listing [51](#)  
  multiple [82](#)  
  transition [34](#)  
  value [33](#)  
waveform display [54](#)

waveform viewers [54](#)  
  Verdi [55](#)  
windows  
  console [26](#)

