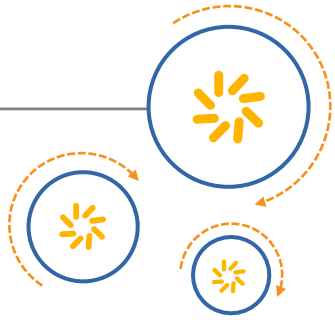
Qualcomm Technologies, Inc.



Hexagon TRACE32

User Guide

80-N2040-28 Rev. C

January 13, 2016

Questions or comments: support.cdmatech.com

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Contents

Tables

1 Introduction

1.1 Overview

TRACE32® is an industry-standard in-circuit debugging system developed by Lauterbach GmbH. It supports many target processors, including ARM® and Hexagon™ processors.

TRACE32 consists of both hardware and software:

- The hardware links the host computer with the hardware debug platform
- The software provides a GUI debug interface on the host computer

It can be used to debug and trace on hardware SURF boards, RUMI and ZeBu emulation platforms, and the Hexagon software simulator. All of these targets require the TRACE32 software; however, only the SURF boards and emulation targets require the TRACE32 hardware (which consists of an attached TRACE32 pod and license dongle).

This document describes only the Hexagon-specific features of TRACE32. For information on generic TRACE32 features see the Lauterbach documentation (which is provided with the Hexagon software development tools).

This document covers the following topics:

- Installing TRACE32 software on the host computer
- Configuring the TRACE32 hardware pod
- Creating shortcuts for the software
- Loading Hexagon binary images from the ARM processor
- Configuring the Hexagon processor for debugging
- Using TRACE32 to debug applications
- Debugging multithreaded applications
- Debugging applications on the software simulator

NOTE This document does not cover ETM configuration for TRACE32. For more information see the document `Training_Hexagon_ETM.pdf` in the Lauterbach TRACE32 documentation.

1.2 Block diagram

Figure 1-1 shows the block diagram of a TRACE32-based debugging system.

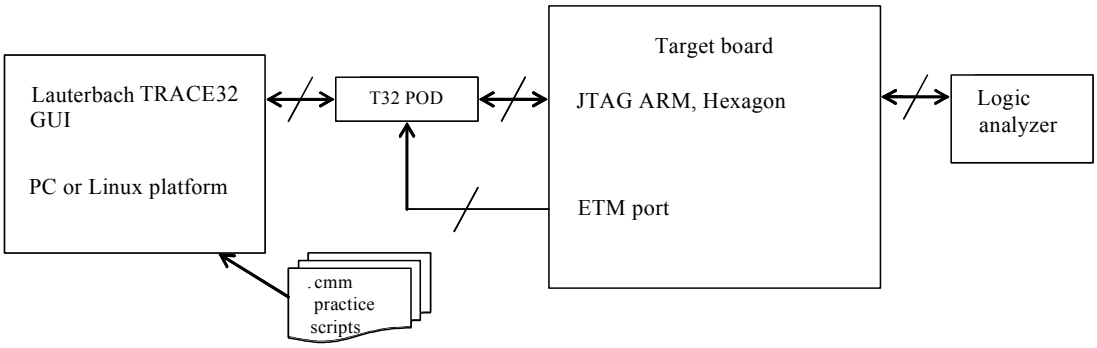


Figure 1-1 TRACE32 debugging system

1.3 Reporting problems

Most problems reported by TRACE32 users are ultimately caused not by the debugger but by incorrectly-configured hardware and software.

Before reporting any problems or requesting support, be sure that you're using the most recent version of TRACE32:

- Check for the latest patch version in the TRACE32 release directory.
- Check your current TRACE32 version by typing "`VERSION`" at the TRACE32 `B::` prompt.
- Include your TRACE32 version with any problem report or support request.

When reporting a problem or requesting support, be sure to do the following:

- When possible, create a simple test case which demonstrates the problem.
- Scripting errors are common sources of problems. If using them provide information on where they can be found, or attach them to the problem report.
- Debugger screen shots can be invaluable (and less work to create than written descriptions).

2 Installing TRACE32 Software

2.1 Overview

This chapter describes the procedure for installing the Hexagon TRACE32 software on a host computer running Windows, Linux (32- or 64-bit), Mac OS X, or Sun.

NOTE The Windows version of TRACE32 supports all Server, Desktop, and Embedded variations of Windows XP, Windows Vista, Windows 7, and Windows 8.

2.2 Install procedure

Obtain a copy of the standard Lauterbach install disc (a DVD titled "`POWERVIEW Product Software`").

Verify that the install disc is the most recent version available.

Locate the file `installation.pdf` that is stored on the install disc.

Perform the installation procedure described in `installation.pdf`.

3 Starting TRACE32

3.1 Overview

This chapter describes how to start the TRACE32 software.

3.2 TRACE32 command

Before TRACE32 can be started, the current directory must be set to the directory containing the TRACE32 executable file (`c:\t32` for Windows).

Table 3-1 lists the commands for starting TRACE32 on the host computer:

- The command name specifies the target processor (ARM or Hexagon)
- The `-c` option specifies the target-specific pod configuration file
- The `-s` option specifies the TRACE32 startup script

Table 3-1 TRACE32 start command

Host	Target	Command
Windows	ARM	`t32marm.exe -c arm_config.cfg -s arm_startup.cmm`
	Hexagon	`t32mqdsp6.exe -c hexagon_config.cfg -s hexagon_startup.cmm`
Linux	ARM	`t32marm -c arm_config.cfg -s arm_startup.cmm`
	Hexagon	`t32mqdsp6 -c hexagon_config.cfg -s hexagon_startup.cmm`

Specific file names for the pod configuration and startup script files are not shown in Table 3-1, because these file names are target-dependent.

NOTE On Windows, TRACE32 can alternatively be started with the command `t32start.exe`.

For more information on TRACE32 pod configuration files and startup scripts see Chapter 4 and Chapter 5.

For more information on starting TRACE32 (including the alternate start command) see the document `training_icd.pdf` in the Lauterbach TRACE32 documentation.

4 TRACE32 Pod Configuration

4.1 Overview

The TRACE32 hardware that links the host computer with the hardware debug platform is referred to as the *pod*.

The pod requires the following types of information to communicate with the host computer and hardware debug platform:

- Host software configuration
- Ethernet port assignment (if not using USB pod)

TRACE32 obtains this information from configuration files stored on the host computer.

Separate configuration files are used for the ARM and Hexagon processors.

NOTE Predefined configuration files are provided with each target system – users do not have to create these files or modify them.

For more information on pod configuration see the document `training_icd.pdf` in the Lauterbach TRACE32 documentation.

4.2 ARM configuration file

The following code example shows the TRACE32 ARM configuration file (`arm_mconfig.cfg`).

```
PRINTER=WINDOWS

; Environment variables
OS=
ID=unique_ARM_id
TMP=C:\T32\Temp
SYS=C:\T32

; Ethernet on Host information
PBI=
NET
NODE=ip_address_of_T32_probe
CORE=1

; Remote Control Access
RCL=NETASSIST
PORT=20000
```

4.3 Hexagon configuration file

The following code example shows the TRACE32 Hexagon configuration file (`hexagon_mconfig.cfg`).

```
PRINTER=WINDOWS

; Environment variables
OS=
ID=unique_Hexagon_id
TMP=C:\T32\Temp
SYS=C:\T32

; Ethernet on Host information
PBI=
NET
NODE=ip_address_of_T32_probe
CORE=2

; REMOTE Control Access
RCL=NETASSIST
PORT=20001
```

5 Loading Hexagon Images

5.1 Overview

This chapter describes how Hexagon binary images are loaded onto the hardware debug platform.

NOTE On many target systems, when the target device comes out of reset, its ARM core is the one that runs. The ARM is used to load code for the device's other cores into memory.

5.2 Script files

TRACE32 supports the use of a scripting language (named `PRACTICE`) to enable the automated execution of a sequence of commands. A script file is used to load Hexagon binary images into the hardware debug platform.

The load script files are dependent on the target system.

NOTE Predefined load script files are provided with each target system – users do not have to create these files or modify them.

For more information on load script files see the document `training_practice.pdf` in the Lauterbach TRACE32 documentation.

6 Configuring The Hexagon Processor

6.1 Overview

TRACE32 uses script files to load Hexagon binary images into the hardware debug platform (Section 5.2). This chapter describes how the script file is used to configure the Hexagon processor for executing the loaded image. It covers the following topics:

- Basic configuration
- Semihosting configuration
- RTOS application configuration

6.2 Basic configuration

The following TRACE32 properties must be set for all binary images:

- Processor version
- Symbol information

Processor version

The Hexagon processor version must be specified. For more information on processor versions see the *Hexagon Programmer's Reference Manual*.

Here is an example of the relevant code in the load script file:

```
System.cpu HexagonV4M
```

Symbol information

To use TRACE32, application symbol information must be loaded along with the application image.

Here is an example of the relevant code in the load script file:

```
data.load.elf filename
```

NOTE TRACE32 defines many other commands that can be used in load scripts. For more information see the following documents in the Lauterbach TRACE32 documentation: `general_func.pdf`, `general_ref_a.pdf` through `general_ref_z.pdf`, `ide_func.pdf`, and `ide_ref.pdf`.

6.3 Semihosting configuration

For applications that execute on a hardware target (or on the Hexagon simulator in a virtual platform), a host runtime environment must be simulated by the debugger using the ARM ANGEL interface. This process is known as *semihosting*.

To support the runtime environment the following configuration must be performed:

- The debugger must set a hardware breakpoint at the `trap0` handler.
- The debugger must be informed about the `trap0` handler symbol. The ANGEL address defaults to EVB+0x20.
- The debugger must be informed about the (application-specific) heap and stack locations.

Here is an example of the relevant code in the load script file:

```
; ---           Semi-hosting Setup              ---
;
;
break.set event_handle_trap0 /onchip
term.reset
term.heapinfo 0x0 0x10000 0x10000 0x10000

term.method ANGEL event_handle_trap0

term.scroll on
term.mode string
term.gate
```

6.4 RTOS applications

To support the runtime environment for RTOS application systems, the following configuration must be performed:

- RTOS awareness module
- Memory management
- Load symbol information

For more information see the appropriate Hexagon RTOS documentation.

7 Debugging Multithreaded Applications

7.1 Overview

This chapter describes the ISDB registers that are used to debug multithreaded applications. These registers control the following events:

- Which hardware threads enter into debug mode when a hardware or software breakpoint occurs
- Which hardware threads resume when an external resume request occurs

7.2 IDSB debug control registers

Table 7-1 lists the ISDB registers used to control multithreaded application debugging.

Table 7-1 ISDB debug control registers

Register Name	Bits	Description
ISDBCFG0	21:16	Hardware threads that resume execution when external resume request is received.
	13:8	Hardware threads that enter debug mode when external breakpoint is requested.
ISDBCFG1	29:24	Hardware threads that enter debug mode when hardware breakpoint 1 is hit.
	21:16	Hardware threads that enter debug mode when hardware breakpoint 0 is hit.
	13:8	Hardware threads that enter debug mode when software breakpoint is hit.

Each 6-bit register field corresponds to the six hardware threads (which are numbered 0-5, with 0 corresponding to the LSB in each register field). For each bit in these fields:

- 0: Corresponding hardware thread not specified
- 1: Corresponding hardware thread specified

The default setting for these register fields is 111111.

NOTE The use of register fields to control the hardware threads enables them to enter debug mode or resume execution in groups.

8 Debugging Applications on the Software Simulator

8.1 Overview

TRACE32 is normally used to debug programs on a hardware debug platform. However, when used with the MCD software extension, TRACE32 can be used to debug programs running on the Hexagon processor instruction set simulator (`hexagon-sim`).

This chapter describes how to use TRACE32 and MCD to debug programs on the simulator.

NOTE This chapter is not intended as documentation for either TRACE32 or the simulator – it assumes users are experienced with both these tools.

For more information see the document *Hexagon Simulator User Guide*.

8.2 MCD

The Multi-Core Debug (MCD) extension is a dynamic library which serves as an interface between TRACE32 and the simulator.

- It supports full multi-core debugging capabilities.
- It is available for both the Windows and Linux versions of TRACE32.

8.3 Installation

TRACE32 must be specifically configured to load the MCD dynamic library file. This is done using the TRACE32 configuration file, which specifies the filename (and optionally the pathname) of the MCD dynamic library.

The Hexagon processor software development tools provide the following example files to support the use of MCD:

- MCD dynamic library file
- TRACE32 configuration file
- TRACE32 startup script file

Table 8-1 lists the file names and their locations in the tools release directory.

Table 8-1 MCD support files

	Platform	T32	Name	Location
MCD Dynamic Library File	Linux	$T32SYS/bin/pc_linux/t32mqdsp6 $T32SYS/bin/pc_linux/t32mqdsp6-qt	hexagon-mcd32.so (or T32_MCD.so)	<*install_root*>\qc\lib\iss
		$T32SYS/bin/pc_linux64/t32mqdsp6 $T32SYS/bin/pc_linux64/t32mqdsp6-qt	hexagon-mcd64.so	
	Windows	%T32SYS%\bin\windows\t32mqdsp6.exe	hexagon-mcd32.dll (or T32_MCD.dll)	<*install_root*>\Tools\bin
		%T32SYS%\bin\windows64\t32mqdsp6.exe	hexagon-mcd64.dll	
TRACE32 Configuration File	Linux	–	linux.cfg	<*install_root*>/Examples/TRACE32/Linux_Example
	Windows	–	win.cfg	<*install_root*>\Examples\TRACE32\Windows_Example
TRACE32 Startup Script File	Linux	–	hexagon.cmm	<*install_root*>/Examples/TRACE32/Linux_Example
	Windows	–	hexagon.cmm	<*install_root*>\Examples\TRACE32\Windows_Example

The name of the dynamic library file is specified in the example configuration file with the following command:

```
PBI=MCD hexagon-mcd64.so     (Linux version)
PBI=MCD hexagon-mcd64.dll    (Windows version)
```

The pathname of the dynamic library file can be specified in two ways:

- A system environment variable (Table 8-2)
- The TRACE32 configuration file

Table 8-2 lists the path variables that are set to include the dynamic library pathname.

Table 8-2 MCD support variables

Platform	Variable	Location
Linux	LD_LIBRARY_PATH	<*install_root*>/Tools/lib/iss
Windows	PATH	<*install_root*>\Tools\bin

Alternatively, the MCD command in the TRACE32 configuration file can be modified to include the complete path to the dynamic library file. For example:

```
PBI=MCD C:\Qualcomm\HEXAGON_Tools\version\Tools\bin\hexagon-mcd64.dll
```

NOTE The startup script files are specified as command arguments in the command used to launch TRACE32 (Section 8.4.1).

8.4 Quick start guide

MCD can launch a new instance of the simulator, or connect to an existing simulator instance.

The MCD extension is configured in TRACE32 with the PRACTICE command SYSTEM.MCDCONFIG.

NOTE SYSTEM.MCDCONFIG must be performed *before* attempting the SYSTEM.MODE.ATTACH command.

8.4.1 Launching TRACE32/MCD with configuration file

The following commands launch TRACE32 and MCD using the example configuration and script files provided with the Hexagon processor tools release.

Linux:

```
t32mqdsp6 -c <install_root>/Examples/TRACE32/Linux_Example/linux.cfg
          -s <install_root>/Examples/TRACE32/Linux_Example/hexagon.cmm
```

Windows:

```
t32mqdsp6.exe -c <install_root>\Examples\TRACE32\Windows_Example\win.cfg
              -s <install_root>\Examples\TRACE32\Windows_Example\hexagon.cmm
```

8.4.2 Configuring MCD with SYSTEM.MCDCONFIG

MCD is configured in TRACE32 with the following command:

```
SYSTEM.MCDCONFIG [arch=version]
                 [hostname=hostname]
                 [port=port_number]
                 [debug=mask]
                 [simargs=valid_sim_arguments]
                 [interactive=(yes|no)]
```

The command arguments are defined as follows:

arch

The optional `arch` argument specifies the Hexagon processor version to simulate. The argument value is specified as `v`*X*, where *X* is one of the processor versions specified in the simulator option `-mv`*X*.

For example, specifying `v5a_256` as the `arch` argument causes MCD to launch the simulator with option `-mv5a_256`. For more information on the simulator options see the *Hexagon Simulator User Guide*.

The `arch` argument must match the processor architecture of the executable file being debugged.

NOTE The `arch` argument is ignored when attaching to an existing simulator.

As an alternative to using `arch`, the Hexagon processor version can be specified in the `simargs` argument using the simulator option `-mv`*X*.

hostname

The optional `hostname` argument specifies the host name of the machine a simulator instance is already running on. Any valid host name can be specified (including “localhost”).

`hostname` is used to connect TRACE32 to an existing (i.e., already running) instance of the Hexagon processor simulator. The instance may be running on the current machine or on another machine.

NOTE If `hostname` is specified, `port` must also be specified.

port

The optional `port` argument specifies the TCP/IP port that will be used to communicate with a simulator instance. The port number value can be any number that is acceptable to `strtoul()` and does not violate the standard rules for TCP/IP port numbers.

`port` is used to connect TRACE32 to an existing instance of the simulator. It must be specified if the simulator is launched independently of TRACE32.

`port` is also used to launch a simulator instance on the same machine TRACE32 is running on, but using a port number other than the default value (808368). In this case the `port` parameter is specified, but `hostname` is not.

Alternate port numbers are specified in cases where it is necessary to avoid port number collisions (for example, when running multiple MCD-extended debuggers on the same host machine, with each communicating through a different port).

NOTE As an alternative to using `port`, the port number can be specified in the `simargs` argument using the simulator option `-G`.

debug

The optional `debug` argument selectively controls the printing of diagnostic information by the MCD extension itself.

`debug` specifies a mask value which determines the type of information printed. The mask value bits are defined as follows:

```
/* DEBUG control, settable with SYS.MCDCONFIG debug=xx */
#define REGISTER_OPS 1     // register read/write message
#define MEMORY_OPS 2       // memory read/write msgs
#define STATUS_REPLY 4     // status reply msgs from simu
#define FLOW_MSGS 8        // program flow messages
#define PACKET_FLOW 0x10   // low-level packet diagnostics
#define ERROR_MSGS 0x20    // error states
#define TXLIST_MSGS 0x40   // transmit list diags
#define TRIGGER_OPS 0x80   // breakpoint diags
#define ALL_MSGS 0xFF      // all of the above
```

The diagnostic information is written to `stderr`, which can be captured by redirecting it to a file when launching TRACE32.

For example (on Windows):

```
t32mqdsp6.exe -c win.cfg win.cmm 2 > my_debug_output.txt
```

The resulting file is ASCII text.

`debug` is used to debug the MCD extension itself (which can be useful if it is the suspected source of a system bug). The generated diagnostic file may provide insight into the cause of a particular problem; however, it is primarily intended for use by the MCD maintainers.

NOTE The generated diagnostic file can grow rapidly and become extremely large.

When using `debug` consider minimizing the number of open TRACE32 windows, as each open window generates constant traffic to the simulator.

simargs

The optional `simargs` argument specifies simulator command arguments which will be passed to the simulator instance launched by MCD.

If used, the `simargs` argument must appear as the last argument in the `SYSTEM.MCDCONFIG` command.

The specified simulator command arguments are not checked by TRACE32 – the user is responsible for verifying that they are valid.

NOTE The best way to do this is to manually launch the simulator with the specified command arguments and see if they work as expected.

`simargs` is enabled only when MCD launches a new simulator instance. If MCD connects to an existing instance, `simargs` will collect the specified simulator command arguments but then do nothing with them.

interactive

The optional `interactive` argument controls whether popup windows appear when an error occurs (such as syntax errors or premature simulator exits).

`interactive` accepts the following values:

- `yes` – Enable popup windows (default)
- `no` – Disable popup windows

NOTE Popup windows are supported only in Windows.

Using the MCD arguments

The simplest and least error-prone way to launch a new simulator instance is to use only the `simargs` argument.

The simplest way to attach to an existing simulator instance is to specify only the `port` argument and optionally the `hostname` argument (which defaults to "`localhost`" if not specified). Then, after MCD attaches to the simulator, it will query the simulator for all the necessary details.

A Using TRACE32

A.1 Overview

This chapter describes various issues related to using TRACE32. It covers the following topics:

- Error messages "`debug port fail`" or "`debug port time out`"
- Debugger window displays flickering data
- Using wildcards in the symbol browser
- Warning message when stepping into functions
- Error message "`running(off)`"
- Error message "`Emulation debug port fail...`"
- Debugging exceptions in the target application
- TRACE32 fails with message "`bus error`"
- TRACE32 fails with message "`access timeout, processor running`"
- TRACE32 terminates with request for license file
- Running the debugger without the GUI

A.2 Error messages “debug port fail” or “debug port time out”

The TRACE32 debugger command `system.up` is typically the first command executed in a debug session.

If this command generates the error message “`debug port fail`” or “`debug port time out`”, open the debugger `AREA` window to view any additional messages that might have been generated (for example, “`target processor in reset`” is simply a follow-up error message).

The “`debug port`” error messages can be generated for a number of reasons:

- The target system has no power, or the debug cable is not connected to the target. This results in the additional error message “`target power fail`”.
- The command `system.cpu` was specified with an invalid processor type.
- The JTAG interface experienced an electrical problem – check the schematic.
- The debug features need to be enabled (jumpered) on the target. For example, the debugger will not work if signal `nTRST` is directly connected to ground on the target side. (This is a known issue on some SURF boards.)
- The target system is in an unrecoverable state – re-power it and try again.
- The target system cannot communicate with the debugger while in reset. If this occurs, instead of using the `system.up` command try using `system.mode attach` followed by `break`, or alternatively “`system.option enreset off`”.
- The default JTAG clock speed is too fast, especially when the target processor is emulated or when using an FPGA-based target. In this case try using the command “`system.jtagclock 50kHz`”; then after the target is working, try gradually increasing the specified clock speed.
- The target processor needs adaptive clocking. In this case specify RTCK mode with the command “`system.jtagclock rtck`” (which is typical for ARM cores).
- The debugger multicore settings are missing when the target is a daisy-chained multi-core system. To check this, enter the command “`diag 3400`” and then view the `AREA` window:
  - The value “`ir_width > 5`” indicates that the multicore settings are necessary.
  - The value “`ir_width = 4`” (ARM9) indicates that the multicore settings are unnecessary.
  - If no value is displayed, there may be a problem with the JTAG interface.
- The target processor has no clock.
- The target processor is held in reset.
- A watchdog timer exists which needs to be deactivated.

A.3 Debugger window displays flickering data

If the data in a debugger window is flickering, try one of the following solutions:

- Ensure that `turbo` mode is disabled (using the command "`system.option turbo off`").
- The default JTAG clock speed may be too fast. Try using the command "`system.jtagclock 50kHz`"; then if the flickering disappears, try gradually increasing the specified JTAG clock speed.
- Check that the MMU is correctly programmed. If not, the data displayed may not be the data you think it is.

A.4 Warning message when stepping into functions

When the TRACE32 debugger breaks/steps into a function, it attempts to read source code information based on the debug symbols it encounters in the newly-entered function. If the source code information can't be found, the debugger will generate a warning message.

If a warning message appears while you are debugging a program, two options are available for helping TRACE32 find the correct source path:

Solution 1:

Use the following command to remove six path variables and replace them with a string containing the path to the source code files. For example:

```
d.load.elf ..\..\build\ig_server.reloc /STRIPPART 6 /PATH L:\user\tools
```

Solution 2:

Use the command `symbol.sourcepath` to directly specify the source file. For example:

```
symbol.sourcepath + L:\user\tools\rtos\okl-1.3.4\libs\qurt\src\qmsgq\sys-iguana\qMsgQClient.c
```

A.5 Error message "running(off)"

If the TRACE32 debugger displays the message "`running(off)`" after you click on Continue, this indicates that a thread continued executing past a `stop` instruction. (Note that a thread normally cannot execute past a `stop`.)

The most probable cause for this event is that the target application crashed, which then caused the RTOS kernel to execute an assert which stopped all the threads. To debug the cause of the assert, use the procedure described in Section A.7.

A.6 Error message “Emulation debug port fail…”

If the TRACE32 debugger displays the message:

```
Emulation debug port fail. Unable to set up debugging, ISDB clock off
```

…this means that no clocks are reaching ISDB (as indicated by the ISDB enable register field ISDB_CLK_OFF being set to 1).

This usually indicates is that no clocks are being driven to the Hexagon at the point where the debugger tries to access ISDB. The debugger sets JTAG_ISDB_EN and then polls ISDB_CLK_OFF; if it does not see the value flip to zero after a period of time, the debugger gives up.

By default the debugger polls ISDB_CLK_OFF 100 times before giving up. The poll count can be changed by using the command “`DIAG 3003. xxx`”, where `xxx` specifies the poll count.

NOTE Changing the poll count will affect other timeouts, and may result in strange behavior if the count is set too high. Note also that the `DIAG` command is not supported by Lauterbach.

A.7 Debugging exceptions in the target application

Optimally, the RTOS kernel would respond to a user page fault or unhandled exception in the target application by performing the following actions:

1. Stop all other executing threads
2. Flush memory
3. Report the register state of the current hardware thread
4. Stop itself

However, hardware limitations in the Hexagon V2 processor prevent the current hardware thread from immediately stopping the other hardware threads.

Therefore, the best method for catching these exceptions is to set breakpoints on the following label symbols:

- `hexagon_page_fault`
- `hexagon_error_exception`

When the breakpoint is hit, use the debug tools to obtain the necessary register and thread information from the TCB (thread control block).

Table A-1 shows the memory structure of the TCB.

Table A-1 TCB structure

Field	Description
myself_global	Thread global ID
thread_state	Thread state ■ `running` (0x2): Thread running or runnable if context = -1 ■ `waiting_forever` (-1): Thread waiting to receive IPC from other threads ■ `waiting_timeout` (0x5): Thread waiting to receive IPC notify from other threads ■ `polling` (0xb): Thread waiting to send IPC to another thread, which is not ready to receive the IPC ■ `halted` (0xd): IRQ thread waiting to receive interrupt
partner	Thread partner Contains thread ID of the other thread involved in an IPC operation (or -1 for a wait-for-message from anyone)
priority	Thread priority
context	Hexagon processor unit (0-3) that hardware thread is currently using. Set to -1 if unit not scheduled.
scheduled_unit	Hexagon processor unit (0-3) that hardware thread is scheduled to use.

NOTE The information in this section applies only to the Hexagon V2 processor.

A.8 TRACE32 fails with message “bus error”

If the TRACE32 debugger displays the message “`bus error`” and then fails, this indicates that either TRACE32 or the RTOS awareness module attempted to access undefined memory.

The TRACE32 debugger maintains a cache copy of the MMU translations (which are a superset of the static mapping from ELF files, L4 page tables, and TLB).

The debugger scripts configure the MMU and provide information on the kernel MMU mapping and the base address for the root task page table. These scripts must be configured correctly.

The debugger is set to trigger an autoscan whenever it finds a mismatch. This scan involves the RTOS awareness module providing correct page table pointers. The RTOS awareness module may not be initialized to the correct physical address. A workaround is to set only on-chip breakpoints.

A.9 TRACE32 fails with message “access timeout…”

If the TRACE32 debugger displays the message “`access timeout, processor running`” and loses control of the target, this indicates that the debugger has tried to execute a command that accesses the target, but the target was running.

NOTE This message also appears when the stop of the target does not work (e.g., Hexagon is in the OFF state).

A.10 TRACE32 terminates with request for license file

Whenever TRACE32 is started with an unlicensed software component (e.g., processor or GDI extension), Lauterbach allows 60 minutes of free use, after which the application automatically terminates with a message stating that a valid license is required.

To use TRACE32 without having to restart it every 60 minutes, you must purchase licenses for the components in question.

A.11 Running the debugger without the GUI

To start the TRACE32 debugger without the GUI add the following command to the TRACE32 configuration file (`config.t32`):

```
SCREEN=OFF
```

NOTE Running TRACE32 without the GUI is typically done when a user needs to execute automated tests.