Qualcomm Technologies, Inc.

Qualcomm® Hexagon™ Simulator

User Guide

80-N2040-1784 Rev. E

June 8, 2021

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

Contents

Figures

Tables

1 Introduction

This document describes the Qualcomm® Hexagon™ processor instruction set simulator, which simulates the execution of Hexagon programs.

This document is intended for developers and support staff who work with system simulations and co-simulation models.

1.1 Conventions

`Courier` font is used for computer text and code samples, for example, `hexagon_<function_name>()`.

The following notation is used to define command syntax:

- Square brackets enclose optional items, for example, `[label]`.
- **Bold** indicates literal symbols for example, `[comment]`.
- The vertical bar character, `|`, indicates a choice of items.
- Parentheses enclose a choice of items for example, `(add|del)`.
- An ellipsis, `. . .` , follows items that can appear more than once.

1.2 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://createpoint.qti.qualcomm.com/.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

2 Overview

The simulator supports the following features:

- Interactive debugging (via TRACE32 or LLDB debugger)

  The simulator supports interactive debugging indirectly by serving as a simulation engine for the TRACE32 debugger (via MCD) or the text-based LLDB debugger. The simulator can also be used directly as a non-interactive simulation engine.

- Execution profiling (via gprof profiler)

  Execution profiling is supported using the text-based gprof profiler. Profiling can be performed on any Hexagon application (standalone, RTOS-based, single-threaded, or multi-threaded).

  The Hexagon profiler can collect and postprocess execution time statistics to produce an HTML that can be opened and navigated in any browser.

- Execution trace and statistics (via simulator or PMU)

  The simulator can generate execution traces and statistics. Statistics can be generated in either a simulator-specific format, or as Hexagon PMU statistics. The simulator-specific statistics can be generated for all or selected parts of the target application.

- External devices (via co-simulator plug-ins)

  The simulator supports plug-in co-simulators to enable co-simulations with external devices.

- Execution timing (via timer interface) and cache maintenance (via cache interface)

  The simulator provides timer and cache APIs (implemented as C libraries) that enable Hexagon applications to collect execution timing information and maintain the processor caches while they are being simulated.

- Client-controlled simulation (via APIs)

  The simulator provides an API that enables client program control of a Hexagon simulation, modeling the Hexagon core in a system simulation environment. The API is also used to develop and perform co-simulations with external devices and higher-level system simulations.

The simulator is not cycle-accurate. With timing enabled (see Section 3.3.8), the simulator is cycle-approximate with a performance target of 3% cycle accuracy when compared to similarly configured hardware. Various aspects of the micro-architecture are modeled accurately, but other factors such as bus clock ratio and bus delay will impact the overall accuracy.

2.1 Processor versions

The simulator supports multiple versions of the Hexagon processor:

- When used as a debugger simulation engine, the simulator automatically determines the processor version of an object file from information stored in the file.
- When used directly, the simulator defines command options (such as `-mv62`) that specify the target Hexagon processor architecture to simulate.

For more information on these (and related) command options, see Section 3.3.3.

For more information on the Hexagon processors, see the applicable *Qualcomm Hexagon Programmer's Reference Manual*.

Not all processor versions are supported in a specific Hexagon tools release.

Table 2-1 Supported processor versions

Tools release	Processor versions
8.0.x	V5, V55, V56, V60, V61, V62
8.1.x	V5, V55, V56, V60, V61, V62, V65
8.2.x	V5, V55, V56, V60, V61, V62, V65, V66
8.3.x	V55, V56, V60, V61, V62, V65, V66, V67, V67 Small Core (V67t)
8.4.x	V62, V65, V66, V67, V67 Small Core (V67t), V68

2.2 Type definitions

Throughout this document, variable types are specified with type definitions. These definitions are defined in the `HexagonTypes.h` file in the release distribution.

Table 2-2 Hexagon-specific type definitions

Type Name	Description
`HEX_<size>s_t` `HEX_<size>u_t`	Numeric value. ■ `<size>` = Digit denoting object size in bytes ■ `s` = Signed value ■ `u` = Unsigned value
`HEX_VA_t`	Hexagon virtual address. **NOTE:** `HEX_VA_t` is defined to be equivalent to `HEX_4u_t`.
`HEX_PA_t`	Hexagon physical address. **NOTE:** `HEX_PA_t` is defined to be equivalent to `HEX_4u_t`.
`HEXAPI_<name>`	API enumeration parameter.

3 Use the simulator

This chapter explains how to execute an application on the simulator. The following block diagram shows how the simulator is used.

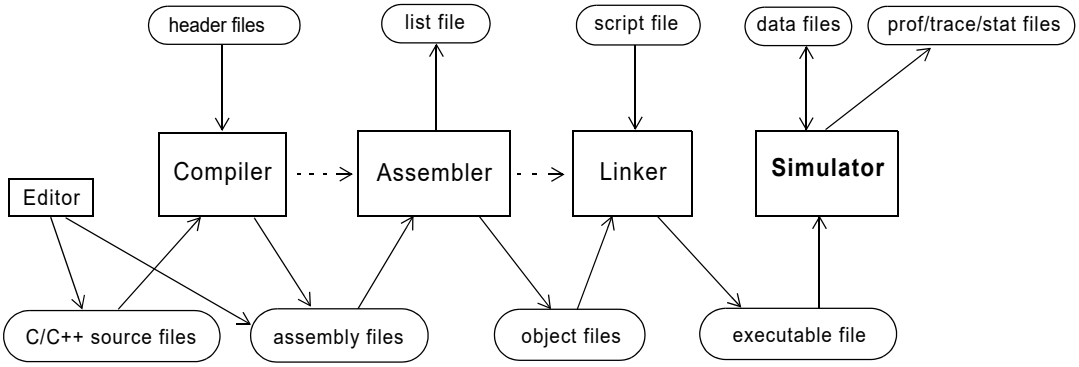

Figure 3-1 Using the simulator

You can use the simulator as follows:

- Simulation engine

  The simulator supports interactive source-level debugging for the TRACE32 or LLDB debugger. This use of the simulator is invisible to debugger users.

- Standalone application

  The simulator performs non-interactive simulation of target applications: target applications are executed straight through to the end. This is useful for tasks such as regression tests.

- API

  The simulator supports Hexagon processor core simulation in a system simulation environment. The simulator API offers a rich interface for simplified integration into system simulation environments.

3.1 Input to the simulator

The simulator accepts executable object files produced by the linker.

- Profile files – Binary data files that contain information on how the target application executed. This information is displayed by inputting the file to the Hexagon profiler.
- Trace files – Contain histories of how the application executed. Separate trace files can be generated for program counter accesses and for memory, bus, and cache accesses.
- Statistics files – Contain summaries of instruction usage, cache usage, and bus traffic.

The simulator might read or write data files as part of simulating file input/output operations performed by the target application. It might also generate files containing profile, trace, or statistics information.

3.2 Run the simulator

To run the simulator from a command line, type:

hexagon-sim [*option***...]** *input_file*

The simulator accepts the file name of the target application to be executed along with one or more command-line options. For example:

```
hexagon-sim -mv62 --rtos q.cfg a.out
```

Command switches are used to control various simulator options. A switch consists of one or two dash characters followed by a switch name and optional parameter. Note that switch names are case-sensitive. Switches must be separated by at least one space.

The command switch (`--`) when delimited by spaces on either side is used to separate the command arguments of the target application from those of the simulator. For example:

```
hexagon-sim --rtos q.cfg a.out -- 10  // 10 is target app arg
hexagon-sim --rtos q.cfg -- a.out 10  // alternate form
```

The file name argument can be specified on either side of the `--` switch.

To list the available command options, type:

```
hexagon-sim --help
```

The simulator displays on the screen the proper command line syntax, followed by a list of the available command options.

3.3 Simulator options

The simulator options are used to control simulation features from the command line. They are specified by the command switches listed below.

Several simulator options are used to specify the simulated target hardware environment. While these options do have default values, the defaults are not universally suitable for all target hardware environments. Thus QTI recommends explicitly specifying all the target hardware options as part of a simulation.

This practice is especially important when simulating program performance. For example, in cycle-sensitive simulations the bus ratio and bus penalty should be specified for all buses (processor, AHB, AXI2) used in the target hardware environment. If simulated time is the goal of the simulation, the DSP clock frequency should be specified in addition to parameters that have a significant impact on the cycle count.

Many additional revision IDs (or core IDs) are not listed here. To view them for a specific Hexagon simulator version, use **`hexagon-sim -h -mv`**<*x*> with the respective hexagon architecture version, <x>.

NOTE: Some options have alternate abbreviated switches defined for ease of use.

Simulator information (Section 3.3.1)

```
--build_tag
--help [version] | -h [version]
--version | -v
```

Status messages (Section 3.3.2)

```
--quiet | -q
--verbose
```

Processor version (Section 3.3.3)

```
-mv62 | -mv65  | -mv66 | -mv67 | -mv67t | -mv68
```

Processor attributes (Section 3.3.4)

```
--ahbbuspenalty bus cycles
--ahbbusratio ratio
--axi2buspenalty bus cycles
--axi2busratio ratio
--buspenalty bus cycles
--busratio ratio
--dsp_clock MHz
--stackstart address
--v2p_translation mode
```

Simulation environment (Section 3.3.5)

```
--connection_timeout <seconds>
--coredump filename
--cosim_file filename
--gdbserv port | -G port
--nullptr action
--reconnect
--sim_in filename | -I filename
--sim_out filename | -O filename
--sim_err filename | -E filename
--simulated_returnval
--usefs pathname
```

RTOS applications (Section 3.3.6)

```
--rtos filename
--symfile filename
```

Memory initialization (Section 3.3.7)

```
--memfill value
--memfill_rand seed
```

Processor modeling (Section 3.3.8)

```
--bypass_idle
--timing
--timing_nodbc
```

TCM modeling (Section 3.3.9)

```
--tcm:lowaddr address
--tcm:highaddr address
```

System configuration modeling (Section 3.3.10)

```
--etm_base value
--l2cfg_base value
--l2tcm_base value
--subsystem_base value
```

Bus modeling (Section 3.3.11)

```
--ahb:lowaddr address
--ahb:highaddr address
--axi2:lowaddr address
--axi2:highaddr address
```

gprof Profiling (Section 3.3.12)

```
--fast-profile
--profile | -p
--profile_timezero mode
```

Statistics (Section 3.3.13)

```
--packet_analyze filename
--pmu_statsfile filename
```

Trace (Section 3.3.14)

```
--bustrace filename | -b filename
--coproctrace filename
--memtrace filename | -m filename
--pctrace filename | -t filename
--pctrace_min filename | -u filename
--pctrace_nano filename
--uarchtrace filename
```

Cache trace (Section 3.3.15)

```
--dcachetrace filename
--icachetrace filename
--l2cachetrace filename
```

Filtering (Section 3.3.16)

```
--pcfilter startaddress-endaddress
--timefilter_ns starttime-endtime
```

3.3.1 Simulator information

These options are used as sole command arguments; they do not perform any simulation.

--build_tag

Display the git tags used to build the simulator and accompanying libraries to the console, and exit.

The output can be redirected into a file. The contents of the file are useful when reporting a problem related to the simulator.

--help [*version*]
-h [*version*]

Display simulator command options and exit.

If a processor version option is specified as an argument (`-help -mv62`, and so on), the simulator additionally lists information on all the processor version options that are defined for the specified processor version.

If the specified option includes both a processor version and micro-architecture (such as `-mv62a_512`), the simulator additionally lists the information for only that option.

Using the *version* argument is the only way to list all the processor version options that are defined for Hexagon V62 and later processors.

For more information on the processor version options, see Section 3.3.3.

--version
-v

Display the version numbers of the simulator and its associated API, and exit.

For more information on the simulator API, see Chapter 4.

3.3.2 Status messages

--quiet
-q

Do not display simulator-generated messages.

--verbose

Display any warning messages.

Warnings can be generated by multiple causes such as processor interrupts and exception events (Section 3.5) being generated by the target application.

3.3.3 Processor version

-mv62
-mv65
-mv66
-mv67
-mv67t
-mv68

Specify the Hexagon processor version and micro-architecture to simulate. These versions have properties listed when you use the `hexagon-sim -h -mv<x>` option with the respective Hexagon architecture version, `<x>` (see Section 3.3.1).

The simulator input file must have been generated with a processor version that is compatible with the specified processor version.

The default version is the one specified in the executable object file.

The generic options (such as `-mv62`) are provided for cases where the specific micro-architecture is not relevant.

NOTE: Not all `-mv<x>` options are supported in a specific Hexagon tools release. For more information, see Section 2.1.

For more information on the processors, see the applicable *Qualcomm Hexagon Programmer's Reference Manual*.

3.3.4 Processor attributes

--buspenalty *bus cycles*

Specify the AXI (main) bus access latency in terms of bus cycles. The latency is specified as an integer value. A space must appear after the switch name.

For Hexagon V5 and later processors, latencies are expressed in terms of bus cycles.

For Hexagon V6x processors, the default latency is 75 bus cycles.

The bus is simulated at its own frequency, as determined by the `--dsp_clock` and `--busratio` option settings.

--busratio *ratio*

Specify the AXI (main) bus frequency in terms of a ratio with the Hexagon processor frequency.

For Hexagon V6x processors, the default ratio is 2.

If the ratio is set to 1, a bus cycle is defined to equal a pcycle. If the ratio is set to 2, a bus cycle is defined to be twice as long as a pcycle (and so on). For example, if the processor runs at 600 MHz and the ratio is 3, the bus runs at 200 MHz.

--ahbbuspenalty *bus cycles*

Specify the AHB bus access latency in terms of bus cycles. The latency is specified as an integer value. A space must appear after the switch name.

For Hexagon V5 and later processors, latencies are expressed in terms of bus cycles.

For Hexagon V6x processors, the default latency is 75 bus cycles.

The bus is simulated at its own frequency, as determined by the `--dsp_clock` and `--ahbbusratio` option settings.

--ahbbusratio *ratio*

Specify the AHB bus frequency in terms of a ratio with the Hexagon processor frequency.

For Hexagon V6x processors, the default ratio is 2.

--axi2buspenalty *bus cycles*

Specify the AXI2 bus access latency in terms of bus cycles. The latency is specified as an integer value. A space must appear after the switch name.

For Hexagon V5 and later processors, latencies are expressed in terms of bus cycles.

For Hexagon V6x processors, the default latency is 75 bus cycles.

The bus is simulated at its own frequency, as determined by the `--dsp_clock` and `--axi2busratio` option settings.

--axi2busratio *ratio*

Specify the AXI2 bus frequency in terms of a ratio with the Hexagon processor frequency.

For Hexagon V6x processors, the default ratio is 2.

--dsp_clock *MHz*

Specify the simulated processor clock speed (in MHz). The clock speed is specified as an integer value. A space must appear after the switch name.

The clock speed determines the ratio of simulated to real time (Section 3.4).

The default clock speed is platform-specific – it is typically in the range of 450 to 600 MHz.

NOTE: If you intend to convert cycle counts to simulated execution times, QTI recommends explicitly specifying a clock speed and not depending on the default value.

--stackstart *address*

Specify the base address of the processor stack for thread 0. Used for standalone applications.

The default stack base address is 0x40000000.

For more information on changing the stack location, see the *Qualcomm Hexagon Stand-alone Application User Guide* (80-N2040-22).

--v2p_translation *mode*

Specify whether or not virtual-to-physical address translation is performed for applications where the Hexagon processor MMU is enabled.

- If `mode` is set to 1, the simulator performs virtual-to-physical address translation.
- If `mode` is set to 0, the simulator assumes all addresses are physical.

The default mode is 0.

When virtual-to-physical address translation is disabled in the simulator, it can still be performed by an external object such as a debugger or OS Awareness module.

3.3.5 Simulation environment

--connection_timeout *<seconds>*

Optionally used when `--gdbserv` is used to limit the waiting time for the debugger front end to connect.

Without this flag, the simulator waits forever. With this flag, the simulator waits for the specified seconds, prints an error message, and then quits (if no connection is made).

--coredump *filename*

Generate a binary core dump file when the target application is terminated after generating an exception.

If an exception occurs when this option is not specified, the simulator will display a message suggesting that you generate a core dump file and analyze it in the debugger. For more information, see the *Hexagon LLDB Debugger User Guide* (80-N2040-31).

--cosim_file *filename*

Specify the co-simulators that are to register with the simulator. A space must appear after the switch name.

The specified file is a text file that contains a list of pathnames specifying co-simulator library files (`*.dll` on Windows, `*.so` on Linux). Alternatively, the co-simulator library file names can be specified in the text file without pathnames, with the pathname being set in an environment variable (`PATH` on Windows, `LD_LIBRARY_PATH` on Linux).

Predefined arguments to a co-simulator are specified after the corresponding library file name. For more information on co-simulators, see Chapter 4.

--gdbserv *port*
-G *port*

Specify TCP/IP socket that the simulator will write status information to during the simulation. The socket is specified as an integer value.

Typically, this option is not user-specified. It is used by the TRACE32 and LLDB debuggers to automatically start the simulator and connect to it.

--nullptr *action*

Specify the action taken by the simulator when the target application performs a NULL pointer dereference. The action is specified as an integer value.

Supported values for *action*:

- 0 – Ignore any NULL pointer dereferences

  A NULL pointer dereference is defined as an indirect memory reference through virtual address 0.

- 1 – Write warning message to the standard output
- 2 – Write error message to standard output and exit simulation (default)

--reconnect

Cause the simulator to continue running when the associated debugger front end exits.

The simulator continues to execute in its current state (on hold, running a simulation, and so on) while also monitoring the socket connection. After a fixed period of time (such as 120 seconds) to allow TCP to wait out an obsoleted socket, the simulator will reconnect if another debugger front-end attempts to connect to it.

By default, the simulator exits whenever the debugger front-end exits.

--sim_err *filename*
-E *filename*

Use the specified file as the standard error stream for the target application. A space must appear after the switch name.

--sim_in *filename*
-I *filename*

Use the specified file as the standard input for the target application. A space must appear after the switch name.

--sim_out *filename*
-O *filename*

Use the specified file as the standard output for the target application. A space must appear after the switch name.

--simulated_returnval

Cause the simulator to return a specific value to its caller indicating the final execution status of the target application.

--usefs *pathname*

Cause the simulator to search for files in the directory with the specified path. It is used for accessing shared object files that are loaded during program execution.

If this option is not specified, the simulator assumes that all such files are stored in the current directory.

NOTE: This option affects only files being opened for reading. New files are always created in the current directory.

3.3.6 RTOS applications

--rtos *filename*

Simulate the specified target application as an RTOS application.

The *filename* argument specifies the name of a text file that contains the pathname for an *RTOS Awareness module*, a dynamic library that enables the simulation and debugging of an application system. The specified RTOS Awareness module must match the processor version used in the simulation (Section 3.3.3). For more information, see the corresponding RTOS documentation.

If the `--rtos` option is not used, the simulator assumes that the target application is a standalone application.

NOTE: The timer co-simulator must always be specified (with `--cosim_file`) when running an RTOS application. For more information, see the corresponding RTOS documentation.

--symfile *filename*

Load symbols from the specified relocatable file. A space must appear after the switch name.

When an RTOS application system is built, the symbol information contained in the relocatable files is removed (stripped) from the final executable file. The simulator requires this information to execute the application system; therefore, to provide this information, the affected relocatable files must be specified with the `--symfile` option.

Multiple files must be specified with multiple `--symfile` options.

NOTE: When running an RTOS application, the RTOS kernel file must always be specified with this option.

3.3.7 Memory initialization

--memfill *value*

Initialize all bytes of simulated processor memory with the specified value before loading and executing the target application.

The default value is 0x1F.

--memfill_rand *seed*

Initialize all bytes of simulated processor memory with random values before loading and executing the target application. The specified integer value indicates the seed of the random number sequence used to generate the random values.

NOTE: This option is used for application testing in situations where uninitialized values might be causing application errors.

3.3.8 Processor modeling

`--bypass_idle`

Skip the execution of pcycles while the processor is idle due to all hardware threads being in `wait` or `off` mode. This option speeds up the simulation when the processor is idle for a significant amount of time.

If the number of skipped cycles is nonzero, the simulator screen messages include an extra line that indicates the number of pcycles spent while the processor is idle in this way (Section 3.4).

By default, this option is OFF for all architectures. Using it can accelerate programs with the DSP frequently in Idle mode, but using it slows down programs that rarely have the DSP in Idle mode.

`--timing`

Model the following processor micro-architecture as part of the simulation:

- Cache (attributes are modeled, but not the actual data)
- Multi-threading mode (SMT, IMT)
- Processor stalls

When timing is enabled, the simulation more accurately models the operation of the processor hardware. This impacts the speed of the simulation and improves the accuracy of the profiling PMU statistics collected (Section 3.8).

Cache

For Hexagon V5 and later processors, perfect caches are assumed and no modeling of data is performed. Instead, all data requests that would be serviced from the cache in the actual hardware are serviced by the backing memory storage model.

Multi-threading mode

For some Hexagon V6x processors, processor modeling includes the modeling of Simultaneous Multi-Threading (SMT) mode. In SMT mode, the four hardware threads are divided into two clusters. In many cases, packets can execute simultaneously on both clusters, so up to two packets can commit in a cycle.

Without SMT, the processor modeling default is Interleaved Multi-Threading (IMT) mode, where packets are executed on the hardware threads in round-robin order. (This mode is commonly known as a *barrel processor*.)

NOTE: For more information on caches and multi-threading, see the applicable *Qualcomm Hexagon Programmer's Reference Manual*.

`--timing_nodbc`

A legacy option maintained for backward compatibility.

For Hexagon V5 and later processors, this option is internally mapped to `-timing`.

3.3.9 TCM modeling

NOTE: For more information on TCM, see the applicable *Qualcomm Hexagon Programmer's Reference Manual*.

--tcm:lowaddr *address*
--tcm:highaddr *address*

Use the specified low and high memory addresses as the address range of the TCM memory area. A space must appear after the switch name.

NOTE: QTI recommends reading the TCM starting address from the CFG table and avoiding hard-coding the TCM address inside your code.

3.3.10 System configuration modeling

These options are supported only in the Hexagon 7.x tool releases.

For more information on system configuration modeling, see the applicable *Qualcomm Hexagon System-level Architecture Specification*.

NOTE: Avoid using these options unless absolutely necessary. The effect is to load a specific value into a specific slot of the CFG table (a read-only area in the actual hardware) and nothing else.

--etm_base *value*

Specify the ETM base address value that is stored in the configuration table referenced by the Hexagon system-level register CFGBASE.

The base address value specifies bits [35:16] of a 36-bit physical memory address. Bits [15:0] are assumed to be 0.

--l2cfg_base *value*

Specify the L2 configuration base address value that is stored in the configuration table referenced by the Hexagon system-level register CFGBASE.

The base address value specifies bits [35:16] of a 36-bit physical memory address. Bits [15:0] are assumed to be 0.

--l2tcm_base *value*

Specify the L2 TCM base address value that is stored in the configuration table referenced by the Hexagon system-level register CFGBASE.

The base address value specifies bits [35:16] of a 36-bit physical memory address. Bits [15:0] are assumed to be 0.

The simulator sets the value of the CFGBASE register itself to (*value* + `0x18`), where `0x18` denotes `0x180000` in the physical address space.

If this option is not used, the simulator assigns the L2 TCM base address a default value that is internally determined.

--subsystem_base *value*

Specify the subsystem base address value that is stored in the configuration table referenced by the Hexagon system-level register CFGBASE.

The base address value specifies bits [35:16] of a 36-bit physical memory address. Bits [15:0] are assumed to be 0.

3.3.11 Bus modeling

NOTE: To enable an address decode range for AHB or AXI2, both the low and high addresses must be specified. For details on AHB and AXI2, see Chapter 4.

--ahb:lowaddr *address*
--ahb:highaddr *address*

Use the specified low and high memory addresses as the address decode range for the AHB bus interface. A space must appear after the switch name.

--axi2:lowaddr *address*
--axi2:highaddr *address*

Use the specified low and high memory addresses as the address decode range for the AXI2 bus interface. A space must appear after the switch name.

3.3.12 gprof Profiling

--fast-profile

Generate a profile data file (named `gmon`) for use with the text-based Hexagon code coverage profiler.

Fast profile data files contain information on how the target application executed, with reduced called-function accuracy to improve profiling times. Only the function call graph is affected. Every function called is still counted. The information is displayed by inputting the files to the code coverage profiler.

When the target application is a multi-threaded standalone application this option creates separate profile data files for each hardware thread (named `gmon.t_0`, `gmon.t_1`, ...).

When the target application is a multi-threaded RTOS application, this option creates separate profile data files for each software thread (named `gmon.name1`, `gmon.name2`, ...) and an additional profile data file for the application startup code (named `gmon.t_Startup_0`).

By default, the simulator generates profile data files in a format that is specific to the Hexagon processor. The processor-specific data format differs from the standard GNU format by enabling histogram buckets to contain values up to 2^{64}-1. The values are pseudo-encoded values of variable length. This change removes the scaling inaccuracies that resulted from rounding values stored in the GNU-style profile data files.

NOTE: The `gmon.t_Startup_0` file is created only in the 7.x tools releases.

For more information on profile data files, see Section 3.6. For more information on the code coverage profiler, see the *Hexagon Code Coverage Profiler User Guide* (80-N2040-20).

--profile
-p

Generate a profile data file (named `gmon`) for use with the text-based Hexagon gprof profiler or code coverage profiler.

Profile data files contain information on how the target application executed. The information is displayed by inputting the files to the gprof profiler.

When the target application is a multi-threaded standalone application this option creates separate profile data files for each hardware thread (named `gmon.t_0`, `gmon.t_1`, ...).

When the target application is a multi-threaded RTOS application, this option creates separate profile data files for each software thread (named `gmon.name1`, `gmon.name2`, ...) and an additional profile data file for the application startup code (named `gmon.t_Startup_0`).

By default, the simulator generates profile data files in a format that is specific to the Hexagon processor. The processor-specific data format differs from the standard GNU format by enabling histogram buckets to contain values up to 2^{64}-1. The values are pseudo-encoded values of variable length. This change removes the scaling inaccuracies that resulted from rounding values stored in the GNU-style profile data files.

NOTE: The `gmon.t_Startup_0` file is created only in the 7.x tools releases.

For more information on profile data files, see Section 3.6. For more information on the gprof profiler, see the *Qualcomm Hexagon gprof Profiler User Guide* (80-N2040-29).

--profile_timezero *mode*

Specify whether or not gprof profiling starts from time zero.

For RTOS applications, a zero start time indicates that profile data collection begins immediately, and thus includes the OS boot-up instructions.

If `mode` is set to `1`, gprof profile data collection starts from time zero.

If `mode` is set to `0`, OS boot-up instructions are not included in the profile data.

NOTE: Previous simulator versions that did not support this option behaved as if the mode were set to 0.

3.3.13 Statistics

--packet_analyze *filename*

NOTE: This option is supported only in Hexagon V6x processors. It applies to the Hexagon processor and coprocessor concurrently.

Generate a packet statistics file with the specified name. A space must appear after the switch name.

Packet statistics files contain the number of commits, stalls, and bus accesses performed by the instruction packet at the specified memory address. The stalls and bus accesses are expressed in terms of PMU events.

The Hexagon profiler can postprocess the output to produce an HTML file.

If a file name is not specified, a packet statistics file is not generated.

For more information on packet statistics files, see Section 3.9.

--pmu_statsfile *filename*

Generate a PMU statistics file with the specified name. A space must appear after the switch name.

PMU statistics files contain the raw PMU event values for each hardware thread. The PMU monitors a wide variety of events related to instruction scheduling, and bus and cache accesses.

This option must be used with the `--timing` option (Section 3.3.8).to ensure that the generated statistics file contains valid data.

If this option is not specified, a PMU statistics file is automatically generated with the default file name `pmu_statsfile.txt`.

For more information on PMU events, see Section 3.8.

3.3.14 Trace

--bustrace *filename*
-b *filename*

Generate a bus trace file with the specified name. A space must appear after the switch name.

Bus trace files contain a record of the buses accessed during execution of the target application.

NOTE: This option must be used with the `--timing` option (Section 3.3.8).

For more information on bus trace files, see Section 3.7.3.

--coproctrace *filename*

Produce a trace of micro-architecture-level activities for the HVX coprocessor.

Because it is a micro-architecture-level trace, the file size is huge and the contents might be slightly different between simulator versions. For the output to be useful, the operation requires `--timing`.

--memtrace *filename*
-m *filename*

Generate a memory trace file with the specified name. A space must appear after the switch name.

Memory trace files contain a record of the memory accesses performed during execution of the target application (including both instruction and data accesses).

For more information on memory trace files, see Section 3.7.2.

--pctrace *filename*
-t *filename*

Generate a program counter trace file with the specified name. A space must appear after the switch name.

Program counter trace files contain the following information:

- A record of the program counter (PC) value at each cycle in the target application, including all stall cycles.
- The instruction packet executed at each PC value (for committed cycles only).
- The contents of all processor registers after each cycle.

For more information on program counter trace files, see Section 3.7.1.

--pctrace_min *filename*
-u *filename*

Generate a minimal program counter trace file with the specified name. A space must appear after the switch name.

Minimal program counter trace files contain less information than regular program counter trace files – they include only a record of the program counter (PC) value at each cycle in the target application. Unlike regular program counter trace files, they do not include information on stalls, packets, or registers.

NOTE: This option is a subset of `--pctrace` and overrides it.

For more information on program counter trace files, see Section 3.7.1.

--pctrace_nano *filename*

Generate an ultra minimal program counter trace file with the specified name. A space must appear after the switch name.

Ultra minimal program counter trace files contain less information than minimal program trace files. They are intended for use when you are first investigating the behavior of a program and need only minimal information on the program counter (PC) value.

NOTE: This option is a subset of `--pctrace` and `--pctrace_min`, and it overrides both of them.

For more information on program counter trace files, see Section 3.7.1.

--uarchtrace *filename*

Generate a micro-architecture trace file with the specified name. A space must appear after the switch name.

Micro-architecture trace files contain a record of major micro-architecture events (cache misses, packets executed, stall cycles, bus requests, and so on).

Micro-architecture trace files can generate huge amounts of data. Therefore, this option should be used with filtering to limit the collection of trace file data to relatively small time periods.

NOTE: Because of differences in the processor architecture, the contents of a micro-architecture trace file might differ between processor versions.

For more information on micro-architecture trace files, see Section 3.7.4.

3.3.15 Cache trace

--dcachetrace *filename*

Generate a data cache trace file with the specified name. A space must appear after the switch name.

Data cache trace files contain a record of the data cache accesses during execution of the target application.

For more information on data cache trace files, see Section 3.7.6.

--icachetrace *filename*

Generate an instruction cache trace file with the specified name. A space must appear after the switch name.

Instruction cache trace files contain a record of the instruction cache accesses during execution of the target application.

For more information on instruction cache trace files, see Section 3.7.7.

--l2cachetrace *filename*

The `--l2cachetrace` switch causes the simulator to generate an L2 cache trace file with the specified name. A space must appear after the switch name.

L2 cache trace files contain a record of the L2 cache accesses during execution of the target application.

NOTE: The cache trace options can be used only if processor modeling is enabled (see Section 3.3.8).

For more information on L2 cache trace files, see Section 3.7.7.

3.3.16 Filtering

--pcfilter *startaddress-endaddress*

Constrain the collection of trace and statistical data on instruction execution to the specified memory address range. No space can appear between the dash and the address values.

Data collection begins when the instruction at the specified start address is executed, and it continues until the instruction at the specified end address is executed.

Once constrained data collection ends, it will not start again during the program execution (even if the start address is executed again).

NOTE: Filtering applies to all forms of data collection in the simulator: profile, statistics, and trace data.

Filtering by address works in conjunction with filtering by time (using the `--timefilter_ns` option). In particular, filtering begins when either the start-address or start-time condition is satisfied, and it ends when either the end-address or end-time condition is satisfied.

--timefilter_ns *starttime-endtime*

Constrain the collection of trace and statistical data on instruction execution to the specified simulation time range. Time is expressed in units of nanoseconds. No space can appear between the dash and the address values.

Data collection begins when the simulation reaches the specified start time, and it continues until the simulation reaches the specified end time.

Once constrained data collection ends, it will not start again during the program execution.

NOTE: Filtering applies to all forms of data collection in the simulator: profile, statistics, and trace data.

Filtering by time works in conjunction with filtering by address (using the `--pcfilter` option). In particular, filtering begins when either the start-time or start-address condition is satisfied, and it ends when either the end-time or end-address condition is satisfied.

3.4 Screen messages

When the simulator is executed, it displays the following information on the screen:

```
$ hexagon-sim -mv62 a.elf
Hexagon-sim INFO: the rev_id used in the simulation is 0x00004062
(v62a_512)
hello, world

Done!
T0: Insns=5648 Tcycles=7931
T1: Insns=0 Tcycles=0
T2: Insns=0 Tcycles=0
T3: Insns=0 Tcycles=0
Total: Insns=5648 Pcycles=47587
```

The processor version that is being simulated (in this case, V62) is displayed immediately after the simulator command.

While not necessary, QTI recommends that you always specify the processor version on the command line (see Section 3.3.3). Specify the exact revision ID (such as `-mv62a_512`) so the simulator behaves as close as possible to the usage scenario. Remember that within the same architecture versions, all the core variations can have very different parameters.

Any text written to the standard output by the target application (`hello, world` in this case) is displayed immediately after the processor version information.

When the simulator executes to the end of the target application, it displays the following information:

- The status message, `Done!`
- The number of instructions executed and the thread cycle count (Tcycles) for each hardware thread (`T0` through `T5` for processor versions with six threads)
- The total number of instructions executed and total processor cycle count (Pcycles) for all hardware threads

NOTE: For Hexagon V5 and later processors, the end of simulation statistics are focused on packet counts and pcycles.

NOTE: The simulator-generated messages can be suppressed with the `--quiet` option (Section 3.3.2).

Additional information

In some Hexagon tools releases, the simulator displays the following additional information on the screen:

- The simulator speed (in terms of millions of instructions per wall-clock second)
- The processor clock rate, the speed ratio of the simulator to the actual processor, and the simulator execution time in wall-clock seconds

The processor clock rate (600 MHz in this case) is determined by the `--dsp_clock` option (Section 3.3.4).

The speed ratio indicates that the target application was simulated at a rate that is a certain number of times slower than a 600 MHz Hexagon processor would execute the same application in real time (based on wall-clock seconds). The ratio is based on processor cycles, not on instructions per second – it is computed with the following formula:

(Pcycles / wall_clock_seconds) / processor_clock_MHz

NOTE: The reported simulator speed and speed ratio may vary according to factors such as the overall computer load and complexity of instructions being simulated. However, any such variations will not in any way affect the performance of the algorithm being simulated.

Idle modeling

When `--bypass_idle` is used (Section 3.3.8) and the number of recorded idle cycles is nonzero, the simulator screen messages include an extra line indicating the total number of processor cycles simulated, which includes the simulated idle cycles:

```
Total simulated Pcycles (including wait) = 1000013401
```

3.5 Warning messages

The simulator displays warning messages on the screen to indicate various events. Here are some of the more common ones:

- Incorrectly-specified command-line options or option arguments
- Command line is too long
- NULL pointer dereferences were performed by the target application
- Hexagon processor interrupts or exceptions that occur while the target application is executing

For example:

```
WARNING: NULL pointer dereference: TNUM=0 PCYC=23860 PC=4a5 VADDR=0

WARNING: register_tlb_missrw_exception in arch/system.c:339:
         TLB miss-RW exception detected on tnum=0 PC:2400,
         badva=0x3ffffff8 access_type=S cycle=1866
```

Warning messages for NULL pointer dereferences are controlled by the `--nullptr` option (Section 3.3.5).

Warning messages for processor interrupts and exceptions are controlled by the `--verbose` option (Section 3.3.1).

For more information on processor interrupt and exception events, see the applicable *Qualcomm Hexagon Programmer's Reference Manual*.

3.6 Profile data files

Profile data files are binary files that contain information on how the target application executed (for example, call graphs and execution time per function). This information is not directly human-readable; it is displayed by inputting the profile data file to the Hexagon code coverage profiler or gprof profiler.

Table 3-1 Command option used for generating profile data files

Target profiler	Command option	Profile Data File	Option description
hexagon-coverage	`--fast-profile`	`gmon.t_x`	Section 3.3.12
hexagon-gprof	`--profile`	`gmon.t_x`	Section 3.3.12

In some cases the simulator may generate multiple profile data files for an application. For more information, see Section 3.3.12.

For more information on the code coverage profiler, see the *Qualcomm Hexagon Code Coverage Profiler User Guide* (80-N2040-20).

For more information on the gprof profiler, see the *Qualcomm Hexagon gprof Profiler User Guide* (80-N2040-29).

NOTE: Some profile data might be inaccurate due to data normalization.

3.7 Trace files

Trace files are text files that contain a detailed record of the execution events that occur during the execution of a target application.

The simulator can generate separate trace files for the following execution events:

- Program counter (PC) accesses
- Hardware thread stalls
- Memory accesses
- Bus accesses
- Micro-architecture events
- Data cache accesses
- Instruction cache accesses
- L2 cache accesses

The generation of each type of trace file is controlled by the corresponding trace option (Section 3.3.14 and Section 3.3.15).

A trace file contains a sequence of trace entries, with each entry occupying one or more lines in the file (depending on the trace type). An individual trace entry consists of a series of *`name = value`* pairs that display information specific to the trace type.

NOTE: Trace files can become indefinitely large for longer programs. They can be reduced in size by filtering the generated trace data (Section 3.3.16).

3.7.1 Program counter trace files

Program counter trace files contain the following information:

- A record of the program counter (PC) value at each cycle in the target application, including all stall cycles.
- The instruction packet executed at each PC value (for committed cycles only).
- The contents of all processor registers after each cycle.

File syntax

Trace entries for program counter trace files have the following (multi-line) syntax:

```
Tnum VA=addr PA=addr PCYC=val instruction/status
Thread registers:
register: value  ...
Global registers:
register: value ...
```

NOTE: The trace file syntax described here applies to V60 and later processors. The syntax used for earlier processor versions differs somewhat in field names and field ordering.

Options

T*num*

Hardware thread number (decimal digit).

VA=*addr*

Virtual address of instruction (hexadecimal value with leading `0x`).

PA=*addr*

Physical address of instruction (hexadecimal value with leading `0x`).

PCYC=*val*

Processor cycle (decimal value).

instruction/status

Instruction packet disassembly or status of thread.

*register***:** *value*

Register name/value pair (hexadecimal value with leading `0x`).

NOTE: A minimal program counter trace file can be generated using the simulator option `--pctrace_min` (Section 3.3.14). The minimal trace file lists only the hardware thread number, PC virtual/physical address, and cycle number.

3.7.2 Memory trace files

Memory trace files contain a record of the memory accesses performed during execution of the target application (including both instruction and data accesses).

File syntax

Trace entries for memory trace files have the following syntax:

```
TNUM=num:TYPE=(IF|DR|DW):PCYC=val:PC=addr:VA=addr
                        :PA=addr:WIDTH=val:DATA=val:val:
```

NOTE: The trace file syntax described here applies to Hexagon V60 and later processors. The syntax used for earlier processor versions differs somewhat in field names and field ordering.

Options

TNUM=*num*

Hardware thread number (decimal digit).

TYPE=*type*

Memory access type.

IF

Instruction fetch from memory.

DR

Data read from memory.

DW

Data write to memory.

PCYC=*val*

Processor cycle (decimal value).

PC=*addr*

Instruction address making request (hexadecimal value with no leading `0x`).

VA=*addr*

Virtual address accessed (hexadecimal value with no leading `0x`).

PA=*addr*

Physical address accessed (hexadecimal value with no leading `0x`).

WIDTH=*val*

Number of bytes read (possible values 1, 2, 4, 8, 12, 16).

DATA=*val*`:`*val*`:`

Actual data values read (two hexadecimal values with no leading `0x`).

NOTE: When `WIDTH=12` or `16`, only 8 data bytes are printed.

3.7.3 Bus trace files

Bus trace files contain a record of the buses accessed by the target application.

File syntax

Trace entries for bus trace files have the following syntax:

```
PCYC=val:Tnum:PC=addr:(BUSREQ|BUSRSP)
          :TYPE=(IFETCH|DREAD|DWRITE):ID=val:PA=addr
          :WIDTH=val[:DELAY=val]
```

NOTE: The trace file syntax described here applies to Hexagon V60 and later processors. The syntax used for earlier processor versions differs somewhat in field names and field ordering.

Options

PCYC=*val*

Processor cycle (decimal value).

T*num*

Hardware thread number (decimal digit).

PC=*addr*

Packet address making request (hexadecimal value with no leading `0x`).

BUSREQ

Bus access request.

BUSRSP

Bus returned data or responded to request.

TYPE=*type*

Memory access type.

IFETCH

Instruction fetch.

DREAD

Data read.

DWRITE

Data write.

ID=*val*

Bus request identifier.

PA=*addr*

Physical address accessed (hexadecimal value with no leading `0x`).

WIDTH=*val*

Number of bytes accessed (decimal value).

DELAY=*val*

Latency (in pcycles) between bus request and response (decimal value).

NOTE: A physical address can appear in a bus trace without appearing in the corresponding memory trace, if a cache line containing the address is prefetched but never used.

3.7.4 Micro-architecture trace files

Micro-architecture trace files contain a record of major micro-architecture events (cache misses, packets executed, stall cycles, bus requests, and so on).

File syntax

Trace entries for micro-architecture trace files have the following syntax:

```
PCYC=val:Tnum:PC=addr:(BUSREQ|BUSRSP)
          :TYPE=(IFETCH|DREAD|DWRITE):ID=val:PA=addr
          :WIDTH=val[:DELAY=val]

PCYC=val:Tnum:PC=addr:STALL:stalltype

PCYC=val:Tnum:PC=addr:COMMIT

PCYC=val:Tnum:PC=addr:instruction
```

NOTE: The trace file syntax described here applies to Hexagon V60 and later processors. The syntax used for earlier processor versions differs somewhat in field names and field ordering.

Options

PCYC=*val*

Processor cycle (decimal value).

T*num*

Hardware thread number (decimal digit).

PC=*addr*

Packet address making request (hexadecimal value with no leading `0x`).

BUSREQ

Bus access request.

BUSRSP

Bus returned data or responded to request.

TYPE=*type*

Memory access type.

IFETCH

Instruction fetch.

DREAD

Data read.

DWRITE

Data write.

ID=*val*

Bus request identifier.

PA=*addr*

Physical address accessed (hexadecimal value with no leading `0x`).

WIDTH=*val*

Number of bytes accessed (decimal value).

DELAY=*val*

Latency (in pcycles) between bus request and response (decimal value).

STALL=*stalltype*

Packet stall (unquoted string value).

COMMIT

Packet commit.

instruction

Instruction packet disassembly.

3.7.5 Instruction cache trace files

Instruction cache trace files contain a record of the instruction cache accesses during execution of the target application.

File syntax

Trace entries for instruction cache trace files have one of the following syntaxes:

```
PCYC=val:Tnum:PC=addr:ICACHE:(MISS|REPLACE)
                                :VA=addr:PA=addr
```

NOTE: The trace file syntax described here applies to Hexagon V60 and later processors. The syntax used for earlier processor versions differs somewhat in field names and field ordering.

Options

PCYC=*val*

Processor cycle (decimal value).

T*num*

Hardware thread number (decimal digit).

PC=*addr*

Packet address making request (hexadecimal value with no leading `0x`).

ICACHE

Instruction cache operation.

MISS

Instruction cache miss.

REPLACE

Instruction cache replace.

VA=*addr*

Virtual address accessed (hexadecimal value with no leading `0x`).

PA=*addr*

Physical address accessed (hexadecimal value with no leading `0x`).

3.7.6 Data cache trace files

Data cache trace files contain a record of the data cache accesses during execution of the target application.

File syntax

Trace entries for data cache trace files have one of the following syntaxes:

```
PCYC=val:Tnum:PC=addr:DCACHE:(MISS|HIT)
                           :TYPE=(DCFETCH|DREAD|DWRITE)
                           :VA=addr:PA=addr[:WAY=val]
```

NOTE: The trace file syntax described here applies to Hexagon V60 and later processors. The syntax used for earlier processor versions differs in field names and field ordering.

Options

PCYC=*val*

Processor cycle (decimal value).

T*num*

Hardware thread number (decimal digit).

PC=*addr*

Packet address making request (hexadecimal value with no leading `0x`).

DCACHE

Data cache operation.

MISS

Data cache miss.

HIT

Data cache hit.

TYPE=*type*

Memory access type.

DCFETCH

Data cache fetch.

DREAD

Data read.

DWRITE

Data write.

VA=*addr*

Virtual address accessed (hexadecimal value with no leading `0x`).

PA=*addr*

Physical address accessed (hexadecimal value with no leading `0x`).

WAY=*val*

Number of L1 cache ways (decimal value).

3.7.7 L2 cache trace files

L2 cache trace files contain a record of the L2 cache accesses during execution of the target application.

File syntax

Trace entries for L2 cache trace files have one of the following syntaxes:

```
PCYC=val:Tnum:PC=addr:L2CACHE:cacheop:VA=addr:PA=addr:
                                        INDEX=val[:WAY=val]
```

NOTE: The trace file syntax described here applies to Hexagon V60 and later processors. The syntax used for earlier processor versions differs in field names and field ordering.

Options

PCYC=*val*

Processor cycle (decimal value).

T*num*

Hardware thread number (decimal digit).

PC=*addr*

Packet address making request (hexadecimal value with no leading `0x`).

L2CACHE:*cacheop*

L2 cache operation.

MISS

Data cache miss.

HIT

Data cache hit.

LOAD

Memory load.

STORE

Memory store.

SCALAR

Access by processor core.

IPREFETCH

Access due to instruction prefetch.

REPLACE

Access due to instruction cache replace.

CLEAN

Instruction cache replace does not require memory write.

DIRTY

Instruction cache replace requires memory write.

VA=*addr*

Virtual address accessed (hexadecimal value with no leading `0x`).

PA=*addr*

Physical address accessed (hexadecimal value with no leading `0x`).

INDEX=*val*

L2 cache index (decimal value).

WAY=*val*

Number of L2 cache ways (decimal value).

3.8 PMU statistics files

The simulator collects PMU statistics while an application is running. When the application terminates, the collected statistics are written to a PMU statistics file.

PMU statistics files are text files that contain the same execution information that the Hexagon Performance Monitor Unit generates to support on-target performance tracking:

- Instruction scheduling details
- Bus access events
- Cache access events

NOTE: The statistics file name is determined by the `--pmu_statsfile` option (Section 3.3.13). The default name is `pmu_statsfile.txt`.

For more information on the symbols used in PMU statistics files to represent Hexagon processor execution events, see the applicable *Qualcomm Hexagon Programmer's Reference Manual*.

3.9 Packet statistics files

The simulator collects statistics on the instruction packets executed while an application is running. When the application terminates, the simulator writes the collected statistics to a *packet statistics file*.

Packet statistics files are text files that contain the following execution information for each instruction packet:

- Address – Packet start address (in virtual memory)
- `commits` – Packet execution count
- `stalls` – Packet stall counts (for each stall type)
- `bus` – Bus access counts (for each access type)

The packet statistics file is enabled by the `--packet_analyze` option (Section 3.3.13).

Packet statistics files are written in JSON format, which can be read by Python or Perl scripts (http://json.org/example). To postprocess the JSON file produced with the `--packet-analyze` option, see the *Hexagon Profiler User Guide* (80-N2040-10).

The following code is an example of a single instruction packet entry from a packet statistics file:

```
{ // beginning of file
        "0x000069bc": {  // instruction packet address
                "commits": 12,
                "stalls": {
                        "FE_NEWVALUE_MISPREDICT_CYCLES": 9,
                        "IU_FETCH_CROSS_CYCLES": 3,
                        "IU_HIT_CYCLES": 26,
                        "DCACHE_DEMAND_MISS_CYCLES": 157,
                        "TOTAL_STALLS": 195
                },
                "bus": {
                        "AXI_READ_REQUEST": 1
                }
        }, // end of first packet
    // subsequent packets are listed here
```

Appendix A defines the symbols used in packet statistics files to represent the stall and bus access event types.

NOTE: The packet entries in a packet statistics file are not sorted.

NOTE: The JSON format used in a packet statistics file will not change in the future. However, additional metadata might be added to the file header, and additional attributes might be added to individual packets.

4 Interfaces

This chapter describes the following interfaces provided with the simulator:

- Timer interface (Section 4.1) – Used by a target application to collect execution statistics on specific parts of its own code.
- Cache interface (Section 4.2) – Used by a target application to maintain the processor cache while it is being simulated.
- Application programming interface (API) (Section 4.3) – Used by client programs to control simulations without user input.
- API callbacks (Section 4.4) – Set up and used via API calls.

Both the Timer interface and Cache interface are DSP domain utility libraries. They are to be linked in when building the target ELF to be simulated. Their effects come from the simulation.

The API operates in the host domain to make the simulator more versatile by customization. Its effect is on the simulation host. If you do not plan to build a customized simulator, the Simulator System API is irrelevant.

4.1 Timer interface

The simulator supports a built-in timer to enable users to collect execution statistics on specific parts of the target application being simulated. The information is collected by inserting timer function calls directly into the application source code.

The timer interface provides the following information:

- Number of times the specified section of code was executed
- Total number of cycles executed in the code section
- Minimum, maximum, and average number of cycles executed during each pass through the code section
- Processor cycle count
- Hardware thread processor cycle counts

The timer interface functions are accessed by including the library header file, `hexagon_sim_timer.h`.

NOTE: To use the timer interface, an application must be linked with the library file, `libhexagon.a`. For more information, see the appropriate application build document.

4.1.1 Timer functions

The timer interface's timer functions are used to get the number of cycles simulated while executing specific sections of code in the target application. The following example shows how the timer functions are used:

```
#include "hexagon_sim_timer.h"
...
main()
{
  hexagon_sim_init_timer();
  ...
  hexagon_sim_start_timer();

  <timed user code>;

  hexagon_sim_end_timer();
  ...
  hexagon_sim_show_timer(stdout);
}
```

The timer has the following properties:

- Only one timer instance is supported for each application.
- Timing information is collected cumulatively across multiple calls to `hexagon_sim_start_timer()` and `hexagon_sim_end_timer()`. This enables the timer functions to appear in loop code and still collect useful timing information.

The timer can be used to collect execution statistics on multiple parts of an application (each delimited by `hexagon_sim_start_timer()` and `hexagon_sim_end_timer()`). However, because only a single timer instance is supported for each application, the following restrictions apply:

- The timed parts of the application must be non-overlapping (and thus non-nested).
- Each time the application transitions between the timed parts, `hexagon_sim_show_timer()` and `hexagon_sim_init_timer()` must be called to respectively capture the execution statistics from the previous part and reset the timer for the next part.

4.1.1.1 hexagon_sim_end_timer()

Stops the operation of the simulation timer.

Prototype

```
void hexagon_sim_end_timer();
```

Detailed description

The timer is assumed to have been previously started with `hexagon_sim_start_timer()`. If `hexagon_sim_end_timer()` is called without a matching call to `hexagon_sim_start_timer()`, the validity of the resulting timing information) is undefined.

Timing information is collected cumulatively across multiple calls to `hexagon_sim_start_timer()` and `hexagon_sim_end_timer()`. `hexagon_sim_show_timer()` is called to summarize and write the collected timing information.

For a programming example, see Section 4.1.1.

NOTE: `hexagon_sim_end_timer()` is independent of the simulation timer, and it can be called independently of the timer-related functions.

Returns

None.

4.1.1.2 hexagon_sim_init_timer()

Before calling the other timer functions, initializes (or resets) the simulation timer for subsequent timer operations.

Prototype

```
void hexagon_sim_init_timer();
```

Detailed description

The sample count and all cycle counts are set to zero. For a programming example, see Section 4.1.1.

Returns

None.

4.1.1.3 hexagon_sim_prof_off()

Disables timer data collection even if the timer is started (`hexagon_sim_show_timer()`).

Prototype

```
void hexagon_sim_prof_off();
```

Detailed description

To re-enable timer data collection, call `hexagon_sim_prof_on()`.

Returns

None.

4.1.1.4 hexagon_sim_prof_on()

Re-enables timer data collection after it was disabled using `hexagon_sim_prof_off()`.

Prototype

```
void hexagon_sim_prof_on();
```

Returns

None.

4.1.1.5 hexagon_sim_show_timer()

Summarizes the collected timer information and write it to the specified stream.

Prototype

```
void hexagon_sim_show_timer(FILE *file);
```

Parameters

`in`	*file*	Pointer to the file descriptor of an open stream to which timer information is written.

Detailed description

This function summarize information collected so far by the timer, and then it writes the corresponding execution statistics to the specified stream.

The summary includes the following information:

- Number of times the section of code was executed (samples)
- Total number of cycles executed in the code section
- Minimum, maximum, and average number of cycles executed during each pass through the code section

For example:

```
Samples=25
Total cycles=5691225
Max cycles=234223
Min cycles=211092
Avg cycles=227649
```

The code section is specified by calls to `hexagon_sim_start_timer()` and `hexagon_sim_end_timer()`.

For a programming example, see Section 4.1.1.

Returns

None.

4.1.1.6 hexagon_sim_start_timer()

Starts the operation of the simulation timer.

Prototype

```
void hexagon_sim_start_timer();
```

Detailed description

The timer is assumed to have been previously initialized with `hexagon_sim_init_timer()`.

Timing information is collected cumulatively across multiple calls to `hexagon_sim_start_timer()` and `hexagon_sim_end_timer()`.

`hexagon_sim_show_timer()` is called to summarize and write the collected timing information.

For a programming example, see Section 4.1.1.

Returns

None.

4.1.2 Cycle count function

Cycle counts are defined as the number of cycles executed by the target application since it first began executing.

The Hexagon processor has multiple hardware threads, each with a dedicated cycle count. The cycle counts are not synchronized: when the processor cycle count increments, each of the cycle counts may or may not increment, depending on whether the corresponding hardware thread is currently in a wait state.

NOTE: The cycle count function, while part of the timer interface, is independent of the timer and can be called independently of the timer-related functions.

Thread cycle counts are for hardware threads only – they have no relation to the software threads supported by the RTOS. For accurate cycle counts on RTOS applications, use the RTOS profiling services or the standalone tools hexagon-gprof and hexagon_profiler (both are part of the standard Hexagon tools release).

4.1.2.1 hexagon_sim_read_pcycles()

Returns the total processor cycle count, which indicates the number of processor cycles executed for all hardware threads on the Hexagon processor.

Prototype

```
unsigned long long hexagon_sim_read_pcycles();
```

Detailed description

The processor cycle count is reset to 0 after a processor reset.

The following example shows how `hexagon_sim_read_pcycles()` can be used to determine the processor cycle count for a specific section of the application code:

```
start_cycles = hexagon_sim_read_pcycles();
<profiled user code>;
pcycle_count = hexagon_sim_read_pcycles() – start_cycles;
```

Returns

Processor cycle count.

4.2 Cache interface

The simulator optionally models the Hexagon processor cache. In certain cases it is useful to manage the cache data before performing an Angel call during simulation.

A target application uses the cache interface to manage the cache data during simulation. The interface provides functions that wrap the processor cache maintenance instructions so a single function call can perform the cache operation on a range of addresses. The interface provides wrappers for the following cache maintenance instructions:

- `dccleana`
- `dccleaninva`
- `dcinva`

The cache interface functions are accessed by including the library header file, `hexagon_cache.h`. To use the cache interface, an application must be linked with the library file, `libhexagon.a`. For more information, see the appropriate application build document. For more information on the cache maintenance instructions, see the applicable *Qualcomm Hexagon Programmer's Reference Manual*.

4.2.1 Cache functions

Use the cache interface functions to perform various combinations of flush and invalidate operations on the specified cache data.

4.2.1.1 hexagon_buffer_clean()

Flushes dirty cache data in a specified address range.

Prototype

```
void hexagon_buffer_clean(const uint8_t *address,
                          size_t nbytes);
```

Parameters

`in`	*address*	Pointer to the starting address of the bytes to be flushed.
`in`	*nbytes*	Number of bytes to be flushed.

Detailed description

Issue a `dccleana` instruction for each cache line in the range specified by the input parameters. This instruction looks up the data cache at the address. If the address is in the cache and has dirty data, it flushes that data out to memory.

Returns

None.

4.2.1.2 hexagon_buffer_cleaninv()

Flushes dirty cache data and invalidates cache lines in a specified address range.

Prototype

```
void hexagon_buffer_cleaninv(const uint8_t *address,
                             size_t nbytes);
```

Parameters

in	*address*	Pointer to the starting address of the bytes to be flushed.
in	*nbytes*	Number of bytes to be flushed.

Detailed description

Issue a `dccleaninva` instruction for each cache line in the range specified by the input parameters. This instruction looks up the data cache at the address. If this address is in the cache and has dirty data, it flushes that data out to memory. The line is then invalidated, whether or not any dirty data was written.

Returns

None.

4.2.1.3 hexagon_buffer_inv()

Invalidates cache lines in a specified address range.

Prototype

```
void hexagon_buffer_inv(const uint8_t *address,
                        size_t nbytes);
```

Parameters

in	*address*	Pointer to the starting address of the bytes to be flushed.
in	*nbytes*	Number of bytes to be flushed.

Detailed description

Issue a `dcinva` instruction for each cache line in the range specified by the input parameters. This instruction looks up the data cache at the address. If this address is in the cache, it is invalidated.

Returns

None.

4.3 Simulator System API

The simulator supports an API that clients can use to enable client program control of a Hexagon simulation. The API is used to perform co-simulation of the Hexagon processor with external devices and with higher-level system simulations.

The simulator is designed as a library that can be dynamically loaded by a system simulator. It supports two simulation modes:

- Standalone simulation (as `hexagon-sim`)
- Simulation as part of a larger system simulation environment.

The simulator library exports functions that can be called by the system simulation environment. This API supports simulation configuration as well as simulation control.

The simulator library additionally supports the loading of co-simulation models and communication with them through a dedicated API.

4.3.1 Simulator components

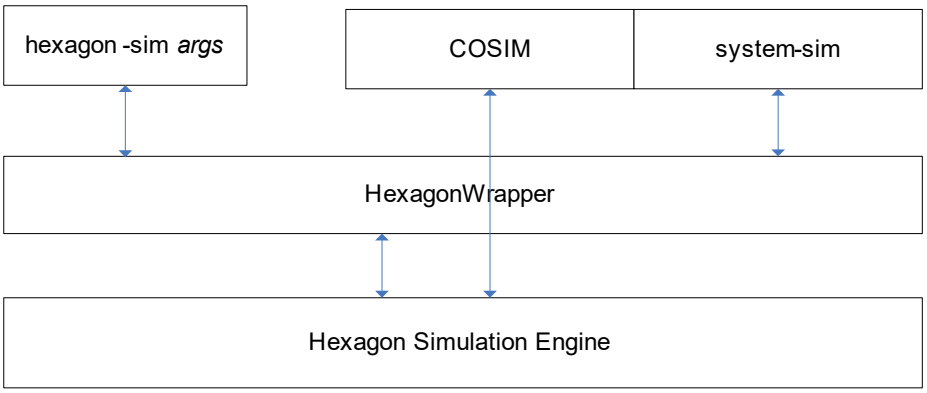

Figure 4-1 Simulator components used in standalone and system simulations

`hexagon-sim` is a command used to execute standalone simulations from a command line interface. The command arguments specify the simulation environment.

`system-sim` represents a host program that invokes a Hexagon simulation as part of a larger system simulation environment.

The Hexagon constructor (named `HexagonWrapper`) is used in both standalone and system simulations to create and configure instances of the Hexagon simulator.

The Hexagon simulation engine is invoked by the constructor to perform the simulation.

COSIM is a co-simulation module that models one or more devices external to the Hexagon processor. Co-simulations communicate with the simulation engine through the simulator API.

4.3.2 Invoke simulator in Standalone mode

In Standalone mode, the simulator is configured using command-line options.

```
hexagon-sim [options] executable
```

This executable is analogous to a system simulation environment where the target platform consists of only the Hexagon core. The `hexagon-sim` executable uses the simulator API to simulate the operation of the Hexagon core. An overview of the API functionality is provided below.

The command-line options are first parsed by `hexagon-sim` to determine the specific Hexagon processor version that is being requested to be simulated. This is determined by either the existence of an option specifically stating the version to use (such as `-mv62`) or, if not explicitly stated, the processor version is determined by examining the provided executable. In the latter case, the version is set to match the executable.

`hexagon-sim` first creates an instance of the `HexagonWrapper` class (Section 4.3.6) with the specified processor version, and then calls the configuration member functions (Section 4.3.8) based on the command-line arguments passed to it.

After configuration, the simulator either runs until the program terminates, or passes control to a debugger. In debug mode, the simulator can be run either in interactive mode or with a debugger like TRACE32 or GDB.

The following pseudo-code describes the functionality of `hexagon-sim`.

```
#include "HexagonTypes.h"
#include "HexagonWrapper.h"
main(int argc, char** argv)
{
  HEXAPI_Cpu main_arch_type;
  HexagonWrapper *hex;

  main_arch_type = findArch(argc, argv); // bail if invalid

  // Load architecture specific library based on command line
  // option and/or binary image. Put files in current dir
  hex = new HexagonWrapper(main_arch_type);

  // Call configuration methods based on command-line
  // options passed to the simulator

  ConfigureOptions(argc, argv);

if (InteractiveModeRequested)
  { start interactive mode }
  else
  { load image and run }
}
```

4.3.3 Invoke simulator in System Simulation mode

This section describes the API calls for working with the Hexagon simulator in a System Simulation mode.

Each processor version of the Hexagon core has its own simulator library (which is named either `libhexagonissv<x>.so` or `libhexagonissv<x>.dll`). This library is located at `$(hexagon_TOOLS_INSTALL_DIR)/Tools/lib/iss`.

The `HexagonWrapper` class (`libwrapper.a`) is statically linked to the system simulation application that wishes to instantiate a Hexagon core. This wrapper class knows where to find the library to load for the specified simulator by using one of the following methods:

- The loader's environment variable setting (`PATH` on Windows, `LD_LIBRARY_PATH` on Linux)
- Linking the application using a specific path to the linker (the `-rpath` option of `ld`)
- Relative to the path of the `hexagon-sim` executable (when using `hexagon-sim`)

The processor version type provided to the `HexagonWrapper` constructor determines which library is loaded.

For example, if the environment variable points to the library to load, the compile line would look like this:

```
g++ -o sys_sym sys_sym.c
  -I$(HEXAGON_TOOLS_INSTALL_DIR)/Tools/include/iss
  -L$(HEXAGON_TOOLS_INSTALL_DIR)/Tools/lib/iss -lwrapper
```

The environment variable is set to:

```
$(HEXAGON_TOOLS_INSTALL_DIR)/Tools/lib/iss
```

To dynamically bind the simulator library, the runtime link path can be hard-coded into the binary as follows:

```
g++ -o sys_sym sys_main.c
  -Wl,-rpath,$(HEXAGON_TOOLS_INSTALL_DIR)/Tools/lib/iss -lwrapper
```

When `hexagon-sim` is invoked from the command line, the library path is determined, relative to the path from where `hexagon-sim` was installed.

4.3.4 File handling

All simulator API functions that deal with paths or file names make the following assumptions:

- The user has the correct file permissions
- The entity specified by a name string has the same type as what is expected by the specification

Thus, if an API function requires a string whose content describes a path (directory), the referenced directory must exist and have the correct permissions. If the string describes a file, the API assumes it is in the correct format for the platform (`C:\...` or `/usr/...`) and it exists with the correct permissions, or, in the case of a simulator-created file, the directory in which the file is to be created exists and has permissions that allow the file to be created.

An enumerated type (`HEXAPI_OpenMode`) is optionally passed along with file names that directs the simulator to open the specific file in a way specified by the caller. This allows, for instance, concatenation of trace results from different portions of the simulation into a single file. Each API specifies what the simulator uses by default as the open mode.

NOTE: Opening files with `HEX_MODE_WRITE` or `HEX_MODE_WRITEBINARY` overwrites any existing data in the file.

The configuration APIs only check the validity of the specified files, but not the validity of any file content. A configure command might succeed, but it will return an error later in simulation. For instance, `ConfigureCosim()` accepts the name of a configuration file. The cosim configuration file contains names of cosim libraries to load. If one of the cosim library names is invalid, the `ConfigureCosim()` function succeeds, but an error is generated later.

The simulator maintains open file handles for each file specified and does not close files until the `HexagonWrapper` destructor is called (typically by deleting the instance). Reading or writing these files before the `HexagonWrapper` instance is deleted gives unpredictable results.

4.3.5 Status results

Most simulator API functions return a function result value indicating the result of the operation.

The API defines a standard set of symbols (of type `HEXAPI_Status`) for the result values, as shown in the following table.

Table 4-1 API status results

Symbol	Description
HEX_STAT_SUCCESS	Success
HEX_STAT_ERROR	Error
HEX_STAT_RANGE_ERROR	Incorrect range
HEX_STAT_INVALID_ARGS	Invalid arguments
HEX_STAT_CANNOT_CONFIG	Cannot configure simulator
HEX_STAT_CANNOT_TRANSLATE	No translation for specified `vaddr` exists
HEX_STAT_FILE_ACCESS_ERROR	File access error
HEX_STAT_MEM_ACCESS_ERROR	Cannot access memory
HEX_STAT_DEVICE_NOT_FOUND	Cannot find registered bus device
HEX_STAT_NO_ACTIVE_THREADS	No threads active
HEX_STAT_LOAD_ELF_ERROR	Error in loading ELF binary for Hexagon

NOTE: The `Run()` API function returns a function-specific result value of type `HEXAPI_CoreState`.

4.3.6 Simulator constructor (HexagonWrapper)

Creates an instance of the simulator for use as a handle for driving the Hexagon simulation. This handle can be further configured according to the target system configuration.

Prototype

```
HexagonWrapper(HEXAPI_Cpu cpu_ver);
```

Parameters

in	*cpu_ver*	Hexagon processor version. ■ HEX_CPU_V62 ■ HEX_CPU_V65 ■ HEX_CPU_V66 ■ HEX_CPU_V67 ■ HEX_CPU_V67T ■ HEX_CPU_V68

Detailed description

Tracing, profiling, or remote debugging features can be enabled by calling configuration functions on the handle.

A simulator instance handle can be used as a client by a system scheduler, and it can be made to run for a specified number of cycles. Cosims to model external devices can be constructed and plugged in as clients to the simulator library.

NOTE: Not all of the processor versions are supported in a specific Hexagon tools release. For more information, see Chapter 3.

Hexagon V55 and later processors have many `cpu_ver` values defined for their various micro-architectures. These values are not listed above. To determine them, use the `-help` option in `hexagon-sim` to list the `-mv<x>` processor version options defined in the simulator. Then construct the necessary `cpu_ver` value using the pattern `HEX_CPU_VXX_YY`, where `XX` indicates the processor version and micro-architecture, and `YY` indicates the sum of the L2 cache and TCM sizes. For example, `hexagon-sim -h -mv62` will show all revision IDs for V62 supported in this `hexagon-sim` instance. `Hexagon-sim` instances from different Hexagon tools release might show different revision IDs.

NOTE: Instead of `-help`, you can also use the `rev_id` values listed in `$(hexagon_TOOLS_INSTALL_DIR)/include/iss/HexagonTypes.h` as a source.

Usage

The following example shows how to use the constructor to create and configure an instance of the simulator for use as a handle for driving the Hexagon simulation.

```
#include "HexagonWrapper.h"   // defines enum HEXAPI_Cpu

sys_main()
{
  // instantiate HEXAGON wrapper
  HexagonWrapper *dsp = new HexagonWrapper(HEX_CPU_V62);
  HEX_4u_t actualCyc = 0;
  HEX_4u_t simRC;
  HEXAPI_CoreState runstate;

  // Configure DSP
  dsp->ConfigureTCM(0xD8000000);   // if you intend to use TCM
  dsp->ConfigureExecutableBinary("bootimg.pbn");
  // configure cosim here
  dsp->EndOfConfiguration();   // after all pre-run configuration

  // Run DSP


  // Most users intend to run straight through the entire program
  runstate = dsp->Run(&simRC);

  // if "runstate" is HEX_CORE_FINISHED
  // "simRC" will contain the return value of the user main()
  // of the simulated program. However, most usage scenarios
  // focus on side effects and ignore the return value.

  // To simulate a program in repeated short segments, do this:
  while (...)
  {
     // run for at most 100 pcycles at a time
     runstate = dsp->Step(100, &actualCyc, &simRC);

     // check various states and decide whether to exit loop
     ...
  }
}
```

NOTE: The function calls described in the following sections are public methods of the `HexagonWrapper` class.

4.3.7 Co-simulation

A co-simulation is a software module that models one or more devices external to the Hexagon processor. It executes in parallel with the Hexagon simulator, and it communicates with the simulator through the simulator API.

One use for co-simulations is the creation of specialized monitoring components.

NOTE: Co-simulations must be written as single-threaded modules. Although you can use a cosim as a multi-threaded module, QTI does not supported it because the simulator library and l2vic/qtimer cosim are not qualified for multi-thread operations.

4.3.7.1 Build co-simulations

Co-simulations are built as shared objects or dynamically-loaded libraries. They have the following build requirements.

Windows requirements

- Supports Microsoft Visual Studio 2010 or later
- Must specify DLL file, `libhexagonissv<x>.dll`
- Must specify `/dll` option

For example:

```
link.exe vcodec.o /dll libhexagonissv<x>.dll /libpath:c:\hexagon\
Tools\lib\iss /out:vcodec.dll;
```

Linux requirements

- Object files must be compiled with `-fPIC` option
- No need to specify file, `libhexagonissv<x>.so`
- Must use `-shared` option when linking

For example:

```
g++ -c -fPIC vcodec.cpp -o vcodec.o;
g++ -shared -o vcodec.so vcodec.o;
```

4.3.7.2 Execute co-simulations

Co-simulations are executed by being specified as arguments to the Hexagon simulator:

- The simulator command option `--cosim_file` is used to specify the co-simulations for a standalone simulation. (For more information, see Chapter 3.)
- The simulator API function, `ConfigureCosim()`, is used to specify the co-simulations for a system simulation.

`--cosim_file` and `ConfigureCosim()` both accept the pathname of a text file as an argument. The specified text file contains a list of pathnames that in turn specify the co-simulator library files.

NOTE: While it is possible to set up environment variables so the path is not specified, it is usually better for the cosim specification file to use the full path to identify the cosim module.

Arguments can be passed to a co-simulation by optionally specifying them after the corresponding co-simulator library file name.

For example:

```
/user/me/mycosim/path/cosim1.so arg1 arg2    - Linux
/user/me/mycosim/path/cosim2.so

C:\cosims\cosim1.dll arg1 arg2               - Windows
C:\cosims\cosim2.dll
```

4.3.8 Simulator configuration

The `HexagonWrapper` class member functions, which are described in this section, are used to configure the Hexagon simulator library. All the configuration functions have to be called before calling the runtime simulator functions or the simulator control functions.

NOTE: Simulator configuration applies to system simulations only – it does *not* apply to co-simulation modules (Section 4.3.7).

Unless otherwise noted, if a configuration function of a certain type is called multiple times, the last configuration overrides the previous setting for that configuration.

All file names referred to as parameters must be in the correct form for the platform expected to be run on. Essentially, the passed-in string must be able to be passed as-is to the `fopen()` system call.

4.3.8.1 ConfigureRemoteDebug()

Specifies TCP/IP socket that the simulator will write status information to during the simulation.

Prototype

```
HEXAPI_Status ConfigureRemoteDebug(HEX_4u_t portNum);
```

Parameters

`in`	*portNum*	Assign a TCP port number that the simulator listens on for remote debugging. The debugger (`hexagon-gdb` OR Trace32 with MCD) attaches to this port. (`portNum` > 1024)

Detailed description

The socket is specified as an integer value.

This function is analogous to the `--gdbserv` (`-G`) option in `hexagon-sim`.

NOTE: The debugger connection will be made only when `Run()` is called.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_INVALID_ARGS – Invalid port number.

4.3.8.2 ConfigureCosim()

Plugs one or more external cosims into the simulator.

Prototype

```
HEXAPI_Status ConfigureCosim(const char *pCosimConfigFile,
                             HEXAPI_OpenMode mode = HEX_MODE_READ);
```

Parameters

in	*pCosimConfigFile*	Pointer to the configuration file for the cosims. The configuration file contains path to cosim libraries, followed by (optional) arguments to the cosims. This string should contain the fully qualified name of the configuration file.
in	*mode*	Open mode for the file. ■ HEX_MODE_READ (default) – Reading (Text mode) ■ HEX_MODE_READBINARY – Reading a binary file (not recommended)

Detailed description

A cosim can register to get callbacks for memory reads/writes, bus accesses, PC execution events, and time-based events. Multiple cosims can be specified and are listed on separate lines of the configuration file.

This function is analogous to the `--cosim_file` option in `hexagon-sim`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_FILE_ACCESS_ERROR – File access error.

Example

Cosim configuration file content:

```
/local/Hexagon_tools_install/Tools/lib/iss/timer.so 32000 0xAB000000 2
/local/Hexagon_tools_install/Tools/lib/iss/timer.so 19200000 0xAB004000 3
/local/MM_cosim/ebi_bus.so 0x100000
```

4.3.8.3 ConfigureOSAwareness()

Plugs an OS Awareness module into the simulator.

Prototype

```
HEXAPI_Status ConfigureOSAwareness(
                              const char *pOsamConfigFile,
                              HEXAPI_OpenMode mode = HEX_MODE_READ);
```

Parameters

in	*pOsamConfigFile*	Pointer to the fully qualified name of the configuration file for the OS Awareness module. This file contains the complete path to the OS Awareness module shared object or DLL. The string should contain the fully qualified name of the configuration file.
in	*mode*	Open mode for the file. ■ HEX_MODE_READ (default) – Reading (Text mode) ■ HEX_MODE_READBINARY – Reading a binary file (not recommended)

Detailed description

The OS Awareness module can register callbacks for the following:

- The debugger, to obtain software thread state, mutex information, and so on
- PC execution events
- Virtual-to-physical translation for servicing Angel calls (semi-hosting)

This function is analogous to the `-rtos` option in `hexagon-sim`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_FILE_ACCESS_ERROR – File access error.

4.3.8.4 ConfigureExecutableBinary()

Specifies the binary file that is to be loaded into the Hexagon simulator for simulation.

Prototype

```
HEXAPI_Status ConfigureExecutableBinary(const char *pElfFile);
```

Parameters

`in`	*pElfFile*	Pointer to the fully qualified name of the binary Hexagon ELF file of the program to be simulated. This file is opened using `HEX_MODE_READBINARY` mode.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_FILE_ACCESS_ERROR – File access error.

4.3.8.5 ConfigureAppCommandLine()

Allows command line arguments to be passed to the simulated program.

Prototype

```
HEXAPI_Status ConfigureAppComandLine(int argc, char **argv);
```

Parameters

in	*argc*	Number of arguments in the command line.
in	*argv*	Double pointer to an array of strings.

Detailed description

The command line arguments are used to pass arguments to the simulated application.

When individual arguments contain whitespace separators, the argument vector must include the escape character, `"\""` , at the front and end of the string to preserve the interpretation of the argument. For example, `my_argv[3]="\"Hexagon Core\""`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_INVALID_ARGS – Invalid port number.

Example

```
hexagon-sim --timing jpeg.exe ---o pic.ppm -i pic.jpg
```

The command line to be passed to the simulated application:

```
jpeg.exe -o pic.ppm -i pic.jpg
```

4.3.8.6 ConfigureSimStdin()

Configures standard input of the simulated program.

Prototype

```
HEXAPI_Status ConfigureSimStdin(
                              const char *pStdin,
                              HEXAPI_OpenMode mode = HEX_MODE_READ);
```

Parameters

in	*pStdin*	Pointer to the fully qualified name of the file whose contents are to be used as standard input of the simulated program. The default value is the STDIN of the host platform.
in	*mode*	Open mode for the file ■ HEX_MODE_READ (default) – Reading (Text mode) ■ HEX_MODE_READBINARY – Reading a binary file

Detailed description

Any read operation by the simulated program that specifies `stdin` explicitly or implicitly gets its input from the file associated with the `pStdin` file.

Passing a `NULL` value in the `pStdin` file will revert to `stdin` of the calling process.

This function is analogous to the `--sim_in` option in `hexagon-sim`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_FILE_ACCESS_ERROR – File access error.

4.3.8.7 ConfigureSimStdout()

Configures standard output of the simulated program.

Prototype

```
HEXAPI_Status ConfigureSimStdout(
                              const char *pStdout,
                              HEXAPI_OpenMode mode = HEX_MODE_WRITE);
```

Parameters

in	*pStdout*	Pointer to the fully qualified name of the file whose contents are to be used as standard output of the simulated program. The default value is STDOUT of the host platform.
in	*mode*	Open mode for the file. ■ HEX_MODE_WRITE (default) – Create for writing; Text mode ■ HEX_MODE_WRITEBINARY – Create binary file for writing ■ HEX_MODE_APPEND – Append text ■ HEX_MODE_APPENDBINARY – Append binary data

Detailed description

Any write operation by the simulated program that specifies `stdout` explicitly or implicitly writes its output to the file associated with the `pStdout` file.

Passing a NULL value in the `pStdout` file will revert to `stdout` of the calling process.

This function is analogous to the `--sim_out` option in `hexagon-sim`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_FILE_ACCESS_ERROR – File access error.

4.3.8.8 ConfigureSimStderr()

Prototype

```
HEXAPI_Status ConfigureSimStderr(
                            const char *pStderr,
                            HEXAPI_OpenMode mode = HEX_MODE_WRITE);
```

Parameters

in	*pStderr*	Pointer to the fully qualified name of the file whose contents are to be used as standard error output of the simulated program. The default value is STDERR of the host platform.
in	*mode*	Open mode for the file ■ HEX_MODE_WRITE (default) – Create for writing; Text mode ■ HEX_MODE_WRITEBINARY – Create binary file for writing ■ HEX_MODE_APPEND – Append text ■ HEX_MODE_APPENDBINARY – Append binary data

Detailed description

Configures the standard error output of the simulated program. Any write operation by the simulated program that specifies `stderr` explicitly or implicitly writes its output to the file associated with the `pStderr` file.

Passing a NULL value in the `pStderr` file will revert to `stderr` of the calling process.

This function is analogous to the `--sim_err` option in `hexagon-sim`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_FILE_ACCESS_ERROR – File access error.

4.3.8.9 ConfigureCoreFrequency()

Specifies the simulated processor clock speed (in Hertz).

Prototype

```
HEXAPI_Status ConfigureCoreFrequency(HEX_8u_t ClkHz);
```

Parameters

`in`	*ClkHz*	Hexagon clock frequency in Hertz.

Detailed description

The clock speed is specified as an integer value. It determines the ratio of simulated to real time.

The default clock speed is platform-specific. The typical range is 450 MHz to 600 MHz.

This function is like the `--dsp_clock` option in `hexagon-sim`, except that the function argument is expressed in Hz while the option argument is expressed in MHz.

NOTE: If you intend to convert cycle counts to simulated execution times, QTI recommends explicitly specifying a clock speed and not depending on the default values.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_INVALID_ARGS – Frequency out of range.

4.3.8.10 ConfigureTimingMode()

Enables or disables timing mode.

Prototype

```
HEXAPI_Status ConfigureTimingMode(HEXAPI_TimingMode mode);
```

Parameters

`in`	*mode*	Timing mode. ■ HEX_NOTIMING – Disable timing mode (default) ■ HEX_TIMING – Enable timing mode with data-backed data caches ■ HEX_TIMING_NODBC – Enable timing mode without data-backed data caches

Detailed description

This function enables or disables cache and core pipeline stall modeling. Correct external bus behavior is guaranteed only when timing mode is enabled.

PMU statistics are generated correctly for all events only when timing mode is enabled. Otherwise, when timing mode is disabled, statistics will be correctly generated for some but not all of the events.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.

4.3.8.11 ConfigureBypassIdle()

Skips the execution of pcycles while the processor is idle because all hardware threads are in Wait or Off mode.

Prototype

```
HEXAPI_Status ConfigureBypassIdle(bool enable);
```

Parameters

`in`	*enable*	Profile mode. ■ TRUE – Enable idle modeling ■ FALSE – Disable idle modeling (default)

Detailed description

This function speeds up the simulation when the processor is idle for a significant amount of time.

If the number of skipped cycles is nonzero, the value returned by `GetSimulatedCycleCount()` includes the number of cycles spent while the processor is idle in this way.

This function is analogous to the `--bypass_idle` option in `hexagon-sim`.

By default, this function is disabled (FALSE) for all architectures. Setting it to TRUE can accelerate programs with the DSP frequently in Idle mode, but using it will slow down programs that rarely have the DSP in Idle mode.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.

4.3.8.12 ConfigureAHB()

Specifies the address decode range destined for the AHB bus interface.

Prototype

```
HEXAPI_Status ConfigureAHB(HEX_PA_t AHBStartAddr,
                           HEX_PA_t AHBEndAddr);
```

Parameters

in	*AHBStartAddr*	Starting address of the AHB decode range (inclusive).
in	*AHBEndAddr*	Ending address of the AHB decode range (inclusive).

Detailed description

When Timing mode is switched ON, bus transactions with addresses within the provided AHB range are routed to the AHB controller. In Standalone mode, this function causes all memory accesses in the specified range to be routed to an internal bus controller that completes the transaction with a specific delay. This delay can be overridden by the `--ahbbuspenalty` option in `hexagon-sim` or by specifying a timed delay using `ConfigureAHBBusPenalty()`.

Additionally, the Hexagon simulator models the bus unit queuing mechanism correctly for Hexagon V6x processors.

All memory transactions outside the given range are routed to an internal AXI bus controller, which also delays the completion of bus requests by the same amount given for the AHB delay.

Bus transactions can be intercepted by writing a cosim model and registering to receive callbacks for bus transactions to a specified memory range (see `AddBusAccessCallback()`). The bus transactions (which can be performed using any algorithm) are completed by calling the `BusTransactionFinished()` method of the `HexagonWrapper` class.

This function is analogous to the `--ahb` option in `hexagon-sim`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_RANGE_ERROR – Incorrect range.

4.3.8.13 ConfigureAXI2()

Specifies the address decode range destined for the AXI2 bus interface.

Prototype

```
HEXAPI_Status ConfigureAXI2(HEX_PA_t AXI2StartAddr,
                            HEX_PA_t AXI2EndAddr);
```

Parameters

in	*AXI2StartAddr*	Starting address of the AXI2 decode range (inclusive).
in	*AXI2EndAddr*	Ending address of the AXI2 decode range (inclusive).

Detailed description

When Timing mode is switched ON, bus transactions with addresses within the provided AXI2 range are routed to the AXI2 controller. In Standalone mode, this function causes all memory accesses in the specified range to be routed to an internal bus controller that completes the transaction with a specific delay. This delay can be overridden by the `--axi2buspenalty` option in `hexagon-sim` or by specifying a timed delay using `ConfigureBusPenalty()`.

Also, the Hexagon simulator models the bus unit queuing mechanism correctly for Hexagon V6x processors.

All memory transactions outside the given range are routed to an internal AXI bus controller, which also delays the completion of bus requests by the same amount given for the AXI2 delay.

Bus transactions can be intercepted by writing a cosim model and registering to receive callbacks for bus transactions to a specified memory range (see `AddBusAccessCallback()`). The bus transactions (which can be performed using any algorithm) are completed by calling the `BusTransactionFinished()` method of the `HexagonWrapper` class.

This function is analogous to the `--axi2` option in `hexagon-sim`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_RANGE_ERROR – Incorrect range.
- HEX_STAT_ERROR – Invalid processor version.

4.3.8.14 ConfigureBusRatio()

Specifies the AXI (main) bus clock rate as a non-fractional ratio of the processor clock.

Prototype

```
HEXAPI_Status ConfigureBusRatio(HEX_4u_t busRatio);
```

Parameters

`in`	*busRatio*	Ratio of the AXI (main) bus clock to the processor clock. Default value: 2

Detailed description

If the ratio is set to 1, a bus cycle is defined to equal a pcycle. If the ratio is set to 2, a bus cycle is defined to be twice as long as a pcycle (and so on). For example, if the processor runs at 600 MHz and the ratio is 3, the bus runs at 200 MHz.

This function is analogous to the `--busratio` option in `hexagon-sim`.

NOTE: To convert cycle counts to simulated execution times, QTI recommends explicitly specifying the AXI bus ratio and not depending on the default values.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ERROR – Processor does not support this function.

4.3.8.15 ConfigureAHBBusRatio()

Specifies the AHB bus clock rate as a non-fractional ratio of the processor clock.

Prototype

```
HEXAPI_Status ConfigureAHBBusRatio(HEX_4u_t busRatio);
```

Parameters

in	*busRatio*	Ratio of AHB bus clock to processor clock. Default value: 2

Detailed description

This function is analogous to the `--ahbbusratio` option in `hexagon-sim`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ERROR – Processor does not support this function.

4.3.8.16 ConfigureAXI2BusRatio()

Specifies the AXI2 bus clock rate as a non-fractional ratio of the processor clock.

Prototype

```
HEXAPI_Status ConfigureAXI2BusRatio(HEX_4u_t busRatio);
```

Parameters

in	*busRatio*	Ratio of the AXI2 bus clock to the processor clock. Default value: 2

Detailed description

This function is analogous to the `--axi2busratio` option in `hexagon-sim`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ERROR – Processor does not support this function.

4.3.8.17 ConfigureBusPenalty()

Configures the default latency for AXI (main) bus accesses across the entire memory range.

Prototype

```
HEXAPI_Status ConfigureBusPenalty(HEX_8u_t BusPenaltyInterval,
                                  HEXAPI_Interval units);
```

Parameters

in	*BusPenaltyInterval*	AXI (main) bus penalty in simulation real-time units.
in	*units*	Units that represent the BusPenaltyInterval parameter. ■ HEX_MILLISEC – Milliseconds ■ HEX_MICROSEC – Microseconds ■ HEX_NANOSEC – Nanoseconds ■ HEX_PICOSEC – Picoseconds ■ HEX_PCYCLE -- pcycles

Detailed description

This latency can be overridden by connecting a bus cosim for appropriate address ranges. The latency is determined by the core clock frequency and simulates real time as if the core were running at the specified frequency. There is no direct relation to real time (wall-clock time).

The granularity of the time increment is bounded by the current core frequency setting. The finest granularity available corresponds to the time taken by a single processor cycle. All time values are scaled according to the current processor clock frequency and are rescaled if the core frequency changes.

This function is like the `--buspenalty` option in `hexagon-sim`, except that the function accepts a time value while the option accepts a cycle count.

For Hexagon V5 and later processors, the bus is simulated at its own frequency, as determined by the `ConfigureCoreFrequency()` and `ConfigureBusRatio()` settings.

If units = HEX_PCYCLE, the busRatio and busPenalty parameters are ignored for the registered callback. Use this option with care.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_INVALID_ARGS – Incorrect parameters.

4.3.8.18 ConfigureAHBBusPenalty()

Configures the default latency for AHB bus accesses across the entire memory range.

Prototype

```
HEXAPI_Status ConfigureAHBBusPenalty(HEX_8u_t BusPenaltyInterval,
                                     HEXAPI_Interval units);
```

Parameters

in	*BusPenaltyInterval*	AHB bus penalty in simulation real-time units.
in	*units*	Units that represent the BusPenaltyInterval parameter. ■ HEX_MILLISEC – Milliseconds ■ HEX_MICROSEC – Microseconds ■ HEX_NANOSEC – Nanoseconds ■ HEX_PICOSEC – Picoseconds

Detailed description

This latency can be overridden by connecting a bus cosim for the appropriate address ranges. The latency is determined by the core clock frequency and simulates real time as if the core were running at the specified frequency. There is no direct relation to real time (wall-clock time).

The granularity of the time increment is bounded by the current core frequency setting. The finest granularity available corresponds to the time taken by a single processor cycle. All time values are scaled according to the current processor clock frequency and are rescaled if the core frequency changes.

This function is like the `--ahbbuspenalty` option in `hexagon-sim`, except that the function accepts a time value while the option accepts a cycle count.

For Hexagon V5 and later processors, the bus is simulated at its own frequency, as determined by the `ConfigureCoreFrequency()` and `ConfigureAHBBusRatio()` settings.

NOTE: To specify the bus penalty in terms of cycles, contact Hexagon technical support for assistance.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_INVALID_ARGS – Incorrect parameters.

4.3.8.19 ConfigureAXI2BusPenalty()

Configures the default latency for AXI2 bus accesses across the entire memory range.

Prototype

```
HEXAPI_Status ConfigureAXI2BusPenalty(HEX_8u_t BusPenaltyInterval,
                                      HEXAPI_Interval units);
```

Parameters

in	*BusPenaltyInterval*	AXI2 bus penalty in simulation real-time units.
in	*units*	Units that represent the BusPenaltyInterval parameter. ■ HEX_MILLISEC – Milliseconds ■ HEX_MICROSEC – Microseconds ■ HEX_NANOSEC – Nanoseconds ■ HEX_PICOSEC – Picoseconds

Detailed description

This latency can be overridden by connecting a bus cosim for the appropriate address ranges. The latency is determined by the core clock frequency and simulates real time as if the core were running at the specified frequency. There is no direct relation to real time (wall-clock time).

The granularity of the time increment is bounded by the current core frequency setting. The finest granularity available corresponds to the time taken by a single processor cycle. All time values are scaled according to the current processor clock frequency and are rescaled if the core frequency changes.

This function is like the `--axi2buspenalty` option in `hexagon-sim`, except that the function accepts a time value while the option accepts a cycle count.

For Hexagon V5 and later processors, the bus is simulated at its own frequency, as determined by the `ConfigureCoreFrequency()` and `ConfigureAXI2BusRatio()` settings.

NOTE: To specify the bus penalty in terms of cycles, contact Hexagon technical support for assistance.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_INVALID_ARGS – Incorrect parameters.

4.3.8.20 ConfigureTCM()

Specifies the start of TCM.

Prototype

```
HEXAPI_Status ConfigureTCM(HEX_PA_t TCMStartAddr);
```

Parameters

`in`	*TCMStartAddr*	Starting address of the TCM decode range.

Detailed description

The size of TCM is determined by the specific Hexagon processor version, micro-architecture, and L2 cache configuration. When Timing mode is switched ON, within the TCM range do not result in external bus transactions. The latency of TCM is accurately modeled for the given processor version.

This function is analogous to the `--tcm:lowaddr` options in `hexagon-sim`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_INVALID_ARGS – Invalid address. This error is issued when the start address plus the TCM size overflows the address space.

4.3.8.21 ConfigureSubsystemBase()

Specifies the subsystem base address value that is stored in the configuration table referenced by the Hexagon system-level register CFGBASE.

NOTE: This function is supported only for the Hexagon Tools 7.x and later releases.

Prototype

```
HEXAPI_Status ConfigureSubsystemBase(HEX_4u_t subStart);
```

Parameters

`in`	*subStart*	Address value of the CFGBASE subsystem base.

Detailed description

The base address value specifies bits [35:16] of a 36-bit physical memory address. Bits [15:0] are assumed to be 0.

This function is analogous to the `--subsystem_base` option in `hexagon-sim`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.

4.3.8.22 ConfigureL2tcmBase()

Specifies the L2 TCM base address value that is stored in the configuration table referenced by the Hexagon system-level register CFGBASE.

NOTE: This function is supported only for the Hexagon Tools 7.x and later releases.

Prototype

```
HEXAPI_Status ConfigureL2tcmBase(HEX_4u_t l2tcmStart);
```

Parameters

`in`	*l2tcmStart*	Address value of the CFGBASE L2 TCM base.

Detailed description

The base address value specifies bits [35:16] of a 36-bit physical memory address. Bits [15:0] are assumed to be 0.

This function is analogous to the `--l2tcm_base` option in `hexagon-sim`.

If this function is not called, the simulator assigns the L2 TCM base address a default value that is internally determined.

The simulator sets the value of the CFGBASE register itself to (`l2tcmStart + 0x18`), where `0x18` denotes `0x180000` in the physical address space.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.

4.3.8.23 ConfigureL2cfgBase()

Specifies the L2 configuration base address value that is stored in the configuration table referenced by the Hexagon system-level register CFGBASE.

NOTE: This function is supported only for the Hexagon Tools 7.x and later releases.

Prototype

```
HEXAPI_Status ConfigureL2cfgBase(HEX_4u_t l2cfgStart);
```

Parameters

`in`	*l2cfgStart*	Address value of the CFGBASE L2 configuration base.

Detailed description

The base address value specifies bits [35:16] of a 36-bit physical memory address. Bits [15:0] are assumed to be 0.

This function is analogous to the `--l2cfg_base` option in `hexagon-sim`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.

4.3.8.24 ConfigureEtmcfgBase()

Specifies the ETM base address value that is stored in the configuration table referenced by the Hexagon system-level register CFGBASE.

NOTE: This function is supported only for the Hexagon Tools 7.x and later releases.

Prototype

```
HEXAPI_Status ConfigureEtmcfgBase(HEX_4u_t etmStart);
```

Parameters

`in`	*etmStart*	Address value of the CFGBASE ETM base.

Detailed description

The base address value specifies bits [35:16] of a 36-bit physical memory address. Bits [15:0] are assumed to be 0.

This function is analogous to the `--etm_base` option in `hexagon-sim`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.

4.3.8.25 ConfigureMemFill()

Specifies a value to be used for all uninitialized memory.

Prototype

```
HEXAPI_Status ConfigureMemFill(HEX_1u_t value);
```

Parameters

`in`	*value*	One-byte value used to fill all of memory (that is, the initial value of uninitialized memory). Default: 0x1F

Detailed description

The simulator fills all of memory with the specified value before simulation starts.

This function is analogous to the `--memfill` option in `hexagon-sim`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.

4.3.8.26 ConfigureMemFillRandom()

Specifies a value to be used as a random number seed.

Prototype

```
HEXAPI_Status ConfigureMemFillRandom(HEX_4u_t seed);
```

Parameters

`in`	*seed*	Value used to seed the random number generator for memory fill operations. The simulator fills all of memory (that is, the initial value of uninitialized memory) with random numbers using the `rand()` function of the host platform.

Detailed description

The simulator uses the `rand()` host function to initialize values for all uninitialized memory. The behavior of this function depends on the host implementation of the random number generator.

This function is analogous to the `--memfill_rand` option in `hexagon-sim`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.

4.3.8.27 ConfigureNULLPointerBehavior()

Instructs the simulator on how NULL pointer dereferences should be handled.

Prototype

```
HEXAPI_Status ConfigureNULLPointerBehavior(HEXAPI_Nullptr behavior);
```

Parameters

`in`	*behavior*	Describes how the simulator should handle NULL pointer dereferences. ■ HEX_NULLPTR_IGNORE – No warnings ■ HEX_NULLPTR_WARN – Issue a warning (default) ■ HEX_NULLPTR_FATAL – Terminate simulation after a warning message

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.

4.3.8.28 ConfigureCoreDump()

Generates a binary core dump file when the target application is terminated after generating an exception.

Prototype

```
HEXAPI_Status ConfigureCoreDump(const char *filename);
```

Parameters

`in`	*filename*	Pointer to the fully qualified name of the core dump file generated by the simulator.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.

NOTE: This function returns `HEX_STAT_SUCCESS` even if the simulator is unable to subsequently create the core dump file.

4.3.8.29 ConfigureGProf()

Configures the data dump for the GNU profiler.

Prototype

```
HEXAPI_Status ConfigureGProf(const char *gprofPath = NULL);
```

Parameters

`in`	*gprofPath*	Pointer to the string providing the path in which the `gprof` files are to be created.

Detailed description

For details, see Chapter 3.

The profiling files are created in the specified directory. If a NULL directory is supplied, the files are created in the current working directory. The simulator overwrites existing files without warning.

This function is analogous to the `--profile` option in `hexagon-sim`.

NOTE: If this function is not called, gprof profiling is disabled. Calling this function either with or without a `gprofPath` argument will enable gprof profiling. Once enabled, there is no way to disable gprof profiling for this session.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_FILE_ACCESS_ERROR – Problem accessing the file.

4.3.8.30 ConfigureProfileMode()

Specifies whether gprof profiling starts from time zero.

Prototype

```
HEXAPI_Status ConfigureProfileMode(bool bTimeZero);
```

Parameters

`in`	*bTimeZero*	Profile mode. ■ TRUE – gprof profile data collection starts from time zero ■ FALSE – OS boot-up instructions not included in profile data

Detailed description

For RTOS applications, a start time of zero indicates that profile data collection begins immediately, and thus includes the OS bootup instructions.

If `bTimeZero` is set to TRUE, gprof profile data collection starts from time zero.

If `bTimeZero` is set to FALSE, gprof profile data collection does not start from time zero, and thus does not include the OS bootup instructions in the profile data.

This function is analogous to the `--profile_timezero` option in `hexagon-sim`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.

4.3.8.31 ConfigurePmuStatisticsFile()

Specifies the file that the collected PMU statistics will be written to.

Prototype

```
HEXAPI_Status ConfigurePmuStatisticsFile(
                            const char *pmuStatsFile,
                            HEXAPI_OpenMode mode = HEX_MODE_WRITE);
```

Parameters

in	*pmuStatsFile*	Pointer to the fully qualified name of the file to which Hexagon PMU statistics are to be written.
in	*mode*	Open mode for the file. ■ HEX_MODE_WRITE (default) – Create for writing; Text mode ■ HEX_MODE_WRITEBINARY – Create binary file for writing ■ HEX_MODE_APPEND – Append text ■ HEX_MODE_APPENDBINARY – Append binary data

Detailed description

For Hexagon V6x processors, the collected data is not written to the output file until the `HexagonWrapper` destructor is called.

NOTE: By default, PMU statistics are collected for the entire program. To limit the collection to specific parts of a program, use `ConfigurePCRangeFilter` and `ConfigureTimeRangeFilter`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_FILE_ACCESS_ERROR – Problem accessing the file.

4.3.8.32 ConfigurePacketAnalysis()

Configures the collection of instruction packet execution statistics, and enables the collection of statistics data.

NOTE: This function is supported only in Hexagon V6x processors. It returns HEX_STAT_ERROR if it is called with any other version.

Prototype

```
HEXAPI_Status ConfigurePacketAnalysis(
                            const char *filename,
                            HEXAPI_OpenMode mode = HEX_MODE_WRITE);
```

Parameters

`in`	*filename*	Pointer to the fully qualified name of the file to which Hexagon packet statistics are to be written.
`in`	*mode*	Open mode for the file. ■ HEX_MODE_WRITE (default) – Create for writing; Text mode ■ HEX_MODE_WRITEBINARY – Create binary file for writing ■ HEX_MODE_APPEND – Append text ■ HEX_MODE_APPENDBINARY – Append binary data

Detailed description

The collected data is not written to the output file until the `HexagonWrapper` destructor is called. This operation is performed in sync on the Hexagon processor and coprocessor.

The `mode` argument is optional – if omitted, the file open mode defaults to the text file format of the host platform.

This function is equivalent to the `--packet_analyze` option in `hexagon-sim`.

By default, statistics are collected for the entire program. To limit the data collection to specific parts of a program, use `ConfigurePCRangeFilter()` and `ConfigureTimeRangeFilter()`. Alternatively, the data collection can be controlled dynamically using `EnablePacketAnalysis()`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_INVALID_ARGS – Incorrect parameters.
- HEX_STAT_FILE_ACCESS_ERROR – Problem accessing the file.
- HEX_STAT_ERROR – Invalid processor version.

4.3.8.33 ConfigureInstHistogram()

Configures the generation of instruction histogram data and enables the collection of histogram data.

NOTE: This function is supported only in Hexagon V6x processors. It returns HEX_STAT_CANNOT_CONFIG if it is called with any other version.

Prototype

```
HEXAPI_Status ConfigureInstHistogram(
                          const char *filename,
                          HEXAPI_OpenMode mode = HEX_MODE_WRITE);
```

Parameters

`in`	*filename*	Pointer to the fully qualified name of the file to which Hexagon instruction histogram data is to be written.
`in`	*mode*	Open mode for the file. ■ HEX_MODE_WRITE (default) – Create for writing; Text mode ■ HEX_MODE_WRITEBINARY – Create binary file for writing ■ HEX_MODE_APPEND – Append text ■ HEX_MODE_APPENDBINARY – Append binary data

Detailed description

The `mode` argument is optional – if omitted, the file open mode defaults to the text file format of the host platform.

This function is equivalent to the `--ihist` option in `hexagon-sim`.

By default, histogram data is collected for the entire program. To limit the data collection to specific parts of a program, use `ConfigurePCRangeFilter()` and `ConfigureTimeRangeFilter()`. Alternatively, the data collection can be controlled dynamically using `EnableInstHistogram()`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_FILE_ACCESS_ERROR – Problem accessing the file.

4.3.8.34 ConfigurePCRangeFilter()

Enables starting and stopping of statistics collection and tracing based on instruction execution.

Prototype

```
HEXAPI_Status ConfigurePCRangeFilter(HEX_VA_t PCStart,
                                     HEX_VA_t PCEnd);
```

Parameters

in	*PCStart*	Address of an instruction that, when committed, starts the profiling data collection.
in	*PCEnd*	Address of an instruction that, when committed, ends the profiling data collection.

Detailed description

This function applies to all forms of statistics collection: gprof and normal execution statistics. This also applies to trace generation.

When the instruction at the `PCStart` address is committed, statistics collection is started. This means that statistics for the `PCStart` packet are not in the trace data. When the instruction at the `PCStart` address is committed, statistics collection terminates. This means that statistics the `PCStart` packet are in the trace data. Statistics collection will never be restarted, even if the instruction at `PCStart` is executed again.

This routine merely enables the collection of statistical data. To print the statistical information, use `EmitPerfStatistics()`.

This function works in conjunction with `ConfigureTimeRangeFilter()`. Statistics collection begins when either the `TimeStart` or `PCStart` conditions are satisfied and terminates whenever either the `TimeEnd` or `PCEnd` conditions are met.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.

4.3.8.35 ConfigureTimeRangeFilter()

Enables starting and stopping of statistics collection and tracing based on elapsed simulation time.

Prototype

```
HEXAPI_Status ConfigureTimeRangeFilter(HEX_8u_t TimeStart,
                                       HEXAPI_Interval StartUnits,
                                       HEX_8u_t TimeEnd,
                                       HEXAPI_Interval EndUnits);
```

Parameters

in	*TimeStart*	Time value that indicates when to start profiling data collection.
in	*StartUnits*	Units that represent the TimeStart parameter. ■ HEX_MILLISEC – Milliseconds ■ HEX_MICROSEC – Microseconds ■ HEX_NANOSEC – Nanoseconds ■ HEX_PICOSEC – Picoseconds
in	*TimeEnd*	Time value that indicates when to end profiling data collection.
in	*EndUnits*	Units that represent the TimeEnd parameter. ■ HEX_MILLISEC – Milliseconds ■ HEX_MICROSEC – Microseconds ■ HEX_NANOSEC – Nanoseconds ■ HEX_PICOSEC – Picoseconds

Detailed description

This function applies to all forms of statistics collection: gprof and normal execution statistics. This also applies to trace generation.

When `TimeStart` time has elapsed, statistics collection is started. When `TimeEnd` time has elapsed, statistics collection terminates. Statistics collection will never be restarted.

This routine merely enables the collection of statistical data. To print the statistical information, use `EmitPerfStatistics()`.

This function works in conjunction with `ConfigurePCRangeFilter()`. Statistics collection begins when either the `TimeStart` or `PCStart` conditions are satisfied and terminates whenever either the `TimeEnd` or `PCEnd` conditions are met.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.

4.3.8.36 EndOfConfiguration()

Signals the end of simulator configuration.

Prototype

```
HEXAPI_Status EndOfConfiguration();
```

Detailed description

Indicate to the simulator that the configuration functions will no longer be called.

Calling this function causes the simulator to evaluate configuration options and load cosims, OS Awareness modules, symbol files, and so on If a binary image was provided, it is loaded into Hexagon simulator memory.

NOTE: Any configuration function that is called after this function returns an error (`HEX_STAT_CANNOT_CONFIG`).

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Cannot configure the simulator. End of the configuration phase.
- HEX_STAT_ERROR – An error exists in the configuration (for example, executable image is not valid or is not readable).

4.3.8.37 SetTracing()

Enables or disables tracing at any time during execution.

Prototype

```
HEXAPI_Status SetTracing(HEXAPI_TracingType TracingType,
                         const char *pTraceOutputFile,
                         HEXAPI_OpenMode mode = HEX_MODE_WRITE);
```

Parameters

in	*TracingType*	■ HEX_TRACE_PC ■ HEX_TRACE_MEM ■ HEX_TRACE_BUS ■ HEX_TRACE_DCACHE ■ HEX_TRACE_ICACHE ■ HEX_TRACE_STALL ■ HEX_TRACE_L2CACHE ■ HEX_TRACE_PC_MIN – Minimal PC trace This type is a subset of HEX_TRACE_PC. ■ HEX_TRACE_PC_NANO – Smaller than minimal PC trace ■ HEX_TRACE_UARCH – Micro-architecture trace This type is the union of HEX_TRACE_ICACHE, HEX_TRACE_DCACHE, HEX_TRACE_L2CACHE, HEX_TRACE_BUS, and HEX_TRACE_STALL.
out	*pTraceOutputFile*	Pointer to the fully qualified name of the file to contain the respective trace's data.
in	*mode*	Open mode for the file. ■ HEX_MODE_WRITE (default) – Create file for writing text mode ■ HEX_MODE_WRITEBINARY – Create binary file for writing ■ HEX_MODE_APPEND – Append text ■ HEX_MODE_APPENDBINARY – Append binary data

Detailed description

The configuration of all trace types must occur before the call to `EndOfConfiguration()`.

Tracing can be enabled or disabled at any time during execution.

- To disable tracing, pass a NULL pointer as the `pTraceOutputFile` argument in a call for that type. If `pTraceOutputFile` is NULL, the mode is ignored.
- If the `pTraceOutputFile` argument is specified as non-NULL, a trace file is opened with the mode specified for the designated `TracingType`.

Example

```
HEX_TRACE_PC_MIN is a subset of HEX_TRACE_PC.
HEX_TRACE_UARCH is the union of HEX_TRACE_ICACHE, HEX_TRACE_DCACHE,
HEX_TRACE_L2CACHE, HEX_TRACE_BUS, and HEX_TRACE_STALL.
[tools/cosims]:
commit f6c7debe9cbeaa4b9814deb3ac06f57dfac12cbe Date: Mon Jan 26
09:36:36 2015 -0600
[tools/include]:
commit 0e39da28de130336781f1054e15ca29835e95b90 Date: Fri Jul 23
16:38:55 2010 +0000
```

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_INVALID_ARGS – Invalid tracing type.
- HEX_STAT_FILE_ACCESS_ERROR – File pointer error.

4.3.8.38 ConfigureMaxPcycle()

Controls how many pcycles to simulate.

Prototype

```
HEXAPI_Status ConfigureMaxPcycle(HEX_8u_t limit=0);
```

Parameters

in	*limit*	Number of pcycles to simulate.

Detailed description

This configuration call must occur before the call to `EndOfConfiguration()`. The default argument is zero, meaning there is no limit (default simulator behavior).

If the simulation is terminated because the cycle limit is reached, the hexagon-sim return status is HEX_CORE_CYCLE_LIMIT.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Invalid tracing type.
- HEX_CORE_CYCLE_LIMIT – File pointer error.

4.3.9 External device API

Device registration functions are used by cosims or the system simulation application to register its callback functions with the Hexagon simulator. All the registration functions can be called before or after the configuration phase is complete. Each of these device registration/de-registration functions can be called multiple times.

4.3.9.1 AddBusAccessCallback()

Registers a function to be called when the core issues a bus request for the memory within the specified range.

Prototype

```
typedef HEXAPI_TransactionStatus
     (*bus_transaction_request_callback)  (void *handle,
     HEX_PA_t address, HEX_4u_t lengthInBytes,
     HEX_1u_t *data, HEX_4u_t requestID,
     HEXAPI_BusAccessType type, HEX_4u_t threadNum,
     HEXAPI_BusBurstType burst);

HEXAPI_Status AddBusAccessCallback(
                           void *handle,
                           HEX_PA_t startAddr,
                           HEX_PA_t endAddr,
                           bus_transaction_request_callback brtc);
```

Parameters

in	*handle*	Pointer to the external bus model handle returned from `RegisterCosim()` or `RegisterCosimArgs()`.
in	*startAddr*	Lower bound (inclusive) of the address range to be registered for this device.
in	*endAddr*	Upper bound (inclusive) of the address range to be registered for this device.
in	*btrc*	Pointer to a function to be called when a bus request is made to the given address region. See `BusTransactionRequestCallback()`.

Detailed description

The registered function must have the given prototype and returns zero if the bus request can be accepted. If the bus request cannot be accepted, a non-zero value is returned and the core stalls the requesting thread and re-issues the request at a later time.

If the request is accepted, the core keeps the requesting thread stalled until it receives a `BusTransactionFinished()` call to indicate completion of the transaction.

The intent is to allow you to model a bus. Typically, the called function (`btrc`) registers an alarm callback (`AddTimedCallback()`) to receive notification from the simulator that the expected latency for the operation has expired. At that time, the bus model signals completion of the requested bus operation.

Overlapping address ranges that are registered by different external models are rejected.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_INVALID_ARGS – Bus address ranges overlap.

4.3.9.2 RemoveBusAccessCallback()

Removes callback notification for an external bus model from the simulator.

Prototype

```
HEXAPI_Status RemoveBusAccessCallback(void *handle);
```

Parameters

in	*handle*	Pointer to the external bus model handle returned from RegisterCosim() or RegisterCosimArgs().

Detailed description

The callbacks associated with the handle are removed for all address ranges for which a callback was registered.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_DEVICE_NOT_FOUND – Cannot find registered bus device.

4.3.9.3 AddFrequencyChangeCallback()

Used by an external device model that requires simulator notification every time the Hexagon core frequency changes. It can use the new frequency to calculate appropriate delays as required.

Prototype

```
typedef void (*frequency_change_callback)
     (void *handle, HEX_4u_t newFrequency);

HEXAPI_Status AddFrequencyChangeCallback(void *handle,
                                         frequency_change_callback fcc)
```

Parameters

in	*handle*	Pointer to the external bus model handle returned from RegisterCosim() or RegisterCosimArgs().
in	*fcc*	Pointer to a function to call for frequency change notification. See FrequencyChangeCallback().

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_DEVICE_NOT_FOUND – Cannot find registered bus device.

4.3.9.4 RemoveFrequencyChangeCallback()

Used by an external device model to remove itself from a list of devices that have registered to get a callback every time the core frequency changes.

Prototype

```
HEXAPI_Status RemoveFrequencyChangeCallback(void *handle);
```

Parameters

in	*handle*	Pointer to the external bus model handle returned from RegisterCosim() or RegisterCosimArgs().

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_DEVICE_NOT_FOUND – Cannot remove call to FrequencyChangeCallback() for the device.

4.3.9.5 AddTimedCallback()

Used by an external device model that requires notification from the simulator that interval simulator-time units have expired. This notification continues to be delivered by the simulator until canceled.

Prototype

```
typedef void (*timed_callback) (void *handle);

HEXAPI_Status AddTimedCallback(void *handle,
                               HEX_8u_t interval,
                               HEXAPI_Interval i_typ,
                               timed_callback tc);
```

Parameters

in	*handle*	Pointer to the external bus model handle returned from RegisterCosim() or RegisterCosimArgs().
in	*interval*	Interval in terms of simulator-time units.
in	*i_typ*	Units that represent the interval parameter. ■ HEX_MILLISEC – Milliseconds ■ HEX_MICROSEC – Microseconds ■ HEX_NANOSEC – Nanoseconds ■ HEX_PCYCLE – Processor cycles ■ HEX_PICOSEC – Picoseconds
in	*tc*	Pointer to a function to be called every interval time units. See TimedCallback().

Detailed description

The granularity of the time increment is bounded by the current core frequency setting. The finest granularity available corresponds to the time taken by a single processor cycle. All time values are scaled according to the current processor clock frequency and are rescaled if the core frequency changes.

The parameter handle can be anything. The addition and removal of a timedCallback is keyed on this handle. Have a unique handle for each instance of a new callback. If you add two callbacks using the same handle and then remove one callback, both callbacks are removed.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_DEVICE_NOT_FOUND – Cannot register the callback.

4.3.9.6 AddTimedCallbackFP()

Used by an external device model that requires notification from the simulator that a specified number of simulator-time units have expired. This notification continues to be delivered by the simulator until canceled.

NOTE: This function is identical to `AddTimedCallback()`, except that the time interval is expressed as a double-precision floating point value.

Prototype

```
typedef void (*timed_callback) (void *handle);

HEXAPI_Status AddTimedCallbackFP(void *handle,
                                 HEX_8f_t interval,
                                 HEXAPI_Interval i_typ,
                                 timed_callback tc);
```

Parameters

`in`	*handle*	Pointer to the external bus model handle returned from `RegisterCosim()` or `RegisterCosimArgs()`.
`in`	*interval*	Interval in terms of simulator-time units (expressed as double-precision floating point value).
`in`	*i_typ*	Units that represent the interval parameter. ■ HEX_MILLISEC – Milliseconds ■ HEX_MICROSEC – Microseconds ■ HEX_NANOSEC – Nanoseconds ■ HEX_PCYCLE – Processor cycles ■ HEX_PICOSEC – Picoseconds
`in`	*tc*	Pointer to a function to be called every interval time units. See `TimedCallback()`.

Detailed description

The granularity of the time increment is bounded by the current core frequency setting. The finest granularity available corresponds to the time taken by a single processor cycle. All time values are scaled according to the current processor clock frequency and are rescaled if the core frequency changes.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_DEVICE_NOT_FOUND – Cannot register the callback.

4.3.9.7 AddOneShotTimedCallback()

Used by an external device model that requires notification from the simulator that a specified number of simulator-time units have expired. This notification is delivered only once; for repeated notifications, use `AddTimedCallback()` instead.

Prototype

```
typedef void (*timed_callback) (void *handle);

HEXAPI_Status AddOneShotTimedCallback(void *handle,
                                      HEX_8u_t interval,
                                      HEXAPI_Interval i_typ,
                                      timed_callback tc);
```

Parameters

in	*handle*	Pointer to the external bus model handle returned from `RegisterCosim()` or `RegisterCosimArgs()`
in	*interval*	Interval in terms of simulator-time units.
in	*i_typ*	Units that represent the interval parameter. ■ HEX_MILLISEC – Milliseconds ■ HEX_MICROSEC – Microseconds ■ HEX_NANOSEC – Nanoseconds ■ HEX_PCYCLE – Processor cycles ■ HEX_PICOSEC – Picoseconds
in	*tc*	Pointer to a function to be called every interval time units. See `TimedCallback()`.

Detailed description

The granularity of the time increment is bounded by the current core frequency setting. The finest granularity available corresponds to the time taken by a single processor cycle. All time values are scaled according to the current processor clock frequency and are rescaled if the core frequency changes.

NOTE: `RemoveTimedCallback()` is not necessary for one-shot timed callbacks, except to stop the notification before it occurs.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_DEVICE_NOT_FOUND – Cannot register the callback.

4.3.9.8 RemoveTimedCallback()

Used by an external device model to remove itself from a list of devices that have registered to get a callback every interval period.

Prototype

```
HEXAPI_Status RemoveTimedCallback(void *handle);
```

Parameters

`in`	*handle*	Pointer to the external bus model handle returned from `RegisterCosim()` or `RegisterCosimArgs()`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_DEVICE_NOT_FOUND – Cannot remove `TimedCallback` for the device.

4.3.9.9 AddMemWasWrittenCallback()

Registers an external device that monitors memory write events.

Prototype

```
typedef void (*memory_written_callback)
      (void *handle, HEX_PA_t address,
       HEX_8u_t value, HEX_4u_t sizeInBytes);

HEXAPI_Status AddMemWasWrittenCallback(void *handle,
                                       HEX_PA_t startAddr,
                                       HEX_PA_t endAddr,
                                       memory_written_callback mwc);
```

Parameters

in	*handle*	Pointer to the external bus model handle returned from RegisterCosim() or RegisterCosimArgs().
in	*startAddr*	Lower bound of the address range registered for this device.
in	*endAddr*	Upper bound of the address range registered for this device. endAddr is inclusive in the range.
in	*mwc*	Pointer to a function to be called whenever the core writes to memory with an address in the specified range (inclusive). See MemoryWasWrittenCallback().

Detailed description

The mwc function is called by the simulator when a memory write is detected whose address is within the range of [*startAddr, endAddr*].

This function accepts overlapping ranges.

NOTE: mwc is called on *any* memory write event, even if the access results in a cache hit.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_DEVICE_NOT_FOUND – Cannot add memory write monitor callback.

4.3.9.10 RemoveMemWasWrittenCallback()

Removes notifications for an external device that tracks memory write events.

Prototype

```
HEXAPI_Status RemoveMemWasWrittenCallback(void *handle);
```

Parameters

in	*handle*	Pointer to the external bus model handle returned from RegisterCosim() or RegisterCosimArgs().

Detailed description

All memory ranges intercepted by the cosim associated with the handle will have their callback notifications removed.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_DEVICE_NOT_FOUND – Cannot remove memory write monitor.

4.3.9.11 AddMemWasReadCallback()

Registers an external device that monitors memory read events.

Prototype

```
typedef void (*memory_read_callback)
     (void *handle, HEX_PA_t address,
     HEX_8u_t value, HEX_4u_t sizeInBytes);

HEXAPI_Status AddMemWasReadCallback(void *handle,
                                    HEX_PA_t startAddr,
                                    HEX_PA_t endAddr,
                                    memory_read_callback mrc);
```

Parameters

in	*handle*	Pointer to the external bus model handle returned from RegisterCosim() or RegisterCosimArgs().
in	*startAddr*	Lower bound of the address range registered for this device.
in	*endAddr*	Upper bound of the address range registered for this device. endAddr is inclusive in the range.
in	*mrc*	Pointer to a function to be called whenever the core reads from memory with an address in the specified range (inclusive). See MemoryWasReadCallback().

Detailed description

The mrc function is called by the simulator when a memory read is detected whose address is within the range of [*startAddr, endAddr*].

This function accepts overlapping ranges.

NOTE: mwc is called on *any* memory read event, even if the access results in a cache hit.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_DEVICE_NOT_FOUND – Cannot configure the memory read monitor.

4.3.9.12 RemoveMemWasReadCallback()

Removes notifications for an external device that tracks memory read events.

Prototype

```
HEXAPI_Status RemoveMemWasReadCallback(void *handle);
```

Parameters

in	*handle*	Pointer to the external cosim handle returned from RegisterCosim() or RegisterCosimArgs().

Detailed description

All memory ranges intercepted by the cosim associated with the handle will have their callback notifications removed.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_DEVICE_NOT_FOUND – Cannot remove the memory read monitor.

4.3.9.13 AddPCCallback()

Registers a callback function that corresponds to the execution of an instruction indexed by a PC.

Prototype

```
typedef (void) (*pc_callback) (void *handle);

HEXAPI_Status AddPCCallback(void *handle,
                            HEX_VA_t vpc,
                            pc_callback pcc);
```

Parameters

in	*handle*	Pointer to the external cosim handle returned from RegisterCosim() or RegisterCosimArgs().
in	*vpc*	PC address in virtual space where the breakpoint is to be set.
in	*pcc*	Function pointer to the callback function that is called when vpc is reached. See PCCallback().

Detailed description

This callback function is invoked just before the instruction at PC is committed.

Only one callback function that corresponds to a VPC address by one cosim can be registered. The callback is triggered when the currently executing thread's PC register equals this value.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_DEVICE_NOT_FOUND – Cannot register the PC callback at vpc.

4.3.9.14 RemovePCCallback()

Removes a `PCCallback` function that corresponds to the execution of an instruction indexed by a PC.

Prototype

```
HEXAPI_Status RemovePCCallback(void* handle,
                               HEX_VA_t vpc);
```

Parameters

`in`	*handle*	Pointer to the external cosim handle returned from `RegisterCosim()` or `RegisterCosimArgs()`.
`in`	*vpc*	PC address in virtual space where the breakpoint is to be removed.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_DEVICE_NOT_FOUND – Cannot remove the PC callback at `vpc`.

4.3.9.15 AddBeforeSimulationStartsCallback()

Registers a callback function that will be called at the end of `EndOfConfiguration()`.

Prototype

```
typedef (void) (*cosim_callback)(void *handle);

HEXAPI_Status AddBeforeSimulationStartsCallback(void* handle,
                                                cosim_callback cc);
```

Parameters

`in`	*handle*	Pointer to the external cosim handle returned from `RegisterCosim()` or `RegisterCosimArgs()`.
`in`	*cc*	Pointer to a function to be called after configuration has completed but before the first cycle is simulated.

Detailed description

When this callback is made, the simulator has been fully configured, including instantiating all cosims and loading the executable image (if one was specified).

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ERROR – Cannot set the callback.

4.3.9.16 AddEndOfSimulationCallback()

Registers a callback function that will be called at the end of simulation.

Prototype

```
typedef (void) (*cosim_callback)(void *handle);

HEXAPI_Status AddEndOfSimulationCallback(void* handle,
                                         cosim_callback cc);
```

Parameters

in	*handle*	Pointer to the external cosim handle returned from RegisterCosim() or RegisterCosimArgs().
in	*cc*	Pointer to a function to be called after simulation has finished but before resources are deallocated.

Detailed description

When this callback is made, the simulation has terminated but no resources (cosims, memory, statistics, and so on) have been freed.

When this callback is made, the cosim is free to make any calls that do not attempt to advance the simulation. For instance, statistics can be printed, memory can be examined, and so on

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ERROR – Cannot set the callback.

4.3.9.17 AddCoreReadyCallback()

Used by an external device model that requires notification when the core's CORE_READY signal changes state.

Prototype

```
typedef void (*core_ready_callback)
     (void *handle, HEXAPI_CoreReadyState polarity);

HEXAPI_Status AddCoreReadyCallback(void *handle,
                                   core_ready_callback crc);
```

Parameters

in	*handle*	Pointer to the external bus model handle returned from RegisterCosim() or RegisterCosimArgs().
in	*crc*	Pointer to a function to call for core ready state change. See CoreReadyCallback().

Detailed description

This function is used primarily by cosims that are modeling second-level interrupt controllers. The state of this signal indicates whether the core is ready to accept an interrupt from the second-level controller.

The polarity parameter of the callback function can be either CORE_READY or CORE_NOT_READY.

NOTE: No RemoveCoreReadyCallback function is defined in the API because this callback is expected to remain active throughout the simulation.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ERROR – Resource allocation error or invalid processor version.

4.3.9.18 AddPrivilegeChangeCallback()

Used by an external device model that requires notification when the Hexagon processor's privilege mode changes.

Prototype

```
typedef void (*privilege_change_callback)
     (void *handle,
      HEX_4u_t hardware_thread_id,
      HEXAPI_PrivilegeMode mode);

HEXAPI_Status AddPrivilegeChangeCallback(
                                    void *handle,
                                    privilege_change_callback pcc);
```

Parameters

in	*handle*	Pointer to the external bus model handle returned from RegisterCosim() or RegisterCosimArgs().
in	*pcc*	Pointer to a function to call for privilege state change. See CoreReadyCallback().

Detailed description

The hardware_thread_id parameter of the callback function indicates the hardware thread associated with the privilege mode change.

The mode parameter of the callback function indicates the new privilege mode. It can be either MONITOR_MODE, USER_MODE, GUEST_MODE, or INVALID_MODE.

NOTE: Before this callback was added to the simulator system API, the caller was required to poll the simulator to determine when privilege changes occurred.

NOTE: No RemovePrivilegeChangeCallback function is defined in the API because this callback is expected to remain active throughout the simulation.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ERROR – Resource allocation error or invalid processor version.

4.3.9.19 AddQtimerCallback()

Registers an external device that models Qtimer (a standard system module external to the Hexagon processor).

NOTE: This function is supported only in Hexagon V61 and later processors. It returns HEX_STAT_ERROR if it is called with any other version.

Prototype

```
typedef HEX_8u_t (*qtimer_callback) (void *handle, int index);

HEXAPI_Status AddQtimerCallback(void *handle,
                                qtimer_callback qtc);
```

Parameters

`in`	*handle*	Pointer to the external bus model handle returned from `RegisterCosim()` or `RegisterCosimArgs()`.
`in`	*qtc*	Pointer to a function to call for the Qtimer event.

Detailed description

The `qtc` function is called by the simulator when the core needs to access Qtimer.

The registered function accepts an index value that specifies the Qtimer event type. Currently this index value will always be 0, which indicates that the core issued a Qtimer read request. In this case the registered function must return (as a function result value) a 56-bit unsigned integer value indicating Qtimer's current physical counter value.

NOTE: In future versions of the simulator, the index value might be set to other values to indicate that different types of Qtimer requests are being issued.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_DEVICE_NOT_FOUND – Cannot register the callback.
- HEX_STAT_ERROR – Invalid processor version.

4.3.9.20 GetAPIVersion()

Returns the current API version.

Prototype

```
void GetAPIVersion(HEX_4u_t *major,
                   HEX_4u_t *minor,
                   HEX_4u_t *build);
```

Parameters

out	*major*	Pointer to the major API revision number.
out	*minor*	Pointer to the minor API revision number.
out	*build*	Pointer to the build identifier.

Detailed description

This function can be used by either a system simulation environment or a co-simulation model to validate compatibility with the API. The major version number increments when incompatibilities are introduced in the API.

Returns

None.

4.3.9.21 PrintBuildTag()

Print the build tag information for the simulator library. The information is written to the standard output.

Prototype

```
void PrintBuildTag(void);
```

Detailed description

Build tag information is intended primarily for users who are running system simulations, and occasionally need detailed information on how the simulator library was built. For users running standalone simulations, build tag information is generally not useful.

This function is equivalent to the `--build_tag` option in `hexagon-sim`.

NOTE: For simulator API users: if you are not sure which library versions are loaded for your run-time instance, add this call and report to QTI field support for help on determining the exact versions.

Returns

None.

Example

Build tag information is formatted as follows in the standard output:

```
hexagon-sim --build_tag
[arch_v62]:
TAGS co-located: v62_20200227
  commit f8f0b09fd5fb6625dfa4681b46782095e3e0b05f
  Date:   Wed Jan 15 11:08:27 2020 -0600
[arch_v65]:
TAGS co-located: v65_20200917 v65_20201214
  commit 544bd4c15d696c584a940482d2391f48b996ad89
  Date:   Mon Aug 24 12:20:33 2020 -0500
[arch_v66]:
TAGS co-located: v66_20201214
  commit 89c837eeca7f855e12f570aaca0fa4dd0158250a
  Date:   Fri Oct 30 15:21:06 2020 -0500
[arch_v67]:
TAGS co-located: v67_20201214
  commit 9a9892fbe2cf8e185aba62c05bbdf697ee6a36aa
  Date:   Mon Nov 30 23:23:45 2020 -0800
[arch_v68]:
TAGS co-located: v68_20201214
  commit 43dcfbc5d76e378306a74b122d53a082e6a2f7e1
  Date:   Mon Dec 7 19:42:03 2020 -0800
[tools/simulator]:
BRANCH: (no
  commit 249d91cb8546ae512cbe69bb10f7855f3aed3596
  Date:   Fri Feb 26 09:19:35 2021 -0800
[tools/libs]:
```

```
  commit 5e2b0ddfd2679a94528d4e996ad26e16e3eb18ff
  Date:   Fri Feb 5 13:18:43 2021 -0600
[tools/cosims]:
  commit 7451e9c494f361b9abc874e0291da14abdeb068a
  Date:   Mon Mar 2 18:11:24 2020 -0600
[tools/ddrclade]:
  Clade Version: 01, 01, 47
  Clade2 Version: 01, 01, 62
Internal-use-only: S0 SD0
```

4.3.10 Runtime simulator calls

These simulator calls are made either before or after the configuration step. These are used for changing certain simulation options at runtime or to get information at runtime. These calls can be made multiple times.

4.3.10.1 EVB()

Specifies the value of the event vector base register. By default, the EVB is set to zero or to the entry point of the loaded binary file.

Prototype

```
void EVB(HEX_PA_t EVBStartAddr);
```

Parameters

`in`	*EVBStartAddr*	Starting address of the EVB.

Returns

None.

4.3.10.2 CoreFrequency()

Sets or gets the Hexagon frequency anytime during the simulation. Passing a value of zero leaves the current frequency unchanged and returns the current frequency.

Prototype

```
HEX_8u_t CoreFrequency(HEX_8u_t ClkHz);
```

Parameters

`in`	*ClkHz*	Hexagon clock frequency in Hertz. If 0, return the current clock frequency.

Returns

Hexagon clock frequency in Hz.

4.3.10.3 VerboseMode()

Set the level of verbosity. Verbose messages are sent to `stdout` in the simulator.

Prototype

```
void VerboseMode(HEXAPI_VerboseMode mode);
```

Parameters

`in`	*mode*	Verbosity level. ■ HEX_QUIET – No messages printed ■ HEX_NORMAL – Normal end-of-program statistics printed (default) ■ HEX_VERBOSE – Print high-level warnings and messages ■ HEX_REALLY_VERBOSE – Prints lots of informational messages

Returns

None.

4.3.10.4 AddSymbolFile()

Specifies the file from which symbol information is loaded.

Prototype

```
HEXAPI_Status AddSymbolFile(const char *pSymFile);
```

Parameters

`in`	*pSymFile*	Pointer to the fully qualified name of the ELF file that contains symbols.

Detailed description

Multiple symbol files can be provided by calling this function multiple times. If this function is called during the configuration phase, the file handle is added to a list of symbol file handles and the symbols are loaded at the end of the configuration phase.

NOTE: `pSymFile` is opened using `HEX_MODE_READBINARY` mode.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_FILE_ACCESS_ERROR – Problem accessing the file.

4.3.11 Simulator control

These simulator calls can be made only after the configuration step. These are used for runtime control and access to Hexagon core memory and registers. These calls can be made multiple times.

4.3.11.1 LoadExecutableBinary()

Loads or reloads an executable for simulation.

Prototype

```
HEXAPI_Status LoadExecutableBinary (const char *pElfFile = NULL);
```

Parameters

`in`	*pElfFile*	Pointer to the fully qualified name of the Hexagon ELF file executable program to load into memory for simulation. A NULL parameter causes the existing image to be reloaded.

Detailed description

Calling this function without a parameter (or with a NULL parameter value) causes the binary specified with `ConfigureExecutableBinary()` to be loaded or reloaded. This is typically done to restart simulating the current program. The file is opened using `HEX_MODE_READBINARY` mode.

Reset might be required for this function to act properly.

If no file name is provided by this function and `ConfigureExecutableBinary()` was not called, this function has no effect.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Configuration error or image is not loaded.
- HEX_STAT_FILE_ACCESS_ERROR – Cannot load image, or image is invalid for this processor version.

4.3.11.2 Run()

Runs the Hexagon core until it finishes executing (all threads stop) due to an unrecoverable error (such as a double exception) or because a breakpoint was hit.

Prototype

```
HEXAPI_CoreState Run(HEX_4u_t *result);
```

Parameters

`out`	*result*	Pointer to the return code from the simulated program's `exit()` call.

.Detailed description

The execution result is the exit code of the simulated program for normal termination (`HEX_CORE_FINISHED`), or undefined for the other return values.

If a valid call to `ConfigureRemoteDebug()` was made in the configuration phase, the simulator will wait for a connection from the debugger and control will be passed to the debugger. A `Run()` call returns `HEX_CORE_FINISHED` when the debugger exits.

Returns

- HEX_CORE_FINISHED – Program is finished.
- HEX_CORE_BREAKPOINT – Breakpoint is hit.
- HEX_CORE_RESET – Core is in Reset.
- HEX_CORE_ASYNCHRONOUS_BREAK – Asynchronous break.
- HEX_CORE_ERROR – Configuration error.

4.3.11.3 Step()

Steps the Hexagon core for one or more cycles.

Prototype

```
HEXAPI_CoreState Step(HEX_4u_t cycles,
                      HEX_4u_t *cyclesExecuted,
                      HEX_4u_t *result);
```

Parameters

`in`	*cycles*	Number of Hexagon processor cycles to execute.
`out`	*cyclesExecuted*	Pointer to the actual number of cycles executed.
`out`	*result*	Pointer to the return code from the simulated program's `exit()` call.

Detailed description

The actual number of cycles stepped is returned in `cyclesExecuted`. The result is the exit code of the simulated program on normal termination (`HEX_CORE_FINISHED`) and undefined for other return codes.

This function will not connect to a debugger, regardless of whether `ConfigureRemoteDebug()` was successfully called. To connect to a debugger, call `Run()` in conjunction with `AddTimedCallback()` to execute the program for a specific time period with the debugger connected.

Returns

- HEX_CORE_SUCCESS
- HEX_CORE_FINISHED – Program is finished.
- HEX_CORE_BREAKPOINT – Breakpoint is hit.
- HEX_CORE_RESET – Core is in Reset.
- HEX_CORE_ASYNCHRONOUS_BREAK – Asynchronous break.
- HEX_CORE_ERROR – Configuration error.

4.3.11.4 StepTime()

Steps the processor the number of cycles equivalent to the `time` input parameter.

Prototype

```
HEXAPI_CoreState StepTime(HEX_8u_t time,
                          HEXAPI_Interval units,
                          HEX_4u_t *cyclesExecuted,
                          HEX_4u_t *result);
```

Parameters

`in`	*time*	Number of time units to step.
`in`	*units*	Units that represent the time parameter. ■ HEX_MILLISEC – Milliseconds ■ HEX_MICROSEC – Microseconds ■ HEX_NANOSEC – Nanoseconds ■ HEX_PICOSEC – Picoseconds
`in`	*cycles*	Number of Hexagon cycles to execute.
`out`	*cyclesExecuted*	Pointer to the actual number of cycles executed.
`out`	*result*	Pointer to the return code from the simulated program's `exit()` call.

Detailed description

The actual number of cycles stepped is returned in `cyclesExecuted`. The result is the exit code of the simulated program on normal termination (`HEX_CORE_FINISHED`) and undefined for other return codes.

This function will not connect to a debugger, regardless of whether `ConfigureRemoteDebug()` was successfully called. To connect to a debugger, call `Run()` in conjunction with `AddTimedCallback()` to execute the program for a specific time period with the debugger connected.

Returns

- HEX_CORE_SUCCESS
- HEX_CORE_FINISHED – Program is finished.
- HEX_CORE_BREAKPOINT – Breakpoint is hit.
- HEX_CORE_RESET – Core is in Reset.
- HEX_CORE_ASYNCHRONOUS_BREAK – Asynchronous break.
- HEX_CORE_ERROR – Configuration error.

4.3.11.5 SetInterrupt()

Asserts an interrupt to the Hexagon core.

Prototype

```
HEXAPI_Status SetInterrupt(HEX_4u_t interruptNum,
                           HEXAPI_InterruptPinState state);
```

Parameters

`in`	*interruptNum*	Interrupt number for the Hexagon core. Range: 0 to 31
`in`	*state*	■ INT_PIN_DEASSERT – Deassert the interrupt `interruptNum` ■ INT_PIN_ASSERT – Assert the interrupt `interruptNum`

Detailed description

The interrupt is asserted by setting the appropriate bit in the interrupt pending register of the core to the appropriate state. The wrapper ensures that the interrupt is asserted or deasserted regardless of whether the interrupt is positive level- or edge-triggered or negative level- or edge-triggered.

If the core is to vector to the interrupt handler, interrupts must be enabled and the interrupt mask must be set correctly.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_INVALID_ARGS – Incorrect interrupt number.

4.3.11.6 AssertNMI

Asserts a non-maskable interrupt to the core. It remains asserted until it is deasserted.

Prototype

```
HEXAPI_Status AssertNMI();
```

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ERROR – NMI is already asserted, or this function was called during the configuration phase.

4.3.11.7 DeassertNMI()

Deasserts the NMI signal to the core. If no NMI is asserted, this function has no effect.

Prototype

```
HEXAPI_Status DeassertNMI();
```

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ERROR – NMI is already deasserted, or this function was called during the configuration phase.

4.3.11.8 ClearInterrupt

Clears a specific interrupt in the interrupt pending register of the Hexagon core.

Prototype

```
HEXAPI_Status ClearInterrupt(HEX_4u_t interruptNum);
```

Parameters

`in`	*interruptNum*	Interrupt number for the Hexagon core. Range: 0 to 31

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_INVALID_ARGS – Incorrect interrupt number.

4.3.11.9 ClearAllInterrupts()

Clears all interrupts in the interrupt pending register of the Hexagon core.

Prototype

```
void ClearAllInterrupts();
```

Returns

None.

4.3.11.10 SetBreakpoint

Sets a breakpoint at a virtual address.

Prototype

```
HEXAPI_Status SetBreakpoint(HEX_VA_t vpc);
```

Parameters

in	*vpc*	Virtual address where a breakpoint is to be set.

Detailed description

The list of breakpoints is maintained as an unbounded list within the simulator. Multiple calls to set a breakpoint on a certain address results in only one breakpoint being set. The simulator returns from a Run or Step command at the point just before the instruction at the specified address commits. The address passed to this function must be the address of the first instruction of a packet for the breakpoint to be recognized.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ERROR – Problem setting the breakpoint.

4.3.11.11 ClearBreakpoint()

Clears a breakpoint at the address specified by the provided virtual address. No action is taken if there is no breakpoint set at the given address.

Prototype

```
HEXAPI_Status ClearBreakpoint(HEX_VA_t vpc);
```

Parameters

in	*vpc*	Virtual address for which breakpoint is to be cleared

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ERROR – Problem clearing the breakpoint.

4.3.11.12 ClearAllBreakpoints()

Clears all breakpoints.

Prototype

```
HEXAPI_Status ClearAllBreakpoints();
```

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ERROR – Problem clearing the breakpoints.

4.3.11.13 AssertReset()

Puts the Hexagon core into Reset mode.

Prototype

```
HEXAPI_Status AssertReset();
```

Detailed description

When in reset mode, the core cannot step or run. The registers are cleared and so are all statistics and profile data counters. PC is set to EVB value.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ERROR – Core is already in reset, or this function was called during the configuration phase.

4.3.11.14 DeassertReset()

Takes the Hexagon core out of Reset mode.

Prototype

```
HEXAPI_Status DeassertReset();
```

Detailed description

The simulator vectors to the reset vector PC value (described by the EVB register). The EVB must be set at least once before the core is taken out of reset.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ERROR – Core is not in reset, or this function was called during the configuration phase.

4.3.11.15 BusTransactionFinished()

Used by an external bus cosim to indicate completion of a bus transaction.

Prototype

```
void BusTransactionFinished(HEX_1u_t * data,
                            HEX_4u_t size,
                            HEX_4u_t requestID);
```

Parameters

in	*data*	Pointer to the data returned by the external bus co-simulation.
in	*size*	Size of data in bytes.
in	*requestID*	ID of the bus transaction. This ID must correspond to the ID provided by a previous `BusTransactionRequestCallback()` call.

Detailed description

The `requestID` for this call corresponds to the ID passed to the bus transaction request callback function by the simulator. See `AddBusAccessCallback()` and `BusTransactionRequestCallback()`.

Returns

None.

4.3.11.16 WriteThreadRegister()

Writes a 32-bit value to a Hexagon thread register. `HEXAPI_TH_REG` is a thread register enumeration that is provided with `HexagonTypes.h`.

Prototype

```
HEXAPI_Status WriteThreadRegister(HEX_4u_t threadNum,
                                  HEXAPI_TH_REG index,
                                  HEX_4u_t value);
```

Parameters

`in`	*threadNum*	Selected hardware thread for the register to be set.
`in`	*index*	Selected thread register for the hardware thread.
`in`	*value*	Value to be written.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Function was called during the configuration phase.
- HEX_STAT_INVALID_ARGS – Invalid register number.

4.3.11.17 ReadThreadRegister()

Reads a 32-bit value from a Hexagon thread register. `HEXAPI_TH_REG` is a thread register enumeration that is provided with `HexagonTypes.h`.

Prototype

```
HEXAPI_Status ReadThreadRegister(HEX_4u_t threadNum,
                                 HEXAPI_TH_REG index,
                                 HEX_4u_t *value);
```

Parameters

`in`	*threadNum*	Selected hardware thread for the register to be read.
`in`	*index*	Selected thread register for the hardware thread.
`out`	*value*	Pointer to the value read from the register.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Function was called during the configuration phase.
- HEX_STAT_INVALID_ARGS – Invalid register number.

4.3.11.18 WriteVectorRegister()

Writes an unsigned 32-bit value to an HVX vector register.

NOTE: This function is supported only in Hexagon V6x processors. It returns HEX_STAT_ERROR if it is called with any other version.

Prototype

```
HEXAPI_Status WriteVectorRegister(char *regtype,
                                  HEX_4u_t tnum,
                                  int extno,
                                  HEX_4u_t reg_id,
                                  HEX_4u_t word_id,
                                  HEX_4u_t value);
```

Parameters

in	*regtype*	Pointer to the register type. ■ v – Access the V register (data) ■ q – Access the Q register (predicate)
in	*tnum*	Hardware thread ID.
in	*extno*	Register context. ■ 4,5,6,7 – HVX register contents ■ Other – Reserved
in	*reg_id*	■ V register – 0 to 31 ■ Q register – 0 to 3
in	*word_id*	■ V register – 0 to 15 ■ Q register – 0 to 1
in	*value*	Value written to the register.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Function was called during the configuration phase.
- HEX_STAT_INVALID_ARGS – Invalid register number.
- HEX_STAT_ERROR – Invalid processor architecture.

4.3.11.19 ReadVectorRegister()

Reads an unsigned 32-bit value from an HVX vector register.

NOTE: This function is supported only in Hexagon V6x processors. It returns HEX_STAT_ERROR if it is called with any other version.

Prototype

```
HEXAPI_Status ReadVectorRegister(char *regtype,
                                 HEX_4u_t tnum,
                                 int extno,
                                 HEX_4u_t reg_id,
                                 HEX_4u_t word_id,
                                 HEX_4u_t *value_addr);
```

Parameters

in	*regtype*	Pointer to the register type. ■ v – Access the V register (data) ■ q – Access the Q register (predicate)
in	*tnum*	Hardware thread ID.
in	*extno*	Register context. ■ 4,5,6,7 – HVX register contents ■ Other – reserved
in	*reg_id*	■ V register – 0 to 31 ■ Q register – 0 to 3
in	*word_id*	■ V register – 0 to 15 ■ Q register – 0 to 1
out	*value_addr*	Pointer to the value read from the register.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Function was called during the configuration phase.
- HEX_STAT_INVALID_ARGS – Invalid register number.
- HEX_STAT_ERROR – Invalid processor architecture.

4.3.11.20 WriteGlobalRegister()

Writes a 32-bit value to a Hexagon global register. `HEXAPI_G_REG` is a global register enumeration that is provided with `HexagonTypes.h`.

Prototype

```
HEXAPI_Status WriteGlobalRegister(HEXAPI_G_REG index,
                                  HEX_4u_t value);
```

Parameters

`in`	*index*	Selected global register.
`in`	*value*	Value to be written.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Function was called during the configuration phase.
- HEX_STAT_INVALID_ARGS – Invalid register number.

4.3.11.21 ReadGlobalRegister()

Reads a 32-bit value from a Hexagon global register. `HEXAPI_G_REG` is a global register enumeration that is provided with `HexagonTypes.h`.

Prototype

```
HEXAPI_Status ReadGlobalRegister(int index,
                                 HEX_8u_t *value);
```

Parameters

`in`	*index*	Selected global register.
`out`	*value*	Pointer to the value read from the register.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Function was called during the configuration phase.
- HEX_STAT_INVALID_ARGS – Invalid register number.

4.3.11.22 WriteTLBRegister()

Writes a 64-bit value to a Hexagon TLB register.

Prototype

```
HEXAPI_Status WriteTLBRegister(int index,
                               HEX_8u_t value);
```

Parameters

in	*index*	Selected TLB register.
in	*value*	Value to be written.

Detailed description

TLB register values are written to the low-level architecture library, which does not return any result status information. Thus this function returns only HEX_STAT_SUCCESS as a result value.

Returns

HEX_STAT_SUCCESS

4.3.11.23 ReadTLBRegister()

Reads a 64-bit value from a Hexagon TLB register.

Prototype

```
HEXAPI_Status ReadTLBRegister(int index,
                              HEX_8u_t *value);
```

Parameters

in	*index*	Selected TLB register.
out	*value*	Pointer to the value read from the register.

Detailed description

TLB register values are read from the low-level architecture library, which does not return any result status information. Thus, this function returns only HEX_STAT_SUCCESS as a result value.

Returns

HEX_STAT_SUCCESS

4.3.11.24 WriteMemory()

Writes `value` to Hexagon memory.

Prototype

```
HEXAPI_Status WriteMemory (HEX_PA_t paddr,
                           HEX_4u_t size,
                           HEX_8u_t value);
```

Parameters

`in`	*paddr*	Physical address where data is to be written.
`in`	*size*	Data size in bytes (1, 2, 4, or 8 bytes).
`in`	*value*	Value to be written to Hexagon memory.

Detailed description

This function is an incoherent write function in that the cache is not updated. The data value is written to the memory by potentially initiating a debug bus transaction.

If the bus transaction fails, this function returns `HEX_STAT_MEM_ACCESS_ERROR` and no action is taken.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Function was called during the configuration phase.
- HEX_STAT_INVALID_ARGS – Invalid size.
- HEX_STAT_MEM_ACCESS_ERROR – Cannot access memory.

4.3.11.25 ReadMemory()

Reads up to a 64-bit value from Hexagon memory.

Prototype

```
HEXAPI_Status ReadMemory(HEX_PA_t paddr,
                         HEX_4u_t size,
                         void *value);
```

Parameters

`in`	*paddr*	Physical address where data is read.
`in`	*size*	Data size in bytes (1, 2, 4, or 8 bytes).
`out`	*value*	Pointer to the value read from Hexagon memory.

Detailed description

This function bypasses the cache and reads directly from memory, which could result in a debug bus transaction. If the bus transaction fails, this function returns `HEX_STAT_MEM_ACCESS_ERROR` and no action is taken.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Function was called during the configuration phase.
- HEX_STAT_INVALID_ARGS – Invalid size.
- HEX_STAT_MEM_ACCESS_ERROR – Cannot access memory.

4.3.11.26 WriteVirtual()

Writes up to a 64-bit value at the virtual address specified by `vaddr`.

Prototype

```
HEXAPI_Status WriteVirtual(HEX_VA_t vaddr,
                           HEX_4u_t ASID,
                           HEX_4u_t size,
                           HEX_8u_t value);
```

Parameters

`in`	*vaddr*	Virtual address where data is to be written.
`in`	*ASID*	ASID to match when doing virtual-to-physical address translation. An ASID value of `0xFFFFFFFF` will match any ASID with a translation for the `vaddr`
`in`	*size*	Data size in bytes (1, 2, 4, or 8 bytes).
`in`	*value*	Value to be written.

Detailed description

This is a convenience function that performs virtual to physical address translation, writes the data into the cache (if present) and writes to memory if the data is not in cache.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Function was called during the configuration phase.
- HEX_STAT_INVALID_ARGS – Invalid size.
- HEX_STAT_CANNOT_TRANSLATE – Cannot translate `vaddr`.

4.3.11.27 ReadVirtual()

Reads up to a 64-bit value at the virtual address specified by `vaddr`.

Prototype

```
HEXAPI_Status ReadVirtual(HEX_VA_t vaddr,
                          HEX_4u_t ASID,
                          HEX_4u_t size,
                          void *value);
```

Parameters

`in`	*vaddr*	Virtual address from where data is to be read.
`in`	*ASID*	ASID to match when doing virtual-to-physical address translation. An ASID value of `0xFFFFFFFF` will match any ASID with a translation for the `vaddr`.
`in`	*size*	Data size in bytes (1, 2, 4, or 8 bytes).
`out`	*value*	Pointer to the value that was read.

Detailed description

This is a convenience function that performs virtual to physical address translation, probes the cache for the data and reads from memory if the data is not in cache.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Function was called during the configuration phase.
- HEX_STAT_INVALID_ARGS – Invalid size.
- HEX_STAT_CANNOT_TRANSLATE – Cannot translate `vaddr`.

4.3.11.28 ReadSymbolValue()

Reads the address or value that corresponds to a symbol.

Prototype

```
HEXAPI_Status ReadSymbolValue(const char *Symbol,
                              HEX_4u_t *value);
```

Parameters

`in`	*Symbol*	Pointer to the symbol to be read from the simulator.
`out`	*value*	Pointer to the value that corresponds to the symbol read from the symbol files or executable loaded in the simulator.

Detailed description

The simulator can provide this information if it can read symbol information from the binary or the symbol files.

NOTE: The string passed to this function is the `mangled` symbol name. The simulator does not perform any transformation on the symbol name passed in.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Function was called during the configuration phase.
- HEX_STAT_CANNOT_TRANSLATE – Symbol was not found.

4.3.11.29 TranslateVirtualToPhysical()

Translates a given virtual address (with ASID) into a physical address.

Prototype

```
HEXAPI_Status TranslateVirtualToPhysical(HEX_VA_t vaddr,
                                         HEX_4u_t ASID,
                                         HEX_PA_t *paddr);
```

Parameters

in	*vaddr*	Virtual address that requires translation.
in	*ASID*	ASID for which the `vaddr` is valid. Specifying `0xFFFFFFFF` for the ASID will match any ASID with a translation for the `vaddr`.
out	*paddr*	Pointer to the physical address that corresponds to the specified virtual address.

Detailed description

The translation algorithm looks in the RTOS structures for a valid translation. In the case of a standalone program (that is, no RTOS module is loaded), the TLB entries are searched for the translation.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_TRANSLATE – No translation for the specified `vaddr` exists.

4.3.11.30 CycleToTime()

Translates a given cycle count into units of time, taking into account the current core frequency.

Prototype

```
void CycleToTime(HEX_8u_t cycleCount,
                 HEX_8u_t *time,
                 HEXAPI_Interval *units);
```

Parameters

`in`	*cycleCount*	Number of cycles.
`out`	*time*	Pointer to the estimated amount of time in units returned that it would take the simulator to execute the number of cycles given.
`out`	*units*	Units that represent the time parameter. ■ HEX_MILLISEC – Milliseconds ■ HEX_MICROSEC – Microseconds ■ HEX_NANOSEC – Nanoseconds ■ HEX_PICOSEC – Picoseconds

Returns

None.

4.3.11.31 TimeToCycles()

Translates a time value to a specified number of processor cycles.

Prototype

```
void TimeToCycles(HEX_8u_t time,
                  HEX_8u_t *cycleCount,
                  HEXAPI_Interval units);
```

Parameters

`in`	*time*	Amount of time in units as specified by the `units` parameter.
`out`	*cycleCount*	Pointer to the estimated number of cycles that match the `time` parameter. **NOTE:** Due to rounding effects, the number can be off by ±1 cycle.
`in`	*units*	Units that represent the time parameter. ■ HEX_MILLISEC – Milliseconds ■ HEX_MICROSEC – Microseconds ■ HEX_NANOSEC – Nanoseconds ■ HEX_PICOSEC – Picoseconds

Returns

None.

4.3.11.32 GetElapsedSimulationTime()

Returns the elapsed time in the specified units since the start of simulation.

Prototype

```
HEXAPI_Status GetElapsedSimulationTime(HEX_8u_t *time,
                                       HEXAPI_Interval units);
```

Parameters

`out`	*time*	Pointer to the number of `units` that have elapsed since the start of the simulation.
`in`	*units*	Units that represent the time parameter. ■ HEX_MILLISEC – Milliseconds ■ HEX_MICROSEC – Microseconds ■ HEX_NANOSEC – Nanoseconds ■ HEX_PICOSEC – Picoseconds

Detailed description

The actual elapsed time is converted to the specified units, provided no underflow occurs.

NOTE: The time returned reflects all frequency change requests made by `CoreFrequency()`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ERROR – Time cannot be converted to the specified unit. This occurs if the actual elapsed time converted to the requested unit results in an elapsed time of zero.

4.3.11.33 GetSimulatedCycleCount()

Returns the number of simulated cycles since the start of simulation.

Prototype

```
HEXAPI_Status GetSimulatedCycleCount(HEX_8u_t *cycles);
```

Parameters

out	*cycles*	Pointer to the number of simulated cycles since the start of the simulation.

Detailed description

The cycles returned are the actual number of cycles simulated by the simulator and may or may not match the contents of the PCYCLE register.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ERROR – Function was called before EndOfConfiguration().

4.3.11.34 EmitPerfStatistics()

Prints the performance statistics into `buffer`.

Prototype

```
HEXAPI_Status EmitPerfStatistics(HEX_8u_t startTimeSec,
                                 HEX_8u_t startTimeUsec,
                                 HEX_8u_t endTimeSec,
                                 HEX_8u_t endTimeUsec,
                                 char *buffer,
                                 HEX_4u_t bufferSize);
```

Parameters

`in`	*startTimeSec*	Start time of the simulation run (in seconds).
`in`	*startTimeUsec*	Start time of the simulation run (in microseconds).
`in`	*endTimeSec*	End time of the simulation run (in seconds).
`in`	*endTimeUsec*	End time of the simulation run (in microseconds).
`out`	*buffer*	Pointer to the buffer from which the run statistics for performance will be printed.
`in`	*bufferSize*	Maximum size of the data to be copied into `buffer` (in bytes).

Detailed description

This routine puts a maximum of `bufferSize` bytes into the buffer. The input parameters are starting and ending time (real time) used to calculate the ratio to real time and simulator speed. The parameters are paired so that 1.1 seconds of real time is expressed as 1 second and 100000 microseconds. The output printed to the buffer is of the following form:

```
Done!
        T0: Insns=6761 Tcycles=3530
        T1: Insns=0 Tcycles=0
        T2: Insns=0 Tcycles=0
        T3: Insns=0 Tcycles=0
        T4: Insns=0 Tcycles=0
        T5: Insns=0 Tcycles=0
        Total: Insns=6761 Pcycles=21192
        Simulator speed=0.419547 Mips
        Ratio to Real Time (600 MHz) = ~1/456
            (elapsed time = 0.016115s)
```

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_INVALID_ARGS – Buffer pointer is NULL.
- HEX_STAT_ERROR – Function was called before `EndOfConfiguration()`.

4.3.11.35 EnablePacketAnalysis()

Enables or disables the generation of instruction packet execution statistics in the file specified in `ConfigurePacketAnalysis()`.

NOTE: This function is supported only in Hexagon V6x processors. It returns HEX_STAT_ERROR if it is called with any other version.

Prototype

```
HEXAPI_Status EnablePacketAnalysis(bool enable_disable);
```

Parameters

`in`	*enable_disable*	Analysis mode. ■ TRUE – Enable packet analysis ■ FALSE – Disable packet analysis

Detailed description

`ConfigurePacketAnalysis()` implicitly enables this function.

This operation is performed on the Hexagon processor and coprocessor concurrently.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_INVALID_ARGS – Incorrect parameter.
- HEX_STAT_ERROR – Invalid processor version.

4.3.11.36 ResetPacketAnalysis()

Resets internal data records used for packet analysis.

NOTE: This function is supported only for Hexagon V6x processors. It returns HEX_STAT_CANNOT_CONFIG if it is called with any other processor version.

Prototype

```
HEXAPI_Status ResetPacketAnalysis(void)
```

Detailed description

This function is not required during initialization. `ConfigurePacketAnalysis()` will generate the initial reset-to-zero.

This function can be called when the collected data is to be reset to zero.

To clear the internal data records for the next analysis, this function can be called immediately after calling `DumpPacketAnalysis()`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ERROR – Invalid processor version.

4.3.11.37 DumpPacketAnalysis()

Dumps available internal data records that were collected for packet analysis

NOTE: This function is supported only for Hexagon V6x processors. It returns HEX_STAT_CANNOT_CONFIG if it is called with any other processor version.

Prototype

```
HEXAPI_Status DumpPacketAnalysis(FILE *file)
```

Parameters

in	*file*	Pointer to the file descriptor of an open stream to which Hexagon packet statistics are to be written.

Detailed description

You must provide a valid FILE pointer and properly open and close the pointer. The FILE pointer can be any valid file name.

This function is not required to dump the collected packet analysis data at the end of a simulation. Properly deleting the HexagonWrapper object will generate the final dump.

This function can be called when the data is to be dumped before the simulator exits.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ FILE_ACCESS_ERROR – Bad file pointer.
- HEX_STAT_ CANNOT_CONFIG –Invalid processor version.

4.3.11.38 EnableInstHistogram()

Enables or disables the generation of instruction histogram data in a file specified in `ConfigureInstHistogram()`.

NOTE: This function is supported only in Hexagon V6x processors. It returns HEX_STAT_ERROR if it is called with any other version.

Prototype

```
HEXAPI_Status EnableInstHistogram(bool enable_disable);
```

Parameters

`in`	*enable_disable*	Instruction histogram mode. ■ TRUE – Enable generation of histogram data ■ FALSE – Disable generation of histogram data

Detailed description

`ConfigureInstHistogram()` implicitly enables this function.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_INVALID_ARGS – Incorrect parameter.
- HEX_STAT_ERROR – Invalid processor version.

4.3.11.39 ResetInstHistogram()

Resets internal data records used for an instruction histogram.

NOTE: This function is supported only for Hexagon V6x processors. It returns HEX_STAT_CANNOT_CONFIG if it is called with any other version.

Prototype

```
HEXAPI_Status ResetInstHistogram (void)
```

Detailed description

This function is not required during initialization. `ConfigureInstHistogram()` will generate the initial reset-to-zero.

This function can be called when the collected data is to be reset to zero.

To reset internal data records for the next analysis, this function can be called immediately after calling `DumpInstHistogram()`.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ CANNOT_CONFIG – Invalid processor version.

4.3.11.40 DumpInstHistogram()

Dumps internal data records collected for an instruction histogram.

NOTE: This function is supported only for Hexagon V6x processors. It returns HEX_STAT_CANNOT_CONFIG if it is called with any other version.

Prototype

```
HEXAPI_Status DumpInstHistogram (const char *msg)
```

Parameters

`in`	*msg*	Pointer to the message string (not to a file name).

Detailed description

You must provide a valid string as a message. The message string appears in the dump file followed by the dump contents.

The `msg` parameter can be NULL. You are responsible for dealing with this extra message line if `msg` is not NULL. The Hexagon Profiler cannot handle unexpected strings mixed into the data file.

The `msg` parameter is not a file name. There is no control over the filename used for an instruction histogram besides the original `ConfigureInstHistogram()` call.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ FILE_ACCESS_ERROR – Bad file pointer.
- HEX_STAT_ CANNOT_CONFIG – Invalid processor version.

4.3.11.41 GetPowerStatistics()

Returns a set of three performance measurement values that can be plugged into an equation that models the linear dynamic power for Hexagon processor cores.

Prototype

```
HEXAPI_Status  GetPowerStatistics(bool reset,
                                  float *all_wait,
                                  float *avg_active,
                                  float *avg_pkt);
```

Parameters

`in`	*reset*	Performance measurement values. ■ TRUE – Reset the values ■ FALSE – Return the current values
`out`	*all_wait*	Pointer to the percentage of time spent in All-Wait mode. Range: 0.0 to 1.0
`out`	*avg_active*	Pointer to the effective number of threads running (that is, the number of threads running and not stalled, averaged over time).
`out`	*avg_pkt*	Pointer to the average number of instructions per packet.

Detailed description

By default these measurement values reflect the performance of the simulation since it first began running. However, you can also reset the measurement values in order to limit the collection of performance data to specific parts of a simulation.

The `reset` parameter controls whether the function resets the measurement values (TRUE), or whether it returns their current values (FALSE).

NOTE: When `reset` is TRUE, the returned measurement values are invalid.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_INVALID_ARGS – One or more NULL input pointers; attempted to use with an invalid processor version.

Power calculation equation

Rather than provide a function that computes the power equation, the simulator system API provides only the function `GetPowerStatistics` to return certain parameters used in the power equation. This is done for two reasons:

- Some of the equation parameters are user-supplied.
- It enables users to modify the equation if necessary.

Equation parameters

The power equation includes three types of parameters:

- Parameters returned by `GetPowerStatistics`
- Parameters supplied by the user
- Parameters obtained from silicon measurements (or from pre-silicon estimates)

The user-supplied and silicon measurement parameters are statically defined for a given processor core, while the parameters returned by `GetPowerStatistics` are dynamic and generated during a simulation.

Table 4-2 lists the parameters used in the power equation.

Table 4-2 Power equation parameters

Type	Name	Detailed description
GetPowerStatistics	`allwaitFraction`	Percentage of time spent in All-Wait mode (`all_wait` in `GetPowerStatistics`)
	`N`	Effective number of threads running (that is, the number of threads running and not stalled, averaged over time (`avg_active` in `GetPowerStatistics`)
	`IPP`	Average number of instructions per packet (`avg_pkt` in `GetPowerStatistics`)
User-supplied	`frequencyHEXAGON`	Hexagon processor clock frequency normalized to 600 MHz
	`frequencyAXI`	AXI clock frequency, normalized to 133 MHz
	`v`	Core voltage normalized to the nominal voltage for the technology
	`nMax`	Maximum number of hardware threads
Silicon measurements (or pre-silicon estimates)	`Pallwait`	All-wait dynamic power for AXI slave logic at 133 MHz
	`P1`	Dhrystone power at maximum core frequency when a single thread is running
	`Pall`	Dhrystone power at maximum core frequency when all threads are running

Linear power model equation

The equation variables `m`, `b`, and `P` are defined in terms of the equation parameters listed in Table 4-2:

```
m = (Pall - P1) / (Nmax - 1)
b = P1 - m
P = m * N + b
```

The (V3) empirical dynamic power scaling factor `IPPadjustment` is based on the average number of instructions per packet:

```
IPPadjustment = 0.14 * (IPP - 2.2) + 1
```

The dynamic power (represented by `dynamicPower` in the following example) is based on the amount of time spent both running and in All-Wait mode:

```
runtimePower = (1 – allwaitFraction) * frequencyHEXAGON *
                                          IPPadjustment * P

allwaitPower = allwaitFraction * frequencyAXI * Pallwait

dynamicPower = (runtimePower + allwaitPower) * v2
```

4.3.11.42 EnablePmu()

Enables Hexagon PMU operation.

NOTE: This function is supported only in Hexagon V6x processors. It returns HEX_STAT_CANNOT_CONFIG if it is called with any other version.

Prototype

```
HEXAPI_Status EnablePmu(HEX_4u_t tnum = 0xffffffff);
```

Parameters

in	*tnum*	Reserved for future use.

Detailed description

The `tnum` parameter is reserved for future use and is not required in the function call:

```
EnablePmu();  // param not required – assigned default value
```

The PMU is automatically enabled before the simulation starts.

When the PMU is enabled, the simulator will collect PMU statistics. To begin collecting PMU statistics in the middle of a program, you can do two things:

- Call `DisablePmu()` at the beginning of the simulation, and then call `EnablePmu()` after hitting a breakpoint
- Call `ResetPmu()` after hitting a breakpoint (preferred method)

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Processor version does not support PMU.

4.3.11.43 DisablePmu()

Disables Hexagon PMU operation.

NOTE: This function is supported only in Hexagon V6x processors. It returns HEX_STAT_CANNOT_CONFIG if it is called with any other version.

Prototype

```
HEXAPI_Status DisablePmu(HEX_4u_t tnum = 0xffffffff);
```

Parameters

in	*tnum*	Reserved for future use.

Detailed description

The `tnum` parameter is reserved for future use and is not required in the function call:

```
DisablePmu();  // param not required – assigned default value
```

The PMU is automatically enabled before the simulation starts.

When the PMU is enabled, the simulator will collect PMU statistics. To begin collecting PMU statistics in the middle of a program, you can do two things:

- Call `DisablePmu()` at the beginning of the simulation, and then call `EnablePmu()` after hitting a breakpoint
- Call `ResetPmu()` after hitting a breakpoint (preferred method)

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Processor version does not support PMU.

4.3.11.44 ResetPmu()

Resets Hexagon PMU counts to zero.

NOTE: This function is supported only in Hexagon V6x processors. It returns HEX_STAT_CANNOT_CONFIG if it is called with any other version.

Prototype

```
HEXAPI_Status ResetPmu(HEX_4u_t tnum = 0xffffffff);
```

Parameters

in	*tnum*	Reserved for future use.

Detailed description

The `tnum` parameter is reserved for future use and is not required in the function call:

```
ResetPmu();  // param not required – assigned default value
```

The PMU is automatically enabled before the simulation starts.

When the PMU is enabled, the simulator will collect PMU statistics. To begin collecting PMU statistics in the middle of a program, you can do two things:

- Call `DisablePmu()` at the beginning of the simulation, and then call `EnablePmu()` after hitting a breakpoint
- Call `ResetPmu()` after hitting a breakpoint (preferred method)

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Processor version does not support PMU.

4.3.11.45 DumpPmu()

Prints PMU statistics to a specified file.

NOTE: This function is supported only in Hexagon V6x processors. It returns HEX_STAT_CANNOT_CONFIG if it is called with any other version.

Prototype

```
HEXAPI_Status DumpPmu(FILE *fp,
                      int including_iss_only_stats,
                      HEX_4u_t tnum = 0xffffffff);
```

Parameters

`in`	*fp*	File to print PMU statistics to.
`in`	*including_iss_only_stats*	Statistics to print. ■ 0 – Include PMU statistics supported by the hardware ■ 1 – Include PMU statistics supported by both the hardware and simulator
`in`	*tnum*	Reserved for future use.

Detailed description

The `tnum` parameter is reserved and is not required in the function call:

```
DumpPmu(f,1);  // tnum not required – assigned default value
```

This function does not print certain information that is included in the PMU statistics file generated by the standalone simulator (revision ID, cache size, full command line used to launch the simulator).

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Processor version does not support the specified PMU register.

4.3.11.46 GetPmuIndexedStats()

Reads a 64-bit value from the Hexagon PMU statistics register specified by the indicated hardware thread and PMU register index.

Prototype

```
HEXAPI_Status GetPmuIndexedStats(HEX_4u_t tnum,
                                 HEX_4u_t pe_num,
                                 HEX_8u_t *value);
```

Parameters

`in`	*tnum*	Hardware thread of the target PMU register.
`in`	*pe_num*	Index value of the target PMU register. Range: 0 to 255
`out`	*value*	Pointer to the value read from the PMU register.

Detailed description

Before calling `GetPmuIndexedStats()` to access a PMU register, you must first call `PmuIsStatModeled()` to verify that the register is modeled for the current processor version.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Processor version does not support the specified PMU register.

4.3.11.47 PmuIsStatModeled()

Reads a 32-bit value that indicates the model status of the Hexagon PMU statistics register specified by the indicated hardware thread and PMU register index.

Prototype

```
HEXAPI_Status PmuIsStatModeled(HEX_4u_t tnum,
                               HEX_4u_t pe_num,
                               HEX_4u_t *value);
```

Parameters

in	*tnum*	Hardware thread of the target PMU register.
in	*pe_num*	Index value of the target PMU register. Range: 0 to 255
out	*value*	Pointer to the model status. 0 – PMU register is not modeled 1 – PMU register is modeled

Detailed description

The model status value indicates whether the PMU register is modeled on the current processor version.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Processor version does not support the specified PMU register.

4.3.11.48 PmuGetName()

Returns a pointer to a character string containing the symbolic name of the Hexagon PMU statistics register specified by the indicated hardware thread and PMU register index.

Prototype

```
HEXAPI_Status PmuGetName(HEX_4u_t tnum,
                         HEX_4u_t pe_num,
                         const char **name);
```

Parameters

`in`	*tnum*	Hardware thread of the target PMU register.
`in`	*pe_num*	Index value of the target PMU register. Range: 0 to 255
`out`	*name*	Pointer to the character string containing the symbolic name of the specified PMU register.

Detailed description

Before calling `PmuGetName()` to access a PMU register, you must first call `PmuIsStatModeled()` to verify that the register is modeled for the current processor version.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Processor version does not support the specified PMU register.

4.3.11.49 PmuIsMaskable()

Returns an integer value that indicates whether the specified PMU event is local (per thread) or global (across all threads).

Prototype

```
HEXAPI_Status PmuIsMaskable(HEX_4u_t tnum,
                            HEX_4u_t pe_num,
                            HEX_4u_t *value);
```

Parameters

`in`	*tnum*	Hardware thread of the target PMU register.
`in`	*pe_num*	Index value of the target PMU register. Range: 0 to 255
`out`	*value*	Pointer to the value. ■ 0 – PMU event is non-maskable (global) ■ 1 – PMU event is maskable (local)

Detailed description

Before calling `PmuIsMaskable()` to determine the status of a PMU event, you must first call `PmuIsStatModeled()` to verify that the corresponding register is modeled for the current processor version.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_CANNOT_CONFIG – Processor version does not support the specified PMU register.

4.3.11.50 AxiSlaveAccess()

Performs a read or write operation on the AXI slave port.

Prototype

```
typedef void (*axi_bus_finished_callback)
      (void *,           /* actually "system_t *" */
      int threadno,      /* thread number  */
      HEX_PA_t paddr,    /* physical addr  */
      HEX_4u_t width,    /* size of access */
      int type,          /* type of access */
      int id,            /* identifier for access */
      unsigned char *dataptr);

HEXAPI_Status AxiSlaveAccess(HEX_4u_t tnum,
                             HEX_PA_t pAddr,
                             HEX_4u_t width,
                             HEX_4u_t type,
                             axi_bus_finished_callback cb,
                             HEX_4u_t id,
                             unsigned char *dataptr,
                             HEX_4u_t *rval);
```

Parameters

in	*tnum*	Hardware thread.
in	*pAddr*	Address to access.
in	*width*	Size of access (in bytes).
in	*type*	Type of access (read or write).
in	*cb*	Callback function (called when transaction complete).
in	*id*	AXI transaction identifier.
in	*dataptr*	Pointer to the data pointer (source for write).
out	*rval*	Pointer to the value. ■ 0 – Transaction is rejected ■ 1 – Transaction is accepted

Detailed description

If the `cb` parameter is specified as NULL, the operation is performed using a non-blocking call, and the data is transferred immediately.

If `cb` is non-NULL, the operation is performed using a blocking call. In this case *no* data is transferred during the call. Instead, the callback function referenced by `cb` is responsible for calling `WriteMemory()` or `ReadMemory()` to initiate the actual data transfer.

For the transaction to complete, the return value must be `HEX_STAT_SUCCESS` and `*rval` must be equal to 1.

The `dataptr` parameter is used only for write operations. For reads, `ReadMemory()` must be called when `cb` is called.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ERROR – Invalid processor version.

4.3.11.51 ReconnectMode()

Specifies the behavior of the simulator when the associated debugger front-end exits.

Prototype

```
void ReconnectMode(int mode);
```

Parameters

`in`	*mode*	Reconnect mode. ■ 1 – Simulator continues when the debugger front end exits ■ 0 – Simulator exits when the debugger front end exits Default: 0

Detailed description

If the simulator is configured to continue when the debugger front end exits, the simulator continues to execute in its current state (on hold, running a simulation, and so on) while also monitoring the socket connection. After a fixed period of time (such as 120 seconds) to allow TCP to wait out an obsoleted socket, the simulator reconnects if another debugger front end attempts to connect to it.

Returns

None.

4.3.11.52 PostMessageToSimulator()

Posts a message to the simulator.

Prototype

```
HEXAPI_Status PostMessageToSimulator(HEXAPI_RxMsgType msg,
                                     void * param_addr);
```

Parameters

in	*msg*	Message type. HEX_RXMSG_EXIT_EARLY – Stop the Hexagon core by triggering an asynchronous break (Run()). No parameter is defined for this message. **NOTE:** No other message types are currently defined.
in	*param_addr*	Pointer to the message parameter. **NOTE:** If the specified message type does not define a message parameter, this argument is ignored.

Detailed description

Depending on the message type, an optional data parameter can be specified as part of the message.

The HEX_RXMSG_EXIT_EARLY message type can be used with callbacks such as AddTimedCallback() to assert control over the simulator while it is running.

NOTE: The post operation is performed using a blocking call – it will not return until the message is posted to the simulator. Note, however, that the simulator might not immediately act on the message itself.

Returns

- HEX_STAT_SUCCESS
- HEX_STAT_ERROR – Message type is not recognized, or the message parameter cannot be decoded.

4.4 Callbacks

This section describes the prototypes and expected functionality of the various callback functions used for event notification from the simulator. All of these functions are expected to have a C-callable interface.

All of these functions require a handle as their first argument. The simulator treats this handle as the identifier for the entity (co-simulation or system simulation application). The simulator never dereferences this handle; the requesting entity is to have a placeholder for any context that might be required. A typical usage is to allocate and populate a context structure for a co-simulation.

4.4.1 Co-simulation required functions

For a co-simulation library, the following functions must be defined:

- `GetCosimVersion()`
- `UnRegisterCosim()`
- Either `RegisterCosim()` or `RegisterCosimArgs()`

  The short form, `RegisterCosim()`, is for cosims that do not require arguments. For example, a monitor that collects profiling information.

  The long form, `RegisterCosimArgs()`, is for cosims that can be configured. For example, a secondary interrupt controller whose configuration parameters might be its size (the number of interrupts it can service), base address of its memory-mapped registers, and primary Hexagon interrupt to which it is attached.

The `RegisterCosim()` and `GetCosimVersion()` functions are the first functions called by the simulator; these calls allow the cosim to initialize itself.

- The simulator first calls `GetCosimVersion()` to determine the API version compatibility.
- Then the simulator calls either `RegisterCosimArgs()` or `RegisterCosim()`:
  - The simulator looks first for `RegisterCosimArgs()` and, if present, calls it and does not look for `RegisterCosim()`.
  - If `RegisterCosimArgs()` does not exist, the simulator calls `RegisterCosim()`.
- The simulator calls `UnRegisterCosim()` during its shutdown procedure.

During the cosim initialization, the cosim can register to receive event notifications from the simulator. Typically, a cosim registers a function to be called when any of its memory-mapped registers are read or written, or it registers a function to be periodically called based on elapsed simulator-time.

During the cosim shutdown procedure, the cosim is expected to clean up any resources that it used (including removing callbacks, freeing memory, and so on) and prepare for shutdown.

4.4.1.1 GetCosimVersion()

Used by the simulator to determine the version of `HexagonWrapper.h` that was used to build the cosim. This function is required for determining API compatibility.

Prototype

```
char * GetCosimVersion (void);
```

Detailed description

This global function is called by the simulator library before it calls `RegisterCosim()` or `RegisterCosimArgs()`. The value that the cosim must return is defined in `HexagonWrapper.h` as `HEXAGON_WRAPPER_VERSION`.

This is a required function that must be implemented in every cosim.

Returns

Version number of the cosim from `HexagonWrapper.h` (Section 4.3.6)

4.4.1.2 RegisterCosim()

Used by the simulator to allow a cosim to self-identify and register for callbacks.

Prototype

```
void * RegisterCosim(char **name,
                     HexagonWrapper *simPtr);
```

Parameters

`out`	*name*	Double pointer to the name of the cosim. This value is allocated and filled in by the cosim and passed to the simulator.
`in`	*simPtr*	Pointer to the Hexagon simulator.

Detailed description

This function is called by the simulator after the call to `GetCosimVersion()`. The `HexagonWrapper` pointer argument provides a handle to the API interface, which can be used by the cosim to register for callbacks.

Before returning from the function call, the cosim must assign itself a unique name using the `**name` argument.

This is a required function that must be implemented in every cosim, unless `RegisterCosim()` is used instead.

Returns

Arbitrary pointer used to identify this cosim.

4.4.1.3 RegisterCosimArgs()

Used by the simulator to allow a cosim to self-identify, register for callbacks, and parse arguments passed to the cosim.

Prototype

```
void * RegisterCosimArgs(char **name,
                         HexagonWrapper *simPtr,
                         char *args);
```

Parameters

`out`	*name*	Double pointer to the name of the cosim. This value is allocated and filled in by the cosim and passed to the simulator.
`in`	*simPtr*	Pointer to the Hexagon simulator.
`in`	*args*	Argument string to the cosim.

Detailed description

This function is called by the simulator after the call to `GetCosimVersion()`. The `HexagonWrapper` pointer argument provides a handle to the API interface, which can be used by the cosim to register for callbacks.

Before returning from the function call, the cosim must assign itself a unique name using the `**name` argument.

The `args` string can be parsed to process command line arguments passed to the cosim.

This is a required function that must be implemented in every cosim, unless `RegisterCosim()` is used instead.

Returns

Arbitrary pointer used to identify this cosim.

4.4.1.4 UnRegisterCosim()

Used by the simulator to allow a cosim to prepare for termination.

Prototype

```
void UnRegisterCosim(void *handle);
```

Parameters

`in`	*handle*	Pointer to the handle given to the simulator when the cosim was registered.

Detailed description

This global function is called by the simulator library. The installed cosim is requested to clean up and prepare for termination. The handle passed in is the same handle returned from `RegisterCosim()` or `RegisterCosimArgs()`.

This function is required and must be implemented in every cosim.

Returns

None.

4.4.2 Callback functions

All callback functions are expected to be C-callable. The handle passed to the callback function is the same handle provided when the callback was registered with the simulator.

NOTE: The callback function names are for illustration purposes only; they can be renamed.

4.4.2.1 BusTransactionRequestCallback()

Called by the simulator library to initiate a bus transaction.

Prototype

```
HEXAPI_TransactionStatus BusTransactionRequestCallback(
       void *handle,
       HEX_PA_t addr,
       HEX_4u_t lengthInBytes,
       HEX_1u_t * data,
       HEX_4u_t requestID,
       HEXAPI_BusAccessType batype,
       HEX_4u_t threadNum,
       HEXAPI_BusBurstType bt);
```

Parameters

in	*handle*	The handle given to the simulator when the callback request was added.
in	*addr*	Address of the write or read that corresponds to the transaction.
in	*lengthInBytes*	Size of data in bytes.
in	*data*	Pointer to the data returned by external bus.
in	*requestID*	ID used to identify this bus transaction. This ID must be provided to the `BusTransactionFinished()` call to complete this bus request.
in	*batype*	Type of bus access. ■ HEX_INSTRUCTION_FETCH ■ HEX_DATA_READ ■ HEX_DATA_WRITE ■ HEX_DATA_CASTOUT ■ HEX_DATA_READ_LOCKED ■ HEX_DATA_WRITE_LOCKED ■ HEX_SYNCH ■ HEX_BARRIER ■ HEX_DATA_READ_PREFETCH ■ HEX_INSTRUCTION_PREFETCH (cont.)

in	*batype* (cont.)	■ HEX_DEBUG_READ ■ HEX_DEBUG_WRITE ■ HEX_DEBUG_READ_LOCKED ■ HEX_DEBUG_WRITE_LOCKED
in	*threadNum*	The hardware thread number of the thread making the request.
in	*bt*	Type of bus burst. ■ HEX_BURST_FIXED – Multiple data items to the same address, such as FIFO ■ HEX_BURST_INCREMENTAL – Multiple data items with incremental addresses ■ HEX_BURST_WRAPPED – Incremental addresses with wraparound, such as filling a cache line

Detailed description

This interface allows the external bus model to model data as well as timing behavior. To indicate the end of the transaction, the external device calls `BusTransactionFinished()`

A `batype` value of `HEX_DEBUG_READ`, `HEX_DEBUG_WRITE`, `HEX_DEBUG_READ_LOCKED`, or `HEX_DEBUG_WRITE_LOCKED` indicates a debug-related transaction. In this case the cosim should perform the requested action and immediately complete the request by calling `BusTransactionFinished()`. If the transaction cannot be completed, an error is returned.

NOTE: Debug read and write transactions are also designated with a `requestID` of `0xFFFFFFFF`.

Returns

- TRANSACTION_SUCCESS
- TRANSACTION_REPLAY – Reissue the transaction.
- TRANSACTION_FAIL – Transaction failure.
- TRANSACTION_LOCKED – Memory is locked.

4.4.2.2 TimedCallback()

Called by the simulator library at certain intervals, provided the cosim registered to be called using `AddTimedCallback()`.

Prototype

```
void TimedCallback(void *handle);
```

Parameters

in	*handle*	Pointer to the handle given to the simulator when the callback request was added.

Returns

None.

4.4.2.3 MemoryWasWrittenCallback()

Called by the simulator library when data is written to system memory (inclusive of cache, TCM, and bus transactions), provided the cosim registered to be called using `AddMemWasWrittenCallback()`.

Prototype

```
void MemoryWasWrittenCallback(void *handle,
                              HEX_PA_t addr,
                              HEX_8u_t value,
                              HEX_4u_t sizeInBytes);
```

Parameters

in	*handle*	Pointer to the handle given to the simulator when the callback request was added.
in	*addr*	Address of the memory write.
in	*value*	Value written to memory.
in	*sizeInBytes*	Size of the data (1, 2, 4, or 8 bytes).

Returns

None.

4.4.2.4 MemoryWasReadCallback()

Called by the simulator library when data is read from system memory (inclusive of cache, TCM, and bus transactions), provided the cosim registered to be called using `AddMemWasReadCallback()`.

Prototype

```
void MemoryWasReadCallback(void *handle,
                           HEX_PA_t addr,
                           HEX_8u_t value,
                           HEX_4u_t sizeInBytes);
```

Parameters

in	*handle*	Pointer to the handle given to the simulator when the callback request was added.
in	*addr*	Address of the memory read.
in	*value*	Value read from memory.
in	*sizeInBytes*	Size of the data (1, 2, 4, or 8 bytes).

Returns

None.

4.4.2.5 FrequencyChangeCallback()

Called by the simulator library when the Hexagon core frequency is changed by an external agent, provided the cosim registered to be called using `AddFrequencyChangeCallback()`.

Prototype

```
void FrequencyChangeCallback(void *handle,
                             HEX_8u_t coreFrequency);
```

Parameters

in	*handle*	Pointer to the handle given to the simulator when the callback request was added.
in	*coreFrequency*	New frequency of the Hexagon core.

Returns

None.

4.4.2.6 PCCallback()

Called by the simulator library when the Hexagon processor is ready to commit the instruction at the PC specified with the `AddPCCallback()` function.

Prototype

```
void PCCallback(void *handle);
```

Parameters

`in`	*handle*	Pointer to the handle given to the simulator when the callback request was added.

Returns

None.

4.4.2.7 CoreReadyCallback()

Called by the simulator library when the core's CORE_READY signal changes state, provided the cosim registered to be called using `AddCoreReadyCallback()`. This function indicates whether the core is ready to accept second-level interrupts.

Prototype

```
void CoreReadyCallback(void *handle,
                       HEXAPI_CoreReadyState polarity);
```

Parameters

`in`	*handle*	Pointer to the handle given to the simulator when the callback request was added.
`in`	*polarity*	State of polarity. ■ CORE_READY ■ CORE_NOT_READY

Returns

None.

4.4.3 Cosim example

The following example implements a simple cosim using the system simulation APIs. This cosim merely prints a message after a specified time period has elapsed.

```
#include "HexagonWrapper.h"

typedef struct {  // Simple context structure
  int interval;
  HexagonWrapper *simPtr;
} TimerContext;
void timed_callback(void *handle); // Routine to be called
int parseAndValidateArgs(char *args, TimerContext *c);

extern "C" { void INTERFACE *RegisterCosimArgs(char *name,
                                HexagonWrapper *simPtr,
                                char *args)
{
  int cerror;
  TimerContext *pTimer = (TimerContext *)
                                calloc(1, sizeof(TimerContext);
  pTimer->simPtr = simPtr; // Save pointer
  // Parse options for cosim - sets callback interval
  cerror = parseAndValidateArgs(args, pTimer);
  if (cerror == 0) {
    printArgUsage(cerror);
    exit(0);
  }
  // Get called back every <interval>ns
  simPtr->AddTimedCallback(pTimer, pTimer->interval,
                                HEX_NANOSEC, timed_callback);
  return (void *) pTimer;
} }

void timed_callback(void *handle)
{
  TimerContext c = (TimerContext *) handle;
  printf("Called back, interval=%dns\n", c->interval);
  c->simPtr->RemoveTimedCallback((void *) c);
}

int parseAndValidateArgs(char *args, TimerContext *c)
{
  return (c->interval = atoi(args)) != 0);
}

extern "C" { void INTERFACE UnRegisterCosim(void *handle)
{
  TimerContext c = (TimerContext *) handle;

  // Free resources allocated.  Should also remove callback, but
  // that was done above.

  free(c);
}
```

```
char * INTERFACE GetCosimVersion(void)
{   // Required vor version checking by the simulator
  return HEXAGON_WRAPPER_VERSION;
}
}
```

Build this cosim

1. Compile the cosim and link it into a shared library.
2. Build a cosim configuration file that contains the pathname to the shared library followed by an argument specifying the interval after which the message is printed. For example:

   ```
   /<path_to_library>/cosim.so 200
   ```

   NOTE: This command specifies that the cosim should be loaded and the string, 200, passed to the cosim initialization routine (`RegisterCosimArgs()`).

3. Specify the cosim configuration file to the simulator either by command-line option or API function (see Section 4.3.7 for details).

Output

When the cosim executes, it prints out the message after 200ns of elapsed simulation time.

A Statistics

The simulator collects execution statistics on the applications it executes. The statistics summarize the various types of Hexagon processor events that occurred while the application was running.

When an application terminates, the simulator writes the collected statistics to a dedicated statistics file (Section 3.3.9). The symbols that appear in the file define the types of statistics collected.

Because the types of execution events vary across the Hexagon processors, different types of statistics are collected for each processor version. For more information, see the applicable *Qualcomm Hexagon Programmer's Reference Manual*.