Qualcomm® Hexagon™ Standalone Application

User Guide

80-N2040-22 Rev. K

December 2, 2019

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

Contents

1 Introduction

This document describes the build procedure, runtime support library, and example program for standalone Qualcomm® Hexagon™ applications that execute on the Hexagon processor.

These standalone applications are software programs that perform specific tasks such as vocoding. There are two types of applications:

- RTOS applications that execute with a real-time operating system
- Standalone applications that execute without operating system support

1.1 Conventions

- Courier font is used for computer text, for example, `printf("Hello world\n");`.
- The following notation is used to define the syntax of functions and commands:
  - Square brackets enclose optional items, for example, **`help`** `[command]`.
  - **`Bold`** is used to indicate literal symbols, for example, the brackets in *`array`***`[`***`index`***`]`**.
  - The vertical bar character, `|`, is used to indicate a choice of items.
  - Parentheses are used to enclose a choice of items, for example, `(`**`on`**`|`**`off`**`)`.
  - An ellipsis, `. . .` , follows items that can appear more than once.
  - *`Italics`* are used for terms that represent categories of symbols.

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 Build procedure

The build procedure for standalone Hexagon processor applications is identical to the build procedure for standard Linux applications. The procedure supports user configuration of the following runtime system properties:

- Stack and heap
- Memory management
- Caches
- ISDB
- TCM

This document does not describe the following runtime environment properties for standalone applications:

- Default IMASK and other register values
- Permissions for non-fixed TLB entries (such as rwx for each 1MB segment)

NOTE: These properties are currently not user-configurable and are subject to change in future tools releases.

2.1 Runtime system configuration

The runtime system for standalone Hexagon processor applications is configurable to support various hardware and software configurations.

The program start address is set with the linker option, `--section-start`. For example:

```
hexagon-clang -Xlinker --section-start -Xlinker .start=address hello.c
```

The `-Xlinker` compiler option is used to specify that the following argument is passed to the linker as a command option.

2.2 L1 and L2 cache attributes

L1 and L2 cache attributes can be set at runtime by calling the library function, add_translation() (Section 3.2).

2.3 Parameters

The runtime system is configured primarily by passing various parameters to the linker on the compiler command line. In the following example, three parameters (`STACK_SIZE`, `HEAP_SIZE`, and `STACK_START`) are passed to specify the application stack and heap:

```
hexagon-clang -Wl,--defsym,STACK_SIZE=0x10000 -Wl,--defsym,
HEAP_SIZE=0x10000 -Wl,--defsym,
STACK_START=global_end+HEAP_SIZE+STACK_SIZE hello.c
```

The `-Wl` compiler option specifies that the following two arguments are passed to the linker as a single option. For example, the first `-Wl` option in the example above specifies the following linker option:

```
--defsym STACK_SIZE=0x10000
```

`-Xlinker` can also be used to pass parameters to the linker. However, in this case it is less efficient than `-Wl` because it can pass only one symbol at a time.

If you are linking an application directly (for example, by using the `hexagon-link` command), these parameters can be set directly using the `--defsym` linker option.

Section 2-1 lists the parameters defined for configuring the runtime system.

- The default parameter values are consistent with developing a standalone application that runs on the Hexagon processor simulator.
- The mode-controlling parameters use nonstandard values. Specifically, the value 1 indicates that a mode or feature is disabled (rather than enabled).

Table 2-1 Runtime system parameters

Parameter	Description	Category
`STACK_START`	Base address of the program stack. Default = (end of heap) + (stack size)	Application
`STACK_SIZE`	Maximum stack size. Default = 1 MB	
`HEAP_START`	Base address of the program heap. Default = End of the global data area	
`HEAP_SIZE`	Heap size. Default = 64 MB	
`PRE_INIT`	Address of the function that is called before any initialization is performed (if defined).	
`POST_EXIT`	Address of the function that is called after the program finishes executing, with no simulator or RTOS present (if defined).	

Table 2-1 Runtime system parameters (cont.)

Parameter	Description	Category
`ANGEL_SUPPORT`	Specifies whether support for Angel semi-hosting is enabled. ■ 0 – Use default setting ■ 1 – Semi-hosting support is disabled ■ 2 – **Semi-hosting support is enabled (Default)** Angel support is valid only if ISDB is secure and in trusted mode (by setting ISDB_SECURE_FLAG and ISDB_TRUSTED_FLAG).	Application (cont.)
`ENABLE_DMT`	Specifies whether dynamic multi-threading is enabled. ■ 0 – Use default setting ■ 1 – Multi-threading is disabled ■ 2 – **Multi-threading is enabled (Default)** Valid only for processor version V5 or greater	Multi-threading
`EVENT_VECTOR_BASE`	Event vector table base address (used to set EVB register).	Events
`I_CACHE_ENABLE`	Specifies whether the instruction cache is enabled. ■ 0 – Use default setting ■ 1 – Cache is disabled ■ 2 – **Cache is enabled (Default)**	Cache
`I_CACHE_HW_PREFETCH`	Specifies whether hardware instruction cache prefetching is enabled. ■ 0 – Use default setting ■ 1 – Prefetching is disabled ■ 2 – **Prefetching is enabled (Default)**	
`D_CACHE_ENABLE`	Specifies whether the data cache is enabled. ■ 0 – Use default setting ■ 1 – Data cache is disabled ■ 2 – **Data cache is enabled (Default)**	
`D_CACHE_HW_PREFETCH`	Specifies whether the hardware data cache is enabled. ■ 0 – Use default setting ■ 1 – Prefetching is disabled ■ 2 – **Prefetching is enabled (default)**	
`L2_CACHE_SIZE`	Size of the L2 cache. ■ 0 – **Target-specific maximum L2 cache (Default)** ■ 1 – 0 KB L2 cache (all TCM) ■ 2 – 64 KB L2 cache ■ 3 – 128 KB L2 cache ■ 4 – 256 KB L2 cache ■ 5 – 512 KB L2 cache ■ 6 – 1024 KB L2 cache	
`L2_PARITY`	Specifies whether L2 parity is enabled. ■ 0 – **Parity is disabled (Default)** ■ 1 – Parity is enabled	

Table 2-1 Runtime system parameters (cont.)

Parameter	Description	Category
`L2_WB`	Specifies whether L2 write back is enabled. ■ 0 – **Write back is enabled (Default)** ■ 1 – Write back is disabled	Cache (cont.)
`TCM_BASE_ADDR`	Base address of the TCM memory. This address is processor dependent. Valid only if L2/TCM partitioning enabled.	
`ENABLE_TRANSLATION`	Specifies whether MMU page table translation is enabled. ■ 0 – Use default setting ■ 1 – Translation is disabled ■ 2 – **Translation is enabled (Default)** The MMU handles virtual-to-physical address translation.	TLB
`USE_DEFAULT_TLB_MISS_HANDLER`	Specifies which TLB miss handler is used. ■ 0 – Use default setting ■ 1 – Use RTOS-provided TLB miss handler ■ 2 – **Use default TLB miss handler (Default)**	
`TLB_MAP_TABLE_PTR`	Address of table of the default TLB entries. This address is used to load the TLB entry on a TLB miss. Valid only if default TLB miss handler used.	
`ISDB_SECURE_FLAG`	Specifies whether ISDB is secure. ■ 0 – Use default setting ■ 1 – **ISDB is not secure (Default)** ■ 2 – ISDB is secure	Debugging
`ISDB_TRUSTED_FLAG`	Specifies whether ISDB is trusted. ■ 0 – Use default setting ■ 1 – **ISDB untrusted mode (Default)** ■ 2 – ISDB trusted mode	
`ISDB_DEBUG_FLAG`	Specifies whether ISDB debugging is enabled. ■ 0 – Use default setting ■ 1 – **ISDB debug is off (Default)** ■ 2 – ISDB debug is on	
`CORE_DUMP_BASE`	Base address of the core dump (if defined; otherwise use the default location).	
`ENABLE_PCYCLE`	Specifies whether auto-incrementing of the PCYCLE register is enabled. ■ 0 – Use default setting ■ 1 – Auto-incrementing is disabled ■ 2 – **Auto-incrementing is enabled (Default)**	Profiling

3 Runtime support library

The standalone runtime support library supports the following features:

- Memory management
- Multi-threaded programming
- Thread synchronization (using mutexes)
- Interrupt handling
- HVX engine management

The library is accessed by including the library header file, `hexagon_standalone.h`. (For an example program, see Chapter 4).

Following are the functions and macros defined in the library.

Table 3-1 Runtime support library

`add_translation()`	Section 3.2
`add_translation_fixed()`	Section 3.3
`add_translation_extended()`	Section 3.4
`thread_create()`	Section 3.5
`thread_create_extended()`	Section 3.6
`thread_stop()`	Section 3.7
`thread_join()`	Section 3.8
`thread_get_tnum()`	Section 3.9
`lockMutex()`	Section 3.10
`unlockMutex()`	Section 3.11
`trylockMutex()`	Section 3.12
`register_interrupt()`	Section 3.13
`SIM_ACQUIRE_HVX`	Section 3.14
`SIM_RELEASE_HVX`	Section 3.15
`acquire_vector_unit()`	Section 3.16
`release_vector_unit()`	Section 3.17
`set_double_vector_mode()`	Section 3.18
`clear_double_vector_mode()`	Section 3.19
`power_vector_unit()`	Section 3.20

3.1 Cache properties

The address translation functions (Section 3.2 through Section 3.4) accept a parameter that specifies the following cache properties of a remapped memory page:

- Cached or uncached
- Cache write-back (WB) or write-through (WT)

These properties can be specified for the L1 data and instruction caches, and for the L2 cache.

Table 3-2 lists the cache properties and the values used to specify them.

Table 3-2 Cacheability values

Value	L1 data cache	L1 instruction cache	L2 cache
0	Cached, WB	Cached	Uncached
1	Cached, WT	Cached	Uncached
2	reserved	Reserved	Reserved
3	reserved	Reserved	Reserved
4	Device-type	Uncached	Uncached
5	Cached, WT	Uncached	Cached, WT
6	Uncached	Uncached	Uncached
7	Cached, WB	Cached	Cached, WB [1]
8	Cached, WB	Cached	Cached, WT
9	Cached, WT	Cached	Cached, WB
10	Cached, WB	Uncached	Cached, WB/AUX [2]
11	Cached, WT	Uncached	Cached, WT/AUX
12	reserved	Reserved	Reserved
13	Uncached	Uncached	Cached, WT
14	reserved	Reserved	Reserved
15	Uncached	Uncached	Cached, WB

[1] If L2$ supports write-back and WB feature is enabled; otherwise use WT.

[2] Lines will be allocated into Auxiliary partition of L2$.

Table 3-3 lists the V67 Small Core (V67t) cache properties and the values used to specify them.

Table 3-3 Cacheability values (V67 Small Core)

Value	L1 data cache	L1 instruction cache	L2 Behavior
0	Reserved	Reserved	Reserved
1	Cacheable	WT Cached	Uncached
2	Device-type, SFC	Uncached	Uncached
3	Uncached SFC	Uncached	Uncached
4	Device-type	Uncached	Uncached
5	Cacheable, WT	Uncached	Uncached
6	Uncached	Uncached	Uncached
7	Cacheable, WB	Cached	Cached, WB
8-15	Reserved	Reserved	Reserved

3.2 Address translation

Remaps a Hexagon processor memory page.

Prototype

```
void add_translation(void *va,
                     void *pa,
                     int cacheability)
```

Parameters

va	Pointer to the virtual memory address.
pa	Pointer to the corresponding physical memory address.
cacheability	Cache properties (Table 3-2). By default, the cacheability setting of each page is 0.

Detailed description

The add_translation() function is not designed to be thread safe. It can be called only from thread 0 and before any other threads are created.

The library defines memory as consisting of 4096 1 MB pages. By default, each page is assigned a one-to-one mapping between virtual and physical memory. However, standalone applications can remap individual pages to different areas in physical memory.

The application code remaps a page by calling add_translation(). For example:

```
add_translation(0x100000, 0xD8000000, 2);
add_translation(0x200000, 0xD8100000, 4);
```

For more information on memory management and caches, see the appropriate *Qualcomm Hexagon Vxx Programmer's Reference Manual*.

3.3 Address translation (fixed TLB)

Remaps a Hexagon processor memory page using a fixed TLB entry.

Prototype

```
void add_translation_fixed(int index
                           void *va,
                           void *pa,
                           int cacheability,
                           int permissions)
```

Parameters

index	Index of a fixed TLB entry (1 through 5).
va	Pointer to the virtual memory address. If this parameter is NULL, the entry specified by the index parameter is invalid.
pa	Pointer to the corresponding physical memory address.
cacheability	Cache properties (Table 3-2).
permissions	Memory access rights: ■ 1 – Read ■ 2 – Write ■ 4 – Execute Multiple memory access rights can be specified by adding the individual permission values. For example, read/write permission is specified with the value 3 (1+2).

Detailed description

The add_translation_fixed() is not designed to be thread safe. It can be called only from thread 0 and before any other threads are created.

The default TLB miss handler maintains a table of TLB entries for the address translations performed by a standalone application (Section 3.2). The handler uses a round-robin replacement strategy for all TLB entries except the first six:

- TLB index zero (0) is reserved and always provides translation for the TLB table.
- TLB entries 1 through 5 are excluded from the TLB-entry replacement strategy (that is, they are fixed) to support programs that require specific address translations.

The application code remaps a page using one of the fixed TLB entries by calling add_translation_fixed(). For example:

```
add_translation_fixed(1, 0x100000, 0xD8000000, 2, 4);
add_translation_fixed(4, 0x200000, 0xD8100000, 4, 7);
```

3.4 Address translation (extended)

Remaps a Hexagon processor memory page using direct access to a TLB entry.

NOTE: This function is valid only for Hexagon processor version V5 or greater.

Prototype

```
int add_translation_extended(int index
                             void *va,
                             uint64_t pa,
                             unsigned int page_size,
                             unsigned int xwru,
                             unsigned int cccc,
                             unsigned int asid,
                             unsigned int aa,
                             unsigned int vg)
```

Parameters

index	Index of a fixed TLB entry (1 through 5).
va	Pointer to the virtual memory address.
pa	Corresponding physical memory address.
page_size	Page size (in bytes): ■ 1 – 4 KB ■ 2 – 16 KB ■ 4 – 64 KB ■ 8 – 256 KB ■ 16 – 1 MB ■ 32 – 4 MB ■ 64 – 16 MB
xwru	Integer value interpreted as a 4-bit value representing the X, W, R, and U bits in a TLB entry.
cccc	Cache properties (Table 3-2).
asid	Integer value interpreted as a 7-bit value representing the ASID bit field in a TLB entry.
aa	Integer value interpreted as a 2-bit value representing the A1 and A0 bits in a TLB entry.
vg	Integer value interpreted as a 2-bit value representing the V and G bits in a TLB entry.

Returns

- 0 — If the specified page is successfully remapped.
- Non-0 — Otherwise.

Detailed description

This function is similar to add_translation_fixed() (Section 3.3), but it allows you to directly set the individual fields in a TLB entry. This function can also be used to specify page mappings with physical addresses that exceed 32 bits.

This add_translation_extended() function is not designed to be thread safe. It can be called only from thread 0 and before any other threads are created.

All TLB fields must be set to valid values. For more information on TLB entries, see the appropriate *Qualcomm Hexagon Vxx Architecture System-level Specification*.

NOTE: Allocating a `page_size` of 4 KB, 16 KB, 64 KB, or 256 KB (any size less than 1 MB) will leave the remainder of the address space around the allocated `page_size` address area, within the 1 MB block in which it sits, unusable unless you manually create a TLB table entry. Any memory access of the unusable memory space will result in a TLB miss-RW exception.

3.5 Create thread

Creates and starts a new thread.

Prototype

```
void thread_create(void (*pc) (void *),
                   void *sp,
                   int threadno,
                   void *param);
```

Parameters

pc	Pointer to the function executed by the thread.
sp	Pointer to the memory area used as a thread stack. This parameter must be 8-byte aligned. Also, it must be set to the highest stack address in the thread stack memory area because the stack grows downward. For more information on stacks, see the appropriate *Qualcomm Hexagon Vxx Programmer's Reference Manual*.
threadno	Hardware thread number assigned to the thread (0 through 5).
param	Pointer to the data structure accessed by the thread.

3.6 Create thread (extended)

Creates and starts a new thread using stack protection attributes.

NOTE: This function is intended for use only with processor version V61 or greater. If used with earlier processor versions, the `framekey` and `stacksize` attributes are ignored, and the function is then equivalent to thread_create().

Prototype

```
void thread_create_extended(void (*pc) (void *),
                            void *sp,
                            int threadno,
                            unsigned framekey,
                            unsigned stacksize,
                            void *param);
```

Parameters

pc	Pointer to the function executed by thread.
sp	Pointer to the memory area used as thread stack. This parameter must be 8-byte aligned. Also, it must be set to the highest stack address in the thread stack memory area because the stack grows downward.
threadno	Hardware thread number assigned to thread (0 through 5).
framekey	32-bit value used to scramble return addresses stored on stack.
stacksize	Size (in bytes) of memory area used as thread stack. This parameter must be greater than 0; it is used to set the FRAMELIMIT control register. For more information on stacks, see the appropriate *Qualcomm Hexagon Vxx Programmer's Reference Manual*.
param	Pointer to the data structure accessed by thread.

Detailed description

This function is similar to thread_create() (Section 3.5), but it allows you to specify the `framekey` and `stacksize` stack protection attributes.

3.7 Stop thread

Stops execution of the currently executing thread.

Prototype

```
void thread_stop(void);
```

3.8 Join thread

Suspends the currently executing thread until the specified threads stop.

Prototype

```
void thread_join(int mask);
```

Parameters

mask	Bit mask that specifies one or more hardware threads. Bits 0 through 5 in the value indicate whether the corresponding hardware thread numbers are specified (for example, bit 0 = thread 0).

3.9 Get thread number

Returns the hardware thread number (0 through 5) assigned to the currently executing thread. This function is implemented as a macro.

Prototype

```
int thread_get_tnum(void);
```

3.10 Lock mutex

Locks a specified mutex.

Prototype

```
void lockMutex(int *mutex);
```

Parameters

mutex	Pointer to the address of the variable used as a mutex.

Detailed description

The variable referenced by mutex must be a global variable that is initialized to 0 before use as a mutex.

- If the mutex variable is declared in internal memory, the variable's memory attributes must be specified as cached and write-back; otherwise, the mutex behavior is undefined.
- If the mutex variable is declared in external memory, the variable's memory attributes must be specified as uncached; otherwise, the mutex behavior is undefined.

External-memory mutexes have additional system requirements. For more information, see the appropriate *Qualcomm Hexagon Vxx Programmer's Reference Manual*.

3.11 Unlock mutex

Unlocks a specified mutex. (For more information, see Section 3.10.)

Prototype

```
void unlockMutex(int *mutex);
```

Parameters

mutex	Pointer to the address of the variable used as a mutex

3.12 Trylock mutex

Attempts to lock a specified mutex.

Prototype

```
int trylockMutex(int *mutex);
```

Parameters

mutex	Pointer to the address of the variable used as a mutex

Detailed description

If the specified mutex is not being used, this function performs a normal lock operation and returns the result value 1. Otherwise, it does not attempt to lock the mutex and returns the result value 0.

For more information, see Section 3.10.

3.13 Register interrupt

Assigns a callback function to an interrupt.

Prototype

```
void register_interrupt(int intno,
                        void (*IRQ_handler)(int intno));
```

Parameters

intno	Interrupt number (0 through 31).
IRQ_handler	Pointer to the interrupt callback function.

Detailed description

Interrupt callbacks are functions that are called by an interrupt. A callback function is defined in the application. It must be defined to accept the interrupt identifier as a function argument. The argument enables a single callback function to be written so it can handle multiple interrupts.

The Hexagon processor uses an L2 vectored interrupt controller (L2VIC) to manage interrupts. When the handler function is called, the `intno` parameter is always set to 31, with the actual interrupt number (0 through 1023) being stored in the Hexagon system-level register VID.

To access the actual interrupt number, the handler can define a simple register-access function. For example:

```
static inline uint32 get_int_number(void)
{
    uint32 reg;
    asm volatile ("%0=vid;"
                  :"=r"(reg));
    return reg;
}

void IRQ_handler(int intno)
{
    uint32 INT_number;
    INT_number = get_int_number();
    switch (INT_number) {
         case 0:
                // Code to process interrupt number 0
                break;

         case 1:
                // Code to process interrupt number 1
                break;

         case default:
                // Code to process interrupts 2 to 1023
             break;
    }
}
```

If no callback function is assigned to an interrupt, the interrupt triggers an empty callback function. For more information, see the appropriate *Qualcomm Hexagon Vxx Programmer's Reference Manual*.

3.14 Acquire HVX engine

Acquires an HVX engine before using it. Use of this macro is system-dependent.

Prototype

```
SIM_ACQUIRE_HVX;
```

Detailed description

When the number of hardware threads is greater than the number of HVX units, a thread using HVX instructions must first explicitly acquire one of the HVX engines.

A thread acquires an HVX engine by calling the macro, `SIM_ACQUIRE_HVX`. When a thread is finished using the HVX engine, it must call `SIM_RELEASE_HVX` to release the engine (Section 3.15).

3.15 Release HVX engine

Releases an HVX resource. Use of this macro is system-dependent.

Prototype

```
SIM_RELEASE_HVX;
```

3.16 Acquire HVX unit

Grabs a vector unit.

Prototype

```
int acquire_vector_unit(hexagon_vector_wait_t wait);
```

Parameters

wait	The input can be one of the following: ■ HEXAGON_VECTOR_WAIT – Wait until a resource is free. The SIM_ACQUIRE_HVX macro calls acquire_vector_unit() with HEXAGON_VECTOR_WAIT. ■ HEXAGON_VECTOR_NO_WAIT – Return with or without the resource acquired. ■ HEXAGON_VECTOR_CHECK – Check to see if a resource is free.

Returns

- 1 – If a vector unit as been acquired.
- 0 – If a vector unit has not been acquired.
- Number of vector units that are free. Returned when `HEXAGON_VECTOR_CHECK` is passed. `HEXAGON_VECTOR_CHECK` does not lock the unit.

3.17 Release HVX unit

Unlocks a vector unit.

Prototype

```
void release_vector_unit();
```

Detailed description

Unlocking the vector resource allows other threads to use it. This thread's SSR:XA and SSR:XE bits are reset.

Subsequent HVX instructions will fault with an illegal execution of coprocessor instruction, SSR:CAUSE of 0x16.

The `SIM_RELEASE_HVX` macro calls release_vector_unit().

3.18 Set Double Vector mode

Puts a vector unit into Double Vector mode.

Prototype

```
void set_double_vector_mode();
```

Detailed description

Code used in this mode must be built with `-mhvx=double`.

The `SIM_SET_HVX_DOUBLE_MODE` macro calls set_double_vector_mode().

3.19 Clear Double Vector mode

Clears the 128-byte vector mode bit in the SYSCFG register.

Prototype

```
void clear_double_vector_mode();
```

Detailed description

Code used in this mode must be built with `-mhvx=single`.

The `SIM_CLEAR_HVX_DOUBLE_MODE` macro calls clear_double_vector_mode().

3.20 Power vector unit

Provides subsystem addresses for clock, reset, and power delay; and enables or disables the unit.

Prototype

```
extern void power_vector_unit (uint32_t volatile *clockbase,
                               uint32_t volatile *resetbase,
                               uint32_t volatile *powerbase,
                               int delay, int state);
```

Parameters

clockbase	Pointer to the clock subsystem address.
resetbase	Pointer to the reset subsystem address.
powerbase	Pointer to the power delay subsystem address.
state	Specify whether the unit is disabled (0) or enabled (1).

Detailed description

This function is used for very low-level verification and configuration on hardware.

This function will not do anything if it is called while running on a simulator (it is a no-operation function). The function will perform its stated operations only when running on real hardware.

4 Example

Following is a demonstration of how to use standalone applications and the runtime support library (Chapter 3).

NOTE: The example program files are stored in the Hexagon tools installation folder in the directory, `Examples/StandAlone_Applications`.

Mandelbrot is the example program provided with the Hexagon tools releases. It computes a fractal image and displays it using character graphics. The program is contained in the single file, `mandelbrot.c`, which includes the header file, `hexagon_standalone.h` that is used to access the runtime support library.

To build the program, follow the instructions in the associated README file.

When executed, the program performs the following steps:

1. It spawns multiple worker threads using the runtime support library function, thread_create().

   The number of threads created depends on the Hexagon processor version being used.

2. Each worker thread independently computes one part of the fractal image.
3. After the computations are completed, one of the worker threads locks a mutex (using the library function, lockMutex()), and writes the computed fractal image into the image buffer.
4. The program's master thread (thread 0) then displays the computed fractal image on the console and unlocks the mutex.
5. When the program finishes executing, the Hexagon simulator creates a file named `pmu_statsfile.txt`, which contains the simulation statistics information.

NOTE: Because the program executes on the simulator, it takes some time before the computations are completed and the fractal image is displayed.