Thread Manager Specifications

Purpose and Overview

The purpose of this document is to three fold. First define the operations of the thread manager. Second, as a reference to software interfacing with the Thread Manager. Third to solve insomnia, as I am sure if anyone is reading this document, who is not already asleep, will fall asleep. In particular, this document covers version 1.3 of the Thread Manager.

The Thread Manager is a hardware component, within a Hybrid-threads system, who's goal is to control the order of operations of all the threads running on the system. In some regards, you may refer to the Thread Manager as the low level mechanism enforcing the hthreads library calls.

Thread Manager Access Registers

The Thread Manager utilizes a series of address mapped registers to initiate its various functional operations. In general, most of these registers use read operations. Each register is 32 bits wide.

The following registers are used to access a variety of functions built in to the FPGA. All registers are accessed as having a 32 bit width even though the implementation may only use a subset of the full 32 bits. In these cases, the least significant bits are utilized, and the upper bits are padded with zeros for read operations and ignored for write operations. Write operations performed on read only registers generate an exception to the CPU.

The depth of a given register specifies the number of successive, 32 bit locations that are utilized by this register. Registers having a depth greater than one are utilizing the least significant address lines to specify a parameter to be passed to a hardware function associated with this register. To keep each access on an even 32 bit boundary, the parameter to be passed in must be multiplied by 4 before adding to the base address of the register.

Example: A particular register accepts an 8 bit parameter encoded into the address used to access the register (depth=256). The address corresponding to a parameter of zero equals the base address of the register. The address corresponding to a parameter of one equals the base address of the register plus four.

References in the following descriptions to a thread’s status refer to that thread’s entry in the Thread ID Table and utilize the format specified in the Thread ID Table Encoding on page 4.

References in the following descriptions to “queue” refer to the Ready to Run Queue, implemented as a linked list within the Thread ID Table, where each thread has a pointer to the next thread in the queue.

Note : This document is a modification of revision two of the Software Thread Manager API. It was modified to be a preliminary design document for a Thread Manager.

add_thread

Read only

Depth = 256
Overview
Reading this register adds the encoded thread ID to the queue. If the thread happens to be a SW thread, the thread ID is added to the queue. If the thread happens to be a HW thread, the scheduler sends a “RUN” command to the command register of the thread.

Error Checks
The thread’s status must be: USED, NOT_EXITED, and NOT_QUEUED

Calls to Scheduler

is_queued(threadID)
Scheduler will return a 1 on the “data” line if the thread ID is queued, else will return a 0.

enqueue(threadID)
Scheduler will add the given thread ID to the queue, if the thread ID happens to be a SW thread. If the thread ID is a HW thread, a “RUN” command is sent to the command register of the HW thread.

Pseudo-Code

attr = threads[threadID]
if ( attr.used and !attr.exited )
  q = is_queued(threadID)
  if ( !q )
    retVal = enqueue(threadID)
    if ( retVal != error )
      return 0
    else
      return error
  else
    return error
else
  return error

Return Value on Success
0

Return Value on Error
If the error check fails, the value returned is the thread’s current status with the ERR_BIT set to indicate that the operation was unsuccessful. The error value will be 0b000.

clear_thread
Read only.
Depth = 256
Overview

Reading this register deallocates the encoded thread ID by setting the thread’s status to, NOT_USED, NOT_EXITED, NOT_JOINED, NOT_DETACHED, and the thread’s PID field is set to zero.

Error Checks

The calling thread must match the current_cpu_thread

Calls to Scheduler

Pseudo-Code

```c
if ( parentID == current_cpu_thread )
    attr = !used, !exited, !joined, !detached
    attr.parent = 0
    threads[threadID] = attr
else
    return error
```

Return Value on Success

0

Return Value on Error

If the error check fails, the value returned is the thread’s current status with the ERR_BIT set to indicate that the operation was unsuccessful.

create_thread_detached

Read only.

Depth = 1

Overview

This register is read to retrieve an unused thread ID in preparation for creating a new, detached thread.

If successful, the status of the new thread is updated to be; USED, NOT_EXITED, NOT_JOINED, Detached, and the thread’s parent ID field(PID) is set to zero.

Error Checks

Make sure that all thread IDs are not already used.

Calls to Scheduler

Pseudo-Code

```c
if ( newThreadIDAvail )
    return new threadID
    attr = used, !exited, !joined, detached
    attr.parent = 0
```
threads[threadID] = attr
else
    return error

**Return Value on Success**
The thread ID for the new thread. The ID is returned on bits 23..30, all other bits are 0.

**Return Value on Error**
If no thread ID is available the read operation returns ERROR_IN_STATUS + the ERR_BIT set.

create_thread_joinable
Read only.
Depth = 1

**Overview**
This register is read to retrieve an unused thread ID number in preparation for creating a new joinable thread. The new thread must explicitly be joined later to prevent memory leaks.

If successful, the status of the new thread is updated to be; USED, NOT_EXITED, NOT_JOINED, NOT_DETACHED, the thread’s parent ID field(PID) is set to the value of the calling thread. The value of the calling thread is always the value of the current CPU thread. Hardware may not create threads.

**Error Checks**
Make sure that all thread Ids are not already used.

**Calls to Scheduler**

**Pseudo-Code**

```c
if ( newThreadIDAvail )
    return new threadID
    attr = used, !exited, !joined, !detached
    attr.parent = current_cpu_thread
    threads[threadID] = attr
else
    return error
```

**Return Value on Success**
The thread ID for the new thread. The ID is returned on bits 23..30, all other bits are 0.

**Return Value on Error**
If a thread ID is not available the operation returns ERROR_IN_STATUS plus the ERR_BIT set.

detach_thread (REMOVE)
This call is being removed from the Thread Manager.


**exit_thread**

read only, depth = 256

**Overview**

This register is read to update the status of the calling thread to show that it has terminated. The calling thread should encode its ID. The effects of this operation vary depending on the current status of the thread, in particular if the thread is joinable or detached and are summarized below.

If the thread’s status shows it to be detached, the thread ID is deallocated, changing its status to, NOT_USED, NOT_EXITED, NOT_JOINED, NOT_DETACHED, and its PID field to zero.

If the thread’s status is NOT_DETACHED, the thread’s status is updated to USED, EXITED. Then, if the thread’s PID ≠ 0, and the thread status is furthermore JOINED, check if the parent thread’s status = USED, NOT_EXITED, NOT_QUEUED. If so, add the parent thread’s ID to the queue, and return zero. If the parent happens to be HW, the scheduler will send a “RUN” command to the hardware thread.

**Error Checks**

If the calling thread is JOINED, check that the parent thread is not queued. This indicates that the parent did not block until the calling thread was finished.

**Calls to Scheduler**

isQueued(parentID)

Scheduler will return a 0 on the “data” line if the thread ID is queued, else will return a 1.

enqueue(parentID)

Scheduler will add the parentID to the queue.

**Pseudo-Code**

attr = threads[threadID]
if attr.detached
    set attributes of thread to unused
else // joined or joinable
    set attributes to used & exited
if attr.joined
    parentID = attr.parent
    if ( isQueued(parentID) )
        return error
else
    enqueue(parentID)

**Return Value on Success**

0

**Return Value on Error**

Return THREAD_ALLREADY_QUEUED plus the error bit is set.
**get_idle_thread (REMOVE)**
This call is being removed from the Thread Manager.

**is_detached**
Read only.
Depth = 256

**Overview**
This call returns either a 1 (yes) or a 0 (no) indicated if the passed in thread ID is detached or not. In general it is assumed that the caller will only ask about existing threads, however if the thread is not used, a 0 is returned.

**Error Checks**

**Calls to Scheduler**

**Pseduo-Code**

```
attr = threads[childID]
if attr.used and attr.detached
    return 1
else
    return 0
```

**Return Value on Success**

1 if the thread ID is detached, 0 if the thread ID is either joinable, joined, or not used.

**Return Value on Error**

**join_thread**
Read only.
Depth = 256

**Overview**
This register is read to join the encoded thread ID(child) to the encoded current thread (parent). The child thread’s status is first checked to verify that it is, USED, NOT_JOINED, and NOT_DETACHED. If the test passes, the child’s status is checked to see if it has already exited. If it has, the value 0 + THREAD_ALREADY_TERMINATED is returned, else the child’s status is changed to joined, and a value of zero is returned. The calling thread, is removed from the
queue, via the scheduler's dequeue call. If the calling thread is HW, it is the HW thread responsibility to wait until it is set to RUN.

**Error Checks**

The child thread status must be USED, NOT_JOINED, NOT_DETACHED

The child thread has the passed in parentID listed as its parent.

**Calls to Scheduler**

dequeue(threadID)

removes the threadID from the queue.

**Pseduo-Code**

```plaintext
attrChild = threads[childID]
attrParent = threads[parentID]
if attrChild.USED and attrChild.NOT_JOINED, and attrChild.NOT_DETACHED
  if attrChild.parent == parentID
    if attrChild.EXITED
      return THREAD_ALREADY_TERMINATED
    else
      attrChild.JOINED = true
      dequeue(parentID)
      return 0
  else
    return error
else
  return error
```

**Return Value on Success**

0 or THREAD_ALREADY_TERMINATED

**Return Value on Error**

stuff

**next_thread**

Read only.

Depth = 1

**Overview**

This register is read to retrieve the ID of the next thread in the queue. The thread manager reads the ID off of the next_thread_id line and returns it immediately. The thread manager then calls the scheduler's dequeue. The scheduler removes the top thread ID from the queue. If the queue is empty, the scheduler makes no changes to the queue.

By a matter of principal, a hardware thread should never call next_thread.

**Error Checks**
Calls to Scheduler
dequeue()

Scheduler returns the value of the next thread to receive CPU time. Scheduler will return the value of the idle thread if there are no other threads scheduled to run.

Pseudo-Code

wait for next_thread_value == 1
threadID = next_thread_id
attr = threads[threadID]
if attr.used and !attr.exited
    return threadID
else
    return error

dqueue()

Return Value on Success

The ID of the thread to run next. The thread manager returns the thread ID in bits 23..30 with bit 31 set for all error conditions.

Return Value on Error

Returns an exception if the thread manager returns a value of a thread id that is either not used or exited.

que_length (REMOVE)

This call is being removed from the Thread Manager.

read_thread

read (write if in debug “stop” mode), depth = 256

Overview

Reading this register returns the encoded thread IDs row from the Thread ID Table without producing any side effects. The ERR_BIT and auxiliary status bits (E1, E2, E3) are returned as zeros. If the design is in debug, stopped mode, writing to register sets this row value to the data written.

Error Checks

Calls to Scheduler

Pseudo-Code

attr = threads(threadID)
return attr
**Return Value on Success**
Thread attributes for passed in thread ID.

Return Value on Error

**set_idle_thread (REMOVE)**
This call is being removed from the Thread Manager.

**current_cpu_thread (DEPRICATED)**
Read only.
Depth = 1

**Overview**
Returns the thread ID of the thread currently running on the CPU.
Assumes a single CPU system.
For a hardware thread, the HW should read its own THREADID register, and should not make a call to the thread manager to retrieve this value.

**Error Checks**

**Calls to Scheduler**

**Pseudo-Code**

```
return current_cpu_thread_id
```

**Return Value on Success**
The ID is returned in bits 23..30, with bit 31 set for all error conditions, else cleared.

**Return Value on Error**

**yield_thread**
Read only.
Depth = 1

**Overview**
This register is read to place the current SW thread back on the queue and then return the ID of the next thread in the queue. If the queue is empty, the current SW thread is not re-added to the queue, and its ID is returned instead.
By a matter of principal, this call should only be made by a SW thread.

**Error Checks**

Check the return value of dequeue() and enqueue(), to make sure the scheduler did not have an error during its operation.

**Calls to Scheduler**

dequeue()

Scheduler returns the value of the next thread ID to run.

enqueue(threadID)

Scheduler adds the passed in thread ID to the queue to eventually run.

isQueueEmpty()

Scheduler returns 1 if the queue is empty, else returns 0.

**Pseudo-Code**

```
attr = threads[threadID]
if ( attr.used and !attr.exited )
  if ( isQueueEmpty() )
    return current_cpu_thread
  else
    threadID = valueOf(nextThreadRegister)
    dequeue()
    enqueue(callingID)
    return threadID
else
  return ERROR_IN_STATUS
```

**Return Value on Success**

The ID of the thread to run next. The thread manager returns the thread ID in bits 23..30 with bit 31 set for all error conditions.

**Return Value on Error**

Returns ERROR_FROM_SCHEDULER, if either the dequeue or enqueue operations fail.

Returns ERROR_IN_STATUS if the current thread running on the CPU is not used and not exited.

**Troubleshooting Registers**

The following registers are included to enhance troubleshooting the system’s operation.

**exception_address**

Read only.
Depth = 1

Overview

Certain events cause a critical exception to be raised. These events should not occur during normal operation and are indicative of a failure within the system. When one of these events is detected, a code representing the particular type of event is stored in the exception_cause register and the associated address that was being accessed is stored in the exception_address register. The system can then read these registers to determine the reason for the interrupt. The causes and codes are listed in the exception_cause section.

Error Checks

Calls to Scheduler

Pseudo-Code

```c
return exception_address
exception_address = 0
```

Return Value on Success

Address of the register that caused the exception.

Return Value on Error

exception_cause

Read only.

Depth = 1

Overview

Certain events cause a critical exception to be raised. These events should not occur during normal operation and are indicative of a failure within the system. When one of these events is detected, a code representing the particular type of event is stored in the exception_cause register and the associated address that was being accessed is stored in the exception_address register. The system can then read these registers to determine the reason for the interrupt. The causes and codes are listed below.

- **EXCEPTION_ILLEGAL_STATE** (b 1111): The Thread Manager entered a state that is not recognized.
- **EXCEPTION_WRITE_TO_READ_ONLY** (b 0001): An attempt was made to write to a read only register.
- **EXCEPTION_UNDEFINED_ADDRESS** (b 0010): An attempt was made to access (either write or read) to a register that does not exist.
• **EXCEPTION_TO_SOFT_RESET** (b 0011): A timeout occurred while performing a soft reset.

• **EXCEPTION_TO_SCHD_NEXT_THREAD** (b 1000): A timeout occurred while waiting for the Scheduler to make a scheduling decision, and raise the sch2tm_next_id_value line.

• **EXCEPTION_TO_SCHD_ISQUEUED** (b 0100): A timeout occurred while performing an isQueued operation to the Scheduler.

• **EXCEPTION_TO_SCHD_ENQUEUE** (b 0101): A timeout occurred while performing an enqueue operation to the Scheduler.

• **EXCEPTION_TO_SCHD_DEQUEUE** (b 0110): A timeout occurred while performing an dequeue operation to the Scheduler.

• **EXCEPTION_TO_SCHD_ISEMPTY** (b 0111): A timeout occurred while performing an isQueueEmpty operation to the Scheduler.

• **EXCEPTION_SCHD_INVALID_THREAD** (b 1001): The scheduler returned a thread ID, on the sch2tm_next_id lines that is either not used or exited.

**Error Checks**

**Calls to Scheduler**

**Pseudo-Code**

```
return exception_address
exception_address = 0
```

**Return Value on Success**

Address of the register that caused the exception.

**Return Value on Error**

**soft_reset**

Read or write.

Depth = 1

**Overview**

Writing to this register will reset the selected hardware components of the hybrid-threads system. Individual components may be reset, based on the value that is written. Each bit corresponds to a particular hardware component. They are as follows:

• Bit 27: All hardware threads.

• Bit 28: Spin Lock.

• Bit 29: Semaphores.
- Bit 30: Scheduler.
- Bit 31: Thread Manager.

Reading this register returns all zeros with a one in any position corresponding to an IP that failed to signal completion before an encoded time delay (default delay = 4096 clock cycles). The bit position are the listed in the above bullet list.

**Error Checks**

**Calls to Scheduler**

**Pseudo-Code**

```plaintext
#On write
reset all internal and external signals of the thread manager
reset thread table
send signal to reset each intellectual property core

#On read
return reset_status
```

**Return Value on Success**

On write, there is no return.

On read, returns 1, in the correct bit position, of each intellectual property core that did not reset correctly. Returns a 0 in that position, if it did reset correctly.

**Return Value on Error**

On read, returns 1, in the correct bit position, of each intellectual property core that did not reset correctly. Returns a 0 in that position, if it did reset correctly.

**soft_start**

Read or write.

Depth = 1

**Overview**

Writing to this register de-asserts the soft_stop and all soft_reset signals. The soft_stop signal is used by all system intellectual properties to halt operation. De-asserting the signal, allows for the eventual restarting of the system. The soft_reset group of signals, is used to ask the various intellectual properties of the system to reset themselves.

Reading this register returns all zeros. I have to admit, I am left wondering why the read is even here. ’Cause, yeah, it basically does nothing. Use at your own pleasure.
Error Checks

Calls to Scheduler

Pseudo-Code

#On Write
soft_reset = 0
reset_done = 0
core_stop = 0

#On Read
return 0

Return Value on Success
On write, there is no return.
On read, 0.

Return Value on Error

soft_stop
Read or write.
Depth = 1

Overview
Writing to this register asserts the soft_stop signal. The soft_stop signal is used by all system intellectual properties to halt operation. In other words, writing to this register stops the hybrid-thread hardware components. Any value may be written to this register to initiate the stop.

Reading this register returns the value of the soft_stop signal in the LSB, all other bits are zero.

Error Checks

Calls to Scheduler

Pseudo-Code

#On Write
core_stop = 1

#On Read
return core_stop
Return Value on Success

On write, there is no return value.

On read, a 1 indicates the hybrid-thread cores are stopped, a 0 indicates they are not stopped.

Return Value on Error

Scheduler Interface

The Thread Manager and the Scheduler communicate through a dedicated interface. This interface is detailed below.

There are eight signals between the Thread Manager and the Scheduler. Four signals going in each direction.

Thread Manager to Scheduler Signals

tm2sch_opcode (6 bits): The opcode tells the scheduler what function to perform. There are six valid values.

- no_op (0b00 0000): Thread Manager is not asking for any information.
- is_queued (0b00 0001): Thread Manager is asking if the thread id set on the data line is currently queued by the scheduler. Scheduler returns “0x0000 0001” on the data_return line if the thread is queued, else scheduler will return “0x0000 0000.”
- enqueue (0b00 0010): Thread Manager is asking for the thread id set on the data line to be added to the queue.
- dequeue (0b00 0011): Thread Manager is asking the scheduler to remove the top thread id from the queue.
- is_queue_empty (0b00 0110): Thread Manager is asking if the queue in the scheduler is empty. Scheduler returns a 1 if the queue is empty, else it returns 0.

tm2sch_data (32 bits): Certain opcodes require additional information to be passed to the scheduler. This information is passed on the tm2sch_data signal. The Thread Manager will place this data on the tm2sch_data lines prior to setting tm2sch_request high. The Thread Manager will keep tm2sch_data set, until the Scheduler transitions the sch2tm_busy signal from high to low (indicated an end of the request).

tm2sch_request (1 bit): When the Thread Manager is making a request to the scheduler, the tm2sch_request line will go high (read 1). This is for all opcodes except no_op. The Thread Manager will keep tm2sch_request high until the scheduler acknowledges by setting the sch2tm_busy signal high. At all other times, Thread Manager will keep tm2sch_request low (read 0).

tm2sch_cpu_thread_id (8 bits): The Thread Manager will keep this signal set to the value of its current_cpu_thread_id register. This value will be the thread id that is currently running on the cpu. It will not contain any information regarding hardware threads.
tm2sch_DOB (32 bits): The data out lines, for the “B” port of the Thread Manager's block ram.

Scheduler to Thread Manager Signals

sch2tm_busy (1 bit): The scheduler will keep the sch2tm_busy signal low (read 0), until it responds to a request from the Thread Manager. The Thread Manager sets tm2sch_request high when it is requesting information from the Scheduler through an opcode. Scheduler will set sch2tm_busy high (read 1) to acknowledge the request. It will keep sch2tm_busy high until it is done performing the request. If information is passed back to the thread manager, the scheduler will set this information on the sch2tm_data lines prior to setting busy low.

sch2tm_data_return (32 bits): Certain opcodes require the scheduler to return information to the Thread Manager. For these opcodes, the scheduler returns the needed information on the sch2tm_data signal. The Scheduler will place the data on this signal prior to setting the sch2tm_busy signal low. The Scheduler will keep the data on the signal until the next request by the Thread Manager.

sch2tm_next_id (8 bits): The scheduler will set the value of the thread id that is to run next (after a yield call). The value is only valid when sch2tm_next_id_valid is high.

sch2tm_next_id_valid (1 bit): When this signal is high, the Scheduler is indicating that the value on sch2tm_next_id is valid. A low value indicates that the value on sch2tm_next_id can not be trusted.

sch2tm_ADDRB (9 bits): The address lines for the “B” port of the Thread Manager's block ram.

sch2tm_DIB (32 bits): The data in lines, for the “B” port of the Thread Manager's block ram.

sch2tm_ENB (1 bit): The enable line, for the “B” port of the Thread Manager's block ram.

sch2tm_WEB (1 bit): The write enable line, for the “B” port of the Thread Manager's block ram.
Thread ID Table Encoding

The Thread Manager uses 256 rows of BRAM. Each row is split between the used thread id stack (bits 0 through 7) and the state of each thread (bits 16 through 31). There is also 8 unused bits (bits 8 through 15). As a note, for any given row, the value of the thread id stack is not related to the value of the state of the thread.

The general encoding for the Thread ID Table is as follows:

<table>
<thead>
<tr>
<th>0</th>
<th>7</th>
<th>8</th>
<th>15</th>
<th>16</th>
<th>23</th>
<th>24</th>
<th>25</th>
<th>26</th>
<th>27</th>
<th>28</th>
<th>29</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread ID</td>
<td>(not used)</td>
<td>PID</td>
<td>D0</td>
<td>J0</td>
<td>U0</td>
<td>X0</td>
<td>E3</td>
<td>E2</td>
<td>E1</td>
<td>ERR_BIT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Thread ID Stack

The thread manager maintains a stack of available thread ids. These are the thread Ids that could be returned to the CPU from a create_thread call. The first eight bits of the thread ID table is used to keep the stack.

Thread State

The state of the thread is maintained in bits 16 through 31 of the thread ID table. The index into the table is the thread id. For example, index 0 is for thread id 0, index 1 is for thread id 1, and so on. The meaning of each bit of a row is listed below:

**Parent ID (PID)**

Bits 16 through 23

This is the ID of the parent of this thread.

**Detached**

Bit 24

A “1” indicates this thread was created as a detached thread., a “0” indicates this thread is not detached.

**Joinable**

Bit 25

If the detached bit is 0 (thread is not detached), a “0” indicates this thread is joinable, a “1” indicates this thread is joined. If the detached bit is 1 (thread is detached), the joinable bit has no meaning.

**Used**

Bit 26

A “1” indicates this thread is used, a “0” indicates this thread is not used.
Exited
Bit 27
A “1” indicates this thread is not exited, a “0” indicates this thread is exited.

Error Value
Bits 28 through 30
The meaning of the error bits is as follows. Some error values are call dependent.

0b000:
- ERROR_IN_STATUS: Error in thread PID, error in thread status, or no Ids available.
- JOIN_ERROR_UNKNOWN: Could not join a thread for an unknown reason.

0b001:
- THREAD_ALREADY_TERMINATED: Thread is already terminated, returned during the join thread call. This error may be interpreted as a warning, it does not indicate that anything went wrong.

0b010:
- THREAD_ALREADY_QUEUED: Can not queue the thread because it is already queued.

0b011:
- ERROR_FROM_SCHEDULER: The scheduler could not successfully complete a requested operation.

0b100:
- JOIN_ERROR_CHILD_JOINED: Trying to join on a thread that is already joined.

0b101:
- JOIN_ERROR_NOT_CHILD: Trying to join on a thread, where the current CPU thread is not the parent of the thread it is trying to join on.

0b110:
- JOIN_ERROR_CHILD_DETACHED: Trying to join on a detached thread.

0b111:
- JOIN_ERROR_CHILD_STATUS: Trying to join on a thread that is not used.

Error Condition
Bit 31
A “1” indicates there was an error during the method call, check the error value for the specific error. A “0” indicates the method call completed successfully.
Thread Manager Block Diagram

Changes from Previous Version
This section summarizes the changes to the Thread Manager API from the previous version.

Removed or Deprecated Calls
• queue_length: This call is removed. The Thread Manager no longer manages the queue. Queue management, along with accessing attributes of the queue, are now entirely done via the Scheduler.
• idle_thread: This call is removed. The Thread Manager no longer manages the idle thread. The idle thread is not set and retrieved via the Scheduler.
• current_thread: This call is deprecated, it may not be available in future versions of the Thread Manager.
• detached_thread: This call is removed. If a thread is to be detached, it should be created as detached.
Modified Calls

- **exit_thread**: This call's depth changed from 1 to 256. The calling thread, regardless if it is hardware or software, must encode its thread ID. The Thread Manager no longer assumes this call was made by the thread running on the CPU.

New Calls

- **is_detached**: This call returns either a 1 (yes) or a 0 (no) indicated if the passed in thread ID is detached or not. In general it is assumed that the caller will only ask about existing threads, however if the thread is not used, a 0 is returned.