*Application Report*
*SPRAA25 – May 2004*

# *Synchronizing DSP/BIOS Threads*

*Prashanth L A, Eldo Tony Kuruvilla* *Software Development Systems*

**ABSTRACT**

DSP programming has transitioned from single loop programs to complex multi-threaded applications that utilize real-time features. Multi-threaded applications need to synchronize access to shared resources.

DSP/BIOS provides multiple mechanisms for synchronizing threads. The mechanisms provide ways to disable thread types, change thread priorities, and provide for mutual exclusion through semaphores and locks.

The mechanisms you choose depend upon the types of threads you need to synchronize. This application note describes various DSP/BIOS thread synchronization primitives and related issues and constraints.

DSP/BIOS is a trademark of Texas Instruments.

**Contents**

**Figures**

**Tables**

## 1 Issues for Thread Synchronization

DSP/BIOS enables you to structure applications as a collection of threads, each of which carries out a modularized function. Thread use OS services—based on semaphores, mutex, interrupt protection, and changing thread priorities—to achieve synchronization.

In real time applications, multiple threads require access to common resources such as queues, shared variables, and lists. Critical sections are used to access these common resources without any conflicts. Techniques such as semaphores, locks, and disabling threads can protect critical sections.

An application programmer must ensure that critical sections have the following attributes:

- **Safety.** At most one thread may be executing during the critical section. That is, there must be mutual exclusion for access to shared resources.
- **Liveness.** If one or more threads run the section that performs mutual exclusion, eventually at least one of them enters the critical section. That is, no deadlock occurs.

The different DSP/BIOS thread types and the synchronization mechanisms that can be associated with each are discussed in detail in this application note. These mechanisms fall into three general categories:

- Disabling Threads, page 5
- Changing Thread Priority, page 11
- Mutual Exclusion, page 14

Before discussing mechanisms in detail, let's first review some issues related to thread synchronization—the priority levels provided by the DSP/BIOS thread types, and how problems can occur in critical sections.

## 1.1 Thread Priority and Preemption

DSP/BIOS supports several types of program threads with different priorities. Execution and preemption characteristics of a thread depend upon its type. Figure 1 shows the different DSP/BIOS thread types in order from highest to lowest priority.

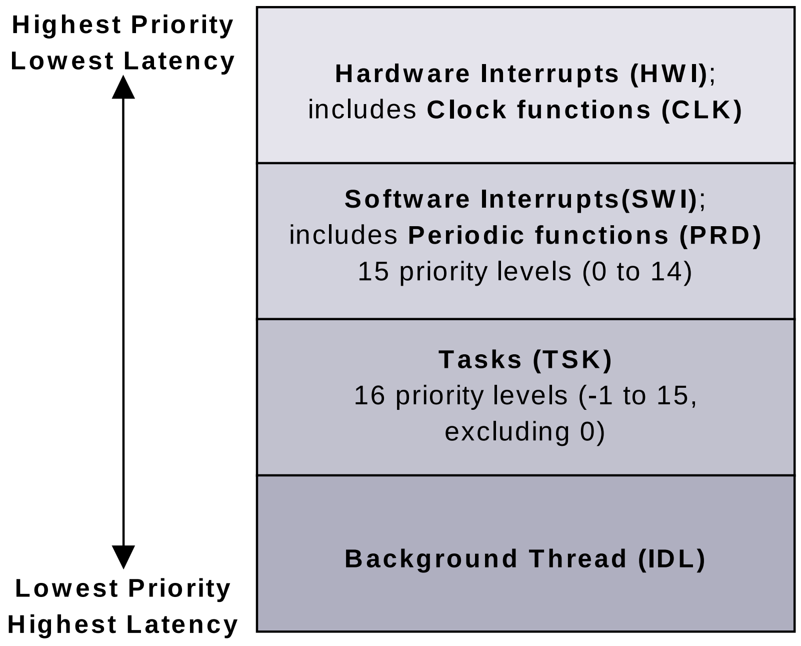

Highest Priority
Lowest Latency
Hardware Interrupts (HWI); includes Clock functions (CLK)
Software Interrupts(SWI); includes Periodic functions (PRD) 15 priority levels (0 to 14)
Tasks (TSK) 16 priority levels (-1 to 15, excluding 0)
Background Thread (IDL)
Lowest Priority
Highest Latency

**Figure 1. DSP/BIOS Thread Priorities**

The thread types (from highest to lowest priority) are:

- Hardware interrupts (HWI), includes CLK functions
  - have the highest priorities
  - one HWI can interrupt another
- Software interrupts (SWI), includes PRD functions
  - have lower priority than hardware interrupts
  - 14 sub-levels of priority
  - can be preempted by a higher priority SWI or HWI
- Tasks (TSK)
  - have lower priority than software interrupts
  - can be preempted by a higher priority task
  - have 16 sub-levels of priority
  - If priority is less than 0, the task is barred from execution until its priority is raised at a later time by another thread. A priority value of 0 is reserved for the TSK_idle task defined in the default configuration.Background thread (IDL)
  - Has the lowest priority level and can be preempted by a TSK, SWI or HWI

Thread priorities impose a synchronization mechanism that protects higher-priority threads. For example, HWI threads are protected against SWIs and TSKs. Similarly, SWI threads are protected against TSKs. In other words, a critical section in a HWI thread need not be protected against SWI/TSK interventions.

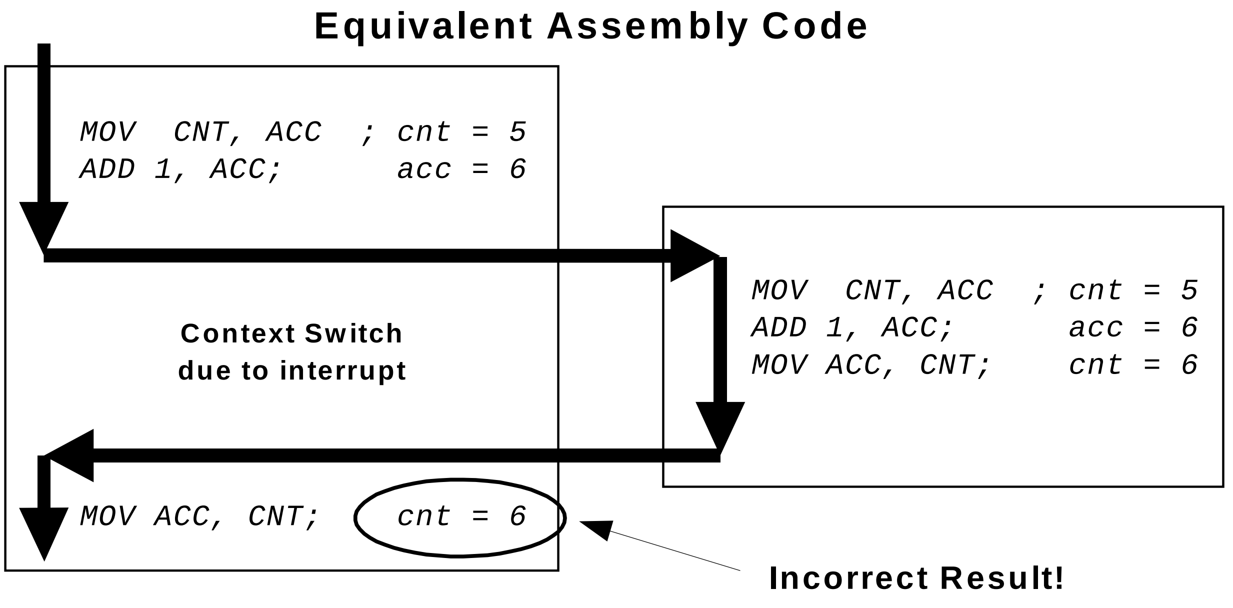
## 1.2 Race Conditions and Critical Sections

Race conditions can occur in an application when multiple threads compete for the same resource at the same time. The cause of race conditions typically is an unsynchronized access to shared resources. These are annoyingly difficult to detect. Figure 2 illustrates a simple race condition arising out of unsynchronized access to a shared variable “cnt”. The threads here may be HWIs, SWIs or TSKs.

In the code in Figure 2, two threads have different priorities. Both increment the same count variable (cnt). The low_pri_thread() has lower priority than high_pri_thread().

```
Void low_pri_thread(Void)
{
   // begin critical section
   cnt = cnt + 1;
   // end critical section
}
```

```
Void high_pri_thread(Void)
{
   // begin critical section
   cnt = cnt + 1;
   // end critical section
}
```



```
MOV  CNT, ACC  ; cnt = 5
ADD 1, ACC;      acc = 6
```

```
MOV  CNT, ACC  ; cnt = 5
ADD 1, ACC;      acc = 6
MOV ACC, CNT;    cnt = 6
```

```
MOV ACC, CNT;    cnt = 6
```

**Figure 2. Illustration of Race Condition**

The example in Figure 2 generates an incorrect result. The initial value of the *cnt* variable is 5 and its value should be 7 after both threads increment it. However, because of thread pre-emption the final *cnt* value is 6 instead of 7.

To avoid such situations, access to shared resources needs to be performed atomically. Such atomic access to shared resources constitutes a critical section. The three instructions that increment *cnt* should be treated as a "critical section" in this example.

Figure 2 uses a simple “cnt” variable to illustrate the race condition. In real-time systems, shared resources such as queues and shared data structures are used.

Context switches occur between different threads. To achieve criticality, various synchronization primitives need to be used. When synchronizing threads, you should be aware of interdependencies and should deploy OS synchronization services carefully so that they do not lead to deadlocks, high interrupt latency, and other problems.

## 2 Disabling Threads

To protect a critical section in a multi-threaded application, you can disable all threads except the thread currently executing before entering the critical section. This prevents the running thread from being preempted by other threads while executing the critical section. After the critical section is complete, you can re-enable all threads. This enabling and disabling of threads can be accomplished using DSP/BIOS API calls.

Table 1 summarizes the API calls used for disabling various thread types. These APIs are discussed in more detail in the sections that follow.

**Table 1. Thread Scheduling Status for Disable API Calls**

| | **Hardware Interrupts (HWI)** | **Software Interrupts (SWI)** | **Tasks (TSK)** |
|---|---|---|---|
| HWI_disable | Disabled | Disabled | Disabled |
| SWI_disable | Enabled | Disabled | Disabled |
| TSK_disable | Enabled | Enabled | Disabled |

Table 2 shows the number of CPU cycles involved in executing each API call, its calling context and additional information.

**Table 2. Summary of Disable/Enable APIs**

| | **C62x Cycles** | **Calling Context** | **Notes** |
|---|---|---|---|
| HWI_disable/enable | 12/12 (24)* | HWI, SWI, TSK | This block is best suitable for short critical sections. Fast and fewer cycles. |
| SWI_disable/enable | 24/64 (88)* | SWI, TSK | Nested calls are supported. |
| TSK_disable/enable | 64/104 (168)* | TSK | Nested calls are supported. |

* For more information on timing numbers, see “DSP/BIOS Timing Benchmarks for CCS 2.2” (SPRA 900)

## 2.1 Disabling and Enabling HWIs

Recall from Section 1.1, Thread Priority and Preemption, that higher-priority threads can preempt lower-priority threads. Disabling hardware interrupt (HWI) threads protects all thread types against preemption by HWIs.

A thread can use DSP/BIOS API calls to disable all hardware interrupts before entering the critical section. After completing the critical section, HWIs can be re-enabled. Hardware interrupts can be disabled and re-enabled using the following APIs:

- **HWI_disable.** Disables interrupts globally. On the C6000 platform, HWI_disable clears the GIE bit in the control status register (CSR). On the C5000 and C2800 platforms, HWI_disable sets the INTM bit in the ST1 register. On both platforms, this prevents the CPU from taking any maskable hardware interrupt.
- **HWI_enable.** Renables the GIE bit on the C6000 platform or clears the INTM bit in the ST1 register on the C5000 and C2800 platforms.
- **HWI_restore.** Restores the value to the state that existed before HWI_disable was called.

The following example uses HWI_disable and HWI_restore. Before disabling interrupts, the current interrupt status information is captured in the key", which is later used for restoration.

```
key = HWI_disable();
cnt = cnt+1;
HWI_restore(key);
```

### *Issues to Consider*

The following issues need to be considered when using HWI_disable, HWI_enable, and HWI_restore:

- All thread types (HWIs, SWIs, and TSKs) can call HWI_disable and HWI_enable/HWI_restore.
- While HWIs are disabled, no DSP/BIOS calls that exercise the scheduler should be made. This includes calls that may block and calls that post other threads. Such calls typically require that HWI interrupts be enabled or that the timeout for blocking be 0.

  The following example illustrates the problem that can occur if SWI_post or SEM_post is called within a critical section where HWI interrupts are disabled.

  ```
  key = HWI_disable();
  cnt = cnt+1;
  SWI_post(&SWI0);
  HWI_restore(key);
  ```

  SWI_post has a pre-condition that hardware interrupts should be enabled (that is, intm=0 or GIE=1) if invoked outside the context of an ISR. There is a clear violation of this constraint in line 3 of this example, since HWI_disable sets the intm or clears GIE.

  Also, SWI_post or SEM_post calls in a critical section could cause another thread to run with interrupts disabled. This is especially undesirable if that thread happens to be a TSK, since interrupts can be disabled for a long time.

- HWI_enable enables interrupts regardless of the previous context. For example:

  ```
  HWI_disable();
  cnt = cnt+1;
  HWI_enable();
  ```

  In this example, it is possible that interrupts were already disabled prior to the call to HWI_disable. But, the call to HWI_enable enables interrupts without taking into consideration the previous context. Hence, it is recommended that you use HWI_restore instead of HWI_enable, since HWI_restore restores the intm/GIE value to the state that existed before HWI_disable was called.

- Nesting of HWI_disable and HWI_enable calls is not supported. As explained previously, HWI_enable enables interrupts irrespective of the previous context, and hence nesting of HWI_disable/HWI_enable is not possible.

- Interrupt latency is directly affected when disabling interrupts. It is advisable to keep critical sections as short as possible to avoid long delays for servicing other interrupts.

## 2.2 Using IER and IMR Masks

The interrupt enable register (IER) determines whether or not the CPU responds to interrupts. By specifying the masks, other HWIs can be disabled while a critical section is being executed. IER masks operate by affecting individual bits in the interrupt enable register, as opposed to affecting all interrupts on a global basis by setting the intm or clearing the GIE.

Masks can be set up for individual interrupts. On a 'C54x target such as the 'C5402, the mask for the timer interrupt (TINT) can be defined as:

```
#define   TINTMASK  0x0008
```

After defining the IMR mask, the C54_disableIMR/C54_enableIMR APIs can be called to protect a critical section. The following example illustrates the usage of these APIs.

```
oldmask = C54_disableIMR(mask);
cnt = cnt + 1;
```

The masking functions for different ISAs are as follows:

**Table 3. Interrupt Masking Functions for Different ISAs**

| | TMS320C54x | TMS320C55x | TMS320C28x | TMS320C62x |
|---|---|---|---|---|
| Disable certain interrupts | C54_disableIMR(mask) | C55_disableIER0(mask) C55_disableIER1(mask) | C28_disableIER(mask) | C62_disableIER(mask)/ C64_disableIER(mask) |
| Enable certain interrupts | C54_enableIMR(oldmask) | C55_enableIER0(oldmask) C55_enableIER1(oldmask) | C28_enableIER(oldmask) | C62_enableIER(oldmask)/ C64_enableIER(oldmask) |

### Issues to Consider

The following issues need to be considered when calling C*xx*_disableIER and C*xx*_enableIER:

- All thread types (HWIs, SWIs, and TSKs) can call C*xx*_disableIER and C*xx*_enableIER.
- If you use these APIs, you need to be aware of interdependencies between threads. For example, suppose T1 is a TSK that calls C54_disableIMR and C54_enableIMR. Suppose that both T1 and T2 (another TSK thread) share the *cnt* variable. In T1, the interrupt that invokes or wakes up T2 needs to be masked to synchronize access to *cnt*. This prevents a context switch to T2 during the critical section, and makes access to the shared variable atomic.
- While interrupts are disabled, DSP/BIOS calls that can cause a context switch to a thread that shares the protected resource should not be made. Inside the Cxx_disableIER/ Cxx_enableIER block, calls to SWI_post, SEM_post, TSK_yield, or other APIs that cause other threads to run can violate the critical section constraints. This is illustrated in the following example:

  ```
  oldmask = C54_disableIMR(mask);
  cnt = cnt+1;
  SWI_post(&SWI0);
  C54_enableIMR(mask);
  ```

  In this example, SWI_post can cause a context switch to SWI0. This can cause unsynchronized access to the shared variable, if *cnt* happens to be shared with SWI0. Another problem with calling a scheduling API with interrupts disabled is the thread that runs as a result of the scheduling API is run with interrupts disabled. This is especially undesirable if that thread is a TSK, since interrupts can be disabled for a long time.
- C*xx*_disableIER is used to disable selected interrupts, allowing other interrupts to occur. However, if another interrupt occurs during this region, it could cause a context switch to a SWI or TSK. You can prevent this by using SWI_disable and SWI_enable around the entire region. The context switch will eventually occur after the SWI_enable call.
- It is advisable to keep critical sections as small as possible to avoid long delays for servicing other interrupts.

## 2.3 Using HWI_enter/HWI_exit and the HWI Dispatcher

Logical priorities among HWI threads can be achieved using the IER/IMR masks of the HWI_enter/HWI_exit macros or the HWI dispatcher. A brief description of these is as follows:

- **HWI_enter.** An API (assembly macro) used to save the appropriate context for a DSP/BIOS interrupt service routine (ISR).
- **HWI_exit.** An API (assembly macro) used to restore the context that existed before a DSP/BIOS interrupt service routine (ISR) was invoked.
- **HWI Dispatcher.** Provides an easy way to write ISR in C without the application having to save and restore context.

As described in Section 2.2, "Using IER and IMR Masks", other HWIs can be disabled while a section of code is being executed by specifying the IER/IMR masks.

The following code uses the HWI_enter and HWI_exit macros on a C54x target to achieve logical priorities among HWI threads.

```
HWI_enter MASK,IMRDISABLEMASK
; isr code
HWI_exit MASK,IMRRESTOREMASK
```

Support ISR1 is the HWI thread that invokes HWI_enter and HWI_exit. ISR1 shares *cnt* with ISR2. In ISR1, the IMRDISABLEMASK of HWI_enter masks ISR2. This prevents a context switch to ISR2 within the critical section. Hence, access to the shared variable is atomic.

It is important to note that HWI_enter and HWI_exit must be used at the beginning and end of the entire ISR code. They cannot be used to protect only the critical section of the ISR code. At the end of the ISR, interrupts are re-enabled using the IMRRESTOREMASK of the HWI_exit macro.

HWI objects can be configured to use the dispatcher in the configuration file (.cdb). Criticality can be achieved by specifying the interrupt masks (IER/IMR) in the HWI object properties in the configuration file. The HWI dispatcher, in effect, calls the function for that HWI object from within an HWI_enter/HWI_exit macro pair, thus disabling interrupts specified in the masks before calling the function and re-enabling them before exiting the function.

### *Issues to Consider*

The following issues need to be considered when invoking the HWI_enter and HWI_exit macros or using HWI dispatcher:

- Only HWI threads can invoke the HWI_enter/HWI_exit macros or use the HWI dispatcher.
- If you use these APIs, you need to be aware of interdependencies between threads. In particular, be aware of all interrupts that could possibly violate the critical section contained in the ISR and mask these interrupts using the IER/IMR masks of the HWI_enter/HWI_exit macros or in the HWI dispatcher configuration.
- HWI_enter and HWI_exit cannot be used to protect only a portion of the code in an ISR. The following example does not achieve criticality and may even lead to fatal errors, since nesting of HWI_enter/HWI_exit code is not possible within a single ISR.

  ```
  HWI_enter MASK,IMRDISABLEMASK
  ;rest of the isr code

  HWI_enter MASK,IMRDISABLEMASK
  ; critical section code
  HWI_exit  MASK,IMRRESTOREMASK

  ;rest of the isr code
  HWI_exit  MASK,IMRRESTOREMASK
  ```

- The HWI_enter/HWI_exit macros and HWI dispatcher disable selected interrupts, allowing other interrupts to occur. If another interrupt occurs during the protected region, it could cause a context switch to a SWI or TSK. You can prevent this by using SWI_disable and SWI_enable around the entire region. The context switch will eventually occur after SWI_enable call.

- Interrupt latency is directly affected when disabling interrupts. It is advisable to keep critical sections as small as possible to avoid long delays for servicing other interrupts.

## 2.4 Disabling and Enabling SWIs

Section 2.1, "Disabling and Enabling HWIs" showed how to disable HWI threads to protect critical sections. The principle of disabling threads applies to SWI threads also. Preemption by SWI threads can be prevented using the SWI_disable and SWI_enable APIs. SWI_disable disables all other SWI threads from running until SWI_enable is called. HWI threads can still run.

Use of the SWI_disable and SWI_enable calls to protect critical sections is similar to the HWI_enable/HWI_disable calls. This is illustrated in the following example:

```
SWI_disable();
cnt = cnt+1;
SWI_enable();
```

### *Issues to Consider*

The following issues need to be considered when calling SWI_disable and SWI_enable:

- SWI_disable or SWI_enable cannot be called within a HWI thread.
- SWI_disable/SWI_enable calls don't protect a critical section from being preempted by HWIs.
- SWI_disable and SWI_enable calls can be nested. SWI_disable must be followed by a matching number of SWI_enable calls. Only the outermost SWI_enable call actually enables software interrupts.
- The scheduling requested by calls such as SEM_post or TSK_yield within a SWI_disable/ SWI_enable block are performed only after SWI_enable.
- A SEM_pend call within a SWI_disable/SWI_enable block immediately returns with a FALSE return value if the semaphore is not available, even if the timeout is non-zero or SYS_FOREVER.

## 2.5 Disabling and Enabling TSKs

Similar to HWI and SWI threads, critical section can be protected using TSK_disable and TSK_enable calls. TSK_disable disables the DSP/BIOS task scheduler. The current task continues to execute until TSK_enable is called.

Use of the TSK_disable and TSK_enable calls is similar to HWI/SWI enable/disable calls. This is illustrated in the following example:

```
TSK_disable();
cnt = cnt+1;
TSK_enable();
```

***Issues to Consider***

The following issues need to be considered when calling TSK_disable and TSK_enable:

- TSK_disable/TSK_enable don't protect a critical section execution from being preempted by HWIs and SWIs.
- No kernel operations that can cause the current task to block can be made within a TSK_disable/TSK_enable block. This includes SEM_pend (unless timeout is 0), TSK_sleep, and TSK_yield.
- TSK_disable/TSK_enable cannot be called from a SWI or HWI.
- TSK_disable and TSK_enable calls can be nested. Task switching is not reenabled until TSK_enable has been called as many times as TSK_disable.
- TSK_disable can prohibit ready tasks of higher priority from running since the task scheduler is disabled within the TSK_disable/enable block. This latency can be prevented using semaphores, which are discussed in Section 4.1, "Mutual Exclusion Using Semaphores".
- Scheduling calls such as SEM_post and TSK_yield within the TSK_disable/TSK_enable block are taken only after TSK_enable.

# 3 Changing Thread Priority

In the previous sections, disabling threads was discussed as a method of synchronization. In this section, we discuss changing thread priorities to protect critical sections. This method is based on the fact that the priorities of SWI and TSK threads can be changed at run-time to prevent their preemption while executing a critical section.

You can make the currently executing thread the highest priority thread of its type in the application before entering a critical section. After executing the critical section, you restore the thread's priority to its original level.

### 3.1 Changing SWI Priority

SWI priorities can be modified using the SWI_raisepri and SWI_restorepri APIs:

- **SWI_raisepri.** Raises the priority of the currently running SWI to the priority passed as the argument.
- **SWI_restorepri.** Restores the priority of the SWI to the priority prior to the SWI_raisepri call.

The following example uses SWI_raisepri and SWI_restorepri:

```
/* Raise priority of current SWI to that of highest priority SWI in the application */
mask = SWI_getpri(&highest_pri_swi);

key = SWI_raisepri(mask);
cnt = cnt + 1;
SWI_restorepri(key);
```

In this example, SWI_raisepri sets the current SWI to the highest priority used in the application before entering the critical section. Before raising the priority of the SWI, the current priority mask is captured in the “key”. After executing the critical section, the SWI’s priority is restored to the original level using the “key”.

#### *Issues to Consider*

The following issues need to be considered when calling SWI_raisepri and SWI_restorepri.

- The SWI_raisepri/SWI_restorepri calls can be made only from the context of a SWI. They cannot be called from HWI or TSK threads.
- SWI priorities may be changed during the maintenance lifetime of an application. Hence, this method for achieving mutual exclusion is dangerous, as later changes in thread priorities can break the mutual exclusion.
- Use of SWI_raisepri requires that you know the highest priority SWI in the system. Otherwise, you can raise a SWI executing a critical section to the highest priority possible (priority 14).
- This method cannot protect a critical section from HWIs. HWI threads have higher priority than SWI threads, and hence can break the mutual exclusion.
- SWI_raisepri cannot lower the priority of a SWI.

### 3.2 Changing Task Priority

In the previous section, changing the SWI thread priority to protect shared resources was seen. The principle of changing thread priority applies to TSK threads also. Criticality is achieved here by using TSK_setpri/TSK_create APIs.

- **TSK_create.** Creates a new task object and makes it ready for execution. The task’s execution priority attribute can be specified.
- **TSK_setpri.** Sets the execution priority of a task and returns that task's old priority value.

A task can be created dynamically as in the following example.

```
TSK_Attrs attrs;
TSK_Handle task;

attrs = TSK_ATTRS; /* Default task attributes */
attrs.priority = PRIORITY;

task = TSK_create(func, &attrs);
```

The following example uses TSK_setpri to protect a critical section.

```
Uns oldpri;

oldpri = TSK_setpri(TSK_self(), newpri);
cnt = cnt + 1;               /* Critical Section */
TSK_setpri(TSK_self(), oldpri);
```

As done previously with SWI threads, TSK_setpri makes the current task the highest priority task in the application before entering the critical section. Before raising the priority of the current task, the priority mask is captured in “oldpri”. After executing the critical section, the current task’s priority is restored to its original level using “oldpri”.

### *Issues to Consider*

The following issues need to be taken considered when using TSK_setpri:

- This method cannot protect critical sections from SWIs and HWIs. Hence, from the perspective of thread synchronization, the TSK_setpri call can only be used to synchronize access to a resource shared by task threads.
- Inside the critical section, calls to SWI_post or SEM_post or TSK_yield can violate the critical section constraints. This is shown in the following example:

  ```
  oldpri = TSK_setpri(TSK_self(), newpri);
  cnt = cnt + 1;               /* Critical Section */
  SWI_post(&SWI0)
  TSK_setpri(TSK_self(), oldpri);
  ```

  In this example, the call to SWI_post causes a context switch to SWI0. This can cause unsynchronized access to the shared variable cnt, if cnt is shared with SWI0. Hence, critical sections cannot contain DSP/BIOS API calls that can cause a context switch to a thread with which the variable is shared.
- The new priority should not be zero (0). This priority level is reserved for the TSK_idle task.
- As with SWI threads, TSK priorities may be changed during the maintenance lifetime of an application. Hence, this method for achieving mutual exclusion is dangerous, as later changes in thread priorities can break the mutual exclusion.
- Use of TSK_setpri requires that you know the highest priority TSK in the system. Otherwise, you can raise a TSK executing a critical section to the highest priority possible (priority 15).

## 4 Mutual Exclusion

DSP/BIOS provides structures like semaphores and locks to support synchronization in a multi-threaded environment. The following sections demonstrate the use of semaphores and locks for achieving mutual exclusion (mutex).

Table 4 shows the number of CPU cycles involved in executing the API calls discussed in this section, their allowed calling contexts, and some additional information.

**Table 4. Synchronization APIs Summary**

| | C62x Cycles | Calling Context | Notes |
|---|---|---|---|
| SEM_pend/post | 228/264 (492)* | TSK, SWI, HWI | Deadlocks and priority inversion could occur in the program. |
| LCK_pend/post | 252/296 (548)* | TSK | Fixes the recursive deadlock problem. Priority inversion could still occur. |

* For more information on timing numbers, see "DSP/BIOS Timing Benchmarks for CCS 2.2" (SPRA 900)

### 4.1 Mutual Exclusion Using Semaphores

SEM objects are counting semaphores that can be used for task synchronization and mutual exclusion. Counting semaphores keep an internal count of the number of corresponding resources available. When the count is greater than 0, tasks do not block when acquiring a semaphore.

SEM objects can be created statically using the Configuration Tool or dynamically using the SEM_create API call. To create a mutex semaphore, the SEM must be created with an initial count of one. The following APIs are used to achieve mutual exclusion:

- **SEM_pend.** If the semaphore count is greater than zero, SEM_pend decrements the count and returns TRUE. Otherwise, SEM_pend suspends the execution of the current task until SEM_post is called or the timeout expires.
- **SEM_post.** Readies the first task waiting for the semaphore. If no task is waiting, SEM_post simply increments the semaphore count and returns.

The following example uses SEM_pend and SEM_post to achieve mutual exclusion.

```
Void task0(void)
{
    SEM_pend(mutex, SYS_FOREVER);
    'Shared resource/critical section'
    SEM_post(mutex);
}
```

```
Void task1(void)
{
    SEM_pend(mutex, SYS_FOREVER);
   'Shared resource/critical section'
    SEM_post(mutex);
}
```

In this example, the mutex semaphores were created with an initial count of 1. Each task calls SEM_pend before entering the critical section. If the mutex count is 1, SEM_pend changes the count from 1 to 0 to acquire a mutex. If the mutex count is 0, SEM_pend blocks execution to wait for the mutex. After executing the critical section, the tasks call SEM_post, which changes the count from 0 to 1 to release the mutex.

***Issues to Consider***

The following issues need to be considered when calling SEM_pend and SEM_post.

- This method cannot protect critical sections from SWIs and HWIs.
- SEM_pend and SEM_post can be called from HWI and SWI threads only if the timeout is 0.
- Semaphores can lead to a *recursive semaphore deadlock*, which is a situation where two or more threads are directly or indirectly waiting for the same shared resource and none of them can continue executing without getting access to the critical resource. This is shown in the following example.

```
Void task()
{
    SEM_pend(mutex, SYS_FOREVER);
    `access shared resource`
    func();     /* Call to func */
    SEM_post(mutex);
}

Void func()
{
    SEM_pend(mutex, SYS_FOREVER);   /*Will block forever. Deadlock. */
    `access shared resource`
    SEM_post(mutex);
}
```

In this example, the second call to SEM_pend blocks the task forever. This is a typical example of functions shared by multiple code paths that cause recursive calls to SEM_pend leading to deadlock. This form of deadlock is relatively easy to detect when debugging, but the fix can complicate the code.

## 4.2 Mutual Exclusion Using Locks

Locks (LCK) can be used to achieve mutual exclusion in the same way as SEMs. LCK objects can be created statically using the Configuration Tool or dynamically using the LCK_create API call. The following APIs are used to achieve mutual exclusion:

- **LCK_pend.** Acquires ownership of a lock, which grants the current task exclusive access to the corresponding resource. If the lock is already owned by another task, LCK_pend suspends execution of the current task until the resource becomes available.
- **LCK_post.** Relinquishes ownership of the lock, and resumes execution of the first task (if any) awaiting availability of the corresponding resource.

If the current task calls LCK_pend more than once with the same lock, ownership remains with the current task until LCK_post is called an equal number of times. This feature of LCK can be used to overcome the recursive semaphore deadlock scenario displayed in the previous section. This is illustrated in the following example.

```
Void task()
{
   LCK_pend(mutex, SYS_FOREVER);
   `access shared resource`
   func();                        /* Call to func */
   LCK_post(mutex);
}

Void func()
{
  /*LCK allows owner to acquire the same lock multiple times. No deadlock. */
  LCK_pend(mutex, SYS_FOREVER);
  `access shared resource`
  LCK_post(mutex);
}
```

In this example, the second call to LCK_pend allows the task to acquire the resource, since the same LCK is already owned by that same task.

### *Issues to Consider*

The following issues need to be considered when calling LCK_pend and LCK_post.

- This method cannot protect critical sections from SWIs and HWIs.
- LCK_pend and LCK_post cannot be called from HWI and SWI threads.

## 4.3 Avoiding Multi-Mutex Deadlocks

A *multi-mutex deadlock* can occur between multiple threads in a system with more than one mutex where threads wait on shared resources. A multi-mutex deadlock can occur with both LCK and SEM objects.

The following example shows this scenario with LCK API calls.

```
Void task1()
{
   LCK_pend(lockA, SYS_FOREVER);
   'access shared resource A'
   f1();
   LCK_post(lockA);
}

Void f1()
{
   LCK_pend(lockB, SYS_FOREVER);
   /* Deadlock */
   'access shared resource B'
   LCK_post(lockB);
}
```

```
Void task2()
{
   LCK_pend(lockB, SYS_FOREVER);
   'access shared resource B'
   g2();
   LCK_post(lockB);
}

Void g2()
{
   LCK_pend(lockA, SYS_FOREVER);
   /* Deadlock */
   'access shared resource A'
   LCK_post(lockA);
}
```

Both threads block on shared resources held by each other, and a deadlock ensues. The Kernel Object View (KOV) can be used to detect this kind of deadlock in Code Composer Studio.

It is recommended that mutexes be acquired in a well-understood manner to avoid multi-mutex deadlocks. Also the number of mutexes in the application should be minimized. Simple applications should try to work with a single mutex.

## 4.4 Avoiding Priority Inversion

A *priority inversion* occurs when a higher priority task must wait for the completion of a lower priority task over an indefinite period. This scenario is illustrated in Figure 3.

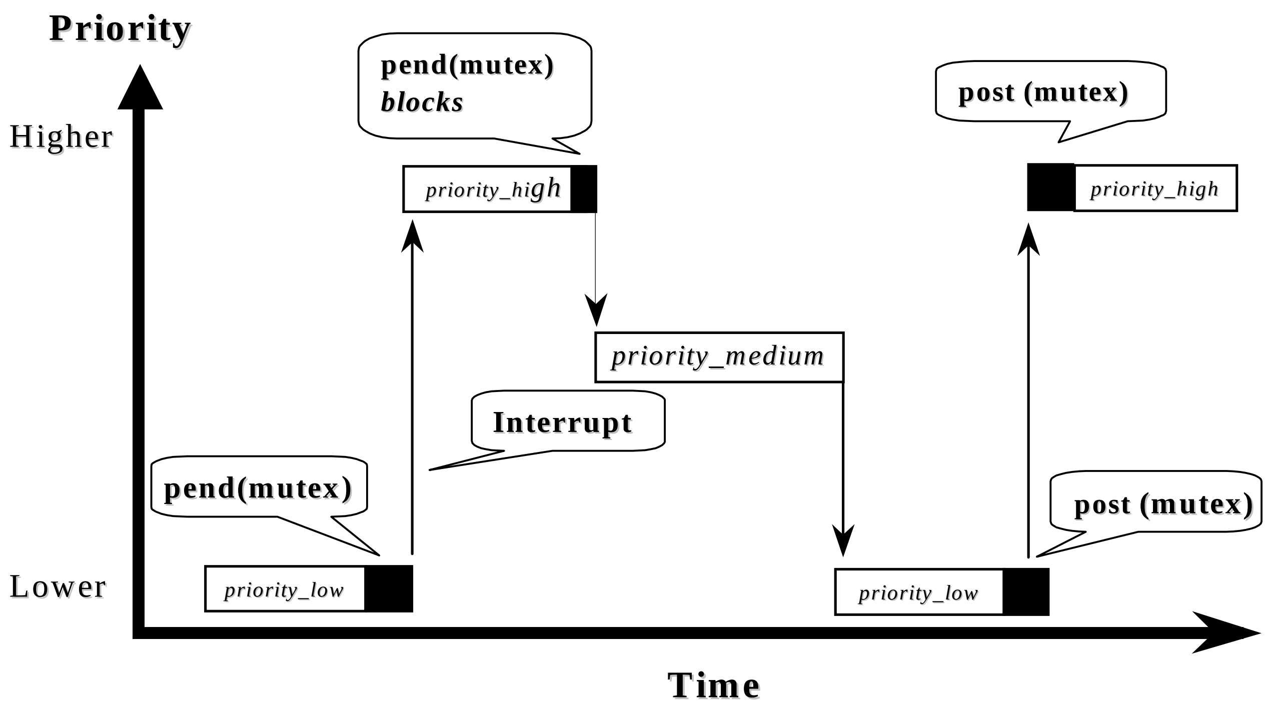


**Figure 3. Priority Inversion**

In Figure 3, *priority_low*, *priority_medium* and *priority_high* are tasks with differing priorities.

The *priority_low* task acquires a resource by pending on the mutex semaphore. When *priority_high* preempts *priority_low* and contends for the resource by pending on the same mutex semaphore, *priority_high* becomes blocked.

If *priority_high* were blocked no longer than the time it takes *priority_low* to finish using the shared resource, there would be no issue. However, in this case the *priority_low* task is preempted by the *priority_medium* thread. This prevents the *priority_low* thread from executing and relinquishing the shared resource. This causes the *priority_high* task to block for a long time. Such a scenario can drastically affect the real time behavior of a system.

The solution for such a scenario in a DSP/BIOS application is to temporarily raise the priority of the *priority_low* task to the highest priority that uses the same mutex. This enables the *priority_low* task to finish the critical section and relinquish the mutex without being preempted by medium priority tasks. This is illustrated in Figure 4.

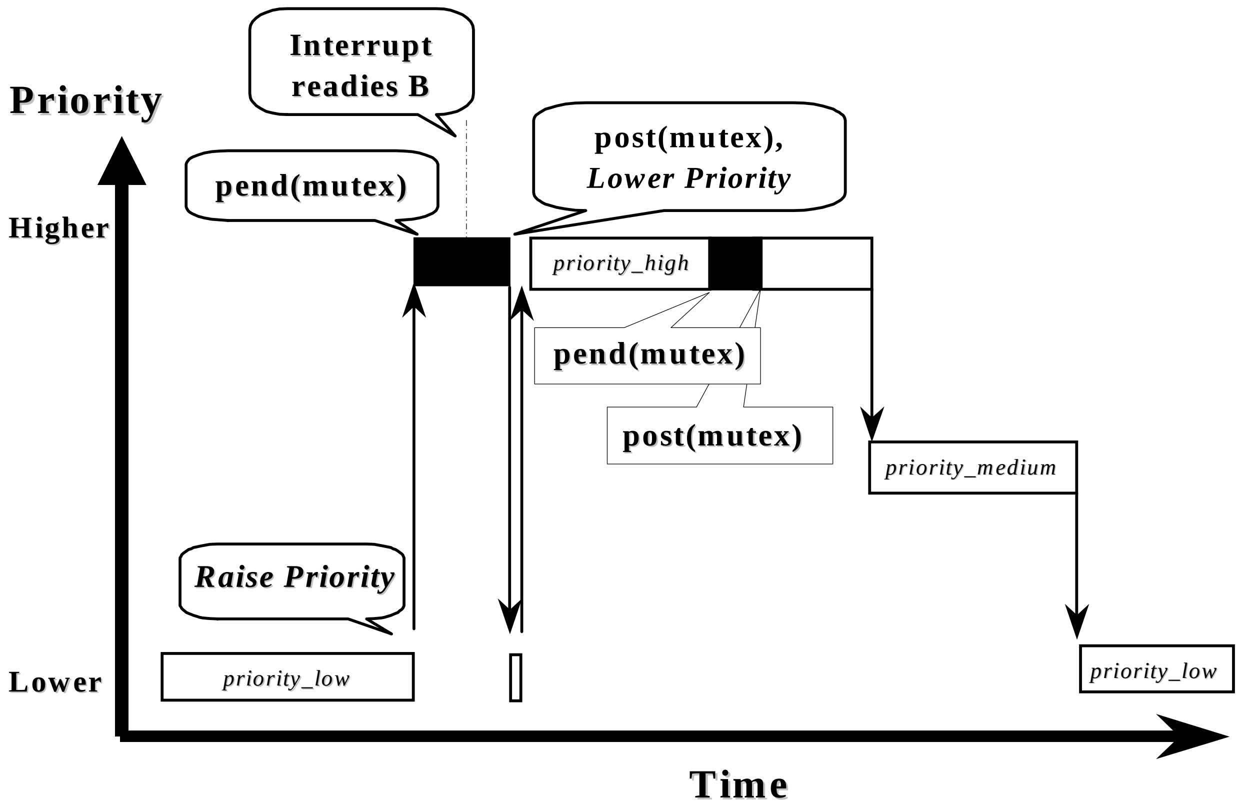


**Figure 4. Raise Priority Before Mutex**

In Figure 4, the priority of the *priority_low* task is raised to that of the *priority_high* task before *priority_low* starts executing the critical section. This is done using the TSK_setpri API call. After *priority_low* (now executing with high priority) completes the critical section and relinquishes the mutex, its priority is lowered to its previous state. This enables the *priority_low* task to complete the critical section without being preempted by the *priority_medium* task.

## 5 RTS Reentrancy Issues

Most of the Run Time Support (RTS) library functions are reentrant and thread safe when used with DSP/BIOS. Functions that are re-entrant can be invoked multiple times by different threads without leading to race conditions. RTS library functions have a mechanism built in that allows for thread safety, and DSP/BIOS utilizes this mechanism.

When not using DSP/BIOS, by default this mechanism is disabled and certain RTS functions are not reentrant. *malloc()* is the most common example. Also *stdio* functions (*fopen, fread,* etc.) and *printf* are also not reentrant. Basically, any function that needs to communicate with Code Composer Studio is not reentrant (when used without DSP/BIOS). Also, functions that hold state, such as *strtok*, are also not reentrant.

_lock() and _unlock() are supplied by the run-time support library. These locking functions are initialized to support reentrancy by DSP/BIOS. Specifically, _lock() and _unlock() are initialized to point to LCK_pend and LCK_post, which use a mutex LCK_Obj. It is important to note that lock() and unlock() cannot be implemented with a semaphore because there is the possibility of nested calls. (See Section 4.2, "Mutual Exclusion Using Locks" for information on how LCK solves the recursive semaphore deadlock scenario.) *printf* is one such RTS function where we encounter nested lock and unlock calls.

Using the LCK_pend/LCK_post mechanism to achieve reentrancy imposes the constraint that printf calls cannot be made from an ISR. This is because incorrect scheduling can occur since LCK_pend calls should not be made from HWI/SWI context.

## 6 CSL Reentrancy Issues

In the Chip Support Library (CSL), all the *_open()* and *_close()* functions are thread-safe and reentrant. This is because all of them access the allocation data-structures within IRQ_globalDisable/IRQ_globalRestore blocks. The reentrancy of these functions can be used to ensure reentrancy for the entire module. Before using any CSL API, the application code should *_open* the peripheral and verify that it received a valid handle. Two threads cannot receive a handle to the same peripheral (because the *_open* function is thread safe). This in turn ensures that two threads would not call the same CSL API for the same device. And hence two threads would not attempt to manipulate the same peripheral register.

There could be issues if you decide to share a handle between two threads. But then you can easily ensure that both threads make the CSL calls wisely. There are more issues—not all CSL modules are handle-based—which mean that they do not have *_open/_close* functions. In such cases you need to solve reentrancy issues for calls to these modules.

## 7 Conclusion

Before adopting a particular synchronization method for any DSP application, it is important that you make the application design as simple as possible.

Study the relative priorities of different threads in the system. Over-prioritization of threads should be avoided. This allows you to avoid needless context switches and priority inversion. Also you should try to minimize the number of SEM and LCK mutexes used. Over-use of SEM and LCK for mutual exclusion can lead to issues like multi-mutex deadlocks.

The choice of a particular method requires careful analysis of the application’s real time requirements. The use of APIs in each method has associated overheads of which you should be aware. There is no one right method for any application. Choose the method that makes sense for the requirements of your system. If need be, an application can use multiple methods for thread synchronization.

## 8 References

1. *TMS320 DSP/BIOS User’s Guide* (SPRU423)
2. *TMS320C28x DSP/BIOS Application Programming Interface Reference Guide* (SPRU625)
3. *TMS320C5000 DSP/BIOS Application Programming Interface Reference Guide* (SPRU404)
4. *TMS320C6000 DSP/BIOS Application Programming Interface Reference Guide* (SPRU403)
5. *DSP/BIOS Timing Benchmarks for Code Composer Studio 2.2* (SPRA900)

**IMPORTANT NOTICE**

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Copyright © 2004, Texas Instruments Incorporated