Draft

Zero-Latency Interrupts and TI-RTOS on C2000 Devices

Todd Mullanix

TI-RTOS Apps Manager

Feb, 11, 2018

Agenda

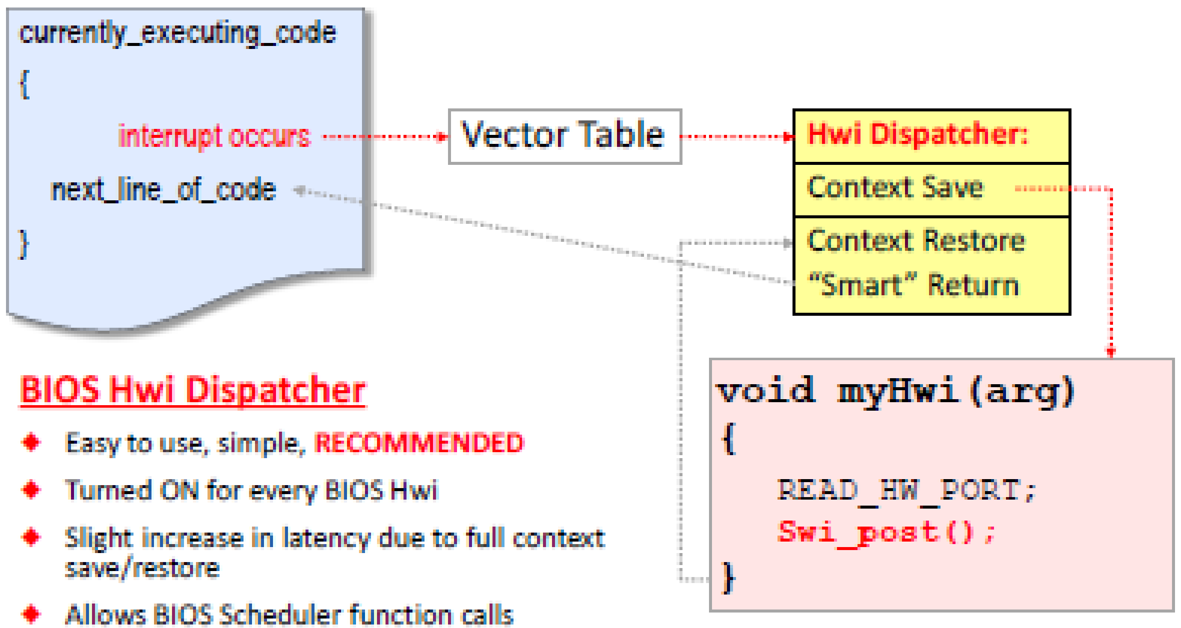
Here's the high-level view of what we will cover in this presentation:

1. Typical kernel interrupt
2. What is a zero-latency interrupt?
3. When to use a zero-latency interrupt
4. Steps to add a zero-latency interrupt into a TI-RTOS based F28379D example.

Managed Interrupts in TI-RTOS Kernel

When the TI-RTOS kernel (aka SYS/BIOS or sometimes just BIOS) manages an interrupt, it is executed via the Hwi Dispatcher as opposed to using the *interrupt* keyword.

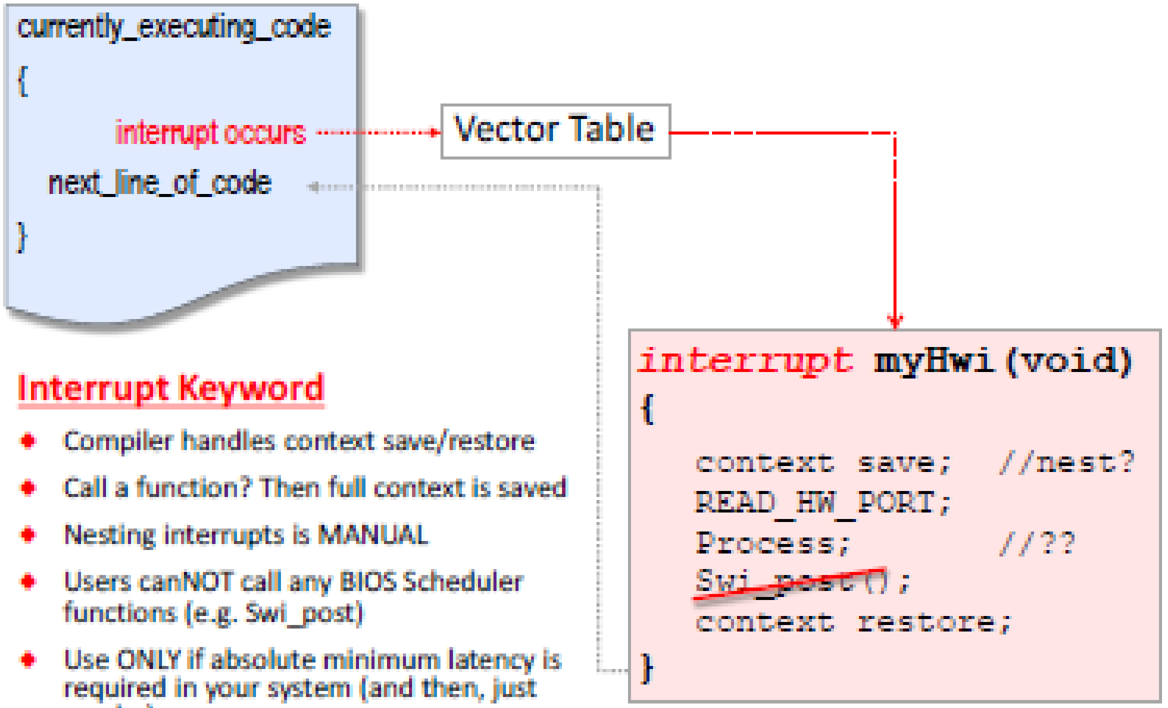
Using the BIOS Hwi Dispatcher



BIOS Hwi Dispatcher

- Easy to use, simple, RECOMMENDED
- Turned ON for every BIOS Hwi
- Slight increase in latency due to full context save/restore
- Allows BIOS Scheduler function calls
- Saves code space (all INTs share common save/restore routine)
- Performs "smart return" – returns to highest priority pending thread

Using the *interrupt* Keyword



Interrupt Keyword

- Compiler handles context save/restore
- Call a function? Then full context is saved
- Nesting interrupts is MANUAL
- Users canNOT call any BIOS Scheduler functions (e.g. Swi_post)
- Use ONLY if absolute minimum latency is required in your system (and then, just maybe)

What is a Zero-Latency Interrupt?

You can have an interrupt not be managed by the kernel. The interrupt will run independently of the kernel. We call this a “zero-latency interrupt”*.

There are four key points about zero-latency interrupts.

1. The **kernel adds no overhead** when the interrupt runs (thus the name zero-latency).
2. The kernel will **never disable** a zero-latency interrupt.
3. The zero-latency interrupt **cannot make any calls into the kernel that would impact the scheduler** (.e.g. `Semaphore_post()` is not allowed in a zero-latency interrupt).
4. There is a **slightly negative performance impact** on the kernel with C28 devices when there is a zero-latency interrupt in the system. This is because the kernel has to use the mask when disabling/enabling the interrupts it manages.

* Some people prefer the name “unmanaged interrupt”

When to use a Zero-Latency Interrupt?

When the TI-RTOS kernel manages an interrupt, an overhead is incurred. The overhead is detailed in the `SYS/BIOS Release Notes->Sizing and Timing Benchmarks->Target`. For example, to the right shows some of the timing benchmarks for a TMS320F280049M device (with hard FP). Note: the SYS/BIOS User Guide discusses these benchmarks in more details.

There are two key areas to consider about interrupt timing

1. **Maximum duration the kernel ever disables interrupts**: This value is denoted by the "Interrupt Latency" benchmark. During this interval, an asserted interrupt will not run. If this value might have a negative effect on one of your interrupts, you should consider making it a zero-latency interrupt.
2. **Overhead incurred by having the kernel manage the interrupt**: The kernel adds some overhead to the running of an ISR. This value is denoted by the "Hwi dispatcher" value. It is mostly comprised on the "Hwi dispatcher prolog" and "Hwi dispatcher epilog". If this value might have a negative effect on one of your interrupts, you should consider making it a zero-latency interrupt.

Please note: you timing numbers may vary based on compiler settings, flash wait-states, etc.

C28x with hard FP Timing Benchmarks

Target Platform: ti.platforms.tms320x28:TMS320F280049M:1

Tool Chain Version: 16.9.1

BIOS Version: bios_6_52_00_11_eng

XDCTools Version: xdctools_3_50_03_33_core

Benchmark	Cycles
Interrupt Latency	153
Hwi_restore()	19
Hwi_disable()	13
Hwi dispatcher prolog	210
Hwi dispatcher epilog	155
Hwi dispatcher	366
Hardware Interrupt to Blocked Task	588
Hardware Interrupt to Software Interrupt	420

Lab

For this presentation we are going to first play with two different examples on the F28379D controlCARD (or LaunchPad).

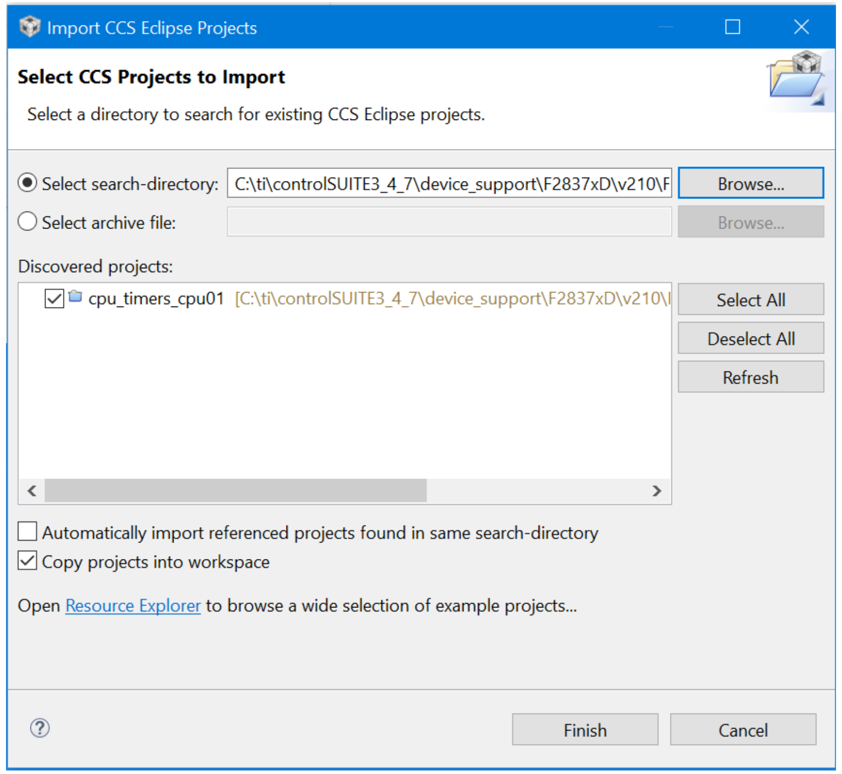
1. **ControlSuite's cpu_timer Example**: Non-RTOS based application that configures and runs 3 timers.
2. **SYS/BIOS' Task Mutex Example**: Simple application where the kernel has a couple tasks accessing a shared resource. Each task sleeps for a bit and then tries to get the resource via a semaphore.

The goal is to add another timer into the Task Mutex Example. The timer will run at 30us, so we are going to make it a zero-latency interrupt. We'll freely steal some of the code from the cpu_timer example to accomplish this.

Import and Build cpu_timers example

In CCS, **import** the cpu_timer project in controlSUITE.

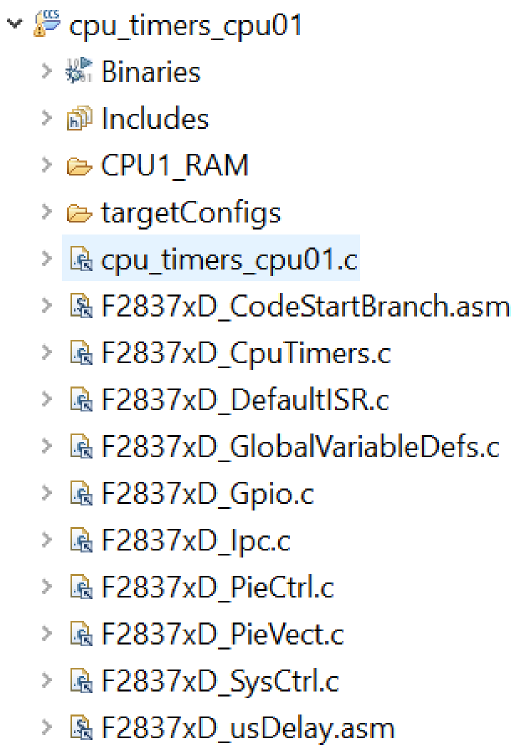
controlSUITE\device_support\F2837xD\v210\F2837xD_examples_Cpu1\cpu_timers



If you are using the **F28379D LaunchPad** (instead of the controlCard), please add the following compiler predefined symbol

`_LAUNCHXL_F28379D`

The LaunchPad has a slower external oscillator than the controlCARD.



Build the project.

This project configures **3 timers to run every second** and increment a unique counter.

Run cpu_timers example

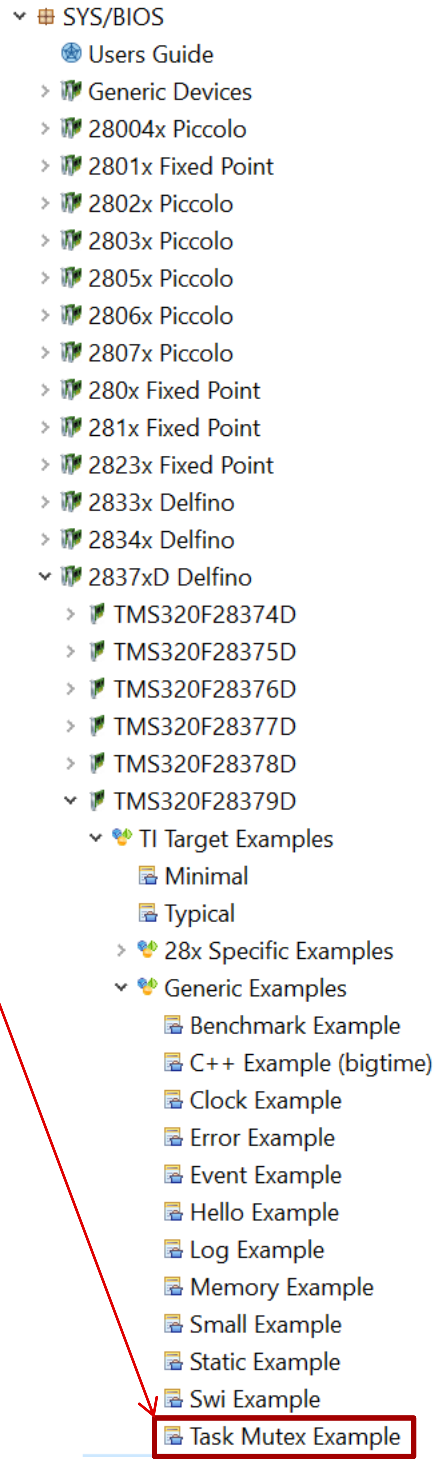
Load and run the cpu_timers .out file on CPU1. After N number of seconds, pause the core and look at the CpuTimer0, CpuTimer1, and CpuTimer2 variables. You'll see that their InterruptCount corresponds to the number of seconds you have ran the application (in this case 5 seconds).

Variables | *Expressions* | Registers

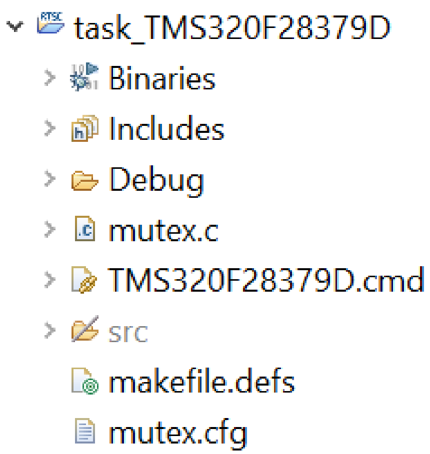
Expression	Type	Value
CpuTimer0	struct CPUTIMER_VARS	{RegsAddr=0x00
RegsAddr	struct CPUTIMER_REGS *	0x00000C00 {TIN
InterruptCount	unsigned long	5
CPUFreqInMHz	float	200.0
PeriodInUSec	float	1000000.0
CpuTimer1	struct CPUTIMER_VARS	{RegsAddr=0x00
RegsAddr	struct CPUTIMER_REGS *	0x00000C08 {TIN
InterruptCount	unsigned long	5
CPUFreqInMHz	float	200.0
PeriodInUSec	float	1000000.0
CpuTimer2	struct CPUTIMER_VARS	{RegsAddr=0x00
RegsAddr	struct CPUTIMER_REGS *	0x00000C10 {TIN
InterruptCount	unsigned long	5
CPUFreqInMHz	float	200.0
PeriodInUSec	float	1000000.0

Build and Run SYS/BIOS Mutex Example

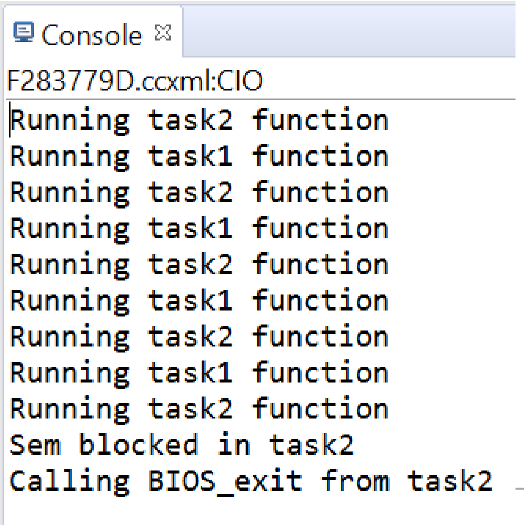
In CCS, **import** Task Mutex example via TI Resource Explorer Classic.



Build and load the project onto CPU1.



When you **run** the example, you'll see tasks ping-pong back and forth and then terminate.



TEXAS INSTRUMENTS

SYS/BIOS Mutex Example Overview

Here's a look at the pseudo-code for the two tasks

```
Task1 //(lower priority)
{
   while (1) {
      Semaphore_pend(sem, BIOS_WAIT_FOREVER)
      //simulate doing real work…
      resource++;
      Semaphore_post(sem)
      Task_sleep(10)
   }
```

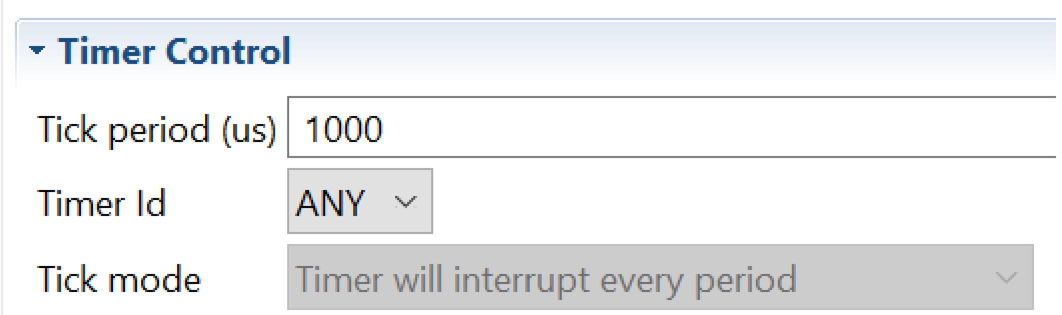
```
Task2 //(higher priority)
{
   while (1) {
      Semaphore_pend(sem, BIOS_WAIT_FOREVER)
      //simulate doing real work…
      resource++;
      Semaphore_post(sem)
      Task_sleep(10)
      finishCount++
      if (finishCount == 5) {
         BIOS_exit(0);
      }
   }
```

The semaphore makes sure the higher priority task does not preempt the lower priority task when it is updating `resource`. Let's look a little more how the `Task_sleep()` works…

Timing in SYS/BIOS

The Clock module by default **grabs a timer** to use for the Clock instances and for driving timing mechanism like `Task_sleep` and `Semaphore_pend` (with a timeout).

Here's the default configuration (from mutex.cfg in the Task Mutex example)



As you can see, ANY is specified. This means the kernel will grab a timer that is **not already used in the .cfg file**. For this lab, will fix this to a specific timer since we want to avoid collisions.

Also note the **default period is 1000us** (1ms). The kernel will do book-keeping during this interrupt (e.g. wake-up tasks whose Task_sleep has expired, call Clock functions that are due to run, etc.)

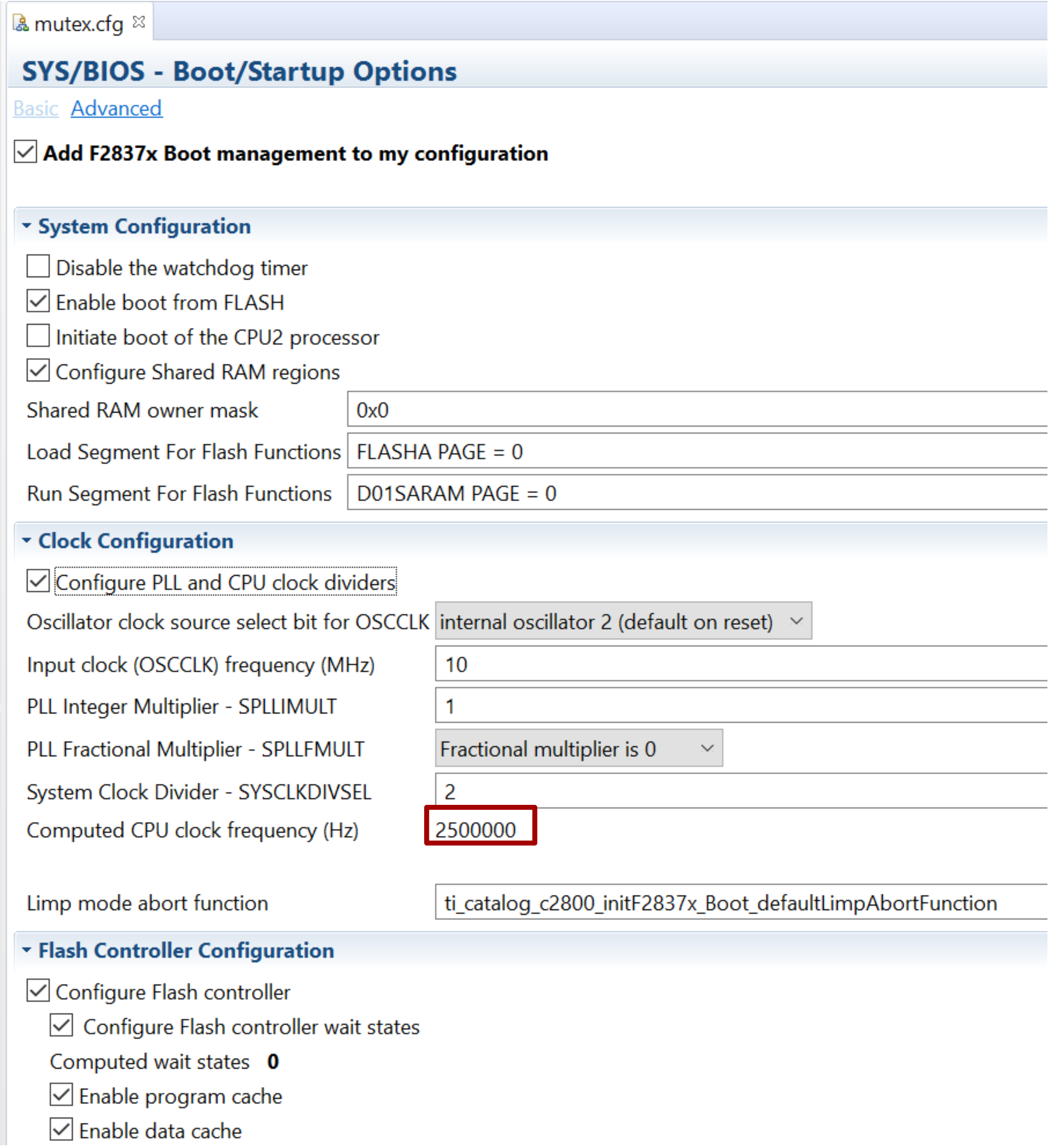
Task_sleep's argument is in ticks. So Task_sleep(10) means to sleep for 10 ticks, or when the Clock's period is 1ms, sleep for 10ms.

CPU Speed in SYS/BIOS

The .cfg file can be used to configure the CPU speed on the C28 devices. Here is the graphical view of the Boot module (after

```
var Boot =
xdc.useModule('ti.catalog.c2800.initF2837x.Boot');
```

was added into the .cfg).



By default, the kernel sets the CPU speed to 2.5MHz.

This can be changed by setting the clock source, multiplers, etc., but for this lab **we'll leave it at 2.5MHz**. We'll need this when configuring the new zero-latency timer.

Task Mutex Example Enhancment: Overview

Let's say we want to add something that needs to be checked every 30us. Here are three options:

1. Make the SYS/BIOS Clock module's timer run at 30us.
2. Use a different timer, via the SYS/BIOS API `Timer_create()`, that runs every 30us.
3. Use a different timer that is not managed by the kernel (aka zero-latency interrupt).

Option 1 is a bad idea because the kernel does lots of book-keeping on a Clock tick. 30us is simply too fast of a period.

Option 2 might work. It takes a minimum of 366 cycles to run an empty Hwi (refer to the SYS/BIOS Release Notes for timing benchmarks). Let's say we bump this up to 400 to have it actually do something. So if you are running faster than 13.3MHz it will work, but you are using lots of cycles.

CPU Speed	uS/cycle	# cycles/interrupt	uS required for each 30us Interrupt
2.5MHz	.4	400	160
13.3MHz	.07519	400	30
50MHz	.02	400	8
200MHz	.005	400	2

That do option 3! Let's look to see how to add a timer (in this case Timer1) as a zero-latency interrupt…

Task Mutex Example Enhancment: Configuration File Changes

First let's make two changes to the kernel configuration file (.cfg). We'll edit the mutex.cfg as a text file (instead of graphically).

1. The default is to let the kernel select a non-used timer. Since it does not know about the zero-latency timer at build time, we will **specify Timer2 as the timer to be used by the kernel**. This is to avoid any conflict with the zero-latency timer interrupt we are going to create. Please add this to the bottom of the .cfg file.

```
Clock.timerId = 2;
```

2. Tell the kernel that **interrupt 13 (Timer1) will be a zero-latency interrupt**. Please add this to the bottom of the .cfg file.

```
Hwi.zeroLatencyIERMask = 0x1000;  // note: the 13th bit is set
```

Task Mutex Example Enhancment: Plugging Interrupt

3. During runtime, we need to **plug in the ISR**. Please add the following **bolded** code to the mutex.c file. The next slides will add the variables and necessary functions.

```
#include <ti/sysbios/family/c28/Hwi.h>

__interrupt void cpu_timer1_isr(void)
{
    CpuTimer1.InterruptCount++;
}

Int main()
{
    …
    tsk2 = Task_create (task2, &taskParams, NULL);
    /* Plug in the zero-latency interrupt */
    Hwi_plug(13, cpu_timer1_isr);

    BIOS_start();    /* does not return */
```

Task Mutex Example Enhancment: Plugging Interrupt

4. Now we need to **configure the timer and enable the interrupt**. Please add the following **bolded** code.

```
Int main()
{
    …
    tsk2 = Task_create (task2, &taskParams, NULL);
    /* Plug in the zero-latency interrupt */
    Hwi_plug(13, cpu_timer1_isr);
    myInitCpuTimers();
    ConfigCpuTimer(&CpuTimer1, 2.5, 30); // CPU is 2.5MHz, timer is 30us
    CpuTimer1Regs.TCR.all = 0x4000;
    Hwi_enableIER(0x1000);

    BIOS_start();    /* does not return */
```

Task Mutex Example Enhancment: Initialize Timer1

5. Please add the following **timer initialization code** into clock.c (somewhere above `main()`). This is modified version of the function in the cpu_timers project. We only need to initialize Timer1.

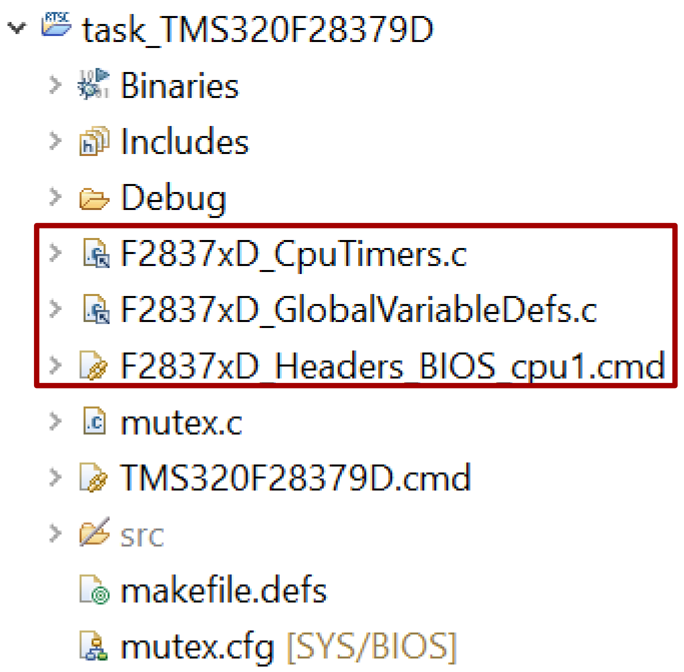
```
#include "F28x_Project.h"

void myInitCpuTimers(void)
{
    CpuTimer1.RegsAddr = &CpuTimer1Regs;
    CpuTimer1Regs.PRD.all  = 0xFFFFFFFF;
    CpuTimer1Regs.TPR.all  = 0;
    CpuTimer1Regs.TPRH.all = 0;
    CpuTimer1Regs.TCR.bit.TSS = 1;
    CpuTimer1Regs.TCR.bit.TRB = 1;
    CpuTimer1.InterruptCount = 0;
}
```

Task Mutex Example Enhancment: Steal code from cpu_timers

6. Copy and paste these **two files from the cpu_timers project** and add them into the Task Mutex project
   - F2837xD_CpuTimers.c
   - F2837xD_GlobalVariableDefs.c

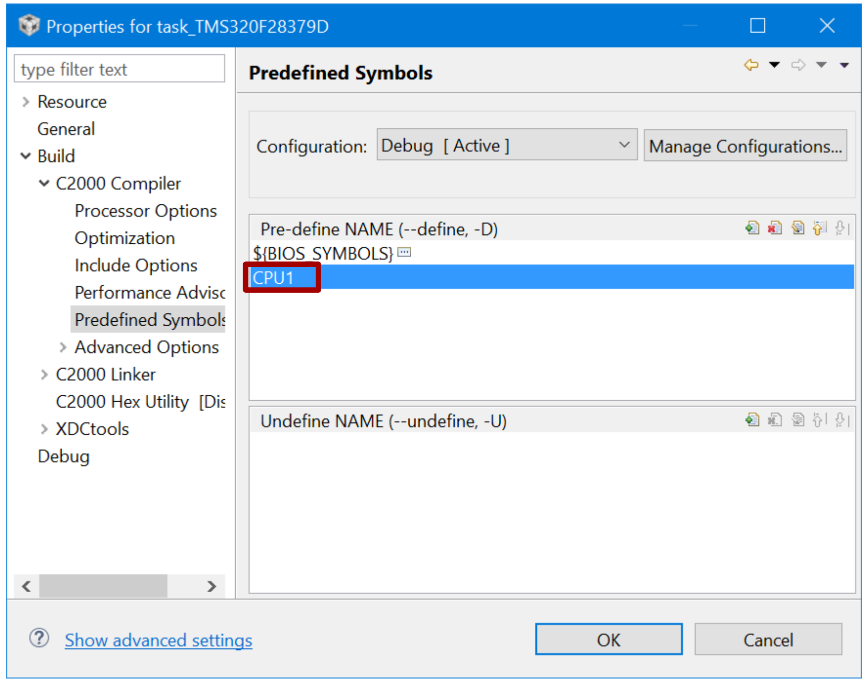
   We'll use global variables and functions from these files instead of redoing them.

7. Let's **add a linker file from controlSUITE that supports SYS/BIOS**. Please add the controlSUITE\device_support\F2837xD\v210\F2837xD_headers\cmd\F2837xD_Headers_BIOS_cpu1.cmd file



Task Mutex Example Enhancment: Linker File

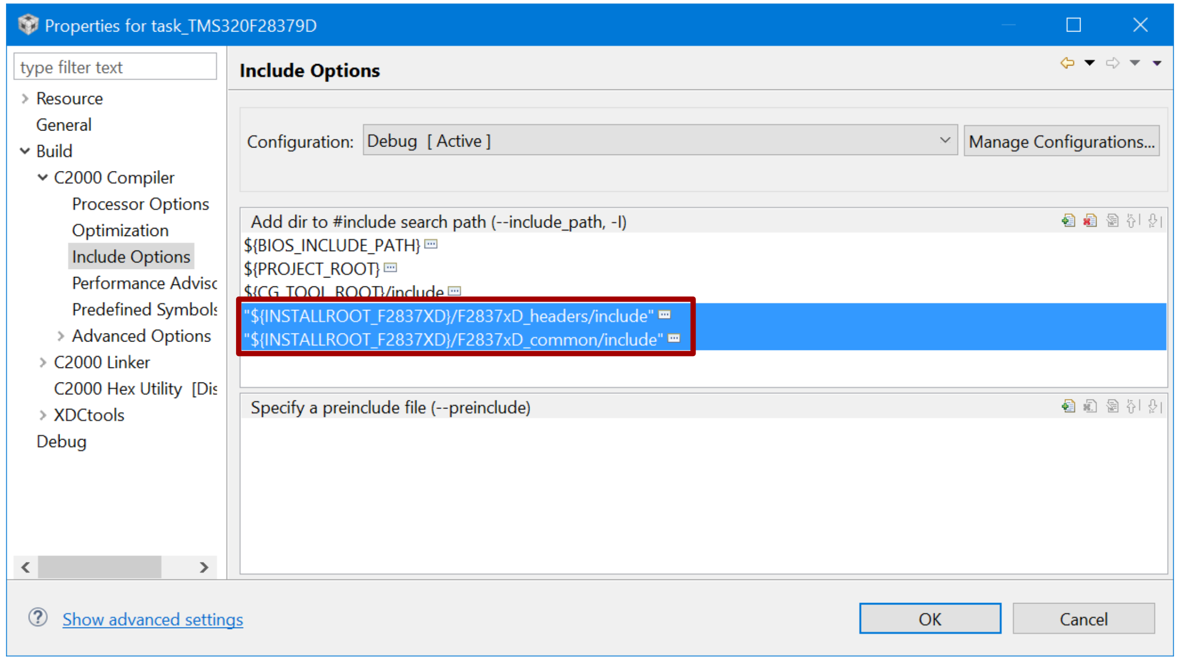
8. The controlSUITE files depend on a **CPU1 define**. So add it into the project.



9. Finally, let's **add includes paths for controlSUITE**. Namely (this is assuming you have a linked resource for controlSUITE in CCS).

"${INSTALLROOT_F2837XD}/F2837xD_headers/include"

"${INSTALLROOT_F2837XD}/F2837xD_common/include"



Task Mutex Example Enhancment: Build/Load/RUN!

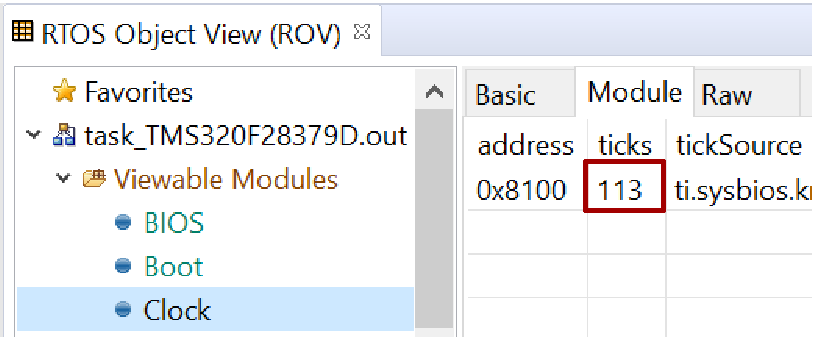
10. Let's **build the project**. You'll probably get a warning☹ We have an open bug report on this. It can be ignored.

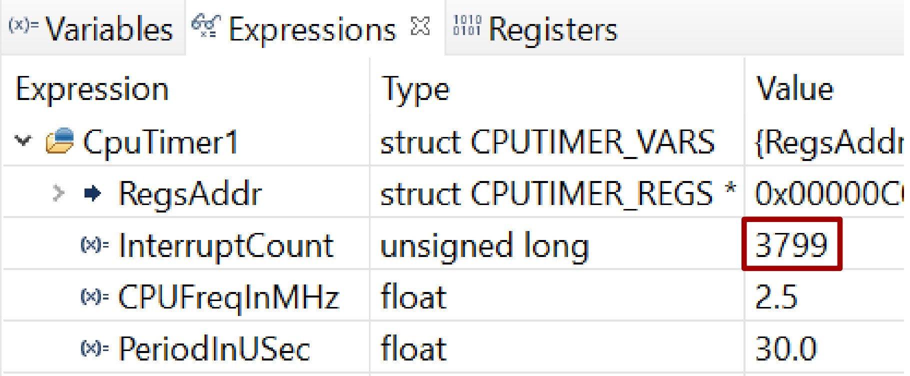
```
"C:/ti/controlSUITE3_4_7/device_support/F2837xD/v210/F2837xD_headers/include/F2837xD_device.h", line 246: warning
#303-D: typedef name has already been declared (with same type)
"C:/ti/controlSUITE3_4_7/device_support/F2837xD/v210/F2837xD_headers/include/F2837xD_device.h", line 247: warning
#303-D: typedef name has already been declared (with same type)
```

11. Now **load the application, set a breakpoint** at the `BIOS_exit()` call in task2, and **run**.

```
        finishCount++;
        if (finishCount == 5) {
            System_printf("Calling BIOS_exit from task2\n");
            BIOS_exit(0);
        }
    }
```

12. If you look at the tick count (ROV->Clock) and the CpuTimer.InterruptCount, they make sense! 113 ticks * 1000us / 30us = ~3766 for InterruptCount. Since we halted after a Clock tick boundary, it's expected that InterruptCount is slightly higher than calculated.





Advanced

1. You can comment out the BIOS_exit and run the application for a longer period. Or you can even run past the breakpoint. The **zero-latency interrupt will continue even though the kernel has exited**.
2. If the zero-latency interrupt **needed to tell the kernel about something**, it could call `Hwi_post()` on an interrupt that is managed by the kernel. That interrupt can call Semaphore_post(), etc.

Additional Resources

- http://processors.wiki.ti.com/index.php/SYS/BIOS_for_the_28x
- SYS/BIOS User Guide (inside the docs directory on an install SYS/BIOS product).

TEXAS INSTRUMENTS