ECE3849
D-Term 2021

Real Time Embedded Systems

Module 3 Part 2

Module 3 Part 2 Overview

- RTOS Interrupt and Task Latency.
- Using rtos.cfg to configure tasks and interrupts.
- Software interrupts.
- System priorities.
- Using Semaphores to control tasks.

ece3849_rtos_int_latency

```
void main(void) {
    IntMasterDisable();

    // initialize general purpose timers 1-3 for periodic interrupts
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER1);
    TimerDisable(TIMER1_BASE, TIMER_BOTH);
    TimerConfigure(TIMER1_BASE, TIMER_CFG_PERIODIC);
    TimerLoadSet(TIMER1_BASE, TIMER_A, TIMER1_PERIOD - 1);
    TimerIntEnable(TIMER1_BASE, TIMER_TIMA_TIMEOUT);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER2);
    TimerDisable(TIMER2_BASE, TIMER_BOTH);
    TimerConfigure(TIMER2_BASE, TIMER_CFG_PERIODIC);
    TimerLoadSet(TIMER2_BASE, TIMER_A, TIMER2_PERIOD - 1);
    TimerIntEnable(TIMER2_BASE, TIMER_TIMA_TIMEOUT);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER3);
    TimerDisable(TIMER3_BASE, TIMER_BOTH);
    TimerConfigure(TIMER3_BASE, TIMER_CFG_PERIODIC);
    TimerLoadSet(TIMER3_BASE, TIMER_A, TIMER3_PERIOD - 1);
    TimerIntEnable(TIMER3_BASE, TIMER_TIMA_TIMEOUT);

    BIOS_start();     /* start SYS/BIOS */
}
```

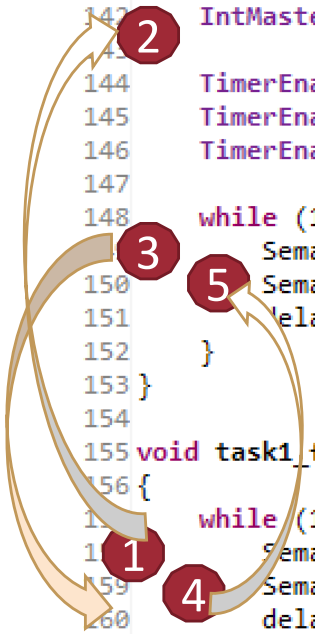
- main()ece3849_int_latency
  - Configures the Timer peripherals like normal.
  - Does NOT configure the interrupt controller. This is now handled by the RTOS.
  - Calls BIOS_start()

```
89 void event0_handler(void)
90 {
91     unsigned long t = TIMER1_PERIOD - TIMER1_TAR_R;
92     if (t > event0_latency) event0_latency = t; // measure
93     TIMER1_ICR_R = TIMER_ICR_TATOCINT; // clear interrupt f
94
95     delay_us(EVENT0_EXECUTION_TIME); // handle event0
96
97     if (TIMER1_MIS_R & TIMER_MIS_TATOMIS) { // next event o
98         event0_missed_deadlines++;
99         t = 2 * TIMER1_PERIOD; // timer overflowed since la
100    }
101    else t = TIMER1_PERIOD;
102    t -= TIMER1_TAR_R;
103    if (t > event0_response_time) event0_response_time = t;
104 }
```

- ISR event_handler functions the same as before.
  - Measures latency, response time and missed deadlines.
  - This example uses direct register accesses instead of function calls.

ece3849_rtos_int_latency

```
140 void task0_func(UArg arg0, UArg arg1)
141 {
142     IntMasterEnable();
143
144     TimerEnable(TIMER1_BASE, TIMER_A);
145     TimerEnable(TIMER2_BASE, TIMER_A);
146     TimerEnable(TIMER3_BASE, TIMER_A);
147
148     while (1) {
149         Semaphore_post(task1_sem);
150         Semaphore_pend(task0_sem, BIOS_WAIT_FOREVER);
151         delay_us(200);
152     }
153 }
154
155 void task1_func(UArg arg0, UArg arg1)
156 {
157     while (1) {
158         Semaphore_post(task0_sem);
159         Semaphore_pend(task1_sem, BIOS_WAIT_FOREVER);
160         delay_us(200);
161     }
162 }
```



- Once the RTOS is running it starts two tasks.
- task1_func higher priority.

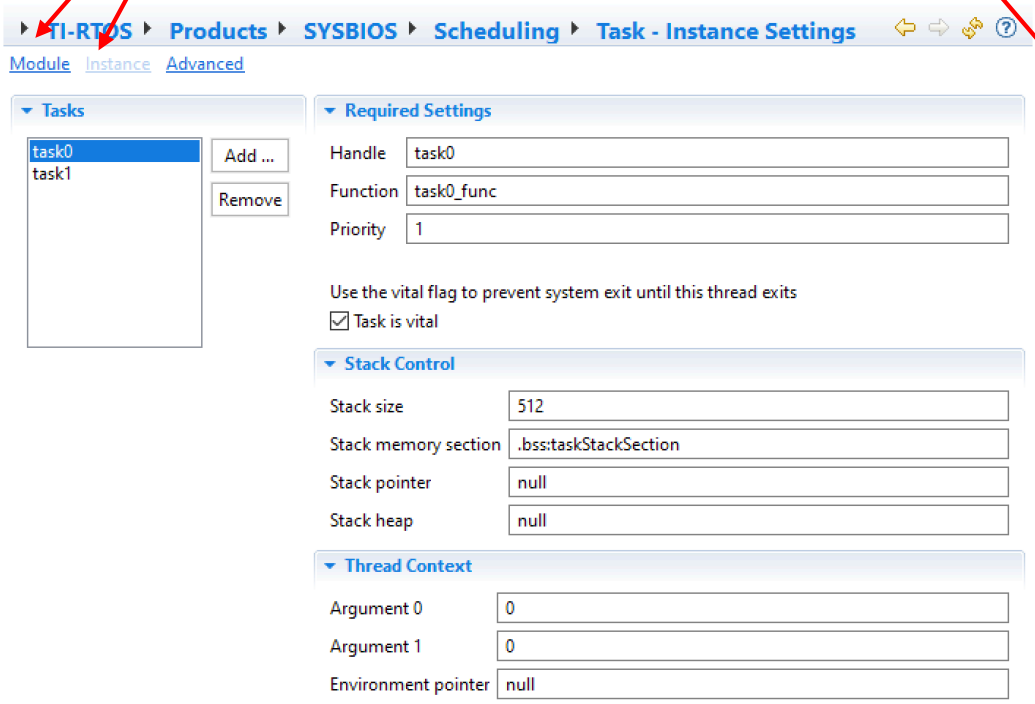
1. task1_func posts to task0_sem, task0_func is in a ready state. It is lower priority than task1_func so it can not start.
2. task1_func pends on task1_sem, it enters a blocked state and allows task0_func starts running.
   - task0_func becomes unblocked and starts running.
     - First time through it enables interrupts and starts the interrupt timers.
     - Then enters while () loop forever.
3. task0_func posts to task1_sem.
4. task1_func becomes unblocked and starts running, as it is a higher priority.
5. task1_func posts to task0_sem and then pends on task1_sem allowing task0_func to start running again.

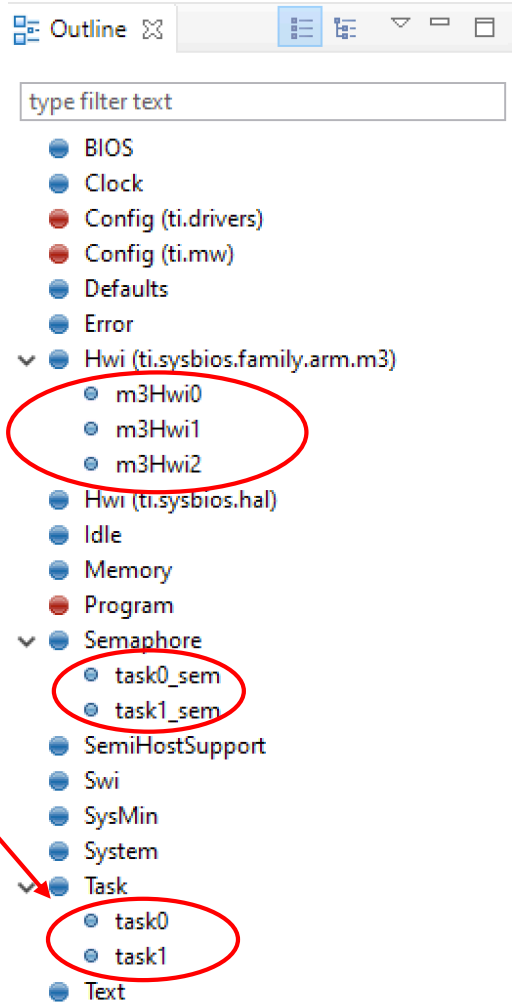
task0_func and task1_func continuously hand off control to each other.

Each taking a turn.

ece3849_rtos_int_latency: rtos.cfg

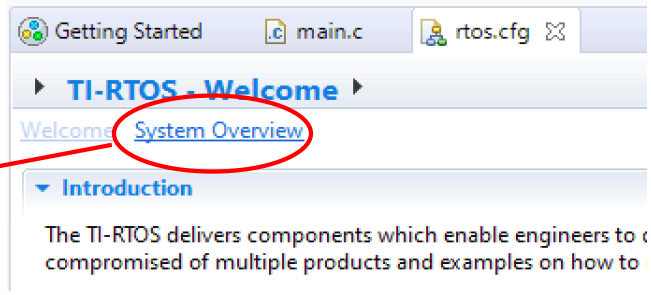
- To open RTOS configuration GUI
  - Right click on rtos.cfg -> Open with XGCONF to open a GUI environment
- Outline panel on right side of screen
  - Shows what Interrupts, Semaphores and tasks are in use.
    - We have 3 Hardware Interrupts.
    - Two semaphores
    - Two tasks
- Clicking on the items will bring up configuration details.
  - There are Module settings applying to all instances of that type.
  - There are Instance settings applying to a specific instance.

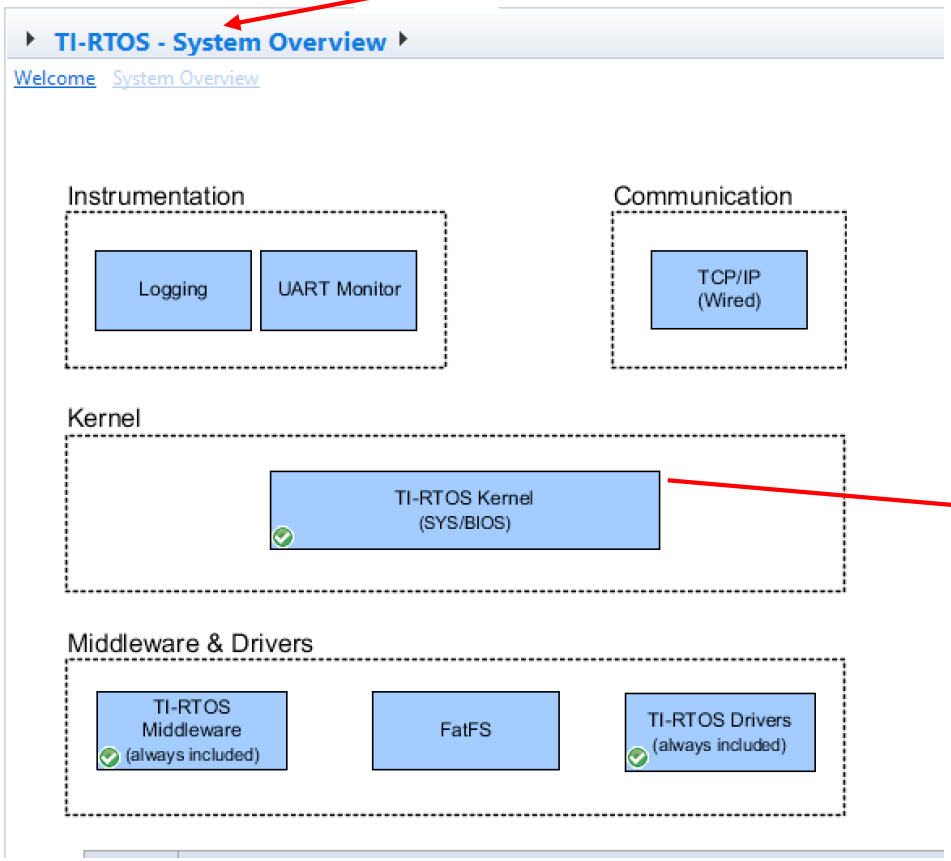


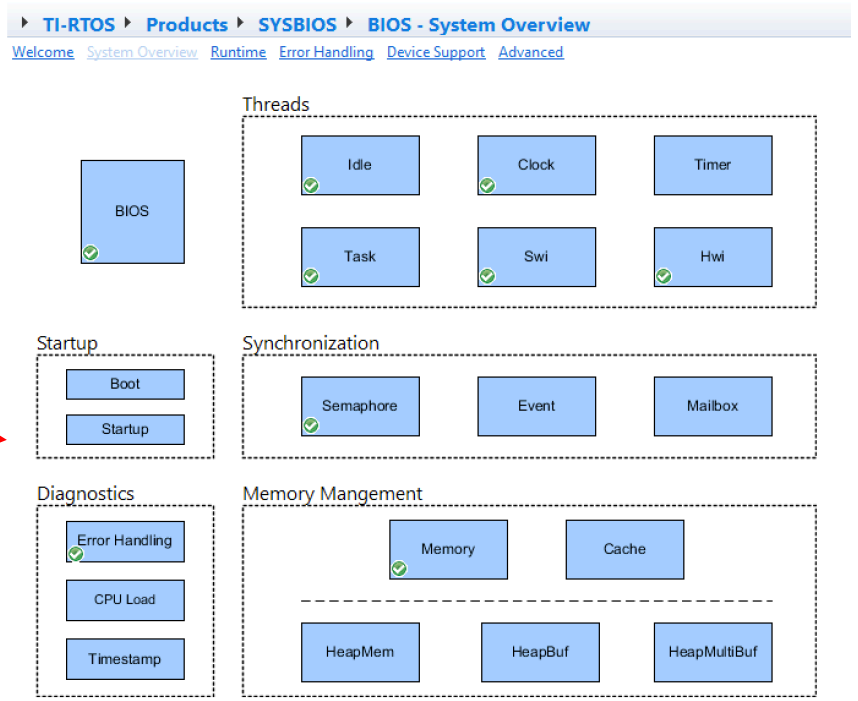


ece3849_rtos_int_latency: rtos.cfg

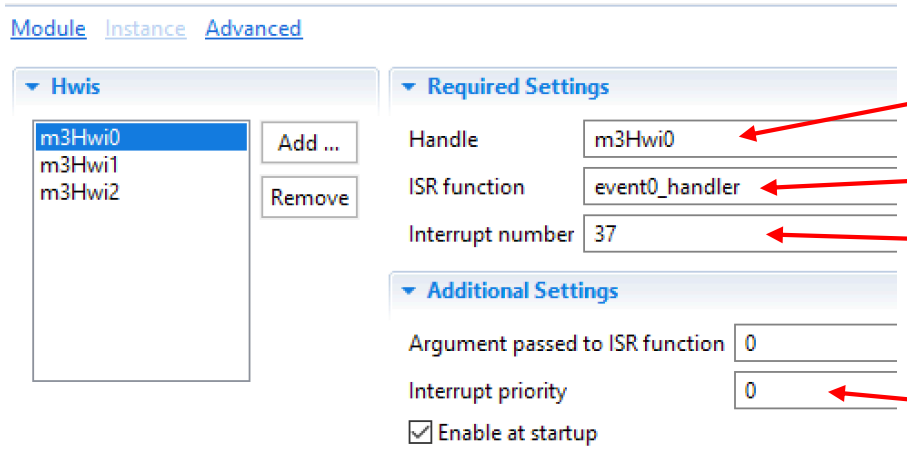
- Clicking System Overview in upper left corner of TI_RTOS window.
  - Shows all possible options.
  - Clicking into the boxes will bring up the functional options for each block.
  - The green checks indicated those features are already added to the configuration.
    - Clicking on a non-checked box will give you the option of adding that functionality to your design.







Interrupt Configuration

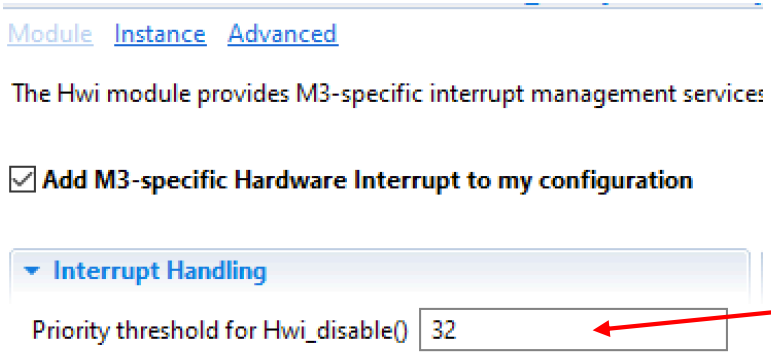


Name of Interrupt

Name of ISR function.

Hardware Interrupt number from datasheet
Table 2-9 in Vector Number column.

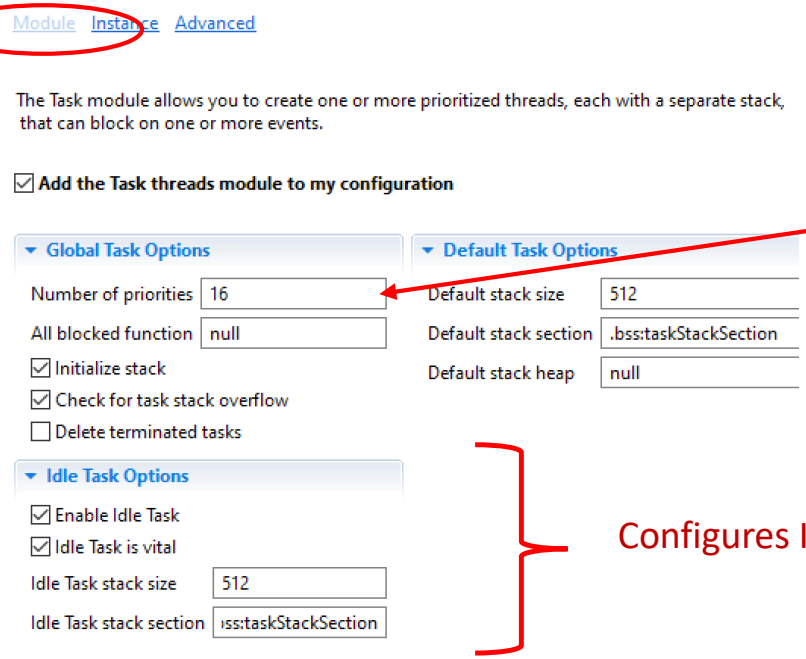
Lower number = higher priority.
0 is the highest priority.
0, 32, 64, 96, 128, etc… are valid numbers.
(only top 3 MSB's of 8-bit number are used to set priority)



Interrupt numbers lower than this are "zero latency interrupts".

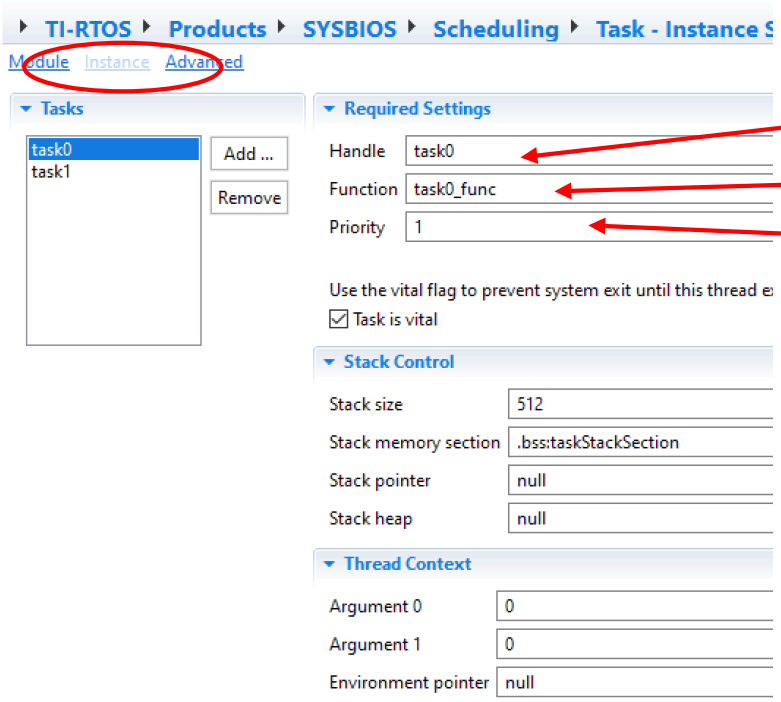
- They are not scheduled by the RTOS, they are highest priority and preempt other interrupts that are handled by the RTOS.
- They also can not use RTOS scheduling mechanisms like semaphores.

Task Configuration



Our hardware supports maximum of 32 priorities, but default is 16.

Configures IDLE task, for when nothing is scheduled



Task name

Function name related to task.

Task priority
Higher number = higher priority.
0 is idle task (lowest priority)

Stack configuration

RMS Preemptive Scheduling

Event	Period	Execution Time	Priority	Latency	Response Time (Latency + ?)	Relative Deadline (Period)	Schedulable ? YES = response < deadline
event0	6 ms	2 ms	High	0 ms	2 ms	6 ms	Yes
event1	8 ms	1 ms	Mid	2 ms	3 ms	8 ms	Yes
event2	12 ms	6 ms	Low	3 ms	12 ms	12 ms	Yes, marginal

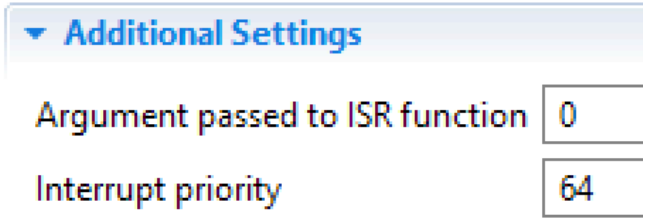
- Event0_handler = "zero latency interrupt" we get same results as with no RTOS.

(x)= Variables | Expressions | Registers

Expression	Type	Value	Expected	Address
(x)= event0_latency/120.0f	float	0.358333319	0 ms	
(x)= event1_latency/120.0f	float	2004.21667	2.0 ms	
(x)= event2_latency/120.0f	float	3005.18335	3.0 ms	
(x)= event0_response_time/120.0f	float	2000.6333	2.0 ms	
(x)= event1_response_time/120.0f	float	3004.91675	3.0 ms	
(x)= event2_response_time/120.0f	float	12010.4336	12.0 ms	
(x)= event0_missed_deadlines	unsigned long	0		0x200015F4
(x)= event1_missed_deadlines	unsigned long	0		0x200015F8
(x)= event2_missed_deadlines	unsigned long	0		0x200015FC
Add new expression				

RTOS Increase Interrupt Latency

- When event0_handler is configured as a ”zero latency interrupt”.
  - It bypasses the RTOS and has the same performance as with no RTOS.
  - It is the highest priority preempting everything controlled by the RTOS.
  - They can not engage with RTOS and so can not post semaphore or issue commands that would change RTOS scheduling.
- By setting the interrupt priority to 64, event0_handler is now RTOS controlled.
  - The RTOS scheduler adds latency to all interrupts.
  - Event0 now has 10x the latency = 4 usec.
  - Event2 which was marginal is now failing.
  -



(x)= Variables | Expressions | Registers

Expression	Type	Value	Address
(x)= event0_latency/120.0f	float	3.99166656	
(x)= event1_latency/120.0f	float	2007.75	
(x)= event2_latency/120.0f	float	2996.65845	
(x)= event0_response_time/120.0f	float	2004.26672	
(x)= event1_response_time/120.0f	float	3008.0166	
(x)= event2_response_time/120.0f	float	14004.4082	
(x)= event0_missed_deadlines	unsigned long	0	0x200015F4
(x)= event1_missed_deadlines	unsigned long	0	0x200015F8
(x)= event2_missed_deadlines	unsigned long	115	0x200015FC
Add new expression			

ece3849_rtos_latency

- In ece3849_rtos_int_latency, the latency and response measurements were in the ISR routine.
- In ece3849_rtos_latency, the measurements have been moved to a task.
  - Tasks are lower priority than interrupts.
  - All interrupts must be serviced before any task can start running again.
  - All interrupts will preempt tasks independent of the tasks priority.
  - Max task latency = execution time of all ISRs + time to schedule the task + context switch time.

ece3849_rtos_latency

```
89 void ISR_Event0(UArg arg0)
90 {
91     TIMER0_ICR_R = TIMER_ICR_TATOCINT; // clear interrupt flag
92     Semaphore_post(semaphore0);
93 }
```

ISR now only clears the peripheral flag and posts to a semaphore.

```
107 void task0_func(UArg arg0, UArg arg1)
108 {
109     unsigned long t;
110     IArg key;
111
112     while(1) {
113         Semaphore_pend(semaphore0, BIOS_WAIT_FOREVER);
114
115         t = TIMER0_PERIOD - TIMER0_TAR_R;
116         if (t > event0_latency) event0_latency = t; // measure la
117
118         delay_us(EVENT0_EXECUTION_TIME); // handle event0
119         event0_count++;
120
121         if (Semaphore_getCount(semaphore0)) { // next event occur
122             event0_missed_deadlines++;
123             t = 2 * TIMER0_PERIOD; // timer overflowed since last
124         }
125         else t = TIMER0_PERIOD;
126         t -= TIMER0_TAR_R;
127         if (t > event0_response_time) event0_response_time = t; /
128     }
129 }
```

The task now contains the same, latency and response time calculations that used to be in the interrupt.

Instead of checking the Interrupt flag it now checks the semaphore status to see if it missed its deadline.

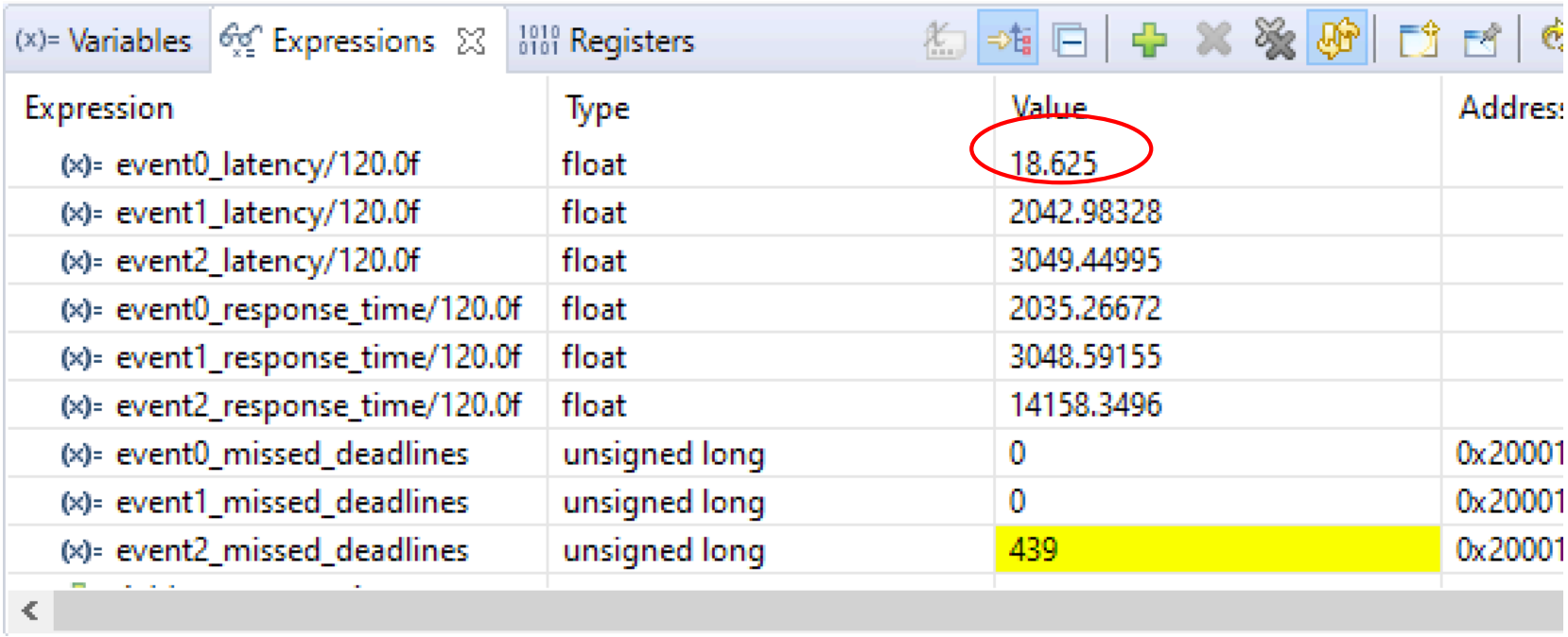
If a post occurred while in the task, then it missed the deadline.

Any guess on how much more latency the highest priority task will have, than its related interrupt?

- ?

ece3849_rtos_latency

- The highest priority task latency is 18 usec.
  - Highest task latency is 4x time longer than the highest priority RTOS interrupt latency of 4 usec.
  - Makes sense there are three interrupts, each with at least 4 usec of latency due to scheduling time.
  - Plus the execution time of each interrupt.
  - Plus the time to prioritize and schedule the tasks once the interrupts are serviced.

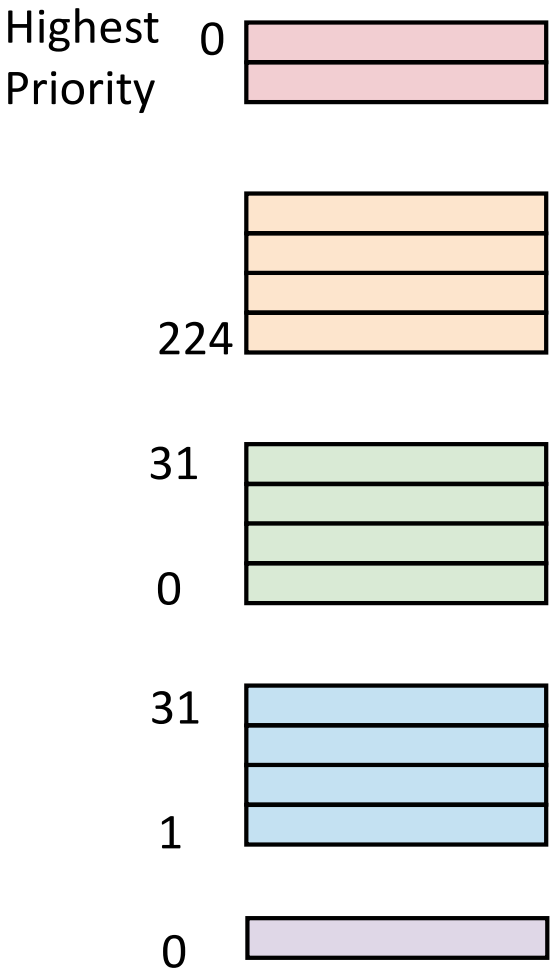


Expression	Type	Value	Addres
(x)= event0_latency/120.0f	float	18.625	
(x)= event1_latency/120.0f	float	2042.98328	
(x)= event2_latency/120.0f	float	3049.44995	
(x)= event0_response_time/120.0f	float	2035.26672	
(x)= event1_response_time/120.0f	float	3048.59155	
(x)= event2_response_time/120.0f	float	14158.3496	
(x)= event0_missed_deadlines	unsigned long	0	0x20001
(x)= event1_missed_deadlines	unsigned long	0	0x20001
(x)= event2_missed_deadlines	unsigned long	439	0x20001

Software Interrupts (Swi)

- Software Interrupts (Swi) behave like Hardware Interrupts (Hwi) except they are triggered by software.
  - They can be triggered by..
    - An ISR – Hardware Interrupt (Hwi).
    - Another software interrupt (Swi).
    - A task.
  - They can unblock other tasks.
  - ISRs (Hwi) and Swi share the same stack.
  - Swi are lower priority than Hwi.
- Hwi and Swi are not allowed to block.
  - Their behavior is optimized to be quick and efficient, blocking would destroy the purpose of the ISR.
  - Their latency and response time is critical. Context switches for preemption are very fast.
    - Once the preemption is done, the context is destroyed.
  - Context switching for blocking operations is complex and time consuming.
    - Task have their own dedicated stacks support this complexity.

Thread Priority



Zero Latency Hwi / ISR
(RTOS can not disable)

Hwi Interrupts /ISR
(RTOS controlled)

Hardware Interrupt Priorities
Lower # = higher priority
0 = highest priority
8 priority levels each
0, 32, 64, 96….

Swi – Software Interrupts

Tasks

IDLE task
(runs when nothing else to do)

RTOS Priority Numbers
Higher # = highest priority
Default highest = 15 in RTOS

IDLE task = 0, lowest priority.

Semaphore: Sharing resources

- Semaphores protect access to a shared resource.
  - They control the blocking and unblocking of tasks.
  - It is like a token or a key.
  - In Ti-RTOS they are called using Semaphore_pend() and Semaphore_post() commands.
    - Other OS's may use key words like get / give, take / release, p / v, or wait / signal.

- Semaphore Analogy: Sharing a key to a room.
  - There is a door to a room with one key that unlocks it, the key is hanging on a hook.
    - Room = shared data
    - Key = semaphore
  - One person at a time is allowed in the room. The first person to arrive takes the key, opens the door and goes inside with it.
  - The door locks behind them.
  - The next person to arrive sees the empty hook. They can not go in and must wait until the key is returned to the hook.
  - If others come, they must also wait in line for the key.
  - The first person comes out of the room, returns the key to the hook.
  - The next person can use the key to unlock the door and access the room.

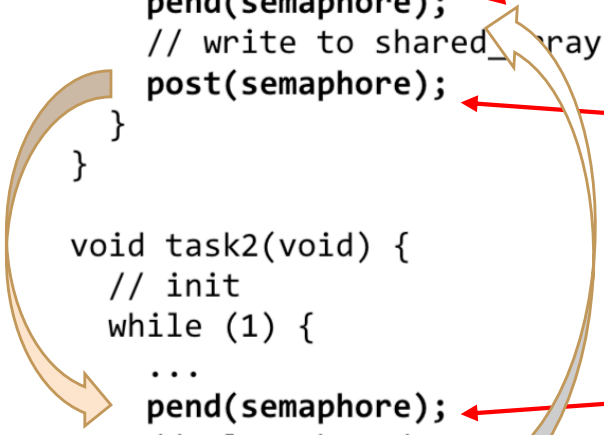
Shared resource: Data buffer

Canonical semaphore application: protection of access to shared data

```
int shared_array[10];

void task1(void) {
  // init
  while (1) {
    ...
    pend(semaphore);
    // write to shared_array (critical section)
    post(semaphore);
  }
}

void task2(void) {
  // init
  while (1) {
    ...
    pend(semaphore);
    // plot shared_array to display (critical section)
    post(semaphore);
  }
}
```



- Initialize semaphore with a post.
- First person takes the key from the hook and enters the room.
- task1 pends on the first semaphore. Since there is a post already, it is unblocked and writes to the array.
- First person returns the key to the hook.
- task1 is done with its critical section posts to the semaphore and pends.
- Second person takes the key and enters the room.
- Task2 is pending, when task1 posts it is unblocked and reads from the array.
- Second person returns the key to the hook.
- task2 is done with its critical section and posts to the semaphore.

Semaphore: Count Value

- Binary Semaphore (single key)
  - 0 resource is not available.
  - 1 resource is available.
- Counting Semaphore (multiple keys on the same hook)
  - 0 resource is not available.
  - Non-zero resource is available.

```
pend(semaphore)
{
  if (semaphore.count > 0)
    semaphore.count--;
  else
    <block, placed on the semaphore waiting list>;
}
```

Pend()

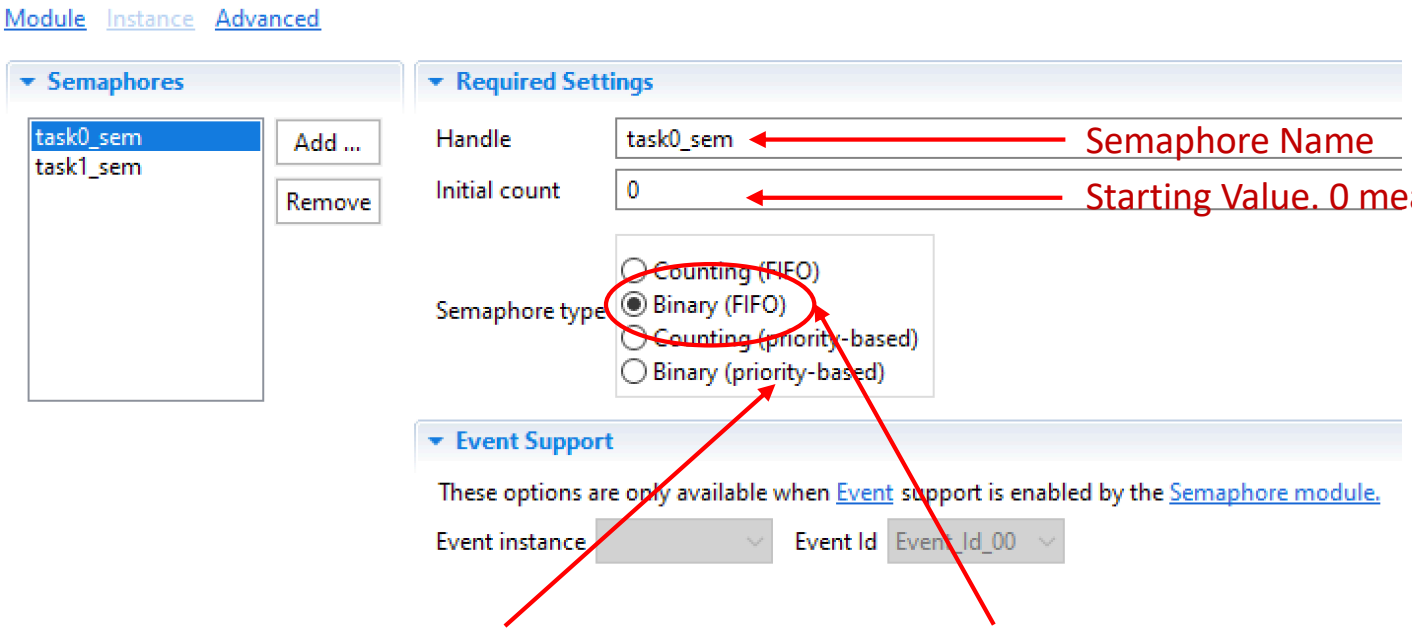
- If the key is available (count > 0)
  - Take the key and decrement the count.
- If it is not available, task is blocked and placed on wait list.

```
post(semaphore)
{
  if (<blocked task exists on the semaphore waiting list>)
    <unblock one task from the semaphore waiting list>;
  else
    semaphore.count++;
}
```

Post()

- If tasks are on the wait list, unblock the next task.
- If no tasks are waiting,
  - Increment count for counting semaphore.
  - Set count to 1 for binary semaphore.

Semaphore Configuration



If multiple tasks are pending,

- FIFO will prioritize based on who is first in the waiting list.
- Priority-based will unblock the highest priority task in the waiting list.

Binary has two states 0 = nothing posted, 1= at least one post.

If you have multiple events posting at a time,
A Counting Semaphore will…

- Increment count for each post, count++
- Decrement count for each pend, count--
- When the count is 0, nothing is posted and threads that pend will be blocked.

Semaphore: TI-RTOS Syntax

Semaphores in TI-RTOS

```
Bool Semaphore_pend(Semaphore_Handle handle, UInt timeout);
```

Wait for a semaphore

ARGUMENTS
handle — handle of a previously-created Semaphore instance object
timeout — return after this many system time units
RETURNS
TRUE if successful, FALSE if timeout
DETAILS
If the semaphore count is greater than zero (available), this function decrements the count and returns TRUE. If the semaphore count is zero (unavailable), this function suspends execution of the current task until post() is called or the timeout expires.
A timeout value of BIOS_WAIT_FOREVER causes the task to wait indefinitely for its semaphore to be posted.
A timeout value of BIOS_NO_WAIT causes Semaphore_pend to return immediately.

Semaphore_pend(task0_sem, BIOS_WAIT_FOREVER);

```
Void Semaphore_post(Semaphore_Handle handle);
```

Signal a semaphore

Semaphore_post(task0_sem);

ARGUMENTS
handle — handle of a previously-created Semaphore instance object
DETAILS
Readies the first task waiting for the semaphore. If no task is waiting, this function simply increments the semaphore count and returns. In the case of a binary semaphore, the count has a maximum value of one.

Tasks use BIOS_WAIT_FOREVER and provides the blocking functionality discussed in lecture.

ISRs (Hwi) or Software interrupts (Swi) that can not block.

- They use BIOS_NO_WAIT to simply decrement the count but do not pend.

Semaphore: Task Scheduling Example

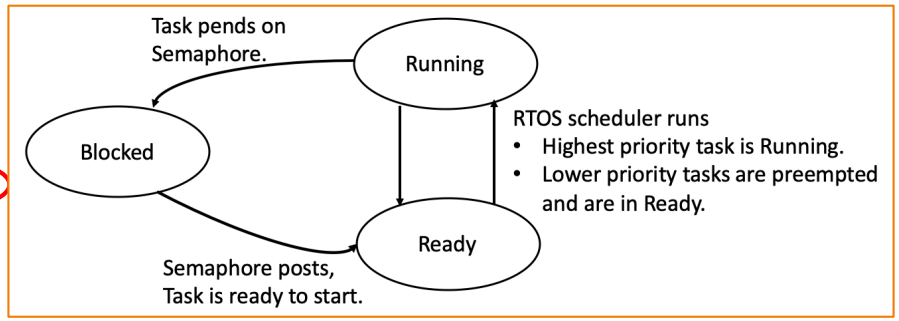
Protection of critical sections

Initialize semaphore `sem0` to count = **1** before it is first used (do not initialize inside a task).

```
int shared_array[10];

void task1(UArg arg0, UArg arg1) {  // high priority
  // init
  while (1) {
    Semaphore_pend(semTask1, BIOS_WAIT_FOREVER); // <- start Blocked
    ...
    Semaphore_pend(sem0, BIOS_WAIT_FOREVER);
    // write to shared_array (critical section)
    Semaphore_post(sem0);
  }
}

void task2(UArg arg0, UArg arg1) {  // low priority
  // init
  while (1) {
    ...        // <- start in Running state
    Semaphore_pend(sem0, BIOS_WAIT_FOREVER);
    // plot shared_array to display (critical section)
    Semaphore_post(sem0);
  }
```



Task1 interrupt occurs and posts to semTask1

Step	Action	task1	task2	sem0		semTask1	
				count	list	count	list
	Start	Blocked	Running	1	-	0	task1
1							
2							
3							
4							
5							
6							
7							
8							

Semaphore: Task State Example

```
int shared_array[10];

void task1(UArg arg0, UArg arg1) {  // high priority
   // init
   while (1) {
③     Semaphore_pend(semTask1, BIOS_WAIT_FOREVER); // <- start Blocked
      ...
④⑦   Semaphore_pend(sem0, BIOS_WAIT_FOREVER);
      // write shared_array (critical section)
⑧     Semaphore_post(sem0);
   }
}

void task2(UArg arg0, UArg arg1) {  // low priority
   // init
   while (1) {
      ...         // <- start in Running state
①     Semaphore_pend(sem0, BIOS_WAIT_FOREVER);
②⑤   // plot shared_array data to display (critical section)
⑥     Semaphore_post(sem0);
   }
}
```

Task1 interrupt occurs and posts to semTask1

Step	Action	task1	task2	sem0		semTask1	
				count	list	count	list
	Start	Blocked	Running	1	-	0	task1
1	task2 pends on sem0	Blocked	Running	0	-	0	task1
2	Interrupt: posts to semTask1	Ready	Running	0	-	0	-
3	Scheduler runs when ISR returns	Running	Ready	0	-	0	-
4	task1 pends on sem0	Blocked	Ready	0	task1	0	-
5	Scheduler runs	Blocked	Running	0	task1	0	-
6	task2 posts to sem0	Ready	Running	0	-	0	-
7	Scheduler runs	Running	Ready	0	-	0	-
8	task1 posts to sem0	Running	Ready	1	-	0	-

Semaphore Uses and Problems

- Semaphore uses.
  - Protection of critical sections.
  - Signaling & task synchronization.
  - Building block for other inter-task communication objects.
    - Mailboxes and message queues.
- Semaphores have an advantage over interrupts.
  - They only affect the run time of tasks for that semaphore.
  - ISRs add latency to all tasks as they have higher priority and would preempt all tasks.
- Semaphore disadvantage.
  - Limited uses in ISRs. ISRs can not pend on semaphores.
  - Increases CPU usage to call the semaphores.
  - Easy to make mistakes.
  - May cause priority inversion.
- Common errors and problems
  - Forgetting to pend on the semaphore.
    - Will not protect the shared data but will appear to be working.
  - Forgetting to post the semaphore.
    - Tasks may pend forever hanging the system.
  - Pending on the wrong semaphore.
    - Will not protect the shared data and cause other tasks to pend for ever.
  - Holding the a Semaphore too long.
    - It is important to protect only the critical section or response time of other shared tasks will be increased.