Real-time Embedded systems

Lab5 Interrupt Management

Example 12 Using a binary semaphore to synchronize a task with an interrupt

This example uses a binary semaphore to unlock a task from an interrupt service routine – so as to effectively synchronize the task with the interrupt.

A simple periodic task *vPeriodTask(void \*pvParameters)* is used to request user to press the button SW0 on the board.

The implementation of the handler task *vHandlerTask (void \*pvParameters)* – the task that is synchronized with the software interrupt through the use of a binary semaphore, is also given. A message is printed out from each iteration of the task, so the sequence in which the task and the interrupt execute is evident from the output produced when the example is executed.

The interrupt service routine *extint\_detection\_callback (void)* is an interrupt handler for the external hardware interrupt (i.e., pressing SW0). It does very little other than ‘give’ the semaphore to unblock the handler task. The macro *portEND\_SWITCHING\_ISR()* is part of the FreeRTOS Cortex-M3 port and is the ISR safe equivalent of *taskYIELD()*. It will force a context switch only if its parameter is not zero (i.e., not equal to pdFALSE).

Note how *xHigherPriorityTaskWoken* is used.

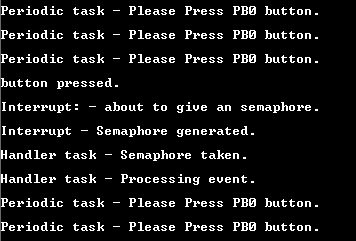
1) It is initialized to pdFALSE before being passed by reference into *xSemaphoreGiveFromISR*(), where it will be set to pdTRUE only if *xSemaphoreGiveFromISR*() causes a task of equal or higher priority than the currently executing task to leave the blocked state.

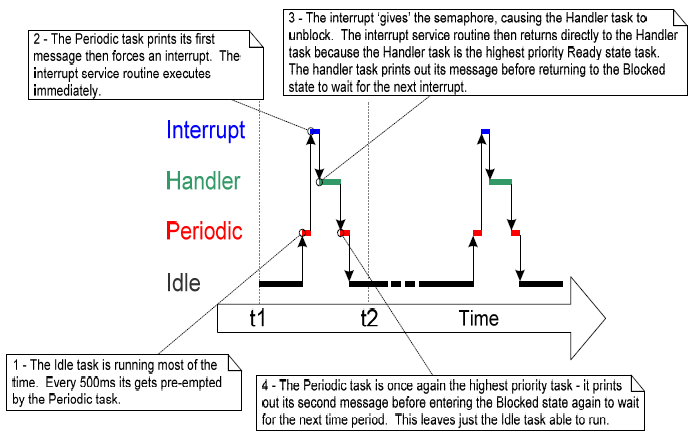
2). Then, *portEND\_SWITCHING\_ISR()* performs a context switching only if *xHigherPriorityTaskWoken* equals pdTRUE. In all other cases, a context switching is not necessary as the task that was executing before the interrupt occurs will still be the highest priority task that is able to run.

The *main()* function creates the binary semaphore and the tasks, configures the software interrupt, and starts the scheduler.

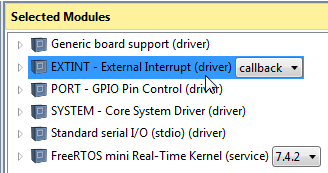
Example 12 produces the output shown in the following figure. As expected, the handler task executes as soon as the interrupt is generated, so the output from the handler task splits the output produced by the periodic task.

The execution sequence follows the pattern below.





Note: please use the ASF wizard to add the EXTINT module into the project.



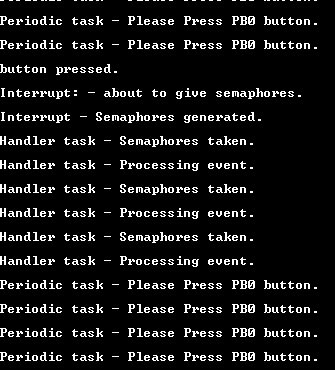
Example 13. Using a counting semaphore to synchronize a task with an interrupt

This example improves the example 12 implementation by using a counting semaphore in place of the binary semaphore.

The main() function is changed to include a call to xSemaphoreCreateCounting() in place of the call to vSemaphoreCreatedBinary().

To simulate multiple events occurring at high frequency, the interrupt service routine is changed to ‘give’ the semaphore more than once per interrupt. Each event is latched in the semaphore’s count value. All other functions remain unmodified from those used in Example 12.

The output produced is shown below. As can be seen, the Handler task processes all three events each time an interrupt is generated. The events are latched into the count value of the semaphore, allowing the Handler task to process them in turn.



Using Queues within an Interrupt Service Routine.

Example 14. Sending and receiving on a queue from within an interrupt

This example demonstrate *xQueueSendToBackFromISR*() and *xQueueuReceiveFromISR*() being used within the same interrupt. As before, a software interrupt is used for convenience.

A periodic task *vIntegerGenerator(void \*pvParamters)* is created that sends five numbers to a queue every 200 milliseconds. It generates a software interrupt only after all five values have been sent.

The interrupt service routine *vSoftwareInterruptHandler(void)* calls *xQueueReceiveFromISR*() repeatedly, until all values written to the queue by the periodic task have been removed, and the queue is left empty. The last two bits of each received valued are used as an index into an array of strings, with a pointer to the string at the corresponding index position being sent to a different queue using a call to *xQueueSendFromISR()*.

The task *puts(void \*pvParameters)* that receives the character pointers from the interrupt service routine blocks on the queue until a message arrives, printing out each string as it is received.

The *main()* function creates the required queues and tasks before starting the scheduler.

The output produced is shown below. As can be seen, the interrupt receives all five integers and produces five strings in response.

