|  |
| --- |
|  |
| **EE489 / EE589 Real-Time Embedded Systems Design** |
| Lab #5  *Dianzhi Yu* |
|  |
| *2/26/2018* |

|  |
| --- |
|  |

# Introduction

A Binary Semaphore can be used to unblock a task each time when a particular interrupt occurs, effectively synchronizing the task with the interrupt. This allows the majority interrupt event processing to be implemented within the synchronized task, with only a very fast and short portion remaining directly in the ISR. The interrupt processing is said to have been ‘deferred’ to a ‘handler’ task.

1. **A list of all FreeRTOS API functions being used**

vPeriodTask(void \*pvParameters);

This function is used to request user to press the button SW0 on the board.

portBASE\_TYPE xHigherPriorityTaskWoken = pdFALSE;

Initialize the xHigherPriorityTaskWoken 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.

portEND\_SWITCHING\_ISR(xHigherPriorityTaskWoken);

This function performs a context switching only if xHigherPriorityTaskWoken equals pdTRUE.

xSemaphoreGiveFromISR(binarySem,xHigherPriorityTaskWoken);

The first parameter is the global variable that is used to reference the semaphore that is used to synchronize a task with an interrupt. The second parameter is the address pointed to the variable xHigherPriorityTaskWoken. 'Give' the semaphore to unblock the task.

xSemaphoreTake(binarySem,0);

Take the semaphore once to start with so the semaphore is empty before the infinite loop is entered. The semaphore was created before the scheduler was started so before this task ran for the first time.

xSemaphoreTake(binarySem, portMAX\_DELAY);

Use the semaphore to wait for the event. The task blocks indefinitely meaning this function call will only return once the semaphore has been successfully obtained, so there is no need to check the returned value.

vSemaphoreCreateBinary(binarySem);

This function is to create a binary semaphore

xSemaphoreCreateCounting(10,0);

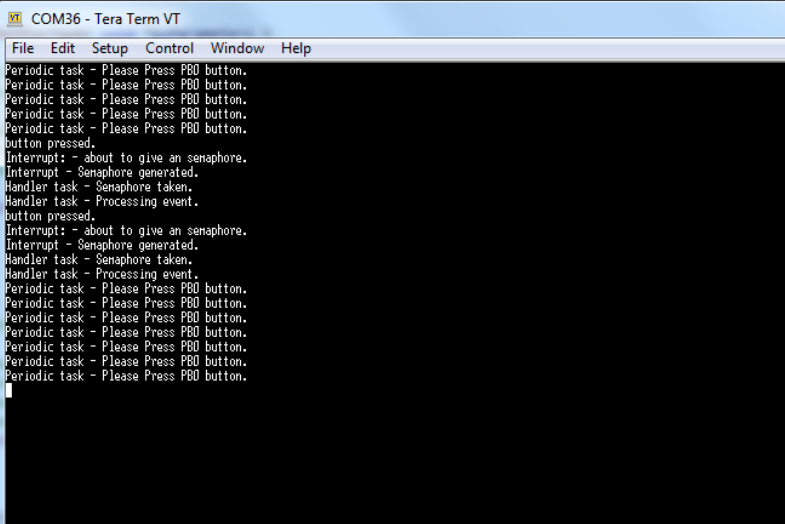
This function is to create a counting semaphore. The semaphore is created to have a maximum count value of 10, and an initial count value of 0. This function can solve some situations like, queues having a length of more than one, or when a counting S is ‘given’, another space in its queue is used. Tasks are interested in whether a queue is empty or not, not the data stored in the queue.

xQueueReceiveFromISR( xIntegerQueue, &ulReceivedNumber, &xHigherPriorityTaskWoken );

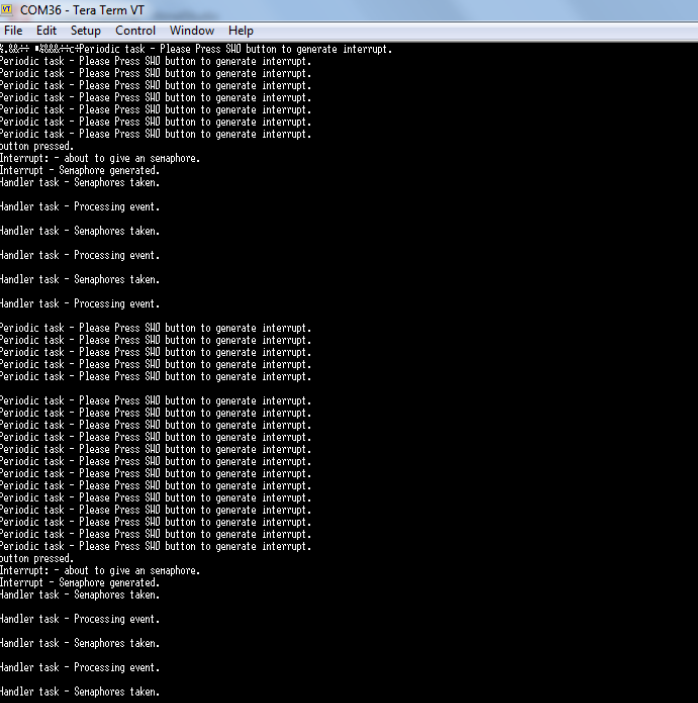
xQueueSendToBackFromISR(xStringQueue,&pcStrings[ulReceivedNumber],xHigherPriorityTaskWoken );

ISR calls this function until all values written to the queue by the periodic task have been removed, and the queue is left empty. And send the string to a different queue using a call to xQueueSendFromISR().

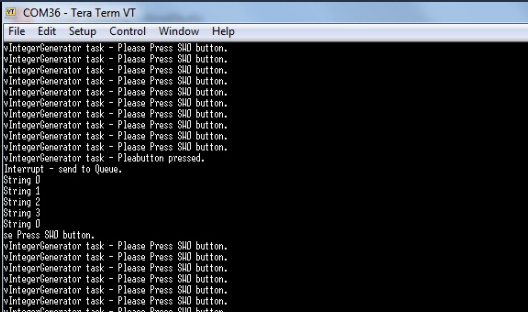
1. **Screen shots of the program execution results or debug windows of Keil µVision**



The periodic task asks an interrupt and when the user pushed the button, then the interrupt executes immediately. The interrupt ‘gives’ semaphore which leads to the Handler task unblock.



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.



When the user presses button to cause the interrupt, the ISR reads from a queue and writes to a queue. Writing a string also unblocks the StringPrinter task.

1. **Conclusion**

In example12, the periodic task preempts the idle task to ask user to push the button. And then interrupt task gives the semaphore to unblock the handler task. The interrupt service routine returns directly to the handler task because it has highest priority. Example 13 improves the example 12 implementation by using a counting semaphore in place of the binary semaphore. In example 14, sending and receiving on a queue within an interrupt which is used to communicate events and to transfer data. When the IntegerGenerator writes 5 values to a queue, then an interrupt executes. The ISR both reads from and writes to a queue.

1. **Appendix: The source code (main.c) with sufficient comments.**

/\*\*EX12

\* \mainpage User Application template doxygen documentation

\*

\* \par Empty user application template

\*

\* Bare minimum empty user application template

\*

\* \par Content

\*

\* -# Include the ASF header files (through asf.h)

\* -# "Insert system clock initialization code here" comment

\* -# Minimal main function that starts with a call to board\_init()

\* -# "Insert application code here" comment

\*

\*/

/\*

\* Include header files for all drivers that have been imported from

\* Atmel Software Framework (ASF).

\*/

#include <asf.h>

#include "semphr.h"

#include "task.h"

/\* To-do: Declare a global variable "binarySem" with the type xSemaphoreHandle.

This is used to reference the semaphore that is used to synchronize a task with an interrupt. \*/

xSemaphoreHandle binarySem;

/\* The tasks to be created. \*/

static void vHandlerTask( void \*pvParameters );

static void vPeriodicTask( void \*pvParameters );

/\* The service routine for the interrupt. This is the interrupt that the

task will be synchronized with. \*/

void externalInterruptHandler( void );

void vApplicationIdleHook(void)

{

while(1);

}

/\*\* UART module for debug. \*/

static struct usart\_module cdc\_uart\_module;

/\*\*

\* \brief Configure UART console.

\*/

static void configure\_console(void)

{

struct usart\_config usart\_conf;

usart\_get\_config\_defaults(&usart\_conf);

usart\_conf.mux\_setting = EDBG\_CDC\_SERCOM\_MUX\_SETTING;

usart\_conf.pinmux\_pad0 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD0;

usart\_conf.pinmux\_pad1 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD1;

usart\_conf.pinmux\_pad2 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD2;

usart\_conf.pinmux\_pad3 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD3;

usart\_conf.baudrate = 115200;

stdio\_serial\_init(&cdc\_uart\_module, EDBG\_CDC\_MODULE, &usart\_conf);

usart\_enable(&cdc\_uart\_module);

}

/\* Used as a loop counter to create a very crude delay. \*/

#define mainDELAY\_LOOP\_COUNT ( 0x11ffff )

void configure\_extint\_channel(void)

{

struct extint\_chan\_conf config\_extint\_chan;

extint\_chan\_get\_config\_defaults(&config\_extint\_chan);

config\_extint\_chan.gpio\_pin = BUTTON\_0\_EIC\_PIN;

config\_extint\_chan.gpio\_pin\_mux = BUTTON\_0\_EIC\_MUX;

config\_extint\_chan.gpio\_pin\_pull = EXTINT\_PULL\_UP;

config\_extint\_chan.detection\_criteria = EXTINT\_DETECT\_BOTH;

extint\_chan\_set\_config(BUTTON\_0\_EIC\_LINE, &config\_extint\_chan);

}

void extint\_detection\_callback(void)

{

portBASE\_TYPE xHigherPriorityTaskWoken = pdFALSE;

/\* Check if EIC push button line interrupt line is pending. \*/

printf("button pressed. \r\n");

printf( "Interrupt: - about to give an semaphore.\r\n" );

/\*

To-do: call the xSemaphoreGiveFromISR() API function.

The first parameter is the global binary semaphore you created above,

The second parameter is the address pointed to the variable xHigherPriorityTaskWoken.

'Give' the semaphore to unblock the task. \*/

xSemaphoreGiveFromISR(binarySem,xHigherPriorityTaskWoken);

/\* Giving the semaphore may have unblocked a task - if it did and the

unblocked task has a priority equal to or above the currently executing

task then xHigherPriorityTaskWoken will have been set to pdTRUE and

portEND\_SWITCHING\_ISR() will force a context switch to the newly unblocked

higher priority task.

NOTE: The syntax for forcing a context switch within an ISR varies between

FreeRTOS ports. The portEND\_SWITCHING\_ISR() macro is provided as part of

the Cortex M3 port layer for this purpose. taskYIELD() must never be called

from an ISR! \*/

portEND\_SWITCHING\_ISR( xHigherPriorityTaskWoken );

printf( "Interrupt - Semaphore generated.\r\n" );

}

void configure\_extint\_callbacks(void)

{

extint\_register\_callback(extint\_detection\_callback, BUTTON\_0\_EIC\_LINE,

EXTINT\_CALLBACK\_TYPE\_DETECT);

extint\_chan\_enable\_callback(BUTTON\_0\_EIC\_LINE, EXTINT\_CALLBACK\_TYPE\_DETECT);

}

/\*-----------------------------------------------------------\*/

int main( void )

{

system\_init();

configure\_console();

printf("Experiment 12 starts\r\n");

configure\_extint\_channel();

configure\_extint\_callbacks();

system\_interrupt\_enable\_global();

/\*

To-do: Call the vSemaphoreCreateBinary() API to create a binary semaphore.

The input parameter is the variable you declared above.

Before a semaphore is used it must be explicitly created. In this example

a binary semaphore is created. \*/

vSemaphoreCreateBinary(binarySem);

/\* To-do: use "if(condition)" expression to check the semaphore was created successfully.

Note if success, the binary semaphore variable will not be NULL. \*/

if(binarySem!=NULL)

{

/\* Create the 'handler' task. This is the task that will be synchronized

with the interrupt. The handler task is created with a high priority to

ensure it runs immediately after the interrupt exits. In this case a

priority of 3 is chosen. \*/

xTaskCreate( vHandlerTask, "Handler", 240, NULL, 3, NULL );

/\* Create the task that will periodically generate a software interrupt.

This is created with a priority below the handler task to ensure it will

get preempted each time the handler task exits the Blocked state. \*/

xTaskCreate( vPeriodicTask, "Periodic", 240, NULL, 1, NULL );

/\* Start the scheduler so the created tasks start executing. \*/

vTaskStartScheduler();

}

/\* If all is well we will never reach here as the scheduler will now be

running. If we do reach here then it is likely that there was insufficient

heap available for the idle task to be created. \*/

for( ;; );

}

static void vHandlerTask( void \*pvParameters )

{

/\* To-do: Call xSemaphoreTake() API. The first parameter is the binary semaphore

variable you created, the second parameter is set to 0.

Take the semaphore once to start with so the semaphore is empty before the

infinite loop is entered. The semaphore was created before the scheduler

was started so before this task ran for the first time.\*/

xSemaphoreTake(binarySem,0);

for( ;; )

{

/\*

To-do: Call xSemaphoreTake() API again. The first parameter is still the global binary semaphore

variable you created above, the second parameter is set as portMAX\_DELAY.

Use the semaphore to wait for the event. The task blocks

indefinitely meaning this function call will only return once the

semaphore has been successfully obtained - so there is no need to check

the returned value. \*/

xSemaphoreTake(binarySem, portMAX\_DELAY);

printf( "Handler task - Semaphore taken.\r\n" );

/\* To get here the event must have occurred. Process the event (in this

case we just print out a message). \*/

printf( "Handler task - Processing event.\r\n" );

}

}

/\*-----------------------------------------------------------\*/

static void vPeriodicTask( void \*pvParameters )

{

/\* As per most tasks, this task is implemented within an infinite loop. \*/

for( ;; )

{

/\* This task is just used to 'simulate' an interrupt. This is done by

periodically generating a software interrupt. \*/

vTaskDelay( 500 / portTICK\_RATE\_MS );

/\* Generate the interrupt, printing a message both before hand and

afterwards so the sequence of execution is evident from the output. \*/

printf( "Periodic task - Please Press PB0 button.\r\n" );

}

}

/\*\*

\* \brief Interrupt handler for EIC interrupt.

\*/

#ifdef test

void externalInterruptHandler(void)

{

portBASE\_TYPE xHigherPriorityTaskWoken = pdFALSE;

/\* Check if EIC push button line interrupt line is pending. \*/

//if (eic\_line\_interrupt\_is\_pending(EIC, GPIO\_PUSH\_BUTTON\_EIC\_LINE)) {

// eic\_line\_clear\_interrupt(EIC, GPIO\_PUSH\_BUTTON\_EIC\_LINE);

printf("button pressed. \r\n");

printf( "Interrupt: - about to give an semaphore.\r\n" );

/\*

To-do: call the xSemaphoreGiveFromISR() API function.

The first parameter is the global binary semaphore you created above,

The second parameter is the address pointed to the variable xHigherPriorityTaskWoken.

'Give' the semaphore to unblock the task. \*/

xSemaphoreGiveFromISR(binarySem,xHigherPriorityTaskWoken);

/\* Giving the semaphore may have unblocked a task - if it did and the

unblocked task has a priority equal to or above the currently executing

task then xHigherPriorityTaskWoken will have been set to pdTRUE and

portEND\_SWITCHING\_ISR() will force a context switch to the newly unblocked

higher priority task.

NOTE: The syntax for forcing a context switch within an ISR varies between

FreeRTOS ports. The portEND\_SWITCHING\_ISR() macro is provided as part of

the Cortex M3 port layer for this purpose. taskYIELD() must never be called

from an ISR! \*/

portEND\_SWITCHING\_ISR( xHigherPriorityTaskWoken );

printf( "Interrupt - Semaphore generated.\r\n" );

}

}

#endif

/\*\*EX13

\* \mainpage User Application template doxygen documentation

\*

\* \par Empty user application template

\*

\* Bare minimum empty user application template

\*

\* \par Content

\*

\* -# Include the ASF header files (through asf.h)

\* -# "Insert system clock initialization code here" comment

\* -# Minimal main function that starts with a call to board\_init()

\* -# "Insert application code here" comment

\*

\*/

/\*

\* Include header files for all drivers that have been imported from

\* Atmel Software Framework (ASF).

\*/

#include <asf.h>

#include "semphr.h"

#include "task.h"

/\* To-do:

Declare a variable named as "xCountingSemaphore" of the type xSemaphoreHandle.

This is used to reference the semaphore that is used to synchronize a task with an interrupt. \*/

xSemaphoreHandle xCountingSemaphore;

/\* The tasks to be created. \*/

static void vHandlerTask( void \*pvParameters );

static void vPeriodicTask( void \*pvParameters );

void extint\_detection\_callback(void);

void vApplicationIdleHook(void)

{

while(1);

}

/\*\* UART module for debug. \*/

static struct usart\_module cdc\_uart\_module;

/\*\*

\* \brief Configure UART console.

\*/

static void configure\_console(void)

{

struct usart\_config usart\_conf;

usart\_get\_config\_defaults(&usart\_conf);

usart\_conf.mux\_setting = EDBG\_CDC\_SERCOM\_MUX\_SETTING;

usart\_conf.pinmux\_pad0 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD0;

usart\_conf.pinmux\_pad1 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD1;

usart\_conf.pinmux\_pad2 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD2;

usart\_conf.pinmux\_pad3 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD3;

usart\_conf.baudrate = 115200;

stdio\_serial\_init(&cdc\_uart\_module, EDBG\_CDC\_MODULE, &usart\_conf);

usart\_enable(&cdc\_uart\_module);

}

/\* Used as a loop counter to create a very crude delay. \*/

#define mainDELAY\_LOOP\_COUNT ( 0x11ffff )

void configure\_extint\_channel(void)

{

struct extint\_chan\_conf config\_extint\_chan;

extint\_chan\_get\_config\_defaults(&config\_extint\_chan);

config\_extint\_chan.gpio\_pin = BUTTON\_0\_EIC\_PIN;

config\_extint\_chan.gpio\_pin\_mux = BUTTON\_0\_EIC\_MUX;

config\_extint\_chan.gpio\_pin\_pull = EXTINT\_PULL\_UP;

config\_extint\_chan.detection\_criteria = EXTINT\_DETECT\_RISING; //EXTINT\_DETECT\_BOTH;

extint\_chan\_set\_config(BUTTON\_0\_EIC\_LINE, &config\_extint\_chan);

}

void configure\_extint\_callbacks(void)

{

extint\_register\_callback(extint\_detection\_callback, BUTTON\_0\_EIC\_LINE,

EXTINT\_CALLBACK\_TYPE\_DETECT);

extint\_chan\_enable\_callback(BUTTON\_0\_EIC\_LINE, EXTINT\_CALLBACK\_TYPE\_DETECT);

}

/\*-----------------------------------------------------------\*/

int main( void )

{

system\_init();

configure\_console();

printf("Experiment 13 starts\r\n");

configure\_extint\_channel();

configure\_extint\_callbacks();

system\_interrupt\_enable\_global();

/\* To-do: Call xSemaphoreCreateCounting() API to create counting semaphore.

The first parameter is 10. The second one is 0.

The return value is assigned to the xCountingSemaphore variable you created above.

Before a semaphore is used it must be explicitly created. In this example

a counting semaphore is created. The semaphore is created to have a maximum

count value of 10, and an initial count value of 0. \*/

xSemaphoreCreateCounting(10,0);

/\* To-do: use "if(condition)" expression to check the semaphore was created successfully.

Note if success, the counting semaphore variable will not be NULL. \*/

if(xCountingSemaphore!=NULL)

{

/\* Create the 'handler' task. This is the task that will be synchronized

with the interrupt. The handler task is created with a high priority to

ensure it runs immediately after the interrupt exits. In this case a

priority of 3 is chosen. \*/

xTaskCreate( vHandlerTask, "Handler", 240, NULL, 3, NULL );

/\* Create the task that will periodically generate a software interrupt.

This is created with a priority below the handler task to ensure it will

get preempted each time the handler task exits the Blocked state. \*/

xTaskCreate( vPeriodicTask, "Periodic", 240, NULL, 1, NULL );

/\* Start the scheduler so the created tasks start executing. \*/

vTaskStartScheduler();

}

/\* If all is well we will never reach here as the scheduler will now be

running. If we do reach here then it is likely that there was insufficient

heap available for the idle task to be created. \*/

for( ;; );

}

static void vHandlerTask( void \*pvParameters )

{

for( ;; )

{

/\*

To-do: Call xSemaphoreTake() API. The first parameter is still the global semaphore variable xCountingSemaphore, the second parameter is set as portMAX\_DELAY.

Use the semaphore to wait for the event. The task blocks

indefinitely meaning this function call will only return once the

semaphore has been successfully obtained - so there is no need to check

the returned value. \*/

xSemaphoreTake(xCountingSemaphore, portMAX\_DELAY);

puts( "Handler task - Semaphores taken.\r\n" );

/\* To get here the event must have occurred. Process the event (in this

case we just print out a message). \*/

puts( "Handler task - Processing event.\r\n" );

}

}

/\*-----------------------------------------------------------\*/

static void vPeriodicTask( void \*pvParameters )

{

/\* As per most tasks, this task is implemented within an infinite loop. \*/

for( ;; )

{

/\* This task is just used to 'simulate' an interrupt. This is done by

periodically generating a software interrupt. \*/

vTaskDelay( 500 / portTICK\_RATE\_MS );

/\* Generate the interrupt, printing a message both before hand and

afterwards so the sequence of execution is evident from the output. \*/

printf( "Periodic task - Please Press SW0 button to generate interrupt.\r\n" );

}

}

void extint\_detection\_callback(void)

{

volatile bool pin\_state = port\_pin\_get\_input\_level(BUTTON\_0\_PIN);

port\_pin\_set\_output\_level(LED\_0\_PIN, pin\_state);

extint\_chan\_clear\_detected(BUTTON\_0\_EIC\_LINE);

portBASE\_TYPE xHigherPriorityTaskWoken = pdFALSE;

/\* Check if EIC push button line interrupt line is pending. \*/

printf("button pressed. \r\n");

printf( "Interrupt: - about to give an semaphore.\r\n" );

/\* To-do: Call the xSemaphoreGiveFromISR() API three times.

The first parameter is the counting Semaphore you created above.

The second parameter is the reference to the variable xHigherPriorityTaskWoken.

'Give' the semaphore three times. The first will unblock the handler

task, the following 'gives' are to demonstrate that the semaphore latches

the events to allow the handler task to process them in turn without any

events getting lost. This simulates multiple interrupts being taken by the

processor, even though in this case the events are simulated within a single

interrupt occurrence.\*/

xSemaphoreGiveFromISR(xCountingSemaphore,&xHigherPriorityTaskWoken);

xSemaphoreGiveFromISR(xCountingSemaphore,&xHigherPriorityTaskWoken);

xSemaphoreGiveFromISR(xCountingSemaphore,&xHigherPriorityTaskWoken);

/\* Giving the semaphore may have unblocked a task - if it did and the

unblocked task has a priority equal to or above the currently executing

task then xHigherPriorityTaskWoken will have been set to pdTRUE and

portEND\_SWITCHING\_ISR() will force a context switch to the newly unblocked

higher priority task.

NOTE: The syntax for forcing a context switch within an ISR varies between

FreeRTOS ports. The portEND\_SWITCHING\_ISR() macro is provided as part of

the Cortex M3 port layer for this purpose. taskYIELD() must never be called

from an ISR! \*/

portEND\_SWITCHING\_ISR( xHigherPriorityTaskWoken );

printf( "Interrupt - Semaphore generated.\r\n" );

}

/\*\*EX14

\* \mainpage User Application template doxygen documentation

\*

\* \par Empty user application template

\*

\* Bare minimum empty user application template

\*

\* \par Content

\*

\* -# Include the ASF header files (through asf.h)

\* -# "Insert system clock initialization code here" comment

\* -# Minimal main function that starts with a call to board\_init()

\* -# "Insert application code here" comment

\*

\*/

/\*

\* Include header files for all drivers that have been imported from

\* Atmel Software Framework (ASF).

\*/

#include <asf.h>

#include "semphr.h"

#include "task.h"

/\* The tasks to be created. \*/

static void vIntegerGenerator( void \*pvParameters );

static void vStringPrinter( void \*pvParameters );

/\*-----------------------------------------------------------\*/

unsigned char index = 0;

void extint\_detection\_callback(void);

/\* To-do:

Declare two variables of type xQueueHandle.

One queue xIntegerQueue holds integer items; the other xStringQueueu holds the string items.

Usage:

One queue will be read from within an ISR, the other will be written to within an ISR. \*/

xQueueHandle xIntegerQueue, xStringQueueu;

void vApplicationIdleHook(void)

{

while(1);

}

/\*\* UART module for debug. \*/

static struct usart\_module cdc\_uart\_module;

/\*\*

\* \brief Configure UART console.

\*/

static void configure\_console(void)

{

struct usart\_config usart\_conf;

usart\_get\_config\_defaults(&usart\_conf);

usart\_conf.mux\_setting = EDBG\_CDC\_SERCOM\_MUX\_SETTING;

usart\_conf.pinmux\_pad0 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD0;

usart\_conf.pinmux\_pad1 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD1;

usart\_conf.pinmux\_pad2 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD2;

usart\_conf.pinmux\_pad3 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD3;

usart\_conf.baudrate = 115200;

stdio\_serial\_init(&cdc\_uart\_module, EDBG\_CDC\_MODULE, &usart\_conf);

usart\_enable(&cdc\_uart\_module);

}

/\* Used as a loop counter to create a very crude delay. \*/

#define mainDELAY\_LOOP\_COUNT ( 0x11ffff )

void configure\_extint\_channel(void)

{

struct extint\_chan\_conf config\_extint\_chan;

extint\_chan\_get\_config\_defaults(&config\_extint\_chan);

config\_extint\_chan.gpio\_pin = BUTTON\_0\_EIC\_PIN;

config\_extint\_chan.gpio\_pin\_mux = BUTTON\_0\_EIC\_MUX;

config\_extint\_chan.gpio\_pin\_pull = EXTINT\_PULL\_UP;

config\_extint\_chan.detection\_criteria = EXTINT\_DETECT\_RISING; //EXTINT\_DETECT\_BOTH;

extint\_chan\_set\_config(BUTTON\_0\_EIC\_LINE, &config\_extint\_chan);

}

void configure\_extint\_callbacks(void)

{

extint\_register\_callback(extint\_detection\_callback, BUTTON\_0\_EIC\_LINE,

EXTINT\_CALLBACK\_TYPE\_DETECT);

extint\_chan\_enable\_callback(BUTTON\_0\_EIC\_LINE, EXTINT\_CALLBACK\_TYPE\_DETECT);

}

/\*-----------------------------------------------------------\*/

int main( void )

{

system\_init();

configure\_console();

printf("Experiment 14 starts\r\n");

configure\_extint\_channel();

configure\_extint\_callbacks();

system\_interrupt\_enable\_global();

/\* To-do: Call xQueueCreate() API functions twice to generate two queues.

The two parameters of the first xQueueCreate() call are 10 and sizeof(unsigned long) respectively,

the return value will be assigned to the queue variable declared above which holds integer items.

The two parameters of the second xQueueCreate() call are 10 and sizeof(char \*) respectively,

the return value will be assigned to the queue variable declared above which holds string items.

Before a queue can be used it must first be created. Create both queues

used by this example. One queue can hold variables of type unsigned long,

the other queue can hold variables of type char\*. Both queues can hold a

maximum of 10 items. A real application should check the return values to

ensure the queues have been successfully created. \*/

/\* Create the task that uses a queue to pass integers to the interrupt service

routine. The task is created at priority 1. \*/

xTaskCreate( vIntegerGenerator, "IntGen", 240, NULL, 1, NULL );

/\* Create the task that prints out the strings sent to it from the interrupt

service routine. This task is created at the higher priority of 2. \*/

xTaskCreate( vStringPrinter, "String", 240, NULL, 2, NULL );

/\* Start the scheduler so our tasks start executing. \*/

vTaskStartScheduler();

/\* If all is well we will never reach here as the scheduler will now be

running. If we do reach here then it is likely that there was insufficient

heap available for the idle task to be created. \*/

for( ;; );

}

static void vIntegerGenerator( void \*pvParameters )

{

portTickType xLastExecutionTime;

unsigned portLONG ulValueToSend = 0;

/\* Initialize the variable used by the call to vTaskDelayUntil(). \*/

xLastExecutionTime = xTaskGetTickCount();

for( ;; )

{

/\* This is a periodic task. Block until it is time to run again.

The task will execute every 200ms. \*/

vTaskDelayUntil( &xLastExecutionTime, 200 / portTICK\_RATE\_MS );

/\* Send an incrementing number to the queue five times. These will be

read from the queue by the interrupt service routine. A block time is

not specified. \*/

for(; index < 5; index++ )

{

/\* To-do: Call xQueueSendToBack() API.

The first parameter is the integer queue variable you declared at the beginning.

The second parameter is the reference to the variable ulValueToSend.

The last parameter is 0.

\*/

ulValueToSend++;

}

/\* Generate the interrupt, printing a message both before hand and

afterwards so the sequence of execution is evident from the output. \*/

printf( "vIntegerGenerator task - Please Press SW0 button.\r\n" );

}

}

static void vStringPrinter( void \*pvParameters )

{

char \*pcString;

for( ;; )

{

/\* To-do: Call xQueueReceive() API and pass three parameters.

The first parameter is the string queue variable you created at the beginning.

The second one is the address pointed to the string variable pcString.

The last one is portMAX\_DELAY. \*/

/\* Block on the queue to wait for data to arrive. \*/

/\* Print out the string received. \*/

printf( pcString );

}

}

void extint\_detection\_callback(void)

{

volatile bool pin\_state = port\_pin\_get\_input\_level(BUTTON\_0\_PIN);

port\_pin\_set\_output\_level(LED\_0\_PIN, pin\_state);

extint\_chan\_clear\_detected(BUTTON\_0\_EIC\_LINE);

portBASE\_TYPE xHigherPriorityTaskWoken = pdFALSE;

static unsigned long ulReceivedNumber;

/\* The strings are declared static const to ensure they are not allocated to the

interrupt service routine stack, and exist even when the interrupt service routine

is not executing. \*/

static const char \*pcStrings[] =

{

"String 0\r\n",

"String 1\r\n",

"String 2\r\n",

"String 3\r\n"

};

/\* Check if EIC push button line interrupt line is pending. \*/

printf("button pressed. \r\n");

// reset the index

index = 0;

/\* Loop until the queue is empty. \*/

while( xQueueReceiveFromISR( xIntegerQueue, &ulReceivedNumber, &xHigherPriorityTaskWoken )

!= errQUEUE\_EMPTY )

{

/\* Truncate the received value to the last two bits (values 0 to 3 inc.), then

send the string that corresponds to the truncated value to the other

queue. \*/

ulReceivedNumber &= 0x03;

/\*To-do:

If needed, Replace the first parameter "xStringQueue" of xQueueSendToBackFromISR() API with the

string Queue variable your created; otherwise, keep it unchanged.

\*/

xQueueSendToBackFromISR( xStringQueue, &pcStrings[ ulReceivedNumber ],

&xHigherPriorityTaskWoken );

}

portEND\_SWITCHING\_ISR( xHigherPriorityTaskWoken );

printf( "Interrupt - send to Queue.\r\n" );

}