ECEN-361 Lab-10:Interprocess Communication

NAME: ____

Introduction and Objectives of the Lab

This lab shows how various independent tasks can communicate through the RTOS-controlled signaling mechanisms. It also shows the good practice of handling interrupts by using semaphores to signal their respective tasks.

This lab will demonstrate the following Interprocess Communication mechanisms:

1.) Semaphores and Task-Notifications
2.) Mutexes
3.) Protected Global Variables
4.) Process Notification via Software Timers

In the provided FreeRTOS system, tasks are set up to use these mechanisms to communicate with each other. The mechanisms and associated tasks (and assignments) are described in the respective sections of the lab.

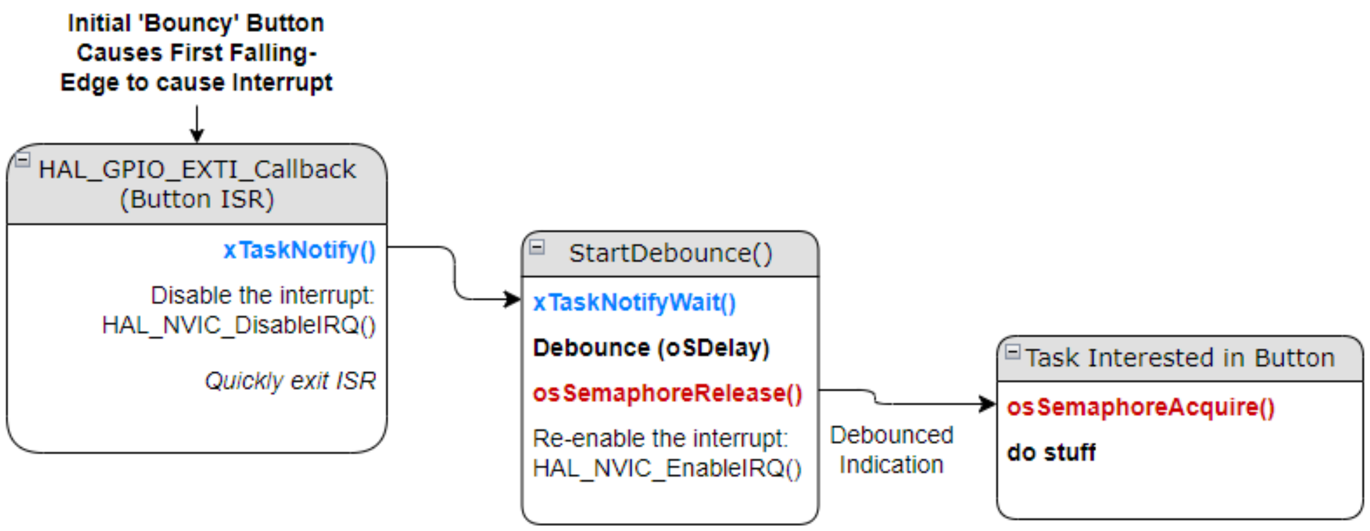
Part 1: RTOS-friendly debounced buttons – Notifications and Semaphores

As discussed in class, all mechanical buttons come with the mechanically generated issue of 'bouncing' which can create multiple input requests to a system where only a single entry is required. Methods to alleviate this problem include external RC circuits on the button input, dedicating a timer to the incoming button, and software counting loops to debounce the input. Note that the 'bounce' characteristics of the switch may vary, but in general they are much less than a human response time. Typically, a debounce time on the order of 10 -30 milliseconds is long enough to get rid of the mechanical flaw and not miss a human reaction time.

In an RTOS environment, the timeout necessary to debug a switch should not come within the ISR servicing the button event. Leaving the RTOS to 'wait' for a debounce inside of the ISR is bad practice as it can be blocking. One simple solution is to use notification mechanisms in the RTOS and trigger them from within the ISR. Two such mechanisms are '**Semaphores**' and '**Task-Notifications**'. **Semaphores** can be binary or counting and can track numbers of events from multiple sources. Task-Notifications are simply a signal from one process to another.

By using a signaling mechanism this way, we create a 'Thread-safe' interrupt service routine. This means that the ISR will not create unexpected behavior in a multi-process environment and supports scalable grow with more/less processes. It's 'safe' to threads.

For a task(s) waiting for a clean, debounced button event, the initial button press can't trigger the task(s), it must first trigger the debounce wait, that then starts the task(s):


Initial 'Bouncy' Button Causes First Falling-Edge to cause Interrupt
HAL_GPIO_EXTI_Callback (Button ISR)
xTaskNotify()
Disable the interrupt: HAL_NVIC_DisableIRQ()
Quickly exit ISR
StartDebounce()
xTaskNotifyWait()
Debounce (oSDelay)
osSemaphoreRelease()
Re-enable the interrupt: HAL_NVIC_EnableIRQ()
Debounced Indication
Task Interested in Button
osSemaphoreAcquire()
do stuff

Part 1 Instructions and Questions (2 Pts.)

1.) Create a Task (Either with the GUI or manually) that will toggle LED_D4 each time Button_1 is pressed.

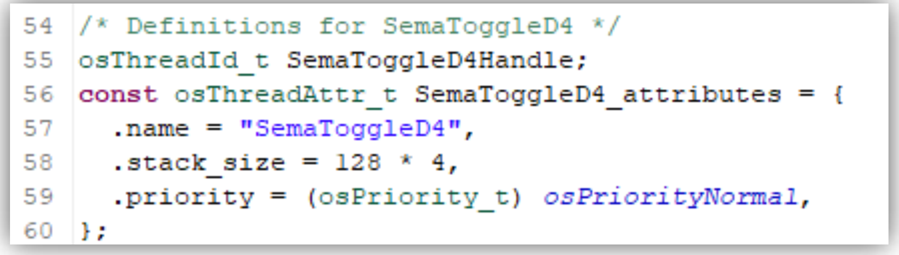
How did your task 'wait' for the debounced button?

____

2.) How long is the time between the button interrupt coming in and it being enabled again?

____

3.) You're given the framework of a second task (Semaphore_Toggle_D3) -- Modify this so that it that also waits for the same Button_1_Semaphore. It should then toggle LED_D3 each time Button_1 is pressed. Note that they both have been created with the same Priority..



```
/* Definitions for SemaToggleD4 */
osThreadId_t SemaToggleD4Handle;
const osThreadAttr_t SemaToggleD4_attributes = {
  .name = "SemaToggleD4",
  .stack_size = 128 * 4,
  .priority = (osPriority_t) osPriorityNormal,
};
```

4.) Do both of (D4 and D3) toggle with a single button press? Describe the behavior?

____

5.) Now change one of the priorities of these two tasks and re-run. How has the behavior changed?

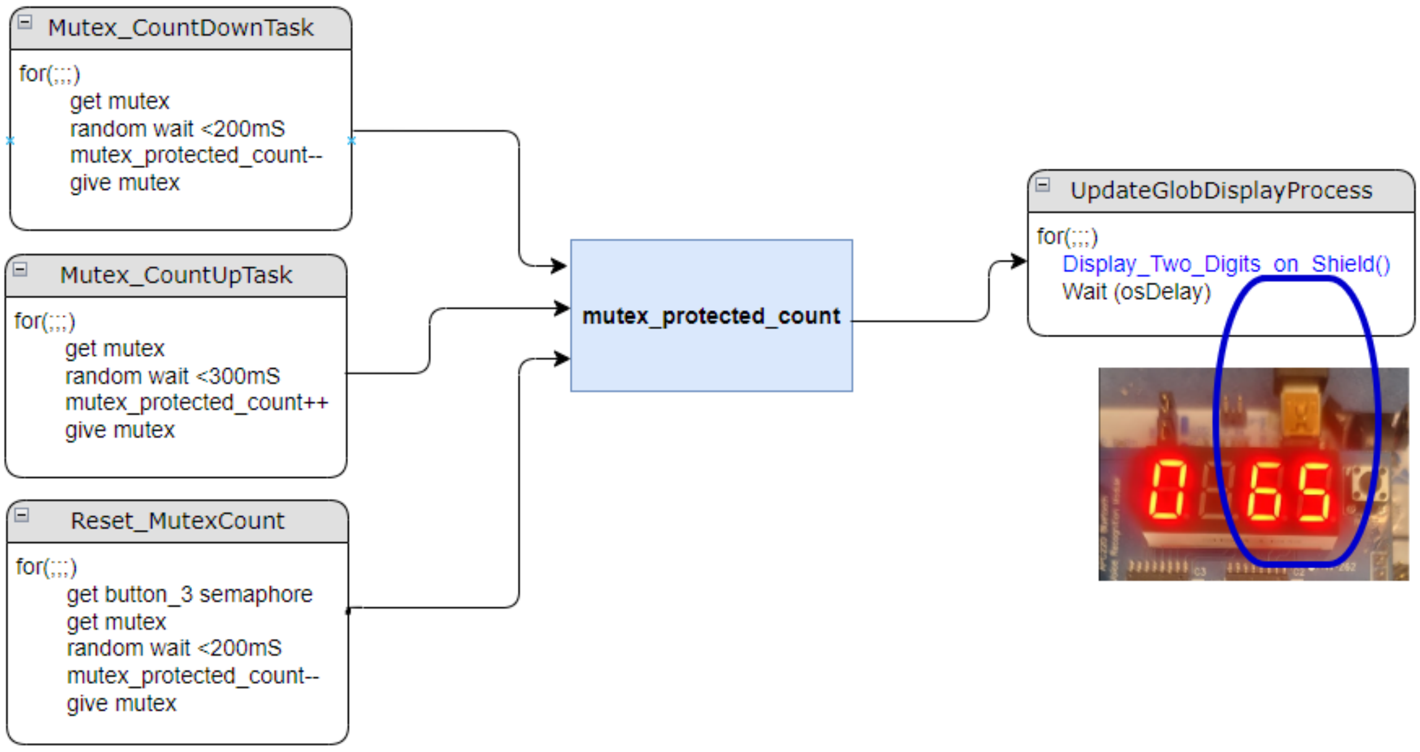
____

Part 2: Mutexes

Mutex’s are used in embedded systems to guarantee exclusive access to a resource that more than one process may need to use. Examples include hardware devices or shared memory locations that could be requested simultaneously.

For the hardware set of our labs, we have only a single 4-digit seven-segment display. If there were multiple producers trying to change the value of this display, one way to guarantee its integrity would be to protect it with a mutex: only one process at a time gets to send output to it.

For this example, the following picture shows how a mutex could protect a value being display on the seven-segment:



The three processes on the right are all competing for access to write to the protected variable. The **mutex_protected_count** is continuously updated to the seven-segment display on the right. The CountDown and CountUp are both trying at random intervals, and the Reset_MutexCount waits for the Button_3 (via its associated semaphore), then resets the current count.

Part 2 Questions (2 Pts.)

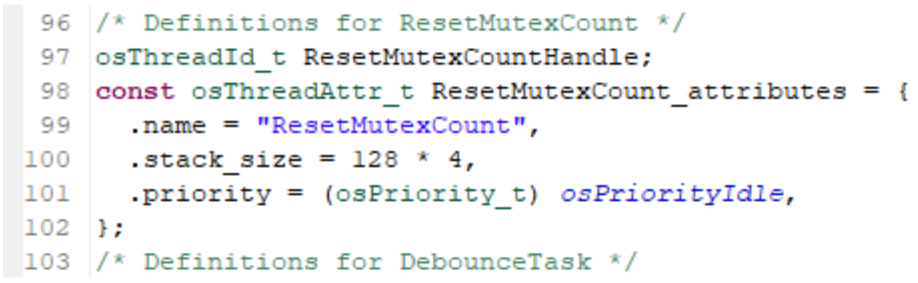
Add another process that uses the mutex to take control and put a “—” on the Two_Digit UpDown SevenSegment Display. Use the following code in your process:

```
for(;;)
{
/* This doesn't change the value, it just clears the display  */
/* If asked to display a negative number, the function displays a "--"
 */
osMutexWait(UpDownMutexHandle,100000);
MultiFunctionShield_Display_Two_Digits(-1);
osDelay(200);
osMutexRelease(UpDownMutexHandle);
osDelay(2);
}
```

7.) Comment on the Up/Down/”—” display that you see.

8.) Is there a ‘priority’ associated with the Mutex? If so, how can it be changed?

9.) Button_3 resets the mutex-protected global variable to “50.” It too, has to wait for the mutex to be granted. Change the priority of the Reset to be osPriorityIdle. This is the lowest priority available.



```
/* Definitions for ResetMutexCount */
osThreadId_t ResetMutexCountHandle;
const osThreadAttr_t ResetMutexCount_attributes = {
  .name = "ResetMutexCount",
  .stack_size = 128 * 4,
  .priority = (osPriority_t) osPriorityIdle,
};
/* Definitions for DebounceTask */
```

Did you see any effect on the ability of Button_3 to reset the count?

Part 3: Software Timers

In a previous lab, we learned how to setup any of the hardware timer blocks to generate periodic interrupts. This is limited by the number of actual timers available on the silicon, and still requires careful coding to insure RTOS-friendly servicing, as discussed with the buttons above. Software timers are easy to deploy, flexible to use, and have no concerns operating in the RTOS environment.

Software timers don't generate an interrupt, they simply execute a procedure when they expire. Button_2 is used to toggle the SW_Timer_7Seg between running/stopped. When the timer expires, it calls a routine to decrement the left-most display digit.

Part 3 Questions (2 Pts.)

1.) .Change the timer period from the current "200" mS to something different. Verify that the decrementing count changes accordingly.

YES – DID IT

____

2.) This timer was created via the GUI (.IOC file). It's type is "osTimerPeriodic" which means it repeats over and over.   What other options can a Software Timer take to change its Type and operation?

____

EXTRA CREDIT Ideas

Extra Credit Ideas (5 pt. for any)

1. The Seven-Seg Display is currently refreshed with a hardware (TIM17) timer. Make this more thread-safe by changing the refresh as a process that is based off a S/W timer.

Write about how you did it, and what the slowest period could be to keep the persistence looking good:

____

2. We only used a binary semaphore in this lab for the switch presses. Change it so that presses are accumulated through a counting semaphore and then handled as they are taken off. Describe any issues with this approach

____

3. The debounce for the switches here used a uwDelay() non-blocking call. Is there any advantage to using a SWTimer here instead? Explain why or why not?

____

4. Any other relevant uses for semaphores, mutexes, or S/W timers ? Describe what you've done and why?

____