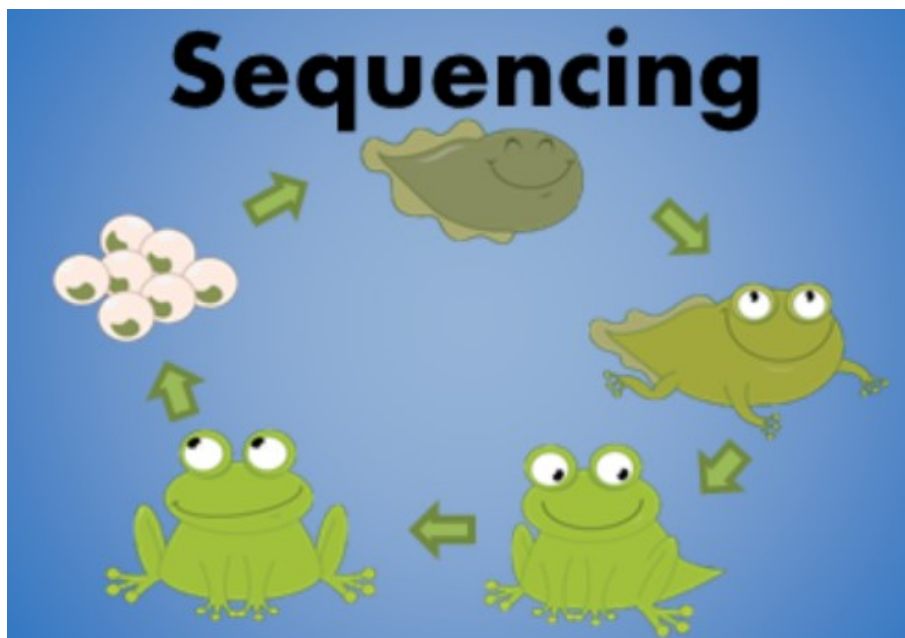


Lab 5 Unilateral Synchronization



Section 1 Lab Objectives

When this lab exercise is completed, the student should be able to use several important microprocessor and RTOS resources to:

1. Use interrupts to trigger an event, set a signal flag or semaphore. tasks.
2. Use signal (event) flags to synchronize several
3. Use semaphores as signal flags.

Section 2 PreLaboratory Preparation

Prior to your scheduled laboratory meeting time the following items need to be completed.

On Line Learning

1. Watch *Understanding Interrupts in 4 minutes !! — NVIC, EXTI IRQ* <https://www.youtube.com/watch?v=6b3AnvSzXXU>
2. Watch *Interrupts — #8 STM32 GPIO button interrupt* https://www.youtube.com/watch?v=qd_tevhJ2eE
3. Watch *Mutex vs Synchronization* <https://www.youtube.com/watch?v=jkRN9zcLH1s>
4. Watch *FreeRTOS on STM32 - 16 Signals* <https://www.youtube.com/watch?v=y084BHYwUBM>

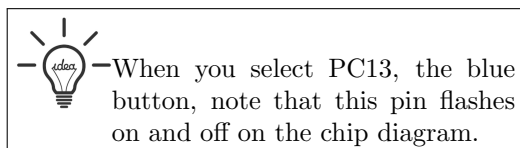
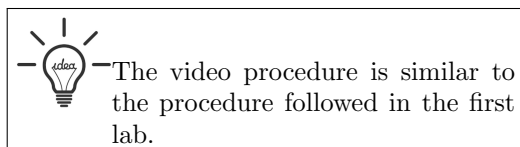
Preparation for the Prelab Quiz

The quiz will be available in lab for the first ten (10) minutes of your laboratory session. You may use any of your prelab preparation as a reference while taking the quiz.

Section 3 Interrupts

Follow the instructions in the pre-lab video *Interrupts — #8 STM32 GPIO button interrupt* with the following modifications.

1. Select our Nucleo-L476RG (not the Nucleo-L303RE).



2. The NVIC Configuration window has changed. Use the configuration in Figure 1.

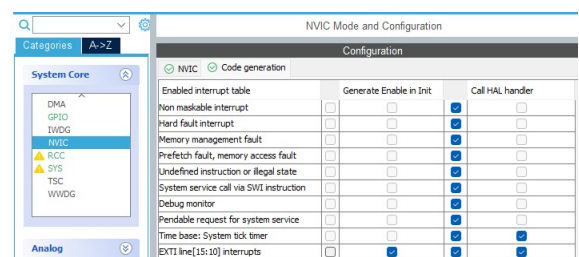
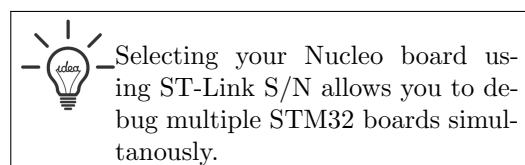
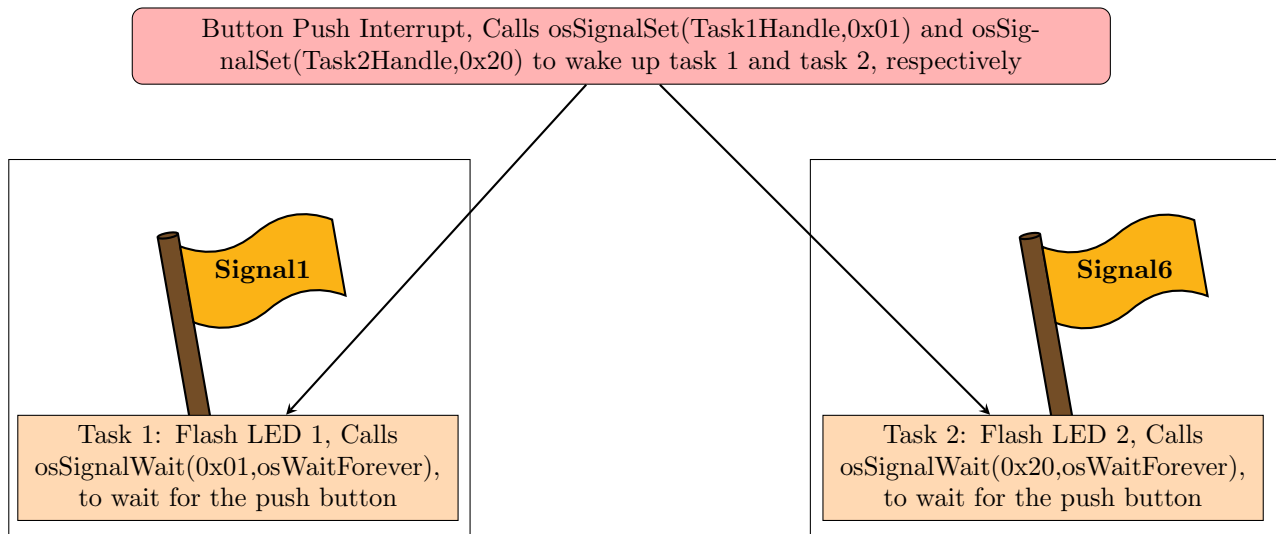



Figure 1: Updated NVIC Configuration Window


3. To turn on the breakpoint, right click on the right side of the line number and select Toggle Breakpoint.
4. Debug Configuration has moved further down the Run menu. STM32 Cortex-M C/C++ Applications has changed to STM32 C/C++ Application.





Section 4 Interrupts and Signal Flags




 Signal flags are part of the task. Each task has up to 32 signals (usually only 8 are enabled). In the diagram above, each task will call `osSignalWait` and wait for the signal to be set. Then the push button will set task 1's signal number 1 and task 2's signal number 6 using the `osSignalSet` command.

 Within a task, you call `osSignalWait(FlagNumber,WaitingTime)` to suspend the task. The task will then wait to be woken up by another task or interrupt that sets the signal flag. A task can only send a wait signal to one of its own flags, so there is no need to identify the calling task in the `osSignalWait` command.

 To set a Signal Flag and release the suspended task, a different task or interrupt calls `osSignalSet(DestinationTaskName, DestinationFlagNumber)`. The sending task or interrupt must specify the task it is sending to and the specific flag in that task.

 The flag numbers are provided in hex. So 0x20 hex sets the sixth bit. This can be seen by converting 0x20 to binary 0010 0000 where position 6 is set.

1. Turn on FreeRTOS and add two tasks to the project from the previous section.
2. Turn on two external (breadboard) LEDs, one for each task.

 Don't forget to make the changes to SYS in the System Core and the Advanced Settings in FreeRTOS.

The pre-lab video *Interrupts — #8 STM32 GPIO button interrupt* sets up the ioc to "Generate peripheral initialization as a pair of '.c/.h' files per peripheral. This setting is in the ioc's Project Manager tab under the Code Generator tab. This setting will tell the ioc to place the tasks in `freertos.c` instead of `main.c`. In my opinion this is a much better configuration because it prevents everything from going into `main.c`. When we want to work on the task, we go to `freertos.c`, when we want to work on the inputs and outputs, we go to `gpio.c`.

3. In the interrupt call back routine (in gpio.c) set signal flag 1 for task 1 the first time the button is pressed and set signal flag six in task 2 the second time the button is pressed.



The task handle names are defined in freertos.c and needed in the gpio.c interrupt callback function to set the Signal Flags. Allowing different parts of your program to share data while maintaining encapsulation is fundamental in modular programming in C. An include file can be used to share the handle names.

4. Create a new header file (File, New, Header File) and name is something like

sharedtasknames.h. Add `#include sharedtasknames.h` to both freertos.c and gpio.c (the two files sharing variables). The code for sharedtasknames.h is given below.

```
1 #ifndef SRC_SHAREDTASKNAMES_H
2 #define SRC_SHAREDTASKNAMES_H
3 // location of osThreadId definition
4 #include "cmsis_os.h"
5 // tells compiler these variables are
6 // defined somewhere else
7 extern osThreadId Task01Handle;
8 extern osThreadId Task02Handle;
9 extern osThreadId Task03Handle;
10 #endif /* SRC_SHAREDTASKNAMES_H */
```

5. Connect Task 1 and Task 2 to external LEDs.
6. Suspend both tasks using `osSignalWait`.
7. When a task flag is set, toggle its LED and loop back to `osSignalWait`.

Section 5 Three Tasks and an Interrupt

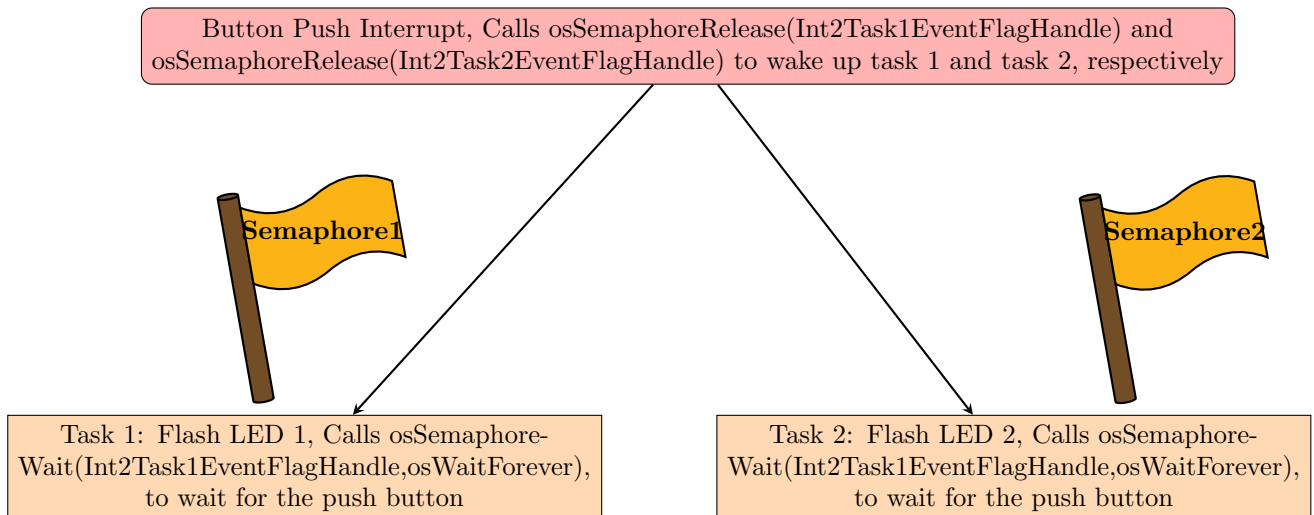
1. Create a project with an interrupt routine for the push-button like the last section.
2. In the ioc, turn on FreeRTOS and create three tasks.
3. Turn on three of our external (breadboard) LEDs, one for each task.
4. When the button is pressed the first time, the first task should flash its LED 10 times. Turn it on for 250ms and then off for 500ms.
5. Set up the second task to flash its LED 10 times. Turn it off for 250ms, on for 250ms and then off again for 250ms.
6. When the button is pressed a second time, the first two tasks should flash their LEDs.
7. When the button is pressed a third time, do something with the third task. Feel free to include the other two tasks if you want to.



You will need to pick a signal flag for task 3.

8. Explain and show your program for a sign-off.

Section 6 Replace the Signal Flags with Semaphores



1. Create a project from the last section's ioc with three tasks and gpio for the LEDs. One of the tasks and LEDs will be unused.
2. In the ioc, create two binary semaphores.



A suggested semaphore name would be `Int2Task1EventFlag` to indicate to future engineers reading your code that you're using the semaphore for unilateral synchronization and not mutual exclusion.



Signal Flags have one drawback: they are a feature specific to FreeRTOS. Semaphores are a standard RTOS component. Two important differences between semaphores and signal flags are: 1) semaphores do not belong to a task, so the handle must be used when both setting and releasing them, and 2) semaphores are created in the unlocked state. Starting in the unlocked state means the first call to `osSemaphoreWait` will find the flag available and continue execution (no wait at all). Adding a call to `osSemaphoreWait` before the forever loop in each task will lock the semaphore.

3. Configure the project to blink the LEDs following a similar pattern as in the previous section. You will need to:
 - Only use two tasks and two LEDs. Leaving the unused task to run `osDelay(1)` will not hurt the project.
 - Create a `.h` file to share the semaphore handles between `freertos.c` and `gpio.c`.
 - Program the tasks in `freertos.c` replacing the signal flags with semaphores.
 - Create a interrupt call-back function in `gpio.c` for the push button that unlocks the semaphores.
4. Show your new project that uses semaphores for a sign-off.

Section 7 Sign-offs

Name: _____

Section 3: Blink when button pressed.

Date

_____.

Section 5: Push button controlling a task.

Date

_____.

Section 6: Second project with Semaphore replacing Event Flags.

Date

_____.

Section 8 Report

By the start of next lab, upload to MyCourses Dropbox in a single document containing.

1. A scan or picture of this sign-off sheet.
2. A brief explanation of signal and semaphore flags.

Reports submitted **one week late** will receive a maximum grade of 70%. Reports submitted more than three weeks late will receive a maximum grade of 40%.