Lab summary:

In this lab, we explored inter-thread communication and synchronization using semaphores, event flags, and message queues in order to create a task driven version of our lab 3 code. In order to properly use these tools, we had to consider the location of and number of post/pend points for the semaphore; how and when the event flag should be used in order to minimize time spent in interrupt handlers; and how message queues make it easy for threads to communicate.

What I learned from this lab:

This lab gave me my first hands-on opportunity with semaphores and message queues. I have experience with mutual exclusion via mutex locks (in C++) and found this to be a really interesting way to achieve the same level of mutual exclusion. I learned that the OS uses many different osXxxId_t objects, each with a pretty similar interface, to create event flags, semaphores, and message queues. I think the big thing here was that rather than having interrupt driven code, or sampling at a regular interval, we could leverage a task driven design. This allows the OS to do a lot of the heavy lifting, while we worry more about how these tasks can communicate, synchronize-with, and ultimately co-exist in an effective and safe (from a data-race perspective) way. This also opens the door for us to begin thinking about task priority and how that idea can fit into a task driven system to provide more important tasks the space to do what they need to within well-defined time limits. This, I think, is the ultimate goal of real-time systems.

Functional Tests:

Note: All functional tests assume the same frame of reference: Hold the board with the LCD closest to the chest, parallel to the ground. The axis of rotation is the wrist.

1. Holding the board stationary, none of the LEDs light up. **PASSED**
2. Holding the board stationary and pressing the blue button, both LEDs light up. **PASSED**
3. Holding the board stationary, then rotating counterclockwise, the green LED will light up. **PASSED**
4. Holding the board stationary, then rotating counterclockwise and pressing the blue button, only the green LED will light up. **PASSED**
5. Holding the board stationary, then rotating clockwise, none of the LEDs light up. **PASSED**
6. Holding the board stationary, then rotating clockwise and pressing the blue button, both LED light up. **PASSED**

Question Responses:

Q: Record how often the GyroInput task is run. Does it match the timer period? Save a screenshot that shows the period.

As can be seen in figure 1 below, the *gyroInputTask* does indeed match the timer's 100ms period. Figure 1 also shows that the *ledOutputTask* is synchronized with the *gyroInputTask*, which is what we would expect, since the *gyroInputTask* uses the message queue in order to synchronize (or rather, inter-thread communicate) with the *ledOutputTask*.

The *ledOutputTask* has a call to *osMessageQueueGet()*, which will put the task into a **blocked** state while it awaits a new message. Because we use the *osWaitForever* timeout value, the *ledOutputTask* will remain blocked – indefinitely – until a message is received. Since the *gyroInputTask* uses a semaphore to synchronize with the application timer, we expect to sample the gyro and send that new gyro value over, via the message queue, every 100ms. Once the message is received, the OS will reschedule the *ledOutputTask* (blocked → running).

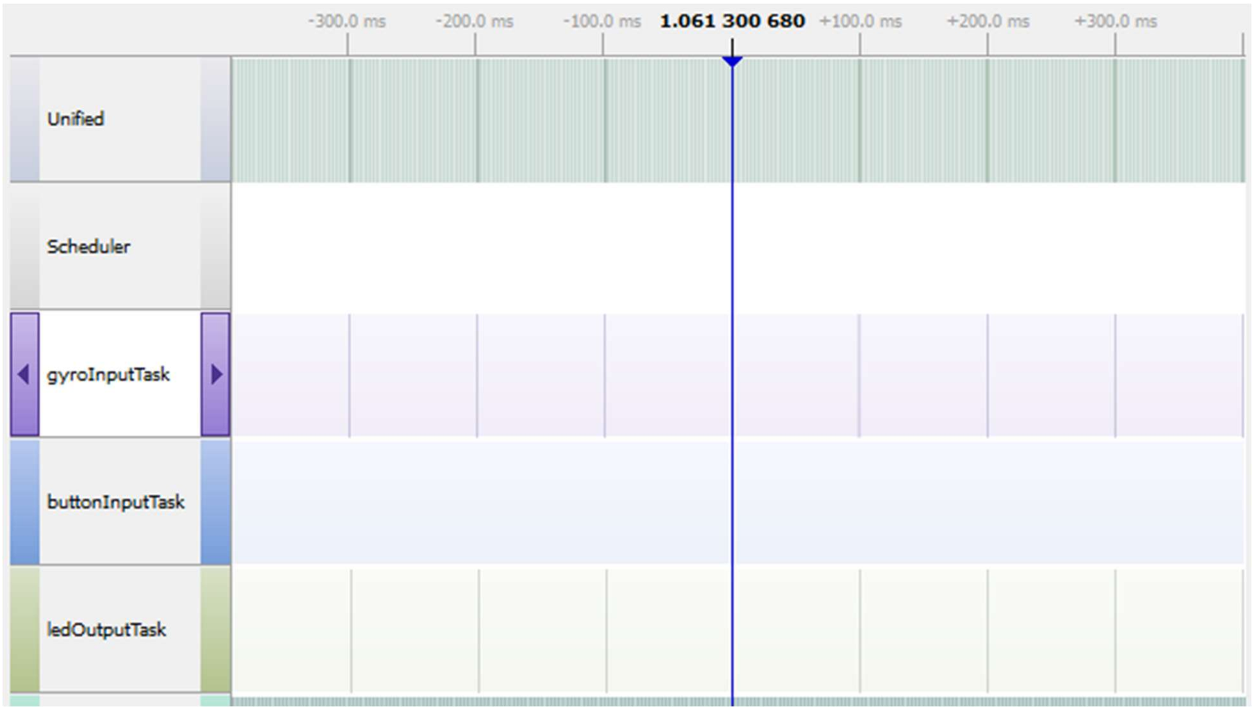

Figure 1: SystemView Recording timeline showing the synchronization between the gyroInputTask and the ledOutputTask every 100ms

<u>**Q: Does the scheduling of the LEDOutput task appear to be synchronized with the ButtonInput task? Explain why.**</u>

Similar to the *gyroInputTask*, the *buttonInputTask* is also synchronized (or rather inter-thread communicates) with the *ledOutputTask*. It achieves this through the message queue, in the same way, but it also uses the button's GPIO interrupt handler to set an event flag using the *osEventFlagsSet()* function. The *buttonInputTask* will synchronize with that event flag and the OS will put the task into a blocked state while it waits (also indefinitely; *osWaitForever*). Once the event flag is set, the *buttonInputTask* will then sample and send the button signal message over the message queue. Once that message is received (because, again, the *ledOutputTask* will wait for a new message), the *ledOutputTask* will process that message. Looking at figure 2, below, we can see this exemplified.

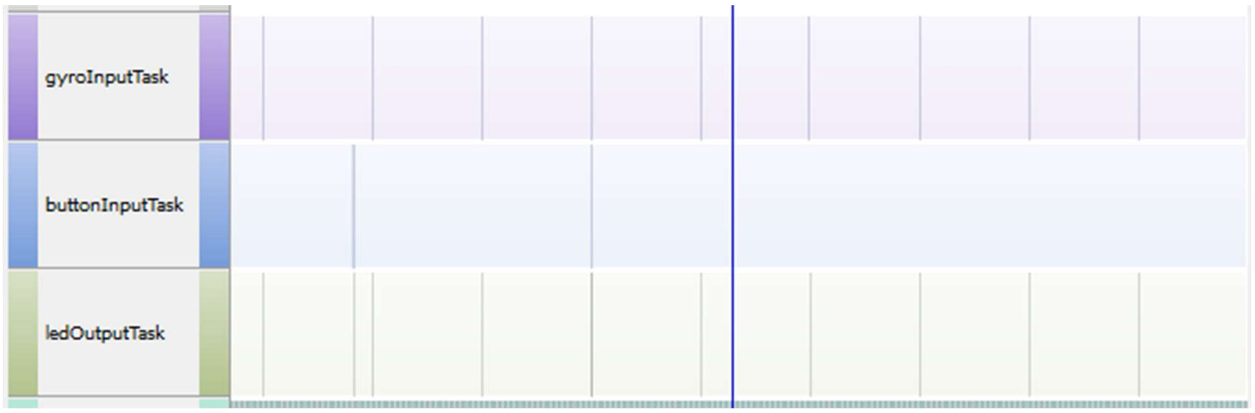

Figure 2: SystemView Recording timeline showing the synchronization between the buttonInputTask and the ledOutputTask