## Objective

This project further explored FreeRTOS. An idle task hook function, periodic task, handler task and change notice interrupt are used to enable and disable LEDs A through D according to the functionality described below.

## Functionality

* LED A is enabled while button 1 is pressed. This is implemented with the idle task hook function.
* LED B is toggled every millisecond. This is done in a periodic task, which is blocked during delay.
* LED C is toggled in a push on/ push off manner. A CN ISR is serviced when button 1 is pressed, within this a semaphore is sent to unblock a handler task which implements a 20 millisecond blocking debounce period.
* LED D is enabled while the C portion of the CN ISR is active.

## Implementation

### Task Priorities

Priority of the handler task and periodic task was determined with the intent to reduce the time between a button press and toggle of LED A. To reduce latency, the handler task was chosen as the highest priority task. In the event of a button press, this allows the kernel to run the ISR, unblock the handler task, and immediately service the handler task. This resulted in button debounce delay beginning as soon as possible, with minimal effect to toggling LED B.

### Periodic Task

The toggle of LED B occurs every millisecond. The implementation for this task followed the exact methodology from project 1.

### Idle Hook Function

When all priority driven tasks are blocking, the Idle Hook Function is run. Within this, button 1 is read and compared to a global variable storing the previous button value. If it has changed, LED A is toggled. This implemented the functionality described above.

### Debouncing Button 1

All debouncing was implemented within the handler task using API function vTaskDelay(). This was a straightforward method to accomplish a 20 millisecond delay that utilized FreeRTOS functionality.

The first debounce period is entered after a semaphore is received from the CN ISR handler. The logic level of the button is allowed to rest, and if it is still logic high, then LED C is inverted. Following this delay, a loop waits for the button level to return to logic low.

Following this, a second 20 millisecond debounce period is entered. This removes the possibility of unwanted change notices occurring upon release of the button.

### Push On/Push Off Button 1 Behavior

LED C control is best described by figure 1 on the following page. Two tools are utilized to minimize latency and keep a task blocking when it is not necessary to be serviced: Semaphores and delays.

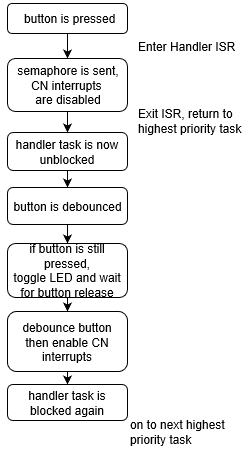


Figure 1: Control Flow Diagram for LED C

The benefit of this method is found when no activity is present on button 1. The block on the handler task waits for a semaphore. If this hasn’t been sent, the processor hums away working on other unblocked tasks of the idle hook function.

## Challenges Faced

The only major problem I encountered came during the implementation of push on/push off behavior. Prior to disabling interrupts during the handler task the kernel would place me back in the ISR on timer ticks. I was unable to resolve the issue without disabling CN interrupts during the duration of the debounce periods. This resolved my issues and the rest of the functionality implementation was very straightforward.

## LED behavior

The following captures show the LED response for different button activity.

Button Press

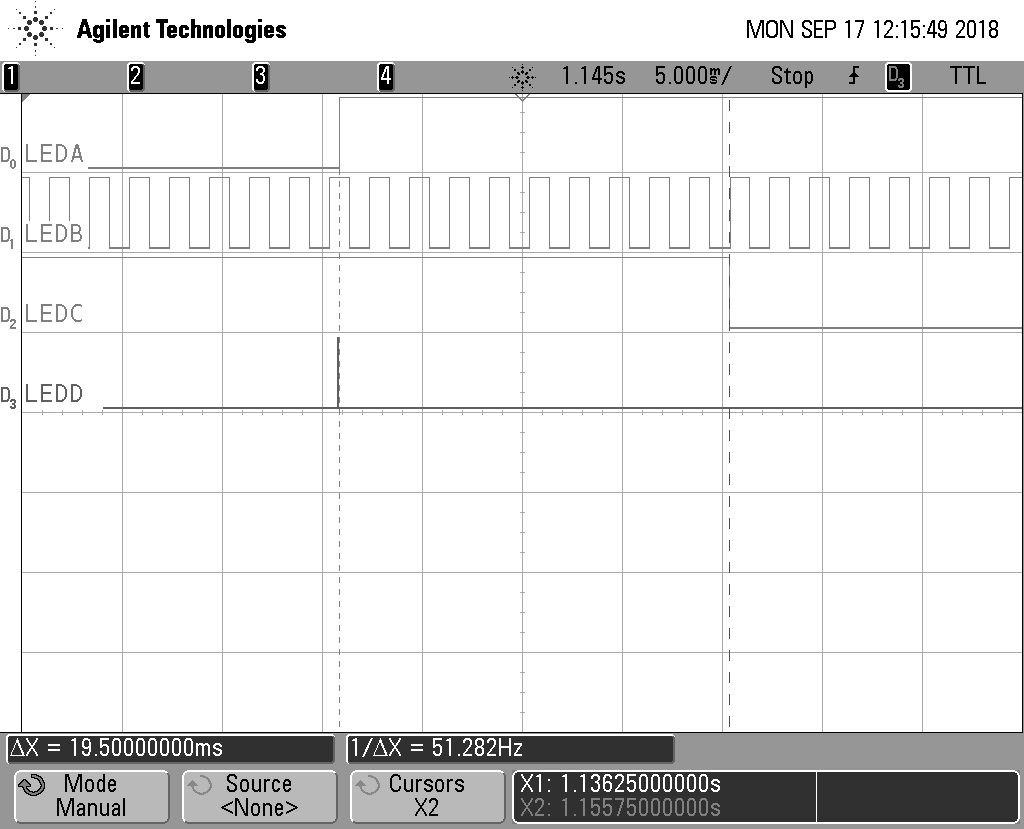


Figure 2: Button Press

LED D is active during ISR servicing, which sends the semaphore. The context is then returned to the handler task and a delay period blocks the function. The idle hook task then gets a turn, which enables LED A. After 20 timer ticks, the handler task unblocks again and LED C is finally toggled.

### Button Release

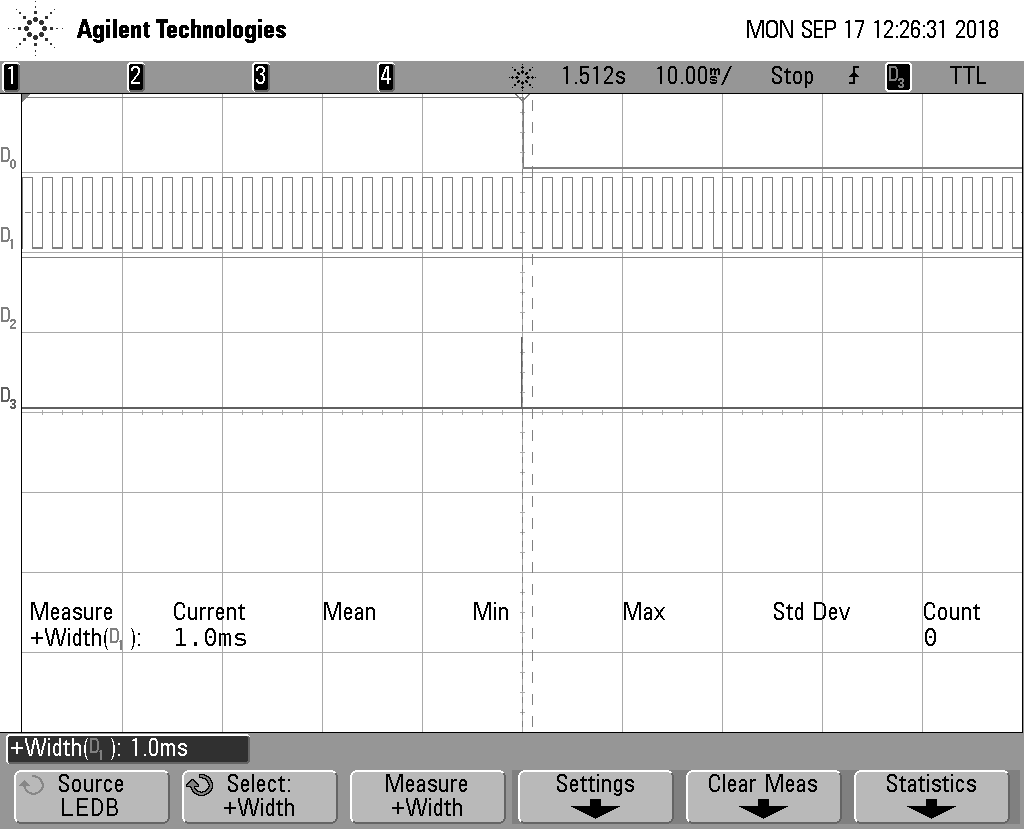


Figure 3: Button Release

This verifies the toggle functionality. Upon release, the CN ISR is serviced, yet LED C maintains it’s last state. Additionally, the measurement of D1 shows the 1 millisecond periodic task that toggles LED B.

## Tracelyzer

### CPU Load

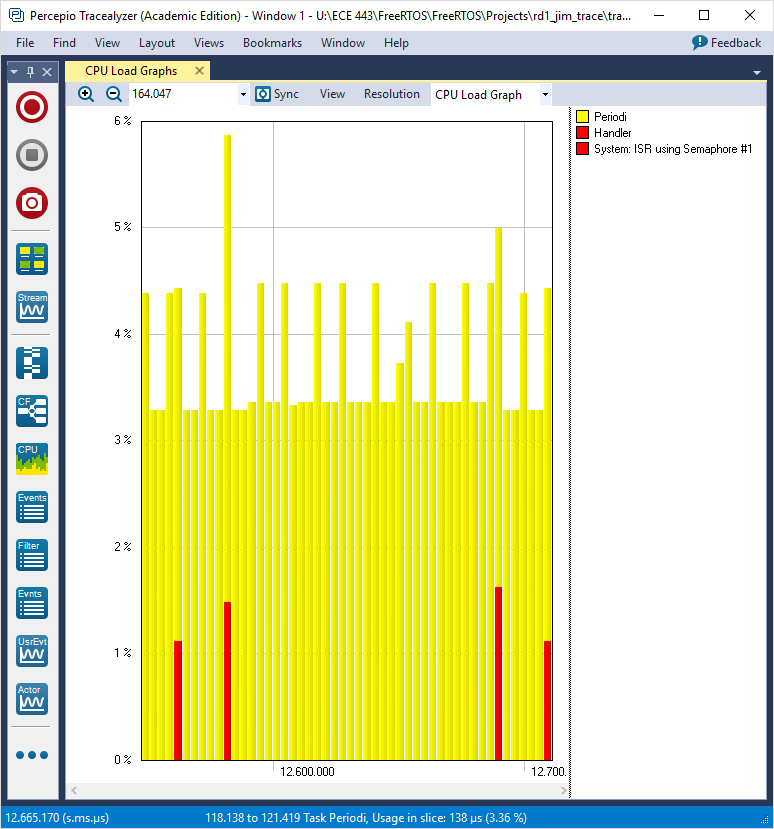


Figure 4: CPU Load

Here we see that the handler task was ran a total of 4 times before halting (twice for button press an twice for button release).

### Trace View

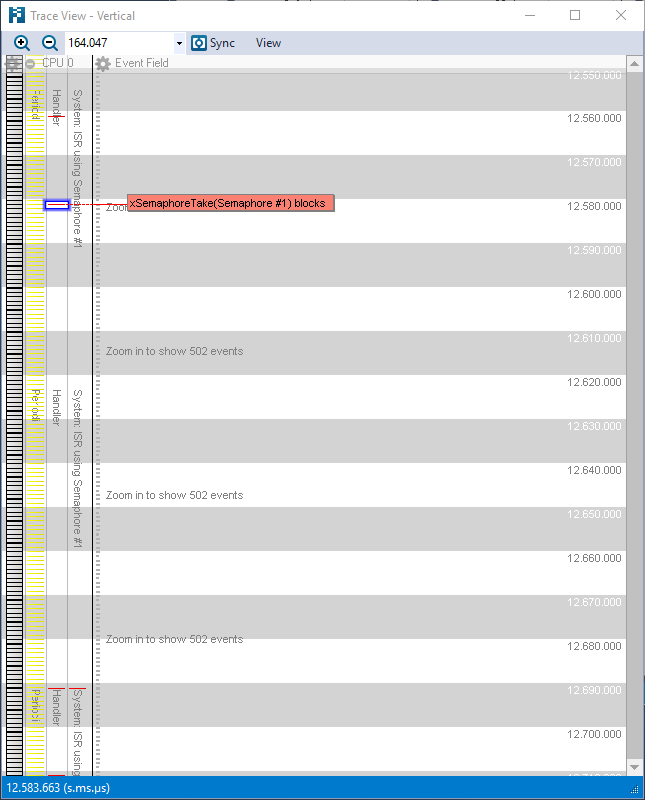


Figure 5: Trace View

This view shows a detailed breakdown of timing and when each task was ran.

## Conclusion

I completed all functionality specified in the requirements. My code is understandable and well commented. And, my report is thorough and encompasses all topics specified in the project handout, with additional comments on any additional functionality. I sincerely hope that this meets and exceeds your expectations.