ECE 443 - Project #2

Collin Heist

September 20, 2019

Contents

1 Design Process **1**

2 Testing & Verification **2**

Listings

1 Idle Task 1
2 Tick Hook 'Task' 1

1 Design Process

In the previous project I chose to implement a twenty millisecond debounce period using a very rudimentary hardware-assisted delay function, called from inside a change-notice ISR. This worked very well for the first project because there was no need to worry about task execution while the buttons were debouncing, however I had to change my implementation for this project.

I chose to instead call **vTaskDelay()** inside the change notice handler task. This allows for other tasks to execute, although there are not any other tasks in this program. However, this is still necessary because while the change notice handler task is *blocked* during the duration of that **vTaskDelay()**, the idle task is performed (as it is the only other 'task'). The idle task's code is shown below, in **Listing 1**

Listing 1: Idle Task

```
void vApplicationIdleHook(void) {
  if (PORTG & BTN1)
    mPORTBSetBits(LEDA);
  else
    mPORTBClearBits(LEDA);
}
```

This idle task ensures that **LEDA** is always set to the current status of **BTN1**, and this remains true while the button is being debounced (something that was not occurring in my previous project).

Because the only task that I needed to manually create was my change notice handler task, my choice of task priority was largely arbitrary. But I did take care to set my task priority to a level below that at which interrupts can interrupt (which is by default 3), to ensure the change notice interrupt always took precedence. I chose a task level of 1, which was the lowest task non-idle task priority available.

In order to save the trouble of manually creating a one millisecond toggle **LEDB** task, which would have created more overhead for task-switching, as well as necessitated more careful planning on my part for task priorities (as the one millisecond toggle would need to occur not matter what), I instead appended my **LEDB** toggle code onto the existing **vApplicationTickHook()** functionality.

The function **vApplicationTickHook()** is automatically called every *tick* of the kernel, which is by default one millisecond. This means that every time the kernel is about to perform a task-switching operation (or even just check if a task should be switched into), my custom tick hook function was called – giving me the desired one millisecond toggle of **LEDB**. That code is shown in **Listing 2**.

Listing 2: Tick Hook 'Task'

```
void vApplicationTickHook(void) {
  if (configUSE_TRACE_FACILITY)
    vTracePrint(trace_1ms_task, "Toggling LEDB");

  LATBINV = LEDB;
```

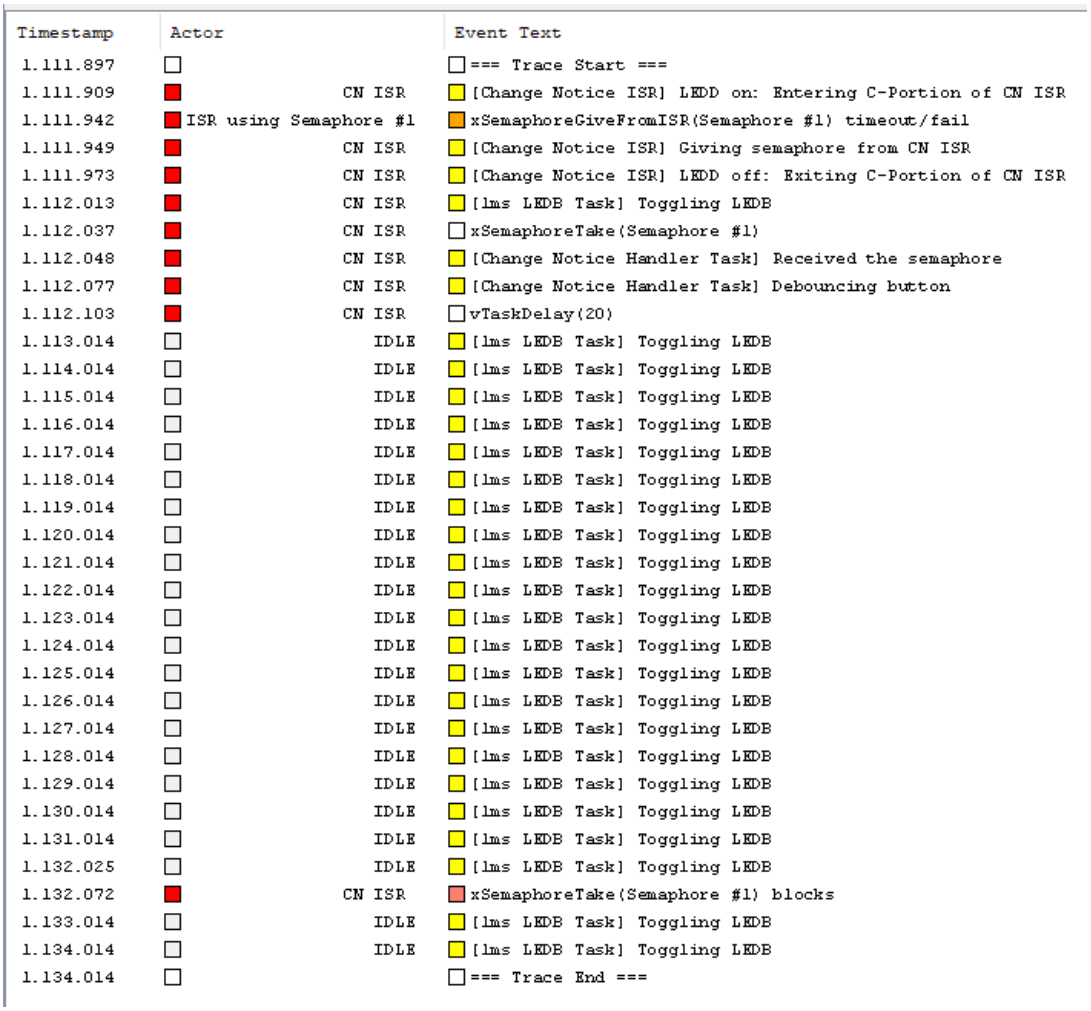
```
    if (configUSE_TRACE_FACILITY && DEBUG_MODE) {
      if (LATB & LEDC)
        trace_ledb_count++;

      if (trace_ledb_count >= 2)
        LATB = LATB;
    }
  }
```

The last few lines are what I used in order to obtain my TraceAlyzer screenshot, but overall adding to this kernel tick function was simple.

Overall, I did not have any problems with this project. The hardest part of this project was setting up TraceAlyzer, as I had to modify parts of the **FreeRTOS** configuration file, and had a lot of problems having the necessary files in my project directory.

2 Testing & Verification

In order to verify the operation of my code, I used the TraceAlyzer plugin for **FreeRTOS**. The project outline had me finish a capture once **LEDB** had been toggled twice while **LEDD** was lit. That resulted in the event log showed in **Figure 1**.



Timestamp	Actor	Event Text
1.111.897		=== Trace Start ===
1.111.909	CN ISR	[Change Notice ISR] LEDD on: Entering C-Portion of CN ISR
1.111.942	ISR using Semaphore #1	xSemaphoreGiveFromISR(Semaphore #1) timeout/fail
1.111.949	CN ISR	[Change Notice ISR] Giving semaphore from CN ISR
1.111.973	CN ISR	[Change Notice ISR] LEDD off: Exiting C-Portion of CN ISR
1.112.013	CN ISR	[1ms LEDB Task] Toggling LEDB
1.112.037	CN ISR	xSemaphoreTake(Semaphore #1)
1.112.048	CN ISR	[Change Notice Handler Task] Received the semaphore
1.112.077	CN ISR	[Change Notice Handler Task] Debouncing button
1.112.103	CN ISR	vTaskDelay(20)
1.113.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.114.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.115.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.116.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.117.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.118.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.119.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.120.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.121.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.122.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.123.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.124.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.125.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.126.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.127.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.128.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.129.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.130.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.131.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.132.025	IDLE	[1ms LEDB Task] Toggling LEDB
1.132.072	CN ISR	xSemaphoreTake(Semaphore #1) blocks
1.133.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.134.014	IDLE	[1ms LEDB Task] Toggling LEDB
1.134.014		=== Trace End ===

Figure 1: TraceAlyzer event log of two **LEDB** toggles after a button press

As evident in the picture, once the button is pressed, the sequence of events occurred as follows:

1. The C-portion of the ISR is executed
2. The timing worked out such that a previous call of **xSemaphoreTake()** expired – this is not a determined event
3. Then, the semaphore was given from the change notice ISR
4. The semaphore was received – unblocking the change notice handler task
5. The 20ms debounce period began, blocking the handler task
6. 20 milliseconds passed
7. The **vTaskDelay()** unblocks and quickly blocks again – waiting for the semaphore once again

This behavior is exactly what is desired based on the project specifications. Which is good!

I also took two captures of the LED pins during a button press and release, also for verification. The first capture shows what happens when a button is first pressed.

Figure 2: Status of all LED's when a button is *pressed*

The final figure shows the status of each LED when the button is released.

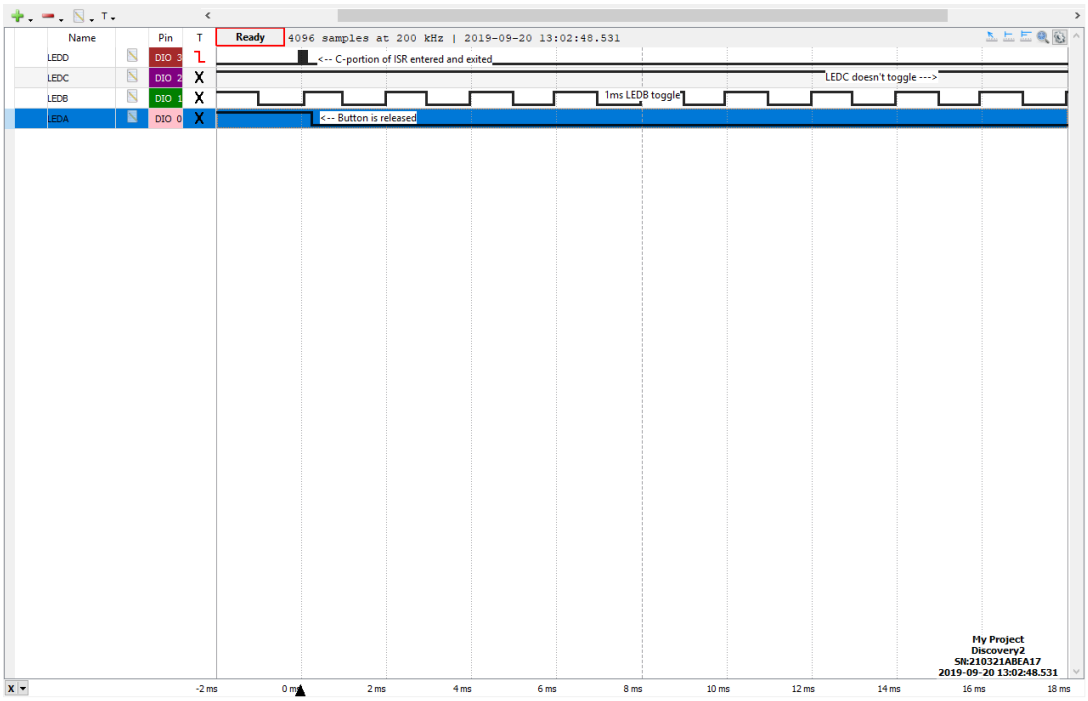


Figure 3: Status of all LED's when a button is *release*