ECE 315 Lab 1 Report

Miro Straszynski
Ebuka Odeluga

ECE 315 — Section H21
February 24, 2026

Abstract

The purpose of this lab was to gain practical experience designing an application using FreeRTOS, incorporating multiple tasks, delays, and queues, while also interfacing with external hardware components. The lab was divided into three parts, each focusing on different system interactions and architectural concepts. The hardware platform used was the Digilent Zybo Z7 development board, based on the AMD Zynq-7010 System-on-Chip (SoC). Two of the five Pmod ports were connected to a 2-digit 7-segment LED display for output, while a third Pmod port interfaced with a 16-button keypad for input.

In the first part of the lab, we implemented communication between the keypad and the 7-segment display. The objective was to display each newly pressed key on the right digit of the display, while shifting the previously pressed key (if any) to the left digit. The second part involved interfacing with the board's built-in RGB LED and push buttons. The goal was to adjust the LED's brightness using Pulse Width Modulation (PWM) whenever a valid push button was pressed. The third part required integrating queues within FreeRTOS to achieve the functionality developed in the first two parts. This emphasized inter-task communication and reinforced structured application design principles. Overall, the lab provided hands-on experience in designing embedded applications using FreeRTOS, focusing on task management, hardware interfacing, and modular system architecture.

Design

Part 1

This code implements Part 1 of the lab, which interfaces a keypad with a two-digit seven-segment display (SSD). The goal of this part is to continuously receive keypad input and display the current and previous key presses without visible flicker.

The keypad is initialized using `InitializeKeyPad()`, which calls `KYPD_begin()` to configure the GPIO as input and loads the default key table for hexadecimal mapping. The SSD is initialized using `SSD_begin(&SSDInst, SSD_BASEADDR)` and configures the SSD GPIO as an output device. A keypad task (`vKeypadTask`) is then created using `xTaskCreate()`, and the scheduler is started with `vTaskStartScheduler()`. FreeRTOS controls execution from this point onward. The keypad task runs in an infinite loop, reading the keypad state and getting the status. The status is the number of keys pressed at a time (one key returns `KYPD_SINGLE_KEY`, multiple keys returns `KYPD_MULTI_KEY` and no keys returns `KYPD_NO_KEY`). When a new single key is pressed (status is `KYPD_SINGLE_KEY`), the previous key variable is set to the current key, and the current key variable is set to the new key. Whenever the status is changed, it prints to the terminal to monitor status transitions. When a single key is pressed, the key value is converted into the 7-segment binary patterns using `SSD_decode()`. The newest key is displayed, then routed to display on the right side of the SSD, and the previous key is routed to display on the left side of the SSD. Since the SSD can only display one digit at a time, the left and right digits are alternated rapidly to make both digits appear continuously lit. We used a delay of 10 ticks to control the refresh speed (`xDelay = 10`), and this gave the illusion of constant illumination for both digits.

The system diagram for part 1 is shown in Figure 1. A task flow diagram for the keypad task is shown in Figure 2.

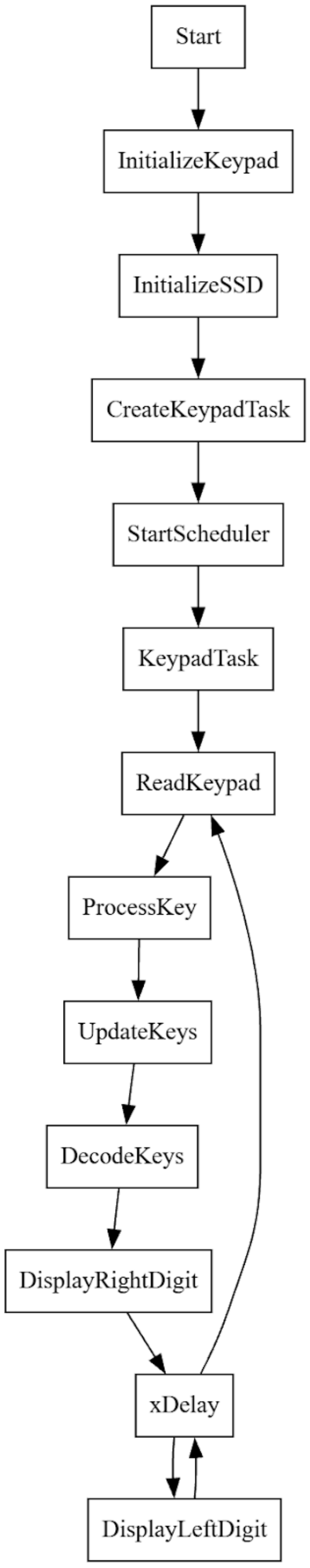

Figure 1: System Task Flow Diagram for Part 1.

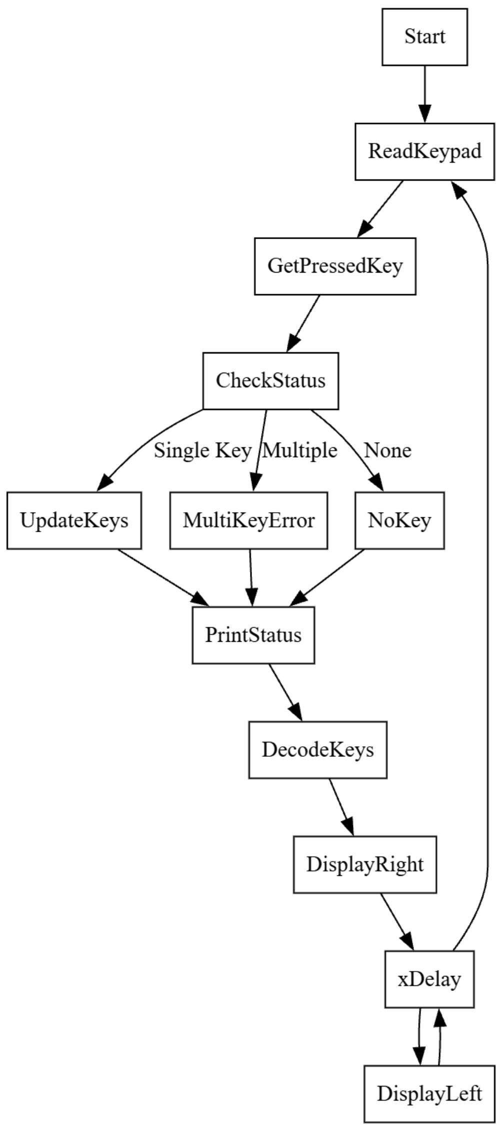

Figure 2: Task Flow Diagram for the Keypad Task.

Part 2

Here, we build on the first part of the lab. In this part, a second task that interfaces RGB LEDs with pushbuttons already on the Zybo Z7 board will run concurrently, of the same priority, with the first task in part 1 under the FreeRTOS scheduler. The goal of this task is to control the brightness of the RGB LED using the pushbuttons and Pulse Width Modulation (PWM).

In addition to the code in part 1, the RGB KED was initialized using `RGB_LED_begin(&RGB_LEDInst, RGB_LED_BASEADDR)`, and the GPIO was configured to be an output. The pushbuttons were initialized with `PUSHBUTTON_begin(&pushButtonInst, PUSHBUTTON_BASEADDR)`, and the GPIO was configured as an input. The second task was created (using `xTaskCreate()`), and the scheduler is started, and execution is fully managed by FreeRTOS. The first task runs as outlined in part 1. In the second task, to achieve PWM, an on delay and off delay is used (the variables xOnDelay and xOffDelay, respectively), which determine the LED ON and OFF duration. The PWM period, which is the sum of the on and off delays, was determined by finding the smallest period in which no flickering was visible for the RGB LED when only one xDelay was used (which turned out to be 24 ticks). The on delay and off delay were then initialized to both be 12 ticks (even split).

The second task operates in an infinite loop that reads pushbutton input and changes the on and off delay based on the button pressed. If Button 0x01 is pressed, xOnDelay is decreased by 1, and xOffDelay is increased by 1, which decreases the ON time and gives the illusion of a dimmer LED. The opposite happens when Button 0x08 is pressed, increasing xOnDelay by 1 and decreasing xOffDelay by 1, increasing the ON time and giving the illusion of a brighter LED. Whenever a valid pushbutton is pressed, the values of the delays are also printed.

The system diagram for part 2 is shown in Figure 3. A task flow diagram for the RGB task is shown in Figure 4.

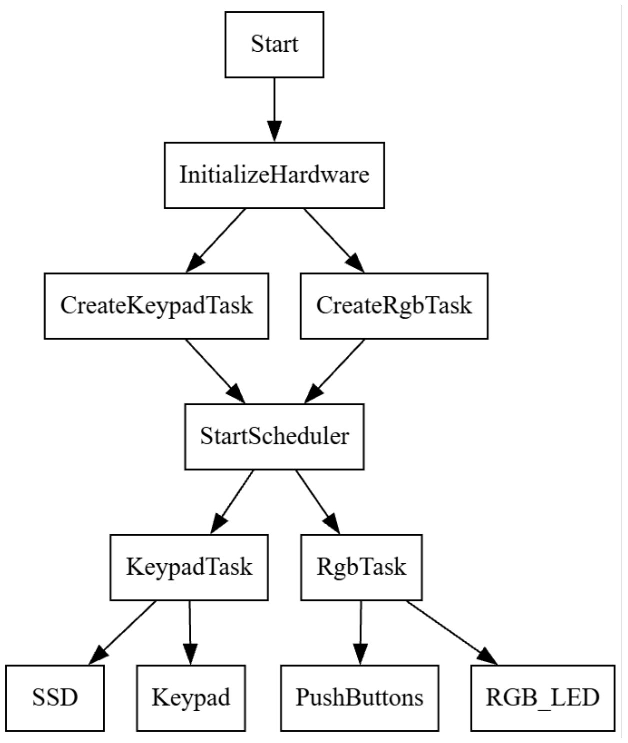

Figure 3: System Task Flow Diagram for Part 2.

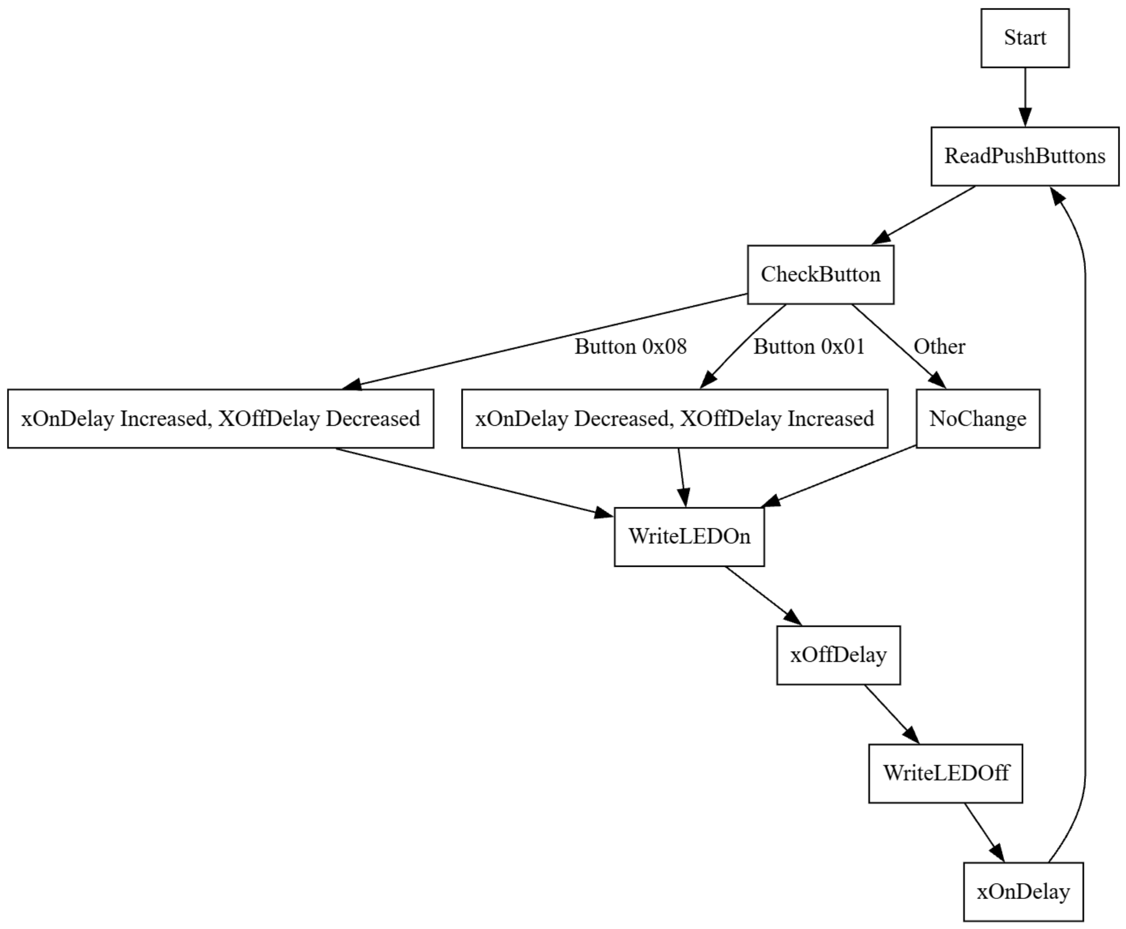

Figure 4: Task Flow Diagram for the RGB Task.

Part 3

This part expands on the first two parts by utilizing queues to communicate data between tasks. The keypad task from part 1 was split into two tasks: vKeypadTask, which handles the keypad reading and decoding, and vDisplayTask, which handles displaying the digits to the SSD. These two subtasks communicate with each other through the queue queueDisplay, which is the size of one element and was initialized in main() using xQueueCreate(). The element that is transported in this queue is a struct called display_data_t, which holds the current key and previous key. The vKeypadTask reads the newest key input, makes the previous_key variable the current key, and the current_key variable the newest key. It then stores these values in display_data_t and writes to the queue using xQueueOverwrite(). The vDisplayTask runs an infinite loop, continuously checking for data to read from the display queue. If display data is received from the queue (xQueueRecieve() returns pdTRUE), then display data is decoded and displayed on the SSD (with the current key being displayed on the right and the previous key being displayed on the left). The display task still uses the same xDelay when alternating rapidly between illuminating both LEDs to give the illusion of constant illumination for both the right and left displays.

The RGB task from part 2 was also split into two tasks: vRGBTask and vButtonTask. These two subtasks communicate similarly with each other through the queue queueButtons, which is also the size of one element. The element transported in this queue is xOnDelay, which is the ON time for the LED. The vButtonsTask reads which button is pressed (either 0x08 or 0x01) and increases/decreases xOnDelay and writes the value into the queue. The vRGBTask also runs an infinite loop using the PWM logic with its current values for the on and off delay, while continuously checking for data to read from the buttons queue. If data is received, it copies the value of the new xOnDelay from the queue and computes the required xOffDelay from it (PWM period - XOnDelay). It then uses the PWM logic with the new values, effectively lowering or increasing the brightness of the RGB LED.

The system architecture diagram for part 3 is shown in Figure 5.

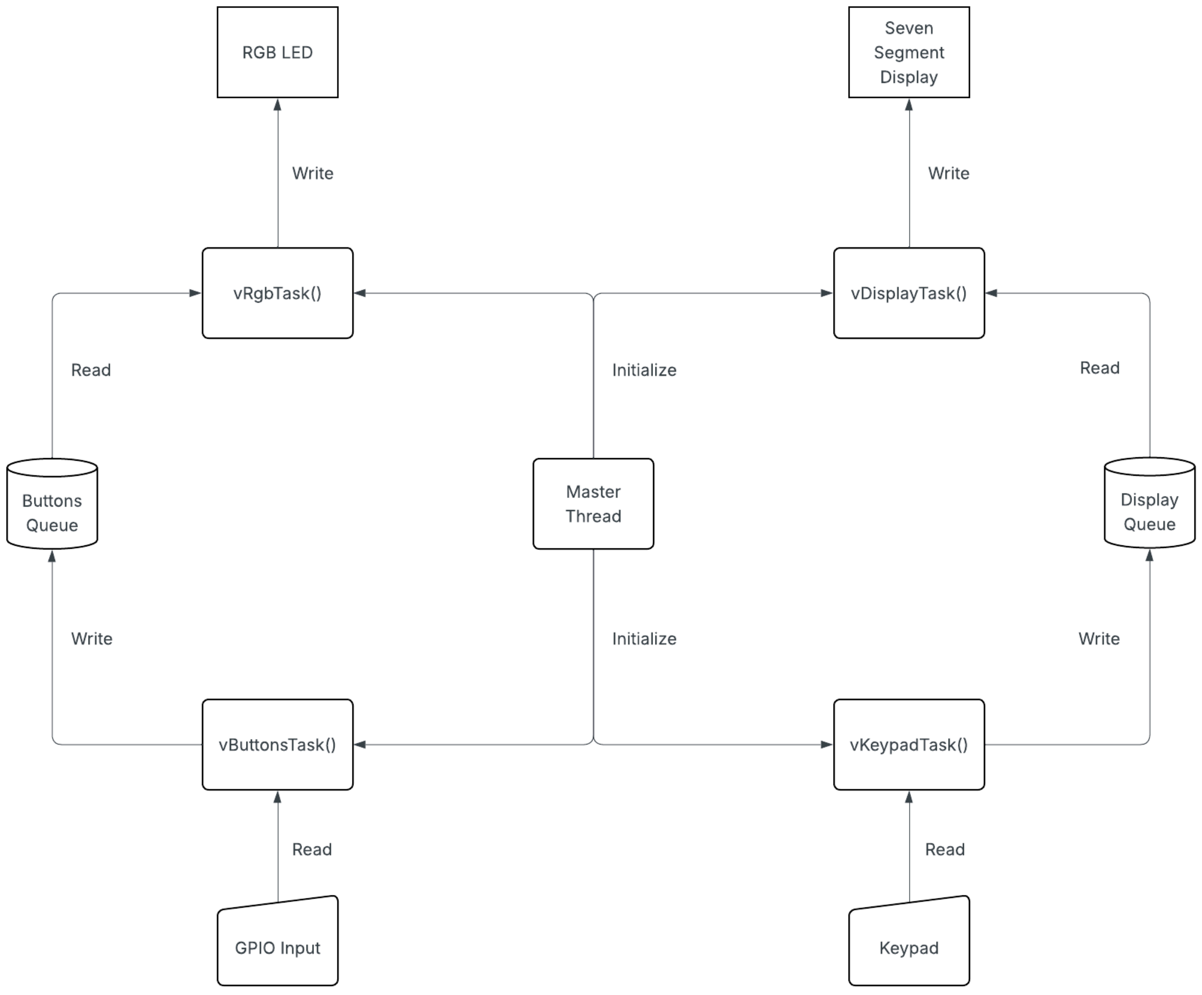

Figure 5: System Architecture Diagram for Part 3.

Testing

Part 1

Testing in part 1 occurred in two parts, first, we tested different xDelay values until we found the maximum delay where no visible flickering occurred between the two digits on the display. An example table of test cases is in Table 1.

Test Case #	**xDelay Value**	**Observation**
1	30	Flickering is visible to the human eye
2	5	Flickering is no longer visible to the human eye
3	10	Flickering isn't visible, and this is the maximum delay value before flickering becomes visible again

Table 1: Test cases for xDelay values in Part 1.

Additionally, we tested the operation of the keypad and verified that the correct digits were being displayed, and that they appeared on the correct side of the seven-segment display. Example test cases are in Table 2. We also tested that each of the digits and letters that can be entered on the keypad are displayed correctly, however this is excluded for the sake of brevity.

Test Case	**Operation**	**Expected Result**
1	Press '8'.	'8' is shown on the right side of the seven-segment display.
2	Press '3', after '8' has been pressed.	'3' is shown on the right side of the seven-segment display, while '8' is pushed to the left side.

Table 2: Test cases for keypad operation in Part 1.

Part 2

In part 2, we followed a similar procedure as above for finding the maximum possible delay period in which flickering was no longer visible in the LED. As test cases follow the same format as above, it will be omitted. Here, we found the optimal delay to be 24 ticks.

We also tested the operation of the buttons to verify that they correctly changed the brightness of the LED, and verified that edge cases were covered. Example test cases are shown in Table 3.

Test Case #	**Operation**	**Expected Result**
1	Press the button corresponding to decreasing brightness (the right-most button).	The brightness of the RGB LED is decreased.
2	Press the button corresponding to increasing brightness (the leftmost button).	The brightness of the RGB LED is increased.
3	Hold, or repeatedly press the right-most button until the brightness of the LED stops changing.	The LED is dim, but not entirely off; an ON delay <= 0 is avoided, and verified in serial output.
4	Hold, or repeatedly press the left-most button until the brightness of the LED stops changing.	The LED is bright, and no longer changes brightness; an OFF delay <= 0 is avoided, and verified in serial output.

Table 3: Test cases for RGB LED brightness control.

Part 3

In part 3, the system remained functionally equivalent given the only changes made were refactoring tasks, so testing occurred no differently from part 1 and 2. Example test cases for part 3 are the same as those in parts 1-2 and will be omitted here.

Conclusion

The objectives of this lab were successfully met. The system correctly interfaced the keypad, seven-segment display, RGB LED, and pushbuttons using FreeRTOS, demonstrating proper task creation, scheduling, and inter-task communication. The seven-segment display reliably showed the current key on the right digit and the previous key on the left without visible flicker, confirming that the selected multiplexing delay achieved persistence of vision. Additionally, the RGB LED brightness was successfully adjusted using software-based PWM, with pushbutton inputs modifying the duty cycle as intended. The final implementation using queues effectively separated hardware control from input processing, resulting in a modular and well-structured embedded system design.

Throughout the lab, minor challenges included tuning delay values to eliminate flicker, ensuring stable PWM behavior, and debugging task synchronization when introducing queues. A more significant issue occurred when implementing queue-based communication for the display. Initially, only the newest key press was sent through the queue, and the vDisplayTask attempted to update both the current and previous key values locally. This resulted in the same digit being shown on both sides of the display, since the historical state was not being preserved correctly across tasks. The issue was resolved by defining a structured data type that contained both the current and previous key values and passing this struct as a single queue element. By transmitting the complete display state instead of reconstructing it in the consumer task, we eliminated the synchronization problem and ensured consistent, correct output. Overall, the lab provided valuable hands-on experience in RTOS-based embedded system design, reinforcing concepts such as task decomposition, producer-consumer architecture, and time-based control in real-time applications.