ECE 443 - Project #3

Collin Heist

September 30, 2019

Contents

Listings

1 Implementation

1.1 Initialization

The most obvious place to start was incorporating my old EEPROM, LCD and UART libraries. Luckily, they were functional back from the ECE 341 lab. I started with changing my hardware initialization function to include each library's initialization function, shown in **Listing 1**. *Note, no code listings show any TraceAlyzer code to avoid clutter.*

Listing 1: Hardware Initialization

```
static void init_hardware() {
  chipKIT_PRO_MX7_Setup();
  initialize_LCD();
  init_eeprom();
  initialize_uart1(19200, ODD_PARITY);

  PORTSetPinsDigitalOut(IOPORT_B, SM_LEDS);
  LATBCLR = SM_LEDS;

  PORTSetPinsDigitalIn(IOPORT_G, BTN1);

  mCNOpen(CN_ON, CN8_ENABLE, 0);
  mCNSetIntPriority(1);
  mCNSetIntSubPriority(0);
  unsigned int x = PORTReadBits(IOPORT_G, BTN1);
  mCNClearIntFlag();
  mCNIntEnable(1);

  ConfigIntUART1(UART_ERR_INT_DIS | UART_RX_INT_EN |
    UART_INT_PR2 | UART_INT_SUB_PR0 | UART_TX_INT_DIS);

  INTEnableSystemMultiVectoredInt();
}
```

Of course, I also needed to declare **BTN1** as an input, and enable the change notice interrupt for that button. Similarly, I also configured the UART peripheral to trigger an interrupt on **RX** (or receive). Both of these interrupts required corresponding assembly files that allowed them to more appropriately work with the **FreeRTOS** task structure. Those files are **cn_isr_wrapper.S** and **uartRX_isr_wrapper.S**.

I chose a priority level of one for my change notice, and two for the UART because the user can theoretically enter characters in very rapid succession, but each button press requires at least 20 milliseconds to debounce. In addition to this, the responsiveness of a button press is largely dictated by the time it takes to read from the EEPROM, display to the LCD, and the project-stated one second delay between clearing the screen and writing a message – all of which are orders of magnitude greater than the delay created by having the UART interrupt trigger.

1.2 Program Design

After the obvious project requirements were taken care of, I started designing the actual functionality of the program. I decided to utilize two semaphores, and two queues. Each wrapper ISR requires a semaphore in order to return to the desired wrapper task, so that much was obviously needed. And I used one queue to handle the 'stream' of addresses where the user-entered messages are stored in the EEPROM, and the second queue contains a list of pending retrieval requests. This pending retrieval queue has meaningless data, and I could have instead used a counting semaphore – but this is functionally equivalent, and allows for additional data to be transmitted in the future (if necessary).

1.3 Message Input & EEPROM Writing

The easiest component to implement was the UART handler. This required its own assembly wrapper file, and is triggered every time a character is entered on the UART interface (over Putty). Below, in **Listing 2**.

Listing 2: UART ISR Handler

```
void isr_uart_RX_handler(void) {
  portBASE_TYPE move_to_higher_priority = pdFALSE;

  if (getStrU1(uart_input, sizeof(uart_input))) {
    putcU1('\n');
    xSemaphoreGiveFromISR(write_to_eeprom,
      &move_to_higher_priority);

    mPORTBClearBits(LEDA);
  }

  mU1RXClearIntFlag();

  portEND_SWITCHING_ISR(move_to_higher_priority);
}
```

This interrupt is called from the assembly wrapper, and using the **getStrU1()** function written into my **comm** library, the buffer **uart_input** is filled with the incoming message. This function returns true when a terminating character is detected, at which point a newline is echoed (to better format the Putty window), and then the semaphore is given to initiate a *write* to the EEPROM. After a semaphore is given, the interrupt flag is cleared, and then a switch is initiated to either the calling task or the newly unblocked EEPROM writing *task* – which is shown below in **Listing 3**.

Listing 3: EEPROM Writing Task

```
static void task_write_EEPROM(void *task_params) {
  unsigned int i = 0;
  static unsigned int eeprom_write_addr = 0;
```

```
    portBASE_TYPE queue_status;

    xSemaphoreTake(write_to_eeprom, 0);
    for (;;) {
      xSemaphoreTake(write_to_eeprom, portMAX_DELAY);

      if (uxQueueMessagesWaiting(eeprom_addr_queue) ==
        MAX_NUM_MSGS) {}
      else {
        unsigned int write_error = NO_ERR;

        write_error = write_eeprom(EEPROM_SLAVE_ADDR,
          eeprom_write_addr, uart_input, UART_MAX_MSG_SIZE);
        if (write_error) {}
        else {
          queue_status = xQueueSendToBack(eeprom_addr_queue,
            &eeprom_write_addr, 0);

          eeprom_write_addr += UART_MAX_MSG_SIZE;
          eeprom_write_addr %= UART_MAX_MSG_SIZE * MAX_NUM_MSGS;
        }
      }

      for (i = 0; i < UART_MAX_MSG_SIZE; i++)
        uart_input[i] = 0;

      if (uxQueueMessagesWaiting(eeprom_addr_queue)
        != MAX_NUM_MSGS)
        mPORTBSetBits(LEDA);
    }
  }
```

As with typical task functions, this is implemented with an infinite for loop. The loop blocks forever – until the **write_to_eeprom** semaphore is given from the UART ISR handler (see **Listing 2**). The reception of this semaphore indicates that a complete message has been written into the **uart_input** character buffer, and is ready to be written to the EEPROM. After the write is finished, the starting address of that message is sent to the address queue, and then the global address variable is incremented and wrapped back around (to enforce the message count limitation).

1.4 EEPROM Reading & LCD Writing

After parsing and writing the messages, button presses need to initiate a read from the EEPROM and display that message to the LCD. For starters, all presses of button one activate the change-notice interrupt, shown below.

Listing 4: Change Notice ISR Handler

```
void isr_change_notice_handler(void) {
  portBASE_TYPE move_to_higher_priority = pdFALSE;

  xSemaphoreGiveFromISR(cn_semaphore,
    &move_to_higher_priority);

  mCNClearIntFlag();
  mCNOpen(CN_OFF, (CN8_ENABLE), 0);

  portEND_SWITCHING_ISR(move_to_higher_priority);
}
```

Very simply, a button press gives the **cn_semaphore**, which unblocks the code shown in **Listing 5**, and then clears the change notice interrupt flag. I did not include a button debounce inside this function because task execution needs to continue to occur during that debounce period. The handler *task* that is unblocked by the semaphore given in the change notice ISR handler executes a vast majority of the change notice code.

Listing 5: Change Notice Handler Task

```
static void task_change_notice_handler(void *task_params) {
  portBASE_TYPE queue_status;
  unsigned int dummy_val = 1;
  unsigned int current_btn1_status = 0;

  xSemaphoreTake(cn_semaphore, 0);
  for (;;) {
    xSemaphoreTake(cn_semaphore, portMAX_DELAY);
    current_btn1_status = PORTG & BTN1;

    vTaskDelay(MS_TO_TICKS(DEBOUNCE_TIME_MS));

    if (previous_BTN1_status == 0 && current_btn1_status)
      queue_status = xQueueSendToBack(eeprom_pending_queue,
        &dummy_val, 0);

    previous_BTN1_status = current_btn1_status;
  }
}
```

This task waits for the change notice ISR semaphore forever, and once received the status of button 1 is read, and then a debounce is performed for **DEBOUNCE_TIME_MS** (default 20, in the header) milliseconds. After this blocking is finished, the previous button status is compared with the current button status (in order to detect only button *presses* and not releases) and then a dummy value is added to the pending request queue. This was addressed earlier, but a counting semaphore could have been used here, but I chose to use a queue here because it was easier, and allows for more exchanging of information in the future. For this implementation, the very presence of a value in the pending queue indicates that a message *wants* to be read from the EEPROM.

After a value is added to the pending queue, the below reading task unblocks. This task is shown in **Listing 6**.

Listing 6: EEPROM Reading Task

```
static void task_read_EEPROM(void* task_params) {
  portBASE_TYPE queue_status;
  unsigned int eeprom_read_addr, dummy_val, i, line_index;

  for (;;) {
    char eeprom_message[UART_MAX_MSG_SIZE+1] = {0};
    char lcd_message[LCD_CHAR_WIDTH+1] = {0};

    xQueueReceive(eeprom_pending_queue, &dummy_val,
      portMAX_DELAY);
    queue_status = xQueueReceive(eeprom_addr_queue,
      &eeprom_read_addr, 0);
    if (queue_status == pdTRUE) {
      unsigned int read_error = NO_ERR;
      read_error = read_eeprom(EEPROM_SLAVE_ADDR,
        eeprom_read_addr, eeprom_message, UART_MAX_MSG_SIZE);

      format_message_LCD(eeprom_message, UART_MAX_MSG_SIZE,
        LCD_CHAR_WIDTH);

      unsigned int num_lines = get_line_count(eeprom_message,
        UART_MAX_MSG_SIZE);

      reset_clear_LCD();
      vTaskDelay(MS_TO_TICKS(LCD_BLANK_PERIOD_MS));
      set_cursor_LCD(SECOND_LINE_START);
      get_row_string(eeprom_message, UART_MAX_MSG_SIZE, 0,
        lcd_message, LCD_CHAR_WIDTH + 1);
      put_string_LCD(lcd_message);

      for (line_index = 1; line_index < num_lines + 1;
        line_index++) {

        vTaskDelay(MS_TO_TICKS(LCD_ROLLING_DELAY_MS));
        reset_clear_LCD();
        set_cursor_LCD(FIRST_LINE_START);
        get_row_string(eeprom_message, UART_MAX_MSG_SIZE,
          line_index - 1, lcd_message, LCD_CHAR_WIDTH + 1);
        put_string_LCD(lcd_message);
        set_cursor_LCD(SECOND_LINE_START);
        get_row_string(eeprom_message, UART_MAX_MSG_SIZE,
          line_index, lcd_message, LCD_CHAR_WIDTH + 1);
        put_string_LCD(lcd_message);
      }

      vTaskDelay(MS_TO_TICKS(LCD_ROLLING_DELAY_MS));
      reset_clear_LCD();
      set_cursor_LCD(FIRST_LINE_START);
      get_row_string(eeprom_message, UART_MAX_MSG_SIZE,
        num_lines, lcd_message, LCD_CHAR_WIDTH + 1);
      put_string_LCD(lcd_message);
      vTaskDelay(MS_TO_TICKS(LCD_ROLLING_DELAY_MS));
      reset_clear_LCD();
    }
  }
}
```

This task accomplishes quite a lot. To begin, two character arrays are used: there is **eeprom_message** that stores the message read from the EEPROM, and **lcd_message** that keeps the temporary string being written to the LCD. The task blocks forever until the pending queue has dummy values in it, which happens in the button press handling task. After this, a value from the address queue (**eeprom_addr_queue**) is attempted to be read from. This read does not block at all so that the task finishes even if no data is available in the EEPROM. Assuming data had been previously written into the address queue, a read from the EEPROM is initiated at that address, and the result is stored in **eeprom_message**. I then call **format_message_LCD()**, which is a function I wrote that loops through all characters in a provided character array, and replaces spaces with newline characters ('$\backslash n$') where necessary, in order to have at most **LCD_CHAR_WIDTH** (16 for our case) characters on each 'line' of the LCD. This function changes the EEPROM message itself.

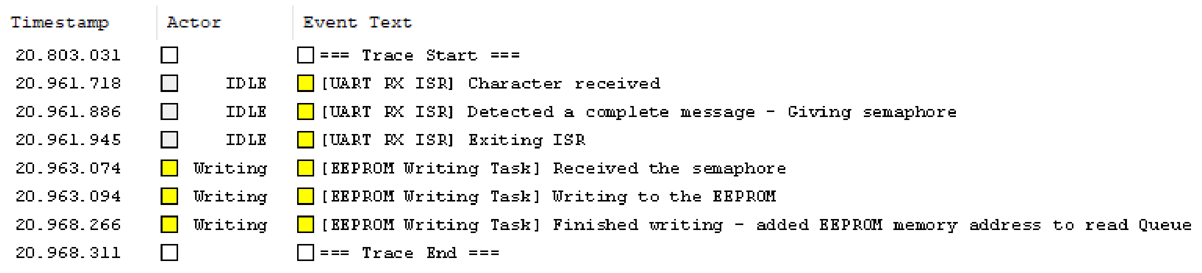
With the message formatted, I could the number of how many lines are in the message (from my **get_line_count()** function). This is used to loop through the message line-by-line. The LCD is cleared and kept blank for the desired delay (1000 ms in the guidelines) – during this time other task execution may occur – and then the LCD writing operations begin. The general outline of this is to loop through each line of the EEPROM message, storing the temporary string inside **lcd_message**, and place that message on the second or first line of the LCD, and after the proper delay this is repeated. Finally, the LCD is cleared and the function exits.

2 Testing & Verification

In order to validate this program, I added a lot of TraceAlyzer outputs whenever various events occur, these can be found in the original code source on the repository.

2.1 Message Write

The first test was a generic message write, which would be initiated by a return-terminated message over the UART connection. Below, in **Figure 1**, is the TraceAlyzer output of this event. The one millisecond heartbeat task is ignored, as are the system outputs (kernel ticks, etc.), in all TraceAlyzer outputs.



```
Timestamp    Actor        Event Text
20.803.031   ☐            ☐=== Trace Start ===
20.961.718   ☐     IDLE   🟨[UART RX ISR] Character received
20.961.886   ☐     IDLE   🟨[UART RX ISR] Detected a complete message - Giving semaphore
20.961.945   ☐     IDLE   🟨[UART RX ISR] Exiting ISR
20.963.074   🟨 Writing   🟨[EEPROM Writing Task] Received the semaphore
20.963.094   🟨 Writing   🟨[EEPROM Writing Task] Writing to the EEPROM
20.968.266   🟨 Writing   🟨[EEPROM Writing Task] Finished writing - added EEPROM memory address to read Queue
20.968.311   ☐            ☐=== Trace End ===
```

Figure 1: TraceAlyzer event log of a complete message **write**, triggered by hitting return on the keyboard

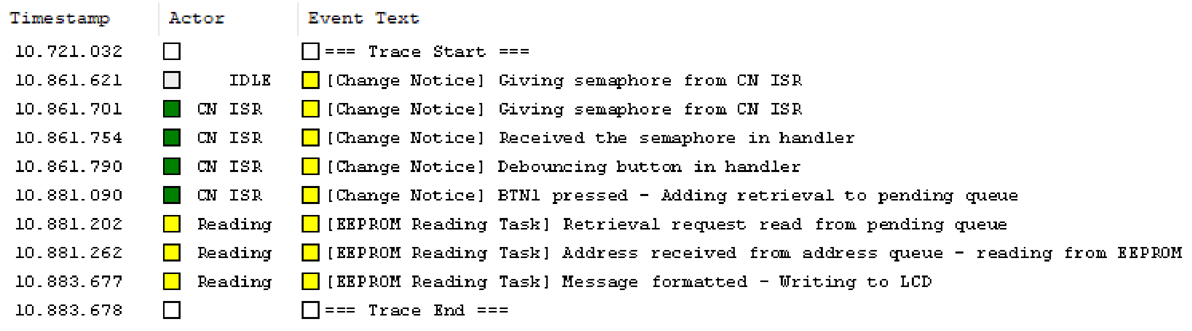
The sequence of events shown in this figure are as follows:

1. The return character is received over the UART interface - triggering the UART ISR.
2. **getStr()** returns true, signifying a complete message, resulting in the giving of the **write_to_eeprom** semaphore.
3. The UART ISR terminates and returned to the newly unblocked EEPROM writing task.
4. The message stored in **uart_input** is written to the EEPROM.
5. The start address in EEPROM memory is added to the address queue.

This sequence of events is exactly as I expect (and is desired) for a message reception.

2.2 Message Read

The next use-case I tested was a message retrieval initiated by a button press while a message was available to be read. Here's the TraceAlyzer output for this:



```
Timestamp    Actor        Event Text
10.721.032   ☐            ☐=== Trace Start ===
10.861.621   ☐     IDLE   🟨[Change Notice] Giving semaphore from CN ISR
10.861.701   🟩 CN ISR    🟨[Change Notice] Giving semaphore from CN ISR
10.861.754   🟩 CN ISR    🟨[Change Notice] Received the semaphore in handler
10.861.790   🟩 CN ISR    🟨[Change Notice] Debouncing button in handler
10.881.090   🟩 CN ISR    🟨[Change Notice] BTN1 pressed - Adding retrieval to pending queue
10.881.202   🟨 Reading   🟨[EEPROM Reading Task] Retrieval request read from pending queue
10.881.262   🟨 Reading   🟨[EEPROM Reading Task] Address received from address queue - reading from EEPROM
10.883.677   🟨 Reading   🟨[EEPROM Reading Task] Message formatted - Writing to LCD
10.883.678   ☐            ☐=== Trace End ===
```

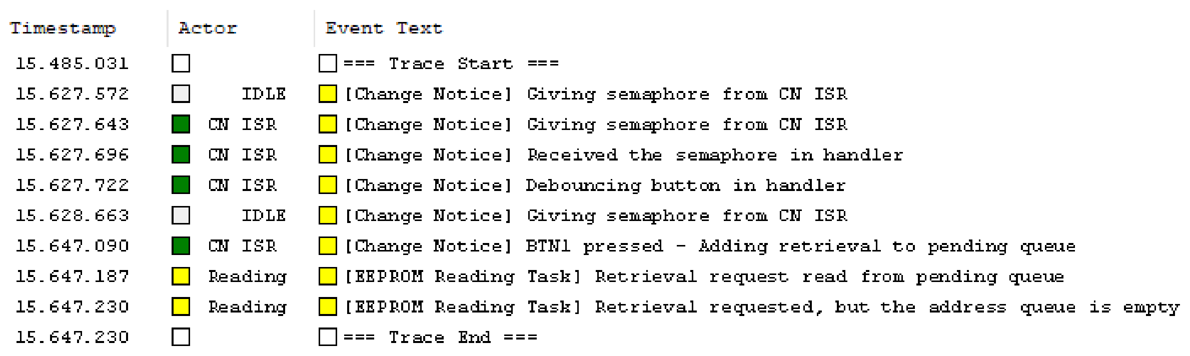
Figure 2: TraceAlyzer event log of a message retrieval from the EEPROM, triggered by a press of **BTN1**

Upon the button being pressed, the following happens:

1. The button press activates the change-notice ISR.
2. The handler ISR gives the **cn_semaphore** and returns to the newly unblocked change notice handler task.
3. The handler task unblocks, and begins debouncing for 20 milliseconds.
4. After the debounce finishes, a button *press* is detected, and a retrieval request is added to the pending queue.
5. The eeprom-reading task is now unblocked since the **eeprom_pending_queue** has a value in it – starting a read from the EEPROM.
6. The message is read from the EEPROM, formatted, and written to the LCD.

2.3 Message Request With No Messages Available

With these two 'typical' use-cases verified, I also checked more particular error-prone situations, such as a button press when **no** messages are available. This is shown in **Figure 3**.



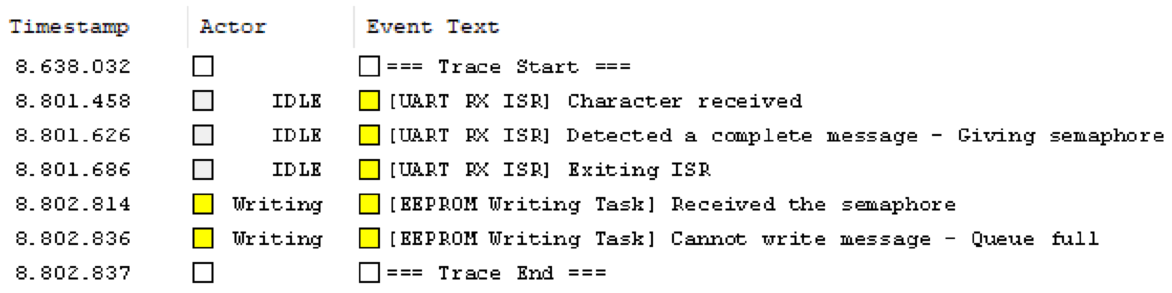
```
Timestamp     Actor          Event Text
15.485.031    ☐              ☐ === Trace Start ===
15.627.572    ☐    IDLE      🟨 [Change Notice] Giving semaphore from CN ISR
15.627.643    🟩 CN ISR      🟨 [Change Notice] Giving semaphore from CN ISR
15.627.696    🟩 CN ISR      🟨 [Change Notice] Received the semaphore in handler
15.627.722    🟩 CN ISR      🟨 [Change Notice] Debouncing button in handler
15.628.663    ☐    IDLE      🟨 [Change Notice] Giving semaphore from CN ISR
15.647.090    🟩 CN ISR      🟨 [Change Notice] BTN1 pressed - Adding retrieval to pending queue
15.647.187    🟨 Reading     🟨 [EEPROM Reading Task] Retrieval request read from pending queue
15.647.230    🟨 Reading     🟨 [EEPROM Reading Task] Retrieval requested, but the address queue is empty
15.647.230    ☐              ☐ === Trace End ===
```

Figure 3: TraceAlyzer event log of a message retrieval request with no available messages

The events shown in this log are practically identical to those that occur in a normal message retrieval request, shown in **Figure 2**, except after the request queue unblocks, the non-blocking **xQueueReceive()** function call does not return **pdTRUE**, indicating a successful read from the queue. Because no addresses had been added to the queue, as no messages had been written to the EEPROM without already being read, the queue is empty and the system is unable to display anything on the LCD.

2.4 User Enters More Than Five Messages

Finally, I tested the situation where the user enters more than five messages at a time. I designed my program to work such that in this situation, nothing will happen to either the global address pointer, or the address queue. If TraceAlyzer is not being used, then technically the user will not know that this message is being ignored. The TraceAlyzer output is shown below:



Timestamp	Actor	Event Text
8.638.032	☐	☐ === Trace Start ===
8.801.458	☐ IDLE	🟨 [UART RX ISR] Character received
8.801.626	☐ IDLE	🟨 [UART RX ISR] Detected a complete message - Giving semaphore
8.801.686	☐ IDLE	🟨 [UART RX ISR] Exiting ISR
8.802.814	🟨 Writing	🟨 [EEPROM Writing Task] Received the semaphore
8.802.836	🟨 Writing	🟨 [EEPROM Writing Task] Cannot write message - Queue full
8.802.837	☐	☐ === Trace End ===

Figure 4: TraceAlyzer event log of the user entering more than five messages

3 Design Overview

I was able to avoid using global variables almost entirely. By using semaphore for initiating writing and reading from the EEPROM, and queues for transferring the sequences of address to read to, I alleviated the need for a lot of global arrays and variables.

If I wanted to avoid using *any* global variables, I could have moved my global **uart_message** character array to a queue that's the length of the maximum message. However, this takes up a lot of memory, and adds a significant amount of overhead with adding to and removing from queues, and would require me to manually take the input from the UART library, which adds to a character array, and add each character to a queue one-by-one.

All in all, every component of my project works as desired. I also made many components of the design adjustable by changing the **#define** statements inside **main.h**. You can change the following:

- **DEBOUNCE_TIME_MS** - How many milliseconds every press of **BTN1** is debounced
- **UART_MAX_MSG_SIZE** - The maximum allowed size of any one message entered over the UART
- **MAX_NUM_MSGS** - The maximum number of messages stored in the EEPROM. If more messages are entered, they are simply ignored
- **LCD_CHAR_WIDTH** - How wide the LCD is (in characters), can be changed to just change how the messages are displayed, does not need to match the actual width of the LCD
- **LCD_BLANK_PERIOD_MS** - How many milliseconds to blank the LCD before a message begins being displayed
- **LCD_ROLLING_DELAY_MS** - How many milliseconds to wait between each upward roll of the LCD