# ECE 443 - Project #6

Collin Heist

28th January 2020

## Contents

# 1 Design

I chose to partition my tasks by purpose. This ended up neatly partitioning the resources being used in this project, as well. In general, I decided to create the following tasks, each with the broad general purpose:

- **IO Reading Task** - Periodic task that reads from the temperature and motor speed buffer.
- **RTR Sending Task** - Periodic task that sends an RTR request from CAN1 to CAN2.
- **Change Notice Handler Task** - A really short task that processes presses of BTN1 into transitions between states of my control unit FSM.
- **Control Unit Task** - A task that implements the functionality of the control unit as two *states* of a finite state machine.
- **PWM Updating Task** - A task that looks at the CAN2 RX channel for desired changes to the PWM output.

# 2 IO Reading Task

As described in the project outline, the IO Unit needed to periodically read the temperature and motor speeds, every 500 milliseconds. I chose to implement this as shown in **Listing 1**.

Listing 1: IO Reading Task

```
static void task_read_IO(void* task_params) {
  float latest_temp, latest_rps;
  const TickType_t task_frequency= MS_TO_TICKS(IO_FREQ_MS);
  TickType_t last_time_awake = xTaskGetTickCount();
  for (;;) {
    latest_temp = read_ir_temp();
    latest_rps = get_average_rps();

    CAN2_refill_RTR_buffer(latest_temp, latest_rps,
      latest_pwm_setting);

    vTaskDelayUntil(&last_time_awake, task_frequency);
  }
}
```

This is a really simple task that on each iteration reads from the IR sensor and gets the average RPS values from my IR sensor and input capture libraries. Because the two *units* should not communicate directly (that's what the CAN network is for), these variables are localized to this task. The exception to this is **latest_pwm_setting**, which is global. This is because this variable is changed inside the PWM updating task

(see **Section 6**). I could have used a queue to pass this data between these two tasks, or perhaps implemented the two tasks together, but I decided to treat the variable as 'private' (in a sense) and not permit any control unit code from accessing it.

The function, **CAN2_refill_RTR_buffer()** is non-blocking and ensures that there is always one data frame inside the CAN2 TX channel. This function's code is shown here:

Listing 2: Refill CAN2 RTR Buffer Function

```
void CAN2_refill_RTR_buffer(float temperature,
  float motor_speed, float pwm_setting) {

  CANTxMessageBuffer* message;

  if ((CANGetChannelEvent(CAN2, CAN_CHANNEL0) &
     CAN_TX_CHANNEL_EMPTY) == 0)
    return;

  message = CANGetTxMessageBuffer(CAN2, CAN_CHANNEL0);

  // Clear the message values
  // Code omitted to save space

  // Format and create the RTR message itself
  // Code omitted to save space

  CANUpdateChannel(CAN2, CAN_CHANNEL0);
}
```

As you can see, I first check if the CAN2 TX channel is empty. If it is, that means that the CAN2 module has responded to an RTR sent by CAN1. In this case, the buffer is first cleared (by assigning each **messageWord** field of the message structure to 0), and then the message is created. Without filling up an entire page, this sets the SID to the CAN2 RTR message ID, and sets the data-length code to 8.

The data itself is then partitioned into 8 sets of bytes. I chose to send a 10x scaled version of the temperature, and PWM setting, and a 100x scaled version of the motor speed. I did this in order to preserve the $\frac{1}{10}th$ place of each variable (and the $\frac{1}{100}th$ place for the motor speed), without taking up the entire 8 available bytes by sending a floating point value. This is achievable because even the 10x scaled values of these variables are well within the range of a *short int* – and for the 100x value of the motor speed, the data is sent across 4 bytes.

Because this function immediately returns if no new value is placed in the RTR TX channel, technically the individual RTR requests can be responded to with old data (up to 2 seconds old in the worst case). I went with this implementation because I found no way of (1) *replacing* old values inside the RTR TX channel after they've been created, and (2) parsing more than one message from CAN1 (should the buffer just be added to), as each time the **CANUpdateChannel()** function is called, the buffer is cleared. In implementation, this is less of a problem than it sounds. Worst case, I could redesign the program to request data from the IO unit more often, and this would alleviate the problem.

This task is given a priority level of 3, with only the change-notice handler task being a higher priority. I chose this because reading from the IO units is important (and should take priority over the Control Unit FSM and PWM updating tasks), but takes very little time to execute and happens very infrequently – alleviating the risk of taking away execution time of other tasks.

# 3 RTR Sending Task

This is by far the simplest task in my project. Once again, the project outline says that an RTR should be sent to the IO unit every 2000 milliseconds, so I implemented this as a periodic task with the only real 'code' being the **CAN1_send_RTR()** function. The code for this task is shown in **Listing 3**.

Listing 3: RTR Sending Task

```
static void task_send_RTR(void* task_params) {
  const TickType_t task_frequency = MS_TO_TICKS(RTR_FREQ_MS);
  TickType_t last_time_awake = xTaskGetTickCount();
  for (;;) {
    CAN1_send_RTR();
    LATBINV = LEDA;

    vTaskDelayUntil(&last_time_awake, task_frequency);
  }
}
```

Unblocking every **RTR_FREQ_MS** milliseconds (2000 as defined in the header), this just calls my CAN library function which sends and RTR from CAN1 to CAN2.

This function is implemented at a task priority level of 3, the same as the IO reading task. I chose this priority level because it should only really be interrupted by the change-notice handler task, and it is fine to have the same priority level as the IO reading task – as they're both very short-execution periodic tasks.

# 4 Change Notice Handler Task

This task is exactly identical to previous projects. The change notice interrupt gives a semaphore to unblock this task, which then debounces the button. Upon a **button press** (as read by comparing the previous state of BTN1 to the current state of BTN1), the current global state of the Control Unit is toggled between configuration and operational mode. This task is the highest priority task in the project, as the responsiveness of the system is directly affected by this task's execution, and the execution time of this task is nearly negligible.

# 5 Control Unit Task

This is the first (and only) notably large task in this project. The entirety of the control-unit functionality (aside from sending RTR's to CAN2) is implemented in this task as a

finite state machine. I will start with the state machine's declaration, and the code that executes every iteration of the task.

## 5.1 Background FSM Code

Listing 4: Control Unit Implementation

```
enum STATE {CONFIGURATION_MODE, OPERATIONAL_MODE}
  current_state = CONFIGURATION_MODE;

static void task_control_FSM(void* task_params) {
  char top_lcd_str[18] = {'\0'};
  char bottom_lcd_str[18] = {'\0'};
  float temp = 0, rps = 0, pwm = 0;
  float pwm_low_point = 0, pwm_high_point = 0;
  float desired_pwm = 0;
  unsigned int new_data_flag = FALSE;

  for (;;) {
    clear_string_buffer(top_lcd_str, sizeof(top_lcd_str));
    clear_string_buffer(bottom_lcd_str, sizeof(bottom_lcd_str));

    if (CAN1_process_RX(&temp, &rps, &pwm) ==
      CAN_MESSAGE_RECEIVED) {

      LATBINV = LEDB;
      new_data_flag = TRUE;
    }

    switch (current_state) {
      case CONFIGURATION_MODE:
        // Code omitted
        break;
      case OPERATIONAL_MODE:
        // Code omitted
        break;
    }

    taskYIELD();
  }
}
```

As typical in **C**, the FSM itself is implemented as an enum, and the machine is initialized to configuration mode (as outlined in the project description).

As for each loop of the task, it starts by clearing the two string buffers being utilized. I wrote the **clear_string_buffer** function to loop through all elements in the given character array and ensure each element is zero (the null character). This is done to prevent old messages from ending up on the LCD, and is important as these buffers are

*technically* static and are therefore never cleaned up on their own.

Afterwards, the RX channel of CAN1 is checked. This function returns **CAN_NO_MESSAGE_RECEIVED** if the RX channel was empty – indicating that no RTR message has been sent or responded to. However, if it returns **CAN_MESSAGE_RECEIVED**, then the three RTR variables (temperature, motor speed, and pwm setting) have their values updated based on the contents of the CAN1 RX channel. This event is noted by setting the new data flag to **TRUE**.

Next, the two states of the FSM are implemented (shown in **Listing 5** and **7**). And the task yields. This is important because this task's execution time can be quite long (with a lot of LCD functionality), and so other tasks should not be starved.

## 5.2 Configuration Mode

Should the FSM currently be in **CONFIGURATION_MODE**, then the following code will execute:

Listing 5: Control Unit - Configuration Mode

```
case CONFIGURATION_MODE:
    LATGCLR = LED1;

    if (temp == 0)
      strcpy(top_lcd_str, BLANK_LINE);
    else
      sprintf(top_lcd_str, "    %02.1f    ", temp);

    set_cursor_LCD(FIRST_LINE_START);
    put_string_LCD(top_lcd_str);

    pwm_low_point = (PORTA & BTN3) ?
      temp : pwm_low_point;
    pwm_high_point = (PORTG & BTN2) ?
      temp : pwm_high_point;

    pwm_low_point = (pwm_high_point < pwm_low_point) ?
      0 : pwm_low_point;
    pwm_high_point = (pwm_high_point < pwm_low_point) ?
      0 : pwm_high_point;

    if (pwm_low_point == 0 && pwm_high_point == 0)
      strcpy(bottom_lcd_str, BLANK_LINE);
    else
      sprintf(bottom_lcd_str, "%02.1f   %02.1f",
        pwm_low_point, pwm_high_point);

    set_cursor_LCD(SECOND_LINE_START);
```

```
    put_string_LCD(bottom_lcd_str);
    break;
```

The project outline gives descriptions as to how the top and bottom lines of the LCD should be displayed during this mode, and a majority of this function is simply manipulating the top and bottom LCD string accordingly. If button 2 or 3 are pressed during this task's execution, then the values for the PWM low and high point are set based off the current temperature.

This task has so many local variables because it explicitly cannot reference the global values of the temperature, motor speed, and PWM cycle – as this would defeat the purpose of the CAN network. The most up-to-date values of the temperature (for this task) is obtained from the code shown in **Listing 4** that checks the CAN1 RX channel.

## 5.3 Operational Mode

Finally, if the unit is in **OPERATIONAL_MODE**, then this code runs:

Listing 6: Control Unit - Operational Mode

```
case OPERATIONAL_MODE:
  LATGSET = LED1;

  if (new_data_flag) {
    new_data_flag = FALSE;
    LATBINV = LEDB;
    sprintf(top_lcd_str, "%d%%      %05.2f", (int)pwm, rps);
    sprintf(bottom_lcd_str, "%03.1f  %03.1f  %03.1f",
      pwm_low_point, temp, pwm_high_point);

    set_cursor_LCD(FIRST_LINE_START);
    put_string_LCD(top_lcd_str);
    set_cursor_LCD(SECOND_LINE_START);
    put_string_LCD(bottom_lcd_str);

    if (temp < pwm_low_point)
      desired_pwm = (float) PWM_MIN_VAL;
    else if (temp > pwm_high_point)
      desired_pwm = (float) PWM_MAX_VAL;
    else
      desired_pwm = (float)((PWM_LINEAR_MAX-PWM_LINEAR_MIN)*
          (temp - pwm_low_point) /
          (pwm_high_point - pwm_low_point) + PWM_LINEAR_MIN);

    CAN1_send_TX(desired_pwm);
    LATBINV = LEDC;
  }
  break;
```

If no new data has been received since the last iteration of this task, nothing new needs to be done. However, if the CAN1 RX channel was not empty and the flag was set, then the LCD strings are reconstructed and displayed onto the screen. The only other functionality in this mode is to compute the newly desired PWM value, based off the current temperature (as well as the low and high set points), and send this data via CAN1 to CAN2.

The **CAN1_send_TX()** function is also quite simple, and sends the desired PWM setting as the 10x scaled value in order to preserve the $\frac{1}{10}th$ place. This is possible (once again) because the maximum value for the PWM is 100.0%, and therefore can be represented by at most 1000 - well within the range of a *short int*.

This task is given a priority of 2, lower than all other tasks except the PWM updating task. This is because all higher priority tasks are quicker to execute, and this task can be quite slow with all the LCD operations – *and* fast-response time from this task is not necessary.

# 6 PWM Updating Task

Finally, the PWM updating task is designed to be a non-periodic task that should ideally execute in round-robin style with the control unit task. They are both given a priority of 2, and yield at the end of their task's execution.

The code for this task is shown in **Listing ??**, below.

Listing 7: Control Unit - Operational Mode

```
static void task_update_pwm(void* task_params) {
  for (;;) {
    if (CAN2_process_RX(&latest_pwm_setting) ==
      CAN_MESSAGE_RECEIVED) {

      if (set_pwm((unsigned int) latest_pwm_setting))
        // An error occurred
      else
        LATBINV = LEDD;
    }

    taskYIELD();
  }
}
```

Should the CAN2 RX channel be non-empty (indicating a new PWM setting is desired by the control unit), then that value is used to set the PWM module. In the event that the RX channel was empty, and no update to the PWM duty-cycle is desired, then this task shall immediately yield.

Here is where the global **latest_pwm_setting** variable is set, and a queue *could* have been used to send this data directly to the IO reading task – but as discussed before, this would add overhead and I simply avoided referencing the global variable inside the control unit functions to appropriately partition my two units.