# COMPENG 4DS4 Project 1 Report

Aaron Pinto pintoa9
Raeed Hassan hassam41
Jingming Liu liuj171
Jeffrey Guo guoj69

March 17, 2023

## Declaration of Contributions

| Task | Contributions |
|---|---|
| Motor | Raeed, Aaron, Jingming |
| Position | Raeed, Jingming |
| LED | Raeed, Jeffrey |
| RC | Raeed, Aaron, Jingming |

## Motor Task

The motor task controls the DC motor's speed by calculating the PWM duty cycle and controlling the FTM module. The task will wait for and receive messages from the motor queue, and will convert the values received on the queue to the appropriate PWM duty cycle for the FTM module. The motor queue can contain one single byte integer, and the value sent into the queue from the RC task will be between −100 and 100. When the motor task receives an integer from the motor queue, it will compare the motor value with the previous motor value, and if the value has changed it will convert the value to its corresponding PWM duty cycle, and update the FTM module with this duty cycle. We only trigger the FTM module when the motor value has changed as we experienced jittering with the motor when the task triggers the FTM module every time when the motor value has not changed.

The motor task is shown in Listing 1.

Listing 1: Motor Task

```c
void motorTask(void *pvParameters) {
    // Motor task implementation
    BaseType_t status;
    int motorInput;
    float motorDutyCycle;

    while (1) {
        status = xQueueReceive(motor_queue, (void*) &motorInput, portMAX_DELAY);

        if (status != pdPASS) {
            PRINTF("Queue Receive failed!.\r\n");

            while (1)
                ;
        }

        if (prevMotorInput != motorInput) {
            prevMotorInput = motorInput;
            motorDutyCycle = motorInput * 0.025f / 100.0f + 0.0760;
            updatePWM_dutyCycle(FTM_CHANNEL_DC_MOTOR, motorDutyCycle);
            FTM_SetSoftwareTrigger(FTM_MOTOR, true);
        }

        vTaskDelay(1 / portTICK_PERIOD_MS);
    }
}
```

## Position Task

The position task controls the servo motor's angle by calculating the PWM duty cycle and controlling the FTM module. The task will wait for and receive messages from the angle queue, and will convert the values received on the queue to the appropriate PWM duty cycle for the FTM module. The angle queue can contain one single byte integer, and the value sent into the queue from the RC task will be between −45 and 45. When the position task receives an integer from the angle queue, it will compare the servo angle value with the previous servo input value, and if the value has changed it will convert the value to its corresponding PWM duty cycle, and update the FTM module with this duty cycle. We only trigger the FTM module when the servo angle value has changed as we experienced jittering with the servo motor when the task triggers the FTM module every time even when the servo angle value has not changed.

The position task is shown in Listing 2.

Listing 2: Position Task

```c
void positionTask(void *pvParameters) {
    // Position task implementation
    BaseType_t status;
    int servoInput;
    float servoDutyCycle;

    while (1) {
        status = xQueueReceive(angle_queue, (void*) &servoInput, portMAX_DELAY);

        if (status != pdPASS) {
            PRINTF("Queue Receive failed!.\r\n");

            while (1)
                ;
        }

        if (prevServoInput != servoInput) {
            prevServoInput = servoInput;
            servoDutyCycle = servoInput * 0.025f / 45.0f + 0.078;
            updatePWM_dutyCycle(FTM_CHANNEL_SERVO_MOTOR, servoDutyCycle);
            FTM_SetSoftwareTrigger(FTM_MOTOR, true);
        }

        vTaskDelay(1 / portTICK_PERIOD_MS);
    }
}
```

## LED Task

The LED task controls the three LEDs on the board by calculating the PWM duty cycle and controlling the FTM module. The task will wait for and receive messages from the LED queue, and will convert the values received on the queue to the appropriate PWM duty cycle for the FTM module. The LED queue can contain one integer pointer, which points to a three element integer array, and the pointer sent into the queue from the RC task will be to an integer array that contains three values from 0 to 255 which correspond to the RGB values of the onboard red, green and blue LEDs. When the LED task receives a pointer from the LED queue, it will convert the values in the array to their corresponding PWM duty cycles, and update the FTM module with this duty cycle, then trigger the FTM module.

The LED task is shown in Listing 3.

Listing 3: LED Task

```c
void ledTask(void *pvParameters) {
    // LED task implementation
    BaseType_t status;
    float red;
    float green;
    float blue;

    uint8_t led_input[3];

    while (1) {
        status = xQueueReceive(led_queue, (void*) &led_input, portMAX_DELAY);

        if (status != pdPASS) {
            PRINTF("Queue Receive failed!.\r\n");

            while (1)
                ;
        }

        red = (led_input[0] / 255.0) * 100;
        green = (led_input[1] / 255.0) * 100;
        blue = (led_input[2] / 255.0) * 100;

        FTM_UpdatePwmDutycycle(FTM_LED, FTM_RED_CHANNEL, kFTM_EdgeAlignedPwm, (uint8_t) red)
            ;
        FTM_SetSoftwareTrigger(FTM_LED, true);

        FTM_UpdatePwmDutycycle(FTM_LED, FTM_GREEN_CHANNEL, kFTM_EdgeAlignedPwm, (uint8_t)
            green);
        FTM_SetSoftwareTrigger(FTM_LED, true);

        FTM_UpdatePwmDutycycle(FTM_LED, FTM_BLUE_CHANNEL, kFTM_EdgeAlignedPwm, (uint8_t)
            blue);
        FTM_SetSoftwareTrigger(FTM_LED, true);

        vTaskDelay(1 / portTICK_PERIOD_MS);
    }
}
```

## RC Task

The RC task receives and decodes data coming from the radio receiver, which is connected via UART on the board. The task reads the bytes that are received from the controller through UART, then attempts to match the second byte of the header message (the second byte is used as the 2 byte numbers sent by the RC controller are in big endian). When the second byte of the header message matches, we read the remaining 17 bytes of the RC receiver data packet and try to match the entire header message. When the entire message matches, we will read the values on channels 8 (mapped to forward, stationary, and reverse motor directions), channels 6 (used to control the motor speed between slow, medium, and fast), as well as the joystick data values on channels 1 and 4 (used to adjust the motor speed and servo angle direction).

These channel data values will range from $1000 - 2000$. Channel 8 will set the motor speed to 0 when the channel is set to 1500, will allow the motor to operate in the forward direction when set to 1000, and will allow the motor to operate in the reverse direction when set to 2000. Channel 6 will allow the motor speed to operate between 0 and a maximum value dependent on the speed setting, as well as enabling the LEDs to operate at a certain colour corresponding to that speed setting. The motor will operate in the slow speed mode when channel 6 is set to 1000, allowing the speed to range from 0 to 20 (controlled by the joystick mapped to channel 1) and set the LED to appear red. The motor will operate in the medium speed mode when channel 6 is set to 1500, allowing the speed to range from 0 to 50 (controlled by the joystick mapped to channel 1) and set the LED to appear yellow. The motor will operate in the fast speed mode when channel 6 is set to 2000, allowing the speed to range from 0 to 1000 (controlled by the joystick mapped to channel 1) and set the LED to appear green. The angle value will range from -45 to 45 depending on the value of channel 4.

The motor speed values will always be mapped a value from -100 to 100, the servo angle values between -45 to 45, and the LED values between 0 to 255. These values are mapped to the expected input ranges for the motor, position, and LED tasks. The values are then sent to the motor, position, and LED queues respectively, where they will be processed by the appropriate tasks.

The RC task is shown in Listing 4.

Listing 4: RC Task

```
void rcTask(void *pvParameters) {
    // RC task implementation
    BaseType_t status;

    RC_Values rc_values;
    uint8_t *ptr = (uint8_t*) &rc_values;

    int motor_value;
    int angle_value;
    uint8_t led_value[3];

    while (1) {
        UART_ReadBlocking(RC_UART, ptr, 1);
        if (*ptr != 0x20)
            continue;
        UART_ReadBlocking(RC_UART, &ptr[1], sizeof(rc_values) - 1);
        if (rc_values.header == 0x4020) {

            if (rc_values.ch8 == 1500) {
                motor_value = 0;
            } else {
                switch (rc_values.ch6) {
                    case 1000:
                        motor_value = rc_values.ch8 == 1000 ? (rc_values.ch2 - 1000) / 50 :
                            (rc_values.ch2 - 1000) / -50;
                        led_value[0] = 255;
                        led_value[1] = 0;
                        led_value[2] = 0;
                        break;
                    case 1500:
                        motor_value = rc_values.ch8 == 1000 ? (rc_values.ch2 - 1000) / 20 :
                            (rc_values.ch2 - 1000) / -20;
                        led_value[0] = 255;
                        led_value[1] = 255;
                        led_value[2] = 0;
                        break;
                    case 2000:
                        motor_value = rc_values.ch8 == 1000 ? (rc_values.ch2 - 1000) / 10 :
                            (rc_values.ch2 - 1000) / -10;
                        led_value[0] = 0;
                        led_value[1] = 255;
                        led_value[2] = 0;
                        break;
                }
            }

            angle_value = (rc_values.ch4 - 1500) * 45 / 500;

            status = xQueueSend(motor_queue, (void* ) &motor_value, portMAX_DELAY);
            if (status != pdPASS) {
                PRINTF("Queue Send failed!.\r\n");
                while (1)
```

```c
                    ;
            }

            status = xQueueSend(angle_queue, (void* ) &angle_value, portMAX_DELAY);
            if (status != pdPASS) {
                PRINTF("Queue Send failed!.\r\n");
                while (1)
                    ;
            }

            status = xQueueSend(led_queue, (void* ) &led_value, portMAX_DELAY);
            if (status != pdPASS) {
                PRINTF("Queue Send failed!.\r\n");
                while (1)
                    ;
            }

            vTaskDelay(1 / portTICK_PERIOD_MS);
        }
    }
}
```