Project 1: Towards a More Realistic System Group 10 & Group 44

COMPENG 4DS4: Embedded Systems

Professor: Mohamed Hassan

Due: Wednesday March 13th, 2024

Name: Cameron Beneteau

Email: beneteac@mcmaster.ca

Student Number: 400274092

Name: Shaqif Ahmed

Email: ahmes80@mcmaster.ca

Student Number: 400263303

Name: Tyler Moss

Email: mosst2@mcmaster.ca

Student Number: 400274039

Name: Noah Betik

Email: betikn@mcmaster.ca

Student Number: 400246583

Declaration of Contributions

Tasks	Contributor(s)
Controlling the Car using The Controller	Cameron Beneteau, Shaqif Ahmed, Tyler Moss, and Noah Betik
LED Component	
Motor Component	
RC Component	
Motor Task	
Position Task	
LED Task	
RC Task	

GitHub Repository

https://github.com/COMPENG-4DS4-Winter2024/project1-group-10

https://github.com/COMPENG-4DS4-Winter2024/project1-group-44

Overview

This project enables full control over the car using the various components that were learned about and developed upon during the past three labs. Three main components were developed – the RC component (which handles the remote controller), the Motor component (which handles DC motor speed as well as servo angle to enable movement of the wheels), and the LED component (which enables visual indicators for the DC motor speed).

Controlling the Car using The Controller

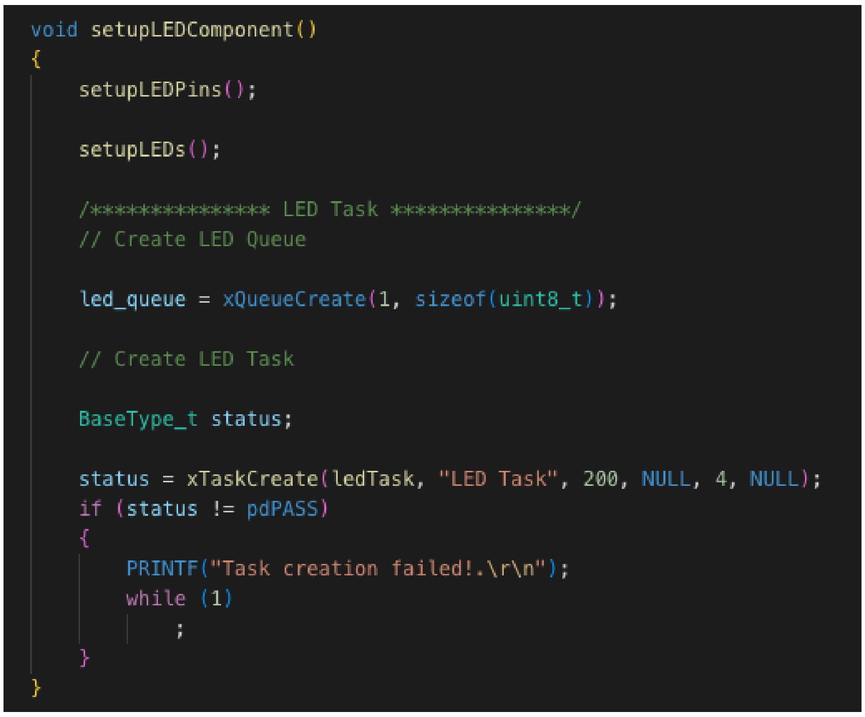
Functionality that corresponds to section 5.1 of the Project Document has been implemented. The speed of the DC motors on the vehicle can be controlled using the joysticks on the remote controller. The speed of the vehicle, as controlled by the user, has three modes: fastest (red), medium (yellow), and slowest (green). The LED on the vehicle changes depending on what speed mode the vehicle is currently operating in. When the switch on the controller is pressed, the vehicle alternates between forward and reverse motion. The front wheels of the vehicle can be turned by changing the angle of the servos through use of the remote controller.

LED Component

The LED component contains the LED task and the LED queue. The inputs to the queue are supplied from the RC task. The LED task function is discussed in its respective section.

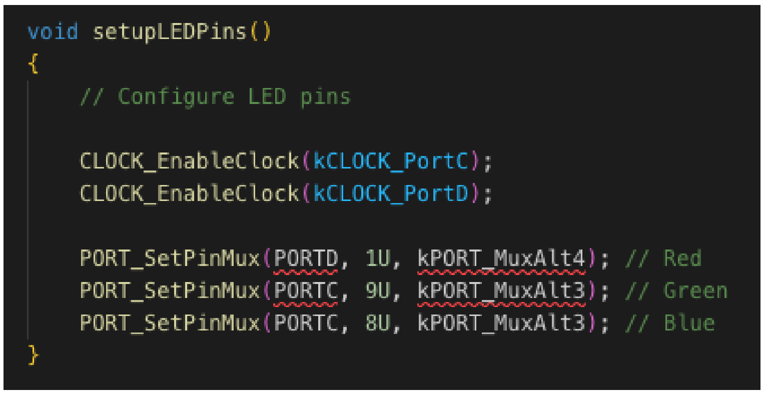
Alongside the LED task function, other functions are included to facilitate proper LED colour adjustment:

- setupLEDComponent()
    - Call setupLEDPins() and setupLEDs()
    - Define the led_queue with a size of 1 uint8_t, with the type QueueHandle_t
    - Create the LED task, set to the status variable of type BaseType_t



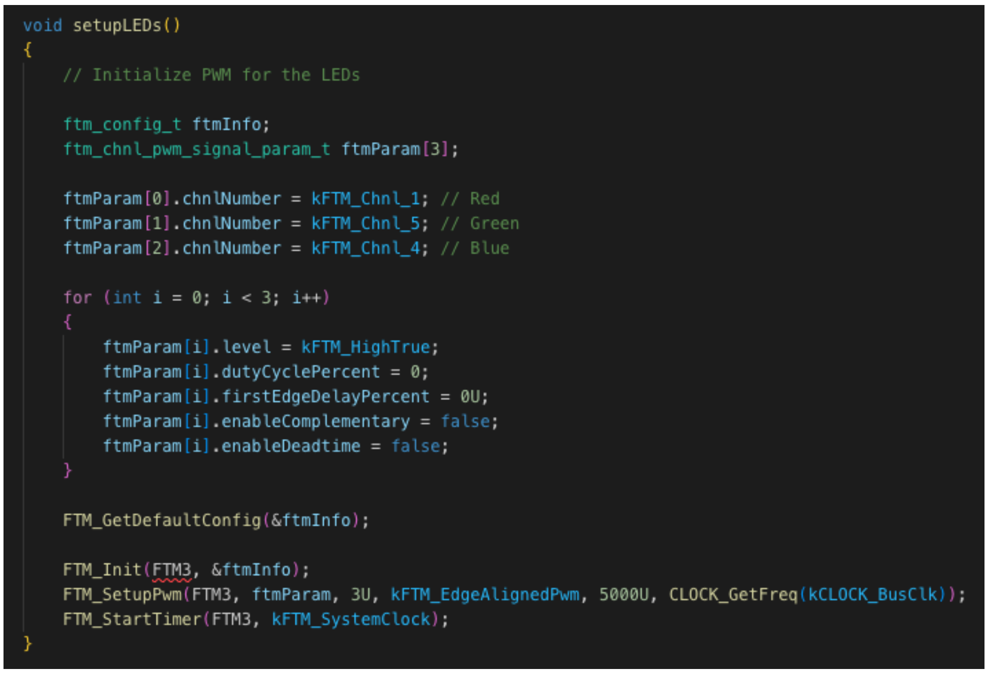
```
void setupLEDComponent()
{
    setupLEDPins();

    setupLEDs();

    /*************** LED Task ***************/
    // Create LED Queue

    led_queue = xQueueCreate(1, sizeof(uint8_t));

    // Create LED Task

    BaseType_t status;

    status = xTaskCreate(ledTask, "LED Task", 200, NULL, 4, NULL);
    if (status != pdPASS)
    {
        PRINTF("Task creation failed!.\r\n");
        while (1)
            ;
    }
}
```

- setupLEDPins()
    - Enable clocks for ports C and D
    - Set pin MUX for PTD1 (Red, MuxAlt4), PTC9 (Green, MuxAlt3) and PTC8 (Blue, MuxAlt3)



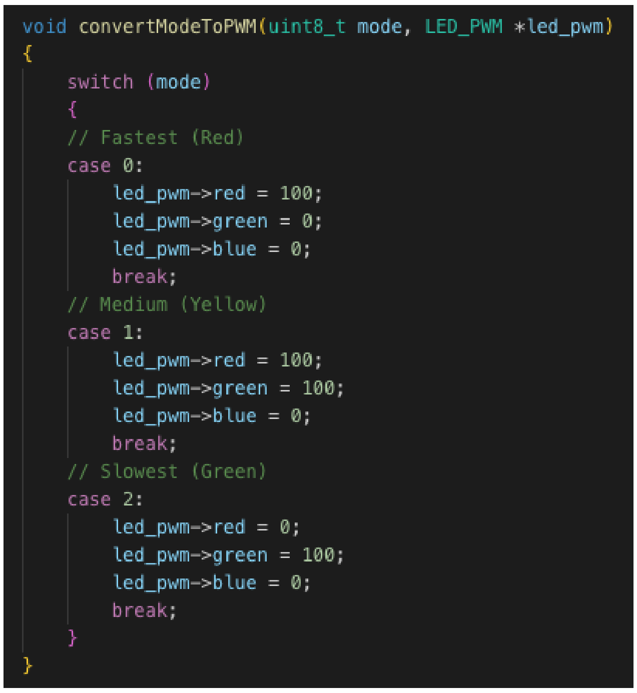
```
void setupLEDPins()
{
    // Configure LED pins

    CLOCK_EnableClock(kCLOCK_PortC);
    CLOCK_EnableClock(kCLOCK_PortD);

    PORT_SetPinMux(PORTD, 1U, kPORT_MuxAlt4); // Red
    PORT_SetPinMux(PORTC, 9U, kPORT_MuxAlt3); // Green
    PORT_SetPinMux(PORTC, 8U, kPORT_MuxAlt3); // Blue
}
```

- setupLEDs()
    - Initialize PWM for all three LEDs



```
void setupLEDs()
{
    // Initialize PWM for the LEDs

    ftm_config_t ftmInfo;
    ftm_chnl_pwm_signal_param_t ftmParam[3];

    ftmParam[0].chnlNumber = kFTM_Chnl_1; // Red
    ftmParam[1].chnlNumber = kFTM_Chnl_5; // Green
    ftmParam[2].chnlNumber = kFTM_Chnl_4; // Blue

    for (int i = 0; i < 3; i++)
    {
        ftmParam[i].level = kFTM_HighTrue;
        ftmParam[i].dutyCyclePercent = 0;
        ftmParam[i].firstEdgeDelayPercent = 0U;
        ftmParam[i].enableComplementary = false;
        ftmParam[i].enableDeadtime = false;
    }

    FTM_GetDefaultConfig(&ftmInfo);

    FTM_Init(FTM3, &ftmInfo);
    FTM_SetupPwm(FTM3, ftmParam, 3U, kFTM_EdgeAlignedPwm, 5000U, CLOCK_GetFreq(kCLOCK_BusClk));
    FTM_StartTimer(FTM3, kFTM_SystemClock);
}
```

- convertModeToPWM(uint8_t mode, LED_PWM *led_pwm)
    - If mode is equal to 0, this is our fastest speed. Set PWM to values that correspond with RED.
    - If mode is equal to 1, this is our medium speed. Set PWM to values that correspond with YELLOW.
    - If mode is equal to 2, this is our slowest speed. Set PWM to values that correspond with GREEN.

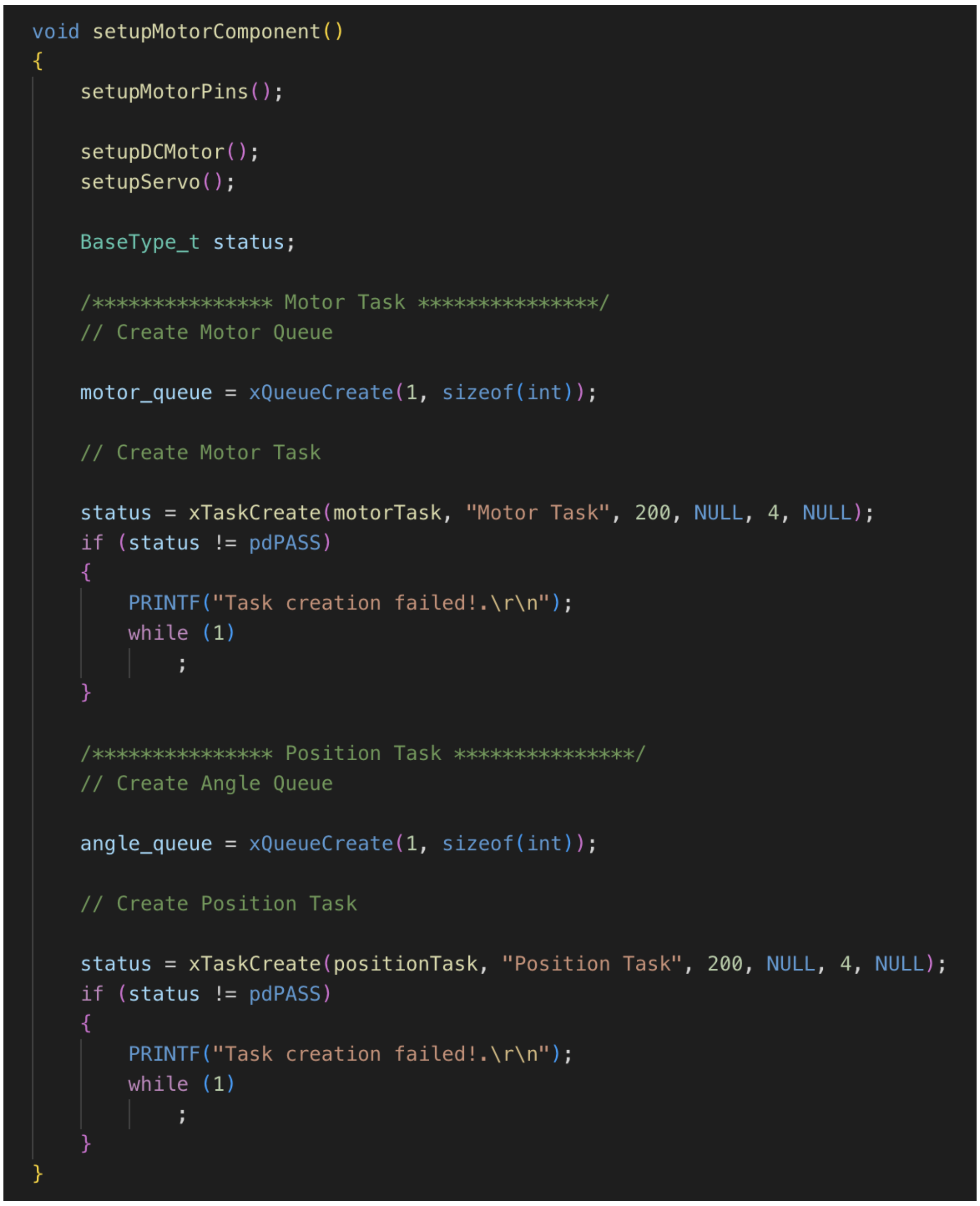


```
void convertModeToPWM(uint8_t mode, LED_PWM *led_pwm)
{
    switch (mode)
    {
    // Fastest (Red)
    case 0:
        led_pwm->red = 100;
        led_pwm->green = 0;
        led_pwm->blue = 0;
        break;
    // Medium (Yellow)
    case 1:
        led_pwm->red = 100;
        led_pwm->green = 100;
        led_pwm->blue = 0;
        break;
    // Slowest (Green)
    case 2:
        led_pwm->red = 0;
        led_pwm->green = 100;
        led_pwm->blue = 0;
        break;
    }
}
```

Motor Component

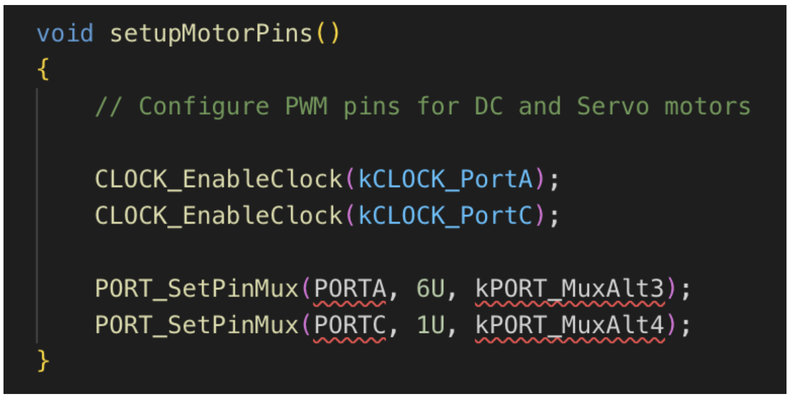
The motor component contains both the motor and position task, as well as the motor and position queue. The inputs to the queues are from the RC task (for our specific implementation). The motor and position task are discussed in their respective sections. Alongside the task functions, various other functions are included to facilitate proper execution within these task functions, such as:

- setupMotorComponent()
  - Calls setupMotorPins(), setupDCMotor(), and setupServo()
  - Defines the motor and position queues of type QueueHandle_t
  - Creates and defines the motor and position tasks of type BaseType_t



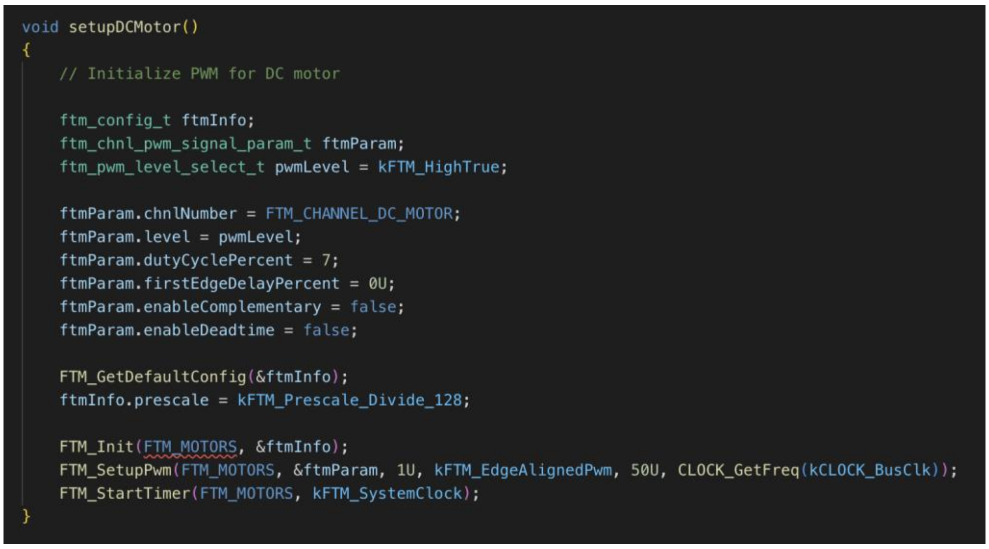
```
void setupMotorComponent()
{
    setupMotorPins();

    setupDCMotor();
    setupServo();

    BaseType_t status;

    /*************** Motor Task ***************/
    // Create Motor Queue

    motor_queue = xQueueCreate(1, sizeof(int));

    // Create Motor Task

    status = xTaskCreate(motorTask, "Motor Task", 200, NULL, 4, NULL);
    if (status != pdPASS)
    {
        PRINTF("Task creation failed!.\r\n");
        while (1)
            ;
    }

    /*************** Position Task ***************/
    // Create Angle Queue

    angle_queue = xQueueCreate(1, sizeof(int));

    // Create Position Task

    status = xTaskCreate(positionTask, "Position Task", 200, NULL, 4, NULL);
    if (status != pdPASS)
    {
        PRINTF("Task creation failed!.\r\n");
        while (1)
            ;
    }
}
```

- setupMotorPins()
  - Enables clocks for ports A and C
  - Sets pin MUX for PTA6 (MuxAlt3) and PTC1 (MuxAlt4)



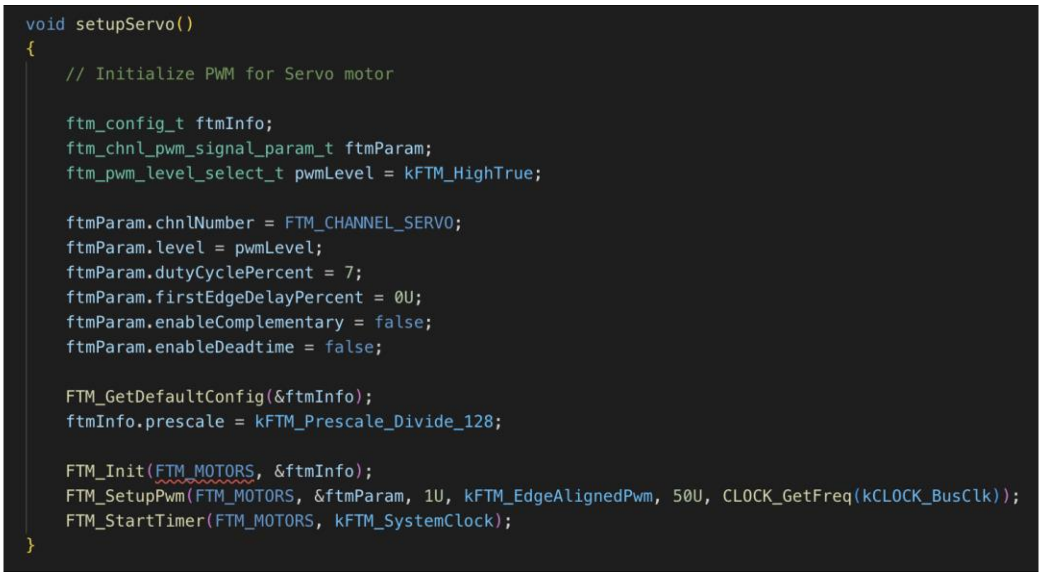
```
void setupMotorPins()
{
    // Configure PWM pins for DC and Servo motors

    CLOCK_EnableClock(kCLOCK_PortA);
    CLOCK_EnableClock(kCLOCK_PortC);

    PORT_SetPinMux(PORTA, 6U, kPORT_MuxAlt3);
    PORT_SetPinMux(PORTC, 1U, kPORT_MuxAlt4);
}
```

- setupDCMotor()
  - Initializes PWM for the DC motor



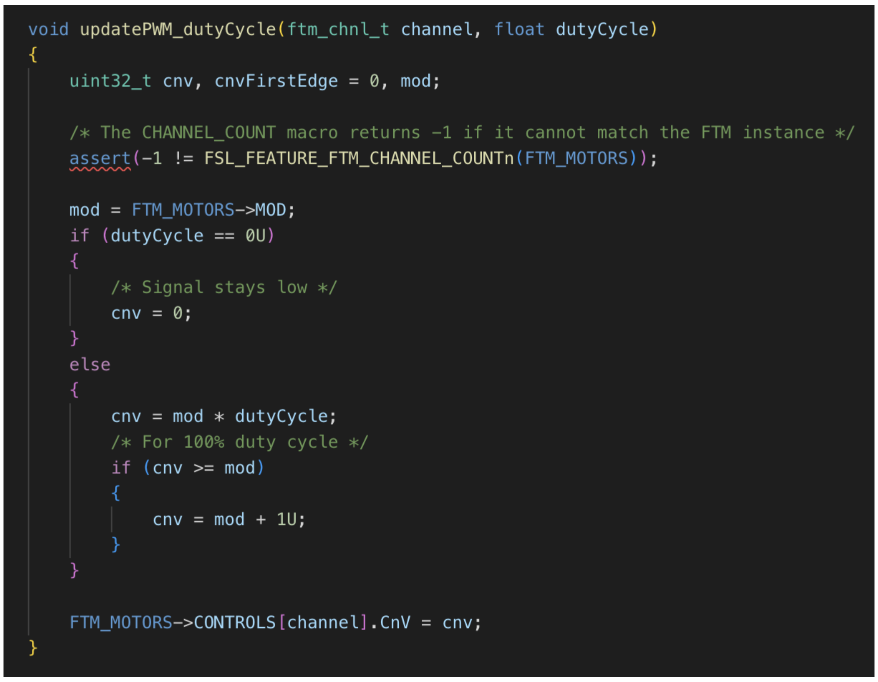
```
void setupDCMotor()
{
    // Initialize PWM for DC motor

    ftm_config_t ftmInfo;
    ftm_chnl_pwm_signal_param_t ftmParam;
    ftm_pwm_level_select_t pwmLevel = kFTM_HighTrue;

    ftmParam.chnlNumber = FTM_CHANNEL_DC_MOTOR;
    ftmParam.level = pwmLevel;
    ftmParam.dutyCyclePercent = 7;
    ftmParam.firstEdgeDelayPercent = 0U;
    ftmParam.enableComplementary = false;
    ftmParam.enableDeadtime = false;

    FTM_GetDefaultConfig(&ftmInfo);
    ftmInfo.prescale = kFTM_Prescale_Divide_128;

    FTM_Init(FTM_MOTORS, &ftmInfo);
    FTM_SetupPwm(FTM_MOTORS, &ftmParam, 1U, kFTM_EdgeAlignedPwm, 50U, CLOCK_GetFreq(kCLOCK_BusClk));
    FTM_StartTimer(FTM_MOTORS, kFTM_SystemClock);
}
```

- setupServo()
  - Initializes PWM for the Servo motor



```
void setupServo()
{
    // Initialize PWM for Servo motor

    ftm_config_t ftmInfo;
    ftm_chnl_pwm_signal_param_t ftmParam;
    ftm_pwm_level_select_t pwmLevel = kFTM_HighTrue;

    ftmParam.chnlNumber = FTM_CHANNEL_SERVO;
    ftmParam.level = pwmLevel;
    ftmParam.dutyCyclePercent = 7;
    ftmParam.firstEdgeDelayPercent = 0U;
    ftmParam.enableComplementary = false;
    ftmParam.enableDeadtime = false;

    FTM_GetDefaultConfig(&ftmInfo);
    ftmInfo.prescale = kFTM_Prescale_Divide_128;

    FTM_Init(FTM_MOTORS, &ftmInfo);
    FTM_SetupPwm(FTM_MOTORS, &ftmParam, 1U, kFTM_EdgeAlignedPwm, 50U, CLOCK_GetFreq(kCLOCK_BusClk));
    FTM_StartTimer(FTM_MOTORS, kFTM_SystemClock);
}
```

- updatePWM_dutyCycle(ftm_chnl_t channel, float dutyCycle)
    - Updates PWM for either the DC motor or servo

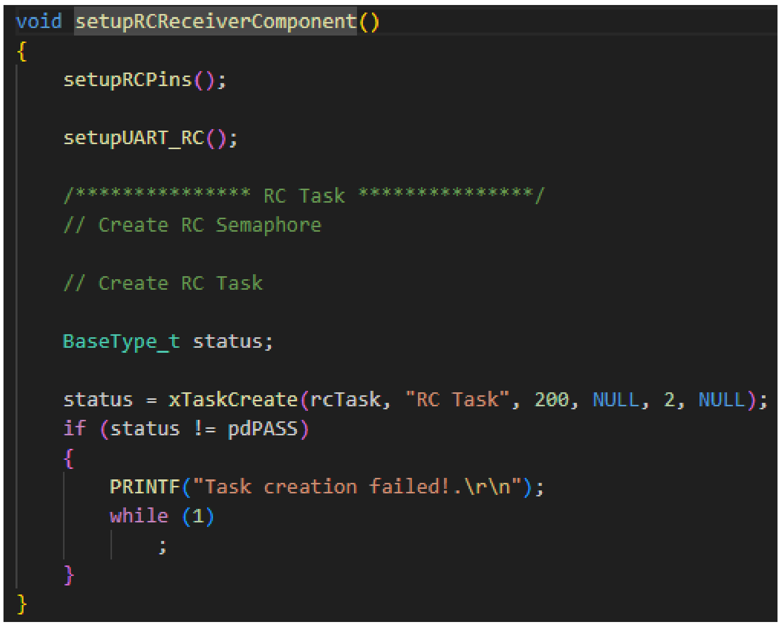


```c
void updatePWM_dutyCycle(ftm_chnl_t channel, float dutyCycle)
{
    uint32_t cnv, cnvFirstEdge = 0, mod;

    /* The CHANNEL_COUNT macro returns -1 if it cannot match the FTM instance */
    assert(-1 != FSL_FEATURE_FTM_CHANNEL_COUNTn(FTM_MOTORS));

    mod = FTM_MOTORS->MOD;
    if (dutyCycle == 0U)
    {
        /* Signal stays low */
        cnv = 0;
    }
    else
    {
        cnv = mod * dutyCycle;
        /* For 100% duty cycle */
        if (cnv >= mod)
        {
            cnv = mod + 1U;
        }
    }

    FTM_MOTORS->CONTROLS[channel].CnV = cnv;
}
```

RC Component

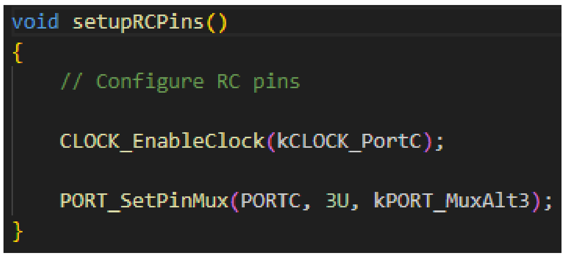
The RC component includes the RC task and the hold semaphore as an optional input from the (bonus) terminal component. In addition to the task function which handles the acquisition and redirection of controller input, other functions are implemented to facilitate proper execution for the main RC task.

- void setupRCReceiverComponent()
    - Calls setup functions for UART and required pins
    - Initializes the RC task



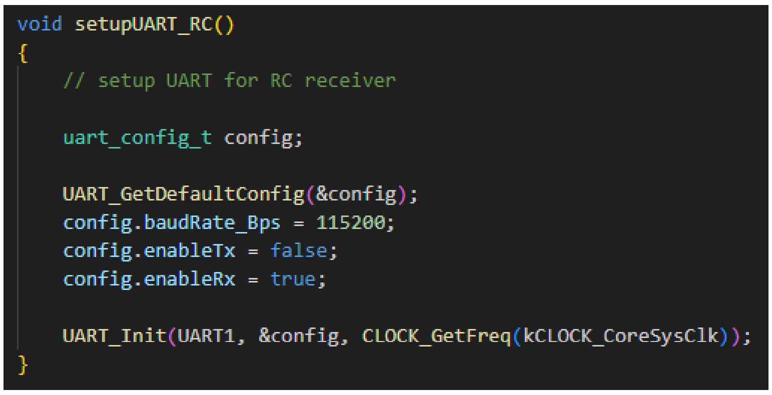
```
void setupRCReceiverComponent()
{
    setupRCPins();

    setupUART_RC();

    /*************** RC Task ***************/
    // Create RC Semaphore

    // Create RC Task

    BaseType_t status;

    status = xTaskCreate(rcTask, "RC Task", 200, NULL, 2, NULL);
    if (status != pdPASS)
    {
        PRINTF("Task creation failed!.\r\n");
        while (1)
            ;
    }
}
```

- void setupRCPins()
    - Initializes pin 3 on Port C with ALT3 functionality for receiving data over UART via the controller



```
void setupRCPins()
{
    // Configure RC pins

    CLOCK_EnableClock(kCLOCK_PortC);

    PORT_SetPinMux(PORTC, 3U, kPORT_MuxAlt3);
}
```

- void setupUART_RC()
    - Initializes UART communication (receiving only) with a baud rate of115200



```c
void setupUART_RC()
{
    // setup UART for RC receiver

    uart_config_t config;

    UART_GetDefaultConfig(&config);
    config.baudRate_Bps = 115200;
    config.enableTx = false;
    config.enableRx = true;

    UART_Init(UART1, &config, CLOCK_GetFreq(kCLOCK_CoreSysClk));
}
```

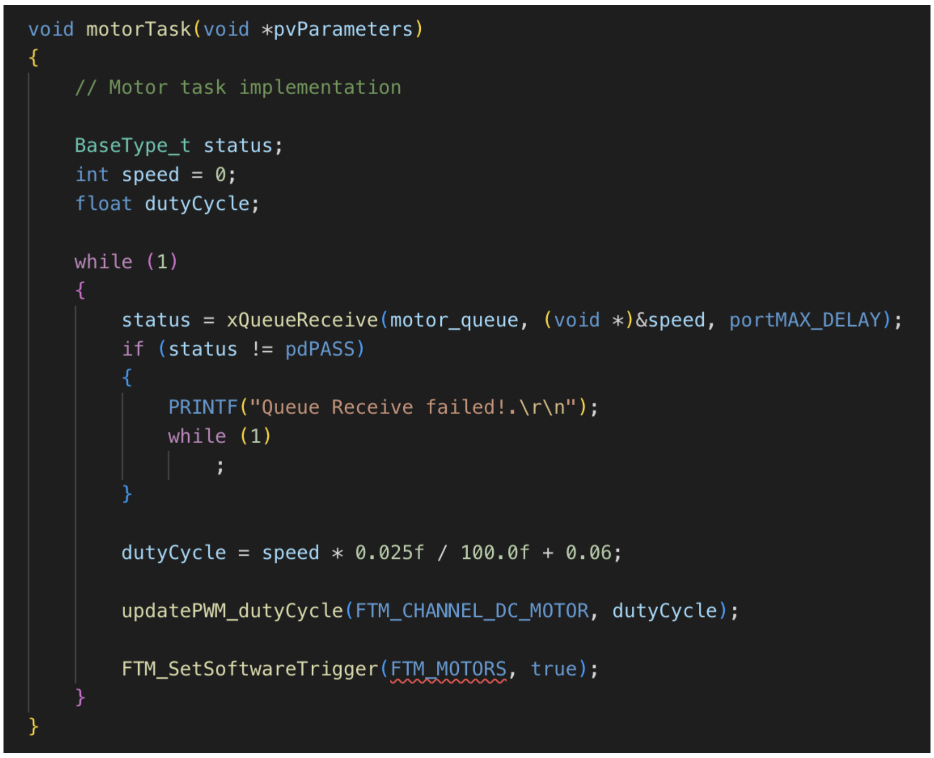
Motor Task

The motor task is responsible for updating the speed of the DC motor on the vehicle, which rotates the wheels, enabling forwards or reverse motion.

The core “motorTask” (in Motor_Control_Component.c) functionality is implemented as follows:

- A while loop to ensure real-time, consistent movement of the vehicle
    - Receiving the next “speed” value from “motor_queue”
    - Calculating the duty cycle with respect to the “speed” value
    - Updating the motor’s PWM duty cycle
    - Setting the software trigger for FTM_MOTORS

This task results in the motor adjusting its duty cycle to the new calculated duty cycle value, which corresponds to the new speed value provided from the queue.



```c
void motorTask(void *pvParameters)
{
    // Motor task implementation

    BaseType_t status;
    int speed = 0;
    float dutyCycle;

    while (1)
    {
        status = xQueueReceive(motor_queue, (void *)&speed, portMAX_DELAY);
        if (status != pdPASS)
        {
            PRINTF("Queue Receive failed!.\r\n");
            while (1)
                ;
        }

        dutyCycle = speed * 0.025f / 100.0f + 0.06;

        updatePWM_dutyCycle(FTM_CHANNEL_DC_MOTOR, dutyCycle);

        FTM_SetSoftwareTrigger(FTM_MOTORS, true);
    }
}
```

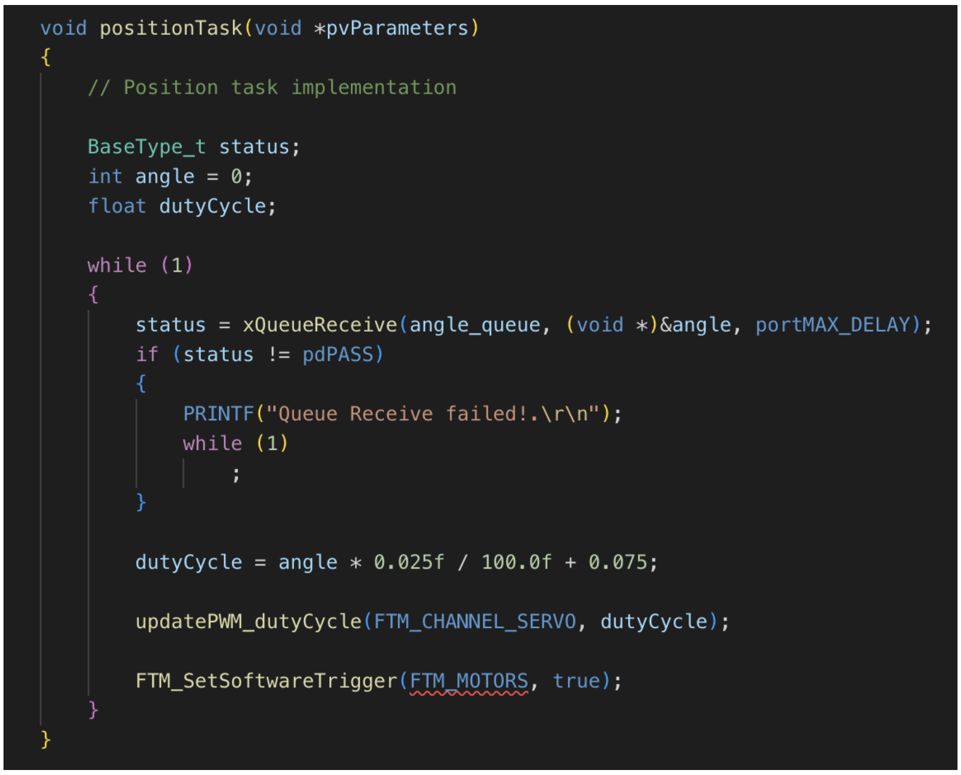
Position Task

The position task is responsible for updating the angle of the servo motor, allowing the vehicle to make turns.

The core “positionTask” (in Motor_Control_Component.c) functionality is implemented as follows:

- A while loop to ensure real-time, consistent adjustment of the servos
    - Receiving the new “angle” value from the angle_queue queue
    - Calculating the new duty cycle value with respect to the new angle value
    - Updating the servo’s PWM duty cycle
    - Setting the software trigger for FTM_MOTORS

This task results in the servo changing its angle dependent on the angle value supplied by the angle queue in real-time.



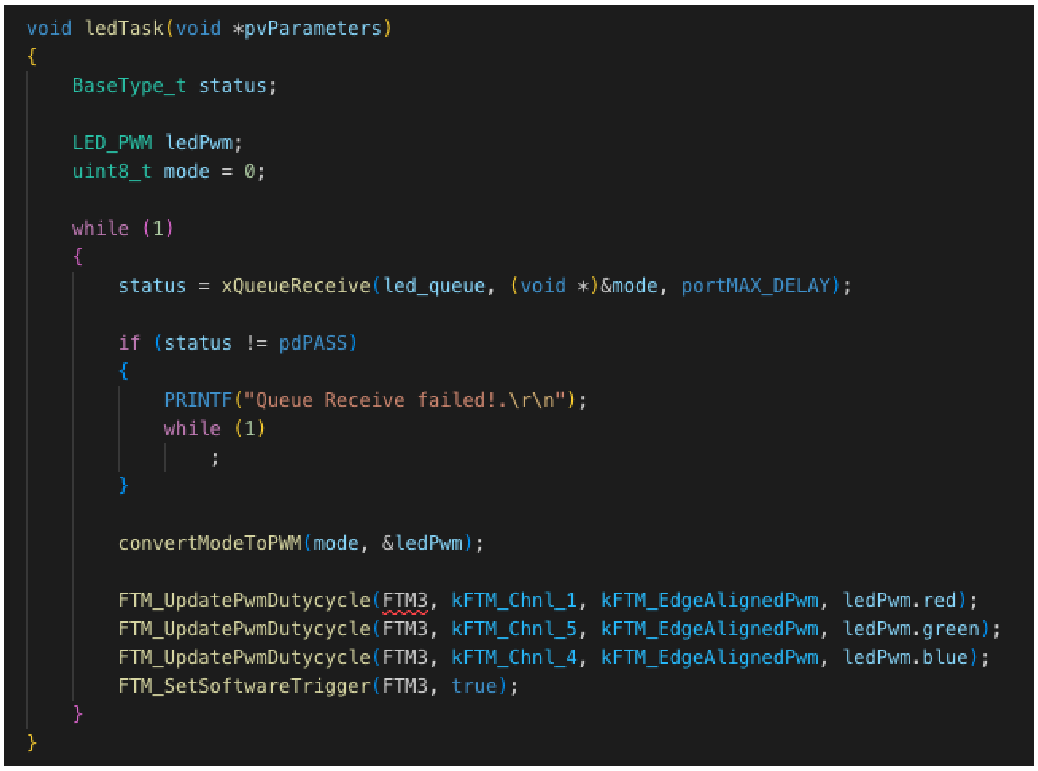
```c
void positionTask(void *pvParameters)
{
    // Position task implementation

    BaseType_t status;
    int angle = 0;
    float dutyCycle;

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

        dutyCycle = angle * 0.025f / 100.0f + 0.075;

        updatePWM_dutyCycle(FTM_CHANNEL_SERVO, dutyCycle);

        FTM_SetSoftwareTrigger(FTM_MOTORS, true);
    }
}
```

LED Task

The LED task is responsible for adjusting the colour of the LED onboard the vehicle according to the speed of the vehicle, in real-time. The core “ledTask” (in LED_Component.c) functionality is implemented as follows:

- Initialize the speed mode a default value of 0
- A while loop to ensure real-time, accurate switching between LED colours
    - Receiving the speed mode from the LED queue, supplied by the RC task
    - Converting the mode to the correct PWM values with convertModeToPWM()
    - Updating the PWM duty cycle for each LED
    - Setting the software trigger for FTM3

This task results in the LED changing to red for the fastest speed mode, yellow for the medium speed mode, and green for the slowest speed mode in real-time.

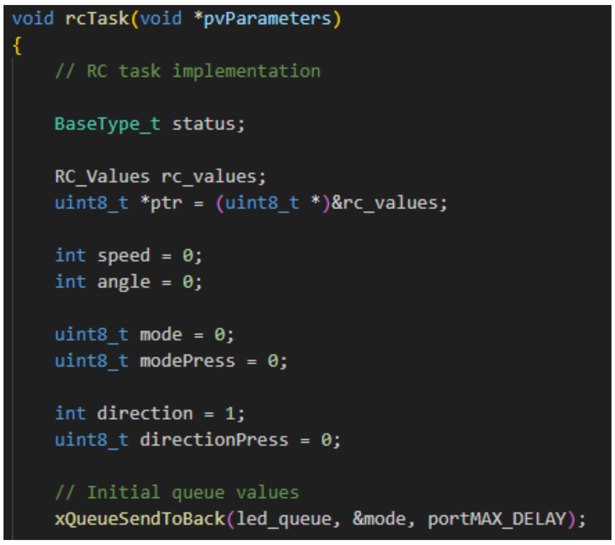


```c
void ledTask(void *pvParameters)
{
    BaseType_t status;

    LED_PWM ledPwm;
    uint8_t mode = 0;

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

        if (status != pdPASS)
        {
            PRINTF("Queue Receive failed!.\r\n");
            while (1)
                ;
        }

        convertModeToPWM(mode, &ledPwm);

        FTM_UpdatePwmDutycycle(FTM3, kFTM_Chnl_1, kFTM_EdgeAlignedPwm, ledPwm.red);
        FTM_UpdatePwmDutycycle(FTM3, kFTM_Chnl_5, kFTM_EdgeAlignedPwm, ledPwm.green);
        FTM_UpdatePwmDutycycle(FTM3, kFTM_Chnl_4, kFTM_EdgeAlignedPwm, ledPwm.blue);
        FTM_SetSoftwareTrigger(FTM3, true);
    }
}
```

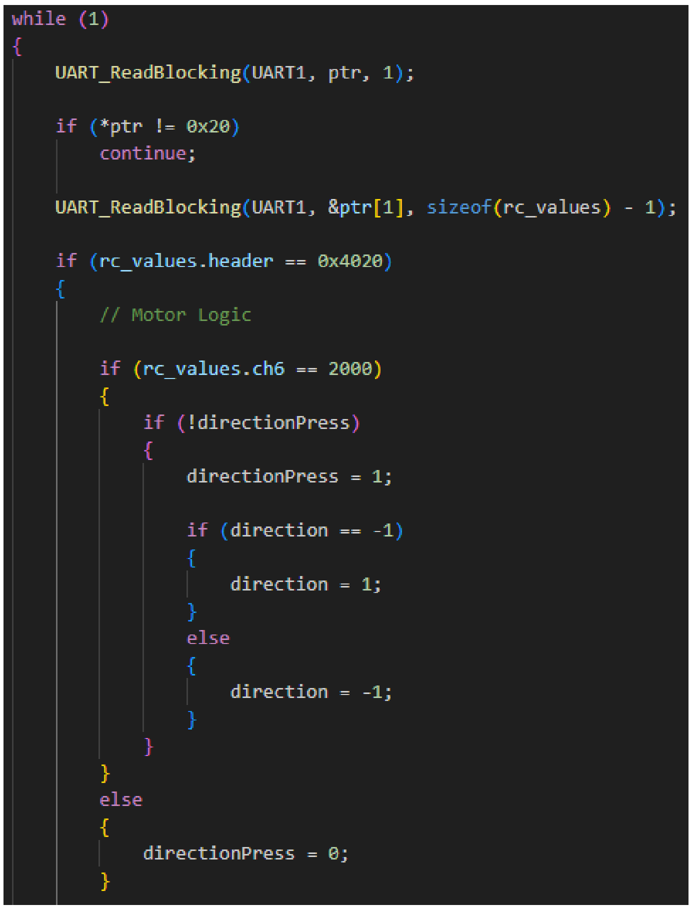
RC Task

The RC task is responsible for reading and decoding data from the controller in real-time. Isolating the channels for the joysticks and buttons & pushing their values to the associated queues controls the speed and direction of the DC motors, angle of the servos, and colour of the LEDs. This functionality is implemented as follows:

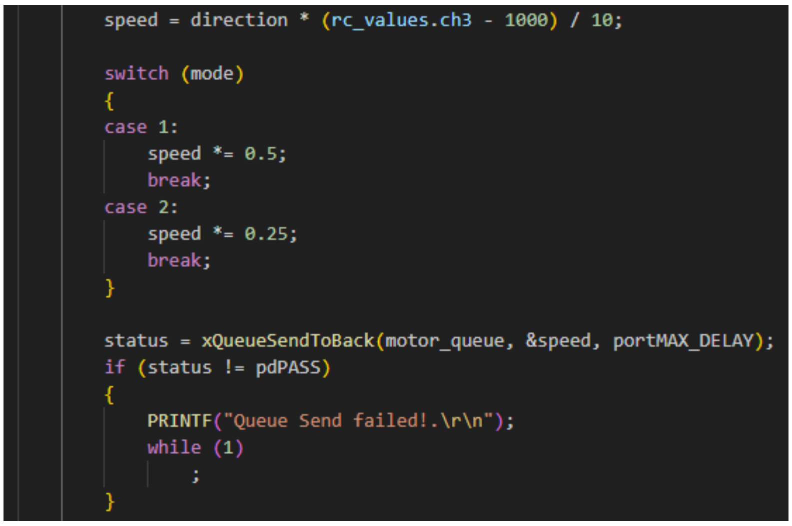
- Initialize the motor speed & servo angle as 0, initialize speed mode as 0
- A while loop to ensure real-time reception of data
    - Read data from all channels over the UART channel & place into predefined struct
    - Confirm proper data format by checking header value for correct value (0x4020)
    - Process channel 6 value for DC motor direction (setting a “direction” variable as +1 or –1 to change direction) & channel 3 value for DC motor speed. Multiply the received speed value by the direction & scaling factors based on the speed mode, then push to motor_queue for Motor Task to handle
    - Process channel 1 value for servo angle & push to the angle_queue for Position Task to handle
    - Process channel 5 value to increment mode if needed (if button is pressed and mode is 2, loop back to 0) and push result to led queue for LED Task to handle



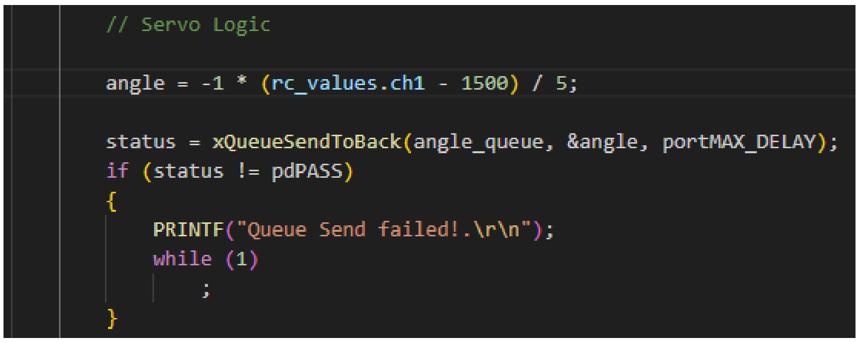
```
void rcTask(void *pvParameters)
{
    // RC task implementation

    BaseType_t status;

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

    int speed = 0;
    int angle = 0;

    uint8_t mode = 0;
    uint8_t modePress = 0;

    int direction = 1;
    uint8_t directionPress = 0;

    // Initial queue values
    xQueueSendToBack(led_queue, &mode, portMAX_DELAY);
```



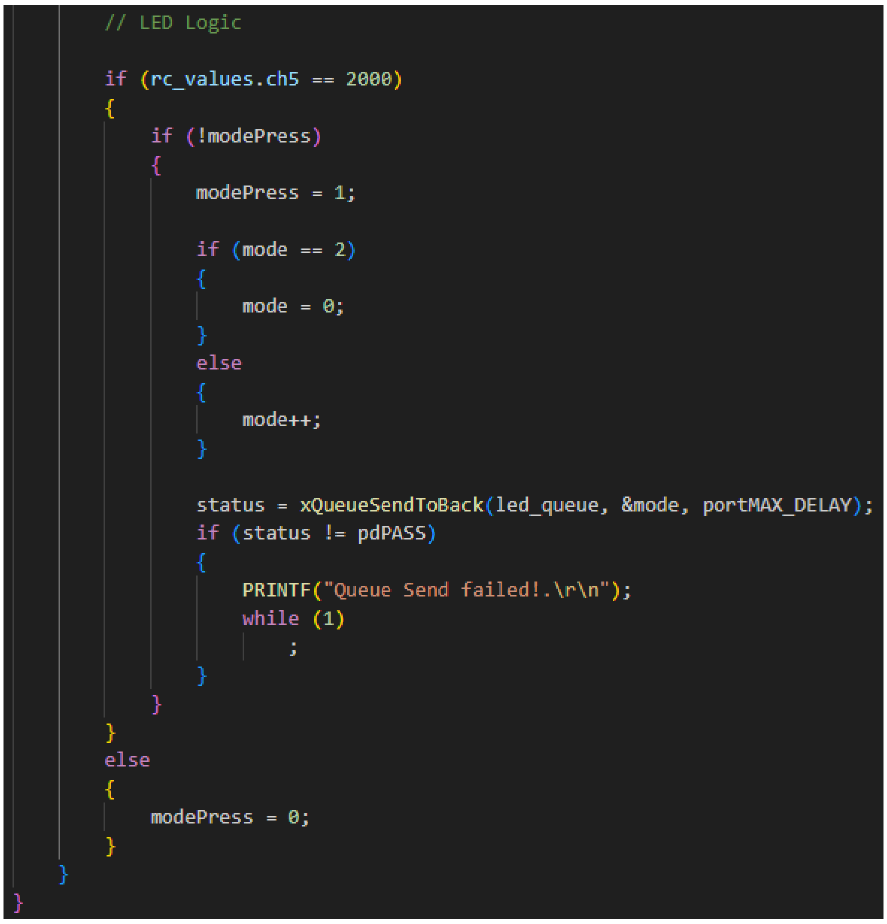
```c
while (1)
{
    UART_ReadBlocking(UART1, ptr, 1);

    if (*ptr != 0x20)
        continue;

    UART_ReadBlocking(UART1, &ptr[1], sizeof(rc_values) - 1);

    if (rc_values.header == 0x4020)
    {
        // Motor Logic

        if (rc_values.ch6 == 2000)
        {
            if (!directionPress)
            {
                directionPress = 1;

                if (direction == -1)
                {
                    direction = 1;
                }
                else
                {
                    direction = -1;
                }
            }
        }
        else
        {
            directionPress = 0;
        }
```



```c
        speed = direction * (rc_values.ch3 - 1000) / 10;

        switch (mode)
        {
        case 1:
            speed *= 0.5;
            break;
        case 2:
            speed *= 0.25;
            break;
        }

        status = xQueueSendToBack(motor_queue, &speed, portMAX_DELAY);
        if (status != pdPASS)
        {
            PRINTF("Queue Send failed!.\r\n");
            while (1)
                ;
        }
```



```c
        // Servo Logic

        angle = -1 * (rc_values.ch1 - 1500) / 5;

        status = xQueueSendToBack(angle_queue, &angle, portMAX_DELAY);
        if (status != pdPASS)
        {
            PRINTF("Queue Send failed!.\r\n");
            while (1)
                ;
        }
```



```
        // LED Logic

        if (rc_values.ch5 == 2000)
        {
            if (!modePress)
            {
                modePress = 1;

                if (mode == 2)
                {
                    mode = 0;
                }
                else
                {
                    mode++;
                }

                status = xQueueSendToBack(led_queue, &mode, portMAX_DELAY);
                if (status != pdPASS)
                {
                    PRINTF("Queue Send failed!.\r\n");
                    while (1)
                        ;
                }
            }
        }
        else
        {
            modePress = 0;
        }
    }
}
```