1. Introduction

The objective of this project was to design, implement, and test an autonomous robotic system capable of navigating through a predefined course with various navigational challenges. Using the Tiva C Series TM4C123GH6PM microcontroller as the primary computational unit, the robot was equipped with an array of sensors and actuators to interact with its environment effectively.

The challenges within the course required the robot to make decisions based on sensor input, such as maintaining a constant distance from a wall using a PID control algorithm, detecting and reacting to black crosslines of varying thickness, and stopping at a designated endpoint. The successful completion of these tasks necessitated the integration of various electronic components and the development of robust software to process sensor data, execute control algorithms, and manage actuator outputs.

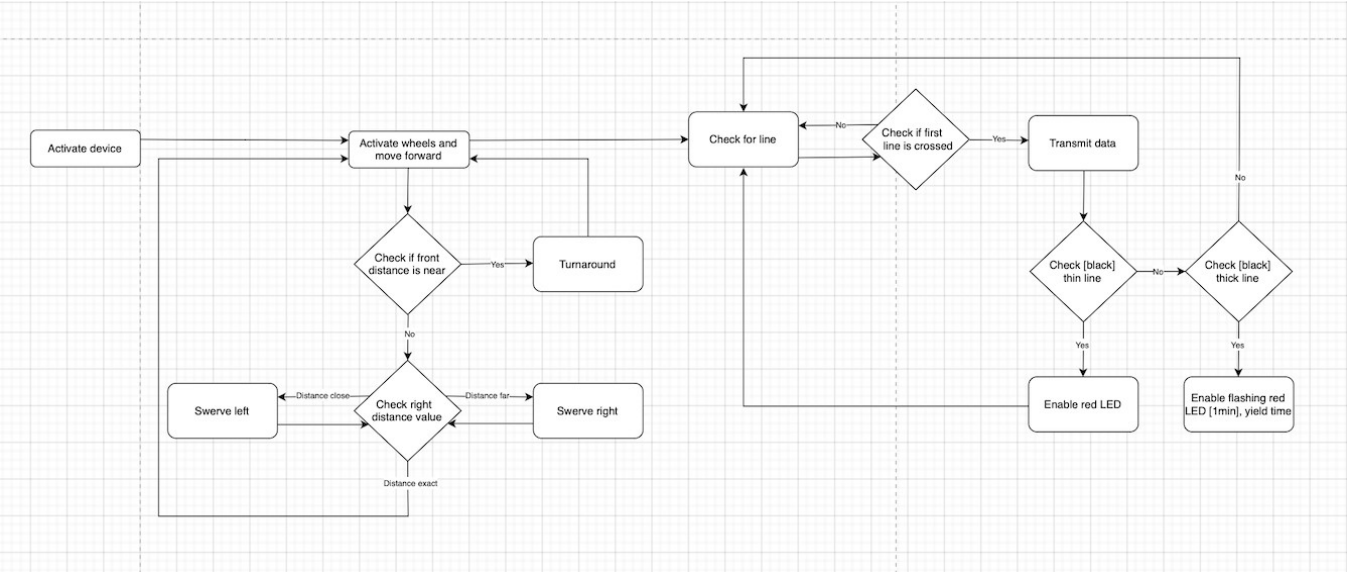
Throughout the project, we utilized the TI-RTOS operating system to structure our software as a collection of tasks and interrupt service routines (ISRs), providing a real-time response to sensor inputs and control outputs. The use of semaphores and timers allowed for synchronization and timing of tasks, ensuring data collection and motor control were performed at specific intervals. The modular approach to coding, with separate functions for each task, facilitated code maintenance and debugging.

A significant part of the project involved data acquisition and transmission, where error values from the PID control were collected every 100 ms, stored temporarily in local buffers, and then written as a block over the communication link to a PC for analysis. The implementation of "ping-pong" buffers ensured a continuous data collection process without loss of data, even while transmitting the collected data to the PC.

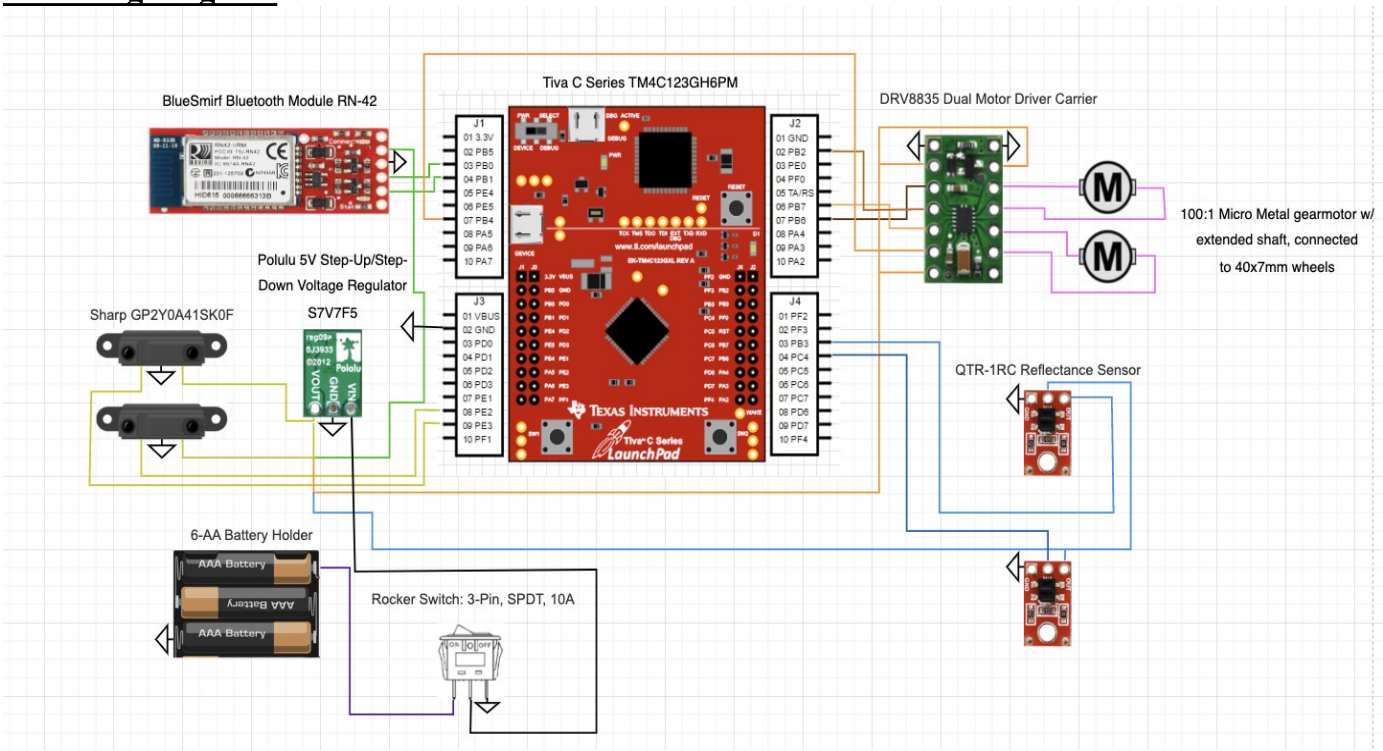
In addition to navigational control, the robot was designed to provide visual feedback through onboard LEDs that indicated its status and reactions to particular events, such as detecting a line or approaching a wall too closely. Furthermore, the project included safety features, such as the ability to perform a controlled stop upon detecting a "stop line" represented by a thick black line on the course.

This report presents a comprehensive account of the project's development process, from initial design considerations to the final implementation and testing phases. It details the technical aspects of the system, including hardware configurations, software algorithms, and the challenges overcome to achieve a fully functional autonomous robot. The subsequent sections of this report will delve into the specifics of each milestone achieved, providing insight into the methodologies applied and the rationale behind design choices.

2. Block Diagram



3. Wiring diagram



4. Pseudo code

// Constants
Declare Constants: SENSOR_MAX_VALUE, PWM_RANGE, ADC_THRESHOLD, etc.

// Global Variables
Declare Global Variables: sensorData, pidSettings, motorSpeeds, controlStates, bufferArrays, etc.

// Function declarations
sendViaUART(const char *buffer, uint32_t size)
handleUARTInterrupt()
controlMotorSpeed()
configureSensorReading()
driveForward(), driveBackward(), turnLeft(), turnRight(), haltRobot()

Main Function

main():
    Configure system clock and enable required peripherals (GPIO, UART, ADC, PWM, Timer)
    Set up UART for robot communication
    Prepare ADC for obtaining sensor data
    Initialize timers and interrupts for line detection
    Set up arrays, flags, and control variables
    Activate UART interrupt for command processing
    Activate Timer interrupts for line detection routine
    Infinite Loop:
    Run BIOS_start()

Interrupt Handlers and Control Tasks

// Timer Interrupt Handlers
timer0InterruptHandler():
    Signal semaphore0

timer1InterruptHandler():
    Perform line detection using sensors
    Adjust LED indications based on line detection
    Manage robot behavior according to line type detected

// UART Interrupt Handler
handleUARTInterrupt():
    Acquire commands via UART
    Execute movements: forward, backward, left, right, stop based on received commands

// PID Control Task
performPIDControl():
    Infinite Loop:
    Await semaphore (Semaphore_pend)
    Activate ADC to gather sensor data
    Compute error from setpoint and sensor data
    Calculate PID output (Proportional, Integral, Derivative)
    Apply PID output to motor control via PWM
    Log error data for analysis

// Auxiliary Functions
delayMilliseconds(ms), delayMicroseconds(us): Timing functions
logData(): Output buffer array data for analysis

// Initialization Routines
setupADC(): Prepare ADC modules for sensor data collection
setupPWM(): Configure PWM for motor control
sendDataUART(const char *buffer, uint32_t size): Transmit data over UART
movementCommands(): driveForward(), driveBackward(), turnLeft(), turnRight(), haltRobot()

5. Real Code with comments

```
extern const ti_sysbios_knl_Semaphore_Handle pidSemaphore;
uint32_t LeftDutyCycle = 100;
uint32_t RightDutyCycle = 100;
// Global variables for PID control
float integral = 0;  //module 6
float previousError = 0;  //module 6
int onVal = 0;
//Milestone 9
#define BUFFER_SIZE 20
int pingBuffer[BUFFER_SIZE];
int pongBuffer[BUFFER_SIZE];
int *currentBuffer = pingBuffer;
int *nextBuffer = pongBuffer;
int bufferIndex = 0;
```

```
int pidCounter = 0;  // Add this counter
int collectData = 0;  // Global variable to control data collection. 0 = don't collect, 1 = collect
#ifdef DEBUG
void
__error__(char *pcFilename, uint32_t ui32Line)
{
}
#endif
// Configure the UART and its pins.  This must be called before UARTprintf().
volatile uint32_t decayCounter = 0;
//functions predefined on top so that anyone can call each function
uint32_t readReflectanceSensor();
void ConfigureTimer40msB();
void handler50ms() ;
void basicPIDControl();
void deadEnd();
void adjustMotorOutput(float output);
void transmitData();
void swapBuffers();
void acquireData(float error);
uint32_t adc_rd(void){
   uint32_t adc0Val[8];
   ADCIntClear(ADC0_BASE, 1);
   ADCProcessorTrigger(ADC0_BASE, 1);
   while(!ADCIntStatus(ADC0_BASE, 1, false)) {};
   ADCSequenceDataGet(ADC0_BASE, 1, adc0Val);
   return adc0Val[0];
}
uint32_t adc_fwd(void)
{
   uint32_t adc0Val[8];
   ADCIntClear(ADC0_BASE, 1);
   ADCProcessorTrigger(ADC0_BASE, 1);
   while (!ADCIntStatus(ADC0_BASE, 1, false))
   {
   };
   ADCSequenceDataGet(ADC0_BASE, 1, adc0Val);
   return adc0Val[2];
}
void deadEnd(void)
{
   while (1)
   {
      Semaphore_pend(pidSemaphore, BIOS_WAIT_FOREVER);
      int frontDistance = (int)adc_fwd();
      if(onVal == 1) {
         // Green LED logic: If the robot is far enough from the wall (> 10cm, or ADC reading above
1000)
         if (frontDistance <= 1000)
         {
```

```
            //GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3 | GPIO_PIN_2 | GPIO_PIN_1, 0x0); // Turn off all LEDs
            //GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, GPIO_PIN_3);                  // Green LED on
        }
        // Yellow LED logic: If the robot is a little close to the wall (< 10cm, ADC reading between 1000 and 3000)
        else if (frontDistance > 1000 && frontDistance < 2300)
        {
            //GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3 | GPIO_PIN_2 | GPIO_PIN_1, 0x0); // Turn off all LEDs
            //GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3 | GPIO_PIN_1, GPIO_PIN_3 | GPIO_PIN_1); // Yellow (Green + Red) LED on
        }
        // Red LED logic + U-turn: If the robot is too close to the wall (< 7cm, or ADC reading less than or equal to 3000)
        else if (frontDistance >= 2300)
        {
            // Turn off all LEDs and turn on the red LED
            //GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3 | GPIO_PIN_2 | GPIO_PIN_1, 0x0);
            //GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_PIN_1);
            // Stop current movement
            PWMOutputState(PWM0_BASE, PWM_OUT_0_BIT, false);
            PWMOutputState(PWM0_BASE, PWM_OUT_1_BIT, false);
            // Perform U-turn
            PWMPulseWidthSet(PWM0_BASE, PWM_OUT_0, pwmMax);  // u- turn left
            PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, pwmMax);
            GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0); // Assuming this controls the direction of one motor
            GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0); // And this controls the direction of the other motor
            // Ensure PWM outputs are enabled
            PWMOutputState(PWM0_BASE, PWM_OUT_0_BIT, true);
            PWMOutputState(PWM0_BASE, PWM_OUT_1_BIT, true);
            // Loop to ensure robot keeps moving until it has cleared the obstacle
            while(adc_fwd() > 1000)
            {
                SysCtlDelay(SysCtlClockGet() / 3 / 1000 * 10); // Delay for 10 milliseconds
            }
            // Restore normal motion. Create a global var set as 0 then in str bluetooth make it one and have the if statement for the lines below
                LeftDutyCycle = 100;
                RightDutyCycle = 100;
                PWMPulseWidthSet(PWM0_BASE, PWM_OUT_0, LeftDutyCycle * pwmMax / 100);
                PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, RightDutyCycle * pwmMax / 100);
                GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, GPIO_PIN_2);
                GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0x0);
        }
    }
  }
}
```

```
void adjustMotorOutput(float output){
   // Define constants
   if(output > 0.99) {
      output = 0.99;
   }
   else if(output < -0.99) {
      output = -0.99;
   }
   if(output < 0) {
      PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, (1 + output)*LeftDutyCycle * pwmMax /
100);
      PWMPulseWidthSet(PWM0_BASE, PWM_OUT_0, RightDutyCycle * pwmMax / 100);
   }
   else if(output >= 0) {
   PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, RightDutyCycle * pwmMax / 100);
   PWMPulseWidthSet(PWM0_BASE, PWM_OUT_0, (1 - output)*LeftDutyCycle * pwmMax / 100);
   }
}
//mile stone 8
// Enumeration for the type of lines
typedef enum {
   NO_LINE,
   FIRST_THIN_LINE,
   SECOND_THIN_LINE,
   THICK_BLACK_LINE
} LineType;
static LineType last_detected_line = NO_LINE;
uint32_t readReflectanceSensor(void) {
   // Reset the decay counter
   decayCounter = 0;
   // Start the 40ms timer for WTimer1B
   TimerDisable(WTIMER1_BASE, TIMER_B);
   // Configure the pin as output and set it to high (charge the capacitor)
   GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_3);
   GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, GPIO_PIN_3);
   // Wait at least 10 Micros
   SysCtlDelay(SysCtlClockGet() / (3 * 100000));
   // Configure the pin as input (high impedance)
   GPIOPinTypeGPIOInput(GPIO_PORTB_BASE, GPIO_PIN_3);
   // Wait for the I/O line to go low
   while(GPIOPinRead(GPIO_PORTB_BASE, GPIO_PIN_3) & GPIO_PIN_3) {
      // The handler40ms will keep increasing the decayCounter every 40ms
      // Do nothing here, just wait for the line to go low
      decayCounter++;
   }
   // Stop the 40ms timer for WTimer1B
   uint32_t ui32Period = SysCtlClockGet() / 1000 * 40 - 1;  // 40ms period
   // Set the load value for Timer1B
   TimerLoadSet(WTIMER1_BASE, TIMER_B, ui32Period - 1);
   TimerEnable(WTIMER1_BASE, TIMER_B);
   return decayCounter;  // This will return how many 40ms ticks occurred before the line went low
```

```
}
//milestone 10
void FlashRedLEDForOneMinute(void) {
    uint32_t halfSecond = SysCtlClockGet() / 2;  // Calculate delay for half a second at 50 MHz
    // Assuming that your SysCtlDelay() function delays for 3 clock cycles,
    // the delay count should be adjusted accordingly:
    uint32_t delayCount = halfSecond / 3;
    int i;
    int j;
    for (i = 0; i < 60; i++) {  // 60 seconds
        for (j = 0; j < 2; j++) {  // Blink twice per second
            // Toggle LED
            GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, GPIOPinRead(GPIO_PORTF_BASE,
GPIO_PIN_1) ^ GPIO_PIN_1);
            // Delay for half a second
            SysCtlDelay(delayCount);
        }
    }
    // Turn off the LED after blinking
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0);
}
static uint32_t startTime = 0;
static uint32_t stopTime = 0;
void startTimer() {
    // Get the current clock count
    startTime = Clock_getTicks();
}
void stopTimerAndSendData() {
    // Get the current clock count
    stopTime = Clock_getTicks();
    // Calculate the elapsed time in milliseconds
    uint32_t elapsedTime = ((stopTime - startTime) * Clock_tickPeriod / 1000)*2; // Convert to
milliseconds
    // Calculate the elapsed time in seconds and then in hundredths of a second
    uint32_t timeInHundredths = (uint32_t)(elapsedTime * 100 / 1000); // Convert milliseconds to
hundredths of seconds
    // Extract seconds and hundredths for separate printing
    uint32_t seconds = timeInHundredths / 100;
    uint32_t hundredths = timeInHundredths % 100;
    // Send the elapsed time to the PC with two decimal places
    UARTprintf("Elapsed time: %u.%02u seconds\n", seconds, hundredths);
}
void handleLineDetection(void) {
    uint32_t cycles = readReflectanceSensor();
    LineType detected_line;
    if(onVal == 1) {
        // Based on threshold
        if(cycles > 3000) {
            // Check again for thick line
            SysCtlDelay(SysCtlClockGet() / 33);  // 160ms delay
            cycles = readReflectanceSensor();
```

```
            if(cycles > 3000) {
                // It's a thick line, flash the red led for a minute and then close the bios after that.
                //shut the bot down
                PWMOutputState(PWM0_BASE, PWM_OUT_0_BIT, false);
                PWMOutputState(PWM0_BASE, PWM_OUT_1_BIT, false);
                stopTimerAndSendData();
                FlashRedLEDForOneMinute();
                // Ensure the LED is turned off before stopping BIOS
                GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3 | GPIO_PIN_2 | GPIO_PIN_1, 0x0);
                // Stop the BIOS
                BIOS_exit(0);  // The argument can be any integer, typically used to indicate the reason for
shutdown.
                detected_line = THICK_BLACK_LINE;  // Update the detected line
            } else {
                // It's a thin line, determine which one based on history
                if(last_detected_line == NO_LINE) {
                    detected_line = FIRST_THIN_LINE;
                    //milestone 9
                    collectData = 1;  // Start data collection
                    startTimer();
                    // Directly enable Green LED
                    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3 | GPIO_PIN_2 | GPIO_PIN_1, 0x0);
                    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, GPIO_PIN_3);
                } else {
                    detected_line = SECOND_THIN_LINE;
                    //milestone 9
                    collectData = 0;  // Stop data collection
                    //milestone 10
                    //Turn on the RED LED
                    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3 | GPIO_PIN_2 | GPIO_PIN_1, 0x0);
                    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_PIN_1);
                }
            }
        } else {
            // If no line detected, just assign the last detected line state
            detected_line = last_detected_line;
        }
        // Update the last detected line state
        last_detected_line = detected_line;
    }
}
//milestone 9
void transmitData() {
    int i;
    // Send data from the full buffer (nextBuffer)
    for (i = 0; i < BUFFER_SIZE; i++) {
        UARTprintf("%i ", nextBuffer[i]);  // Sends each value followed by a space
    }
    // Send Carriage Return and Linefeed to signify end of frame
    UARTprintf("\n");  // Sends each value followed by a space
}
```

```
void swapBuffers() {
   if (currentBuffer == pingBuffer) {
      //Turn on the blue LED when it enter the Ping part of it
      GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3 | GPIO_PIN_2 | GPIO_PIN_1, 0x0);
      GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);
      currentBuffer = pongBuffer;
      nextBuffer = pingBuffer;
   } else {
      //Turn on the yellow LED when it enter the Pong part of it
      GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3 | GPIO_PIN_2 | GPIO_PIN_1, 0x0);
      GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_PIN_1);
      GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, GPIO_PIN_3);
      currentBuffer = pingBuffer;
      nextBuffer = pongBuffer;
   }
}
void acquireData(float error) {
   if (bufferIndex < BUFFER_SIZE) {
      currentBuffer[bufferIndex++] = (int)(error * 100);  // Multiply by 100 to convert to percentage
   } else {
      bufferIndex = 0;
      swapBuffers();  // Switches between ping and pong
      transmitData(); // Sends data to PC
   }
}
//Milestone 9 above
void basicPIDControl(void){
   while(1) {
      Semaphore_pend(pidSemaphore, BIOS_WAIT_FOREVER);

      int desiredPosition = 1800;  // Hypothetical desired position (setpoint)
      int currentPosition = (int)adc_rd();  // Measured position (process variable)

      // Tune your PID constants (Kp, Ki, Kd) as per your requirement.
      float Kp = 1.5, Ki = 0, Kd = 0;

      float output ;
      float error = ((float)desiredPosition - currentPosition) / desiredPosition ;  // Calculate error percent
in decimal
      output = Kp*error ;  // PID output

      previousError = error;

      adjustMotorOutput(output);

      //milestone 9
      if(collectData == 1) {
         pidCounter++;
         if (pidCounter == 1) {  // Only acquire data every second call
            pidCounter = 0;
            acquireData(error);  // Call data acquisition logic
```

```
            }
        }
        int j;
        if (last_detected_line == SECOND_THIN_LINE) {
            // Transmit remaining data in currentBuffer up to bufferIndex
            for (j = 0; j < bufferIndex; j++) {
                UARTprintf("%i ", currentBuffer[j]);
            }
            UARTprintf("\r\n");  // End the transmission

            bufferIndex = 0;  // Empty the buffer
        }


    }


}
void
ConfigureUART(void)
{
    // Enable the GPIO Peripheral used by the UART.
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB); //milestone2
    // Enable UART0
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART1); //milestone2
    // Configure GPIO Pins for UART mode.
    GPIOPinConfigure(GPIO_PA0_U0RX);
    GPIOPinConfigure(GPIO_PB0_U1RX); //milestone2
    GPIOPinConfigure(GPIO_PA1_U0TX);
    GPIOPinConfigure(GPIO_PB1_U1TX); //milestone2
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
    GPIOPinTypeUART(GPIO_PORTB_BASE, GPIO_PIN_0 | GPIO_PIN_1); //milestone2
    // Use the internal 16MHz oscillator as the UART clock source.
    UARTClockSourceSet(UART0_BASE, UART_CLOCK_PIOSC);
    // Enable the UART interrupt
    IntEnable(INT_UART1);
    UARTIntEnable(UART1_BASE, UART_INT_RX | UART_INT_RT);
    // Enable processor interrupts
    UARTConfigSetExpClk(UART1_BASE, SysCtlClockGet(), 115200, (UART_CONFIG_WLEN_8 |
UART_CONFIG_STOP_ONE |UART_CONFIG_PAR_NONE));

    // Initialize the UART for console I/O.
    UARTStdioConfig(1, 115200, SysCtlClockGet());
}
void handler50ms(void) {
    // Clear the timer interrupt
        TimerIntClear(WTIMER0_BASE, TIMER_TIMA_TIMEOUT);
        Semaphore_post(pidSemaphore);
}
```

```
void ConfigureTimer50ms(void)
{
   // Assuming a 50MHz system clock (adjust accordingly)
   uint32_t ui32Period = SysCtlClockGet()/1000 * 50 - 1;  // 50ms period
   // Enable the timer peripheral
   SysCtlPeripheralEnable(SYSCTL_PERIPH_WTIMER0);
   // Configure Timer0 to be periodic
   TimerConfigure(WTIMER0_BASE, TIMER_CFG_A_PERIODIC | TIMER_CFG_SPLIT_PAIR );
   // Set the load value for Timer0
   TimerLoadSet(WTIMER0_BASE, TIMER_A, ui32Period -1);
   // Enable the timer interrupt in the NVIC
   IntEnable(INT_WTIMER0A);
   TimerIntEnable(WTIMER0_BASE, TIMER_TIMA_TIMEOUT);
   // Enable Timer0
   TimerEnable(WTIMER0_BASE, TIMER_A);
}
void handler40msB(void) {
   // Clear the timer interrupt
   TimerIntClear(WTIMER1_BASE, TIMER_TIMB_TIMEOUT);
   // Here, you can set a flag or post a semaphore, etc., based on your application needs
   handleLineDetection();
}
void ConfigureTimer40msB(void) {
   // Assuming a 50MHz system clock (adjust accordingly)
   uint32_t ui32Period = SysCtlClockGet() / 1000 * 40 - 1;  // 40ms period
   // Enable the timer peripheral
   SysCtlPeripheralEnable(SYSCTL_PERIPH_WTIMER1);
   // Configure Timer1B to be periodic
   TimerConfigure(WTIMER1_BASE, TIMER_CFG_B_PERIODIC | TIMER_CFG_SPLIT_PAIR);
   // Set the load value for Timer1B
   TimerLoadSet(WTIMER1_BASE, TIMER_B, ui32Period - 1);
   // Enable the timer interrupt in the NVIC
   IntEnable(INT_WTIMER1B);
   TimerIntEnable(WTIMER1_BASE, TIMER_TIMB_TIMEOUT);
   // Enable Timer1B
   TimerEnable(WTIMER1_BASE, TIMER_B);
}
void handlerBT(void) {
      uint32_t ui32Status;
      // Get the interrrupt status
      ui32Status = UARTIntStatus(UART1_BASE, true);
      // Clear the asserted interrupts
      UARTIntClear(UART1_BASE, ui32Status);
      if (UARTCharsAvail(UART1_BASE))
            {
               uint32_t  ui32Loop ;
               char command[3];
               for(ui32Loop = 0; ui32Loop < 3; ui32Loop++)
               {
                  command[ui32Loop] = UARTCharGet(UART1_BASE);
               }
```

```
            if(command[0] == 'f' && command[1] == 'w' && command[2] == 'd')
            {
               // Forward: Set GPIO_PIN_2 HIGH and GPIO_PIN_4 LOW
               GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, GPIO_PIN_2); // High
               GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0x0);        // Low
            }
            else if(command[0] == 'r' && command[1] == 'e' && command[2] == 'v')
            {
               // Reverse: Set GPIO_PIN_2 LOW and GPIO_PIN_4 HIGH
               GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0x0);        // Low
               GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, GPIO_PIN_4); // High
            }
            else if(command[0] == 's' && command[1] == 't' && command[2] == 'p')
            {
               PWMOutputState(PWM0_BASE, PWM_OUT_0_BIT, false);
               PWMOutputState(PWM0_BASE, PWM_OUT_1_BIT, false);
               last_detected_line = NO_LINE;
               onVal = 0;
            }
            else if(command[0] == 's' && command[1] == 't' && command[2] == 'r')
            {
               PWMOutputState(PWM0_BASE, PWM_OUT_0_BIT, true);
               PWMOutputState(PWM0_BASE, PWM_OUT_1_BIT, true);
               GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, GPIO_PIN_2); // High
               GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0x0);
               onVal = 1;
               last_detected_line = NO_LINE;
            }
            else if(command[0] == 'l' && command[1] == 'o' && command[2] == 'w')
            {
              LeftDutyCycle = 25;
              RightDutyCycle = 25;
              PWMPulseWidthSet(PWM0_BASE, PWM_OUT_0, LeftDutyCycle * pwmMax / 100);
              PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, RightDutyCycle * pwmMax / 100);
            }
            else if(command[0] == 'm' && command[1] == 'i' && command[2] == 'd')
            {
              LeftDutyCycle = 50;
              RightDutyCycle = 50;
              PWMPulseWidthSet(PWM0_BASE, PWM_OUT_0, LeftDutyCycle * pwmMax / 100);
              PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, RightDutyCycle * pwmMax / 100);
            }
            else if(command[0] == 'm' && command[1] == 'a' && command[2] == 'x')
            {
              LeftDutyCycle = 100;
              RightDutyCycle = 100;
              PWMPulseWidthSet(PWM0_BASE, PWM_OUT_0, LeftDutyCycle * pwmMax / 100);
              PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, RightDutyCycle * pwmMax / 100);
            }
         }
}
```

```
int main(void)
{
   SysCtlClockSet(SYSCTL_SYSDIV_5 | SYSCTL_USE_PLL | SYSCTL_XTAL_16MHZ |
                  SYSCTL_OSC_MAIN);
      IntMasterEnable();
      // Enable the GPIO port that is used for the on-board LED.
      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
      SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
      while(!SysCtlPeripheralReady(SYSCTL_PERIPH_PWM0))
      {
      };
      //milestone 8
      SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER2);
      // Wait until the TIMER2 peripheral is ready
      while(!SysCtlPeripheralReady(SYSCTL_PERIPH_TIMER2))
      {
         // No-op; just waiting
      }
      // Once TIMER2 is ready, configure it
      TimerConfigure(TIMER2_BASE, TIMER_CFG_PERIODIC_UP);  // Use up-count mode to
measure time
      SysCtlPWMClockSet(SYSCTL_PWMDIV_64);
      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
      GPIOPinTypePWM(GPIO_PORTB_BASE, GPIO_PIN_6 | GPIO_PIN_7);
      GPIOPinConfigure(GPIO_PB6_M0PWM0);
      GPIOPinConfigure(GPIO_PB7_M0PWM1);
      PWMGenPeriodSet(PWM0_BASE, PWM_GEN_0, pwmMax);
      PWMGenConfigure(PWM0_BASE, PWM_GEN_0, PWM_GEN_MODE_DOWN |
PWM_GEN_MODE_NO_SYNC);
      PWMOutputState(PWM0_BASE, PWM_OUT_0_BIT, false);
      PWMOutputState(PWM0_BASE, PWM_OUT_1_BIT, false);
      PWMPulseWidthSet(PWM0_BASE, PWM_OUT_0, RightDutyCycle * (float)pwmMax / 100);
      PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, LeftDutyCycle * (float)pwmMax / 100);
      PWMGenEnable(PWM0_BASE, PWM_GEN_0);
      SysCtlClockSet(SYSCTL_SYSDIV_5 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
SYSCTL_XTAL_16MHZ);
      SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
      GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_3);
      GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_2);
      ADCSequenceDisable(ADC0_BASE, 1);
      ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 0);
      ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_CH0);
      ADCSequenceStepConfigure(ADC0_BASE, 1, 1, ADC_CTL_CH0);
      ADCSequenceStepConfigure(ADC0_BASE, 1, 2, ADC_CTL_CH1);
      ADCSequenceStepConfigure(ADC0_BASE, 1, 3, ADC_CTL_CH1 | ADC_CTL_IE |
ADC_CTL_END);
      ADCSequenceEnable(ADC0_BASE, 1);  //old ADC set up
      // Enable the GPIO pins for the LED (PF2 & PF3)
      GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1);
      GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_2);
```

```
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_3);
    GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_4);
    GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_2);
    GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_3);

    // Use up-count mode to measure time
    volatile uint32_t ui32Loop;
    // Initialize the UART
    ConfigureUART();
    ConfigureTimer50ms();
    ConfigureTimer40msB();
  BIOS_start();
  return (0);
}
```

6. Discussion of problems, suggestions for improvements

6.1. Discussion of Problems Encountered

Throughout the development of our project, we encountered several challenges that provided invaluable learning experiences and insights into the complexities of robotics and embedded systems. Here we discuss some of the most significant hurdles we faced and the lessons learned from them.

PID Control Fine-Tuning

One of the primary challenges we faced was the fine-tuning of the PID (Proportional-Integral-Derivative) control algorithm. The PID algorithm was critical for ensuring that the robot moved smoothly and responded accurately to the environmental conditions it encountered. Despite numerous adjustments and iterations, finding the perfect equilibrium between the robot's responsiveness and stability proved to be an arduous task. The intricacies of tuning the PID parameters were magnified by the dynamic nature of the robot's operating environment, particularly when navigating through complex maze-like structures.

Sensor Calibration and Environmental Variability

Calibrating the sensors, especially the infrared distance sensors, posed another significant challenge. The sensors were pivotal in helping the robot understand and interact with its surroundings. However, we observed that changes in environmental conditions, such as lighting and surface reflectivity, led to inconsistent sensor readings. This variability sometimes resulted in the robot deviating from its intended path or making erroneous

decisions, highlighting the importance of robust sensor data processing and environmental adaptability in robotics.

Data Transmission Latency

Our project also faced issues related to data transmission, particularly when dealing with real-time communication between the robot and our control systems. During periods of intensive data exchange, we encountered latency issues that affected the timeliness and reliability of the data being transmitted. These delays occasionally hindered the robot's ability to respond promptly to commands or environmental changes, underscoring the need for more efficient and optimized communication protocols in real-time systems.

Software Complexity and Debugging

As the project progressed, the complexity of the software architecture increased significantly. Managing and debugging the multi-threaded code, especially when integrating different modules like sensors, motor control, and communication, became increasingly challenging. We learned the importance of modular design, thorough testing, and comprehensive documentation in managing complex software systems.

Mechanical Robustness

Lastly, the mechanical design and assembly of the robot presented its own set of challenges. Ensuring that the robot was mechanically robust to withstand the operational stresses and environmental interactions required meticulous design considerations and iterative testing. We experienced issues with component wear and alignment, which taught us valuable lessons in mechanical design and the importance of material selection.

6.2. Suggestions for Improvements

Reflecting on the challenges faced during our project, we propose several improvements that could significantly enhance the performance and reliability of our robot. Here are some key areas where focused enhancements could yield substantial benefits:

Advanced PID Control and Adaptive Mechanisms

The PID control algorithm is pivotal for the smooth operation of our robot. To refine its performance, a deeper exploration into iterative tuning of the PID parameters is essential. Additionally, integrating adaptive control mechanisms that adjust in real-time to the robot's dynamics and environmental changes could significantly improve navigation accuracy and stability. This approach would allow the robot to maintain optimal performance even in varying conditions.

Enhanced Sensor Calibration and Environmental Compensation

The inconsistencies in sensor readings due to environmental variability highlight the need for advanced calibration techniques. Implementing environmental compensation algorithms could help mitigate the impact of external factors on sensor accuracy. Regular recalibration routines and algorithms that adjust sensor readings based on detected environmental conditions could lead to more consistent and reliable sensor performance.

Improved Data Transmission Protocols

Enhancing the data transmission protocols is crucial for ensuring real-time responsiveness. Exploring more efficient communication protocols and implementing error-checking mechanisms could address the latency issues we encountered. Optimizing the data packet size and transmission frequency could also improve the timeliness of data exchange, even during peak operation times.

Mechanical Design Enhancements

In the mechanical aspect, future designs could focus on increasing the durability and robustness of the robot. Using materials that offer better wear resistance, improving component alignment, and designing for easy maintenance could significantly extend the robot's operational life.

7. Conclusion

This project's successful conclusion showcases our ability to design and implement a sophisticated autonomous robotic system, utilizing an embedded microcontroller paired with a Real-Time Operating System (RTOS). Our robot's adept navigation through a defined course, its consistent maintenance of a set distance from the walls, and its acute responsiveness to varying line markings exemplify the effective integration of control systems and sensory feedback mechanisms.

A significant aspect of our project was the integration and calibration of sensors, including infrared and light reflectance sensors. This process was crucial for accurate

environmental perception and navigation. The challenges encountered in sensor calibration, particularly under varying environmental conditions, underscored the importance of precision and adaptability in robotics. Our experience in this area highlighted the importance of considering environmental variables in sensor-based systems.

The incorporation of a PID control algorithm was pivotal in achieving precise motor control and stable navigation. The iterative process of tuning the PID parameters provided us with a deeper understanding of control theory and its practical applications. However, achieving a balance between responsiveness and stability within the PID framework presented a noteworthy challenge, requiring continuous adjustments and testing.

Data handling and transmission posed another challenge, particularly during periods of intensive communication between the robot and the PC interface. We encountered latency issues, which emphasized the need for more efficient and reliable data communication methods in real-time systems. This experience has led us to consider alternative communication protocols and error-checking mechanisms in future projects to enhance data transmission reliability.

In conclusion, this project was not just about achieving its technical objectives; it was also a journey of learning and applying the principles of embedded systems, control theory, and real-time systems in a hands-on and engaging manner. The skills and knowledge gained through this project have prepared us for more advanced explorations in the field of robotics and embedded systems. The project's success can be attributed to meticulous planning, iterative design, continuous testing, and an incremental approach to problem-solving, with each milestone building on the foundations of the previous ones. This project has laid a solid foundation for future advancements in the field and has been an invaluable experience in our educational and professional growth.