PRINCIPLES OF EMBEDDED SYSTEMS DESIGN

CMPE 443

Final Project Report

Fall 2019

Wall Follower Car

Team Name: Süper Araba

Team Members:

Afra Akbaş - 2015400024

Dilruba Köse - 2015400219

İrem Uğuz - 2015400165

İrem Üstünboyacıoğlu - 2015400168

Contents

1 Introduction and Summary

In this project, the main goal is to implement a toy car that follows through a wall. The toy car will consist of 4 LEDs, 1 potentiometer, 2 LDRs, 1 motor controller and 1 ultrasonic sensor.

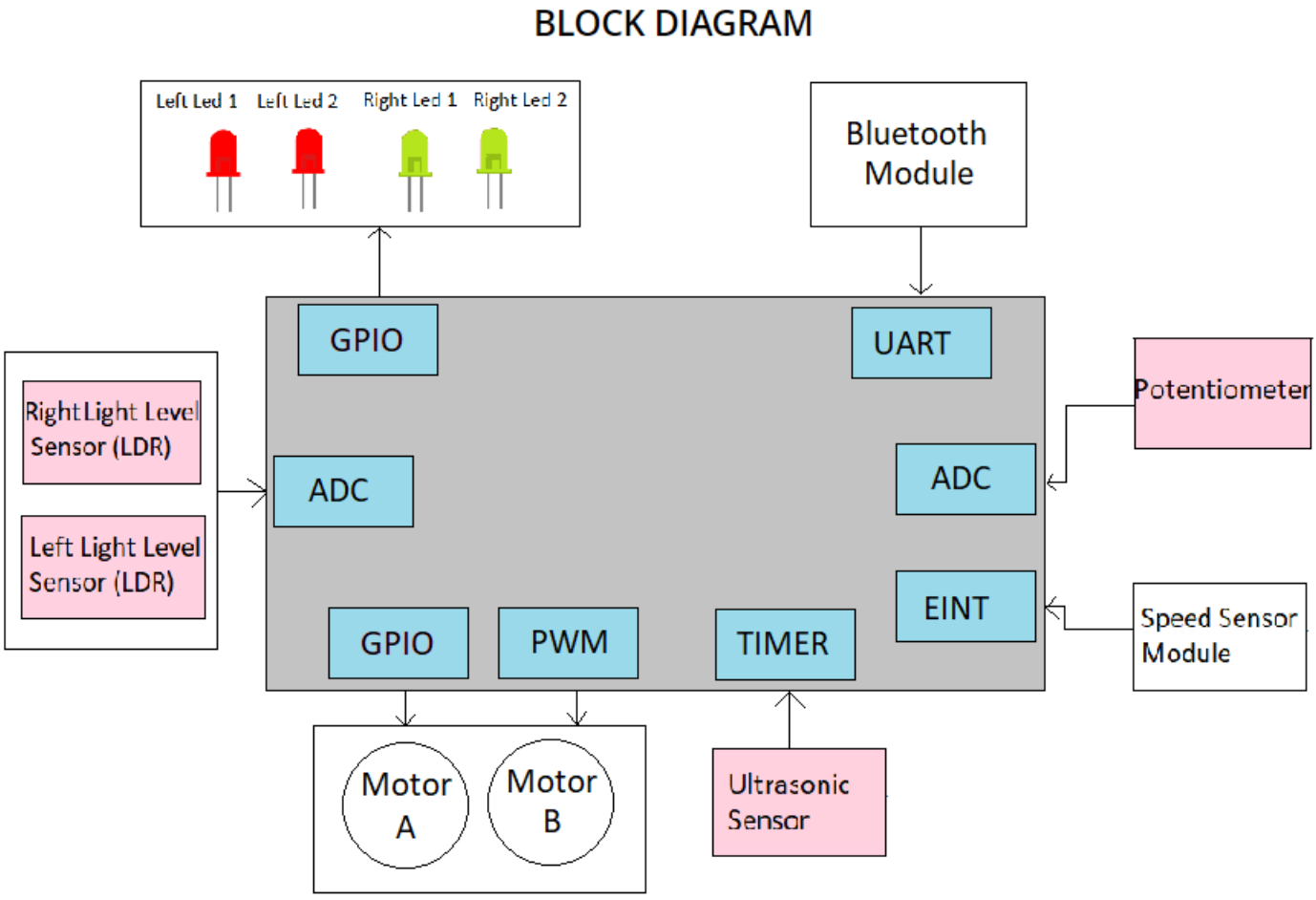
The main expectation of the car is that it can follow through a wall automatically. It will get closer to the wall when it senses the distance is too much and it increases the distance when it senses the wall is too close. The car will need to sense the distance to the wall with its ultrasonic sensor.

The car will set its speed according to the input of potentiometer. When the user turns down potentiometer, the speed should decrease; when the user turns up the potentiometer, the speed should increase gradually. The lowest input of potentiometer should turn down the motor totally and the highest input should set the speed to maximum.

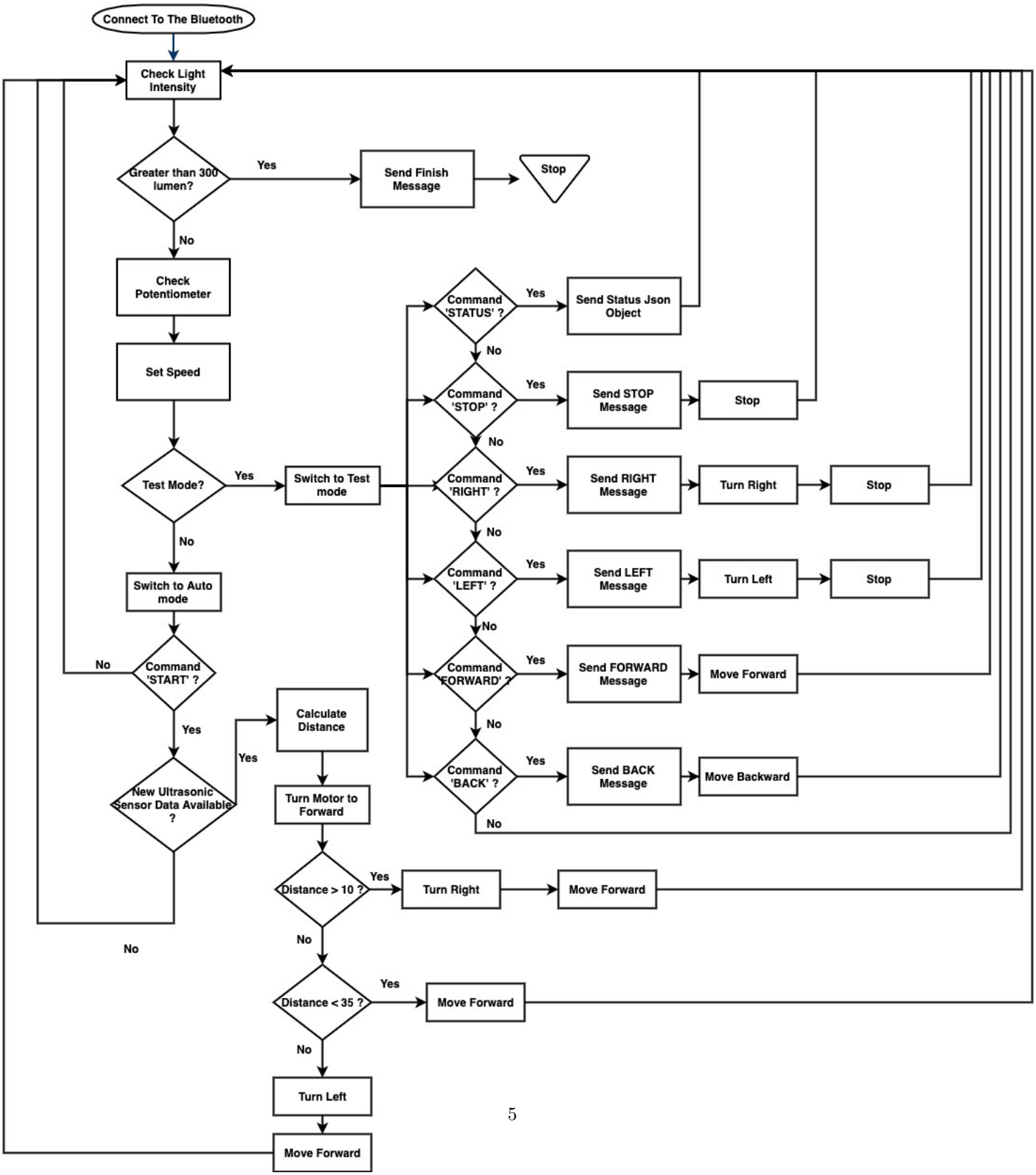
The car will have two LEDs on the front and two LEDs on the back. When the car is moving forward, the front LEDs should turn on. In the same way, when the car is moving backwards, back LEDs should turn on. When the car is turning to the left, left LEDs should blink twice a second. The right LEDs should work in the same way.

The car uses speed sensor to measure the distance. Speed sensor sends light to sense how many time the wheel has turned. We mainly use the speed sensor to check the degree of turn.

2 Block Diagrams



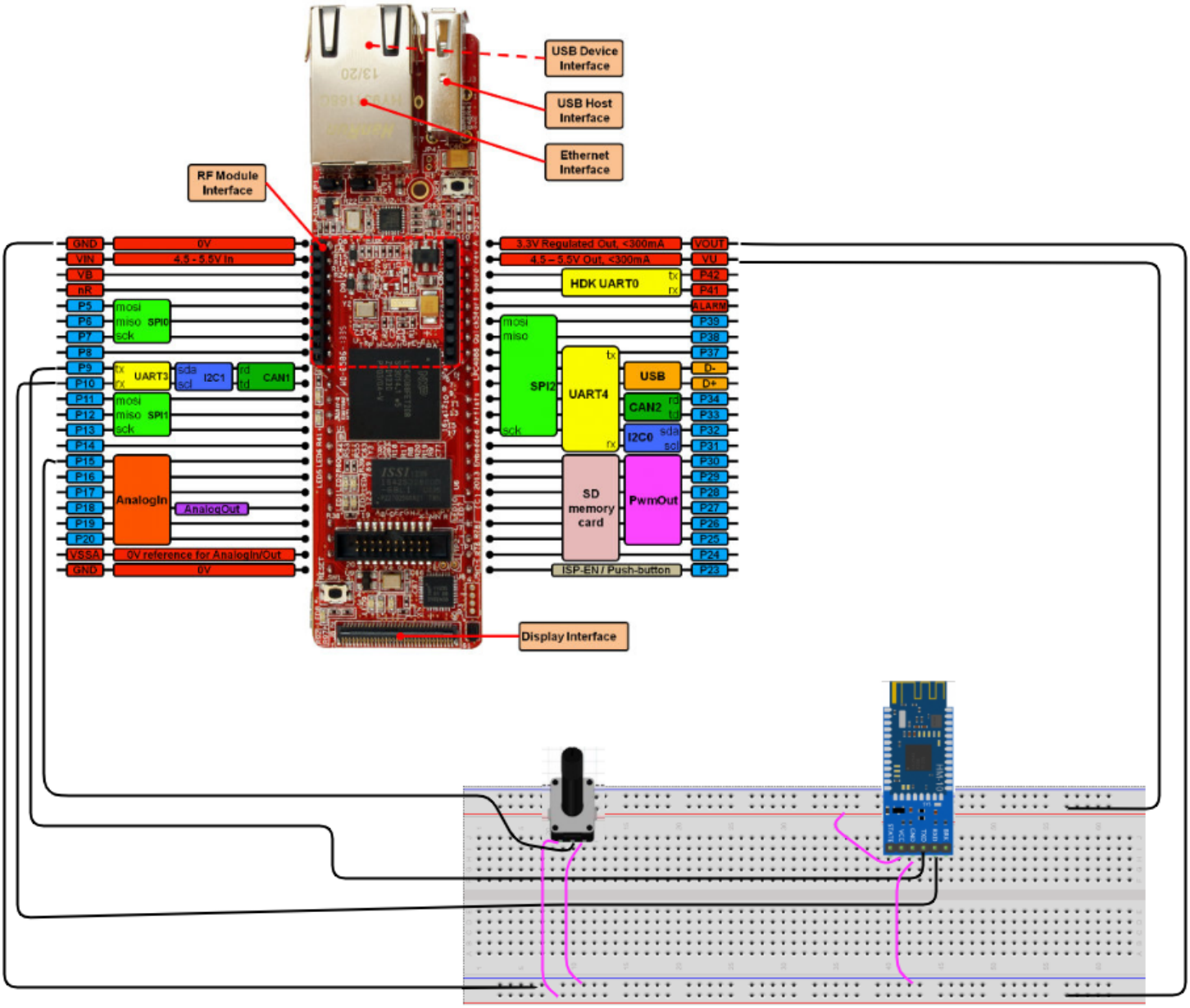
3 A Flowchart Of The System



4 Pin Connection Table

Pin Number	LPC4088 Pin	Functionality	Usage
P30	P1.2	PWM0[1]	LEFT LED 1
P29	P1.3	PWM0[2]	LEFT LED 2
P28	P1.5	PWM0[3]	RIGHT LED 1
P27	P1.6	PWM0[4]	RIGHT LED 2
P5	P1.24	PWM1[5]	MOTOR A ENA
P8	P0.21	GPIO	MOTOR A IN1
P18	P0.26	GPIO	MOTOR A IN2
P6	P1.23	PWM1[4]	MOTOR B ENA
P11	P0.9	GPIO	MOTOR B IN1
P12	P0.8	GPIO	MOTOR B IN2
P23	P2.10	EINT0	Speed Sensor
P9	P0.0	U3_TXD	HM10 TX
P10	P0.1	U3_RXD	HM10 RX
P17	P0.25	ADC0[2]	LDR 1
P16	P0.24	ADC0[1]	LDR 2
P15	P0.23	ADC0[0]	Potentiometer
-	P1.29	T0_MAT1	ADC Timer
P32	P5.2	T3_MAT2	Ultrasonic Trigger Pin
P33	P0.5	T2_CAP1	Ultrasonic Echo Pin

5 Circuit Schematics



We draw circuit schematics for only HM10(Bluetooth) and potentiometer, since other parts are embedded to the car. The only custom circuit we made was for HM10 and potentiometer.

6 Intended Usage Of The System Components

Our code has 6 main classes that are used in the main. We named those classes according to the main car components that we use to implement them. In this section, we will briefly explain what are the use cases of each class. Detailed implementation explanations can be found in next section.

6.1 GPIO

We used GPIO pins to set motors IN1, IN2 pins. The GPIO class has the initialization function to set the functionalities and DIRs of this pins to GPIO output pins. The IOCON addresses of the GPIO pins can be found on the GPIO.h file. To change the IOCON pins, it is enough to change the addresses in the header file.

6.2 PWM

We use PWM to implement two main functionalities. PWM0 is used to implement the blinking of the LEDs and PWM1 is used to set the speed of the motor. We chose to use two different PWMs as prescale values that we decided to use were different. The PWMs are initialized according to this. PWM IOCON pins can be found in the header file and if you want to change the pins and reuse our code, it is enough to change the addresses of the pins.

We also defined functions to determine the duty cycles of the given PWMs. PWM_Write() function takes PWM and the MR number that it uses and sets the duty cycle.

6.3 External

External class is used to initialize EINT0. Initialization function is a generic one that sets the necessary fields. We use counter and rotation variables to store the number of rotations that the wheel makes. If anyone wants to use the speed sensor, it is enough to check the rotation varible.

6.4 HM10

HM10 class is a very generic and reusable class. It is used to make the Bluetooth connection. HM10_Write() function and SendCommand function is used to send the given char[] to connected device. No other initialization is necessary. If anyone wants to use the class, it is enough that they change the TX and RX IOCON pin addresses in the header file. Then the connection must work with no problem.

6.5 Timer

Timer class is mainly used for ultrasonic sensor. Ultrasonic sensor requires two timers as the calculations are made according to the timer counter values that are stored from match and capture registers of the timers. Trigger and echo pins of the ultrasonic sensor can be found and change in the header file. The necessary functions to make calculations are in the c file, ready to use.

6.6 ADC

ADC class has the functions to calculate the values of potentiometer and LDRs. The necessary initialization is made in the init method of the class. To use the ADC class, it is enough to set the IOCON pin address in the header file. Then potentiometer value is returned from ADC_Pm_GetLastValue() and the average of LDR values are returned from ADC_LDR_GetLastValue().

6.7 Main

Most of the main code is specific to our use case. However, there are some functions in the main file that can be used to control the PWMs. set_pwm_rates and motor_pwm_write sets the correct duty cycles to motors and LED PWMs.

7 Implementation Of Functional Parts

7.1 GPIO

In the initialization function of GPIO functionalities of the pins are set to GPIO pins and the DIR bits of the pins are set to 1, to set the direction as output.

As the GPIO pins are used as IN1 and IN2 of motors, to turn the motor clockwise, IN1 must be 1 and IN2 must be 0. MotorA_Clockwise() below is setting the MotorA's direction to clockwise.

```
void MotorA_Clockwise(){
  uint32_t pin_value = PORT0->PIN;
  pin_value |= (1<<21);
  pin_value &= ~(1<<26);
  PORT0->PIN = pin_value;
}
```

Both motors should turn counter clockwise for car to go forward. Motor_Forward() function sets the both motors' direction to counter clockwise.

```
void Motor_Forward(){
  MotorA_CounterClockwise();
  MotorB_CounterClockwise();
```

```
}
```

In the same way, both motors should turn clockwise for car to go backward. Motor_Backward() function sets the both motors' direction to clockwise.

```
void Motor_Backward(){
  MotorA_Clockwise();
  MotorB_Clockwise();
}
```

When one motor goes forward and other motor goes backward, the car will turn to the direction of the backward going motor. Motor_Right() will make the MotorA go counter clockwise(forward) and MotorB to clockwise(backward), making the robot turn right as the MotorA is on the left and MotorB is on the right.

The same approach is done on the Motor_Left() method as well.

```
void Motor_Right(){
  MotorA_CounterClockwise();
  MotorB_Clockwise();
}
```

7.2 PWM

PWM-Init() method initialized the PWMs for LEDs and motors that we used. First, it set the IOCON functionalities of pins with the set-pwm-func() method. Then it enables the PWM0 and PWM1 in PCONP. We use 4 LEDs and 2 motors, so it enables the corresponding pins from the PCR register in PWM0 and PWM1. The PWM Timer Counter and the PWM Prescale Counter are synchronously reset on the next positive edge of PCLK. We want to generate an interrupt when TC matches MR0 and then reset the TC. To do that, it sets the corresponding bits to 1 in MCR and LER registers. PWM and Counter will be enabled with TCR in the end.

```
void PWM_Init() {
    //set the functionalities of all pins
  set_pwm_func(&IOCON_LEFT_LED1);
  set_pwm_func(&IOCON_LEFT_LED2);
  set_pwm_func(&IOCON_RIGHT_LED1);
  set_pwm_func(&IOCON_RIGHT_LED2);
  set_pwm1_func(&IOCON_MOTORA_SPEED);
  set_pwm1_func(&IOCON_MOTORB_SPEED);

  //Enable PWM0 and PWM1
  PCONP |= (1 << 5 | 1<<6);

  //Enable PWM output for corresponding pin.
  PWM0->PCR |= (1<<10 |1<<9 |1<<12 |1<<11);
  PWM1->PCR |= (1<<13 | 1<<12);

  // The PWM Timer Counter and the PWM Prescale Counter are synchronously reset on
    the next positive edge of PCLK
  PWM0->TCR = 1 << 1;

```

```
  PWM0->PR = 0;

  PWM0->MR0 = (PERIPHERAL_CLOCK_FREQUENCY / 1000000) * 1000 * 1000;

  PWM1->TCR = 1 << 1;

  PWM1->PR = 9;

  PWM1->MR0 = (PERIPHERAL_CLOCK_FREQUENCY / 1000000) * 20* 1000;

  //Reset TC, when MR0 matches TC. Also generate an interrupt when MR0 matches the
    value in the TC.
  PWM0->MCR = 1 << 1;

  PWM0->LER |= 1 << 0;

  PWM0->TCR = (1 << 0 | 1 << 3);

  PWM1->MCR = 1 << 1;

  PWM1->LER |= 1 << 0;

  PWM1->TCR = (1 << 0 | 1 << 3);
}
```

PWM-Write() method is useful for changing the duty cycle. To make LEDs blink twice in a second and to arrange motor speed.

```
void PWM_Write(uint32_t T_ON, uint8_t mr_number, PWM_TypeDef *PWMX) {
  if(T_ON > 100) {
    T_ON = 100;
  }

  T_ON = (uint32_t)(((PWMX->MR0) * T_ON) / 100);

  if (T_ON == PWMX->MR0) {
    T_ON++;
  }
  if(mr_number == 1){
    PWMX->MR1 = T_ON;
    PWMX->LER |= 1 << 1;
  }else if(mr_number==2){
    PWMX->MR2 = T_ON;
    PWMX->LER |= 1 << 2;
  }else if(mr_number==3){
    PWMX->MR3 = T_ON;
    PWMX->LER |= 1 << 3;
  }else if(mr_number==4){
    PWMX->MR4 = T_ON;
    PWMX->LER |= 1 << 4;
  }else if(mr_number==5){
    PWMX->MR5 = T_ON;
    PWMX->LER |= 1 << 5;
  }else if(mr_number==6){
    PWMX->MR6 = T_ON;
    PWMX->LER |= 1 << 6;
  }
}
```

It sets the IOCON functionalities of given IOCON-pin. We used a certain functionality for our LEDs.

```
void set_pwm_func(volatile uint32_t* iocon){
  uint32_t ioconRegisterValue = *iocon;
  ioconRegisterValue |= 0x3;
  ioconRegisterValue &= ~(1 << 2);
  *iocon = ioconRegisterValue;
}
```

It sets the IOCON functionalities of given IOCON-pin. We used a certain functionality for our motors .

```
void set_pwm1_func(volatile uint32_t* iocon){
  uint32_t ioconRegisterValue = *iocon;
  ioconRegisterValue |= 0x2;
  ioconRegisterValue &= ~(1 << 2 | 1<<0);
  *iocon = ioconRegisterValue;
}
```

7.3 External

External interrupts are used for speed sensor. Speed sensor sends light and when the wheel is in the hole, speed sensor senses the light.

By using EXTINT, when the speed sensor senses the light, an interrupt will be send. In the interrupt the rotation number of the wheel will be counted.

In the initialization code of the EXTINT we set the functionalities of IOCONS so EXTINT0. Then we set the mode to edge sensitive as the edges of the speed sensor will be counted. Then we set the polarity to low active so the interrupt will be sent on the falling edge.

After doing initializations, we enable interrupts for external interrupt register.

```
void External_Init() {
  //Change the functionality of the push button as EINT0
  IOCON_SPEED_SENSOR |= (1<<0);
  IOCON_SPEED_SENSOR &= ~(1<<1 | 1<<2);

  //Change the External interrupt mode as Edge Sensitive
  EXT->EXTMODE |= (1<<0);

  //Change polarity of the External Interrupt as Low-active
  EXT->EXTPOLAR &= ~(1<<0);
  //Enable interrupt for EINT0_IRQn
  NVIC_EnableIRQ(EINT0_IRQn);
}
```

In the IRQHandler of the EINT0, at first the interrupt is cleared. We count the rotations same with the counter value. We chose to increment rotations for each increase of the counter because the one full rotation of the wheel was too much turning for our car.

```
void EINT0_IRQHandler() {
  //Clear interrupt for EINT0
  EXT->EXTINT |= (1<<0);
```

```
  if(counter<1){
      counter++;
    }else{
      rotations++;
      counter=0;
  }
}
```

7.4 HM10

This HM10-Init() method initialized the UART functions. First, it sets the IOCON functionalities of RX and TX pins which are used in UART communications. Then, it enables the UART in PCONP.In UART, FCR is a register for enabling FIFO. DLL, DLM and FDR values are necessary for the setting baud rate.

```
void HM10_Init() {
  HM10_UART_TX_PIN |= 0x02;
  HM10_UART_RX_PIN |= 0x02;

  PCONP |= 1 <<  25;

  HM10_UART->FCR =  1 << 0
              |  0 << 1
              |  0 << 2
              |  0 << 6;

  HM10_UART->LCR |= (1 << 7);

  //Write correct DLM, DLL and FDR values for 9600 baudrate
  HM10_UART->DLM = 0x01;
  HM10_UART->DLL = 0x04;
  HM10_UART->FDR = 0x01 << 0 | 0x02 << 4;

  HM10_UART->LCR &= ~(1 << 7);

  HM10_UART->LCR =  3 << 0
               | 0 << 2
               | 0 << 3
               | 0 << 4;

  //Enable the Receive Data Available Interrupt.
  HM10_UART ->IER |= (1<<0);

  //Enable UART3_IRQn Interrupt.
  NVIC_EnableIRQ(UART3_IRQn);
}
```

The HM10-SendCommand() sends the given command to Bluetooth screen.

```
void HM10_SendCommand(char* command) {
  HM10_Write(command);
}
```

The HM10-ClearBuffer() method is used for clearing buffer after each use. It should be cleared to have a more reliable and accurate code.

```
void HM10_ClearBuffer() {
  HM10CurrentBufferIndex = 0;
  memset(HM10Buffer, 0, 255);
}
```

The HM10-ReadData() waits until data arrives. When a data arrives, it returns from receiver buffer register.

```
char HM10_ReadData() {
  while (!(HM10_UART->LSR & 0x01));
  return HM10_UART->RBR;
}
```

The HM10-WriteData() writes the data into the transmitter holding register, when data arrives.

```
void HM10_WriteData(char data) {
  while (!(HM10_UART->LSR & 0x20));
  HM10_UART->THR = data;
}
```

This method calls the HM10-WriteData method for writing data.

```
void HM10_Write(char* data) {
  while(*data > 0)  {
    HM10_WriteData(*data++);
  }
}
```

When an interrupt occurs in the UART communication, this UART3-IRQHandler() method is called. It reads data, writes into the buffer and sets the HM10NewDataAvailable to 1 which indicates a new data available.

```
void UART3_IRQHandler() {
  char data;
  data = HM10_ReadData();
  HM10Buffer[HM10CurrentBufferIndex] = data;
  HM10CurrentBufferIndex++;
  HM10NewDataAvailable = 1;
}
```

7.5 Timer

We use the timers for the three functionalities. ADC timer, Ultrasonic sensor Trigger timer and Ultrasonic Echo timer. ADC needs timer to calculate the frequency of the calculations. Ultrasonic sensor calculates the distance based on the time period between sending the ultrasonic sound and receiving it back.

```
void TimerInit(){
  ADC_Timer_Init();
  Ultrasonic_Trigger_Timer_Init();
  Ultrasonic_Capture_Timer_Init();
  Ultrasonic_Start_Trigger_Timer();
}
```

Ultrasonic capture timer will set the timer for receiving the echo signal. We used Timer2 for capture timer. At first we turn on the timer. Then we set the IOCON functionality. We set the PR of the timer to 59 because our clock frequency is 60 MHz and we want timer to increment once every one microsecond. We set the interrupt for every rising edge of the capture register and enable interrupts for Timer2.

```
void Ultrasonic_Capture_Timer_Init() {
   uint32_t value;
  //Change the mode of Timer2 to Timer Mode
  PCONP |= 1<<22;


  TIMER2->CTCR &= ~(1<<0 | 1<<1);

  value = TIMER2_ECHO_IOCON;
  value |= 3;
  value &= ~4;
  TIMER2_ECHO_IOCON = value;

   TIMER2->PR = 59;

  TIMER2->TCR &= ~(1 << 0);

  TIMER2->TCR |= (1 << 1);

  //Change CCR value for getting Interrupt when Rising Edge Occur for CAP 1
  TIMER2->CCR |= (1<<3 | 1<<4 | 1<<5);

  TIMER2->TCR &= ~(1 << 1);

  TIMER2->TCR |= (1 << 0);

  NVIC_EnableIRQ(TIMER2_IRQn);
}
```

The trigger timer will be used to determine the time between the sensor sending the sound wave. We chose the Timer3 for this. We open the Timer3 from PCONP and set the timer mode. Then we set the IOCON function fields to timer functionality. We set the PR to 59 with the same reason as the Echo timer above. We set the EMR values to initial LOW value and when the match occurs, the output will be set to clear. We enable interrupts for Timer3.

```
void Ultrasonic_Trigger_Timer_Init() {
  //Turn on Timer3.
  PCONP |= 1<<23;
  //Change the mode of Timer3 to Timer Mode.
  TIMER3->CTCR &= ~(1<<0 | 1<<1);
  uint32_t value;
  value = TIMER3_TRIGGER_IOCON;
  value |= 3;
  value &= ~4;
  TIMER3_TRIGGER_IOCON = value;

  TIMER3->TCR &= ~(1 << 0);

  TIMER3->TCR |= (1 << 1);

  //Change PR Register value for 1 microsecond incrementing
  TIMER3->PR = 59;

  //Write the Correct Configuration for EMR (LOW output value of Trigger Pin when
    match occurs and Initial value is LOW)
  TIMER3->EMR |= (1<<8);
  TIMER3->EMR &= ~(1<<2 | 1<<9);

  NVIC_EnableIRQ(TIMER3_IRQn);
```

```
24
25    NVIC_SetPriority(TIMER3_IRQn,5);
26
27    NVIC_ClearPendingIRQ(TIMER3_IRQn);
28  }
```

We determine the frequency of analog to digital conversion with ADC timer. We chose Timer0 for this functionality. We turn on Timer0 on PCONP and set the IOCON functionalities.

We toggle the external match bit when there is a match and reset the TC when a match occurs. We will be using Match1 for ADC and we will set it rising edge.

```
1  void ADC_Timer_Init() {
2    TIMER_PIN_IOCON &= ~(0x07);
3    TIMER_PIN_IOCON |= 0x03;
4
5    PCONP |= 1 << 1;
6
7    TIMER0->CTCR = 0x0;
8
9    TIMER0->TCR &= ~(1 << 0);
10
11   TIMER0->TCR |= (1 << 1);
12
13   TIMER0->PR = PERIPHERAL_CLOCK_FREQUENCY / 1000000 - 1;
14
15   //Toggle the corresponding External Match bit/output when MR1 matches with TC.
16    TIMER0->EMR |= (1 << 1 );
17    TIMER0->EMR |= 3 << 6;
18
19   //Reset the TC value whenever MR1 matches it
20    TIMER0->MCR |= (1<<4);
21
22   //Configure MR1 as ADC will start every 100 milliseconds (Do not forget you
       configured ADC when rising edge occurs on TIMER 0 MATCH 1)
23   TIMER0->MR1 = 100000;
24
25   TIMER0->TCR &= ~(1 << 1);
26
27   TIMER0->TCR |= (1 << 0);
28 }
```

In this function, we at first set the output of trigger pin to high. Between every trigger action, it is suggested by the ultrasonic sensor manual, that we wait at least 10 microseconds. That's why we set MR2 to 10 as TC is incremented every 1 microsecond. We enable interrupts for MR2 register.

```
1  void Ultrasonic_Start_Trigger_Timer() {
2    //Change output value of Trigger Pin as HIGH
3    TIMER3->EMR |= (1<<2);
4
5    //Give correct value to corresponding MR Register for 10 microsecond
6    TIMER3->MR2 = 10;
7
8    //Enable interrupt for MR2 register, if MR2 register matches the TC.
9    TIMER3->MCR |= (1<<6);
10
11   //Remove the reset on counters of Timer3.
```

```
  TIMER3->TCR &= ~(1<<1);


  //Enable Timer Counter and Prescale Counter for counting.
  TIMER3->TCR |=(1<<0);
}
```

In IRQ handler of the Timer3 we clear the interrupt at first. We used Timer3 for trigger of ultrasonic sensor. If the ultrasonic trigger has started we will increment the MR2 according to TC as the suggested waiting time is 60 ms. If the trigger has not started, the trigger signal will be sent and the output will be set HIGH.

```
void TIMER3_IRQHandler() {
  //Write HIGH bit value to IR Register for Corresponding Interrupt
  TIMER3->IR |= (1<<2);

  if(ultrasonicSensorTriggerStart == 0) {
    //Change MR2 Register Value for Suggested Waiting
    TIMER3->MR2 = TIMER3->TC + 60000;
    ultrasonicSensorTriggerStart = 1;
  }
  else {
    TIMER3->EMR |= (1 << 2);
    TIMER3->MR2 = 10 + TIMER3->TC;

    ultrasonicSensorTriggerStart = 0;
  }
}
```

Timer2 is used for capturing the echo of the ultrasonic sensor. In the IRQ handler of the Timer2, at first the interrupt will be cleared. If the rising edge is detected, rising time will be captured in ultrasonicSensorRisingCaptureTime. If the falling edge is detected, TC will be captured in ultrasonicSensorFallingCaptureTime variable.

```
void TIMER2_IRQHandler(void)
{
    TIMER2->IR |= 1 << 5;

    if(ultrasonicSensorCaptureRisingEdge == 1) {
      ultrasonicSensorRisingCaptureTime = TIMER2->CR1;

      LPC_TIM2->CCR = (1 << 4) | (1 << 5);
      ultrasonicSensorCaptureRisingEdge = 0;
    }
    else {
      ultrasonicSensorFallingCaptureTime = TIMER2->CR1;
      ultrasonicSensorNewDataAvailable = 1;

      LPC_TIM2->CCR = (1 << 3) | (1 << 5);
      ultrasonicSensorCaptureRisingEdge = 1;
    }
}
```

In the user manual of ultrasonic sensor, the formula to calculate the distance in centimeters is (End Time - Start Time)/58. In calculateUSDistance(), according to this formula, we subtract start capture time from falling capture time. As the TC is incremented once every 1 microsecond, without needing any more conversion, we divide it to 58 and find the distance in cm.

```
uint32_t calculateUSDistance(){
  ultrasonicSensorNewDataAvailable = 0;
  return (ultrasonicSensorFallingCaptureTime - ultrasonicSensorRisingCaptureTime)
    /58;
}
```

7.6 ADC

We use Analog to Digital Converter for 2 things: LDR and Potentiometer. ADC_Init() method initialize the ADC by and enables the interrupt. First, it opens ADC on PCONP register, then it sets necessery bits in registers to make ADC operational without Burst mode. Afterwards, it calls ADC-Pm-Init() for potentiometer initialization and ADC-LDR-Init() for LDR initializations. Lastly, it enables ADC interrupts that will come from potentiometer and LDR.

```
void ADC_Init() {
  uint32_t value;
  //Turn on ADC.
  PCONP |= (1<<12);

  //Set the CLKDIV and make the A/D converter operational without Burst mode.
  value = ADC->CR;
  value &= (0xFFFF0000);
  value |= (ADC_CLKDIV<<8);
  value &= ~(1<<16);

  //Make the A/D converter operational
  value |= (1<<21);
  ADC->CR = value;
  ADC_Pm_Init();
  ADC_LDR_Init();


  //Enable ADC_IRQn (Interrupt Request).
  NVIC_EnableIRQ(ADC_IRQn);

}
```

In ADC-LDR-Init(), first we changed the function value of selected pin to ADC which is 001. Then mode of the pin is set to Analog and CR sel is configured as which channel of ADC is used (AD0[1] AD0[2]) . All of these are done for both LDR1 and LDR2. Lastly interrupt is enabled and ADC-LDR-Start() method is called.

```
void ADC_LDR_Init() {
  uint32_t value;
  //Change the function value of pin to ADC.
  value = LDR1_PIN_IOCON;
  value &= ~(7);
  value |= (1<<0);

```

```
  //Change the mode value of pin to mode which should be selected if Analog mode is used.
  value &= ~(1<<3 | 1<<4 |1<<7);
  LDR1_PIN_IOCON = value;

    //Change the function value of pin to ADC.
  value = LDR2_PIN_IOCON;
  value &= ~(7);
  value |= (1<<0);

  //Change the mode value of pin to mode which should be selected if Analog mode is used.
  value &= ~(1<<3 | 1<<4 |1<<7);
  LDR2_PIN_IOCON = value;

  value = ADC->CR;
  value |= (1<<1 |1<<2);
  ADC->CR = value;

  //Enable interrupt for corresponding pin.
  ADC->INTEN |= (1<<1 | 1<<2);

  ADC_LDR_Start();
}
```

In ADC-Pm-Init(), first the function value of selected pin is chnaged to ADC which is 001. Then mode of the pin is set to Analog and CR sel is configured as which channel of ADC is used (AD0[0]) . Lastly interrupt is enabled and ADC-Pm-Start() method is called.

```
void ADC_Pm_Init() {
  uint32_t value;
  //Change the function value of pin to ADC.
  value = ANALOG_PIN_IOCON;
  value &= ~(7);
  value |= (1<<0);

  //Change the mode value of pin to mode which should be selected if Analog mode is used.
  value &= ~(1<<3 | 1<<4 |1<<7);
  ANALOG_PIN_IOCON = value;

  value = ADC->CR;
  //Change Analog/Digital mode of pin to Analog.
  value |= (1<<0);
  ADC->CR = value;

  //Enable interrupt for corresponding pin.
  ADC->INTEN |= (1<<0);

  ADC_Pm_Start();
}
```

In ADC-LDR-Start(), we set CR register of ADC to start conversion on a rising edge on TIMER0 MATCH1

```
void ADC_LDR_Start () {
  //Code for starting A/D conversion on a rising edge on the TIMER 0 MATCH 1.
```

```
  uint32_t value = ADC->CR;
  value &= ~(15<<24);
  value |= (1<<26);
  ADC->CR = value;
}
```

This method is for getting the last read value from LDR. First set ADC-LDR-New-Data-Available to 0 to prevent read data when there is no. Then return the value.

```
uint32_t ADC_LDR_GetLastValue() {
  ADC_LDR_New_Data_Available = 0;
  return ADC_LDR_Last;
}
```

In ADC-Pm-Start(), we set CR register of ADC to start conversion on a rising edge on TIMER0 MATCH1

```
void ADC_Pm_Start () {
  //Write a code for starting A/D conversion on a rising edge on the TIMER 0 MATCH
    1.
  uint32_t value = ADC->CR;
  value &= ~(15<<24);
  value |= (1<<26);
  ADC->CR = value;
}
```

This method is for getting the last read value from Potentiometer.We first set ADC-Pm-New-Data-Available to 0 to prevent read data when there is no. Then return the value.

```
uint32_t ADC_Pm_GetLastValue() {
  ADC_Pm_New_Data_Available = 0;
  return ADC_Pm_Last;
}
```

This method is run whenever there is an interrupt from ADC. Therefore, we separated it with if conditions to distinguish between LDR interrupt and Potentiometer interrupt. First we clear GDR to make it avaliable again. Then, in first "if" we check ADC STAT register to look if this is an interrupt made by Potentiometer. If so, we get the last read value from ADC DR register and write it to our global variable and shift it to take neccessary bits. Then we set ADC-Pm-New-Data-Available to 1, so that we can understand that there is a potentiometer data from other methods.

In second "if" we check if this is an interrupt from LDR. Then we read last value from DR register and assign it to LDR1-value after shifting. Same is done for LDR2 and then we take the avarage of LDR values to have one last value. We assign it to global ADC-LDR-Last to use it in other methods. Then ADC-LDR-New-Data-Available is set to 1, so that we can understand that there is a LDR data from other methods.

```
void ADC_IRQHandler() {
  ADC->GDR &= ~((uint32_t)1 << 31);

  //Write the converted data (only the converted data) to ADC_Last variable.
  if(ADC->STAT & (1<<0)){
    ADC_Pm_Last = ADC->DR[0];
    ADC_Pm_Last = ADC_Pm_Last << 16;
    ADC_Pm_Last = ADC_Pm_Last >> 20;

    ADC_Pm_New_Data_Available = 1;
  }
  if(ADC->STAT & (1<<2)){
    LDR1_value = ADC->DR[1];
    LDR1_value = LDR1_value << 16;
    LDR1_value = LDR1_value >> 20;
    LDR2_value = ADC->DR[2];
    LDR2_value = LDR2_value << 16;
    LDR2_value= LDR2_value >> 20;
    ADC_LDR_Last = (LDR1_value  + LDR2_value) /2;

    ADC_LDR_New_Data_Available = 1;
  }
}
```

7.7 Main

In main, there is an init() method to initialized all functions that we used in the whole code. Timer, ADC, External Interrupt, HM10, and PWM init methods are used here.

```
void init() {
  TimerInit();
  ADC_Init();
  External_Init();
  __enable_irq();
  PWM_Init();
  HM10_Init();
  Motor_Init();
  PWM_Write(100, 5, PWM1);
  PWM_Write(100, 4, PWM1);
  HM10NewDataAvailable = 1;
  HM10_ClearBuffer();
  HM10_SendCommand("AT\r\n");
  HM10NewDataAvailable = 1;
  PWM_Cycle_Rate(10,PWM1);
}
```

This function takes 5 parameters. 4 of them is used for LEDs and last one for Motor. It sets duty cycle of PWM.

```

void set_pwm_rates(uint32_t left1_rate, uint32_t left2_rate, uint32_t right1_rate,
    uint32_t right2_rate,uint32_t motor_rate){
    PWM_Write(left1_rate, 1, PWM0);
    PWM_Write(left2_rate, 2, PWM0);
    PWM_Write(right1_rate, 3, PWM0);
    PWM_Write(right2_rate, 4, PWM0);
    PWM_Write(motor_rate, 5, PWM1);
```

```
    PWM_Write(motor_rate, 4, PWM1);
}
```

This function takes 1 parameter and sets duty cycle of motor.

```

void motor_pwm_write(uint32_t rate){
  PWM_Write(rate, 5, PWM1);
  PWM_Write(rate, 4, PWM1);
}
```

The update() method has a crucial role in this project. Almost all functions or variables used in this method. There is a 'mode' variable to determine the mode of the car. It checks the LDR and Potentiometer controls and takes actions according to that. In the test mode, Bluetooth communication determines the actions and returns the given command to the screen. In the auto mode, it runs without a Bluetooth command, but when it starts or finishes its move, write to the screen. Its actions are controlled by the ultrasonic sensor and LDR.

```
void update() {
  if(HM10NewDataAvailable && HM10Buffer[HM10CurrentBufferIndex-1]==10){
    strcpy(characters,HM10Buffer);
    HM10_SendCommand(characters);
    HM10_ClearBuffer();
  }
  if(strcmp(characters,"TEST\r\n")==0)
    mode = 0;
  if(strcmp(characters,"AUTO\r\n")==0){
    mode = 1;
    stop = 0;
    finished = 0;
    set_pwm_rates(0,0,0,0,0);
    speed = 0;
  }
  if(!stop){
    if(ADC_LDR_Last<2700){
      speed=0;
      motor_pwm_write(0);
      stop=1;
      started = 0;
      finished = 1;
    } else {
        if(ADC_Pm_Last < 300){
        speed=0;
        motor_pwm_write(0);
      } else if (ADC_Pm_Last < 600){
        speed=10;
        motor_pwm_write(10);
      } else if (ADC_Pm_Last < 2500){
        speed=30;
        motor_pwm_write(30);
      }else if (ADC_Pm_Last < 3500){
        speed=50;
        motor_pwm_write(50);
      } else if (ADC_Pm_Last < 3950){
        speed=60;
        motor_pwm_write(60);
      } else if (ADC_Pm_Last < 4050){
        speed=85;
        motor_pwm_write(85);
```

```
      } else{
        speed=100;
        motor_pwm_write(100);
      }
    }
  }
  if(mode==0){
    started = 0;
    if(strcmp(characters,"STATUS\r\n")==0){
      char buffer[200];
      uint32_t distance= calculateUSDistance();
      sprintf(buffer, "{\"distance\":%d,\"light_level_left\":%d,\"light_level_right
    \":%d,\"op_mode\":\"AUTO\"}\r\n", distance, LDR1_value, LDR2_value);
      HM10_SendCommand(buffer);
    }
    if(strcmp(characters,"FORWARD\r\n")==0){
      Motor_Forward();
      PWM_Cycle_Rate(1,PWM0);
      set_pwm_rates(100,0,100,0,speed);
      stop=0;
    }
    if(strcmp(characters,"BACK\r\n")==0){
      Motor_Backward();
      PWM_Cycle_Rate(1,PWM0);
      set_pwm_rates(0,100,0,100,speed);
      stop=0;
    }
    if(strcmp(characters,"LEFT\r\n")==0){
      Motor_Left();
      uint32_t local_rotation = rotations;
      PWM_Cycle_Rate(1000,PWM0);
      set_pwm_rates(50,50,0,0,speed);
      while((rotations - local_rotation) < 2){}
      set_pwm_rates(0,0,0,0,0);
      stop=1;
    }
    if(strcmp(characters,"RIGHT\r\n")==0){
      Motor_Right();
      uint32_t local_rotation = rotations;
      PWM_Cycle_Rate(1000,PWM0);
      set_pwm_rates(0,0,50,50,speed);
      while((rotations - local_rotation) < 2){}
      set_pwm_rates(0,0,0,0,0);
      stop=1;
    }
    if(strcmp(characters,"STOP\r\n")==0){
      speed=0;
      set_pwm_rates(0,0,0,0,0);
      stop=1;
    }
  }else{
    if(strcmp(characters,"START\r\n")==0){
      started = 1;
      finished = 0;
    }
    if(finished){
      HM10_SendCommand("FINISH\r\n");
      finished = 0;
    }
    if(started){
      uint32_t distance;
      Motor_Forward();
```

```
      PWM_Cycle_Rate(1,PWM0);
      set_pwm_rates(100,0,100,0,speed);
      if(ultrasonicSensorNewDataAvailable){
          distance = calculateUSDistance();
          if(distance<10){
            uint32_t local_rotation = rotations;
            Motor_Right();
            PWM_Cycle_Rate(1000,PWM0);
            set_pwm_rates(0,0,50,50,speed);
            while((rotations - local_rotation) < 1){}
            Motor_Forward();
            PWM_Cycle_Rate(1,PWM0);
            set_pwm_rates(100,0,100,0,speed);
            local_rotation = rotations;
            while((rotations - local_rotation) < 3){}
          }
          else if(distance<35){
            Motor_Forward();
            PWM_Cycle_Rate(1,PWM0);
            set_pwm_rates(100,0,100,0,speed);
          }
          else{
            uint32_t local_rotation = rotations;
            Motor_Left();
            PWM_Cycle_Rate(1000,PWM0);
            set_pwm_rates(50,50,0,0,speed);
            while((rotations - local_rotation) < 1){}
            Motor_Forward();
            PWM_Cycle_Rate(1,PWM0);
            set_pwm_rates(100,0,100,0,speed);
            local_rotation = rotations;
            while((rotations - local_rotation) < 3){}
          }
      }
    }
    if(strcmp(characters,"STATUS\r\n")==0){
      char buffer[200];
      uint32_t distance= calculateUSDistance();
      sprintf(buffer, "{\"distance\":%d,\"light_level_left\":%d,\"light_level_right
    \":%d,\"op_mode\":\"AUTO\"}\r\n", distance, LDR1_value, LDR2_value);
      HM10_SendCommand(buffer);
    }
    if(strcmp(characters,"STOP\r\n")==0){
      speed=0;
      set_pwm_rates(0,0,0,0,0);
      stop=1;
    }
  }
  memset(characters, 0, 255);
}
```

The main method calls the init() method and update function.

```
int main() {
    init();

    while(1) {
        update();
    }
}
```

8 Expense List

- Cables $\rightarrow 16TL$

9 References

Lab Templates - 2019