The PID Controller Line Follower Robot with STM32F103

Conference Paper · January 2024

CITATIONS
0

READS
1,396

3 authors, including:

Serhat Mizrak
Abdullah Gul University
2 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Furkan Durmuş
Abdullah Gul University
5 PUBLICATIONS **0** CITATIONS
SEE PROFILE

The PID Controller Line Follower Robot with STM32F103

Serhat Mızrak
Department of Electrical and Electronic Engineering,
Abdullah Gül University
Kayseri, Turkiye
serhat.mızrak@agu.edu.tr

Furkan Durmuş
Department of Electrical and Electronic Engineering,
Abdullah Gül University
Kayseri, Turkiye
furkan.durmus@agu.edu.tr

Emirhan Şahinoğlu
Department of Electrical and Electronic Engineering,
Abdullah Gül University
Kayseri, Turkiye
emirhan.sahinoglu@agu.edu.tr

***Abstract*— In this paper, presents the design, implementation, and performance analysis of a PID (Proportional-Integral-Derivative) line follower robot using the STM32F103C8T6A microcontroller. The PID line follower robot was established with examination of different gain. The line follower robot is a popular application in the field of robotics and automation, and its performance heavily depends on the PID controller gains. Therefore, this study investigates the effects of different PID gains on the robot's performance. Moreover, The line follower robot has built up utilization with stm32f103c8t6A, L298 motor driver, LF33ABV regulator, DC motor and Joystick.**

Keywords—The Line Follower Robot, ST32F103, PID Controller, L298 Motor, Joystick, L298 Motor Driver,

I. INTRODUCTION

This project is centred around the development of a Line Tracking robot, but with a unique twist. Rather than using conventional sensors, we are harnessing a distinct sensor known as the QTR08C. This sensor is renowned for its digital output, providing precise and reliable data for the robot's line tracking function. In addition to this innovative sensor, we have also integrated a joystick into the robot's design. This joystick serves as an interactive input mechanism, enabling the user to manually control the robot's direction and speed. This feature not only enhances user engagement but also offers a unique perspective for future projects, underscoring the versatility of manual input in robotic control.

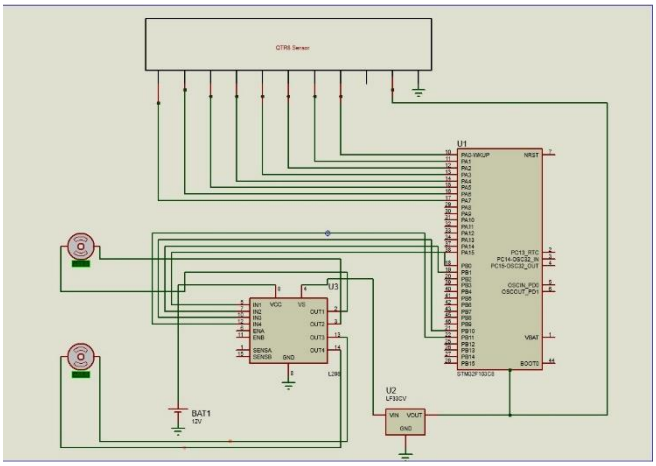
The L298 motor driver plays a crucial role in our robot's operation. This specific driver was chosen for its ability to provide the necessary power and control signals to effectively operate the robot's motors. With this driver, we can fine-tune the robot's movements with precision and reliability.

One of the unique aspects of this project is the use of a PID controller. By adjusting the motor speed and direction based on the PID controller's output, the robot can efficiently follow the line or respond to joystick input. This advanced control system ensures that our robot operates smoothly and responsively, enhancing its line tracking abilities. To ensure stability and reliability in operation, we've made a point to incorporate the LF33ABV regulator. This regulator is tasked with maintaining a steady voltage output for the microcontroller and other components. By doing so, it minimizes fluctuations and ensures consistent performance, which is vital for the robot's accurate and reliable operation.

.

II. DESIGN AND PROCEDURES

COMPONENTS



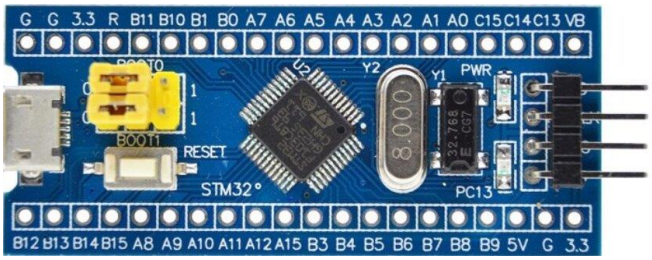
(***Figure 1.1****- circuit diagram on Proteus)*

a. Microcontroller (STM32F103C8)

STM32F103C8T6A, which uses an ARM Cortex-M3 based microcontroller and has a 32-bit Reduced Instruction Set Computing (RISC) architecture, is the board manufactured by STMicroelectronics. The STM32F1 family is used in many projects due to its ease of use and the benefits it provides. Operating at a maximum frequency of 72Mhz, it is very efficient in power consumption and can produce enough power for jobs used in many areas. Its memories are 20 KB SRAM and 64KB Flash memory. It offers a wide network as communication interfaces. It has Universal Synchronous/Asynchronous Receiver/Transmitter (USART), Universal Serial Bus (USB), Serial Peripheral Interface (SPI) communication interfaces. Thanks to these, it can communicate with the devices around it. It has General Purpose Input/Output (GPIO) pins and can be configured in various ways according to the project. It also has the ability to create 12-bit analogue-to-digital converter (ADC), timers, interrupts. It has an operating voltage between 2V and 3.6V, but it has low power consumption and energy saving.

There are many options for programming these cards. STM32CubeIDE and Keil are at the beginning of these.

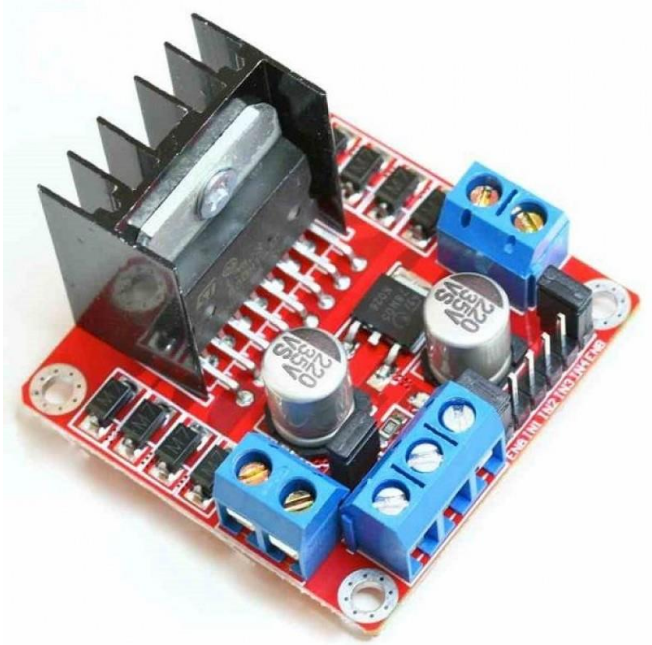
Since the ease of application is high, it can be used for wide purposes in many projects. These include home automation, medical devices, control systems and automotive sector.



(***Figure 1.2****- STM32F103C8)*

b. Motor Driver (L298N)

The L298 Motor Driver is a motor driver integration used in many robotic applications The L298 features a dual H-bridge configuration. This can be used to provide control of two independent DC motors. It offers the ability to independently control the speed and direction of each motor. In addition, the L298 can provide up to 2 Amps of continuous current and up to 3 Amps of peak current. This makes it possible to control larger and more powerful motors. The motor driver has screw terminals for connecting to the motors. This allows easy connection and disconnection of motor cables. Also, PWM signals are often used for control signals of motors connected to the L298. Finally, the L298 has built-in series diodes to accommodate back emf when controlling inductive loads of motors. This prevents the back emf effects that occur when the motors stop and reduces the risk of damaging other components on the circuit. In general, the L298 Motor Driver is widely used as a powerful and flexible motor driver integration. Its wide input voltage range, high power capability and dual H-bridge configuration make it an ideal choice for use in a variety of robotic projects.



(***Figure 1.3****- L298 Motor Driver)*

c. Reflectance Sensor (QTR-8RC)

The reflection sensor QTR-8RC is used in areas such as line tracing. Thanks to the diodes it contains, it emits an invisible light to the environment and can detect the light emitted to the environment with phototransistors. It consists of 8 sensors and is aligned in a certain order. Thus, it is a very suitable sensor for this project as it can follow a certain line and detect change on the line. It is possible to get digital output from each sensor of the sensor separately. The optimum detection distance is 2-3 mm, the maximum distance detection is 9.5 mm. The light sent from the sensors hits the opposite surface and the returning light is detected by the sensor. Since the differences in the surface will change the reflectivity of the surface, the light perceived by the sensor decreases. Thanks to these differences, the position of the thing to be tracked can be determined. In this project, it can be used for distance measurement and obstacle detection as well as line follower. In short, the usage area of this sensor is quite wide.

(Figure 1.4*- QTR8 RC)*

d. Joystick

Joysticks are used for many purposes; they are generally used in game arms in the entertainment industry. There are 2 10K potentiometers in the sensor. One of these potentiometers reads the movement on the X axis and the other potentiometer reads the movement on the Y axis. Since the resistance of the potentiometers will change with the movement of the joystick, it gives an analogue output to the output in proportion to it. Thanks to the values read from these outputs, the desired things can be done in the projects. Thanks to this module, it can be used to move anything in games, steer a car or move a robot.

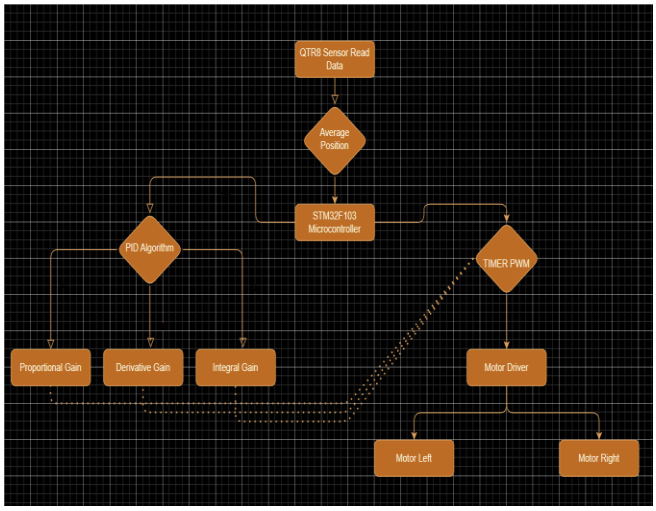
***(Figure 1.5**- Joystick)*

e. LF33ABV

The LF33ABV is a regulator integration and is used to provide voltage regulation in power supplies. With this regulator, the regulator is designed to keep the output voltage at a constant value. For example, it can provide a constant output voltage of 3.3V. This ensures safe and stable operation of other components. Another advantage of this regulator is that output voltage occurs when the input voltage of the regulator is lower than the output voltage. The LF33ABV provides a stable output voltage due to its low dropout voltagecharacteristic. characteristically the LF33ABV regulator is widely used in applications requiring voltage regulation in power supplies. Features such as wide input voltage range, low dropout voltage and thermal protection are important to ensure a stable and reliable power supply. The LF33ABV is a common regulator integration used to ensure accurate and stable operation of microcontrollers, sensors, and other components.

***(Figure 1.6**- LFV33ABV)*

III. SOFTWARE PART



***(Figure 2.1**- System Flow Chart)*

To complete this project, all code was written using Embedded register level C code. First, the QTR8 sensor configuration was done. The sensor datasheet was carefully analyzed and the necessary steps were implemented step by step. Then a data algorithm was developed to correctly retrieve the sensor data. After the algorithm was developed, PWM (Pulse Width Modulation) was generated using timers and the events of accelerating the two motors on the right and left were completed. The sensor data algorithm was then used to develop the PID algorithm. Based on the data from the PID block, PWM values at different powers were transferred to the motors with timers. Finally, the PWM values transferred to the two motors were transferred to the motors with the help of the motor driver and the robot was enabled to follow the line. Joystick was added later as an additional bonus party. When the joystick is added, if no command is given, the robot follows the line with normal PID, if any command comes from the joystick, the interrupt is activated and the robot turns into an RC controlled car and direction commands are given to the robot with the joystick. Now the register level code will be examined in detail by going line by line.

A. Delay Creation with Timer4

As a first step, timer delay functions, which will be used frequently in the later stages of the code, were created using TIM4. The best way to get precise results in delay generation is to use timers. For example, in sensor configurations, it is necessary to get precise and accurate results since operations are performed in millisecond and microsecond scales. In this timer 4 block, first the registers required for the TIM4 counter were coded with bare metal and their configurations were completed. After assigning the counter value to the 16-bit maximum value, enable operations were performed and the timer was expected to be set.

```
void TIM4Config (void)
{

  RCC->APB1ENR |= RCC_APB1ENR_TIM4EN; //ENABLING THE TIM4 BUS
  TIM4->PSC = 71;       //SETTING THE PRESCALER VALUE AS 72-1=71
  TIM4->ARR = 0xFFFF;     //SETTING THE TOP VALUE ARR TO ITS MAX
  TIM4->CR1 |= TIM_CR1_CEN;   //ENABLE THE TIMER4
  while(!(TIM4->SR & (TIM_SR_UIF)));  // WAITING THE TIMER TO BE SET
}
```

Then a delay_us function was created that counts down to the desired microsecond. The main purpose of this function is to count the TIM->CNT register up to the argument value entered into it and produce a precise delay.

```
void delay_us (uint16_t us)
{

  TIM4->CNT &= 0x0000; // RESETTING CNT REGISTER AT FIRST
  while(TIM4->CNT < us); // MICROSECOND DELAY
}
```

The delay_ms() code that generates a millisecond delay is based on calling delay_us (1000) function the number of times which is wanted to be generated.

```
/*
HERE, IN THE PROCESS OF CREATING THE MILISECOND DELAY,
THE MICROSECOND DELAY FUNCTION WAS CALLED HOW MANY TIMES
THE MS DELAY WANTED TO BE CREATED

5 MS DELAY WE WANT = 5 * delay_us(1000)

*/
void delay_ms (uint16_t ms)
{
  for(uint16_t i=0;i<ms;i++)
  {
    delay_us(1000); // 1 ms delay
  }
}
```

In this chapter, the simulation results and experimental results will be given for each block of the system as discussed in design and procedures chapter.

B. PLL Clock Configuration

Here on the STM32f103 board the system clock is switched to the PLL clock. The main logic is based on the variations of a control_flag bit. In short the code controls the system clock. If the system is using PLL clock, it activates the flag bit. It then makes the HSI clock the system clock and waits for it to be set. It deactivates the PLL and waits until it is ready. PLLMULL register settings are made as they should be and PLL is activated. Finally, the PLL is set as the main clock and the process is finished. Thus, a higher system frequency is achieved and it is planned to be used in this way in future processes.

```
    //ARRANGE THE PLLMULL REGISTERS ACCORDINGLY
      RCC->CFGR |= RCC_CFGR_PPRE1_2;
      RCC->CFGR &= ~RCC_CFGR_PLLMULL;
      RCC->CFGR |=  RCC_CFGR_PLLMULL9;

    // SET PLLON BIT AS 1
      RCC->CR |= RCC_CR_PLLON;

    // TIME FOR PLLRDY BIT TO BE SET
      while(!(RCC->CR & RCC_CR_PLLRDY));

    //ALTER THE SYSTEM CLOCK AS PLL
      RCC->CFGR |= RCC_CFGR_SW_PLL;

    // WAIT THE TIME TILL THE PLL IS SWITCHED
      while((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_PLL){}
    }
}
//---------------------------------------------------------------------
```

```
// CLOCK CONFIGURATION ---------------------------------------

void PLL_Clock_Initialization(void)
{
  int control_flag = 0;

    RCC->CFGR &= ~RCC_CFGR_SW;

   // CHECK WHETHER THE SYSTEM CLOCK IS PLL OR NOT
    if (RCC_CFGR_SWS_PLL && (control_flag==0))
    {
      // IF SO, SET THE control_flag TO 1
      control_flag = 1;
    }

      if (control_flag==1){
   // MAKE THE HSI AS THE SYSTEM CLOCK

      RCC->CFGR |= RCC_CFGR_SW_HSI;

   // WAIT TILL HSI IS SWITCHED
      RCC->CFGR &= ~RCC_CFGR_SW;
      while((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_HSI){}

    // SET THE PLLON BIT 0- MAKE IT DISABLED
      RCC->CR &= ~RCC_CR_PLLON;

    // WAIT TILL THE PLLRDY BIT IS CLEARED
      while((RCC->CR & RCC_CR_PLLRDY) == RCC_CR_PLLON){}
```

C. GPIO Initialization

At this stage, the GPIO initialization part was done. The main issue is to make PWM configurations of the timer blocks that will drive the motors. Here, the pins of the PWM signals to be sent to the motor drivers were set in STM32 and the first step of the motor movement phase was completed.

D. QTR8 Sensor Configuration

There are a few important details when configuring the QTR8 sensor. To activate this sensor, some steps given in the datasheet must be followed carefully. Sensors must first be set to 50 mhz output push pull. The second step is to make the GPIOA CRL pins 0. Then you need to make those bits high. After being high, after waiting for at least 10 us, the pins are set as alternate function output push-pull. After waiting

```
void GPIO_Initialization(void)
{
    // Enable port A clock
    RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;

    // Configure GPIOA
    GPIOA->CRL &= ~0xFFFFFFFF; // Clear all configuration bits for GPIOA

    // Enable port B clock
    RCC->APB2ENR |= RCC_APB2ENR_IOPBEN;

    // Configure GPIOB PB0, PB1, PB10, PB11 as alternate function, 50 MHz, push-pull
    GPIOB->CRL |= GPIO_CRL_MODE0 | GPIO_CRL_MODE1 | GPIO_CRL_MODE3; // Set pin 0, 1, 3 to 50 MHz output speed
    // Set pin 0, 1, 3 to alternate function, push-pull output
    GPIOB->CRL |= GPIO_CRL_CNF0_1 | GPIO_CRL_CNF1_0 | GPIO_CRL_CNF3_1;
    // Clear other configuration bits for pin 0, 1, 3
    GPIOB->CRL &= ~(GPIO_CRL_CNF0_0 | GPIO_CRL_CNF1_1 | GPIO_CRL_CNF3_0);
    GPIOB->CRH |= GPIO_CRH_MODE10 | GPIO_CRH_MODE11; // Set pin 10, 11 to 50 MHz output speed
    GPIOB->CRH |= GPIO_CRH_CNF10_1 | GPIO_CRH_CNF11_0; // Set pin 10, 11 to alternate function, push-pull output

    GPIOB->CRH &= ~(GPIO_CRH_CNF10_0 | GPIO_CRH_CNF11_1); // Clear other configuration bits for pin 10, 11
}
```

for another 6ms for debounce, the configuration is completed as described in the datasheet.

```
void QTR8_Sensor_Configuration(void)
{
    // Set GPIOD CRL register to 50 MHz output speed, push-pull
    GPIOD->CRL |= 0x33333333;

    // Configure GPIOA pins 0-3 as digital input with no pull-up/pull-down
    GPIOA->CRL &= ~(GPIO_CRL_CNF0 | GPIO_CRL_CNF1 | GPIO_CRL_CNF2 | GPIO_CRL_CNF3);

    // Configure GPIOA pins 4-7 as digital input with no pull-up/pull-down
    GPIOA->CRL &= ~(GPIO_CRL_CNF4 | GPIO_CRL_CNF5 | GPIO_CRL_CNF6 | GPIO_CRL_CNF7);

    // Set GPIOA ODR register to set pins 0-7 as high
    GPIOA->ODR |= 0x000000FF;

    delay_us(10); // Allow at least 10 microseconds for the sensor output to rise

    // Configure GPIOA pins 0-7 as alternate function, push-pull output
    GPIOA->CRL &= ~(3 << 28) & ~(3 << 24) & ~(3 << 20) & ~(3 << 16) & ~(3 << 12) & ~(3 << 8) & ~(3 << 4) & ~(3 << 0);
    GPIOA->CRL |= 0x88888888;

    delay_ms(6); // 6ms delay
}
```

E. Motor

The motor function takes two input that are left and right speed enabling to control two DC motor speed at the same time. The working principle of the motor function is quite important. The function controls motor speeds by checking the "left_speed" value. If it's less than 0, the right sensor receives "ARR_Base_Value0" and "-1ARR_Base_Value*left_speed", resetting the right motor's speed. If it's greater, the left motor's speed is reset, while the right motor's is adjusted. The same logic applies to the "right_speed" value, resetting the right motor's speed and resetting the left motor's speed.

```
void Motor(int left_speed, int right_speed)
{
    // Control the speed of the left motor
    if (left_speed < 0)
    {
        // The left motor needs to rotate in the reverse direction
        Right_Sensor(ARR_Base_Value * 0, -1 * ARR_Base_Value * left_speed);
    }
    else
    {
        // The left motor needs to rotate in the forward direction
        Right_Sensor(ARR_Base_Value * left_speed, ARR_Base_Value * 0);
    }

    // Control the speed of the right motor
    if (right_speed < 0)
    {
        // The right motor needs to rotate in the reverse direction
void PWM_Configuration (void)
{
 RCC->APB2ENR |= RCC_APB2ENR_IOPBEN | RCC_APB2ENR_AFIOEN;
 RCC->APB1ENR |= RCC_APB1ENR_TIM2EN | RCC_APB1ENR_TIM3EN;

 AFIO->MAPR |= AFIO_MAPR_TIM2_REMAP_FULLREMAP;


 TIM2->CCER |= TIM_CCER_CC4E | TIM_CCER_CC3E;
 TIM3->CCER |= TIM_CCER_CC4E | TIM_CCER_CC3E ;

 TIM3->CR1 |= TIM_CR1_ARPE;
 TIM2->CR1 |= TIM_CR1_ARPE;

 TIM2->CCMR2 |= TIM_CCMR2_OC4M_1 | TIM_CCMR2_OC4M_2 | TIM_CCMR2_OC4PE;
 TIM2->CCMR2 |= TIM_CCMR2_OC3M_1 | TIM_CCMR2_OC3M_2 | TIM_CCMR2_OC3PE;
 TIM3->CCMR2 |= TIM_CCMR2_OC4M_1 | TIM_CCMR2_OC4M_2 | TIM_CCMR2_OC4PE;
 TIM3->CCMR2 |= TIM_CCMR2_OC3M_1 | TIM_CCMR2_OC3M_2 | TIM_CCMR2_OC3PE;


 //PWM freq = Fclk/PSC/ARR  72MHz/1000
 //PWM Duty = CCRX/ARR

 TIM2->PSC = 72-1;
 TIM2->ARR = 1000;

 TIM3->PSC = 72-1;
 TIM3->ARR = 1000;

}
```

F. QTR8 Sensor Data Process

This code fragment contains a function that reads data from the QTR-8 sensor and calculates a position based on this data. The function reads the sensor data from the GPIOA->IDR source and calculates the number of active sensors and their respective positions by checking the states for each sensor. The calculated position is divided by the number of activated sensors to obtain the average position value. The function also reports the number of activated sensors and the status of the corner sensor. If the total number of activated sensors is 0, the variable Final_Case is incremented by one; otherwise, Final_Case is reset to zero. Finally, the calculated average position value is returned by the function.

```
int QTR8_Read_Sensor_Data()
{
  QTR8_Sensor_Configuration();


    int Current_Position = 0;
    int Sensors_Activated = 0;

  sensor_read_GPIO= GPIOA->IDR;

    if(((sensor_read_GPIO & (1<<0))==(1<<0)))
      {

        Current_Position+=1000;
        Sensors_Activated++;
        Corner_Sensor=1;
      }
```

G. PWM_Configuration

The PWM configuration phase involves accessing the APB2ENR and APB1ENR registers on the RCC module to enable clock signals for GPIOB and AFIO. It also allows a full remap of TIM2 using the AFIO->MAPR register. The 4th and 3rd capture/compare channels are enabled by activating CC4E and CC3E bits in the CCER registers of the TIM2 and TIM3 timer units. The ARPE bits in the CR1 registers indicate automatic reload operation. The 4th and 3rd output compare channels are specified by enabling OC4M and OC3M bits in the CCMR2 registers. The PSC registers determine the ratio of timer frequency to master clock frequency. In this way, two different channels were opened for the two motors and the control mechanisms were made more optimal.

H. Error Calculation

For the PID setup, it has to evaluate the errors that the car suffers when driving on the line. The sensor has a reference value of 4500 when driving straight on the line. Every unit deviation from this reference value is recorded as an error. The sensor data function continuously sends the average position value of the sensor according to the number of active sensors and the assigned sum value of 1000. The result of

subtracting this current position value from the reference value is the error. It is necessary to take 10 of these errors and store them in an error matrix. The errors in this error matrix can then be summed up to give a total error, which is continuously updated as the robot moves and turns left or right. Failure to do so can cause stabilization errors in the robot, which can cause it to not follow the line correctly.

İ. Sharp Turning

The sharp turn function is a vital function that should be included especially in line following robots. If this function is not included, when the robot finishes the line, it stops at the end and does not go any further. This is not a situation that can be ignored in terms of functionality. Especially in Z-style turns, the robot may leave the line and not follow the correct track. This is why the sharp turn function is needed. In this way, for example, when the robot reaches the end of the line, it will turn 360 degrees and continue its function at the first place where the line is diagnosed. Another important feature is that it allows the robot to find the path in sharp turns. One of the biggest problems encountered in line follower robot designs is that the robot follows the path correctly but does not continue when it comes to sharp turn points, on the contrary, it returns back the way it came. The solution to this common problem is to implem The sharp turn function is a vital function that should be included especially in line following robots. If this function is not included, when the robot finishes the line, it stops at the end and does not go any further. This is not a situation that can be ignored in terms of functionality. Especially in Z-style turns, the robot may leave the line and not follow the correct track. This is why the sharp turn function is needed. In this way, for example, when the robot reaches the end of the line, it will turn 360 degrees and continue its function at the first place where the line is diagnosed. Another important feature is that it allows the robot to find the path in sharp turns. One of the biggest problems encountered in line follower robot designs is that the robot follows the path correctly but does not continue when it comes to sharp turn points, on the contrary, it returns back the way it came. The solution to this common problem is to implement the sharp turn function correctly. ent the sharp turn function correctly.

J. Joystick

```
void joystick_init(void)
{
  // GPIO port B clock enable
  RCC->APB2ENR |= RCC_APB2ENR_IOPBEN;

  // Configure joystick GPIO pins
  GPIOB->CRH &= ~(0xFFFFFF00);
  GPIOB->CRH |= 0x00000088;
}
```

Void joystick_init(void) function, pins 8 and 9 of the CRH part of the GPIOB port are set by enabling Port B.

```
void ReadJoystickValues(void) {
    joyXValue = (GPIOB->IDR >> JOY_X_PIN) & 0x01;
    joyYValue = (GPIOB->IDR >> JOY_Y_PIN) & 0x01;
}
```

In this section of this code, the X and Y axes on the joystick are defined.

```
void SetMotorSpeed(void) {
    if (joyXValue == 0 && joyYValue == 1) {
        // Only move to the left
        motorspeedl = 10;
        motorspeedr = 50;
        motor_control(motorspeedl, motorspeedr);
    } else if (joyXValue == 1 && joyYValue == 0) {
        // Only move to the right
        motorspeedl = 50;
        motorspeedr = 25;
        motor_control(motorspeedl, motorspeedr);
    } else if (joyXValue == 0 && joyYValue == 0) {
        // No movement, stop the motors
        motorspeedl = 0;
        motorspeedr = 0;
        motor_control(motorspeedl, motorspeedr);
    } else if (joyXValue == 1 && joyYValue == 1) {
        // Move forward or backward
        motorspeedl = 50;
        motorspeedr = 50;
        motor_control(motorspeedl, motorspeedr);
    }
}
```

This code represents a function used to set a motor speed. The function determines the motor speed based on the values of two variables, joyXValue and joyYValue. The first if block checks the case where joyXValue is 0 and joyYValue is 1, in which case the motor is assigned different speed values to move to the left. The second if block checks the case where joyXValue is 1 and joyYValue is 0, in which case the motor is assigned different speed values to move to the right. The third if block checks the case where joyXValue and joyYValue are both 0, in which case the motors are stopped. Finally, the fourth if block checks the case where joyXValue is 1 and joyYValue is 1, in which case the motor is assigned the same speed values to move forward or reverse. In this way, depending on joyXValue and joyYValue, the motor speed is determined, and the corresponding speed values are transmitted to the motors by calling the motor_control function.

```
void interruptConfig(void) {
    RCC->APB2ENR |= RCC_APB2ENR_AFIOEN;

    AFIO->EXTICR[2] &= ~(AFIO_EXTICR3_EXTI8 | AFIO_EXTICR3_EXTI9); // EXTI8 ve EXTI9 için GPIOB seçimini temizle
    AFIO->EXTICR[2] |= AFIO_EXTICR3_EXTI8_PB | AFIO_EXTICR3_EXTI9_PB; // EXTI8 ve EXTI9 için GPIOB seçimi yap

    EXTI->RTSR |= EXTI_RTSR_TR8 | EXTI_RTSR_TR9; // EXTI8 ve EXTI9 tetikleme kenarlarını yükselen kenar olarak ayarla
    EXTI->FTSR &= ~(EXTI_FTSR_TR8 | EXTI_FTSR_TR9); // EXTI8 ve EXTI9 tetikleme kenarlarını düşen kenar olarak ayarla

    EXTI->IMR |= EXTI_IMR_MR8 | EXTI_IMR_MR9; // EXTI8 ve EXTI9 kesmelerini etkinlestir

    NVIC_SetPriority(EXTI9_5_IRQn, -5); // EXTI9_5 kesmesinin önceliğini ayarla
    NVIC_EnableIRQ(EXTI9_5_IRQn); // EXTI9_5 kesmesini NVIC'de etkinlestir
}
```

```
void EXTI9_5_IRQHandler(void) {
    if (EXTI->PR & EXTI_PR_PR8) {
        EXTI->PR |= EXTI_PR_PR8; // EXTI8 kesme istegini temizle

        // Joystick degerlerini oku
        ReadJoystickValues();

        // Gerekli islemleri gerçeklestir
        if (joyXValue == 0 && joyYValue == 0) {
            joystickPressed = 0; // Joystick basilmadi
        } else {
            joystickPressed = 1; // Joystick basildi
        }
    }
```

The code configures an interrupt for EXTI9_5_IRQn, an external interrupt on lines 9 to 5. It enables the AFIO clock, sets the GPIOB port selection for

EXTI8 and EXTI9 and enables rising edge triggering while disabling falling edge triggering for EXTI8 and EXTI9. The interrupt mask is then enabled for EXTI8 and EXTI9. The priority of the EXTI9_5_IRQn interrupt is set and the interrupt is enabled in the NVIC. The code defines the interrupt handler function for EXTI9_5_IRQn and checks if the EXTI8 interrupt pending flag is set. If so, the flag is cleared, and the joystick values are read. Based on these values the necessary actions are performed and the joystickPressed variable is set accordingly.

```
while(1)
{
  ReadJoystickValues();
  if (joyXValue == 0 && joyYValue == 0) {
      joystickPressed = 0; // Joystick basilmadi
       }
  if (joystickPressed == 0) {
          PID_control();
      } else {
          SetMotorSpeed();
      }
```

After calling the ReadJoystickValues() function to read the joystick values, the code checks if both the joyXValue and joyYValue are equal to 0. If this condition is true, it means that the joystick is not pressed, so the joystickPressed variable is set to 0, indicating that the joystick is not pressed.
Next, the code checks if joystickPressed is equal to 0. If this condition is true, it means that the joystick is not pressed. In this case, the code calls the PID_control() function, which likely performs some control algorithm or logic.
On the other hand, if the joystickPressed variable is not equal to 0, it means that the joystick is pressed. In this case, the code calls the SetMotorSpeed() function, which is responsible for setting the motor speed. The exact implementation of this function would depend on the specific requirements of the project or system.Overall, this code snippet reads joystick values, checks if the joystick is pressed, and performs different actions based on the joystick's state. If the joystick is not pressed, a control algorithm (likely PID control) is executed. If the joystick is pressed, the motor speed is set accordingly.

IV. ROBOT MOVEMENT

a. Error Calculation

Error calculation is a step to adjust the motor speeds individually in order to steer the car in the desired direction using the data between the desired position and the sensor. Since there are 8 transmitters and receivers on the sensor, it is desired that the line to be followed is between the 4th and 5th sensors. Each detector on the QTR-8RC sensor is analogue connected to the STM32F103C8T6 and reads. Thus, the microprocessor instantly determines the position of the car. Firstly, the target position is subtracted from the position of the car. This is called error. Then the error values obtained are summed to a variable. Then a value is obtained by subtracting the current error value from the previously obtained error value. These parameters are multiplied by certain coefficients and the results are obtained. PID value is obtained by summing these results. This PID value will be used to guide the vehicle to the target position. Thanks to this, it is ensured that the line is followed by adjusting the speed and direction of the motors.

b. PID CONTROL

Proportional-Integral-Derivative (PID) is used for feedback in a system. Thanks to PID, it enables the system to reach the desired target by feeback. Due to this, it is more likely to be used in real-time control systems. To calculate PID, Proportional, Integral and Derivative Gains should be calculated first. The Kp, Ki and Kd values to be used during these calculations can be reassigned according to the results obtained by testing the vehicle on the line. Thus, the system reaches the desired target more stabilised. Then PID control is obtained by summing these gains. Proportinal Gain affects the performance of the feedback system. Integral Gain helps to stabilise the feedback system by adding the current and previous errors. Derivative Gain adjusts the stability of the feedback system according to the rate of change of the system error.

c. Proportional Term (P)

Proportional parameter is one of the three parameters in the PID control system. P Gain is directly affected by the difference between the current value of the vehicle and the target value. As it can be understood mathematically, if the current value is less than the target value, P Gain becomes negative. If the current value is greater than the target value, the result of the process is positive and P Gain is positive. This difference is called error.

$$P\ Gain = error * Kp$$

can be calculated by the formula.

The higher the gain, the faster the control system reacts to correct the error. Due to this, oscillations occur in the system due to overreaction, but the rise time is reduced. As P Gain decreases, the control system tends to correct the error more slowly. Slower corrections reduce the oscillation in the system but increase the rise time. Due to these factors, it cannot be said that the P control system alone is a stable system. and it shows that the value of P Gain should be chosen in an important way. If the control system is desired to be stable, the other parameters, Integral and Derivative Gain, must also be present in the control system.

d. Integral Term (I)

Integral parameter is one of the three parameters in the PID control system. The sum of the error values of the vehicle travelling on the line affects the Integral Gain. In other words, the system makes the vehicle more stable on the line by summing the error values it has. Therefore, the I parameter should be reset appropriately in the code. If zeroing is not done, even if the system is stable, the I value will never be 0, so it may influence reducing the stability of the system.

$$I = \sum(error) * Ki$$

can be found with the formula.

The I value in the formula represents the I value that is important for PID. Since it tries to keep the system stable accordingly by collecting past errors, it prevents this if the vehicle is to the left or right of the line for a while on the line. Thus, the system gets closer to being stable. In contrast to Proportional Gain, Integral gain has a slower effect on the control system. If the error values collected are getting larger and larger, the I parameter increases and increases the output of the control system accordingly. Excessive increase or incorrect selection of the I parameter may reduce the stability of the system. To increase the stability of the system, the I parameter must be selected correctly. Although the control system created with Proportional and Integral mostly stabilises the vehicle system, it is still not fully stabilised. Therefore, the last parameter of PID, Derivative, should be added.

e. Derivative Term (D)

Derivative parameter is one of the three parameters in the PID control system. When the vehicle is traveling on the line, sudden changes are experienced on the sensor at sudden turns. Derivative parameter sensitizes the system to these sudden changes on the system. Accordingly, the Derivative parameter measures the rate of change of the current error and adjusts the system to produce output accordingly. The derivative is calculated to detect these sudden changes. Hence the name of the parameter. The parameter D is calculated as the negative rate of change of the current error with respect to time. Thus, the derivative measures the rate of change of the error and provides negative feedback to the system. As the rate of change of the current error increases, the value of the derivative will also increase, causing the D parameter to increase. With increasing D parameter, the control output of the system is reduced and this sudden change is tried to be prevented. Thus, it ensures that the system is stable despite sudden changes.

$$D = (d(error) / dt) * Kd$$

is calculated by the formula.

The Kd value is a constant number for the Derivative parameter. The system can be stabilized by changing the value of Kd in the direction of the user. If the Kd value is increased too much, it causes an increase in the D parameter. Due to this, the system becomes more sensitive to sudden changes and causes an increase in noise in the system. If the Kd value is decreased too much, it causes a decrease in the D parameter. Due to this, the system becomes less sensitive to sudden changes and the system responds to sudden changes more slowly and unstably. Due to these factors, choosing the correct value of Kd is a major factor for the stability of the system.With the Proportional parameter, the system responds to the magnitude of the error in the system, with the Integral parameter, the system accumulates previous errors and with the Derivative parameter, the system responds to sudden changes in the line in a better and more stable way. If the control system is created using these three parameters, the system will be more stable and will be able to respond more sensitively to changes.

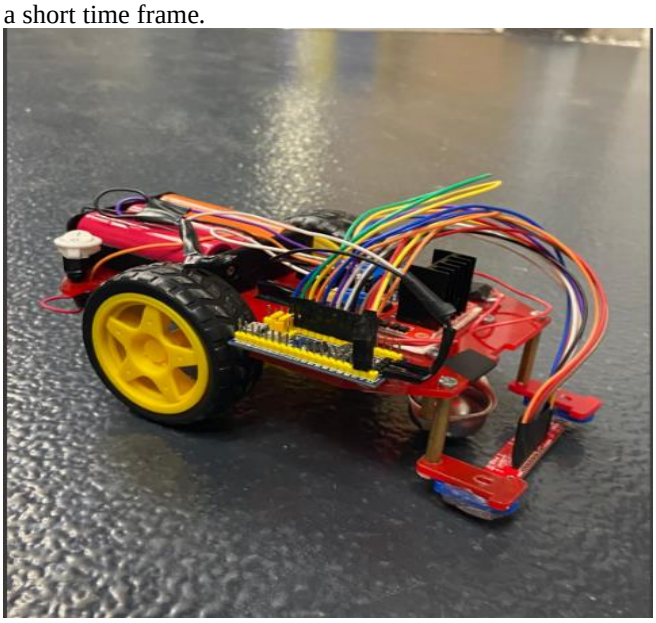
V. RESULT & DISCUSSION

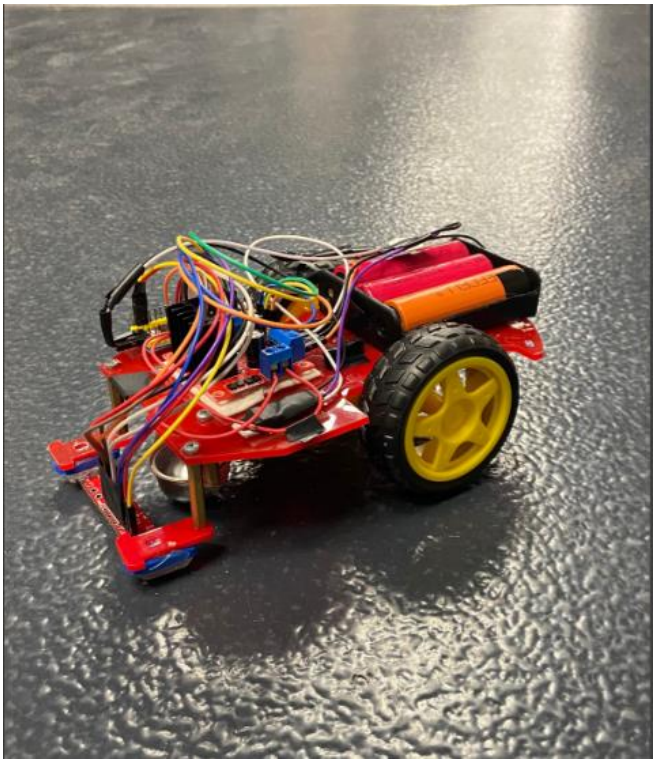
In this project, QTR8C and STM32F103 register based PID Controller Line Follower robot is designed on Keil U Vision 5. While designing this project, PWM pins were selected according to the pins used as mentioned above and PWM values were selected according to the desired frequency and the Timer used for PWM was selected according to these pins. The desired frequency was selected as 1 Mhz and the PSC value was set to 71. In addition, delays were realized with the help of timers. Thanks to the Timer Delays, the QTR8RC sensor was provided with the delay time on the datasheet thanks to precise delays. The PWMs for the motors are not provided via a single PWM, so two separate PWMs and timers are used. Since the QTR8C sensor is an IR sensor, the working phase was controlled through the camera (camera without infrared filter). The average position values taken from the sensor are used to calculate the error tracking value and P, I, D parameters as given above. In addition, the Joystick module was added to the project, allowing the user to manually control the project at any time. The Joystick module added to the project did not work together with the PID control and was therefore only set as manual control. In addition, the joystick button used to turn the car back was used to slow down the car. For those who want to develop this project in the future, it is strongly recommended to develop this project on original cards. Using original cards saves time and money for users

VI. CONCLUSION

In this project, a line following robot was designed in embedded C language using QTR8 IR sensor, L298N motor driver, and 2 DC motors. No library was included in the coding, on the contrary, all codes were written at the most basic level, that is, at the register level. The design followed the line correctly. More than one problem was encountered during the completion of the project. The first problem was to get the sensor data correctly. As it is known, IR sensors are sensors that produce values from the reflection of light. Optimizing the data received from these sensors and eliminating the debounce variable during the exchange were some of the sensor problems. Another sensor problem was to write the algorithm that generates the average value with the received data. After optimizing the sensor data, developing the PID algorithm was one of the most challenging tasks. It was to process the PID according to the current error values coming in continuously and rotate the motors in the direction they should be. The last challenging part is the sharp turn events. During the sharp turn optimization phase, the robot went off the path more than once and exhibited unexpected movements. As a result of the mishaps such as returning from the path it came from before completing the path, the robot was optimized and provided a suitable working environment for the desired functions. To briefly summarize what has been learned in this project, microcontroller coding at the register level, DC motor driving, IR sensor data processing and PID optimization algorithm creation. In addition, important topics such as the use of different peripherals, timer, pwm in embedded C language were mastered in detail. In order to move the project to better points, the first item can be said to use a better quality IR sensor with more optimal data. As another feature, using DC motors with higher efficiency in terms of efficiency can make the system operation more

effective. Moreover, additional features can be added to make the robot more beautiful and effective. As a result, the robot followed the specified line as it should, provided system optimization while running and completed the drawn path in a short time frame.





REFERENCES

[1] Gomes, M. V., Bassora, L. A., Morandin, O., & Vivaldini, K. C. T. (2016, November). PID control applied on a line-follower AGV using a RGB camera. In *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)* (pp. 194-198). IEEE.

[2] Jibrail, S. F., & Maharana, R. (2013). *PID Control of Line Followers* (Doctoral dissertation).

[3] Oguten, S., & Kabas, B. (2021). PID Controller Optimization for Low-cost Line Follower Robots. *arXiv preprint arXiv:2111.04149*.