STM32F446RE based Obstacle Avoiding Car

Mudit Goyal
The LNM Institute of Information Technology
24UEC278@lnmiit.ac.in

Abstract

This project details the design and implementation of an obstacle-avoiding car. The system is built upon an STM32F446RE microcontroller, which serves as the central processing unit. An HC-SR04 ultrasonic sensor is used for real-time distance measurement to detect obstacles in the robot's path. The bot's core logic, implemented in bare-metal C, processes this distance data to control the motors. When an obstacle is detected, that is the distance to obstacle is less than 10cm, the car automatically stops, checks distance on the left and right of the car, and then resumes forward motion in the direction that has more open space. The system provides multi-level user feedback through three-color LEDs (Red, Yellow, Green) indicating proximity, and a variable-frequency buzzer that beeps faster as the robot gets closer to an object. All distance measurements and system status updates are transmitted to a PC via UART for debugging and the data is printed on a Serial Monitor. A user-controlled start/stop button, implemented using an external interrupt on PC13, allows for toggling the system's active state.

1.1 Introduction

- This project aims to develop a car capable of navigating an environment by detecting and avoiding obstacles. Such systems are foundational in the field of robotics and have applications ranging from automated guided vehicles (AGVs) in industrial settings to autonomous vacuum cleaners and exploratory rovers.

- The core of this system is an STM32F446RE microcontroller. Its selection is based on its robust peripheral set, including GPIO, timers (for PWM and timing), UART, and external interrupts, all of which are used in this project.

- The main objective is to create a system where the car:
- **Primary Objective:** To create a system where the car can achieve forward motion without collision by:
    - Continuously sensing its environment.
    - Processing the sensory data to make a decision.
    - Acting upon that decision.
- **Secondary Objective:** To implement a robust user interface, including:
    - A user interrupt-driven button to start and stop the robot's operation.
    - Clear visual and audible feedback mechanisms

1.1.1 Theory

How Interrupts Work

A key feature of any modern MCU is its interrupt system. An interrupt is a hardware-driven mechanism that allows the MCU to respond immediately to urgent events.

- Polling: Without interrupts, the MCU would have to use "polling." This means it would waste most of its time in a loop, repeatedly asking, "Is the button pressed yet? Is the button pressed yet? Is the button pressed yet?" This is extremely inefficient.
- Interrupts: With an interrupt, the MCU can focus on its main job (like driving the motors). When an event occurs (like a user pressing the stop button), the button sends an electrical signal that "interrupts" the processor. The processor immediately pauses its main task, jumps to a special, high-priority function (an Interrupt Service Routine, or ISR) to handle the event (e.g., "stop all motors"), and then seamlessly resumes its main task exactly where it left off.

H-Bridge

- An H-Bridge is an electronic circuit named for its characteristic "H" shaped schematic. Its primary function is to act as a high-power switching interface, allowing a low-power control signal (e.g., from a microcontroller) to manage a high-power load, such as a DC motor.
- The circuit's main purpose is to provide bidirectional control. It uses four electronic switches in a bridge.
- To spin the motor in one direction, two diagonal switches are closed, allowing current to flow through the motor. To reverse the motor, those switches are opened, and the other two diagonal switches are closed. This reverses the current's polarity through the motor, thus reversing its direction.
- Furthermore, the speed of the motor can be controlled by applying a Pulse Width Modulation (PWM) signal to the bridge's enable inputs. By varying the signal's duty cycle, the average voltage delivered to the motor is adjusted, thereby controlling its rotational speed. The H-Bridge also allows for braking (by shorting the motor terminals) and coasting (by disconnecting the motor entirely).

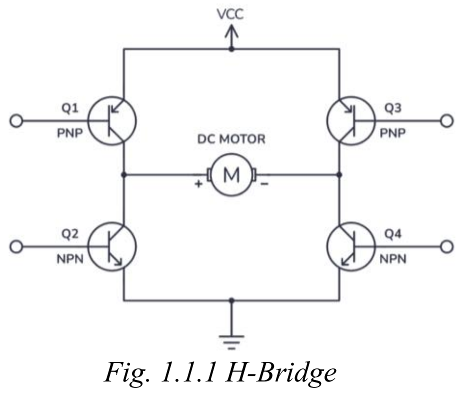


Fig. 1.1.1 H-Bridge

UART Protocol

For debugging and monitoring, the system employs the UART (Universal Asynchronous Receiver-Transmitter) protocol. This is a robust serial communication standard used to send data from the MCU to a host PC.

- Asynchronous: It is "asynchronous" because it does not use a shared clock signal. Instead, the transmitter (robot) and receiver (PC) must be pre-configured to operate at the same speed, known as the baud rate (e.g., 115200 bits per second).

- Transmission: To send data, the microcontroller's UART peripheral takes a byte (one character) and transmits it one bit at a time over a single TX (transmit) wire.
- Data Framing: The UART hardware automatically adds a "start bit" before the byte and a "stop bit" after. This "framing" allows the receiving device to know exactly when a new byte begins and ends, ensuring the data is read correctly.
- Usefulness: This allows the bot to send a continuous stream of text data (like "Distance: 15.2 cm") to a computer, which is invaluable for a human operator to debug, monitor, and verify the system's behaviour.

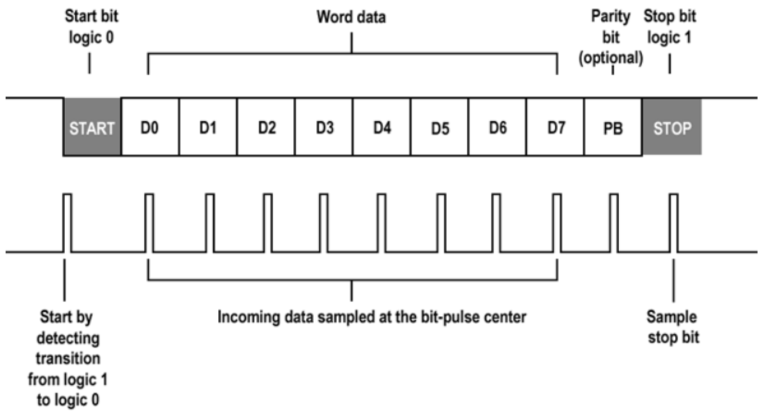


Fig. 1.1.3 UART Frame

SWAT Analysis

Analysis	**Details**
Strengths	Low-cost components, modular C-code drivers, real-time feedback, robust interrupt-driven control.
Weaknesses	Single forward-facing sensor creates blind spots.
Opportunities	Addition of more sensors (sensor fusion), implementation of more complex navigation algorithms, integration of wireless control (e.g., Bluetooth)
Threats	Inaccuracy of ultrasonic sensor on certain surfaces (e.g., soft, angled); obstacles in blind spots.

Table 1.1.1 SWAT Analysis

1.2 Methodology/Procedure

1.2.1 List of Components

- STM32F446RE MCU

- HC-SR04 Ultrasonic Distance Sensor
- Two DC motors
- H-bridge motor driver (L298N)
- 1x Red LED, 1x Yellow LED, 1x Green LED
- 1x Piezo Buzzer
- Batteries
- Connecting wires
- Breadboard
- Metal Chasis

1.2.2 Detailed Description of Components

1) STM32F446RE

The "brain" of this bot is the STM32F446RE microcontroller (MCU). This microcontroller is made by STMicroelectronics. It contains not just a central processor, but also all the necessary components for a complete computing system:

- Core: It is built around a high-performance ARM Cortex-M4 processor core, which includes a Floating-Point Unit (FPU) for fast mathematical calculations (essential for signal processing and control algorithms).
- Memory: It has on-chip Flash memory (to store the program code) and SRAM (to store variables and temporary data while running).

Peripherals: It is packed with a rich set of hardware "peripherals." These are specialized blocks that handle specific tasks without burdening the main processor. This includes timers (for generating PWM signals or precise timing), communication interfaces (like UART, SPI, I2C), and GPIO pins (General Purpose Input/Output) to read buttons and control LEDs.

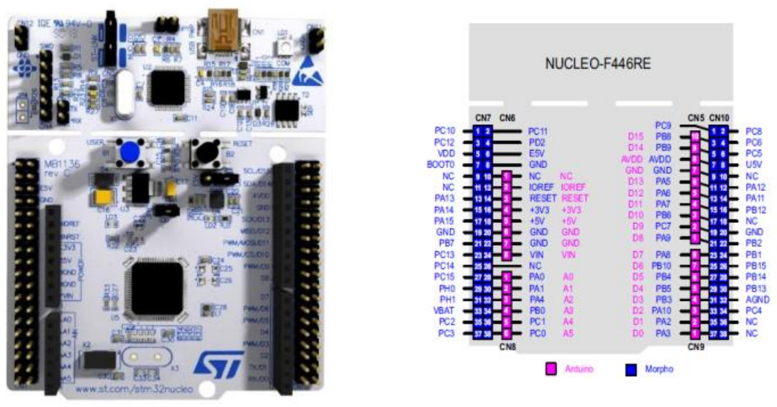


Fig. 1.2.1 Nucleo-F446RE Board and Pin Arrangements

2) HC-SR04 Sensor

The car detects obstacle around it using ultrasonic sensor (HCSR04), which operates on the principle of sonar (sound navigation and ranging).

- microcontroller first initiates a measurement by sending a short digital "trigger" pulse to the sensor.

- In response, the sensor module emits a brief burst of ultrasonic sound waves (e.g., at 40kHz), which are inaudible to humans.
- Simultaneously, the sensor raises its "ECHO" pin to a HIGH logic level.
- These sound waves travel outward, and if they hit an obstacle, they bounce back toward the sensor.
- When the sensor's receiver detects this returning echo, it immediately sets the "ECHO" pin back to a LOW logic level.
- The microcontroller's job is to precisely measure the time (the "pulse width") during which the ECHO pin was HIGH.
- This measured time represents the round-trip travel time of the sound—from the sensor to the object and back again.
- Since the speed of sound in air is a known constant (approximately 343 meters per second), the microcontroller can easily calculate the distance to the object. The formula is, in effect, Distance = (Pulse_Time × Speed_of_Sound) / 2.

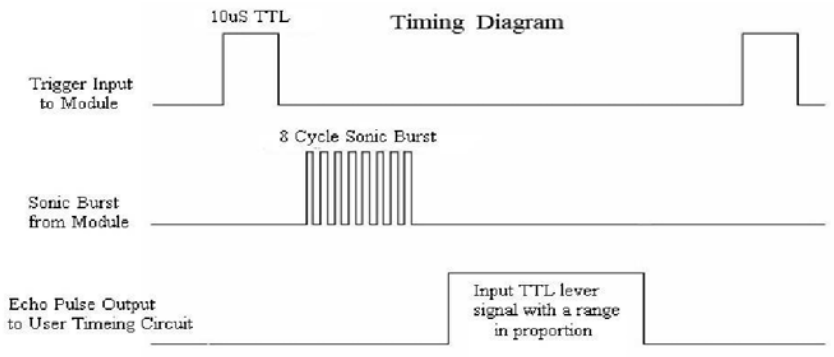

10uS TTL
Timing Diagram
Trigger Input to Module
8 Cycle Sonic Burst
Sonic Burst from Module
Echo Pulse Output to User Timeing Circuit
Input TTL lever signal with a range in proportion

Fig. 1.2.2 Timing Diagram of HCSR04 Sensor

3) **DC Motors**

DC (Direct Current) motors are actuators that convert DC electrical energy into mechanical energy in the form of rotation. In the project, it used to drive the wheels, with one motor controlling the left side and the other controlling the right.

Fig. 1.2.3 DC Motor

4) **H-bridge Motor Driver (L298N)**

The L298N is a dual H-bridge motor driver module. Its primary function is to act as an interface between a low-power controller (like a microcontroller) and the high-power DC motors. The H-bridge circuit allows it to control both the speed of a motor (by varying the voltage, using PWM) and its direction (by reversing the polarity of the current). This module can control two DC motors independently.

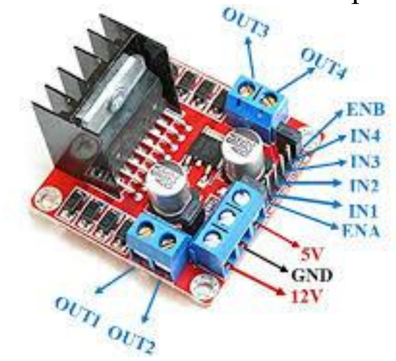


Fig. 1.2.4 L298N

5) **LEDs (Red, Yellow, Green)**

LEDs (Light Emitting Diodes) are semiconductor devices that emit light when an electric current passes through them. In this project, they serve as visual indicators to display the system's status. For example:

- Green LED: Distance > 30cm.
- Yellow LED: Distance > 10cm & < 30cm
- Red LED: Distance < 10cm.

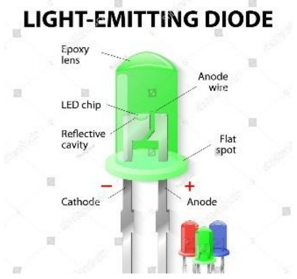


Fig. 1.2.5 LED

6) **Piezo Buzzer**

A piezo buzzer is an audible output device that generates sound using the piezoelectric effect. When a voltage is applied, a piezoelectric element inside it vibrates, producing a tone. It is used to provide audible alerts.

Fig. 1.2.6 Piezo Buzzer

7) **Batteries**

The batteries serve as the portable power source for the entire system. They store chemical energy and convert it to DC electrical energy, supplying the necessary voltage and current to power both the low-power logic components (like the controller) and the high-power components (like the L298N driver and DC motors).

8) **Connecting Wires**

These are electrical conductors (such as jumper wires) used to create the electrical circuit. They carry power from the battery and transmit control signals between all components, such as from the MCU to the L298N, and from the L298N to the motors.

9) **Breadboard**

A breadboard is a prototyping tool used for building and testing temporary electronic circuits without needing to solder. Its internal grid of spring-loaded contacts allows components and wires to be inserted and connected easily, making it ideal for development and testing new circuit designs.

10) **Metal Chassis**

The chassis is the structural frame of the project. It provides a rigid base for mounting all the components (motors, batteries, driver board, and breadboard).

1.2.4 Hardware Connections

- HC-SR04 Sensor:
  - TRIG Pin: PA1
  - ECHO Pin: PA0

- UART2 (Debug)(Internal to Nucleo Board):
  - TX Pin: PA2

- LED Indicators:
  - Red LED: PA5
  - Yellow LED: PA6
  - Green LED: PA7
- Motor Driver:
  - Left Motor (IN1, IN2): PB8, PB9
  - Right Motor (IN3, IN4): PB10, PB4

- Buzzer (PWM):
  - TIM4_CH1: PB6

- User Button (Interrupt)(Internal to Nucleo Board):
  - EXTI Pin: PC13

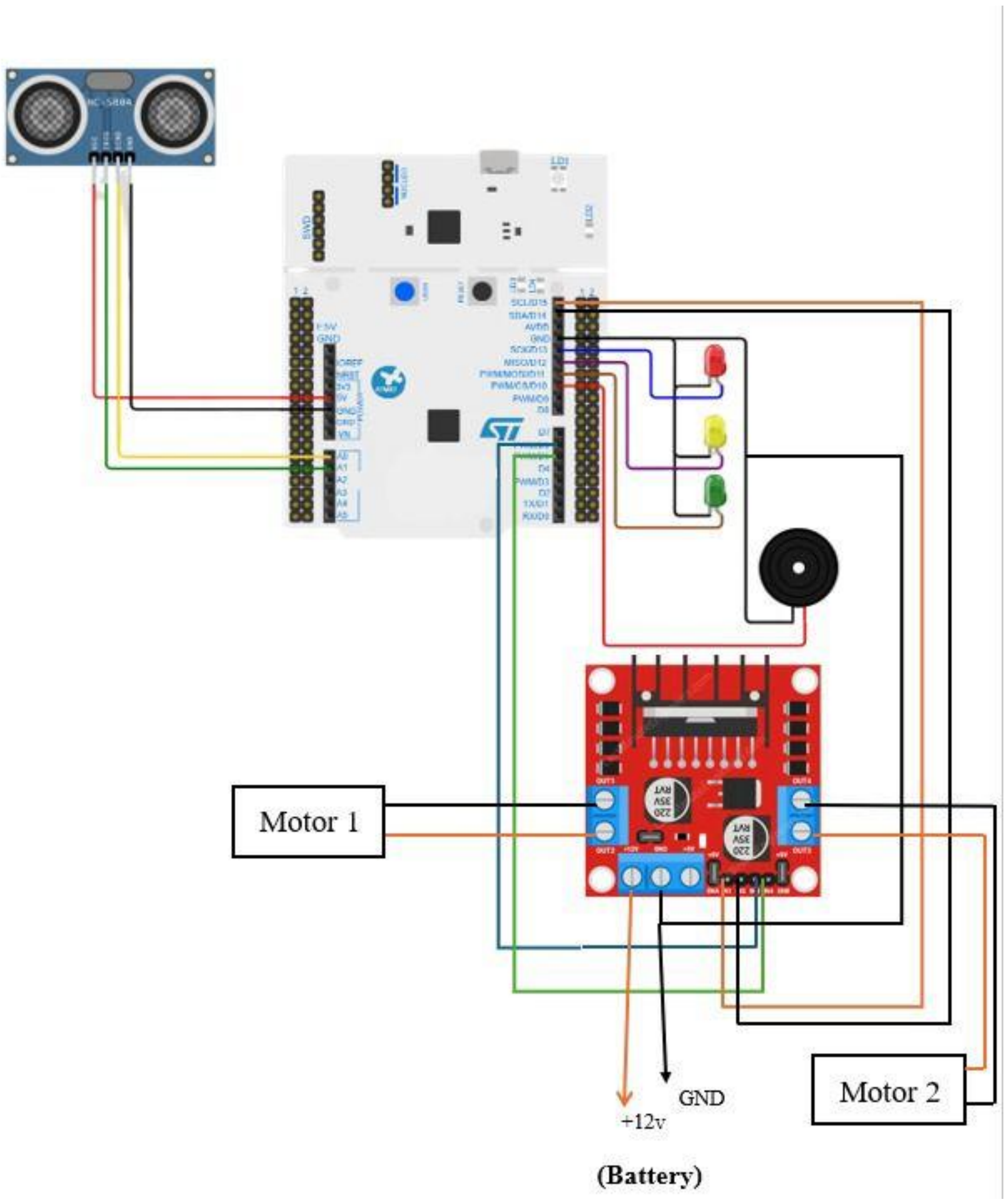


Fig. 1.2.7 Circuit Diagram

1.2.4 Codes

main.c

```
 #include "stm32f4xx.h"
#include "hcsr04.h"
#include "uart.h"
#include "timing.h"
#include "buzz.h"
#include "int.h"
#include "motor.h"
#include <stdio.h>

// Define the duration of each beep in milliseconds
#define BEEP_DURATION_MS 50
#define STOP_DISTANCE_CM 10  // Stop motor when distance < 10cm (1000 in cm*100)
#define TURN_DURATION_MS 800 // Turn for 800ms
#define SCAN_DELAY_MS 300    // Delay after turning to stabilize before measuring

// Global flag to control system state
volatile uint8_t system_active = 0;  // 0 = stopped, 1 = running

// Function to scan and decide direction
void scan_and_decide_direction(void)
{
    int32_t left_distance = 0;
    int32_t right_distance = 0;
    int32_t front_distance = 0;

    printf("Obstacle detected! Starting scan...\r\n");

    // Stop the motor
    motor_stop();
      // Stop for 500ms

    // Turn left and measure
    printf("Scanning left...\r\n");
    motor_turn_left();
    delay_us(TURN_DURATION_MS * 1000); // Turn left for 800ms
    motor_stop();
    delay_us(SCAN_DELAY_MS * 1000);    // Wait for stabilization

    left_distance = HCSR04_ReadDistance_cm_x100(30);
    if(left_distance > 0) {
        printf("Left distance: %d.%02d cm\r\n", left_distance / 100, left_distance % 100);
    } else {
        printf("Left distance: Out of range (assuming clear)\r\n");
        left_distance = 10000; // Assume clear path if out of range
    }

    // Turn back to center (turn right)
    printf("Returning to center...\r\n");
    motor_turn_right();
```

```
delay_us(TURN_DURATION_MS * 1000); // Turn right for 800ms to center
motor_stop();
delay_us(SCAN_DELAY_MS * 1000);

front_distance = HCSR04_ReadDistance_cm_x100(30);
if(front_distance > 0) {
   printf("Front distance: %d.%02d cm\r\n", front_distance / 100, front_distance % 100);
} else {
   printf("Front distance: Out of range (assuming clear)\r\n");
   front_distance = 10000;
}

// Turn right and measure
printf("Scanning right...\r\n");
motor_turn_right();
delay_us(TURN_DURATION_MS * 1000); // Turn right for 800ms
motor_stop();
delay_us(SCAN_DELAY_MS * 1000);

right_distance = HCSR04_ReadDistance_cm_x100(30);
if(right_distance > 0) {
   printf("Right distance: %d.%02d cm\r\n", right_distance / 100, right_distance % 100);
} else {
   printf("Right distance: Out of range (assuming clear)\r\n");
   right_distance = 10000;
}

// Turn back to center (turn left)
printf("Returning to center...\r\n");
motor_turn_left();
delay_us(TURN_DURATION_MS * 1000); // Turn left for 800ms to center
motor_stop();
delay_us(SCAN_DELAY_MS * 1000);

// Decide which direction has more space
printf("\r\n=== Scan Results ===\r\n");
printf("Left: %d.%02d cm | Front: %d.%02d cm | Right: %d.%02d cm\r\n",
       left_distance / 100, left_distance % 100,
       front_distance / 100, front_distance % 100,
       right_distance / 100, right_distance % 100);

// Find the direction with maximum space
if (left_distance >= right_distance && left_distance >= front_distance) {
   printf("Decision: Turn LEFT (most space)\r\n");
   motor_turn_left();
   delay_us(TURN_DURATION_MS * 1000);
   motor_stop();
}
else if (right_distance >= left_distance && right_distance >= front_distance) {
   printf("Decision: Turn RIGHT (most space)\r\n");
   motor_turn_right();
   delay_us(TURN_DURATION_MS * 1000);
   motor_stop();
}
else {
        printf("Decision: Continue FORWARD (most space)\r\n");
        // Already centered, just continue
    }

    printf("Scan complete. Resuming forward movement...\r\n\r\n");
}

int main(void)
{
    SCB->CPACR |= ((3UL << 10*2) | (3UL << 11*2)); //FPU enable

    // Initialize UART first and add delay
    uart2_tx_init();

    // Small delay to let UART stabilize
    for(volatile uint32_t i = 0; i < 100000; i++);

    DWT_Delay_Init();
    HCSR04_Init();
    buzzer_init();
    GPIO();
    INT();
    motor_init();

    // LED GPIO init (PA5, PA6, PA7)
    RCC->AHB1ENR |= (1U << 0);
    GPIOA->MODER |= (1U << 10) | (1U << 12) | (1U << 14); // Set PA5, PA6, PA7 as output

    int32_t distance;

    // Add delay before first printf
    delay_us(100000); // 100ms delay
    printf("System initialized. Press PC13 to start/stop.\r\n");

    while(1)
    {
        // ALWAYS read and print distance regardless of system state
        distance = HCSR04_ReadDistance_cm_x100(30);

        if(distance > 0)
        {
            printf("Distance: %d.%02d cm\r\n", distance / 100, distance % 100);
        }
        else
        {
            printf("Distance: Out of range\r\n");
        }

        // Only activate buzzer, LEDs, and motor when system is active
        if (system_active)
        {
            // Motor control based on distance with scanning logic
            if (distance > 0 && distance < 1000)  // Less than 10cm
            {
              // Call the scanning function
              scan_and_decide_direction();
            }
            else if (distance > 0)
            {
              motor_forward();
            }

            // LED control based on distance
            if(distance > 0 && distance < 1000)
            {
              GPIOA->ODR = (1 << 5); // Red ON, others OFF
            }
            else if(distance >= 1000 && distance < 3000)
            {
              GPIOA->ODR = (1 << 6); // Yellow ON, others OFF
            }
            else if(distance >= 3000)
            {
              GPIOA->ODR = (1 << 7); // Green ON, others OFF
            }
            else
            {
              GPIOA->ODR &= ~((1 << 5) | (1 << 6) | (1 << 7)); // All OFF
            }

            // Buzzer beeping pattern based on distance
            if (distance > 0 && distance <= 3000)
            {
              // Map distance (1-3000) to a beep interval
              const uint32_t INTERVAL_MAX_MS = 500; // Delay at 30cm
              const uint32_t INTERVAL_MIN_MS = 65;  // Delay at <1cm
              const uint32_t D_MAX_VAL = 3000;

              // Calculate the delay between the start of one beep and the start of the next
              uint32_t beep_interval_ms = INTERVAL_MIN_MS + (((uint64_t)distance *
(INTERVAL_MAX_MS - INTERVAL_MIN_MS)) / D_MAX_VAL);

              // Turn the beep on
              buzzer_on();
              delay_us(BEEP_DURATION_MS * 1000); // Beep for a fixed duration

              // Turn the beep off
              buzzer_off();

              // Wait for the rest of the interval before the next beep
              if (beep_interval_ms > BEEP_DURATION_MS)
              {
                delay_us((beep_interval_ms - BEEP_DURATION_MS) * 1000);
              }
            }
            else
            {
              // If out of range, just wait a bit
```

```
            delay_us(50000); // 50ms delay
          }
        }
        else
        {
          // System inactive - turn everything off but keep measuring distance
          buzzer_off();
          motor_stop();
          GPIOA->ODR &= ~((1 << 5) | (1 << 6) | (1 << 7)); // All LEDs OFF
          delay_us(100000); // 100ms delay between distance readings
        }
    }
}

// PC13 interrupt handler - toggles system on/off
void EXTI15_10_IRQHandler(void)
{
    // Check if EXTI13 triggered the interrupt
    if (EXTI->PR & (1U << 13))
    {
        // Clear the pending bit
        EXTI->PR |= (1U << 13);

        // Toggle system state
        system_active = !system_active;

        if (system_active)
        {
          printf("\r\nSystem ACTIVATED\r\n");
        }
        else
        {
          printf("\r\nSystem DEACTIVATED - Distance monitoring only\r\n");
          // Immediately turn everything off
          buzzer_off();
          motor_stop();
          GPIOA->ODR &= ~((1 << 5) | (1 << 6) | (1 << 7));
        }

        // Simple debounce delay
        delay_us(200000); // 200ms debounce
    }
}
```

uart.h

```
#ifndef UART_H_
#define UART_H_
#include "stm32f4xx.h"
#include <stdint.h>
```

```
void uart2_tx_init(void);

#endif /* UART_H_ */
```

uart.c

```
#include "uart.h"

#define GPIOAEN         (1U << 0)
#define UART2EN         (1U << 17)

#define CR1_TE          (1U << 3)
#define CR1_UE          (1U << 13)
#define SR_TXE          (1U << 7)

#define SYS_FREQ        16000000
#define APB1_CLK        SYS_FREQ

#define UART_BAUDRATE   115200

static void uart_set_baudrate(USART_TypeDef *USARTx, uint32_t PeriphClk, uint32_t BaudRate);
static uint16_t compute_uart_bd(uint32_t PeriphClk, uint32_t BaudRate);

void uart2_write(int ch);


int __io_putchar(int ch)
{
	uart2_write(ch);
	return ch;
}


void uart2_tx_init(void)
{
	/*Configure UART GPIO pin*/
	/*Enable clock access to gpio a*/
	RCC->AHB1ENR |= GPIOAEN;

	/*Set PA2 mode to Alternate function mode*/
	GPIOA->MODER &= ~(1U << 4);
	GPIOA->MODER |= (1U << 5);

	/*Set PA2 alternate function type to UART_TX (AF07)*/
	GPIOA->AFR[0] |= (7U << 8);
	GPIOA->AFR[0] &= ~(1U << 11);


	/*Configure UART module*/
	/*Enable clock access to uart2*/
	RCC->APB1ENR |=UART2EN;

	/*Configure baudrate*/
	uart_set_baudrate(USART2,APB1_CLK,UART_BAUDRATE);
```

```
        /*Configure the transfer direction*/
        USART2->CR1 = CR1_TE;

        /*Enable uart module*/
        USART2->CR1 |=CR1_UE;
}


void uart2_write(int ch)
{
         /*Make sure transmit data register is empty*/
        while(!(USART2->SR & SR_TXE)){}
        /*Write to transmit data register*/
        USART2->DR = (ch & 0xFF);
}

static void uart_set_baudrate(USART_TypeDef *USARTx, uint32_t PeriphClk, uint32_t BaudRate)
{
        USARTx->BRR = compute_uart_bd(PeriphClk,BaudRate);
}


static uint16_t compute_uart_bd(uint32_t PeriphClk, uint32_t BaudRate)
{
        return (PeriphClk + (BaudRate/2U))/BaudRate;
}
```

timing.h

```
#ifndef TIMING_H_
#define TIMING_H_

#include "stm32f4xx.h"
#include <stdint.h>

void DWT_Delay_Init(void);
uint32_t micros(void);
void delay_us(uint32_t us);

#endif /* TIMING_H_ */
```

timing.c

```
#include "timing.h"

static uint32_t cycles_per_us = 0;

void DWT_Delay_Init(void)
{
  SystemCoreClockUpdate();
```

```
    cycles_per_us = SystemCoreClock / 1000000U;
    if (cycles_per_us == 0) cycles_per_us = 1;

    CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;   // enable trace
    DWT->CYCCNT = 0;                                  // reset cycle counter
    DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;              // enable cycle counter
}


uint32_t micros(void)
{
    return (uint32_t)(DWT->CYCCNT / cycles_per_us);
}

void delay_us(uint32_t us)
{
    uint32_t start = DWT->CYCCNT;
    uint32_t ticks = us * cycles_per_us;
    while ((uint32_t)(DWT->CYCCNT - start) < ticks) { __NOP(); }
}
```

hcsr04.h

```
#ifndef HCSR04_H_
#define HCSR04_H_

#include "stm32f4xx.h"
#include <stdint.h>

// Pin definitions
#define HCSR04_TRIG_PIN   (1U << 1)   // PA1
#define HCSR04_ECHO_PIN   (1U << 0)   // PA0
void HCSR04_Init(void);
int32_t HCSR04_ReadDistance_cm_x100(uint32_t timeout_ms);

// Return last measured pulse width in microseconds (0 if none)
uint32_t HCSR04_GetLastPulse_us(void);

#endif /* HCSR04_H_ */
```

hcsr04.c

```
#include "hcsr04.h"
#include "timing.h"

static volatile uint32_t last_pulse_us = 0;

void HCSR04_Init(void)
{
    DWT_Delay_Init();

    /* Enable GPIOA */
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
    /* PA1 -> TRIG output */
    GPIOA->MODER &= ~(3U << 2);
    GPIOA->MODER |=  (1U << 2);   // 01 = output
    GPIOA->BSRR = (HCSR04_TRIG_PIN << 16);

    /* PA0 -> ECHO input */
    GPIOA->MODER &= ~(3U << 0);   // 00 = input
    GPIOA->PUPDR &= ~(3U << 0);
}

/* Internal: trigger pulse 10 us */
static void hcsr04_trigger_pulse(void)
{
    GPIOA->BSRR = HCSR04_TRIG_PIN;           // set PA1
    delay_us(10);
    GPIOA->BSRR = (HCSR04_TRIG_PIN << 16);   // reset PA1
}

/* Blocking read, returns distance*100 or -1 */
int32_t HCSR04_ReadDistance_cm_x100(uint32_t timeout_ms)
{
    uint32_t timeout_us = timeout_ms * 1000U;

    /* Trigger */
    hcsr04_trigger_pulse();

    /* Wait for rising edge */
    uint32_t t_start = micros();
    while (!(GPIOA->IDR & HCSR04_ECHO_PIN))
    {
        if ((uint32_t)(micros() - t_start) >= timeout_us) return -1;
    }

    /* Record rising time */
    uint32_t t_rise = micros();

    /* Wait for falling edge */
    while (GPIOA->IDR & HCSR04_ECHO_PIN)
    {
        if ((uint32_t)(micros() - t_rise) >= timeout_us) return -1;
    }

    uint32_t t_fall = micros();

    /* Pulse duration in microseconds*/
    uint32_t duration_us = (uint32_t)(t_fall - t_rise);
    last_pulse_us = duration_us;

    /* Convert to cm * 100 (with rounding):
       distance_cm = duration_us / 58.0
       So dist_x100 = round(duration_us * 100 / 58)
       integer: (duration_us*100 + 29) / 58  (29 for rounding)
    */
    uint64_t tmp = (uint64_t)duration_us * 100ULL + 29ULL;
    uint32_t dist_x100 = (uint32_t)(tmp / 58ULL);

    return (int32_t)dist_x100;
}

uint32_t HCSR04_GetLastPulse_us(void)
{
    return last_pulse_us;
}
```

buzz.h

```
#ifndef BUZZ_H_
#define BUZZ_H_

#include <stdint.h>


void buzzer_init(void);
void buzzer_on(void);
void buzzer_off(void);

#endif /* BUZZ_H_ */
```

buzz.c

```
#include "buzz.h"
#include "stm32f4xx.h"

#define BUZZER_FREQUENCY 2700 // 2.7 kHz

static void buzzer_set_pitch(uint32_t frequency_hz)
{
    if (frequency_hz == 0)
    {
        TIM4->CCR1 = 0;
        return;
    }
    uint32_t arr_value = (1000000U / frequency_hz) - 1;
    TIM4->ARR = arr_value;
    TIM4->CCR1 = arr_value / 2;
}




void buzzer_init(void)
{
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN;
    RCC->APB1ENR |= RCC_APB1ENR_TIM4EN;
    GPIOB->MODER &= ~GPIO_MODER_MODER6_Msk;
    GPIOB->MODER |= GPIO_MODER_MODER6_1;
    GPIOB->AFR[0] &= ~GPIO_AFRL_AFSEL6_Msk;
    GPIOB->AFR[0] |= (2U << GPIO_AFRL_AFSEL6_Pos);

    TIM4->PSC = 15; // 16MHz / 16 = 1MHz counter clock
    TIM4->CCMR1 &= ~TIM_CCMR1_OC1M_Msk;
    TIM4->CCMR1 |= TIM_CCMR1_OC1M_2 | TIM_CCMR1_OC1M_1;
    TIM4->CCMR1 |= TIM_CCMR1_OC1PE;
    TIM4->CR1 |= TIM_CR1_ARPE;
    TIM4->CCER |= TIM_CCER_CC1E;
    TIM4->CR1 |= TIM_CR1_CEN;
}

void buzzer_on(void)
{
    buzzer_set_pitch(BUZZER_FREQUENCY);
}

void buzzer_off(void)
{
    buzzer_set_pitch(0);
}
```

int.h

```
#ifndef INT_H_
#define INT_H_


void GPIO(void);
void INT(void);

#endif /* INT_H_ */
```

int.c

```
#include <stdint.h>
#include "int.h"
#include "stm32f4xx.h"


void GPIO(void)
{
          /*
           * Clock enable for GPIOC
           */
          RCC->AHB1ENR |= ( 1U << 2 );

          //Mode selection

          GPIOC->MODER &= ~( 3U << 26 ); //PC13 INPUT
```

```
        // Pull-up

        GPIOC->PUPDR |= ( 1U << 26 ); //PC13 Pull-up

}

void INT(void)
{
        //SYSCFG Clk EN

        RCC->APB2ENR |= ( 1U << 14 );


        //falling edge trigger

        EXTI->FTSR |= ( 1U << 13);


        //Port Code in SYSCFG_EXTICRx registers


        SYSCFG->EXTICR[3] &= ~(0xF << 4);
        SYSCFG->EXTICR[3] |= (0x2 << 4);   // 0010 = Port C


         //IMR enable

        EXTI->IMR |= ( 1U << 13);


         // interrupt set enable

        NVIC->ISER[1] |=( 1U << 8);

}
```

motor.h

```
#ifndef MOTOR_H_
#define MOTOR_H_

#include <stdint.h>

// Pin definitions for L298N motor driver
#define MOTOR_LEFT_IN1_PIN    8   // PB8
#define MOTOR_LEFT_IN2_PIN    9   // PB9
#define MOTOR_RIGHT_IN3_PIN   10  // PB10
#define MOTOR_RIGHT_IN4_PIN   4   // PB4

// Function prototypes
void motor_init(void);
void motor_forward(void);
void motor_turn_left(void);
void motor_turn_right(void);
void motor_stop(void);

#endif /* MOTOR_H_ */
```

motor.c

```
#include "motor.h"
#include "stm32f4xx.h"

void motor_init(void)
{
   // Enable GPIOB clock
   RCC->AHB1ENR |= (1U << 1);

   // Configure PB8, PB9, PB10, PB4 as output
   GPIOB->MODER &= ~((3U << 16) | (3U << 18) | (3U << 20) | (3U << 8));  // Clear bits
   GPIOB->MODER |= (1U << 16) | (1U << 18) | (1U << 20) | (1U << 8);     // Set as output (01)

   // Set output to push-pull (default)
   GPIOB->OTYPER &= ~((1U << 8) | (1U << 9) | (1U << 10) | (1U << 4));

   // Set to low speed
   GPIOB->OSPEEDR &= ~((3U << 16) | (3U << 18) | (3U << 20) | (3U << 8));

   // No pull-up/pull-down
   GPIOB->PUPDR &= ~((3U << 16) | (3U << 18) | (3U << 20) | (3U << 8));

   // Initialize motors as stopped
   motor_stop();
}

// Move forward - both motors forward
void motor_forward(void)
{
   // Left motor forward: IN1 = HIGH, IN2 = LOW
   GPIOB->BSRR = (1U << MOTOR_LEFT_IN1_PIN);          // Set PB8 HIGH
   GPIOB->BSRR = (1U << (MOTOR_LEFT_IN2_PIN + 16));    // Set PB9 LOW

   // Right motor forward: IN3 = HIGH, IN4 = LOW
   GPIOB->BSRR = (1U << MOTOR_RIGHT_IN3_PIN);         // Set PB10 HIGH
   GPIOB->BSRR = (1U << (MOTOR_RIGHT_IN4_PIN + 16));   // Set PB4 LOW
}

// Turn left - left motor backward, right motor forward
void motor_turn_left(void)
{
   // Left motor backward: IN1 = LOW, IN2 = HIGH
   GPIOB->BSRR = (1U << (MOTOR_LEFT_IN1_PIN + 16));    // Set PB8 LOW
   GPIOB->BSRR = (1U << MOTOR_LEFT_IN2_PIN);          // Set PB9 HIGH

   // Right motor forward: IN3 = HIGH, IN4 = LOW
```

```
    GPIOB->BSRR = (1U << MOTOR_RIGHT_IN3_PIN);          // Set PB10 HIGH
    GPIOB->BSRR = (1U << (MOTOR_RIGHT_IN4_PIN + 16));   // Set PB4 LOW
}

// Turn right - left motor forward, right motor backward
void motor_turn_right(void)
{
    // Left motor forward: IN1 = HIGH, IN2 = LOW
    GPIOB->BSRR = (1U << MOTOR_LEFT_IN1_PIN);           // Set PB8 HIGH
    GPIOB->BSRR = (1U << (MOTOR_LEFT_IN2_PIN + 16));    // Set PB9 LOW

    // Right motor backward: IN3 = LOW, IN4 = HIGH
    GPIOB->BSRR = (1U << (MOTOR_RIGHT_IN3_PIN + 16));   // Set PB10 LOW
    GPIOB->BSRR = (1U << MOTOR_RIGHT_IN4_PIN);          // Set PB4 HIGH
}

// Stop both motors
void motor_stop(void)
{
    // All pins LOW to stop both motors
    GPIOB->BSRR = (1U << (MOTOR_LEFT_IN1_PIN + 16));    // Set PB8 LOW
    GPIOB->BSRR = (1U << (MOTOR_LEFT_IN2_PIN + 16));    // Set PB9 LOW
    GPIOB->BSRR = (1U << (MOTOR_RIGHT_IN3_PIN + 16));   // Set PB10 LOW
    GPIOB->BSRR = (1U << (MOTOR_RIGHT_IN4_PIN + 16));   // Set PB4 LOW
}
```

1.2.5 Process Description

The software is built on a set of modular drivers, each responsible for one piece of hardware. The main.c file acts as the integrator, **combining** these drivers to create the final application logic.

1.1 Clock and Timing (RCC & DWT):

- GPIO clocks are enabled in each driver's _init function .
- Peripheral clocks are also enabled (e.g., RCC->APB1ENR |= UART2EN;, RCC->APB1ENR |= RCC_APB1ENR_TIM4EN;).
- A high-precision microsecond delay is achieved using the ARM core's DWT (Data Watchpoint and Trace) cycle counter, initialized in DWT_Delay_Init().

1.2 UART Driver (uart.c):

- Initializes PA2 to Alternate function mode.
- Sets the corresponding AFR[0] register to AF07 for UART2.
- Configures the USART2 peripheral by calculating and setting the baud rate in the BRR register.
- Enables the Transmitter (CR1_TE) and the UART module itself (CR1_UE).
- Retargets printf by implementing __io_putchar to call uart2_write.

1.3 HC-SR04 Driver (hcsr04.c):

- Implements a blocking-read function HCSR04_ReadDistance_cm_x100.
- This function first sends a 10-microsecond trigger pulse on PA1.
- It then enters a blocking loop, waiting for the PA0 (ECHO) pin to go high, recording the t_rise time.
- It then waits for the pin to go low, recording t_fall.
- The pulse duration (t_fall - t_rise) is converted to distance (in cm*100) using the formula: dist_x100 = ((duration_us * 100) + 29) / 58, which includes rounding.

1.4 Buzzer PWM Driver (Buzz.c):

- Configures PB6 for Alternate function mode (AF02) to connect it to TIM4_CH1.
- Initializes TIM4 as a PWM output. The counter clock is set to 1MHz.
- buzzer_set_pitch controls the sound:
- frequency_hz = 0 sets the CCR1 (Compare) register to 0, stopping the PWM output.
- frequency_hz > 0 sets the ARR (Auto-Reload) register to define the period and CCR1 to half of ARR to create a 50% duty cycle, generating a tone.

1.5 Interrupt Driver (int.c):

- Configures PC13 as a pull-up input.
- Enables the SYSCFG clock (RCC->APB2ENR) to access interrupt configuration registers.
- Maps EXTI13 to Port C by writing 0010 to the correct field in SYSCFG->EXTICR[3].

- Configures the EXTI to trigger on a falling edge (FTSR).
- Unmasks the interrupt in the EXTI->IMR and enables it in the NVIC->ISER.

1.6 Motor Driver (Motor.c):

- Configures four GPIOB pins (PB8, PB9, PB10, PB4) as digital outputs to send control signals.
- Movement Functions: Functions like motor_forward(), motor_turn_left(), and motor_turn_right() set these output pins HIGH or LOW.
- H-Bridge Control: These HIGH/LOW combinations are sent to an H-Bridge (like an L298N), which interprets them to power the motors, causing forward, left, or right movement.
- motor_stop(): Sets all four control pins to LOW, signalling the H-Bridge to cut power and stop both motors.

1.7 Process Flow (Main Loop)

- The main.c file contains the high-level application logic.
    - Initialization: FPU is enabled, all _init() functions for drivers are called, and a "System initialized" message is printed via UART.
    - Infinite Loop: The while(1) loop continuously executes the core logic.
    - Always-On Sensing: On *every* loop iteration, HCSR04_ReadDistance_cm_x100 is called and the result is printed to the UART. This ensures the user *always* has a distance reading, even if the robot is "deactivated".
    - State-Based Action (if (system_active)):
        - If system_active is FALSE: The code jumps to the else block. buzzer_off(), motor_stop(), and all LEDs are turned off. A 100ms delay is added to prevent spamming the UART.
        - If system_active is TRUE: The full obstacle avoidance and feedback logic is executed.
        - Motor Control:
            - If distance > 0 and distance < 1000 (10 cm):
            - motor_stop() is called.
            - delay_us(500000) (500ms) pauses the robot.
            - motor_turn_left() is called.
            - delay_us(800000) (800ms) executes the turn.
            - motor_turn_right() is called.
            - delay_us(800000) (800ms) executes the turn and car returns to center.
            - motor_turn_right() is called.
            - delay_us(800000) (800ms) executes the turn.
            - motor_turn_left() is called.
            - delay_us(800000) (800ms) executes the turn and car returns to center.
            - Bot moves in the direction which has more open space.
            - The loop repeats, and if the path is now clear, the else block will be hit, moving the robot forward.
            - If distance > 1000 (or out of range): motor_forward() is called.
        - LED Feedback: A simple if-else-if chain sets the GPIOA->ODR register to light the appropriate LED (Red < 10cm, Yellow 10-30cm,

Green > 30cm).

    - Buzzer Feedback:
        - If distance > 0 and distance <= 3000 (30 cm):
        - A beep interval is calculated, mapping the distance (1-3000) to an interval range (65ms-500ms). Closer distances result in shorter intervals.
        - buzzer_on() is called.
        - A fixed BEEP_DURATION_MS (50ms) delay occurs.
        - buzzer_off() is called.
        - A final delay (beep_interval_ms - BEEP_DURATION_MS) waits for the rest of the calculated interval. This creates a beeping pattern that gets faster as the robot approaches an obstacle.

- Interrupt Handling (EXTI15_10_IRQHandler):
    - This function runs outside the main loop whenever PC13 is pressed.
    - It checks and clears the EXTI13 pending bit (EXTI->PR).
    - It toggles the system_active flag.
    - It prints a status message ("ACTIVATED" or "DEACTIVATED").
    - If deactivating, it immediately calls buzzer_off() and motor_stop() for safety.
    - A 200ms debounce delay prevents multiple triggers from a single button press.

1.2.6 Flowchart

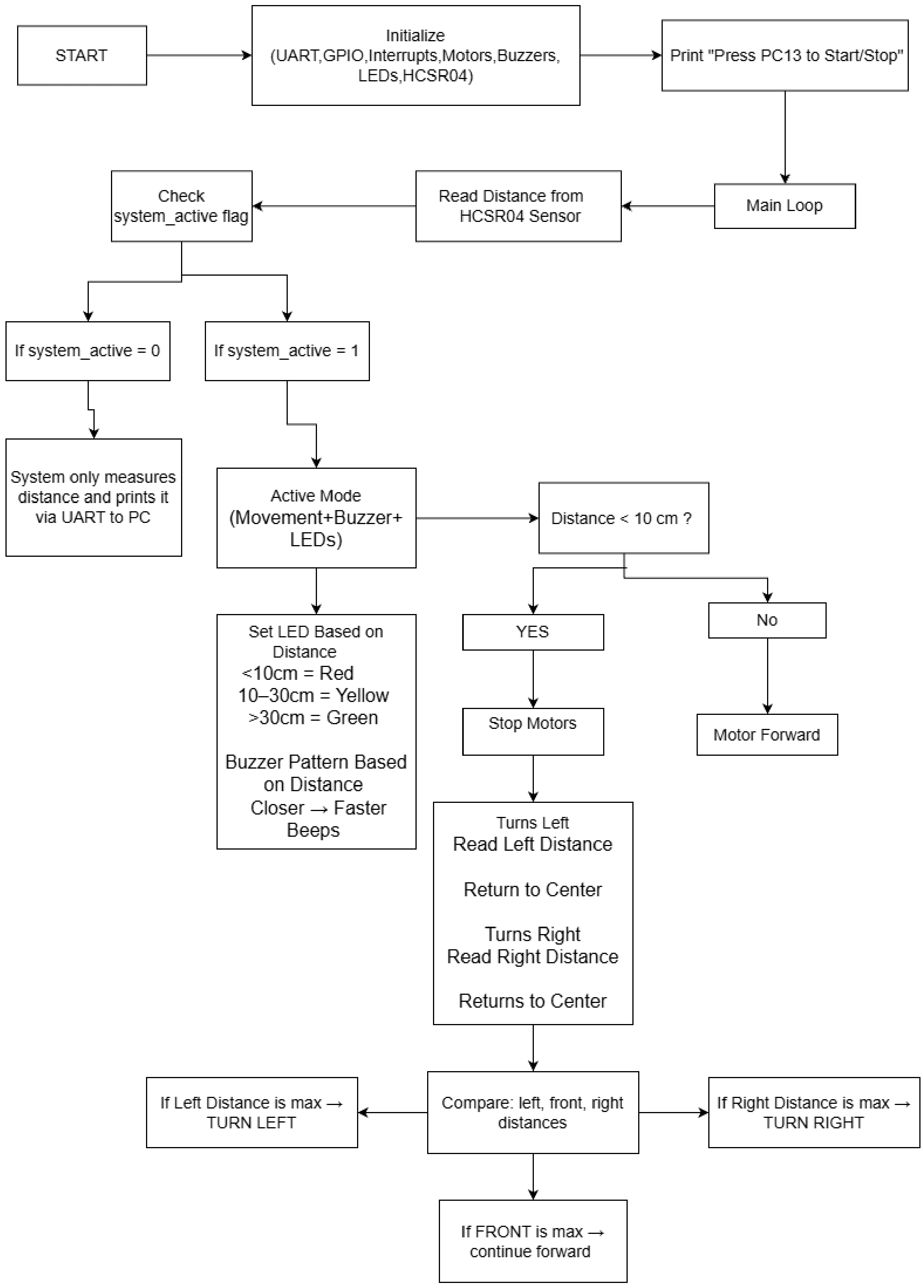

START
Initialize (UART,GPIO,Interrupts,Motors,Buzzers, LEDs,HCSR04)
Print "Press PC13 to Start/Stop"
Main Loop
Read Distance from HCSR04 Sensor
Check system_active flag
If system_active = 0
If system_active = 1
System only measures distance and prints it via UART to PC
Active Mode (Movement+Buzzer+ LEDs)
Distance < 10 cm ?
Set LED Based on Distance <10cm = Red 10–30cm = Yellow >30cm = Green
Buzzer Pattern Based on Distance Closer → Faster Beeps
YES
No
Stop Motors
Motor Forward
Turns Left Read Left Distance
Return to Center
Turns Right Read Right Distance
Returns to Center
Compare: left, front, right distances
If Left Distance is max → TURN LEFT
If Right Distance is max → TURN RIGHT
If FRONT is max → continue forward

Fig. 1.2.8 Flow Chart

1.2.7 Hardware Implementation

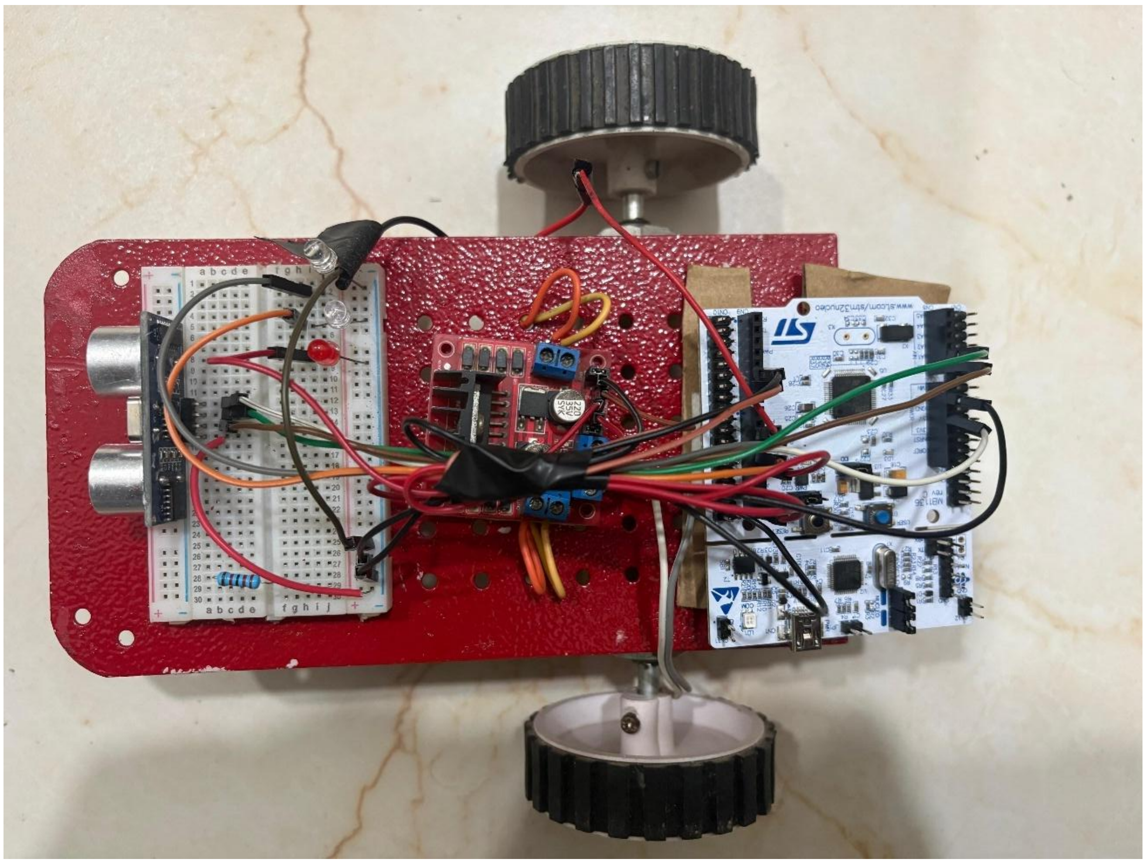


Fig. 1.2.9 Hardware Implementation

1.3 Result

- The system was tested with obstacles placed at different distances. The ultrasonic sensor provided stable readings within the range of 5 cm to 100 cm. The buzzer frequency increased as the car approached obstacles. LED indications changed reliably based on different ranges.
- Motor behavior was consistent, stopping instantly when obstacles were detected at close range and checking distances on both sides before turning in the direction that has more open space.

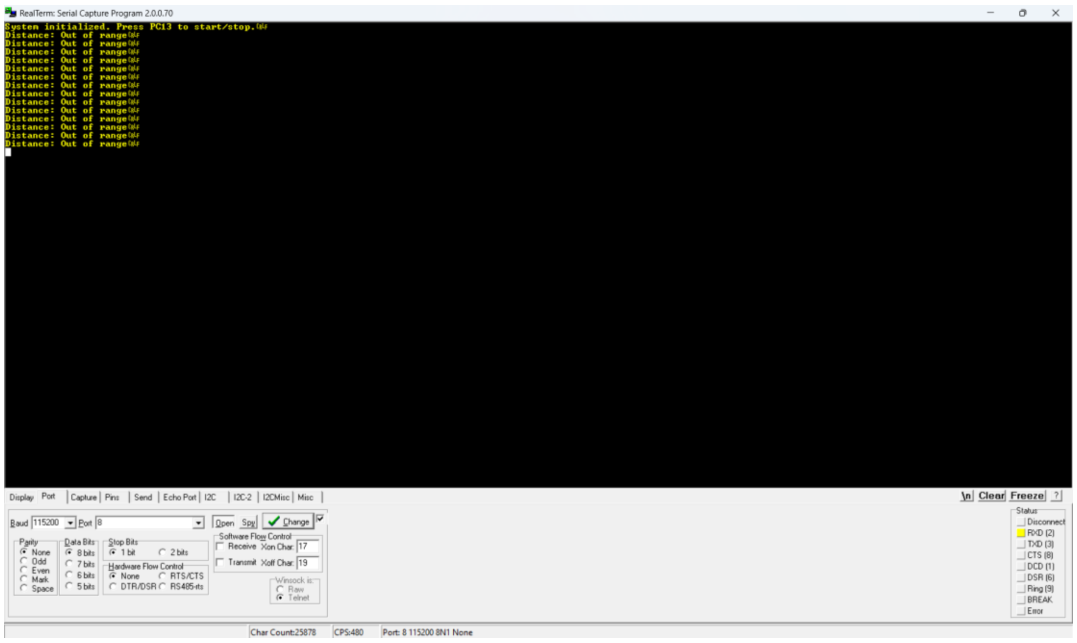
Feature	**Expected Behavior (from code)**	**Observed Result**
System Boot	System initializes peripherals and prints "System initialized. Press PC13 to start/stop."	The message was received correctly in the UART serial monitor.
Idle State	system_active = 0. Robot is stationary, LEDs and buzzer are off. UART continuously prints distance.	The robot remained stationary, and distance readings were printed.
Activation	Press PC13 button. system_active becomes 1. === System ACTIVATED === is printed. Robot starts moving forward (if path > 10cm).	The interrupt fired correctly, the status message was printed, and the robot began moving.
Obstacle Avoid	distance < 1000 (10cm). Robot stops, prints "Obstacle detected!", checks distance both on left and right and move in the direction that has more open space.	The avoidance maneuver was executed exactly as programmed.
LED Feedback	Green LED for distance >= 30cm. Yellow for 10cm <= distance < 30cm. Red for distance < 10cm.	The LEDs correctly transitioned between colors as an object was moved toward and away from the sensor.
Buzzer Feedback	No beep for distance > 30cm. Beeping interval increases from 65ms (min) to 500ms (max) as distance increases from 1cm to 30cm.	The buzzer feedback provided a sense of proximity.
Deactivation	Press PC13 button. system_active becomes 0. === System DEACTIVATED === is printed. Motors and buzzer stop immediately.	The interrupt provided immediate shutdown of all actuators, as expected.

Table 1.3.1 Comparison between Expected and observed results

Observations

The experimental testing of the designed bot was carried out under different environmental and operating conditions. The key observations as screenshot of serial monitor are given below:

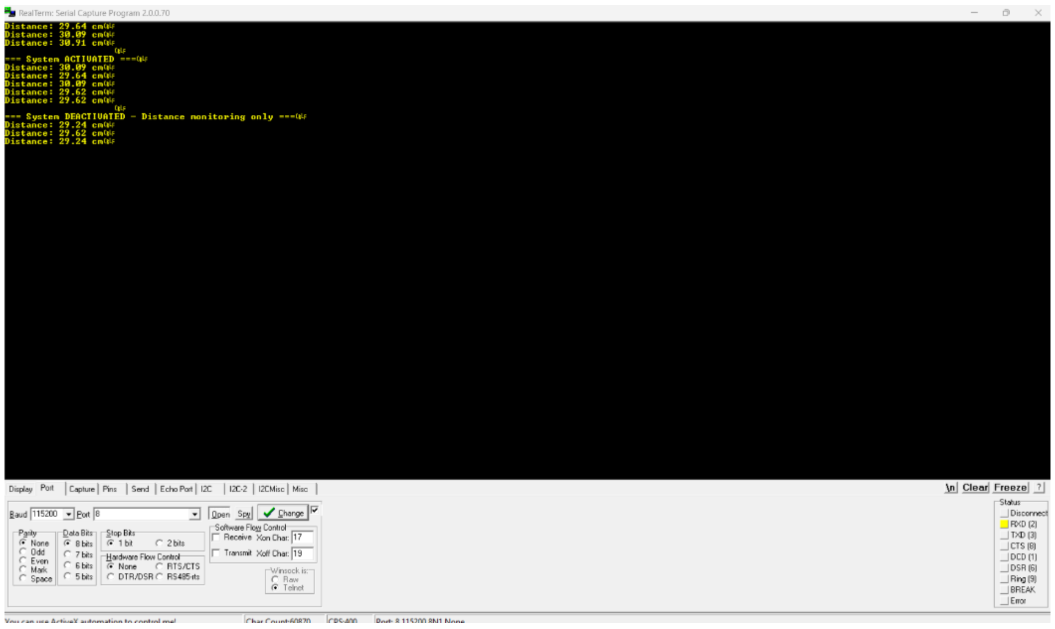
Serial Monitor Screenshot



```
System initialized. Press PC13 to start/stop.
Distance: Out of range
Distance: Out of range
Distance: Out of range
Distance: Out of range
Distance: Out of range
Distance: Out of range
Distance: Out of range
Distance: Out of range
Distance: Out of range
Distance: Out of range
Distance: Out of range
Distance: Out of range
Distance: Out of range
Distance: Out of range
```

Baud 115200 Port 8
Parity: None; Data Bits: 8 bits; Stop Bits: 1 bit; Hardware Flow Control: None
Software Flow Control: Xon Char: 17, Xoff Char: 19
Char Count:25878 | CPS:480 | Port: 8 115200 8N1 None

Fig. 1.3.1 System Initialized



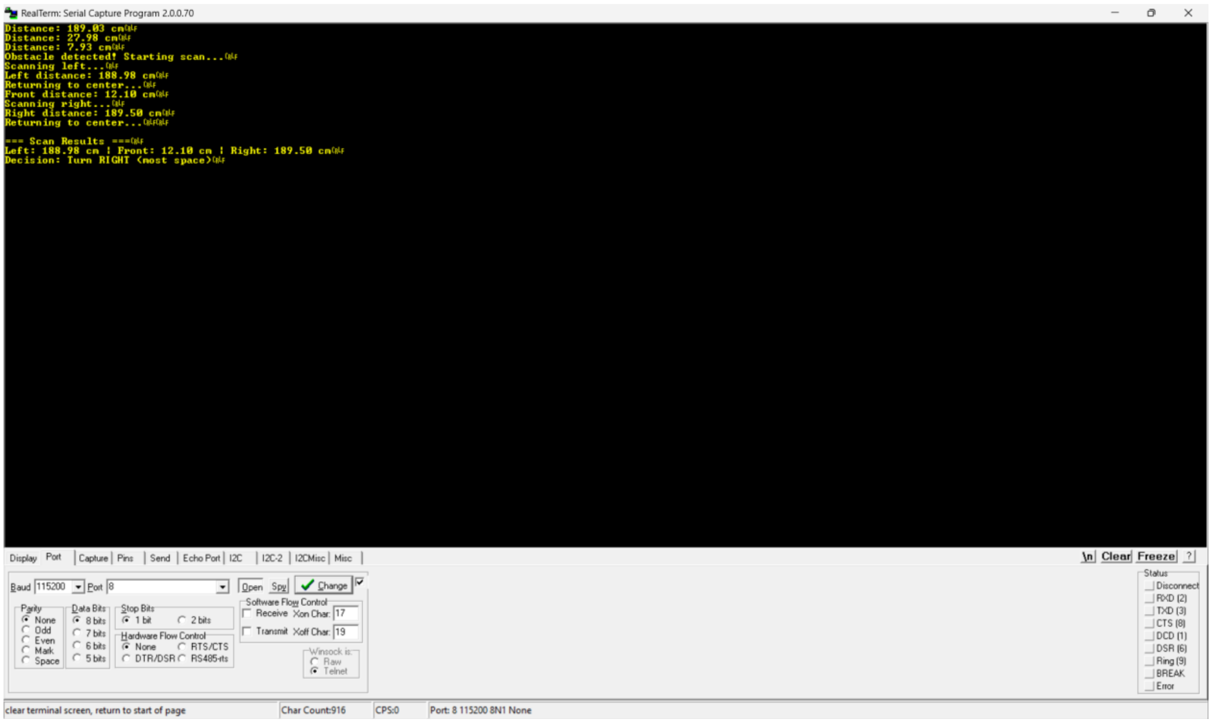
```
Distance: 29.64 cm
Distance: 30.09 cm
Distance: 30.91 cm

=== System ACTIVATED ===
Distance: 30.09 cm
Distance: 29.64 cm
Distance: 30.09 cm
Distance: 29.62 cm
Distance: 29.62 cm

=== System DEACTIVATED - Distance monitoring only ===
Distance: 29.24 cm
Distance: 29.62 cm
Distance: 29.24 cm
```

Baud 115200 Port 8
Parity: None; Data Bits: 8 bits; Stop Bits: 1 bit; Hardware Flow Control: None
Software Flow Control: Xon Char: 17, Xoff Char: 19
You can use ActiveX automation to control me! | Char Count:60870 | CPS:400 | Port: 8 115200 8N1 None

Fig. 1.3.2 System Activated and Deactivated



```
Distance: 189.03 cmCRLF
Distance: 27.98 cmCRLF
Distance: 7.93 cmCRLF
Obstacle detected! Starting scan...CRLF
Scanning left...CRLF
Left distance: 188.98 cmCRLF
Returning to center...CRLF
Front distance: 12.10 cmCRLF
Scanning right...CRLF
Right distance: 189.50 cmCRLF
Returning to center...CRLFCRLF

=== Scan Results ===CRLF
Left: 188.98 cm | Front: 12.10 cm | Right: 189.50 cmCRLF
Decision: Turn RIGHT (most space)CRLF
```

Baud 115200 Port 8

Parity: [✓ None] Odd Even Mark Space
Data Bits: [✓ 8 bits] 7 bits 6 bits 5 bits
Stop Bits: [✓ 1 bit] 2 bits
Hardware Flow Control: [✓ None] RTS/CTS DTR/DSR RS485-rts
Software Flow Control: Receive Xon Char: 17 Transmit Xoff Char: 19
Winsock is: Raw [✓ Telnet]

clear terminal screen, return to start of page | Char Count:916 | CPS:0 | Port: 8 115200 8N1 None

Fig. 1.3.3 Obstacle in front of bot and decision to tun right

1.4 Conclusion

This project successfully demonstrates the design, construction, and programming of an obstacle-avoiding car. By leveraging the capabilities of the STM32F446RE microcontroller and writing custom, low-level drivers in C, a robust and responsive system was achieved. The project successfully integrated multiple hardware peripherals—a sensor (HC-SR04), actuators (DC motors), and feedback devices (LEDs, buzzer)—all controlled by a central control loop.

The implementation of an interrupt-driven start/stop mechanism represents a significant enhancement over a simple polling-based switch, ensuring immediate user responsiveness without compromising the system's primary sensing loop. The feedback system, particularly the proportional-rate buzzer, proved highly effective at communicating the robot's state and perception to the user.

Bibliography

1. STMicroelectronics. (2016). *RM0390 Reference Manual: STM32F446xx advanced Arm®-based 32-bit MCUs*.
2. STMicroelectronics. (2017). *DS10693 Datasheet: STM32F446RE Arm® Cortex®-M4 32-bit MCU*.
3. STMicroelectronics. (2008). *L298N: Dual H-Bridge Motor Driver Datasheet*.
4. SparkFun Electronics. *HC-SR04 Ultrasonic Sensor Datasheet*.
5. Texas Instruments. *UART Protocol Overview and Error Sources*.