RT – CAR PROJECT

Laura Pérez Soto

Florencio Jesús Atienza Miranda

Martyna Baran

Zuzanna Jarlaczynska

INDEX

1. INTRODUCTION

In this report, we will focus on the development and control of an electric vehicle using the STM32F767ZI microcontroller. The device will employ a real-time operating system to manage the motors and other additional components, and the project will be carried out in the STM32CubeIDE software (version 1.13.2), using Ozone for code debugging.

Our project aims to create a circuit based on black lines that the vehicle will follow, guided by infrared sensors. When the vehicle encounters an object at a distance less than 10 cm, it will stop, and a red LED will illuminate. Once the object disappears, the vehicle will resume its path, guided by the infrared sensors. Simultaneously, the distance between the object and the ultrasonic sensor will be displayed on an LCD screen. Additionally, independently from the rest of the project and to verify the proper functioning of the accelerometer, we have created a program that will print the values of the MPU6050 sensor on the LCD screen.

In the initial phase, we will detail all the components necessary for the implementation of our project. These include the STM32F767ZI microcontroller, two L298N drivers for the motors, an ultrasonic sensor for object detection, two infrared sensors for detecting lines on the ground, an LED with a resistor to indicate a specific distance from an object, an LCD screen to display the distance, and an MPU6050 accelerometer to measure the position and acceleration of the vehicle. Three batteries will be used to ensure the proper functioning of the vehicle.

In the second part, after understanding the operation of these components and their relevance to the project, we will proceed to wire all the elements. We will develop a schematic to visualize these connections and detail the configuration of each pin in the STM32CubeIDE. After configuring the pins, we will explain the pseudocode using a flowchart and then detail the developed code, including functions, tasks, and coordination through semaphores.

We will conclude the report with details of the experimental tests, accompanied by images demonstrating the proper operation of the program. Additionally, we will draw conclusions about the project, the obtained results, possible improvements, and the challenges faced.

2. COMPONENTS

In this section, we will provide a detailed analysis of all the components we have used in the development of our project. Additionally, we will highlight the key pins that play a crucial role in various connections.

2.1. STM32F767ZI Microcontroller

The STM32F767ZI microcontroller, as shown in Figure 1, serves as the brain of the vehicle, providing an advanced platform for real-time operations and complex task management. This high-performance microcontroller, based on an ARM Cortex-M7 processor, achieves speeds of up to 216 MHz, making it ideal for demanding applications. It includes a floating-point unit for precise calculations and a wide range of peripherals that offer diverse connectivity options.

Its ample flash memory and RAM are crucial for complex software and real-time operating systems, essential for the multitasking demands of this project. The integration of this microcontroller enables the execution of concurrent tasks, such as sensor data processing and motor control, without compromising performance.

Its I/O interfaces, such as I2C, SPI, and UART, facilitate modular design and the incorporation of sensors and actuators. The advanced interrupt handling capabilities ensure timely responses to sensor inputs, crucial for obstacle detection and line tracking. STM32CubeIDE, the development environment, optimizes firmware creation by providing comprehensive tools for debugging and performance optimization.

This microcontroller not only offers the computational power needed to process inputs from various sensors but also provides the PWM control necessary for precise motor speed regulation. Its robust architecture and feature set make it an ideal choice for this project, where reliability, efficiency, and real-time responsiveness are paramount.

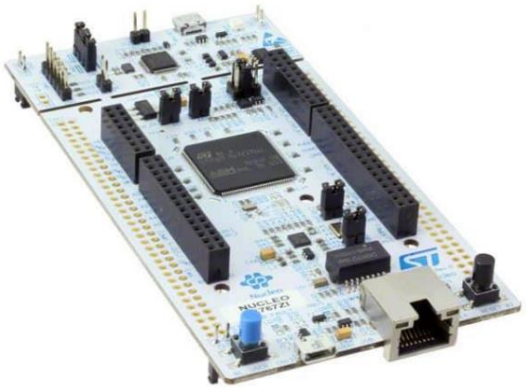

Figure 1: STM32F767ZI Board

2.2. Motors and Drivers

The controller we have used is the L298N. It is a dual motor controller that allows handling two DC motors with independent control of direction and speed. It is designed to drive inductive loads such as relays, solenoids, and especially motors. This controller can be seen in Figures 3 and 4.

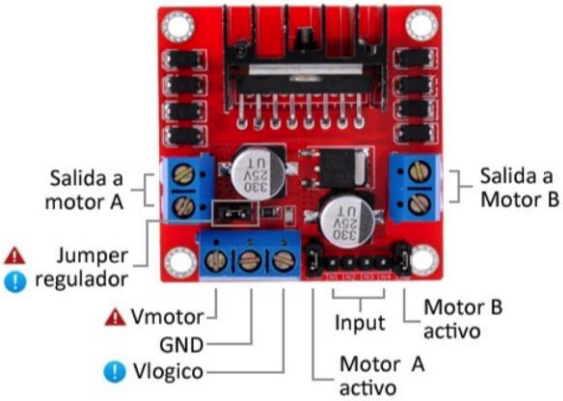

Figure 2: L298N

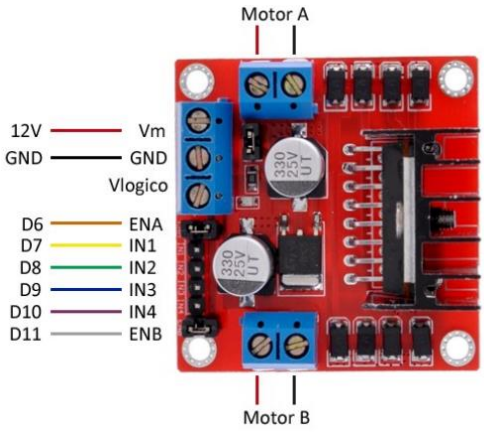

Figure 3: L298N connections to the micro-controller

- **<u>Pins and Connections:</u>**

  - **Motor A/B Outputs**: Connectors for motor outputs, allowing the control of two DC motors or a stepper motor.
  - **Input 1, 2, 3, 4**: Input signals determining the direction of the motors. Controlled through microcontroller GPIOs.
  - **EnA, EnB (Enable A/B)**: Enable or disable motor outputs. Also used to control speed through PWM.
  - **Vmotor** (Motor Voltage): Input for motor voltage (up to 35V).
  - **Vlogic** (Logic Voltage): Input for controller logic voltage (5V).
  - **GND**: Common ground for motor and logic voltage.

<u>Applications in the Project</u>: The controller is essential for the movement and direction of the autonomous vehicle, enabling fine adjustments in the trajectory and speed in response to data from infrared and sonic sensors.

2.3. Infrared Sensors

Infrared line tracking sensors are devices that use an IR emitter and receiver to detect the reflectivity of the surface beneath them, distinguishing between a tracking line and the background. Their main use in this project is for line detection on the ground, allowing the vehicle to follow a predetermined path, such as a black line on a light background.

When the IR emitter sends light, it reflects off the surface and returns to the receiver. A dark line will absorb more light, changing the signal in the digital output that the microcontroller can interpret. This sensor is compatible with an operating voltage of 5 Volts, making it highly compatible with most microcontrollers.

When the IR sensor is positioned over a white surface, both sensor LEDs light up due to the higher reflectivity of white, causing the infrared light to bounce back to the sensor. Conversely, when it detects black, which absorbs infrared light, the LEDs turn off, indicating successful detection of the black line. In this situation, due to the digital pin configuration, the sensor sends 5V to the microcontroller to signal the detection of the black line.

Such sensors can be seen in Figure 4.

Figure 4: Infrared Sensors

- **Pins and Connections:**
  - **Vcc**: Connects to the positive power supply.
  - **GND**: Connects to ground.
  - **Out**: Digital output that activates when the line is detected. Connects to a digital input of the microcontroller.

In our project, this sensor has been essential to make the car follow a circuit created by ourselves.

2.4. Ultrasonic Sensor

The HC-SR04 ultrasonic sensor is a distance measurement device that uses ultrasonic sound waves to detect objects and measure the distance to them. The sensor emits a series of ultrasonic waves that bounce off an object and return to the sensor. The time it takes for these waves to return is proportional to the distance. In this project, it is used to detect obstacles in the vehicle's path, allowing the car to stop to avoid collisions. The sensor can be seen in Figure 5.

Figure 5: HC-SR04 Ultrasonic Sensor

- **Pins and Connections:**
  - **VCC:** 5-Volt power supply to the sensor.
  - **Trigger**: Control input to initiate measurement. A high-level pulse is sent to start the measurement.
  - **Echo**: Output that returns a high-level pulse with a duration proportional to the measured distance.
  - **GND**: Ground.

To measure distances, the HC-SR04 ultrasonic sensor emits a series of ultrasonic pulses and waits for the Echo. The timing diagram, illustrated in Figure 6, shows how this signal is generated and received. A 10 µs TTL pulse activates the sensor (rise time), followed by an ultrasonic burst of 8 cycles. The echo signal returns with a pulse width proportional to the distance to the object. Using STM32CubeIDE, we configure a timer in input capture mode to record the rise and fall time of the echo signal. The period of the echo pulse corresponds to the time it takes for the signal to go and come back from the object, allowing us to accurately calculate the distance.

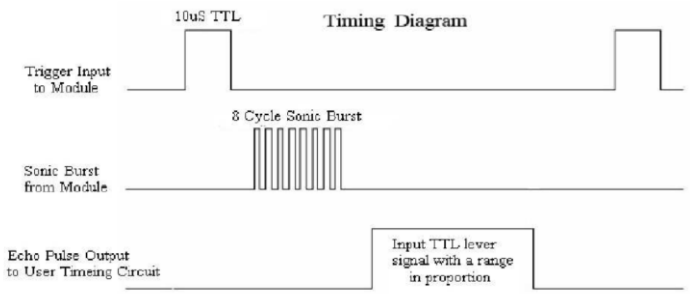

Figure 6: Timing Diagram

2.5. Accelerometer MPU 6050

The MPU-6050 accelerometer is a combined sensor module that includes a gyroscope and an accelerometer, allowing measurement of both angular orientation and linear acceleration. Regarding its operation, the accelerometer measures acceleration in the X, Y, and Z axes, while the gyroscope measures rotation around these axes. These data can be used to calculate the orientation of the vehicle and any changes in its movement or balance. Key features include the sensor's high precision, low power consumption, six degrees of freedom (3 axes of acceleration and 3 of rotation), and its I2C communication capability. In this project, it provides real-time information about the tilt and acceleration of the car, which will be displayed on an LCD screen. The accelerometer can be seen in Figure 7.

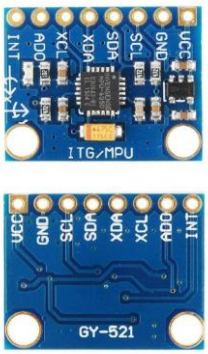

Figure 7 : MPU6050

- **Pins and Connections:**
  - **Vcc**: Positive power supply, typically 3.3V or 5V.
  - **GND**: Ground connection.
  - **SCL y SDA**: I2C communication pins for data transfer to the microcontroller.
  - **INT**: Interrupt output that can be used to signal the microcontroller when data is ready to be read.

2.6. LCD Display

A character LCD (Liquid Crystal Display) is an electronic device capable of displaying information in the form of letters and numbers on a liquid crystal panel. Information is sent to the LCD through dedicated pins for data and control. Users can send characters and specific commands to display information and control how data is presented on the screen.

Key features of these displays include their ability to showcase many characters and their low power consumption, which are essential aspects for projects like ours. Additionally, they often come with backlighting, enhancing visibility in low-light conditions.

In our project, the LCD screen plays a crucial role in the user interface, enabling real-time monitoring and diagnostics of the vehicle's behavior. It provides visual information about measurements from the ultrasonic sensor and the MPU6050 accelerometer-gyroscope, enhancing the understanding and control of the system.

Figure 8: LCD Display

- **Pins and Connections:**

  - **Vcc**: Positive power supply.
  - **GND**: Ground.
  - **VO**: Screen contrast adjustment.
  - **RS (Register Selection)**: Switches between data and command registers.
  - **RW (Read/Write)**: Determines whether reading or writing to the screen.
  - **E (Enable)**: Enables data writing to the registers.
  - **D0-D7**: Data pins for 8-bit communication.
  - **A and K**: Pins for LCD backlighting.

To facilitate communication and programming of the screen, we have adapted the display for I2C communication. To achieve this, we used an I2C adapter module to convert a 16-pin LCD screen interface to a two-wire I2C communication interface, simplifying the connection between the screen and the microcontroller. This can be observed in figure 8.

- **Pins and Connections:**

  - **GND:** Ground.
  - **Vcc:** Positive power source.
  - **SDA:** I2C data line.
  - **SCL:** I2C clock line.

2.7. LED and resistor

Light Emitting Diodes (LEDs) are semiconductor devices that emit light when an electric current is applied. When the microcontroller sends a high-level signal to the pin connected to the LED's anode, it lights up. The LED turns off when the pin is at a low level or is not activated.

In this project, LEDs provide immediate visual feedback on the operational status of the vehicle, which is crucial for user interaction and signaling specific events, such as obstacle detection. The LED will be used with a series resistor to protect it from excessive currents. These components can be seen in Figure 9.

Figure 9: LEDs and resistor

- **Pins and Connections:**
  - **Anode (+):** Connected to a digital output pin of the microcontroller through a current-limiting resistor.
  - **Cathode (-):** Connected to ground (GND).

2.8. Protoboard

A protoboard is a board with interconnected holes that allows for the temporary assembly of electronic circuits without the need for soldering components. In the project, it is used to make quick and modifiable connections between different electronic components, such as the microcontroller, sensors, motor controllers, and LEDs. This board can be seen in Figure 10.

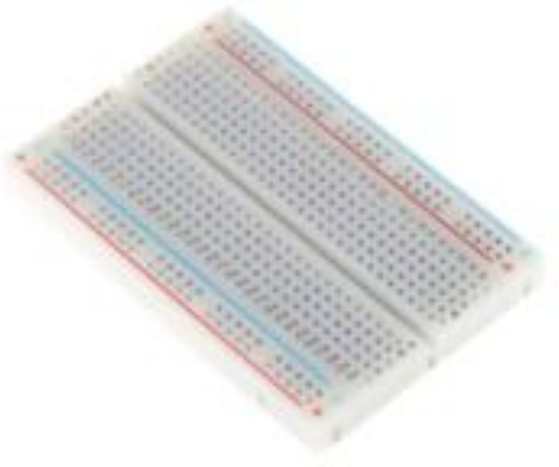

Figure 10: Protoboard

2.9. Batteries and Wires

The batteries supply the necessary electrical power for all the electronic components of the project, while the cables are used to interconnect these components. It is crucial that the batteries have the appropriate charging capacity to support the operation of the vehicle for the desired time and power all the sensors we have used. The cables must have the right length and thickness to ensure the project's good performance, allowing the transmission of electrical and power signals among the various components mounted on the protoboard or connected directly. Due to the high number of sensors and peripherals used, it was necessary to use three batteries, as mentioned in previous sections. These batteries can be seen in Figure 11.

Figure 11: Batteries

2.10. Support

For this project, we have designed a 3D-printed structure to house the vehicle's batteries. This allows for autonomous movement without any issues, providing additional cleanliness and convenience. The structure was created using SolidWorks software, ensuring precise dimensions for integration into the vehicle. However, since this support was developed after the project presentation, we did not consider it necessary to print. Figure 12 shows the design of the support created in SolidWorks.

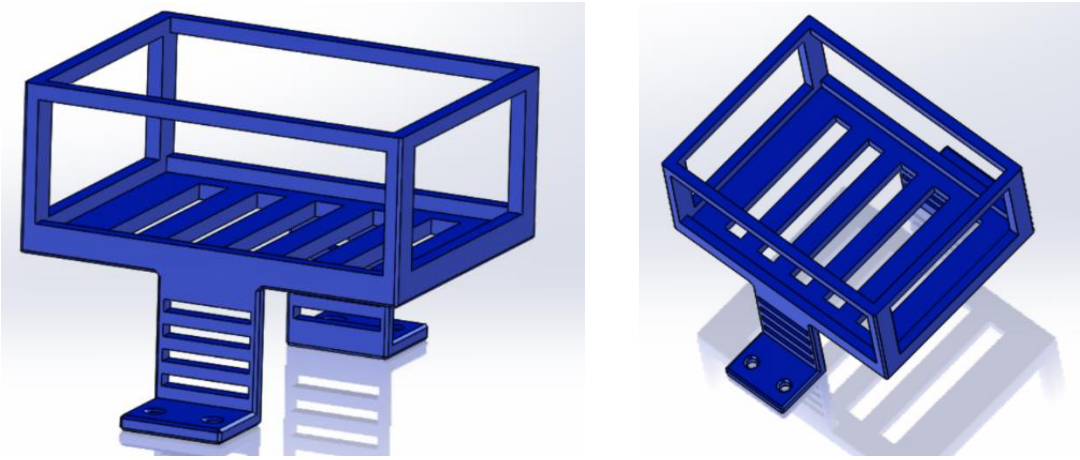

Figure 12: Support developed in SolidWorks

3. CONNECTIONS

Next, we will analyze all the connections made during the project development. To facilitate this task and make it clearer and more organized, we have created a complete schematic of the vehicle, which can be observed in Figure 13.

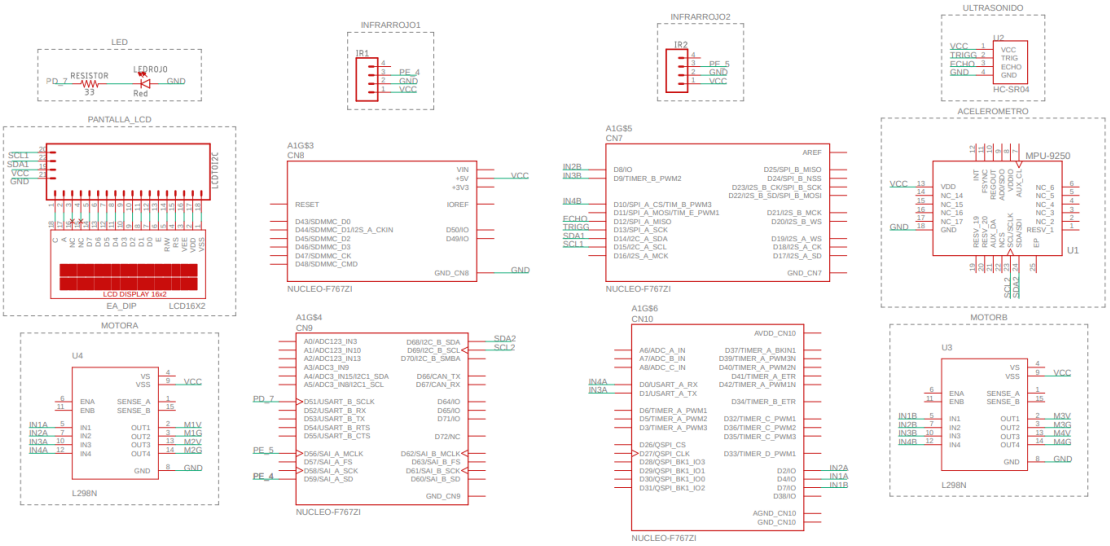

Figure 13: Car's Schematic

As can be seen in Figure 13, at the center are the 4 pin strips of our STM32 board. Around them, all the sensors we have used can be observed, each with its respective pins. In Figures 14 and 15, we can see the fully assembled car with all the sensors connected. As it may seem a bit confusing, that's why we created the schematic to provide a clearer and organized explanation.

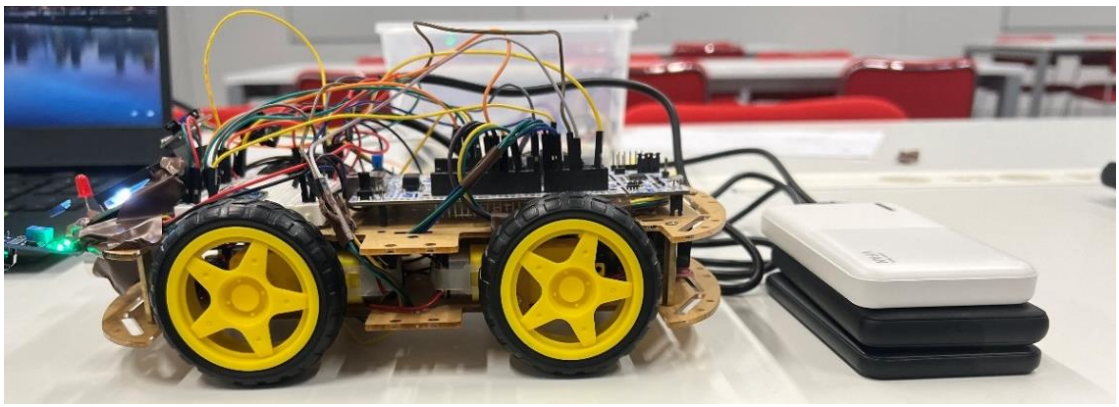

Figure 14 : Car assembly

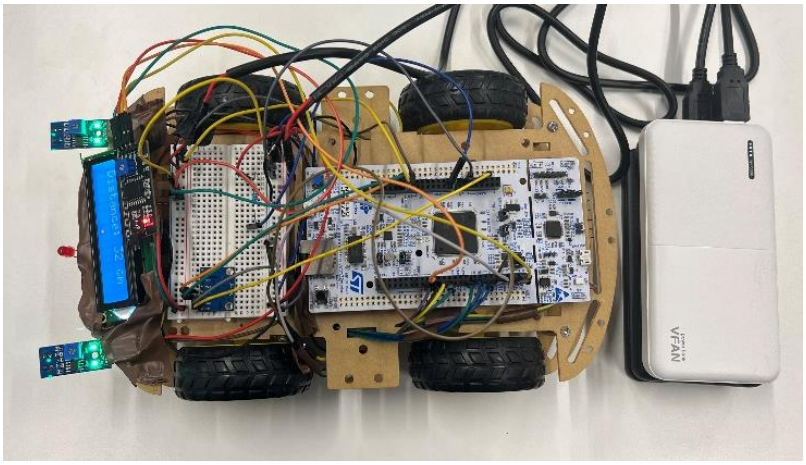

Figure 15: Car assembly

Due to the complexity of our system and the diverse power requirements of the components, we decided to use three separate batteries to optimize the performance and autonomy of the vehicle, as mentioned earlier. We designated one battery exclusively for the STM32 board, ensuring stable operation of the central microcontroller. A second battery was allocated to the sensors, providing a constant and clean power supply to maintain accuracy in data collection. The third battery was used for the drivers and motors, given their higher current demand, ensuring adequate power delivery for the locomotion and maneuverability of the car.

Next, we will explain all the connections made, focusing on the different parts of the schematic shown in Figure 13.

3.1. Infrared sensors connections

The specific schematic for the infrared sensors IR1 and IR2 can be seen in figure 16, where the following connections can be observed:

- **Vcc:** Both sensors are powered with +3.3V to ensure their proper operation.
- **GND:** The grounds of both sensors are connected to the common ground point in the system.
- **Out:** The outputs of the sensors, IR1 connected to pin PE_4 and IR2 to pin PE_5, send digital signals to the microcontroller STM32F767ZI. These signals indicate whether each sensor is over a dark or light surface, which is interpreted by the microcontroller to adjust the car's trajectory and keep it following the black line of the created circuit.

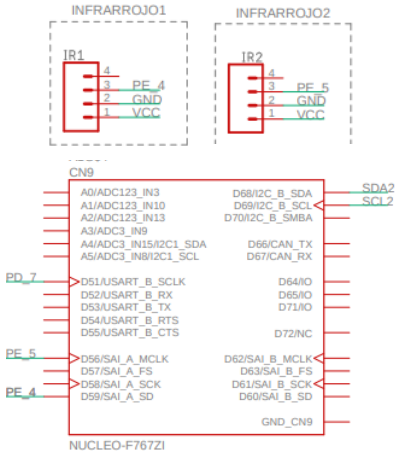

Figure 16: Schematic of the connections of the infrared sensors

These pins will later be initialized as "Digital Inputs" to obtain the signal from the sensors and perform their function.

3.2. Ultrasonic Sensor connections

The specific schematic of the HC-SR04 ultrasonic sensor can be seen in Figure 17, and the connections are as follows:

- **VCC**: Connected to 5V to power the sensor.
- **Trigger**: Connected to pin D13 of the STM32F767ZI microcontroller. This pin is used to send ultrasonic pulses. When activated, it emits a high-frequency sound that is not audible to humans.
- **Echo**: Connected to pin D12 of the microcontroller. This pin receives the echo of the ultrasonic pulse that bounces off objects in front of the sensor.
- **GND**: Connected to ground.

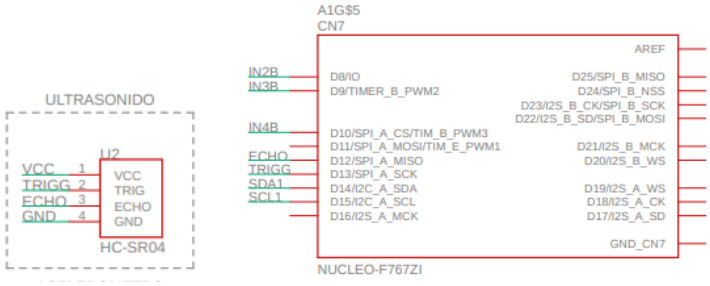

Figure 17: Schematic of the connections of the ultrasonic sensor

The microcontroller activates the TRIGG pin by sending a brief high pulse. Subsequently, the sensor emits an ultrasonic chirp. If there is an object in front of the sensor, the sound bounces back to it and is detected at the ECHO pin. The duration of the ECHO pulse is proportional to the distance to the object. In our program, this information is used by the microcontroller to determine the proximity of obstacles, and if necessary, it can stop the vehicle to avoid collisions.

3.3. Accelerometer MPU 6050 connections

The schematic of the MPU-6050 accelerometer can be seen in Figure 18 and it connects to the STM32F767ZI microcontroller as follows:

- **Vcc**: Connected to +3V3 to power the sensor.
- **GND**: Connected to ground.
- **SCL**: Connected to the microcontroller's I2C_SCL pin, used for the I2C clock line.
- **SDA**: Connected to the microcontroller's I2C_SDA pin, used for the I2C data line.

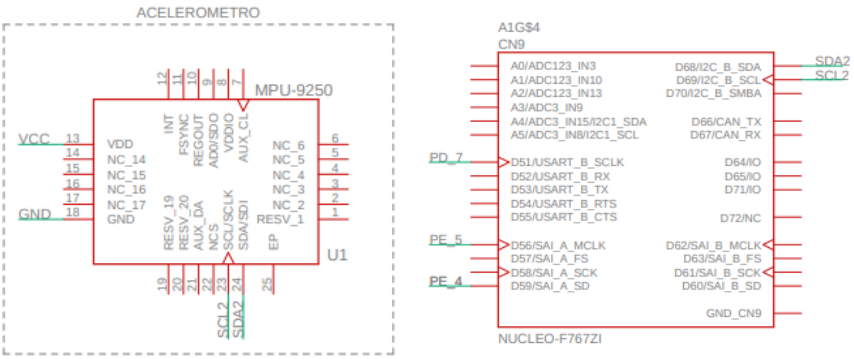

Figure 18: Schematic of the connections of the accelerometer

The MPU-6050 provides accurate information about the orientation and movement of the car by measuring acceleration on the X, Y, and Z axes and angular velocity through the integrated gyroscope. Communication through I2C enables efficient data transfer and simplifies the wiring by requiring only two wires for communication with the microcontroller.

3.4. LCD Display connections

The specific schematic for the LCD screen can be seen in Figure 19, and it is connected and configured as follows:

- **Vcc**: Powered with +5V.
- **GND**: Connected to ground.
- **SCL**: Connected to the I2CA_SCL pin of the STM32F767ZI microcontroller for the I2C clock line.
- **SDA**: Connected to the I2CB_SDA pin for the I2C data line.

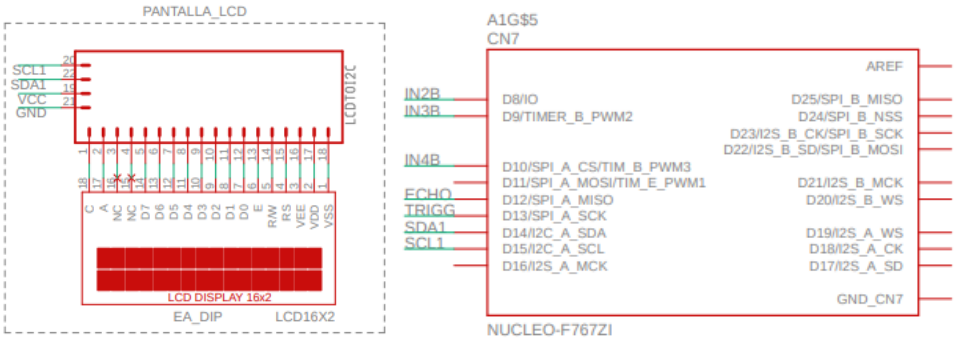

Figure 19: Schematic of the connections of the LCD display

I2C communication reduces the number of pins required to operate the LCD screen, facilitating its integration into the project. The LCD screen displays information, such as the distance to objects detected by the ultrasonic sensor and readings from the accelerometer, allowing real-time monitoring of the car's state and behavior.

3.5. LED and resistor connections

The specific schematic for the LED and resistor is shown in Figure 20, and its connection is made as follows:

- **Anode (+):** Connected to pin PD_7 of the STM32F767ZI microcontroller through a 330-ohm resistor to limit the current and protect the LED.
- **Cathode (-):** Connected to ground (GND).

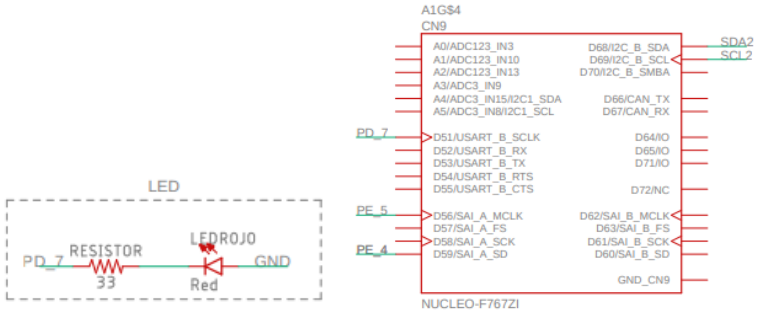

Figure 20: Schematic of the connections of the LED and resistor

When the microcontroller detects an obstacle through the ultrasonic sensor, it sends a high signal to pin PD_7, causing the LED to illuminate, providing a visual indication that the vehicle has stopped. The 330-ohm resistor ensures that the LED receives the appropriate current to operate safely without overloading.

3.6. Drivers connections

We will conclude this section by detailing the schematic of the dual motor drivers L298N, as shown in Figure 21. These drivers will be used to control the vehicle's wheel motors and are connected as follows:

- **Vcc (VS):** Each L298N module is powered by an external source suitable for the motors.
- **GND:** Connected to ground.
- **Control Inputs (IN1, IN2, IN3, IN4):** Connected to microcontroller pins D4, D2, D1, and D0 to control the direction of the motors.
- **Second Controller Control Inputs (IN1, IN2, IN3, IN4):** Connected to microcontroller pins D7, D8, D9, and D10 to control the direction of the motors.
- **Control Outputs (OUT1, OUT2, OUT3, OUT4):** Connected to the two motors.

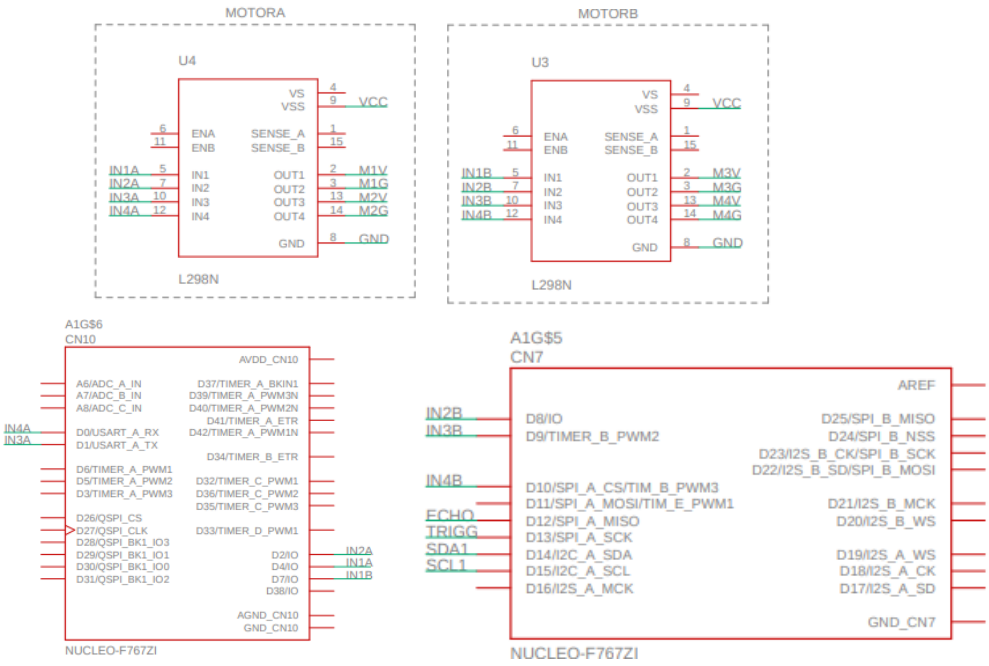

Figure 21: Schematic of the connections of the drivers

4. PIN CONFIGURATION

Once the connection of the various components is complete, we will proceed to configure the pins using the graphical interface of STM32CubeIDE, specifically designed for the STM32F767ZI microcontroller. In Figure 22, all pins configured for the different components of our project can be observed. This graphical interface is essential for the initial configuration of any STM32-based project. It provides a detailed view of the microcontroller package and a visual representation of all available pins, significantly simplifying the process of assigning functions and peripherals.

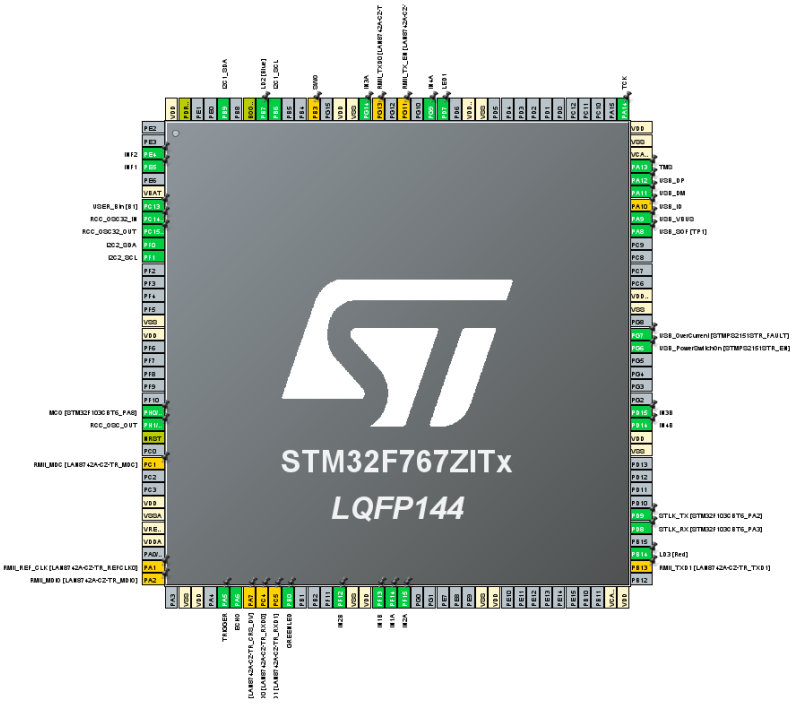

Figure 22: STM32 graphical environment

When starting a new project in STM32CubeIDE, we need to open the file with the. ioc extension, leading us to the pin configuration screen. Here, it is possible to easily configure pins as inputs, outputs, alternative modes for specific peripherals, and analog configurations, among others. This flexibility allows for a robust and adaptive system design, crucial for applications ranging from simple prototypes to complex embedded systems. The tool also helps avoid pin conflicts and configuration errors by providing real-time feedback and warnings if incompatible settings are selected. Additionally, it displays detailed information on the electrical characteristics and capabilities of each pin, which is crucial when designing hardware that requires precision and reliability.

Next, we will proceed to detail specifically the pins configured for each of the sensors and peripherals:

❖ Infrared Sensors

The pins PE4 and PE5 of the STM32 microcontroller are configured as digital inputs in STM32CubeIDE to read the signals from the infrared sensors INF2 and INF1, respectively. These pins are initialized to detect high and low states, which correspond to the detection of white and black surfaces by the sensors. The internal pull-up or pull-down configuration can be adjusted as needed to stabilize the sensor readings. Specifically, when the sensors read zero, it means they are detecting a white surface, while when they read 1, they are detecting a black surface. These pins can be seen in Figure 23.

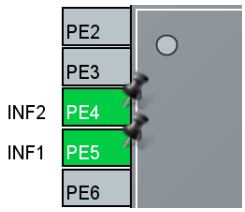

Figure 23: The pins configured for the infrared sensors.

❖ Accelerometer MPU 6050

The pins PF0 and PF1 of the STM32 microcontroller have been configured as I2C2_SDA and I2C2_SCL, respectively, in STM32CubeIDE, establishing the necessary I2C communication for the MPU-6050 accelerometer. These pins enable the transmission of data and clock synchronization between the microcontroller and the sensor. These pins can be seen in Figure 24.

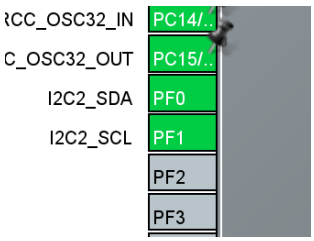

Figure 24: The pins configured for the accelerometer MPU6050.

❖ LCD Display

In STM32CubeIDE, pins PB6 and PB9 are configured to handle I2C communication with the LCD screen. PB6 serves as the I2C clock line (I2C1_SCL), and PB9 serves as the I2C data line (I2C1_SDA). This configuration enables the STM32 microcontroller to communicate with the LCD module through the I2C bus, sending commands and data to control the display of information on the screen. These pins can be seen in Figure 25.

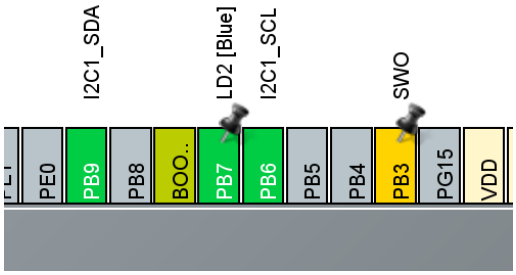

Figure 25: The pins configured for the LCD display.

For the configuration of both the accelerometer and LCD screen pins, we need to go to the connectivity section of the graphical interface, as shown in Figure 26. I2C1 corresponds to the LCD screen, and I2C2 corresponds to the accelerometer.

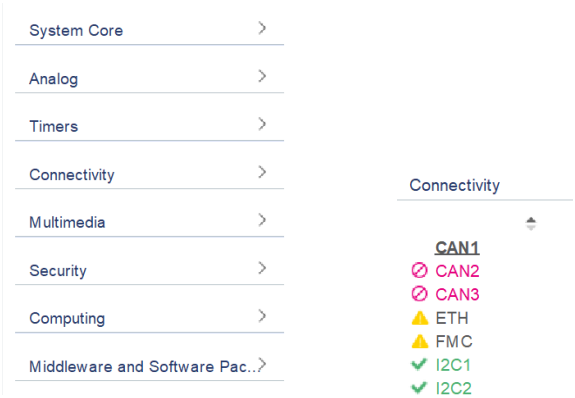

Figure 26: Connectivity Section of the Graphical Interface

❖ Drivers

The pins PF13, PF12, PD15, PD14, PF14, PF15, PG14, and PG9 of the STM32 microcontroller are configured to manage the L298N motor drivers. These pins play a crucial role in controlling and directing the motors connected to the L298N, enabling effective manipulation of the vehicle's movement.

- PF14 and PF15: Configured as digital outputs and connected to the inputs IN1 and IN2 of the first driver to control the direction of Motor A.
- PG14 and PG9: Configured as digital outputs and connected to the inputs IN3 and IN4 of the first driver to control the direction of Motor A.
- PD14 and PD15: Configured as digital outputs and connected to the inputs IN3 and IN4 of the second driver to control the direction of Motor B.
- PF13 and PF12: Configured as digital outputs and connected to the inputs IN2 and IN1 of the second driver to control the direction of Motor B.

These configured pins can be observed in Figure 27.

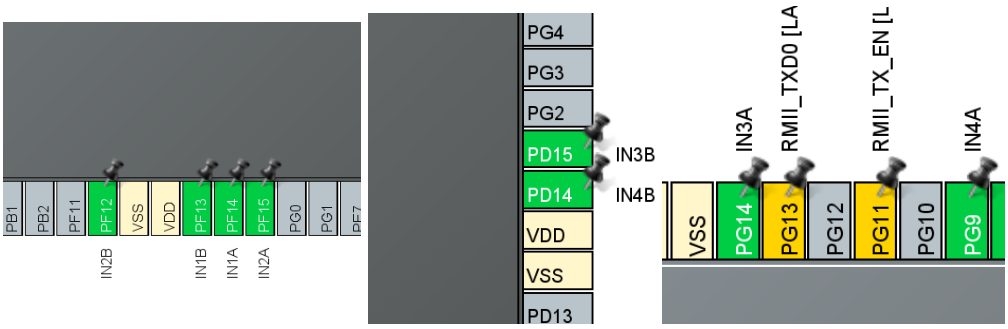

Figure 27: The pins configured for the drivers.

❖ LED

The configuration of pin PD7 on the STM32 microcontroller as a digital output allows controlling LED1. This pin sends signals to the LED, enabling it to turn on or off in response to specific system events. In this case, it will be used to turn on the LED when the distance between the object and the ultrasonic sensor is less than 10 cm. This configuration can be seen in Figure 28.

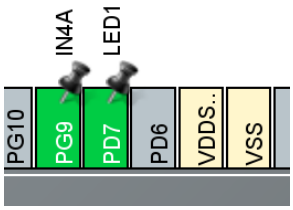

Figure 28: The pins configured for the LED.

❖ Ultrasonic Sensor

We will conclude this section by configuring the necessary pins for the ultrasonic sensor. We will start by setting up pin PA5, which we later renamed as TRIGGER, as an output. This pin will be used to send ultrasonic pulses from the HC-SR04 ultrasonic sensor. By configuring the TRIGGER pin as an output, we allow the microcontroller to control when these ultrasonic pulses are sent. To measure the distance accurately, the sensor emits an ultrasonic pulse and then measures the time it takes for the ECHO pin to receive the pulse reflected by an object.

Next, we will configure the ECHO pin as a timer. To do this, we will navigate to the timer section and specifically select TIM3, using channel 1 of this timer. When choosing the timer, pin PA6 will be automatically selected by default. To ensure that this pin is configured, we will activate the corresponding option in the GPIO settings, as shown in figure 29.

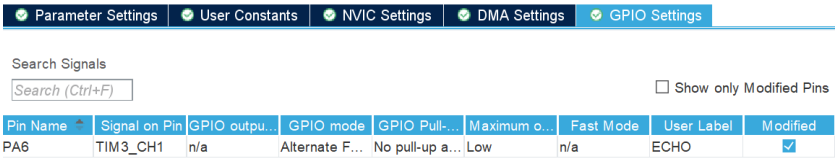

Figure 29: Activation of the timer pin.

Next, we will select the direct input capture mode. Since the microcontroller's internal clock operates at a maximum frequency of 216 MHz, we will configure the prescaler to 215 MHz. This will adjust the timer clock to 1 MHz, which is essential to achieve a time of 1 microsecond per count. Precision in the microsecond range is crucial because the HCSR04 ultrasonic sensors send pulses in a few microseconds, and for accurate pulse width measurement, it is necessary for the timer to operate in this range.

Subsequently, we will choose the maximum value for ARR (Auto Reload Register) for a 16-bit timer, which in this case is 65535. We will also set the initial polarity as rising edge. The configured settings can be observed in figure 30.

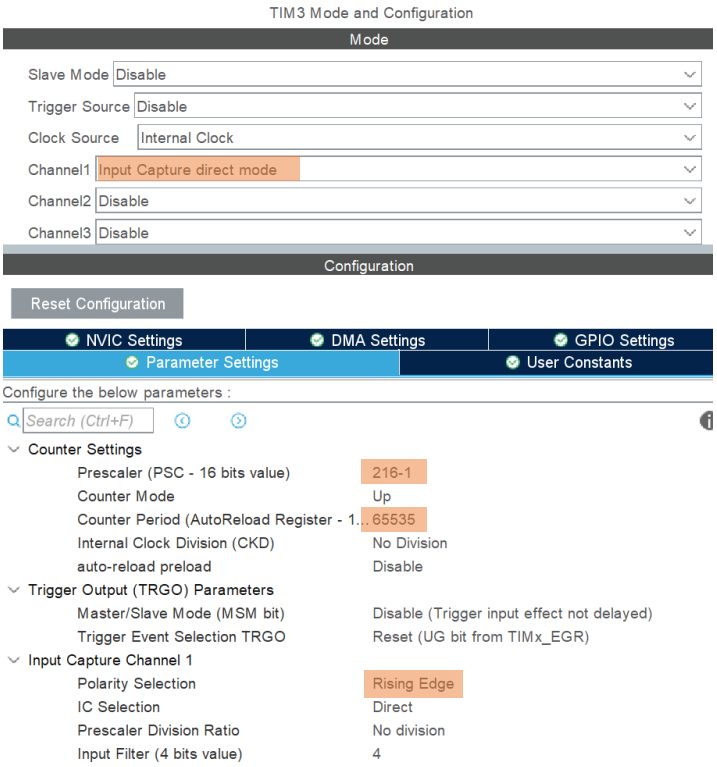

Figure 30: Timer configuration.

Finally, we will enable interrupt capture on the timer, as shown in figure 31.

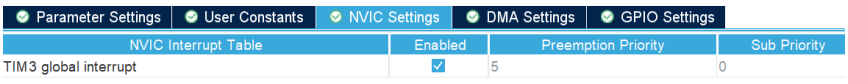

Figure 31: Enable interrupt capture in the timer.

This process ensures an appropriate timer configuration to accurately measure the pulse width of the HCSR04 ultrasonic sensors. In conclusion, the configured TRIGGER and ECHO pins can be observed in figure 32.

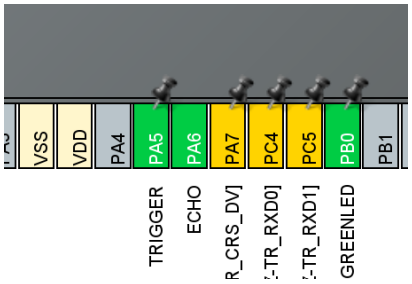

Figure 32: The pins configured for the ultrasonic sensor.

5. ANALYSIS OF THE PSEUDOCODE

In this section, we will create flowcharts for the main program that enables the functionality of the circuit. Subsequently, we will develop another independent flowchart for printing accelerometer values on the LCD.

We will begin by explaining the flowchart depicted in figure 33, which outlines the program's operation. The program starts by reading values provided by the infrared sensors. If both sensors detect white, it means the car should move forward, as it is centered on the circuit. Conversely, if both sensors detect black, it indicates that the circuit has ended, and the car should stop.

In the case where the sensor located on the right end detects the black line while the sensor on the left end detects the white surface, the car will turn right while moving forward. Similarly, if the sensor detecting the black surface is on the left, while the one on the right detects the white surface, the car will turn left while moving forward.

Simultaneously with the infrared sensor readings, the microcontroller will emit a pulse to the TRIGGER pin for 10 ms and then cease pulse emission. This will give us the period, which is the difference between the rise and fall times of the ultrasonic pulse, thus obtaining the time it takes for the pulse to travel from the TRIGGER pin to the object and back to the ECHO pin. By dividing this period by 58, we obtain the distance in centimeters to which the object is from the ultrasonic sensor. Once this distance is obtained, it will be printed on the LCD at 1-second intervals.

We decided to print the distance instead of accelerometer values, as we consider it more useful and in harmony with the rest of the project. Additionally, a check will be performed to see if the distance is less than 10 cm; in that case, the car will stop, and a red LED will light up as a warning. Otherwise, the car will continue moving guided by the infrared sensors, and the LED will remain off.

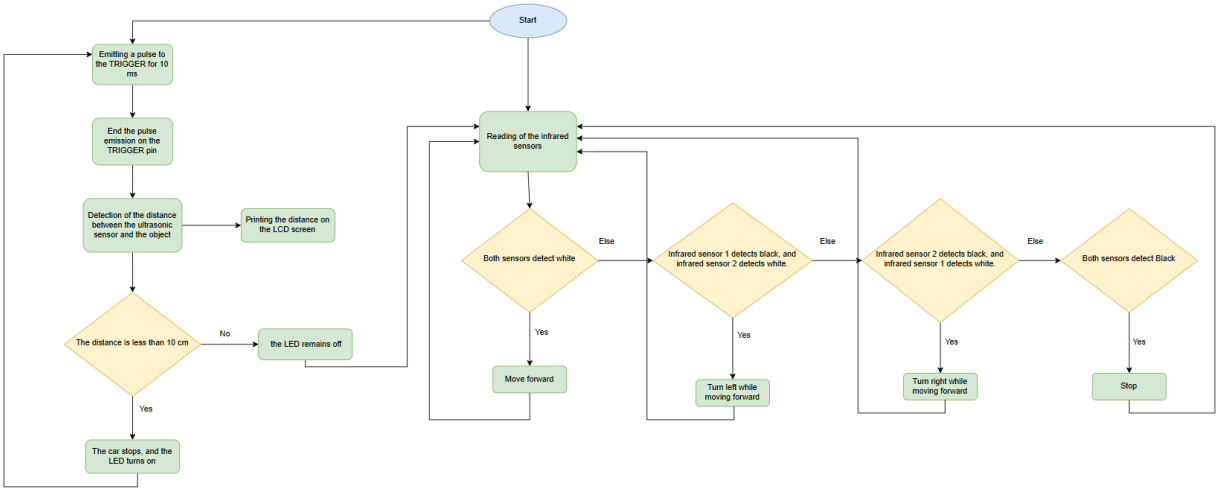

Figure 33: Flowchart of the main program

Finally, we will explain the operation of the flowchart designed to confirm the proper functioning of the accelerometer. This code will be generated independently of the rest of the project, with the sole purpose of verifying the accuracy of the accelerometer. The reason behind this decision lies in the limitations of our LCD, which has only two rows. To print the six necessary measurements (three accelerations and three angular velocities), the values will be printed two at a time. Between each pair of values, the screen will be cleared, and a one-second delay will be introduced before printing the next values. This strategy is necessary to allow the values to be correctly reflected on the screen.

Since the car moves quickly, we do not consider it realistic to print accelerometer values during its movement. Therefore, this code runs with the car stationary. We call the functions responsible for reading the values provided by the MPU6050 sensor and print them on the screen. After one iteration, we will change the car's position to observe the changes reflected in the acceleration and velocity measurements. This flowchart is depicted in Figure 34.

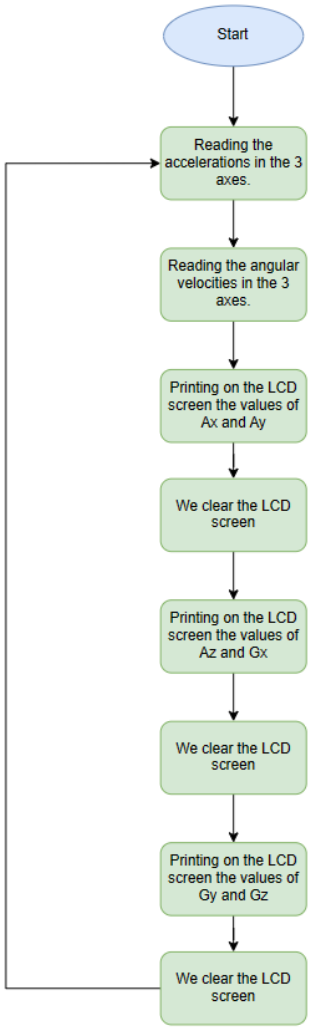

Figure 34: Flowchart for printing accelerometer values on the screen

6. ANALYSIS OF THE DEVELOPED CODE DEVELOPED IN STM32CubeIDE

After analyzing the flowchart, we will proceed to briefly explain the developed code, detailing the declared global variables as well as the created functions. These functions will be used in various tasks coordinated by two semaphores.

6.1 Declaration of global variables

We will start by analyzing the global variables used in the configuration of the ultrasonic sensor. The variables **rise_time** and **fall_time** will be used to store the rise and fall times, respectively, while **period** will represent the difference between these two times.

As for the variable **DISTANCE_THRESHOLD**, it will determine the distance at which the action of stopping the vehicle will be triggered. In other words, when an object is less than 10 cm away from the ultrasonic sensor, the car will automatically come to a halt. The variable **distance** will be used to store the distance between the sensor and the detected object. Additionally, the variable **count** will indicate the number of readings performed.

The slave address will be set using **SLAVE_ADDRESS_LCD** with the value **0x4E** in the STM32 microcontroller. This address will determine which device will send or receive data through the I2C communication protocol.

Finally, the variable **dist** will be employed to store the measured distance between the sensor and the object. This information will be used to print the distance on the LCD screen, providing a visual interface for the measurement taken. All these variables can be observed in code snippet 1.

```
1 uint32_t rise_time, fall_time, period = 0;
2 uint8_t count = 0;
3 int distance;
4 char dist[6];
5 #define SLAVE_ADDRESS_LCD 0x4E
6 #define DISTANCE_THRESHOLD 10
```

Code 1: Global variables required for ultrasound configuration.

Next, we will detail the global variables related to the accelerometer configuration. These variables can be observed in code snippet 2 and are as follows:

- **SMPLRT_DIV_REG** (Sample Rate Divider Register): Specifies the register to configure the sampling frequency of the MPU6050.
- **GYRO_CONFIG_REG** (Gyroscope Configuration Register): Represents the register used to configure the sensitivity and range of the gyroscope.
- **ACCEL_CONFIG_REG** (Accelerometer Configuration Register): Defines the register to configure the sensitivity and range of the accelerometer.
- **ACCEL_XOUT_H_REG** (Accelerometer X-axis High Byte Register): Indicates the register that stores the most significant byte of the raw acceleration value in the X-axis.
- **TEMP_OUT_H_REG** (Temperature Output High Byte Register): Represents the register that stores the most significant byte of the temperature measured by the sensor.
- **GYRO_XOUT_H_REG** (Gyroscope X-axis High Byte Register): Indicates the register that stores the most significant byte of the raw angular velocity value in the X-axis.
- **PWR_MGMT_1_REG** (Power Management 1 Register): This register is used to manage the power configuration and suspension modes of the MPU6050.
- **WHO_AM_I_REG** (Who Am I Register): It is an identification register that contains a specific value allowing verification of whether the MPU6050 is present at the expected I2C address. It helps confirm successful communication with the device.
- **MPU6050_ADDR**: It is the I2C address of the MPU6050. This value is used to communicate with the sensor through the I2C bus.

The variables **Accel_X_RAW**, **Accel_Y_RAW**, and **Accel_Z_RAW** will store the raw values of acceleration in the X, Y, and Z axes, directly read by the sensor. Meanwhile, the variables **Ax**, **Ay**, and **Az** will hold the acceleration values converted to gravity units (g) in the X, Y, and Z axes, respectively.

On the other hand, the variables **Gyro_X_RAW**, **Gyro_Y_RAW**, and **Gyro_Z_RAW** will store the raw values of angular velocity in the X, Y, and Z axes, directly obtained from the MPU6050 sensor. In contrast, the variables **Gx**, **Gy**, and **Gz** will contain the angular velocity values converted to units of degrees per second (°/s) in the X, Y, and Z axes, respectively.

Finally, the variable **but** will be used to store the values of the gyroscope and the accelerometer, with the purpose of printing them later the LCD screen.

```
#define SMPLRT_DIV_REG 0X19
#define GYRO_CONFIG_REG 0X1B
#define ACCEL_CONFIG_REG 0X1C
#define ACCEL_XOUT_H_REG 0X3B
#define TEMP_OUT_H_REG 0X41
#define GYRO_XOUT_H_REG 0X43
#define PWR_MGMT_1_REG 0X6B
#define WHO_AM_I_REG 0X75
#define MPU6050_ADDR 0x68
char but[6];
float Accel_X_RAW, Accel_Y_RAW, Accel_Z_RAW;
float Gyro_X_RAW, Gyro_Y_RAW, Gyro_Z_RAW;
float Ax, Ay, Az;
float Gx, Gy, Gz;
```

Code 2: Global variables required for accelerometer configuration.

Finally, we will declare the variables **Infrared1** and **Infrared2**, which will be responsible for storing the values captured by Infrared sensors 1 and 2, respectively. These variables can be observed in code snippet 3.

```
int Infrared1 = 0;
int Infrared2 = 0;
```

Code 3: Global variables required for infrared configuration.

6.2 Development of the functions

In this section, we will address the functions that will enable the car's movement, as well as those dedicated to configuring the ultrasonic sensor, the MPU6050 accelerometer, and the LCD screen.

- **Movements functions**

In this section, we will detail the functions that enable the vehicle's movement in various directions. We have adopted a systematic approach by creating specific functions for each direction and strategically invoking them at key points in the code. This approach not only improves code readability but also reduces unnecessary complexity.

In our design, the functions governing the vehicle's movement encompass actions such as moving forward, backward, turning right, turning left, and stopping. The execution of each of these movements depends on the careful configuration of the state of the pins associated with the forward or backward motion of the corresponding wheel, alternating between high

and low states. This precise coordination is smoothly executed through the use of the built-in *HAL_GPIO_WritePin()* function.

Each motor has two input pins: IN1, IN2 for the left wheel and IN3, IN4 for the right wheel. To make the vehicle move, it is necessary to set the state of these pins as follows:

- Both pins HIGH/LOW: the wheel does not move.
- One pin HIGH, the other LOW: the wheel moves in one direction.
- One pin LOW, the other HIGH: the wheel moves in the opposite direction.

To make the car turn, we simply move both wheels to one side, while the other pair remains stationary. It is important to note that our two controllers are not aligned in the same direction. Specifically, the controller at the rear is oriented backward in relation to the direction of movement. All these functions can be found in code snippet 4.

```
void movingBackward(){
    //motor A wheels
    HAL_GPIO_WritePin(IN1A_GPIO_Port, IN1A_Pin,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(IN2A_GPIO_Port, IN2A_Pin,GPIO_PIN_SET);
    HAL_GPIO_WritePin(IN3A_GPIO_Port, IN3A_Pin,GPIO_PIN_SET);
    HAL_GPIO_WritePin(IN4A_GPIO_Port, IN4A_Pin,GPIO_PIN_RESET);
    //motor B wheels
    HAL_GPIO_WritePin(IN1B_GPIO_Port, IN1B_Pin,GPIO_PIN_SET);
    HAL_GPIO_WritePin(IN2B_GPIO_Port, IN2B_Pin,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(IN3B_GPIO_Port, IN3B_Pin,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(IN4B_GPIO_Port, IN4B_Pin,GPIO_PIN_SET);
}
void movingForward(){
  //motor A wheels
    HAL_GPIO_WritePin(IN1A_GPIO_Port, IN1A_Pin,GPIO_PIN_SET);
    HAL_GPIO_WritePin(IN2A_GPIO_Port, IN2A_Pin,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(IN3A_GPIO_Port, IN3A_Pin,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(IN4A_GPIO_Port, IN4A_Pin,GPIO_PIN_SET);
    //motor B wheels
    HAL_GPIO_WritePin(IN1B_GPIO_Port, IN1B_Pin,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(IN2B_GPIO_Port, IN2B_Pin,GPIO_PIN_SET);
    HAL_GPIO_WritePin(IN3B_GPIO_Port, IN3B_Pin,GPIO_PIN_SET);
    HAL_GPIO_WritePin(IN4B_GPIO_Port, IN4B_Pin,GPIO_PIN_RESET);
}
void stop(){
  //motor A wheels
     HAL_GPIO_WritePin(IN1A_GPIO_Port, IN1A_Pin,GPIO_PIN_RESET);
     HAL_GPIO_WritePin(IN2A_GPIO_Port, IN2A_Pin,GPIO_PIN_RESET);
     HAL_GPIO_WritePin(IN3A_GPIO_Port, IN3A_Pin,GPIO_PIN_RESET);
     HAL_GPIO_WritePin(IN4A_GPIO_Port, IN4A_Pin,GPIO_PIN_RESET);
     //motor B wheels
     HAL_GPIO_WritePin(IN1B_GPIO_Port, IN1B_Pin,GPIO_PIN_RESET);
     HAL_GPIO_WritePin(IN2B_GPIO_Port, IN2B_Pin,GPIO_PIN_RESET);
     HAL_GPIO_WritePin(IN3B_GPIO_Port, IN3B_Pin,GPIO_PIN_RESET);
     HAL_GPIO_WritePin(IN4B_GPIO_Port, IN4B_Pin,GPIO_PIN_RESET);
}
void turnRightFront(){
  //motor A wheels
    HAL_GPIO_WritePin(IN1A_GPIO_Port, IN1A_Pin,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(IN2A_GPIO_Port, IN2A_Pin,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(IN3A_GPIO_Port, IN3A_Pin,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(IN4A_GPIO_Port, IN4A_Pin,GPIO_PIN_SET);
    //motor B wheels
    HAL_GPIO_WritePin(IN1B_GPIO_Port, IN1B_Pin,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(IN2B_GPIO_Port, IN2B_Pin,GPIO_PIN_SET);
    HAL_GPIO_WritePin(IN3B_GPIO_Port, IN3B_Pin,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(IN4B_GPIO_Port, IN4B_Pin,GPIO_PIN_RESET);
}
void turnLeftFront(){
```

```
50    //motor A wheels
51    HAL_GPIO_WritePin(IN1A_GPIO_Port, IN1A_Pin,GPIO_PIN_SET);
52    HAL_GPIO_WritePin(IN2A_GPIO_Port, IN2A_Pin,GPIO_PIN_RESET);
53    HAL_GPIO_WritePin(IN3A_GPIO_Port, IN3A_Pin,GPIO_PIN_RESET);
54    HAL_GPIO_WritePin(IN4A_GPIO_Port, IN4A_Pin,GPIO_PIN_RESET);
55    //motor B wheels
56    HAL_GPIO_WritePin(IN1B_GPIO_Port, IN1B_Pin,GPIO_PIN_RESET);
57    HAL_GPIO_WritePin(IN2B_GPIO_Port, IN2B_Pin,GPIO_PIN_RESET);
58    HAL_GPIO_WritePin(IN3B_GPIO_Port, IN3B_Pin,GPIO_PIN_SET);
59    HAL_GPIO_WritePin(IN4B_GPIO_Port, IN4B_Pin,GPIO_PIN_RESET);
60 }
```

Code 4: Movements functions.

❖ Ultrasonic Configuration Function

Next, we will break down the function **HAL_TIM_IC_CaptureCallback()**, as seen in code snippet 5. This function, designated as a "callback," is associated with capturing events on channel 1 of the TIM3 timer. Its purpose is to calculate the time elapsed between the rising and falling edges, i.e., obtaining the period. The rising time represents the interval the ultrasonic pulse takes to transition from a low state (0V) to a high state (5V). Conversely, the falling time is the period in which the ultrasonic pulse returns from a high state to a low state.

This function begins by checking if the event has indeed occurred on channel 1 of TIM3. If the variable **count** is equal to 0, it indicates that it is the first read, and therefore, a rising time is being captured, as the ECHO pin was initially configured with this polarity. The captured value is stored in the variable **rise_time**. Subsequently, the value of the **count** variable is updated to 1 to indicate that it is the second read, and the polarity is changed. If **count** is equal to 1, indicating that a read has already been performed, we proceed to capture the value of the falling time, which is stored in the variable **fall_time**. Using these data, we calculate the period as the difference between the falling time and the rising time. Then, we reset the value of **count** and change the polarity so that the next read captures a rising time. In subsequent sections of this project, this period will be used to calculate the distance in centimeters between the object and the ultrasonic sensor.

```
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim) {

  if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1) {

    if (count == 0) {
      rise_time = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
      count = 1;
      __HAL_TIM_SET_CAPTUREPOLARITY(htim, TIM_CHANNEL_1, TIM_INPUTCHANNELPOLARITY_FALLING);
    }

    else if (count == 1) {

      fall_time = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
      __HAL_TIM_SET_COUNTER(htim, 0);

      if (fall_time > rise_time) {
        period = fall_time - rise_time;
      }
      count = 0;
      __HAL_TIM_SET_CAPTUREPOLARITY(htim, TIM_CHANNEL_1, TIM_INPUTCHANNELPOLARITY_RISING);
    }
  }
}
```

Code 5: Function to configure the ultrasound.

❖ LCD Screen Configuration Functions

Next, we will detail the functions used to configure the LCD screen. These functions enable communication between the LCD screen controller and the microcontroller through the I2C bus, facilitating the sending of commands and data to the screen controller.

The **lcd_init2()** function, which can be observed in code 6, is responsible for initializing the LCD screen controller. To achieve this, it will execute a sequence of commands necessary for the proper initialization and configuration of the screen.

```
void lcd_init2(){

    HAL_Delay(50);
    lcd_send_cmd(0x30);
    HAL_Delay(5);
    lcd_send_cmd(0x30);
    HAL_Delay(1);
    lcd_send_cmd(0x30);
    HAL_Delay(10);
    lcd_send_cmd(0x20);
    HAL_Delay(10);


    lcd_send_cmd(0x28);
    HAL_Delay(1);
    lcd_send_cmd(0x08);
    HAL_Delay(1);
    lcd_send_cmd(0x01);
    HAL_Delay(1);
    HAL_Delay(1);
    lcd_send_cmd(0x06);
    HAL_Delay(1);
    lcd_send_cmd(0x0C);
}
```

Code 6: Function to initialize the LCD screen.

We will continue with the **lcd_send_cmd(char cmd)** function, which can be observed in code 7. This function is responsible for sending commands to the LCD screen controller using the I2C protocol. These commands are instructions that control the behavior of the screen, such as the cursor position or clearing the screen. In this function, the RS pin (Register Select) is set to zero, indicating that commands are being sent instead of data.

```
void lcd_send_cmd(char cmd) {

    char data_u, data_l;
    uint8_t data_t[4];

    data_u = cmd&0xf0;
    data_l = (cmd<<4)&0xf0;

    data_t[0] = data_u|0x0C; // en = 1, rs = 0
    data_t[1] = data_u|0x08; // en = 0, rs = 0
    data_t[2] = data_l|0x0C; // en = 1, rs = 0
    data_t[3] = data_l|0x08; // en = 0, rs = 0

    HAL_I2C_Master_Transmit(&hi2c1, SLAVE_ADDRESS_LCD,(uint8_t *) data_t, 4, 100);

}
```

Code 7: Function to send commands to the LCD screen.

On the other hand, the **lcd_send_data(char data)** function, which can be observed in code 8, is responsible for sending data to the LCD screen controller. This data consists of characters that will be displayed on the LCD screen, whether it's ASCII text, graphics, or other desired information. In this case, the RS pin will be configured as 1, indicating that what is being sent are data and not commands.

```
void lcd_send_data(char data) {

    char data_u, data_l;

    uint8_t data_t[4];

    data_u = (data&0xf0);
    data_l = ((data<<4)&0xf0);

    data_t[0] = data_u|0x0D; // en = 1, rs = 1
    data_t[1] = data_u|0x09; // en = 0, rs = 1
    data_t[2] = data_l|0x0D; // en = 1, rs = 1
    data_t[3] = data_l|0x09; // en = 0, rs = 1

    HAL_I2C_Master_Transmit(&hi2c1, SLAVE_ADDRESS_LCD,(uint8_t *) data_t, 4, 100);
}
```

Code 8: Function to send commands to the LCD screen.

Finally, the function **lcd_send_string(char* str)**, seen in code 9, will be responsible for sending a string of characters to the LCD screen controller. To achieve this, it uses the **lcd_send_data()** function, as seen in code 8, to send each character of the string one by one until it reaches the null character ('\0'), indicating the end of the string.

```
1 void lcd_send_string(char*str){
2   while(*str) lcd_send_data(*str++);
3 }
```

Code 9: Function to send a string of characters to the LCD screen.

❖ Accelerometer Configuration Functions

We will conclude this section by detailing the functions used for the configuration of the MPU6050 accelerometer. We will begin by analyzing the initialization function, **MPU6050_Init ()**, as presented in code snippet 10. This function starts by reading the value of the identification register, **WHO_AM_I_REG**. If the read value is 104, which is the correct identification for the MPU6050, additional configurations are performed for its initialization. These configurations include writing zero to the **PWR_MGMT_1_REG** register to activate the sensor and exit sleep mode. Additionally, 0x07 is written to the **SMPLRT_DIV_REG** register to configure the sample rate divider, and zero is written to the **ACCEL_CONFIG_REG** and **GYRO_CONFIG_REG** registers to set the ranges and sensitivity of the accelerometer and gyroscope, respectively.

```
1  void MPU6050_Init(void){
2    uint8_t check, Data;
3    HAL_I2C_Mem_Read(&hi2c2, MPU6050_ADDR , WHO_AM_I_REG, 1, &check, 1, 1000);
4
5    if (check == 104){
6
7    Data = 0;
8    HAL_I2C_Mem_Write(&hi2c2, MPU6050_ADDR , PWR_MGMT_1_REG, 1, &Data, 1, 1000);
9
10   Data = 0x07;
11   HAL_I2C_Mem_Write(&hi2c2, MPU6050_ADDR , SMPLRT_DIV_REG, 1, &Data, 1, 1000);
12
13   Data = 0x00;
14   HAL_I2C_Mem_Write(&hi2c2, MPU6050_ADDR , ACCEL_CONFIG_REG, 1, &Data, 1, 1000);
15
16   Data = 0x00;
17   HAL_I2C_Mem_Write(&hi2c2, MPU6050_ADDR , GYRO_CONFIG_REG, 1, &Data, 1, 1000);
18
19   }
20 }
```

Code 10: Function to initialize the accelerometer.

We will continue with the **MPU6050_Read_Accel ()** function, which can be observed in code snippet 11. The purpose of this function is to read raw data from the accelerometer and calculate accelerations in all three directions. Using **ACCEL_XOUT_H_REG**, raw accelerometer data is read and converted into the variables **Accel_X_RAW**, **Accel_Y_RAW**, and **Accel_Z_RAW** to distinguish acceleration along each axis. Subsequently, we calculate accelerations in the X, Y, and Z directions in units of gravity (g) and store them in the variables **Ax**, **Ay**, and **Az**.

```
void MPU6050_Read_Accel(void){

  uint8_t Rec_Data[6];

  HAL_I2C_Mem_Read (&hi2c2, MPU6050_ADDR, ACCEL_XOUT_H_REG, 1, Rec_Data, 6, 1000);

  Accel_X_RAW = (int16_t)(Rec_Data[0] << 8 | Rec_Data [1]);
  Accel_Y_RAW = (int16_t)(Rec_Data[2] << 8 | Rec_Data [3]);
  Accel_Z_RAW = (int16_t)(Rec_Data[4] << 8 | Rec_Data [5]);

  Ax = Accel_X_RAW/16384.0;
  Ay = Accel_Y_RAW/16384.0;
  Az = Accel_Z_RAW/16384.0;
}
```

Code 11: Function to configure the accelerometer.

Finally, we will explain the **MPU6050_Read_Gyro ()** function, which can be observed in code snippet 12. This function aims to read raw data from the gyroscope using **GYRO_XOUT_H_REG** and subsequently convert those raw values into the variables **Gyro_X_RAW**, **Gyro_Y_RAW**, and **Gyro_Z_RAW**, to distinguish angular velocities in the three axes. Following that, we will calculate angular velocities in the X, Y, and Z directions in units of degrees per second (°/s) and store them in the variables **Gx**, **Gy**, and **Gz**.

```
void MPU6050_Read_Gyro(void){

  uint8_t Rec_Data[6];

  HAL_I2C_Mem_Read (&hi2c2, MPU6050_ADDR, GYRO_XOUT_H_REG, 1, Rec_Data, 6, 1000);

  Gyro_X_RAW = (int16_t)(Rec_Data[0] << 8 | Rec_Data [1]);
  Gyro_Y_RAW = (int16_t)(Rec_Data[2] << 8 | Rec_Data [3]);
  Gyro_Z_RAW = (int16_t)(Rec_Data[4] << 8 | Rec_Data [5]);

  Gx = Gyro_X_RAW/131.0;
  Gy = Gyro_Y_RAW/131.0;
  Gz = Gyro_Z_RAW/131.0;
}
```

Code 12: Function to configure the gyroscope.

6.3. Task coordination and semaphores

In this section, we will begin by explaining the main program, detailing the various tasks created, as well as their coordination through semaphores. Additionally, we will provide details about the additional program, which has the sole purpose of verifying the accelerometer's functionality by printing its values on the LCD screen.

❖ Main program

We will begin by briefly explaining the two binary semaphores created through the STMCubeIDE graphical interface. These semaphores, visible in code snippet 13, are designed to have values of only 0 and 1.

```
1  movingSemHandle = osSemaphoreNew(1, 1, &movingSem_attributes);
2  stopSemHandle = osSemaphoreNew(1, 0, &stopSem_attributes);
```

Code 13: semaphores.

The semaphore **movingSemHandle** is initialized with a value of 1, indicating that it is available to be acquired by any task. On the other hand, the **stopSemHandle** semaphore is initialized to zero, meaning that its value must be incremented by one before it can be used. Now that we have introduced the semaphores that will be used in coordinating the tasks, we will proceed to explain these tasks in detail.

We will begin by detailing the Infrared task, which can be observed in code snippet 14. This task starts by acquiring the **movingSemHandle** semaphore, decrementing its value by one. The responsibility of this task is to control the movements of the car, and the choice of the movement to perform is based on the information received from the infrared sensors. The task begins by reading the values from the two infrared sensors. If both values are zero, it means they are detecting a white surface, and the car will move forward. Conversely, if both sensors have a value of 1, it indicates they are both detecting the black line, signaling the end of the circuit. On the other hand, if one sensor reads 1 and the other reads 0, the car will turn left or right while moving forward, depending on whether the sensor detecting the black line is on the left or right. This task concludes by releasing the **movingSemHandle** semaphore, incrementing its value by one, allowing the semaphore to be acquired again by the same or other tasks.

```
void Infrared(void *argument) {
    while (1) {
      osSemaphoreAcquire(movingSemHandle, osWaitForever);

      Infrared1 = HAL_GPIO_ReadPin(INF1_GPIO_Port, INF1_Pin);
      Infrared2 = HAL_GPIO_ReadPin(INF2_GPIO_Port, INF2_Pin);


       if (Infrared1 == GPIO_PIN_RESET && Infrared2 == GPIO_PIN_RESET ) {

           movingForward();
       }

       else if (Infrared1 == GPIO_PIN_SET && Infrared2 == GPIO_PIN_RESET) {

           turnLeftFront();

       }

       else if (Infrared1 == GPIO_PIN_RESET && Infrared2 == GPIO_PIN_SET) {

           turnRightFront();
       }

       else if (Infrared1 == GPIO_PIN_SET && Infrared2 == GPIO_PIN_SET) {

           stop();
       }

       osSemaphoreRelease(movingSemHandle);

    }
}
```

Code 14: Infrared task.

We will continue with the **DetectTask** task, which is observed in code 15. This task starts by raising the **TRIGGER** pin, causing the emission of an ultrasonic pulse with a duration of 10 ms. Subsequently, the emission of this pulse is deactivated. Additionally, events capturing is enabled on channel 1 of timer 3, allowing the retrieval of the period, i.e., the difference between the falling and rising times. Once this period is calculated, the distance in centimeters to the object of the ultrasonic sensor is determined.

Next, it is checked whether this distance is less than 10 cm. If affirmative, a red LED is lit as a warning, and the **movingSemHandle** semaphore is captured, decreasing its value by one unit. This prevents the **Infrared** task from running again. The value of the **stopSemHandle** semaphore is also increased by one unit, allowing the **StopTask** task, visible in code 16, to capture it. The **StopTask** task acquires the **stopSemHandle** semaphore, decreases its value by one unit, and then calls the **stop** function. After the stopping time has elapsed, it releases the **movingSemHandle** semaphore, increasing it by one unit, enabling other tasks to capture it. If the distance continues to be less than 10 cm, the **movingSemHandle** semaphore will be captured again by the **DetectTask** task, keeping the car stopped and the LED turned on. Otherwise, if the distance is greater than 10 cm, the LED remains off, and the **movingSemHandle** semaphore is acquired by the **Infrared** task.

```
void DetectTask(void *argument){

    while(1){

    HAL_GPIO_WritePin(TRIGGER_GPIO_Port, TRIGGER_Pin,GPIO_PIN_SET);
    HAL_Delay(10);
    HAL_GPIO_WritePin(TRIGGER_GPIO_Port, TRIGGER_Pin,GPIO_PIN_RESET);

    HAL_TIM_IC_Start_IT(&htim3, TIM_CHANNEL_1);
    distance = period/58;


    if (distance < DISTANCE_THRESHOLD){
        HAL_GPIO_WritePin ( LED1_GPIO_Port , LED1_Pin , GPIO_PIN_SET ) ;
        osSemaphoreAcquire(movingSemHandle, osWaitForever);
        osSemaphoreRelease(stopSemHandle);
    } else {

        HAL_GPIO_WritePin(LED1_GPIO_Port, LED1_Pin, GPIO_PIN_RESET);

              }
          }
      }
```

Code 15: Detect Task.

```
void StopTask(void *argument){
  while(1){

     osSemaphoreAcquire(stopSemHandle,osWaitForever);
     stop();
     vTaskDelay(500 / portTICK_PERIOD_MS);
     stop();
     osSemaphoreRelease(movingSemHandle);
  }
}
```

Code 16: Stop Task.

Independently and without coordination through semaphores, as it is a task that we want to be executed continuously, is the **LCDScreenTask**, visible in code 17. This task is responsible for displaying on the LCD screen the distance between the detected object and the ultrasonic sensor. The printing of this data is done at one-second intervals. This time interval is chosen because a shorter interval would not allow the data to be printed properly; printing too soon could prevent the task from having enough time to display the values correctly.

```
void LCDScreenTask(void *argument){
  while(1){

    sprintf(dist, "%d", distance);
    lcd_send_cmd(0x80);
    lcd_send_string(" Ultrasonido");

    lcd_send_cmd(0x80);
    lcd_send_string("Distance: ");
    lcd_send_string(dist);
    lcd_send_string(" cm");
    lcd_send_string(" ");
    vTaskDelay(1000 / portTICK_PERIOD_MS);
  }
}
```

Code 17: LCD screen Task.

The invocation of these tasks will be carried out as shown in code 18, assigning them all the same priority. The coordination of their execution will be managed using semaphores, as we have just explained.

```
if (xTaskCreate(Infrared, "task1", stackSize, NULL, tskIDLE_PRIORITY, NULL) == pdPASS){
 if (xTaskCreate(StopTask, "task2", stackSize, NULL, tskIDLE_PRIORITY, NULL) == pdPASS){
    if (xTaskCreate(DetectTask, "task3", stackSize, NULL, tskIDLE_PRIORITY, NULL) == pdPASS){
       if (xTaskCreate(LCDScreenTask, "task4", stackSize, NULL, tskIDLE_PRIORITY, NULL) == pdPASS){

         vTaskStartScheduler();
          }
       }
    }
}
```

Code 18: Task configuration main procedure.

❖ Additional Procedure

Regardless of the previous procedure, we will configure the **LCDWithAccelerometer** task, as seen in code 19. This task will invoke the functions of the accelerometer and gyroscope to obtain the values of acceleration (Ax, Ay, Az) and angular velocity (Gx, Gy, Gz) in all three axes. Subsequently, these values will be printed on the LCD screen in groups of 2 every 1 second, and the LCD screen will be cleared after each print. The grouping of 2 values is necessary because our LCD screen has only two rows, and to display all values from the MPU6050 sensor, we need 6 rows. This task will be invoked as shown in code 20, and since it is a single task, no semaphores will be needed. This additional program will be used solely to verify the proper functioning of the accelerometer. In our main program, code 20 will be commented out, and when we want to execute this additional program, we will comment out code 18.

```c
void LCDwithAcelerometer(void *argument)
{

  while(1){

        MPU6050_Read_Accel();
        MPU6050_Read_Gyro();

          lcd_send_cmd(0x01);

          HAL_Delay(1000);

          lcd_send_cmd(0x80 | 0x00);
          lcd_send_string("Ax=");
          sprintf(but, "%.2f", Ax);
          lcd_send_string(but);
          lcd_send_string("g");

          lcd_send_cmd(0x80 | 0x40);
          lcd_send_string("Ay=");
          sprintf(but, "%.2f", Ay);
          lcd_send_string(but);
          lcd_send_string("g");

          HAL_Delay(1000);

          lcd_send_cmd(0x01);

          HAL_Delay(250);

          lcd_send_cmd(0x80 | 0x00);
          lcd_send_string("Az=");
          sprintf(but, "%.2f", Az);
          lcd_send_string(but);
          lcd_send_string("g");

          lcd_send_cmd(0x80 | 0x40);
          lcd_send_string("Gx=");
          sprintf(but, "%.2f", Gx);
          lcd_send_string(but);

          HAL_Delay(1000);

          lcd_send_cmd(0x01);

          HAL_Delay(250);

          lcd_send_cmd(0x80 | 0x00);
          lcd_send_string("Gy=");
          sprintf(but, "%.2f", Gy);
          lcd_send_string(but);

          lcd_send_cmd(0x80 | 0x40);
          lcd_send_string("Gz=");
          sprintf(but, "%.2f", Gz);
          lcd_send_string(but);

          HAL_Delay(1000);
       }

      }
```

Code 19: LCD with accelerometer Task.

```c
if (xTaskCreate(LCDwithAcelerometer, "task1", stackSize, NULL, tskIDLE_PRIORITY , NULL) == pdPASS){
    vTaskStartScheduler();
  }
```

Code 20: Task additional procedure.

7. EXPERIMENTAL TESTS

In this section, we will explain the experimental tests conducted, showcasing images with the obtained results to demonstrate the correct functioning of the program.

Initially, we created a circuit using black insulating tape, as shown in Figures 35 and 36. In this circuit, the car will follow the path of the tape indefinitely, guided by the infrared sensors controlling its movement. Simultaneously, distances to objects detected by the ultrasonic sensor will be printed on the LCD screen at 1-second intervals. Throughout this journey, the LED remains off, indicating that no objects are detected within less than 10 cm.

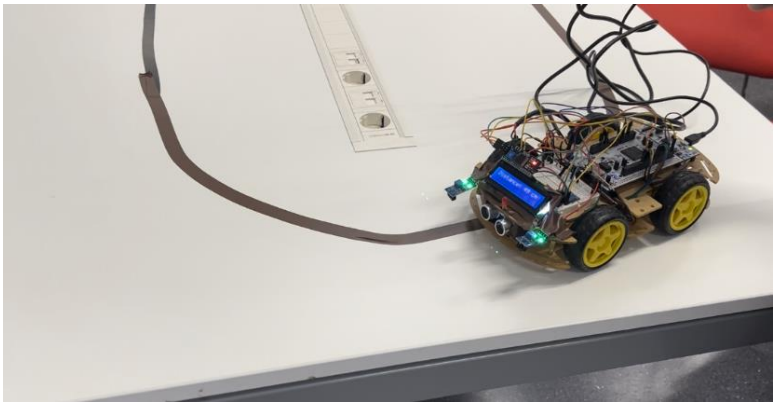

Figure 35: Curved trajectory followed by the car during the circuit tracking.

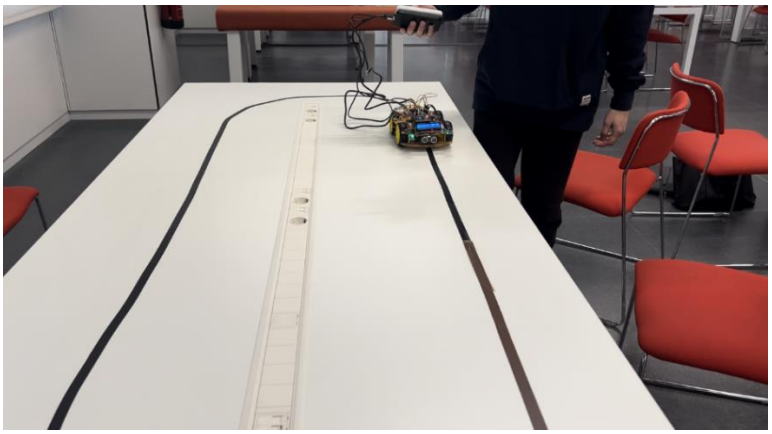

Figure 36: Straight trajectory followed by the car during the circuit tracking.

In Figure 37, we can observe how, by placing an object, specifically the hand, at a distance less than 10 cm, the car will stop, and the red LED will turn on. Once we remove the hand, the car will resume following the circuit, controlled by the readings from the infrared sensors.

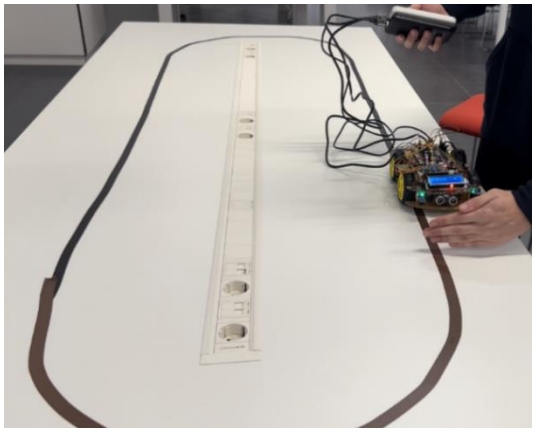

Figure 37: Car stopping due to the detection of an object at a distance less than 10 cm.

Finally, in Figures 38 and 39, we will observe the display on the screen of the distance to the object and two of the six accelerometer values, respectively. In the case of the distance display, since this action occurs at intervals of 1 second, it is necessary to wait approximately 2 seconds for the display on the screen to show the correct distance and not the capture from the previous moment. This can be observed in Figure 38, where the distance printed on the LCD screen is indeed the distance to the object. Additionally, since this distance is less than 10 cm, the red LED is turned on.

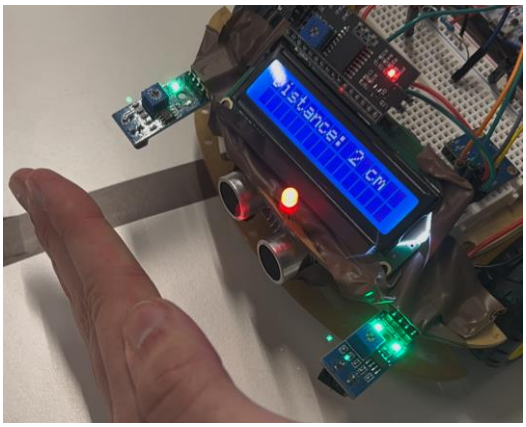

Figure 38: Printing the distance between the sensor and the object on the LCD screen.

Similarly, in Figure 39, we will observe the values captured by the accelerometer, specifically the Gy and Gz values, corresponding to the angular velocities of the Y and Z axes, respectively. As mentioned in previous sections, accelerometer values will be printed in pairs, and each print will occur at intervals of 1 second. Therefore, this test was conducted simply to verify the functionality of the accelerometer. It was performed with the car stationary, and after a complete iteration, we changed its position to observe how the values of acceleration and velocity change. This test was conducted in this way because, given the speed at which the car moves, we would never observe the actual values captured by the accelerometer. Additionally, if we were to move the car before completing the iteration, we would continue to observe values from the previous position until the iteration is complete, instead of observing the values from the current position. This is why we decided to exclude it from the main program since it wouldn't make sense to observe values that are not accurate during the circuit journey, and we opted for displaying the distance, which did show a much more precise and real value.

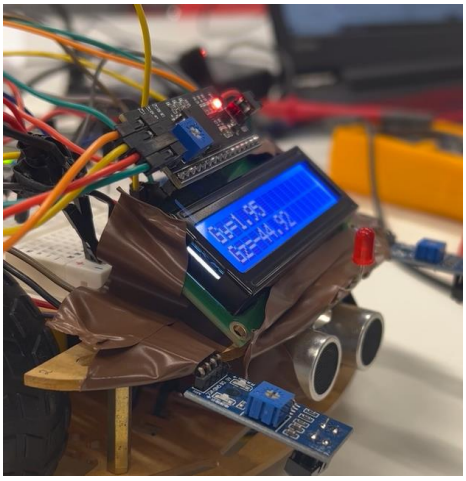

Figure 39: Printing the values of the MPU6050 accelerometer on the LCD screen.

8. CONCLUSIONS

We conclude this project by highlighting the skills and knowledge gained during its development. We experienced a significant advancement in our C++ programming skills and gained a deep understanding of fundamental tools such as STM32CubeIDE and Ozone. Exploring the diversity of sensors and their applications provided valuable insights, along with the ability to effectively integrate hardware and software in real-time embedded systems environments.

This project not only solidified the theoretical concepts learned in the course but also challenged us to apply them in a tangible project. Task management, timers, and semaphores, combined with hardware resource optimization, were essential for the project's success.

Regarding the results, we observed the effectiveness of both the main program, focused on the vehicle's circuit tracking and object detection with distance printing, and the additional program designed to verify the accelerometer's functionality. Although we consider the results satisfactory, we identified areas for improvement, such as the potential functional integration of the accelerometer into the main program.

Addressing the challenges, the need to use three batteries for the proper functioning of the vehicle was a prominent challenge. This was the main reason for not including the accelerometer in the main program, as the need to power an additional sensor jeopardized the operation of the rest of the sensors and, consequently, the proper functioning of the program. Another challenge was the complex integration between software and hardware. Additionally, a detailed understanding of sensor operation and task coordination through semaphores posed fundamental challenges in programming.

It is also important to note that our project underwent significant improvements after the initial presentation. We implemented two infrared sensors, completed the configuration of the accelerometer, and added a red LED to indicate when the object is within less than 10 cm. Additionally, we designed a support structure in SolidWorks to facilitate the movement of the car, housing the three required batteries.

In summary, we feel proud and satisfied with the achievements in this project, which not only expanded our technical skills but also exposed us to valuable challenges in the field of embedded systems.