Embedded Systems

Project Work – A.Y. 2020/21

Team 2 – Self Balancing Robot (SBR)

Alessio Pepe	0622701463	a.pepe108@studenti.unisa.it
Paolo Mansi	0622701542	p.mansi5@studenti.unisa.it
Teresa Tortorella	0622701507	t.tortorella3@studenti.unisa.it

1. Solution

A working Self Balancing Robot (SBR) has been created, the physical implementation of which is shown in Figure 1.

Several components were used, including

- **STM32f401:** The main microcontroller based on ARM technology with which almost all operations are carried out.
- **ESP32 DOIT DevKitC v1:** The microcontroller with built-in Bluetooth, used to control the movements sent to the robot.
- **MPU6050:** A 6-axis gyroscope and accelerometer sensor (3 and 3) used to calculate the inclination of the SBR.
- **DC Motor with reduction:** Motors used to push the wheels as calculated by the software.
- **H-Bridge L298N with Motor Driver:** Used to control the speed and direction of DC motors.
- **Led:** Used to report any errors.
- **Buzzer:** Used to implement a horn.
- **Aluminum frame:** Structure represented in Figure 1 to which all the components are mounted.

To view the operation of the SBR, refer to the demonstration video at the link https://youtu.be/qOr8XTu--TQ.

All code, schematics, images, and a copy of this document and any other resources, are provided to the repository https://github.com/pepealessio/SBL.

2. Design Choices

This section will explain all the design choices made in the realization of the project, referring to Figure 3 that shows a simplified diagram representing the implementation.

First of all, the MPU6050 sensor was used to read the angle of inclination of the motors concerning the ground. The MPU6050 sensor was positioned as close as possible to the axis on which the wheel motors lie to get the best possible read angle. The reading angle is represented as the angle in Figure 2, which explain the preceding sentence. For the reading of the angle, based on the current bibliography, we used a Kalman filter [1] to combine the gyroscope readings and the accelerometer, obtaining a reading that is not noisy and not affected by the movements of the SBR. We considered using more simple filters, such as the complementary filter [2], or even an implementation without filters, but these were respectively too slow and too noisy for proper stability.

For the SBR to have more room to stabilize itself, and thus for gravity to have a more negligible effect on the speed of the fall, it was decided to move the centre of gravity as low as possible. Therefore, all components were placed on the lowest shelf of the structure and additional weights were also placed in the lower part of it, adding steel washers.

A PID regulator, closed on the angle thus read, was then implemented to regulate the power to be given as input to the motors to stabilize the structure. We have chosen to use a PID as this is relatively simple to code and does not require large amounts of computing power to function correctly. We carried out repeated tests to calibrate the PID parameters until we found the most suitable values for our case.

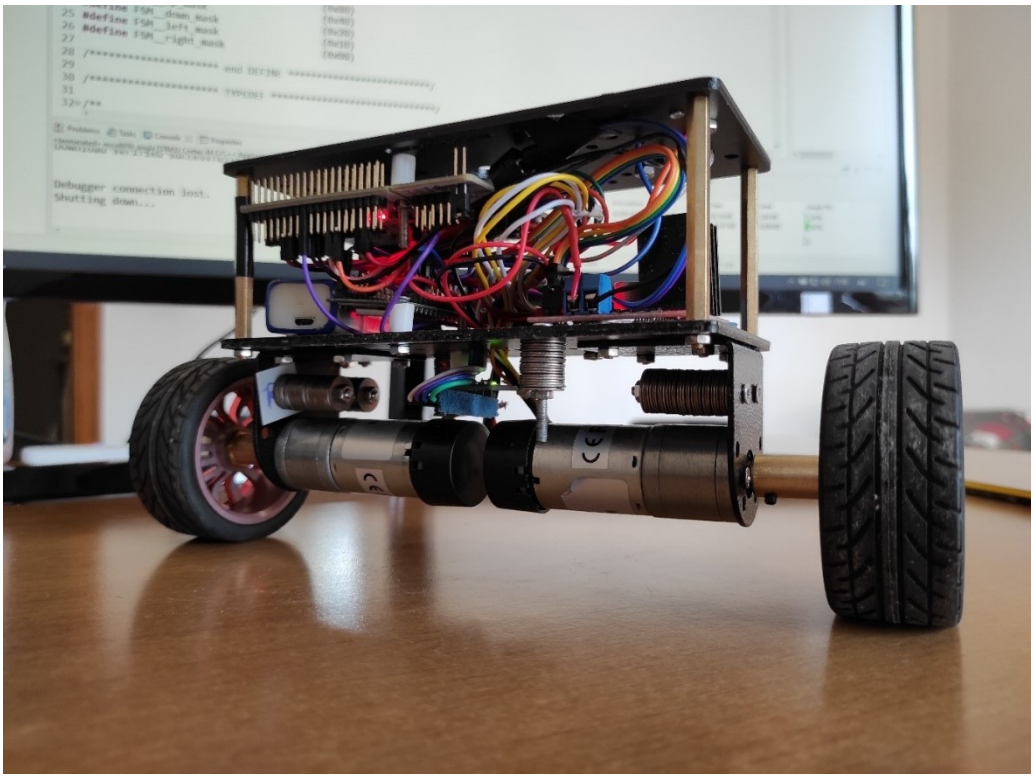

Figure 1. Picture representing the final Self Balancing Robot realized view from the bottom.

The motors receive, via the H bridge, a power that the PID produces to maintain the structure in equilibrium (in terms of duty cycle).

A supervisor was also developed to manage the movement of the SBR. Depending on the movement to perform, the supervisor changes the angle at which the PID tends to balance to trigger movement in that direction. For example, to move forward, approximately one degree is added to the balance target. This offset is removed when the SBR no longer needs to move. Furthermore, the supervisor changes the power assigned to the two motors, coordinating them to perform the rotations.

The supervisor receives the motion commands via a UART communication with the ESP32, which receives them via a Bluetooth connection with a python application on the user's computer. This communication will be analyzed in detail in Chapter 5.

To simplify the calibration operations of the SBR, we decided to place an easy-to-reach button for resetting both the STM and the ESP32 on the top of the structure. An error LED has also been placed at the same location, which lights up with a different frequency depending on the error that occurred. In particular, it flashes at a frequency of 1 Hz in case of a robot fall and flashes at a frequency of 2 Hz in case of an error in the MPU6050 reading. Finally, a horn was implemented, realized via a passive buzzer and controlled via the python application as for movement.

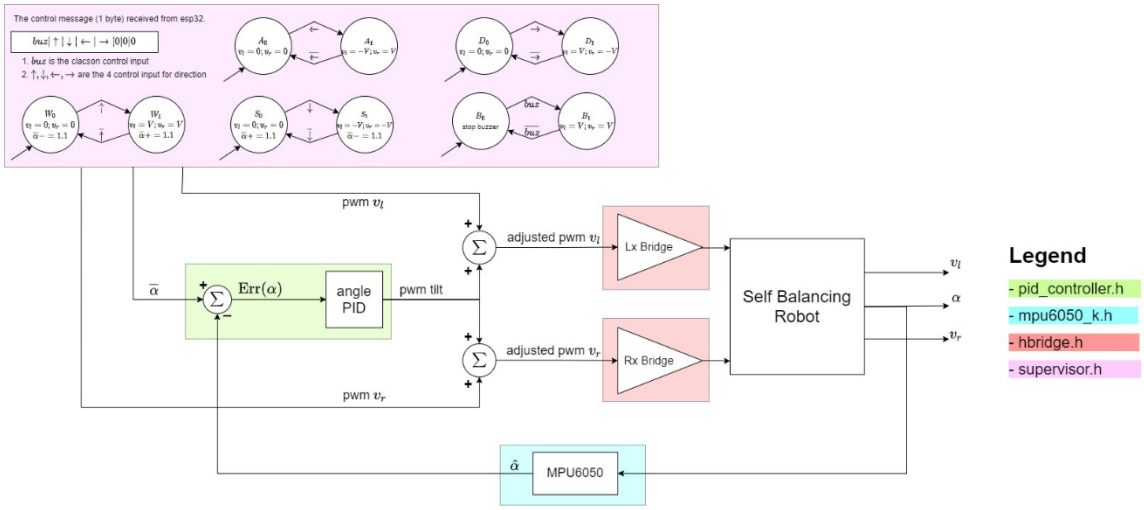

Figure 3. Communication protocol, automaton, and full system representation

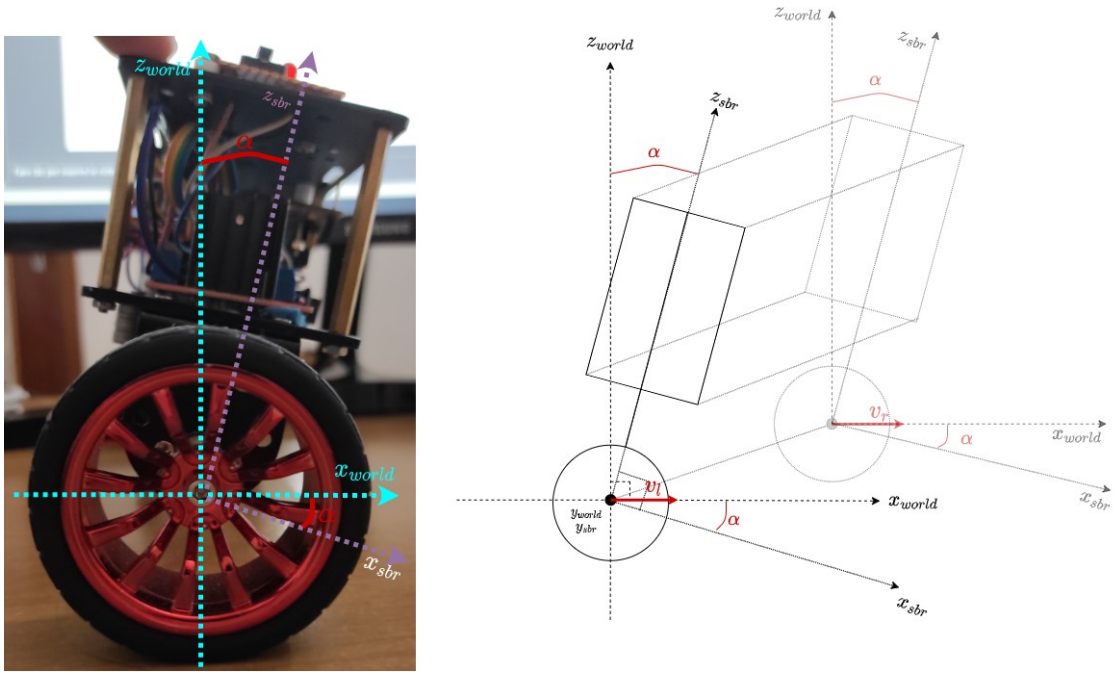

Figure 2. Schematic of the SBR from a physical point of view.

3. Hardware Architecture

The following table shows the main hardware connections, which are also schematized in Figure 4**Errore. L'origine riferimento non è stata trovata.**.

STM32		
Component	**Pin**	**Protocol**
HBridge	PA8	Timer 1 Channel 1
	PA9	Timer 1 Channel 2
	PA10	Timer 1 Channel 3
	PA11	Timer 1 Channel 4
MPU6050	PB8	I2C1 CLK
	PB9	I2C1 SDA
ESP32	PBT	UART2 Rx
Power Supply	E5V, GND	Alimentation
Error Led	PA5	GPIO Output Pin
Reset Button	NRST	Reset Line

H Bridge		
Component	**Pin**	**Protocol**
Power Supply	12V, GND	Alimentation
STM32	IN1	Timer 1 Channel 1
	IN2	Timer 1 Channel 2
	IN3	Timer 1 Channel 3
	IN4	Timer 1 Channel 4
	5V	Alimentation Output
Left Motor	M1_01, M1_02	Alimentation motor 1
Right Motor	M2_01, M2_02	Alimentation motor 2

ESP32		
Component	**Pin**	**Protocol**
STM32	TX0	UART0 Tx
Power Supply	VCC, GND	Alimentation
Buzzer	D23	GPIO Analog Output
	GND	Ground

Power is supplied by a 9-volt battery. The ground is common to all devices. The 9V power supply is supplied directly to the h-bridge and to the esp32 (which requires a supply voltage between 7 and 12 volts). We use the motor driver to transform the voltage from 9V to 5V and power the devices that all require 5V (except for the MPU6050 which requires a voltage of 3.3V, taken from the board).

The STM32 is connected to the ESP32 through the single wire UART port.

The MPU6050 is connected to the i2c1 port on STM32.

The H-Bridge is connected to the 4 channels of the tim1, settled in PWM mode.

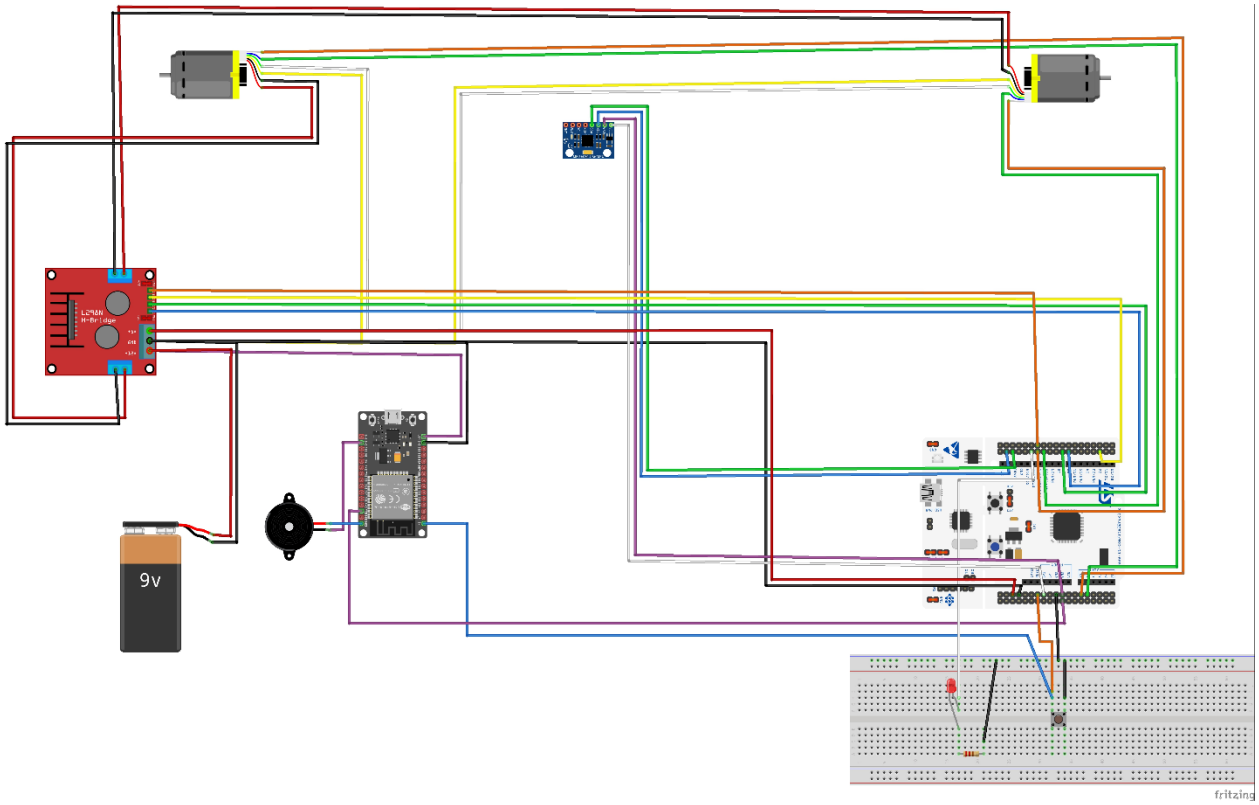

Figure 4. Connection's scheme realized via Fritzing application.

The circuit made on the breadboard is soldered on a test base, mounted on the structure's roof.

4. Software Architecture

The product code is divided into three main folders, one containing the code for the STM32, one with the code for the ESP32, and the last containing the Python code for the control interface.

4.1. STM32

Figure 3 shows the schematic diagram of the implemented system, also highlighting the division into modules.

4.1.1. PID

In the files `pid.h` and `pid.c` the closed PID controller has been defined in angle feedback. The following methods are provided for the implementation of the PID

```
void pid__init(pid_controller_t *hpid, int32_t kp, int32_t ki, int32_t kd, int32_t
target);

void pid__set_target(pid_controller_t *hpid, int32_t target);

int32_t pid__get_target(pid_controller_t *hpid);

int64_t pid__evaluate_output(pid_controller_t *hpid, int64_t read_value);
```

which, respectively, initialize the PID with the desired parameters, set a new target for the PID, read the current target, and calculate the PID output based on the current reading.

4.1.2. MPU

The `mpu6050_k.h` and `mpu6050_k.c` files contain the driver for reading the values from the accelerometer and gyroscope sensors on the MPU6050, as well as compute the angle using the Kalman filter. The following methods are provided for the implementation of the MPU6050 driver

```
uint8_t MPU6050__Init(MPU6050_t *hmpu6050, I2C_HandleTypeDef *I2Cx);

uint8_t MPU6050__Read_All(MPU6050_t *hmpu6050);
```

which, respectively, initialize the sensor with the used i2c port and read all parameter measured by the sensor (accelerometer, temperature, and gyroscope) and compute the Kalman Angle.

4.1.3. Supervisor

The `supervisor.h` and `supervisor.c` files contain the supervisor that receives commands from the Uart and changes the references to the PID. The following methods are provided for the implementation of the supervisor

```
void FSM__init(FSM__current_t *hfsm);

void FSM__set_control_message(FSM__current_t *hfsm, FSM__control_message new_message);

void FSM__update(FSM__current_t *hfsm);
```

which, respectively, initialize the supervisor, set a new control message, and compute the new resulting state (and the new exits).

4.1.4. H-Bridge

The `hbridge.h` and `hbridge.c` files contain the implementation of the control of the motors connected to the hbridge. The following methods are provided for the implementation of the hbridge driver

```
void hbridge__init(hbridge_t *hhbridge, TIM_HandleTypeDef *htim);

void hbridge__set_motor(hbridge_t *hhbridge, int64_t powerl, int64_t powerr);
```

which, respectively, initialize the hbridge and set the desired engine power.

4.1.5. Led

In the `led.h` and `led.c` files the class of a generic led is defined, with the methods for switching the led on and off, respectively

```
void Led__init(Led_t *self, GPIO_TypeDef *GPIOx, uint16_t GPIO_PIN);

void Led__turn_on(Led_t *self);

void Led__turn_off(Led_t *self);

void Led__toggle(Led_t *self);
```

4.1.6. Main loop

Finally, the `main.c` file contains the general execution of the program, which basically implements a scheme for reading inputs, processing outputs, and writing outputs.

- When the program starts, all components, the PID controller and the supervisor are initialized. If the MPU reading shows an error, the motors are switched off and the LED starts flashing at a frequency of 2 Hz, a state in which it remains until the device is restarted. The UART2 port then waits for data, in interrupt mode.
- After that, the values are cyclically read from the MPU, and then the angle is calculated; the powers to be supplied to the motors are evaluated and finally these powers are passed to the motors for balancing.

All of that is implemented in a loop architecture with a fixed interval time.

4.2. ESP32

The code loaded on the ESP32 was created using the Arduino IDE. This has an initialization phase for the UART port and Bluetooth and a main that receives commands via Bluetooth and then transmits them back via UART. There is also a control for switching the buzzer on and off. The communication protocol will be described in the Chapter 5.

4.3. GUI Python

The code for the graphical user interface was created in Python. For Bluetooth communication, the `PyBluez2` library was used. The interface has four buttons for movement, one for the horn and one for repeating a connection attempt to the ESP32. When the program is started, a scan of the Bluetooth devices in the vicinity is carried out, and a connection is attempted to the "SBLGroup2" device, if present. After the connection has been made, pressing the buttons on the screen or the relative buttons on the keyboard sends the data as described in the protocol in the Chapter 5.

5. Protocols

5.1. UART

For communication between ESP32 and STM32 the UART protocol has been used in single wire mode, where ESP32 transmits the data while STM32 transmits the data. A Baud Rate of 115200 has been set on both devices and reception is handled by interrupts on STM32.

5.2. I2C

The I2C protocol has been used for communication with the MPU6050, in fast mode to ensure faster reading of angle values.

5.3. Bluetooth

A Bluetooth socket has been used for communication between the user's computer and the ESP32. Data are sent according to the protocol described below.

5.4. Moving Command Protocol

About the control of the SBR, it was decided to use a byte defined as follows:

$$buz \mid \uparrow \mid \downarrow \mid \leftarrow \mid \rightarrow \mid 0 \mid 0 \mid 0$$

The bit of the buzzer is the bit of the forward direction, is the bit of the backward direction, is the bit of the left direction and is the bit of the right direction. The last three bits are left unused.

When buttons on the GUI are pressed, their bits are set to 1 until the buttons are released. The byte thus created is sent to the ESP32 via Bluetooth, which in turn sends it back to the STM32 via UART. The STM finally places this received message as a supervisor control message.

If no new command is received for 0.3 seconds, the same command is repeated to avoid junk value readings. If nothing is being received from the Bluetooth socket for a while, then a byte where all bits are set to 0 is being sent to the STM32, thus in order to avoid noisy transmissions.

If the buzzer button is pressed, then the ESP32 will sound the buzzer at a frequency of (B5 note), regardless of the STM operation.

A representation of this protocol is schematized in Figure 5.

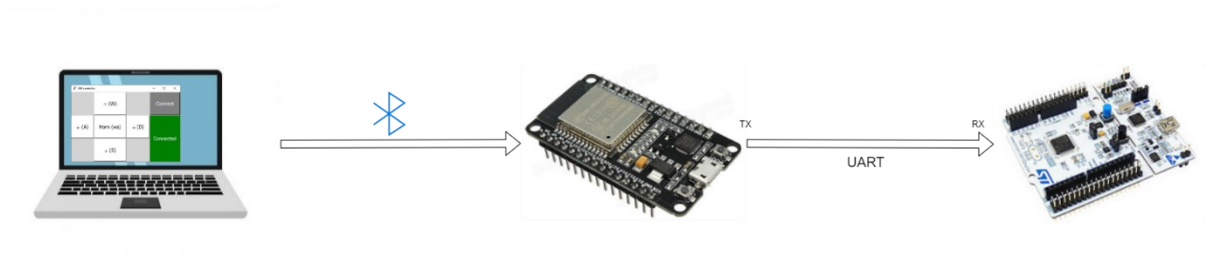

Figure 5. A scheme representing our protocol.

6. References

[1] M. N. M. A. Neto P., «Kalman Filter-Based Yaw Angle Estimation by Fusing Inertial and Magnetic Sensing,» in *Proceedings of the 11th Portuguese Conference on Automatic Control*, 2015.

[2] S. Colton, «The Balance Filter,» *Chief Delphi,* 2007.