UNIVERSITY OF PADOVA

Embedded Real–Time Control

Laboratory report

Bettin Paolo - 2089152
Vitetta Emanuele - 2086147
Merolli Martina - 2072012
Guglielmin Giorgia - 2088623
Bonaventura Luca - 2090005

Academic Year 2022-2023

Contents

Abstract

This report presents the results obtained from five laboratory experiments. For each experiment, it provides an overview of the theoretical background and the resolution of all exercises. To maintain readability, only essential code sections have been included. The remaining material, including various projects and their respective IDEs, can be found in the following Google Drive folder: `https://drive.google.com/drive/folders/1xTdD9A9gXoKNwmuc1KUiapZGkdVKTiEy?usp=sharing`. Each laboratory has its own folder within the drive, containing code files, as well as photos and/or videos demonstrating the code's functionality. Please note that the code may not be extensively cleaned, as it often encompasses multiple exercises within a single executable.

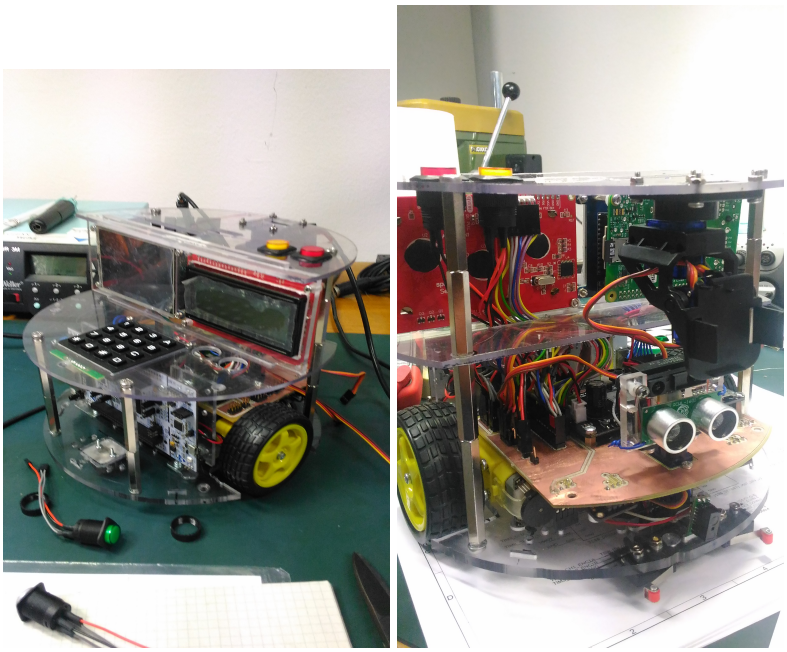
Chapter 1

Laboratory 1

1.1 Introduction

This laboratory aims to interact with external devices connected to the microcontroller via a serial bus. The Tbot, shown in *Figure 1.1*, provides some peripherals such as a keypad and line sensor. To meet the requirements, two Interrupt Service Routines (ISRs) need to be written. The first ISR is responsible for reading the line sensor, while the second ISR is designed to identify which button on the keyboard is pressed. Once the two devices are correctly connected, a new challenge is accomplished: one LED of the TBot should blink at a frequency specified by the user via the keyboard. Two implementations are provided:

- In the first implementation, the user can input single-digit frequencies.
- In the second implementation, the user can select frequencies with multiple digits and confirm them by pressing the "#" key.



(a) The TurtleBot 1 (b) The TurtleBot 2

Figure 1.1: The TurtleBot

1.2 Relevant theoretical notions

1.2.1 Serial Trasmissions

Serial transmissions allow data to be transmitted one bit at a time in sequence over a single transmission line. In this type of communication, data is transmitted asynchronously or synchronously, depending on the specifics of the interface used. In the used system data is always transmitted asynchronously, although also synchronous transmission is supported. In synchronous serial transmission, however, the synchronization between transmitter and receiver takes place using an external clock signal.
The hardware component for synchronous peripherals is called USART (*Universal Synchronous-Asynchronous /Receiver /Transmitter*), while for asynchronous peripherals is called UART.

1.2.2 I2C

As for serial communication, the TBot utilizes the I2C (Inter-integrated Circuit) communication protocol to interact with various peripherals.

The I2C protocol is characterized by a bus architecture, enabling multiple devices to be connected on a single communication line. It operates on a master-slave architecture, where the microcontroller typically acts as the master, and the external peripherals serve as the slaves. The I2C protocol employs two bidirectional lines: a clock line (SCL) and a data line (SDA), which facilitate the transmission of data between devices. Each device on the bus is assigned a unique address, allowing for selective communication between devices.

A crucial operation in I2C communication is the reading of registers. These registers contain essential information about the device's status and the ongoing communication operation. To read the registers of an I2C device, a sequence of I2C commands must be sent to set the address of the desired register.

The sequence for reading a register begins with the microcontroller sending a start signal and concludes with a stop signal. The start and stop conditions are identified by the data line transitioning to a low state (start) or high state (stop) during the clock's high phase. All other bits are transmitted during the low clock phase, ensuring that the start and stop conditions can be distinguished from other data transmissions.

Typically, the data rate for I2C data transmission is 100 kbps in standard mode and 400 kbps in fast mode.

1.2.3 GPIO

A "General Purpose Input/Output" (GPIO) refers to the pins on a microcontroller or microprocessor that can be configured as digital inputs or outputs and that are controllable by software. The `HAL_GPIO` command family simplifies the usage of GPIO pins by offering functions and methods to configure pins as input or output, set the logic levels of the pins, read input pin values, and write output pin values. They can work also in Polling mode: in this case the HAL functions return the process status when the data processing in blocking mode is complete (the operation is considered complete when the function returns the `HAL\_OK` status, otherwise an error status is returned).
In this laboratory it is implemented the function `void HAL_GPIO_EXTI_Callback(uint16_t pin)`. It is a callback function defined in the *Hardware Abstraction Layer* (HAL) for handling external interrupts on GPIO. When an external interrupt occurs on a GPIO pin that is configured as an input with the corresponding interrupt enabled, the `HAL_GPIO_EXTI_Callback()` function is automatically called by the system. This function can be customized by the user to handle specific actions or behaviors in response to the external interrupt event. Other HAL APIs are the following:

- `void HAL_GPIO_Init (GPIO_TypeDef * GPIOx, GPIO_InitTypeDef * GPIO_Init)` → initializes the GPIOx peripheral according to the specified parameters in the GPIO_Init;
- `void HAL_GPIO_DeInit (GPIO_TypeDef * GPIOx, uint32_t GPIO_Pin)` → de-initializes the GPIOx peripheral registers to their default reset values;
- `GPIO_PinState HAL_GPIO_ReadPin (GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)` → reads the specified input port pin;
- `void HAL_GPIO_WritePin (GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin, GPIO_PinState PinState)` →sets or clears the selected data port bit;
- `void HAL_GPIO_TogglePin (GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)` →toggles the specified GPIO pins;
- `void HAL_Delay (uint32_t Delay)` → provides minimum delay (in milliseconds) based on variable incremented.

1.2.4 SX1509

The main device considered in this laboratory is the SX1509. Two SX1509 are connected to the microcontroller through I2C bus and both devices are shares the same I2C line, in this case I2C1. The first SX1509, which in the code is called SX1509_I2C_ADDR1, is used to interact with a line sensor (Pololu QTR Reflectance Sensor), while the second one, SX1509_I2C_ ADDR2, is the keypad engine.

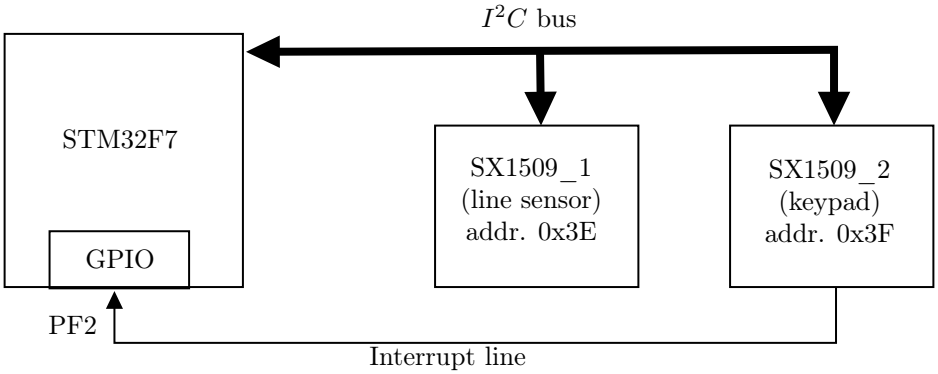


Figure 1.2: Scheme of the SX1509 connections

1.2.5 FreeRTOS

FreeRTOS is a class of Real Time Operative Systems that is designed to be small enough to run on a microcontroller or microprocessor. This system is free but doesn't guarantee warranty and technical support for OS, features implemented the OpenRTOS. The system components will be now explained.

Tasks

In general a task is a unit of execution, in FreeRTOS task has the following property:

- One task executes at a time;
- Tasks have no knowledge of scheduler activity;
- Tasks have their own stack upon which execution context can be saved;
- Tasks can be prioritized.

Tasks has four states:

- Running: actively executing and using the processor;
- Ready: able to execute but not running because another task of equal or higher priority is in the running state;
- Blocked: the task is waiting for an external event and can't continue his execution until the event occurs;
- Suspended: can enter in this state only via API call, with vTaskSuspend and vTaskResume (see FreeRTOS API). Observe that when a task is suspended it doesn't take any microcontroller processing time, while if it is blocked it wastes some.

The transitions scheme is the following:

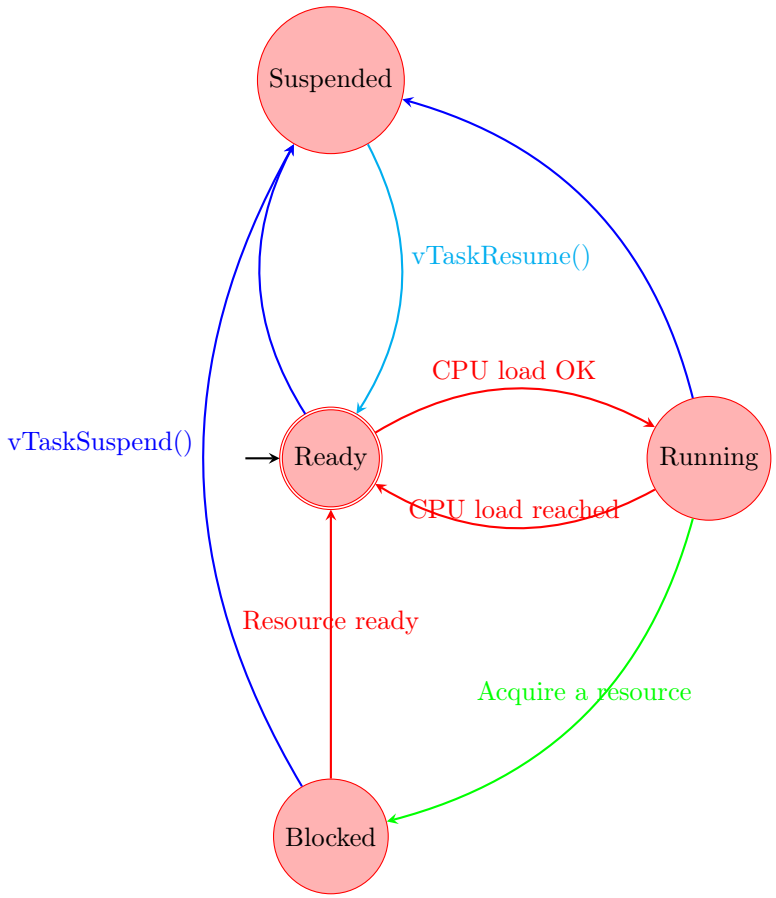


Figure 1.3: Scheme of the transition status of a semaphore

Semaphores

While writing the code for this lab, an essential feature that may be required is to synchronize the execution of processes. In particular, in the extra part of the lab, there is a need to change the state of Led 2 every time the line sensor performs a read. To achieve this synchronization, FreeRTOS binary semaphores are utilized. Binary semaphores are mutex variables that serve two crucial functions:

- `osSemaphoreAcquire()` → We can proceed with the execution of the code only if the semaphore is released;
- `osSemaphoreRelease()` → Release the resource and allow a task stuck in the `osSemaphoreAcquire()` to proceed with the execution;

With that important brick we can solve the problem, see the implementation on 1.4.6.

1.3 Problem analysis

1.3.1 How to read bits inside registers

The Keypad Layout is shown in Figure 1.4. Initially, the bits were read as they are from the register. However, to obtain the correct information from the keypad register, a specific interpretation process is required. When accessing the keypad register, two bytes are received: one for the row status and another for the column status. In these bytes, the most significant bits are always set to 1, while the least significant bits indicate the position of the pressed button, with a corresponding zero value. To properly translate this information, the following decoding sequence is applied:

- The bitwise complement of the register is considered.
- One is subtracted from the complemented value.
- Finally, the number of bits with a value of 1 is determined using the function `uint8_t countBit(uint8_t n)`.

This decoding sequence ensures that the correct translation of the pressed button position is obtained from the register data.

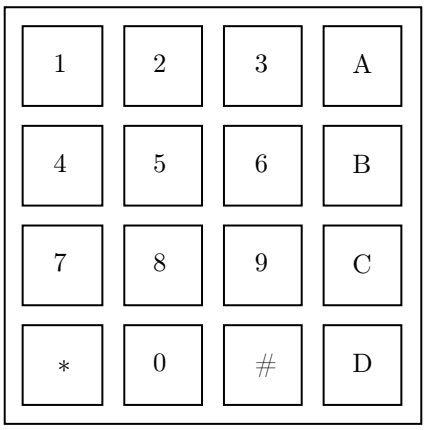


Figure 1.4: Keyboard Layout

1.3.2 From char value to decimal value

In the C programming language, characters are represented using the ASCII (American Standard Code for Information Interchange) character set. Within ASCII, each character is assigned a unique decimal value. For instance, the character '0' corresponds to a decimal value of 48, '1' corresponds to 49, and so on, up to '9' which corresponds to 57.

To convert a single character that represents a digit to its corresponding integer value, a simple method can be employed. By subtracting the decimal value of '0' (48) from the decimal value of the character itself, the desired conversion can be achieved. This approach is effective because the decimal value of the character '0' is always lower than that of any digit character. For example, when working with the character '5', which represents the digit 5, the conversion can be accomplished by subtracting the decimal value of '0' (48) from the decimal value of '5' (53), resulting in the integer value of 5. It's important to note that this conversion method specifically applies to single-digit characters representing numbers ranging from 0 to 9.

1.3.3 Printf

In order to read data from the TurtleBot, we exploited the `printf()` function. Such a function only works in debug mode and in order to receive data properly, some parameters have to be set in the STM32CubeIDE. We did this by following the given instructions. `printf()` is really useful during debugging, because it allows to display the intermediate results.

1.3.4 Proximity sensor issues

In the original source project, the interrupt of the proximity sensor was enabled. Anyway, when an interrupt is called by one of these sensors, the program execution crashes. This is due to the fact that the system is not able to handle such interrupt. To solve this problem we modified the related pin on the **.ioc* file.

1.4 Code structure

The listing below summarizes exercises 1, 2 and 4. Code is reported in only one listing for a reason of space. This part is absent in the `main.c` file of exercise 3, whose necessary code is reported under the relative section.

```
#define INITIAL_WAIT_TIME  1000;
int actual_wait = INITIAL_WAIT_TIME;
int actual_keyboard_value = 0;
int frequency = 0;

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  printf("Interrupt on pin (%d).\n", GPIO_Pin);

  uint8_t buf[2], data;

  HAL_StatusTypeDef status =  HAL_I2C_Mem_Read(&hi2c1, SX1509_I2C_ADDR2 << 1, REG_KEY_DATA_1,
    1, &buf[0], 1, HAL_TIMEOUT);
  if (status != HAL_OK)
    printf("I2C communication error (%X).\n", status);
  status =  HAL_I2C_Mem_Read(&hi2c1, SX1509_I2C_ADDR2 << 1, REG_KEY_DATA_2, 1, &buf[1], 1,
    HAL_TIMEOUT);
  if (status != HAL_OK)
    printf("I2C communication error (%X).\n", status);
  int row = countBit(~buf[0]-1);
  int column = countBit(~buf[1]-1);
  printf("Keypad button: (%c).\n",keypadLayout[column][row]);

  // EASY WAY
  //int frequency = (int) keypadLayout[column][row]-48;

  // HARD WAY
  if(keypadLayout[column][row] == '#')
  {
    frequency = actual_keyboard_value;
    actual_keyboard_value = 0;
  }
  else
  {
    actual_keyboard_value = actual_keyboard_value*10 + ((int) keypadLayout[column][row]-48);
  }

  if (frequency == 0)
    actual_wait = 0;
  else
    actual_wait = 1000 / frequency;
}
```

Listing 1.1: Exercises 1 - 2 - 4

1.4.1 Exercise 1

In this exercise, it is required to recognize and handle the keypad interrupt. For this reason it is implemented the function `void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)`, which will be developed in the next points. To accomplish the task it is added a `printf()` which is useful for debugging (prints the number of the pin on which the interrupt occurs). Furthermore, in order to be able to receive a different interrupt request after a button has been pressed on the keypad, it is required to read both row and column registers. In fact, after both registers are readed they are also automatically cleaned.

When performing register reading, we define `HAL_StatusTypeDef status`, a variable in which we save the values stored in the registers. When reading each register, we check if `status!= HAL_OK`: if such a condition holds true, it means that there is an error in the I2C communication. The values of the registers are made of a byte each, and they are saved in the variable `uint8_t buf[2]`. Notice that only the four less significant bits of each register are relevant, anyway we can read the whole byte.

1.4.2 Exercise 2

In order to correctly translate the values of the registers in two coordinates useful to decode the pressed button, using the given variable `keypadLayout`, we have implemented the auxiliary function `uint8_t countBit(uint8_t n)` (reported in the listing below). The number of the row/column activated on the keypad is given by the number of active bits on the bit-wise complement of the value of the relative register. See 1.3.1 for the explanation.

```
uint8_t countBit(uint8_t n){ // return the number of ones in the n variable
  uint8_t count = 0;
  while(n){
    count += n & 1;
    n>>=1;
  }
  return count;
}
```

1.4.3 Exercise 3

This exercise is different from the ones above. In fact, the reading of the line sensor has to occur an infinite number of times. For this reason, we implement such execution in the `while()` of the `void main()`. As above, we read the value of the line sensor stored in the register and then we check if there is any I2C communication error. In the end, we print the value we get. Observe that the POLULU line sensor is made of 8 different reflectance sensors. Each of them corresponds to a bit in the register and such a bit gets value 1 when the sensor covers a dark reflective object. Furthermore, bits in the register have the same order as the sensors in the physical array. Notice that we print the binary value of the register converted in a decimal base (for example, if the register stores value 00010001 the program is going to print 17).

```
/* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */
   uint8_t buff;
   HAL_StatusTypeDef status =  HAL_I2C_Mem_Read(&hi2c1, SX1509_I2C_ADDR1 << 1, REG_DATA_B, 1,
    &buff, 1, HAL_TIMEOUT);
      if (status != HAL_OK)
        printf("I2C communication error (%X).\n", status);
   printf("(%d).\n", buff);
    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
```

Listing 1.2: Exercise 3

1.4.4 Exercise 4

To implement this exercise some global variables are used. In particular `#define INITIAL_WAIT_TIME  1000` and `int actual_wait = INITIAL_WAIT_TIME` are used to initialize the led with a blinking period of 1000 milliseconds, while `int actual_keyboard_value = 0` and `int frequency = 0` are exploited to use the keypad.

Both point 1 and 2 are implemented. The easy way exploits the ASCII conversion. In fact, as explained in the "Theoretical Notion" section, we have that the value of the frequency is given by the value of the button pressed on the keypad minus 48.

The hard way exploits the decimal representation of numbers. The number which we have typed on the keyboard is stored in the variable `int actual_keyboard_value`. At each iteration, we check if we have pressed the button `'#'`. If yes, we reset `actual_keyboard_value` to zero and we set the frequency for led blinking. Otherwise, we update `actual_keyboard_value`. Each time a button is pressed the value of `actual_wait` is updated. This final part works both for point 1 and 2 of this exercise and also handles the case of `frequency == 0`.

1.4.5 [Extra] Exercise 1

The code for this exercise is reported below. After following the procedure of the handout to setup FreeRTOS, we had to setup also the three requested routines. Observe that `void StartLed1task()` and `void StartLed1task()` have the same structure, but they just act on different LEDs with different delays and they have different priorities.

On the other hand `void StartLineSensorTask()` exploits the same structure already seen in exercise 3. The main difference is that the task is not executed in the `while()` of the `void main()`. In fact, in this case, this task is handled by a dedicated routine that can run in parallel with other tasks respecting the priorities given.

```
void StartLineSensorTask(void *argument)
{
  /* USER CODE BEGIN 5 */
  for(;;)
  {
  uint8_t buff;
```

```
  HAL_StatusTypeDef status =  HAL_I2C_Mem_Read(&hi2c1, SX1509_I2C_ADDR1 << 1, REG_DATA_B, 1, &
    buff, 1, HAL_TIMEOUT);
  if (status != HAL_OK)
      printf("I2C communication error (%X).\n", status);
  printf("(%d).\n", buff);
    osDelay(1000);
  }
  /* USER CODE END 5 */
}

void StartLed1task(void *argument)
{
  /* USER CODE BEGIN StartLed1task */
  for(;;)
  {
     HAL_GPIO_TogglePin(GPIOE, Led5_Pin);
     osDelay(50);
  }
  /* USER CODE END StartLed1task */
}

void StartLed2task(void *argument)
{
  /* USER CODE BEGIN StartLed2task */
  for(;;)
  {
     HAL_GPIO_TogglePin(GPIOE, Led6_Pin);
     osDelay(333);
  }
  /* USER CODE END StartLed2task */
}
```

Listing 1.3: Extra part - Exercise 1

1.4.6 [Extra] Exercise 2

In the code below a binary semaphore is used to synchronize the execution of two tasks: `StartLed1task()` and `StartLineSensorTask()` (see 1.2.5 for the explanation of the semaphores). The workflow is the following (see fig. 1.3): the binary semaphore is created using the `osSemaphoreNew()` function and its handle is stored in the `BinarySemSyncHandle` variable. Then the `StartLineSensorTask()` task periodically releases the binary semaphore at the beginning of each loop iteration using the `osSemaphoreRelease()` function. This allows other tasks waiting on the semaphore to proceed with their execution. The `StartLed1task()` task on the other hand acquires the binary semaphore using the `osSemaphoreAcquire()` function, which blocks the task until the semaphore becomes available. When the semaphore is released by the `StartLineSensorTask()` task, the `StartLed1task()` task is unblocked and allowed to proceed with its execution. By using a binary semaphore to synchronize the execution of these tasks, we ensure that the `StartLed1task` task runs only after the `StatLineSensorTask()` task has completed its work and released the semaphore.

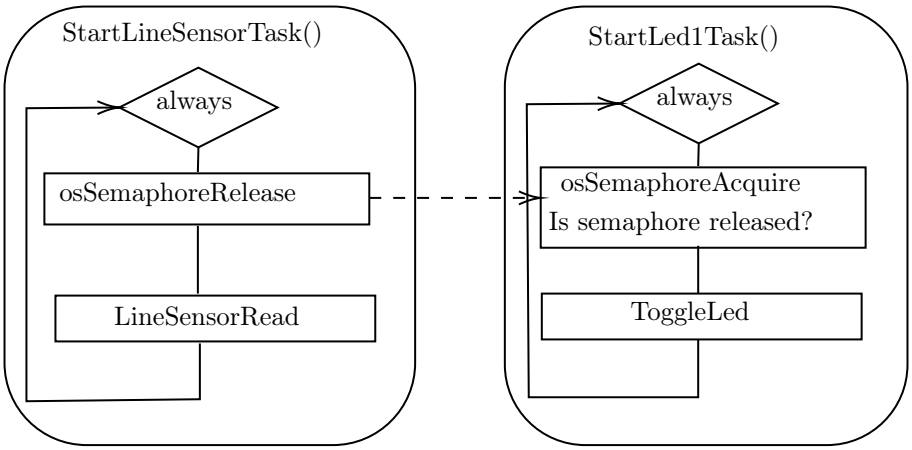


Figure 1.5: Scheme of the code functionality

```c
/* Create the semaphores(s) */
/* creation of BinarySemSync */
BinarySemSyncHandle = osSemaphoreNew(1, 0, &BinarySemSync_attributes);

void StatLineSensorTask(void *argument)
{
  /* USER CODE BEGIN 5 */
  for(;;)
  {
  osSemaphoreRelease(BinarySemSyncHandle); // Release the resource so allow the prosecution of
     the StartLed1task
  uint8_t buff;
  HAL_StatusTypeDef status =  HAL_I2C_Mem_Read(&hi2c1, SX1509_I2C_ADDR1 << 1, REG_DATA_B, 1, &
    buff, 1, HAL_TIMEOUT);
  if (status != HAL_OK)
      printf("I2C communication error (%X).\n", status);
  printf("(%d).\n", buff);
    osDelay(1000);
  }
  /* USER CODE END 5 */
}

/* USER CODE BEGIN Header_StartLed1task */

/* USER CODE END Header_StartLed1task */
void StartLed1task(void *argument)
{
  /* USER CODE BEGIN StartLed1task */
  for(;;)
  {
    osSemaphoreAcquire(BinarySemSyncHandle, osWaitForever); // Aquire the resource
   // The following code is executed only after the semaphore is released
     HAL_GPIO_TogglePin(GPIOE, Led5_Pin);
      osDelay(10);
  }
  /* USER CODE END StartLed1task */
}
```

Listing 1.4: Extra part - Exercise 2

Chapter 2

Laboratory 2

2.1 Introduction

This laboratory aims to develop a controller that stabilize the camera's tilt, in order to make it able to stare a fixed point. The system has inside a camera fixed on a stabilizing system made by two servomotors, accelerometer and gyroscope. The overall scheme of the system used is reported above:

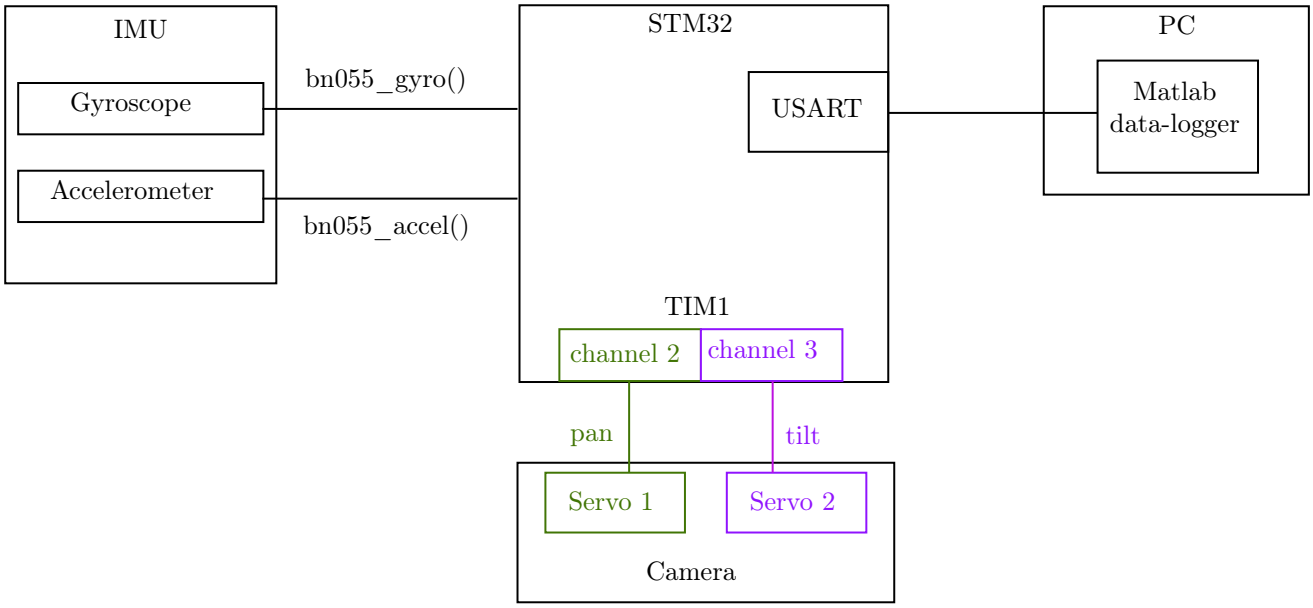


Figure 2.1: Scheme of the overall system

2.2 Relevant theoretical notions

2.2.1 Timer

A timer within a microcontroller is a hardware component that allows counting internal or external clock pulses and generating an interrupt or output signal based on the value of the counter. The timer consists of a counting register and control registers that enable configuring the operating mode.
The operating mode can be set to:

- Counting: Up or Down
- Mode: Continuous or Single
- Prescaler: to reduce the internal clock frequency

Additionally, the timer can be set to generate an interrupt or output signal when the counter reaches a specific value. To use the timer, it's necessary to

- set the control registers (during the microcontroller initialization phase) → Subsequently, the timer starts counting clock pulses based on its configuration;
- the counter reaches the specified value → an interrupt or output signal is generated and microcontroller interrupts its main activity to execute the Interrupt Service Routine (ISR) associated with the timer interrupt.

There are several types of timers available on microcontrollers, each designed to perform a specific function. One of the most common types of timer is the watchdog timer, which is used to detect hardware or software defects. This timer has a configurable period and, if the timer reaches the timeout value, sends a signal to the microcontroller (such as an interrupt) so that it can take the appropriate action, often a reset. Another important function of the timer is the real-time clock (RTC), which is used to measure the passage of time. The RTC can contain registers that keep track of date and time information. Additionally, the RTC can generate an interrupt when a certain time value is reached. Finally, pulse-width modulation (PWM) is another important timer function on microcontrollers. PWM is used to generate waves of varying amplitude, often used in applications such as motor control or LED dimming. PWM is generated by setting the timer to count up to a certain value, then resetting it and repeating the process. The duration of the PWM signal is determined by the value to which the timer is set.
For this laboratory activity TIM1 is used (see table 2.1). In fact, the servo motor is controlled using pulse width modulation (PWM): a control signal (series of pulses) is sent to the servo motor's control circuitry and the width of these pulses determines the position of the servo motor's output shaft. The timer TIM1 is capable to generate PWM.

Type	Timer N.	Resolution	Prescaler	DMA	Max interface clock (MHz)
Advanced-control	TIM1	16 bit	Range 1-65536	Yes	108

Table 2.1: Timer of the Stm32 used in this activity

2.2.2 IMU

The IMU is an essential device for any application that requires precise information on the position and orientation of an object in space. Indeed, an IMU (Inertial Measurement Unit) is an electronic device that measures and reports the specifics of a body's force, angular velocity, and orientation in space using a combination of accelerometers, gyroscopes, and magnetometers. It is commonly used in robotics, drones, and other applications that require precise positioning and orientation information.

- Accelerometers measure linear acceleration
- Gyroscopes measure angular velocity
- Magnetometers measure magnetic fields to determine the orientation of the device relative to the earth's magnetic field (not taken into account in this activity).

The combination of this information allows the IMU to determine the position, velocity, and orientation of the object in the space in which it is located. Notice that cheap sensors are always affected by a lot of noise that has to be compensated somehow.

2.2.3 Open loop control

Open loop control is a control system utilized for managing systems where the output is not adjusted based on feedback from the output itself, but rather through a predefined set of parameters or instructions. In this approach, the control action does not rely on the current state of the system, but rather on the input command provided.

In this laboratory, an open-loop control system is employed for camera stabilization. The system receives information regarding the initial orientation of the camera and subsequently employs a set of motors and sensors to move the camera and maintain its initial orientation. This adjustment process occurs without receiving information about the actual position of the camera but instead relies on a predetermined algorithm that takes into account the initial position and control instructions.

By utilizing this open-loop control system, the camera can be stabilized and maintained in its desired orientation based on the predefined parameters and instructions, even without continuous feedback regarding the camera's actual position.

2.3 Problem analysis

The found problems will be now explained. See 2.4 in order to better comprehend these problems.

2.3.1 Matlab as data logger

In this laboratory it is needed to send the value of the gyroscope and of the accelerometer to the PC. This data are sent via serial port by the stm to the PC. In order to read this data in the PC a Matlab program has been written (see 2.5). Unfortunately, it has been found that when using Matlab to read data from the logger some issues may occur: *serial_ datalog()* function sometimes makes Matlab unable to read anything in the serial port. The solution to this problem is to restart the program or use a logger written in a different language.

2.3.2 Precision of angles

When writing the code we tried to save angles values inside some `int` variables. This was done because such kinds of variables are more efficient in terms of memory. Furthermore, we also initially considered that both servos have a finite precision of 1 degree. Anyway, when executing the program, we noticed that nothing was working (in particular the bonus exercise). This is because C operates some cast while executing mathematical operations. For this reason, we decided to store all variables as `float`.

2.4 Code structure

Before implementing the controllers required in exercise 1 and in bonus points, we first observed the data we got from the logger. In fig.2.3 we can appreciate an example of the data we got by rotating the robot. To get data along all the three axes, we modified the data structure that is sent to the logger. In fact, we have added a third component both for `u` and `w`. In this way we can transmit properly all the three components of the gyroscope and of the accelerometer. In the same code section we also define the variables `pan`, `tilt` that are used to control the servomotor, and the auxiliary variable `float` which is used in bonus exercise.

```
/* declare an ertc_dlog struct */
struct ertc_dlog logger;
struct datalog
{
  float w1, w2, w3;
  float u1, u2, u3;
} logger_data;


int8_t pan = 0;
int8_t tilt = 0;
float angle = 0;
```

Listing 2.1: Logging structure and auxiliary variables

2.4.1 Exercise 1

Let's have a look at listing 2.2. We are looking at the `while()` cycle inside the `void main()`, since this section has to be executed infinitely many times. The `HAL_DELAY()` command regulates the reading of the IMU at the correct frequency. Furthermore, we have to pass the correct values to the structure `logger_data()`. The reading of the IMU is possible thanks to the auxiliary routines `bno055_convert_double_accel_xyz_msq()` and `bno055_convert_double_gyro_xyz_rps()` (already implemented in the base project). Notice that the values we read has to be converted also in the right measurement unit.

To implement the required camera stabilizer we have just to use the formula provided in the handout

$$\theta = \sin\left(\frac{a_y}{g}\right)^{-1}$$

that gives the angle of which the Turtle-bot is rotated with respect to the horizon. It is sufficient to evaluate the tilt angle using the formula above with the sign minus. In fact, to contrast the effect of the rotation and obtain the correct compensation, we need to give as input to the servomotor the opposite angle.

Notation mismatch: The code reported in listing 2.2 perfectly works. Anyway `pan` and `tilt` angles are inverted with respect to their physical meaning (`pan` angle in the code corresponds to the tilt angle in the handout and vice-versa).

2.4.2 [Extra] Exercise 2

We developed a control action that compensates for the rotation around the z-axis. Hence, since the start of the program, the camera has to keep looking at the same point. This clearly is limited to angles that the servo is able to reach. To implement this correction we adopted a numerical integration method using the data from the z-axis of the gyroscope. Since we are reading an angular velocity measured in revolutions per second and we want to get an angle we adopt the following discrete integration formula.

$$\theta^{(k+1)} = \theta^(k) + \omega_{rps} T_s f_{\text{rad2deg}}$$

where ω_{rps} is the speed read from the gyroscope, f_{rad2deg} is the conversion term from radiant to degrees and T_s is the sampling time at what we get the reading from the gyroscope. At each instant we save in the auxiliary variable `theta` the angle of which we are rotated wrt the starting position. As in exercise 1, to correct the position we need to give as input to the servo the opposite angle, namely `tilt`.

Reading the data from the logger we noticed that leaving the bot stuck in a fixed position, the reading from the gyroscope presented a non-null mean value. This bias, which can be brutally estimated considering the mean of such values in a sufficiently long interval of time, could affect the integration. Fortunately, the values of the bias along the three axes are so small that we are not able to notice any drift effect from the desired position. Furthermore, we can consider that the noise affecting the measures is white Gaussian, hence its means is zero and the effect on the integrator are not relevant.

```
  /* USER CODE BEGIN WHILE */
float theta = 0;

  while (1)
  {
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */

    HAL_Delay(20);
    bno055_convert_double_accel_xyz_msq(&d_accel_xyz);
    bno055_convert_double_gyro_xyz_rps(&d_gyro_xyz);

    logger_data.w1 = d_accel_xyz.x;
    logger_data.w2 = d_accel_xyz.y;
    logger_data.w3 = d_accel_xyz.z;
    logger_data.u1 = d_gyro_xyz.x;
    logger_data.u2 = d_gyro_xyz.y;
    logger_data.u3 = d_gyro_xyz.z;

    ertc_dlog_send(&logger, &logger_data, sizeof(logger_data));
    ertc_dlog_update(&logger);

      pan = - asin(d_accel_xyz.y/9.81)*180/3.1416  ;

    theta = theta + (d_gyro_xyz.z)*180/3.1416*0.02 ;
    tilt = - theta;

    /* update pan-tilt camera */
  __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_3,
          (uint32_t)saturate((150+pan*(50.0/45.0)), SERVO_MIN_VALUE, SERVO_MAX_VALUE)); //
    tilt
  __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_2,
          (uint32_t)saturate((150+tilt*(50.0/45.0)), SERVO_MIN_VALUE, SERVO_MAX_VALUE)); //
    pan
  }
  /* USER CODE END 3 */
}
```

Listing 2.2: Code implemented

2.5 Results

The data received from the robot are simultaneously displayed on a Matlab figure in real–time and the results are shown in Figure (2.2) for the accelerometer and Figure (2.3) for the gyroscope.
We invoke the serial datalogger by using the following command in MATLAB:

```
data = serial_datalog('COM3',{'2*single','2*single'}, 'baudrate',115200)
```

where *COM3* is the port of the serial datalogger (it may be different each time) and *'2*single','2*single', 'baudrate',115200* is a cell array of char arrays specifying the type of data the TurtleBot is sending to the PC.

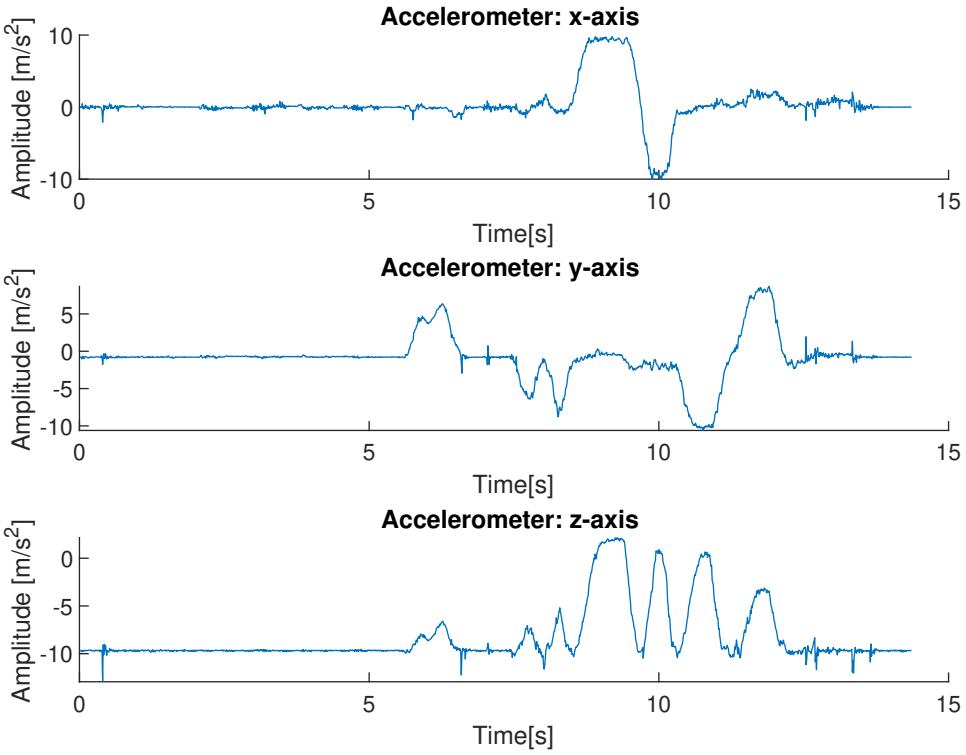


Figure 2.2: Accelerometer

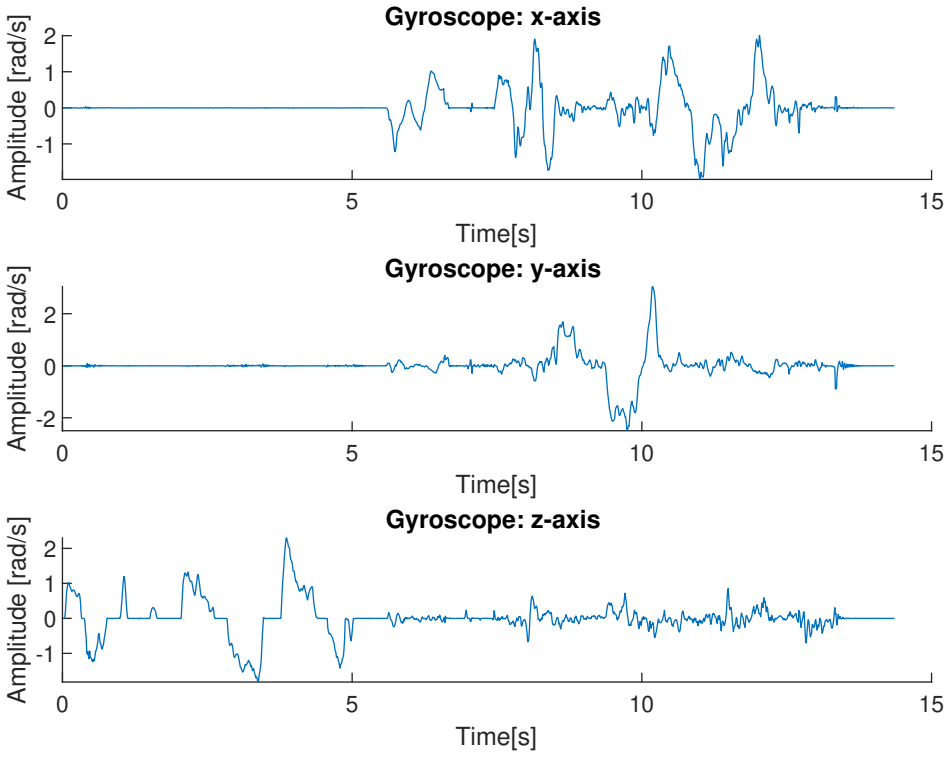


Figure 2.3: Gyroscope

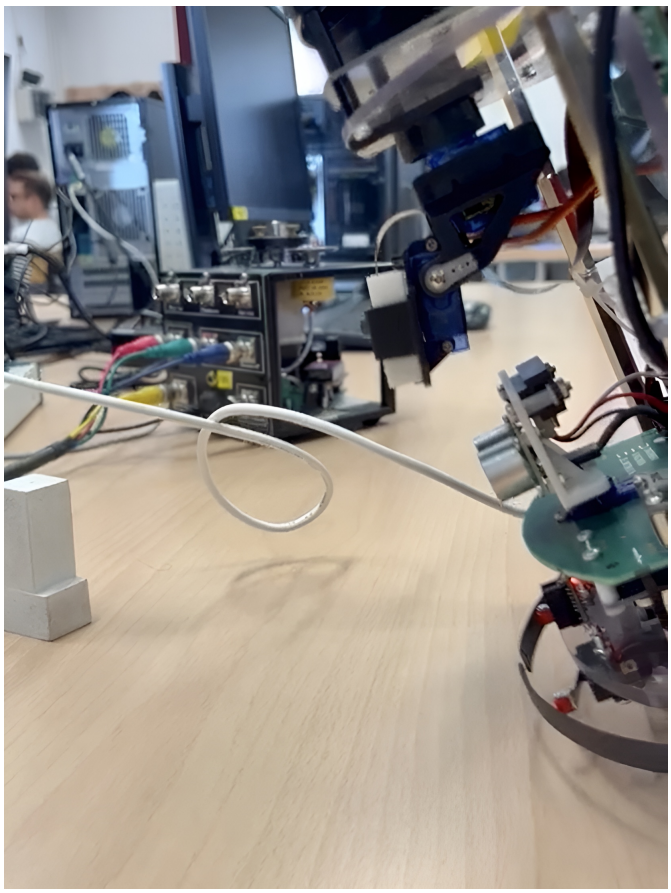


Figure 2.4: Example of camera stabilization

Chapter 3

Laboratory 3

3.1 Introduction

The aim of this laboratory is to program the TurtleBot in order to make its motors follow a constant velocity. The strategy to archive that result is to implement a PI controller. The feedback loop is closed with an encoder that provides us with the instantaneous speed of the motors. To change the reference speed of the motor the keypad is used. The overall system is shown in Figure 3.1. Timers are used as follows:

- TIM 3: Encoder for motor 1;
- TIM 4: Encoder for motor 2;
- TIM 6: Provide the clock for the controller;
- TIM 8: PWM control of both motors

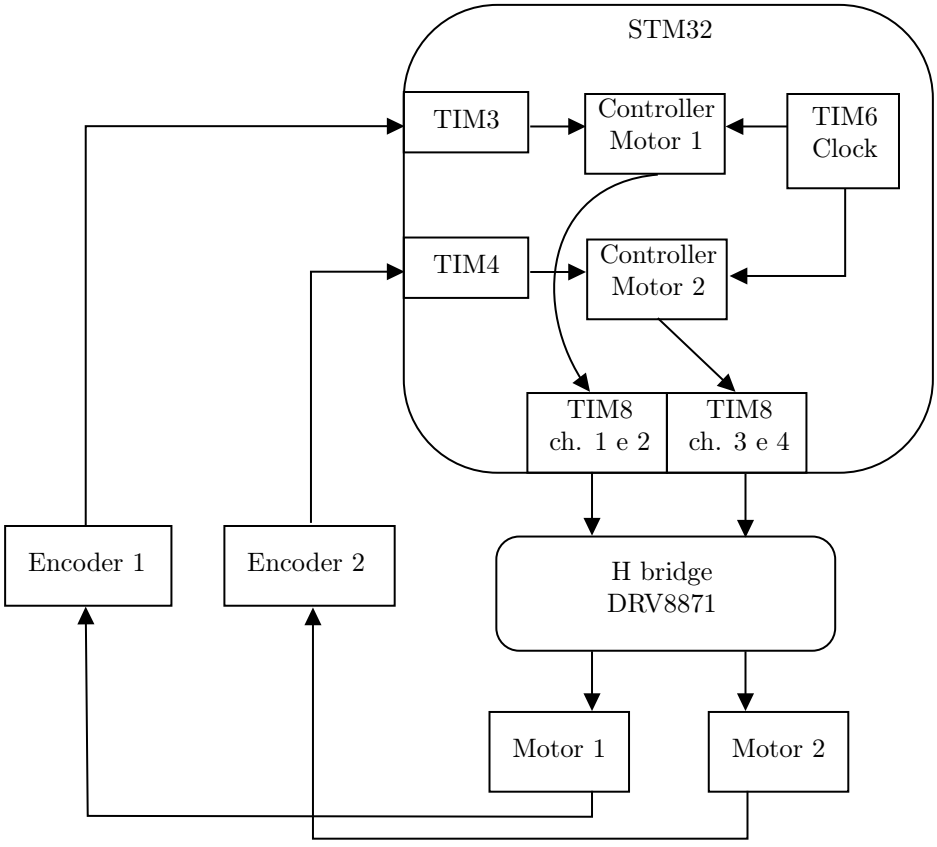


Figure 3.1: Scheme of the overall system

3.2 Theoretical notions

3.2.1 H-bridge

When heavy loads such as motors are integrated into a circuit, it is not feasible to power them directly from the pins of a standard system microcontroller. Instead, the microcontroller is utilized for controlling suitable driver circuitry.

H-bridge drivers have long been employed as a method for facilitating bidirectional motor driving. Through the use of an H-bridge, motor rotation can be driven, and the supply polarity to the motor can be reversed to alter the direction of rotation. It also facilitates braking when necessary.

The fundamental principle of an H-bridge is relatively straightforward. It comprises a configuration of four switches, typically MOSFETs. By activating one set of diagonally-opposed switches, the motor can be driven in one direction (clockwise). Activating the other set of diagonally-opposed switches allows the motor to be driven in the opposite direction (anti-clockwise). A PWM signal is employed to regulate the motor's speed.

DRV8871

This project involves the implementation of a PWM-based motor control system using the DRV8871, a brushed DC motor driver by Texas Instruments. The DRV8871 is specifically chosen for its ability to convert a PWM input into a suitable voltage and current output for driving the motor. With the DRV8871's integrated H-bridge, bidirectional control of the motor is achieved. Important pins:

- **IN1** (Pin 3, Input): This pin serves as an input to the DRV8871, receiving the PWM signal from the microcontroller. It plays a crucial role in controlling the output of the H-bridge.
- **IN2** (Pin 2, Input): Another input pin of the DRV8871 receives the PWM signal from the microcontroller. It complements IN1 in controlling the H-bridge output.
- **OUT1** (Pin 6, Output): This pin represents one of the outputs of the H-bridge. It connects to the motor, allowing the transmission of controlled power signals.
- **OUT2** (Pin 8, Output): The second output pin of the H-bridge, is responsible for facilitating the bidirectional control of the motor in conjunction with OUT1.

In order to use the H-bridge we have to use a timer with the function `__HAL_TIM_SET_COMPARE()`, where we have to specify how long the timer has stay high. In this way, we can control the speed of the motors. In the following table is reported an example of usage:

IN1	IN2	OUT1	OUT2	Mode	C Command to perform only the specified operation
0	0	floating	floating	Coast	__HAL_TIM_SET_COMPARE (&htim8,TIM_CHANNEL_1,0); __HAL_TIM_SET_COMPARE (&htim8,TIM_CHANNEL_2,0);
0	1	L	H	Reverse	__HAL_TIM_SET_COMPARE (&htim8,TIM_CHANNEL_1,0) __HAL_TIM_SET_COMPARE (&htim8,TIM_CHANNEL_2,TIM8_ARR_VALUE)
1	0	H	L	Forward	__HAL_TIM_SET_COMPARE (&htim8,TIM_CHANNEL_1,TIM8_ARR_VALUE); __HAL_TIM_SET_COMPARE (&htim8,TIM_CHANNEL_2,0)
1	1	L	L	Brake	_HAL_TIM_SET_COMPARE (&htim8,TIM_CHANNEL_1,TIM8_ARR_VALUE); __HAL_TIM_SET_COMPARE (&htim8,TIM_CHANNEL_2,TIM8_ARR_VALUE);

3.2.2 Encoder

The timers in STM32 microcontrollers are used for various purposes, including encoder counting, clock generation, and PWM control. The position encoder is a sensor that converts position information into an electrical signal. The focus is on the rotary encoder, specifically the incremental type. This encoder generates pulses when the position changes. In this laboratory, TIM3 and TIM4 are used in Encoder mode. The motors used in the rover are equipped with encoders. The motors have a gearbox with a 120:1 ratio, meaning the motor

rotates 120 times for each wheel revolution. Regarding the encoder, the selected timer needs to be configured in encoder mode. The encoder provides 16 pulses per revolution. In encoder mode 2X, the number of pulses per revolution from the wheel-side is 1920. This number doubles when using encoder mode 4X (reducing quantization error). To prevent counting direction mistakes caused by noise, the "*Input Filter*" is suggested. This filter can be configured using STM32CubeIDE, with a recommended value of 15 for the parameter. The input filter samples the encoder signals at a configurable frequency, determining the logical level based on consecutive samples. The counter period can be set to the number of pulses counted in one revolution (1920 or 3840) or the maximum allowed level (65535). A quadrature encoder, which utilizes two outputs (A and B), is employed. These outputs are called quadrature outputs as they are 90 degrees out of phase. The direction of the motor depends on which signal phase leads over the other.

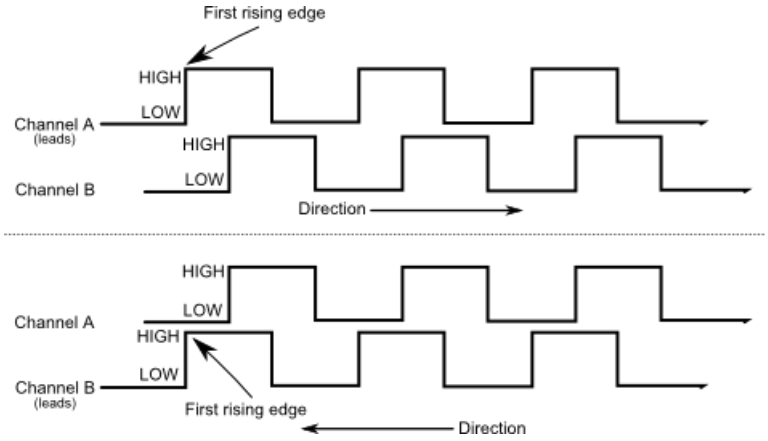


Figure 3.2: Timelines of a quadrature encoder

3.2.3 PI controller

In this laboratory, the control structure is a PI (*Proportional-Integral*) controller:

$$C(s) = K_P + \frac{K_I}{s}$$

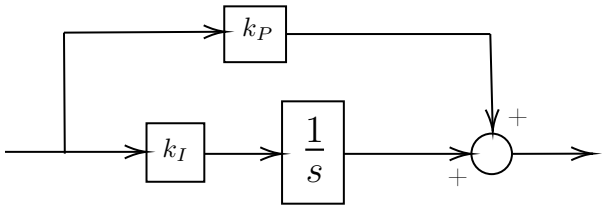


Figure 3.3: Scheme of the PI controller

The proportional-integral (PI) controller is a control mechanism that combines both proportional and integral actions to regulate a system. The proportional control action, regulated by K_P, consists in scaling the error between the desired reference value and the actual variable being controlled. This means that the controller's response is directly proportional to the magnitude of the error. A larger error leads to a stronger control action by the controller.

The integral action of the PI controller ensures that the controller retains information about past error values. Unlike the proportional action, the integral action allows for a non-zero value of the controller gain, denoted as K_I, even when the error signal is zero. This memory of past errors enables the PI controller to precisely bring the process to the desired setpoint. In cases where the proportional action alone would result in no control action, the integral action compensates by continuously integrating the positive tracking error.

However, when the controller is subject to *saturation*, where the control signal is limited by actuator constraints, issues can arise. Despite saturation, the integrator in the PI controller continues to accumulate the positive tracking error, causing the controller output to increase. However, since the actuator is already operating at its limits, the growing control command becomes ineffective for the plant. This situation arises because the integrator has accumulated a significant amount of tracking error during the previous phase, even when the plant output has risen enough to reduce the tracking error to zero.

To resolve this integrator windup problem, it is necessary for the tracking error to remain negative for a sufficient duration. This allows the integrator output to return within the linear operating range of the actuator, a process known as integrator unwinding. Only when the actuator is out of saturation can it effectively respond to the controller's output. Failure to address integrator windup can result in an excessive overshoot of the controlled variable. A possible solution to mitigate the integrator windup effects consists of implementing an integrator anti-windup circuit in the PID controller, as shown in Figure 3.4. Such structure depends on the value K_W. Notice that if this gain is too high, the control action may induce a large undershoot, making the final response really slow.

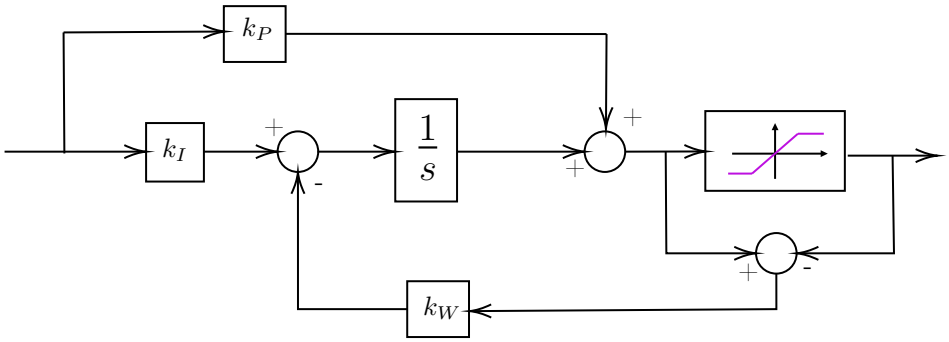


Figure 3.4: Scheme of the PI controller with the anti-windup

3.3 Code Structure

The first thing we set up was the data logger. As in laboratory 2, we used it to pass data to the PC in order to check sensor readings. For each motor, we send to the PC the voltage which is used to control the motor itself, the motor speed and the error with respect to the reference.

In order to implement the bonus point that allows selecting reference values using the keypad, we had to add to the source project the commands and the files needed to handle I2C communication.

In the first listing, we see some of the global variables needed for the activity. In particular, we observe that the value `VBATT` is assumed to be equal to 8V, though this is not true in the real bot (because the battery level changes over time). We can also observe the gains used to make the conversions between voltage and duty cycle (to control the motor) and between speeds in rpm and rad/s. Finally, we can appreciate the macro that defines the desired reference speed.

```
#define TS  0.01
#define VBATT 8.0 // Maximum voltage that can be reached
#define V2DUTY  ((float)(TIM8_ARR_VALUE+1)/VBATT)
#define DUTY2V  ((float)VBATT/(TIM8_ARR_VALUE+1))
#define RPM2RADS  2*M_PI/60
#define TIM3_ARR_VALUE 3840 // Assuming 4x definition
#define VREF 120 * 60 * 0.5 // Speed reference [rpm]
```

Listing 3.1: Global variables define for LAB3.

3.3.1 Exercise 1

To accomplish all the requests of this exercise we describe the procedure only for one motor (the code is reported only for one motor too). The procedure for the other motor is exactly the same. Since the two motors work independently, each of them has its own controller, hence most of the variables are repeated twice. Let's now see all the steps in detail.

First, we had to read the correct speed from the encoder. This part is handled inside the routine `void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)`. Each time the encoder reveals a non-null angular speed, the callback is called. The difference among two consecutive readings of the encoder is stored in the global variables `TIM3_DiffCount, TIM4_DiffCount`. This difference has to be converted into rpm to get the speed of each wheel. Such a task is accomplished by the auxiliary routine `computeRpm`, which takes into account the relations existing between motor speed, wheel speed and encoder reading. After computing the speed, we evaluate the error existing between such value and the reference speed (it is called `speed` because this value can be changed using the keyboard). Such error (converted in rad/s) is used to feed the PI functions that control the motors. The PI function evaluates at each step the proportional and the integral components. Then the final voltage

is saturated using the `float saturate` function. This function limits the absolute value of the output voltage to 4.0V in order to prevent damaging the motors themselves.

The parameters of the PI controllers have been tuned by hand, in order to guarantee satisfactory performances. Any specific requirement was given. Such choice of parameters granted a good transient if considering the initial step response. Furthermore, they also grant a balanced action between the integral and the proportional part. Then the output from the PI controller is converted into the duty cycle values using the constant V2DUTY. Then this value is passed to a routine that implements motor control in forward and coast (forward and brake is the commented part). Finally, we pass the values to the logger (code not reported because is similar to the one in lab. 2).

We also implemented a function that allows one to select the speed reference using the keyboard. To do this, we had to modify the original project to add all the commands useful to initialize the I2C. The working process of this routine is the same as the one we already analysed in laboratory activity 1 (hence such code is not reported).

```
// Global variables used for the econder
int32_t TIM3_CurrentCount;
int32_t TIM3_DiffCount;

// Global variables used speed control
float duty1 = 0;
float error1 = 0;
float vIn1;
float s1;
float v1;

// Global variables used for the PID
// Assume that the error is computed in rpm
float KI = 0.05;
float KP = 0.1;
float KW = 12;
float I1 = 0;
float voltage = 0;

// Saturation function
float VLIM = VBATT / 2;
float saturate(float voltage)
{
  if (voltage > VLIM)
    voltage = VLIM;
  else if (voltage < -VLIM)
    voltage = -VLIM;
  return voltage;
}

// Function to compute the PID without the anti-wind up
float PID1(float error)
{
  voltage = 0;
  I1 = I1 + error * KI * TS;
  voltage = KP * error + I1;
  voltage = saturate(voltage);
  return voltage;
}

float computeRpm(int32_t encoder_read)
{
  float revolutions = 0;
  float motor_rpm = 0;
  revolutions = ((float)encoder_read / 3840.0);
  float wheel_rpm = 60.0 * revolutions / (float)TS;
  motor_rpm = wheel_rpm * 120;
  return (float)motor_rpm;
}

float speed = VREF;
int actual_keyboard_value;

//____________Keyboard Callback____________

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  // && logger.tx_enable
  if (htim->Instance == TIM6)
  {
```

```
61
62      static int32_t TIM3_PreviousCount = 0;
63
64      TIM3_CurrentCount = (int32_t) __HAL_TIM_GET_COUNTER(&htim3);
65
66      /* evaluate increment of TIM3 counter from previous count */
67      if (__HAL_TIM_IS_TIM_COUNTING_DOWN(&htim3))
68      {
69        /* check for counter underflow */
70        if (TIM3_CurrentCount <= TIM3_PreviousCount)
71          TIM3_DiffCount = TIM3_CurrentCount - TIM3_PreviousCount;
72        else
73          TIM3_DiffCount = -((TIM3_ARR_VALUE + 1) - TIM3_CurrentCount) - TIM3_PreviousCount;
74      }
75      else
76      {
77        /* check for counter overflow */
78        if (TIM3_CurrentCount >= TIM3_PreviousCount)
79          TIM3_DiffCount = TIM3_CurrentCount - TIM3_PreviousCount;
80        else
81          TIM3_DiffCount = ((TIM3_ARR_VALUE + 1) - TIM3_PreviousCount) + TIM3_CurrentCount;
82      }
83
84      TIM3_PreviousCount = TIM3_CurrentCount;
85
86      // Compute rpm considering 16 pulses per round
87      // This speed [rpm] is considered at the motor side
88      v1 = computeRpm(TIM3_DiffCount);
89
90      // Compute tracking error
91      error1 = (float)(speed - v1);
92
93      // By reading the maximum value of the error using our voltage limit, we get that
94      // the maximum error is 3600.
95      // To prevent errors due to strange wrong readings of the encoder, we limit this value
96      if(error1 > 3600)
97        error1 = 3600;
98      else if(error1 <-3600)
99        error1 = -3600;
100
101     // Compute control input with PID
102     vIn1 = PID1(RPM2RADS * error1);
103     // Compute duty cycle
104     duty1 = V2DUTY * vIn1;
105
106     //____________Motors Control____________
107     //____________Data logger____________
108   }
109 }
```

Listing 3.2: PID implementation

3.3.2 Exercise 2

In the listing below we see the two versions of the motor control we can implement. The first one, which is not commented, is the forward and coast mode. The commented part, instead, is related to the forward and brake mode. We tested motor response in both of these two setups. We can appreciate the differences in fig. 3.5 and in fig. 3.6. In practice, we can conclude that there the two motor modes offer quite the same performances (even tough the data from forward and coast are a little bit less noisy).

```
1       //____________Motors____________
2       /* Alternate between forward and coast */
3       /* [Commented] alternate between forward and brake */
4
5       // Motor 1
6       if (duty1 >= 0) {
7          __HAL_TIM_SET_COMPARE(&htim8, TIM_CHANNEL_1, (uint32_t)duty1);
8          __HAL_TIM_SET_COMPARE(&htim8, TIM_CHANNEL_2, 0);
9          //__HAL_TIM_SET_COMPARE(&htim8, TIM_CHANNEL_1, (uint32_t)TIM8_ARR_VALUE);
10         //__HAL_TIM_SET_COMPARE(&htim8, TIM_CHANNEL_2, TIM8_ARR_VALUE - duty1);
11      } else {
12        __HAL_TIM_SET_COMPARE(&htim8, TIM_CHANNEL_1, 0);
13        __HAL_TIM_SET_COMPARE(&htim8, TIM_CHANNEL_2, (uint32_t)-duty1);
14      }
```

Listing 3.3: Motor control

3.3.3 Bonus

As done for exercise 1, we report the controller of only one motor. In the listing below we can look at the function `float PIDAW1` that implements a PI with the anti-windup mechanism. In order to implement this control action (that has the aim of reducing the side effects of saturation of the controller), we exploit an auxiliary variable `float buf1[2]` . We store the controller output values before and after the saturation function. Then we remove this quantity, scaled by a certain gain `KW`, from the value that feeds the integrator. Notice that the presence of this array introduces a delay of one step. In fact, the saturation effects are compensated in the controller only one step after the saturation occurs.

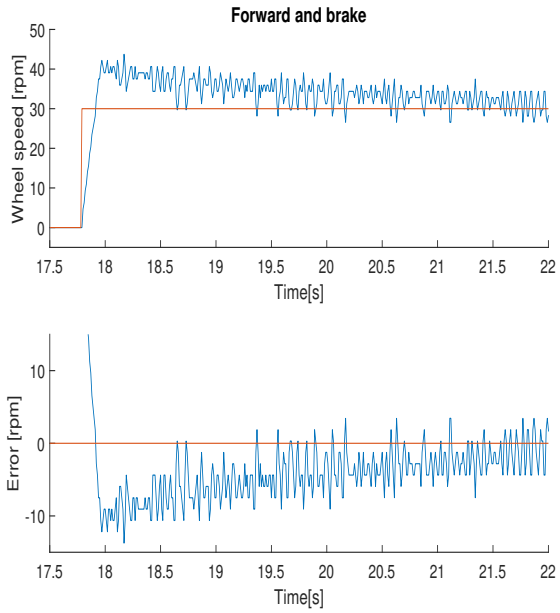
As we can see both in fig. 3.5 and in fig. 3.6, the introduction of the anti-windup, under our choice of parameters, removes completely the overshoot.

```
float buf1[2] = {0, 0};
float PIDAW1(float error)
{
  voltage = 0;
  // I = I + error * KI *TS ;
  I1 = I1 + (error - KW * (buf1[0] - buf1[1])) * KI * TS;
  voltage = KP * error + I1;
  buf1[0] = voltage;
  voltage = saturate(voltage);
  buf1[1] = voltage;
  return voltage;
}
```

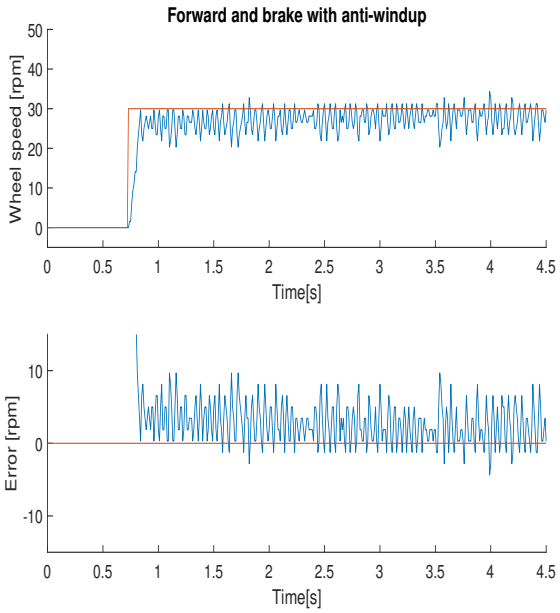
Reference speed We notice that with such a choice of parameters plus the anti-windup, overshoot completely disappears. Anyway, we didn't test the motors using the maximum reference speed we can achieve with a saturation of 8V. In fact, overshoot also depends on the magnitude of the step input we apply (this is due also to the presence of the saturator, which introduces a non-linear behaviour in the step response).

3.4 Results

In the figures below are reported the four possible behaviours of the motor, due to the different controllers (PI with or without anti-windup) and modes (forward and coast/ forward and brake). Furthermore, we can notice that all the measurements are quite noisy. This is mainly due to the fact that we are working in discrete time and there are a lot of nonlinearities in the system. Notice that we considered different instants for the measurements to show, but the size of the temporal window shown is exactly the same. One last thing to be mentioned is the fact that the two motors behave in quite a different way. This fact cannot be appreciated by the two figures below, since the report the data only from motor 1.

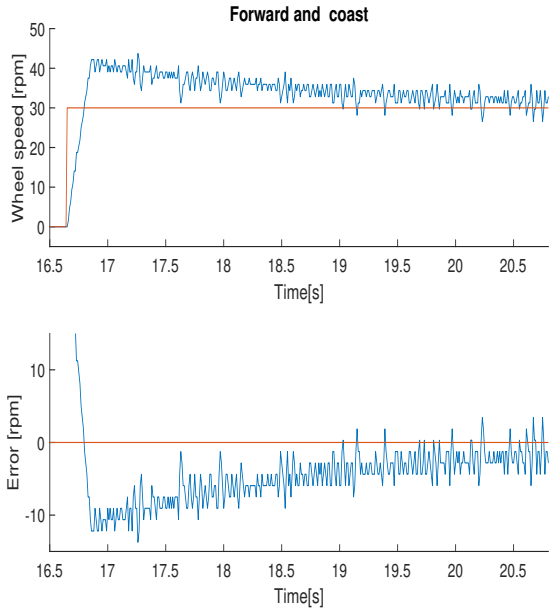


(a) Forward and brake (motor 1)

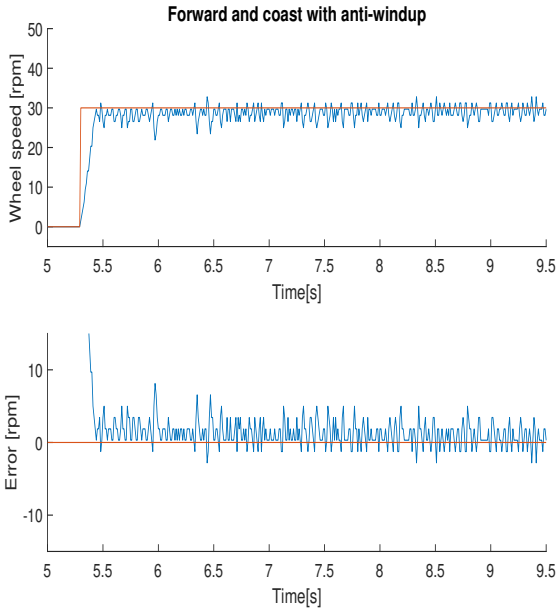


(b) Forward and brake with anti-windup (motor 1)

Figure 3.5: 30 rpm step response with forward and brake



(a) Forward and coast (motor 1)



(b) Forward and coast with anti-windup (motor 1)

Figure 3.6: 30 rpm step response with forward and coast

Chapter 4

Laboratory 4

4.1 Introduction

The aim of this lab is to make the TurtleBot able to follow a line through different paths. The bot has to do it as fast as possible, without losing the path. In order to do that a line sensor is used and a controller has to be implemented to control the speed of each motor.

4.2 Theoretical Notions

4.2.1 The line sensor

The line sensor used is the Pololu qtr-8A. The basic principle of this sensor is easy: there is a LED that emits in the infrared bandwidth, and a phototransistor that detects the presence of light. If there is a black area above the sensor the infrared light will be absorbed and the phototransistor will not detect power. The device used has 8 emitters and reflectance sensors spaced 9.525 mm to detect the line. The phases of the sensor are basically the following:

1. Turn on IR LEDs
2. Set the I/O line to an output and drive it high
3. Allow at least 10 μs for the sensor output to rise
4. Make the I/O line an input
5. Measure the time for the voltage to decay by waiting for the I/O line to go low
6. Turn off IR LEDs

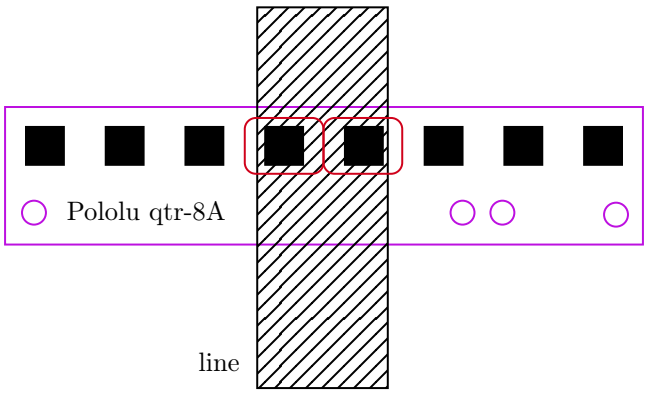


Figure 4.1: Line Sensor

To define the line pursuit objective we need to focus on the position error. Line tracking error represents the difference between the desired line position (centre of the line) and the actual line position detected by our

vision system. The error can be defined as

$$e_{\text{SL}} = \frac{\sum_{n=0}^{N-1} b_n w_n}{\sum_{n=0}^{N-1} b_n} \tag{4.1}$$

where the sequence representing the set of active sensors is

$$b_n = \begin{cases} 1, & \text{if sensor } n \text{ is active} \\ 0, & \text{if sensor } n \text{ is not active} \end{cases} \tag{4.2}$$

and the distance between the sensor at position n and the centre of the sensor array is

$$w_n = \left(\frac{N-1}{2} - n \right) P, \quad n \in \{0, 1, \ldots, N-1\} \tag{4.3}$$

4.2.2 Yaw controller

The yaw control system corresponds to the ability to adjust the yaw angle of a vehicle around its vertical axis, where yaw corresponds to rotation around a horizontal plane. Inside our Turtlebot is designed to adjust the yaw angle to a constant reference point so that the TurtleBot maintains a specific orientation or direction. By using sensors to measure the orientation of an object in space, the error relative to the yaw angle can be calculated by comparing the current angle to the desired or reference angle. The yaw error is defined:

$$\psi_{\text{err}} = \psi_{\text{ref}} - \psi \tag{4.4}$$

where:

$$\psi_{\text{err}} = \arctan\left(\frac{e_{\text{SL}}}{H} \right) \tag{4.5}$$

also assuming that this ratio is close to 0 we can obtain the following approximation

$$\psi_{\text{err}} \approx \frac{e_{\text{SL}}}{H} \tag{4.6}$$

The goal is to minimize this error and keep the rover's yaw angle as close to the desired value as possible. The yaw error is calculated and used as input for the P controller, a proportional controller used to maintain orientation that produces an appropriate output to adjust the roll rate. In fact the controller is designed so that the input is the yaw angle error, denoted as ψ_{err}, and the output is the yaw rate, represented as ψ. By implementing a feedback control on the speed of each wheel using PID controllers, we can approximate our system, denoted as P(s), as an integrator and thus plant transfer function as .

$$P(s) \approx \frac{1}{s} \tag{4.7}$$

4.3 Problem Analysis

4.4 Code structure

In order to make everything works, we just used again Laboratory 3 code as the base project for this activity. Files organisation: when looking at the submitted folders containing all the code used for labs, we can notice that LAB3 and LAB4 contain files with the same name. Anyway, if we look at the *main.c* in both folders, such a file is different. This mismatch in naming correctly each laboratory file was due to some problems with the STM32CubeIDE. Since the structure of laboratory 3 and 4 is very similar, we don't show again the routines to control motors, and the part related to I2C usage. Anyway, we used wireless telemetry data logging (to use this function we just followed the given instructions).

Error evaluation The first thing to do in order to understand the robot's position is to evaluate the line error. We read the line sensor and we compute the error e_{SL} as explained previously in (4.1). To do this we exploit the auxiliary routine `compute_error()` that uses the variable `omega_n` to weights the sensor's readings (see (4.3)). Notice that in the case that any of the sensors assume value 1, then the variable `count` at the end of the `while()` routine is zero. In this case, to prevent an error (division by zero) that may cause unexpected behaviour of the robot, we set the final error to zero manually.

```
const float omega_n[8] = {3.5, 2.5, 1.5, 0.5, -0.5, -1.5,-2.5, -3.5};

float compute_error(uint8_t line_sensor_data) {
  uint8_t sens_data = line_sensor_data;
  float err_sum = 0;
  int i= 0;
  int count = 0;
  while (sens_data) {
    count += sens_data & 1;
    err_sum += omega_n[i]*(sens_data & 1);
    sens_data >>= 1;
    i++;
  }
  if(count == 0)
    return (float) 0;
  return (float)(err_sum/count);
}
```

Listing 4.1: Error evaluation.

Controller implementation To solve all the points of this lab we implemented only one controller. The main idea is very simple. In fact, it works in open loop exploiting some adaptive gains.

The first thing we do is to evaluate the error, which is stored in the variable `crr`. We notice that such a variable assumes only a finite number of values, that are bounded in the interval $[-6, 6]$. Furthermore, all the possible computed errors have a decimal part that is 0 or 0.5.

After evaluating the error, we consider the motor control part. We set up the reference speed `V_ref` and a suitable gain `KPMOTOR` to scale the error. Then we develop our control action. It relies on three different scenarios, based on the magnitude of `crr`. If this magnitude is small (with the given bounds it can only be zero) we pass to both motors the same speed (also a term in `V_ref` appears, but it can be neglected because it is zero). We scale `V_ref` by 0.85 because otherwise the TurtbleBot gets too fast and it cannot track properly the line.

All the other two scenarios are based on the following idea. The bigger the error is, the strong the correction we have to apply. Hence `V_ref` is multiplied by a smaller value if the error is big. Furthermore, we also apply a correction term which is scaled by a higher coefficient. Notice also that we sum the correction term to a motor and we subtract it from the other. This allows a better and faster response. In very sharp edges we can observe that one of the motor stops. With such a choice of parameters, we were able to complete the challenge circuit in about 19 seconds.

Notice that after computing the speeds of each motor, such values are scaled by a suitable gain (chosen empirically), and then passed to the saturator. Such values (supposed to be in V) are finally converted in duty cycle and used to control both motors. Notice that the parameter that defines the saturator strongly impacts the performance of the TurtleBot. In fact, by setting a low saturation value we obtain two main effects: on the one hand we prevent the motors to be damaged, on the other hand, the TBot moves slowly. This makes its whole line tracking smooth. If we want to get better performances we have to push up the saturation level, facing some side effects. As a result, the bot moves faster, anyway when it is on a segment with a straight line it starts oscillating.

Although this problem, our TurtleBot was able to complete also the most challenging circuit, with "salt and pepper" noise and interrupted line segments. The only point in which the line following fails is in the proximity of the shaded grey edge. In this case, there is not enough contrast and the reflectance sensors do not see the line. To complete this circuit we used a lower saturation voltage in order to keep the speed limited. This is because of the interruption of the line on the edges.

```
float ref_1;
float ref_2;
float V_ref = 50;
float KPMOTOR=30.0;

uint8_t buff;
HAL_StatusTypeDef status =  HAL_I2C_Mem_Read(&hi2c1, SX1509_I2C_ADDR1 << 1, REG_DATA_B, 1, &
    buff, 1, HAL_TIMEOUT);
float crr = compute_error(buff);

if(crr > 2.4 && crr < -2.4)
{
  ref_1 = 0.3*V_ref + 1.5*KPMOTOR*crr;
  ref_2 = 0.3*V_ref - 1.5*KPMOTOR*crr;
}
else if(crr > 1.4 && crr < -1.4)
{
  ref_1 = 0.6*V_ref + 1.3*KPMOTOR*crr;
```

```
18    ref_2 = 0.6*V_ref - 1.3*KPMOTOR*crr;
19  }
20  else
21  {
22    ref_1 = 0.85*V_ref + KPMOTOR*crr;
23    ref_2 = 0.85*V_ref - KPMOTOR*crr;
24  }
25
26  // Compute control input
27  vIn1 = saturate1(ref_1*0.05);
28  vIn2 = saturate1(ref_2*0.05);
29
30  // Compute duty cycle
31  duty1 = V2DUTY * vIn1;
32  duty2 = V2DUTY * vIn2;
```

Listing 4.2: Control routine.

Data logging In this activity, we used the wifi logger to analyze the behaviour of the bot. The code is not reported because is exactly the same as the preceding labs.

Possible controller refinements In order to fix oscillations in the straight line sections (due to the robot acceleration when the error is zero), we could have implemented a PI adaptive controller. This solution, adopting some variables gains as seen above, could have fixed the problem.

Important notes Despite we mentioned the Yaw controller, we didn't implement it. Anyway, in order to obtain the yaw error, it is enough to divide `crr` by a suitable value (see (4.6)).

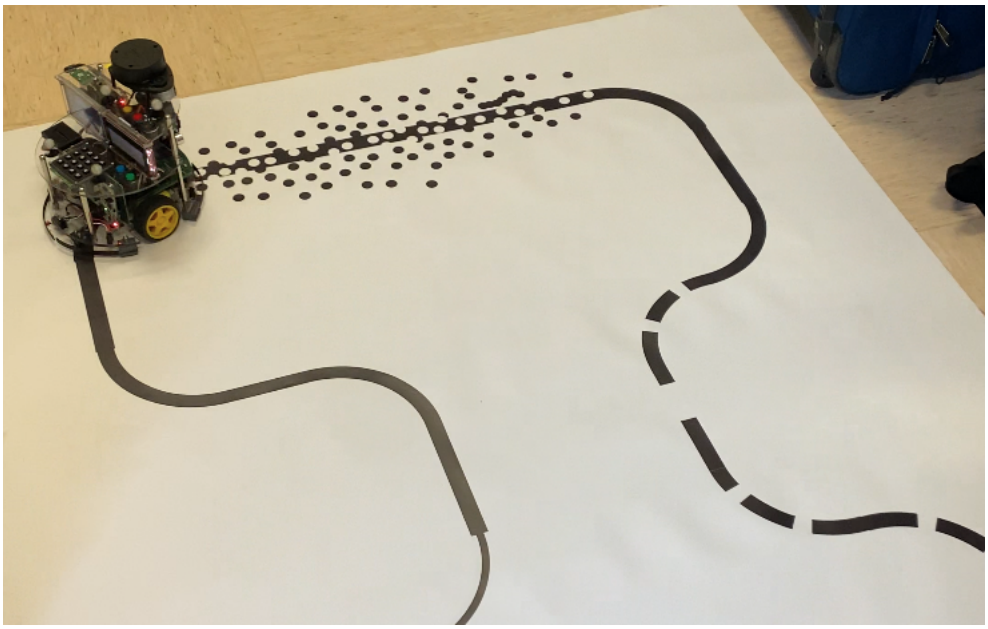


Figure 4.2: TurtleBot on the most challenging track

Chapter 5

Laboratory 5

5.1 Introduction

The aim of this laboratory is to use the task capabilities of FreeRTOS. In particular, this activity is based on the ones already completed in laboratory 1 (extra part) and in laboratory 3.

5.2 Theoretical notions

In this section, we are going to complete theoretical notions already seen in laboratory 1. In particular we are going to see more in details how a *mutex* work and the use of *queues* instead of semaphores

5.2.1 Critical region

When performing operations on data in memory, the data is typically loaded into a register within the CPU, modified there, and then written back to memory. This is done to optimize the speed of computations and take advantage of the CPU's registers. In a multi-threaded or multi-process environment, if two or more tasks attempt to modify the same shared data simultaneously, a race condition can occur. A race condition happens when the behaviour of the program depends on the relative timing of operations in different threads, leading to unpredictable and erroneous results. To prevent race conditions, the region of code where shared resources are accessed and modified is typically referred to as the critical region. Access to the critical region needs to be synchronized to ensure that only one task can enter and modify the shared resource at a time.

Synchronization mechanisms such as mutexes, semaphores, or other forms of locks are used to coordinate access to critical regions. These mechanisms provide mutual exclusion, ensuring that only one thread or process can enter the critical region at any given time. By protecting critical regions with synchronization mechanisms, race conditions can be avoided, and the integrity of shared data can be maintained.

5.2.2 Produce and consumer problem

The Producer-Consumer problem is a classic synchronization problem in concurrent programming, involving two main characters: producers and consumers. The problem arises when multiple producers and consumers share a common buffer or resource for communication and they both try to access this resource at the same time.

The scenario can be visualized as a buffer or queue that is shared between producers and consumers. Producers generate data items and place them into the buffer, while consumers retrieve and process those data items from the buffer. The challenge lies in coordinating access to the shared buffer to avoid race conditions or data inconsistencies. Here's an explanation of the steps involved in the Producer-Consumer problem:

1. Producers and consumers operate concurrently, running in separate threads or processes.
2. Producers generate data items and attempt to place them in the shared buffer.
3. If the buffer is full (all slots occupied), producers must wait until a consumer removes an item from the buffer, creating an empty slot.
4. Consumers retrieve data items from the buffer and process them.
5. If the buffer is empty (no items available), consumers must wait until a producer places a data item in the buffer.

6. The synchronization mechanism ensures that only one producer or one consumer can access the buffer at a time, avoiding conflicts and maintaining the integrity of the data.

To achieve synchronization and mutual exclusion, various synchronization primitives can be used, such as mutexes or semaphores.

5.2.3 Mutex

A mutex (short for mutual exclusion) is a synchronization mechanism used in concurrent programming to protect shared resources from simultaneous access by multiple threads or processes. It provides a way to ensure that only one thread or process can execute a critical section of code, commonly referred to as a protected resource, at any given time. A mutex typically has two states: *locked* and *unlocked*. When a thread wants to access the protected resource, it tries to acquire the mutex. If the mutex is locked by another thread, the requesting thread is blocked (suspended) until the mutex becomes available. Once a thread acquires the mutex, it gains exclusive access to the protected resource. Other threads that attempt to acquire the same mutex will be blocked until it is released by the owning thread. Mutexes are commonly used in multithreaded or multi-process applications to ensure data consistency and prevent race conditions, where multiple threads or processes access shared data concurrently, leading to unpredictable results. Unlike binary semaphores, which can be used for more complex synchronization scenarios involving counting and signalling, mutexes are typically designed for simple mutual exclusion scenarios, where a resource needs to be accessed by only one thread or process at a time. Mutexes are often implemented as operating system primitives and provided as part of threading or synchronization libraries in programming languages. They may differ in terms of implementation details, behaviour, and additional features offered.

5.2.4 Queues

In real-time operating systems (RTOS), queues serve as communication channels between tasks, facilitating the transfer of data. By default, a queue in an RTOS follows the FIFO (First In, First Out) order, but it can be configured to operate as a LIFO (Last In, First Out) structure. In the freeRTOS API, two distinct types of queues are available:

1. **Message queues**: These queues allow the transmission of integer data or pointers only. When creating a message queue, you specify the maximum number of messages it can hold, the size of each message, and the data type being sent. Messages can be simple variables or pointers to more complex structures.

2. **Mail** queues: Mail queues are designed for sending memory blocks. They are particularly useful when larger data structures or buffers need to be transferred between tasks. During the creation of a mail queue, you specify the maximum number of items and the size of each item.

Both message queues and mail queues in freeRTOS support blocking and non-blocking send and receive operations. Tasks can block on a queue when sending or receiving data, waiting for space or data to become available. Non-blocking operations return immediately, allowing tasks to continue execution even if the queue is full or empty.

5.3 Code structure

In this paragraph, the practical execution of the lab will be discussed. As mentioned in the introduction the objective is to coordinate the execution of the tasks using freeRTOS, eliminating the 3 callbacks that control the motor, read the line sensor and communicate with the data logger.
Exercise 1 has the scope to enable FreeRTOS, following the steps in the GUI. After deleting the callbacks, writing tasks and setting the initial value of the counter to 0 the first Exercise is concluded.

5.3.1 Exercise 2

Let's start now start inserting the `osSemaphoreAcquire(binarySemSyncHandle, osWaitForever);` inside the `StartControlTask()`, the task that has the ownership to control the motors. Doing this operation the motors stop running because the semaphore is never released and the system will wait forever. A solution to this issue is to place the `osSemaphoreRelease(binarySemSyncHandle);` inside the `StartLineTask()`. With that solution whenever the system completes the reading of the line sensor the system can continue the execution of the `StartControlTask()`. The system now works properly.

5.3.2 Exercise 3

A much more interesting problem is exercise 3. The aim of this assignment is to implement a system that is able to acquire data in one task and send them in another one. The issue that could occur is that one task read the memory while the other is writing. It could be seen that without protection the system prints casual data. The truth is that the system is reading an incomplete written memory location. This problem can be reconducted to the famous producer and consumer problem. A solution to this problem is to use *Mutex*. Whenever the system needs to read data it acquire the resource and no other tasks with *Mutex* can access on it until the *Mutex* is released. The code is reported below:

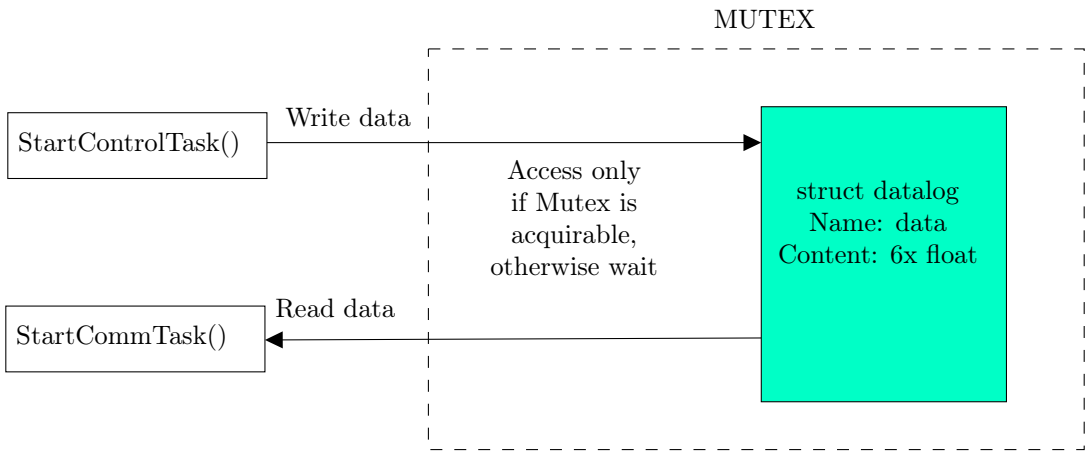


Figure 5.1: Working principle of the code of exercise 3.

```
1  void StartControlTask(void *argument)
2  {
3    for(;;)
4    {
5          osSemaphoreAcquire(binarySemSyncHandle, osWaitForever);
6
7          // control motor code from laboratory 3
8
9          osMutexAcquire(DataMutexHandle, osWaitForever);
10         /* Enter Critical section */
11
12         /* Data passed to the datalogger structure */
13
14         /* Exit Critical section */
15         osMutexRelease(DataMutexHandle);
16   }
17 }
18
19 void StartLineTask(void *argument)
20 {
21   for(;;)
22   {
```

```
23      HAL_StatusTypeDef status =  HAL_I2C_Mem_Read(&hi2c1, SX1509_I2C_ADDR1 << 1, REG_DATA_B, 1,
         &buff, 1, HAL_TIMEOUT);
24      osSemaphoreRelease(binarySemSyncHandle);
25    }
26  }
27
28  void StartCommTask(void *argument)
29  {
30    for(;;)
31    {
32      osMutexAcquire(DataMutexHandle, osWaitForever);
33      ertc_dlog_send(&logger, &data, sizeof(data));
34      ertc_dlog_update(&logger);
35      osMutexRelease(DataMutexHandle);
36    }
37
38  }
```

Listing 5.1: Code of the Mutex exercise.

5.3.3 Exercise 4

Another solution to the previous problem is to use protected queue: data are put in a protected queue and whenever they are available they are taken by th second task and sent. The code is reported below:

```
1   void StartControlTask(void *argument)
2   {
3     for(;;)
4     {
5         osSemaphoreAcquire(binarySemSyncHandle, osWaitForever);
6               //osMutexAcquire(DataMutexHandle, osWaitForever);
7               .
8               .    // control motor code from laboratory 3
9               .
10              /* Data passed to the datalogger structure */
11
12          osMessageQueuePut("queue1", &data, 0, 0);
13            //osMutexRelease(DataMutexHandle);
14        /* Exit Critical section */
15
16    }
17  }
18
19  void StartLineTask(void *argument)
20  {
21    for(;;)
22    {
23      HAL_StatusTypeDef status =  HAL_I2C_Mem_Read(&hi2c1, SX1509_I2C_ADDR1 << 1, REG_DATA_B, 1,
         &buff, 1, HAL_TIMEOUT);
24      osSemaphoreRelease(binarySemSyncHandle);
25    }
26  }
27
28  void StartCommTask(void *argument)
29  {
30    for(;;)
31    {
32      osMessageQueueGet("queue1", &data, 0, 0);
33      if(osMessageQueueGetCapacity("queue1"))
34      {
35          ertc_dlog_send(&logger, &data, sizeof(data));
36          ertc_dlog_update(&logger);
37      }
38    }
39  }
```

Listing 5.2: Code of the queue exercise.

Unfortunately the code of this last exercise doesn't work properly: the system isn't able to correctly send data to the logger. The reason of this issue isn't clear due to the lack of time to debug of this laboratory.