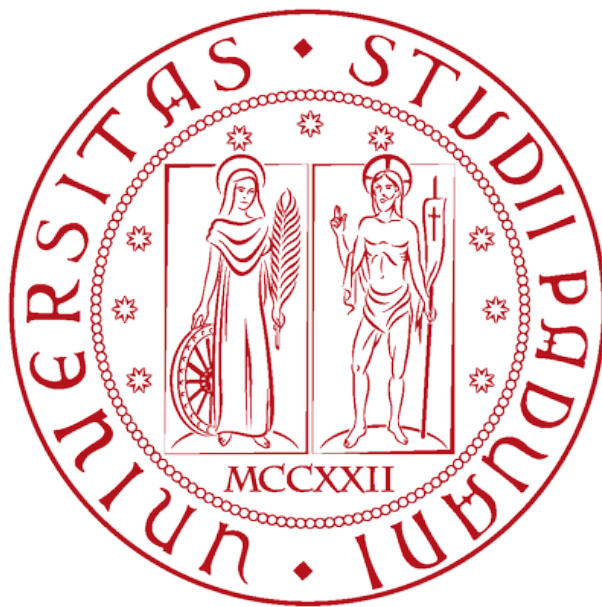


UNIVERSITY OF PADOVA

Embedded Real-Time Control

Laboratory report



Maximillime - (2096189)
Mohammadjavad Rajabi - (2085121)
Pouria Zakariapour - (2072836)
Parsa Majidi - (2080216)

Academic Year 2022-2023

Contents

0	Abstract	2
1	Laboratory 1: I2C	3
1.1	Description	3
1.2	Inter-Integrated Circuit (I2C)	3
1.3	SX1509 I/O Expander	4
1.4	APIs	5
1.5	Exercise 1:	7
1.5.1	Description	7
1.5.2	Code	7
1.5.3	Code Explanation:	7
1.6	Exercise 2:	8
1.6.1	Description:	8
1.6.2	Code:	8
1.6.3	Code Explanation:	9
1.7	Exercise 3:	10
1.7.1	Description:	10
1.7.2	Code:	10
1.7.3	Code Explanation:	10
1.8	Exercise 4:	11
1.8.1	Description:	11
1.8.2	Code:	11
1.8.3	Code Explanation:	11
2	Laboratory 2: Open loop control (camera stabilizer)	12
2.1	IMU	12
2.2	Servo Motor	13
2.3	Exercise1:	14
2.4	Bonus:	15
3	Laboratory 3: Motor Control	17
3.1	Relevant theoretical notions	17
3.2	Another sub	17
3.3	Another sub	17
4	Laboratory 4: Line Following	18
4.1	Relevant theoretical notions	18
4.2	Another sub	18
4.3	Another sub	18
A	Section in the Appendix	19
A.1	Subsection in the Appendix	19

0 Abstract

During this course we worked with a turtlebot that you can see in the Figure 1 The TurtleBot has several buttons, switches, and LEDs connected to the STM32F767 GPIOs



Figure 1: The TurtleBot

1 Laboratory 1: I2C

The purpose of this lab can be categorized like below:

- Inter-Integrated Circuit (I2C)
- SX1509 I/O Expander
- APIs

1.1 Description

This laboratory aims to explore and gain practical experience with some fundamental concepts in Embedded Systems. In particular, we will focus on working with the I2C protocol and the SX1509 module, as well as understanding and utilizing interrupts in the STM32F767 microcontroller. Throughout this laboratory, we will delve into several exercises that will provide us with hands-on experience in applying these concepts.

1.2 Inter-Integrated Circuit (I2C)

The I2C bus is a standard bidirectional interface that uses a controller, known as the master, to communicate with slave devices, Shown in Figure 2.

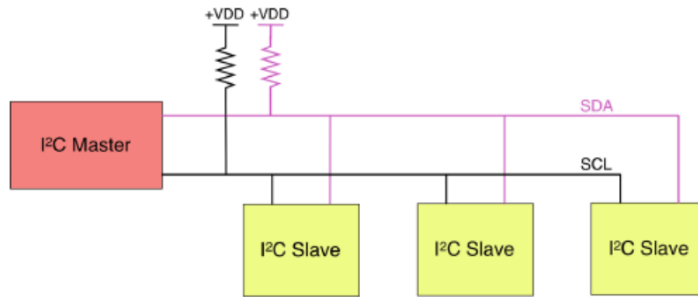


Figure 2: I2C Connection Scheme

A slave may not transmit data unless it has been addressed by the master. Each device on the I2C bus has a specific device address to differentiate between other devices that are on the same I2C bus. Many slave devices will require configuration upon startup to set the behavior of the device. This is typically done when the master accesses the slave's internal register maps, which have unique register addresses. A device can have one or multiple registers where data is stored, written, or read. Many slave devices will require configuration upon startup to set the behavior of the device. The physical I2C interface consists of the serial clock (SCL) and serial data (SDA) lines. Both SDA and SCL lines must be connected to VCC through a pull-up resistor. The size of the pull-up resistor is determined by the amount of capacitance on the I2C lines. Shown in Figure (3a) and Figure (3b)

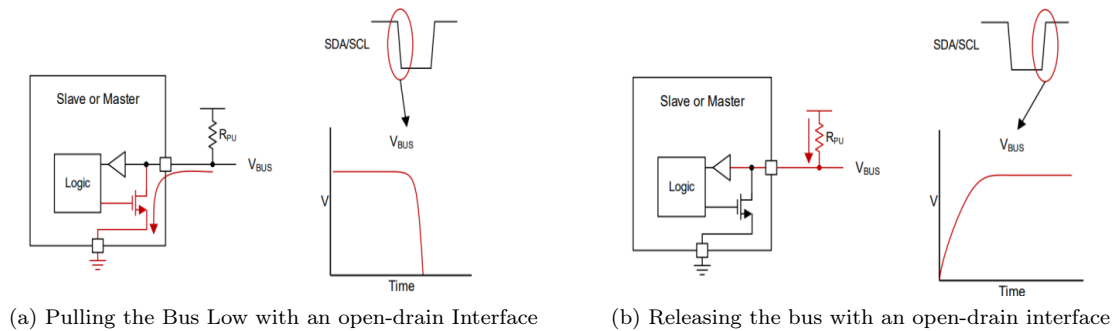


Figure 3

The protocol transmits messages composed by a 9 bit packet. Each message begins with the μC generation of a start signal and ends with a stop signal from the microcontroller, Shown in Figure 4. The start and stop

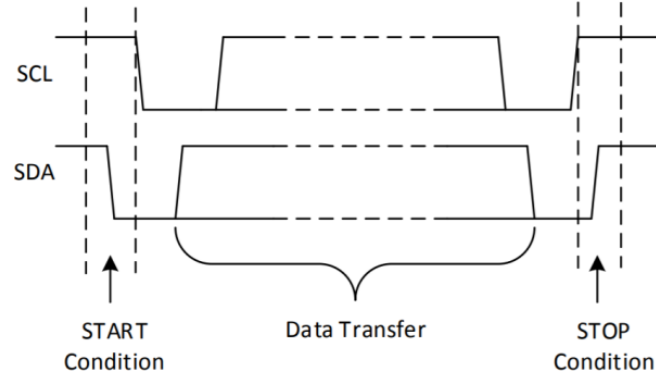


Figure 4: Example of Start and Stop condition

conditions are characterized by data line that moves towards low (start) or high (stop) during the clock high phase. Any other bit is set during the low clock phase; this ensure that the start and stop condition can be discriminated from any other data transmission.

In this communication protocol, each packet consists of 9 bits. The first 8 bits are transmitted by the sending unit, while the 9th bit is set by the receiving unit. The order of the bits within each byte follows a "Most Significant Bit" (MSB) format. The 9th bit serves as an acknowledgment, indicating whether the receiver acknowledges the data. To acknowledge, the receiver sets the data line to a low state at the 9th bit position. Example of Single Byte data transfer in Figure 5.

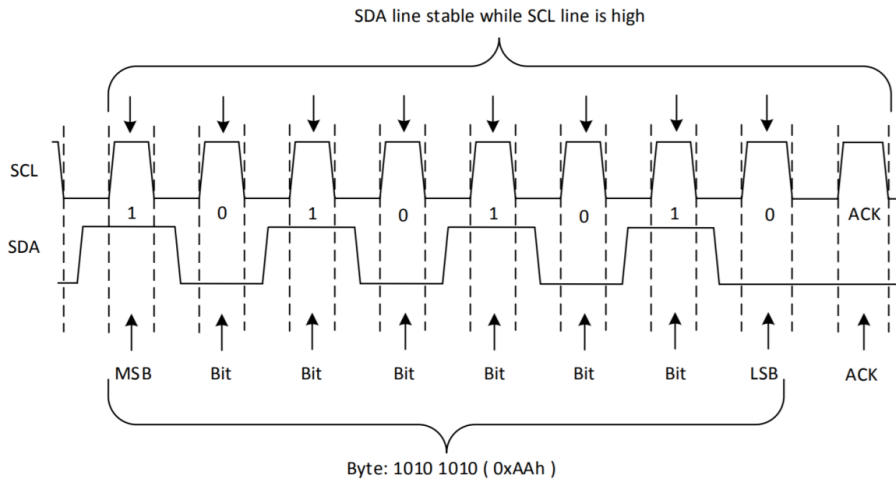


Figure 5: Example of Single Byte Data Transfer

1.3 SX1509 I/O Expander

The SX1509 is complete ultra low voltage 1.2V to 3.6V General Purpose parallel Input/Output (GPIO) expanders ideal for low power handheld battery powered equipment. This family of GPIOs comes in 4-, 8-, 16-channel configuration and allows easy serial expansion of I/O through a standard 400kHz I2C interface. GPIO devices can provide additional control and monitoring when the microcontroller or chipset has insufficient I/O ports, or in systems where serial communication and control from a remote location is advantageous. Keypad application is also supported with the on-chip scanning engine which enables continuous keypad monitoring up to 64 keys without any additional host interaction and further reduce the bus activity. The SX1509 has the ability to generate mask-programmable interrupts based on falling/rising edge of any of its GPIO lines.

In this setup, two SX1509 modules are connected to the microcontroller using the I2C bus. Both devices share the same I2C line, specifically I2C1. To differentiate between the two modules, we assign a specific slave

address to each. The first SX1509, referred to as `sx1509_1`, is assigned the slave address 0x3E, while the second module, known as `sx1509_2`, is associated with the slave address 0x3F. Allocated addresses of slaves implemented in Code 1

Listing 1: I2C Addresses

```
1 #define SX1509_I2C_ADDR1 0x3E           //SX1509 Proxy Sensors I2C address
2 #define SX1509_I2C_ADDR2 0x3F           //SX1509 Keypad I2C address
```

The two lines of Code 2 are addressing of SX1509 registers for Keypad.

Listing 2: Keypad Data registers

```
1 #define REG_KEY_DATA_1 0x27             //RegKeyData1 Key value (column) 1111 1111
2 #define REG_KEY_DATA_2 0x28             //RegKeyData2 Key value (row) 1111 1111
```

This addressing scheme allows the microcontroller to communicate with each SX1509 module individually over the shared I2C bus. Figure 6

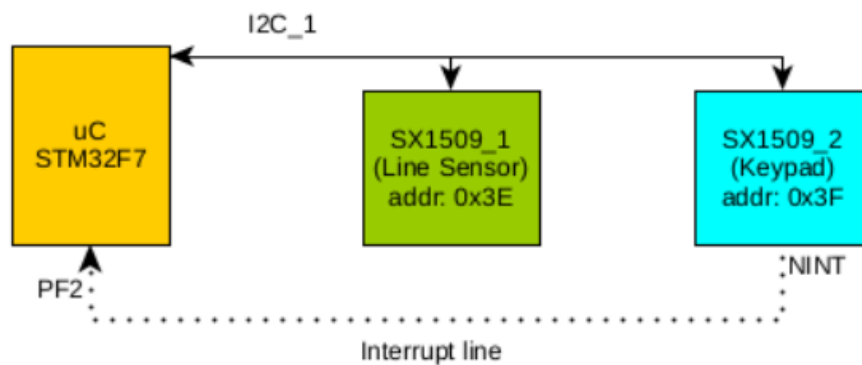


Figure 6: SX1509 connected to Keypad and Line sensor

1.4 APIs

The HAL (Hardware Abstraction Layer) API (Application Programming Interface) is a software layer provided by STM32 microcontrollers that abstracts the underlying hardware functionalities and provides a standardized interface for developers to interact with the microcontroller's peripherals and features. The HAL API serves as a bridge between the application software and the hardware, allowing developers to access and control the microcontroller's peripherals, such as GPIO (General Purpose Input/Output), UART (Universal Asynchronous Receiver-Transmitter), SPI (Serial Peripheral Interface), I2C (Inter-Integrated Circuit), timers, and more. It provides a higher-level programming interface, hiding the low-level hardware details and offering a set of functions that encapsulate complex hardware operations.

Following are useful HAL functions that we used during this Laboratory.

Listing 3: Reading Register values

```
1 HAL_StatusTypeDef HAL_I2C_Mem_Read(I2C_HandleTypeDef *hi2c, uint16_t
2   DevAddress, uint16_t MemAddress, uint16_t MemAddSize, uint8_t *pData,
3   uint16_t Size, uint32_t Timeout);
```

Read an amount of data in blocking mode from a specific memory address

- **hi2c** Pointer to a `I2C_HandleTypeDef` structure that contains the configuration information for the specified I2C.
- **DevAddress** Target device address: The device 7 bits address value in datasheet must be shifted to the left before calling the interface.
- **MemAddress** Internal memory address
- **MemAddSize** Size of internal memory address
- **pData** Pointer to data buffer

- **Size** Amount of data to be sent
- **Timeout** Timeout duration
- **return value** HAL status

Listing 4: Writing Register values

```

1 HAL_StatusTypeDef HAL_I2C_Mem_Write(I2C_HandleTypeDef *hi2c , uint16_t
2   DevAddress , uint16_t MemAddress, uint16_t MemAddSize, uint8_t *pData,
3   uint16_t Size , uint32_t Timeout)

```

Write an amount of data in blocking mode to a specific memory address

- **hi2c** Pointer to a I2C_HandleTypeDef structure that contains the configuration information for the specified I2C.
- **DevAddress** Target device address: The device 7 bits address value in datasheet must be shifted to the left before calling the interface.
- **MemAddress** Internal memory address
- **MemAddSize** Size of internal memory address
- **pData** Pointer to data buffer
- **Size** Amount of data to be sent
- **Timeout** Timeout duration
- **return value** HAL status

1.5 Exercise 1:

1.5.1 Description

In this exercise we are asked to write an ISR that recognize and correctly handles the keypad interrupts. We properly implemented the void HAL_GPIO_EXTI_Callback(uint16_t pin) function, which is used in conjunction with the EXTI (External Interrupt) peripheral to handle interrupts generated by GPIO pins. Then we printed which interrupt has been triggered i.e. the pin. To be able to receive another interrupt from the keypad, we read the registers REG_KEY_DATA_1 and REG_KEY_DATA_2 inside the ISR.

1.5.2 Code

Listing 5: Interrupt Callback

```
1 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin){
2     /*Reading Data From SX1509_I2C_ADDR2 Device Address and REG_KEY_DATA_1 and
3     REG_KEY_DATA_2 Memory Addresses */
4     HAL_StatusTypeDef status;
5     status = HAL_I2C_Mem_Read(
6         &hi2c1,
7         SX1509_I2C_ADDR2 << 1,
8         REG_KEY_DATA_1,
9         1,
10        &colum,
11        1,
12        I2C_TIMEOUT
13    );
14    if (status != HAL_OK) {
15        printf("Error occurred during reading I2C, REG_KEY_DATA_1\n");
16    }
17    status = HAL_I2C_Mem_Read(
18        &hi2c1,
19        SX1509_I2C_ADDR2 << 1,
20        REG_KEY_DATA_2, 1,&row,
21        1,
22        I2C_TIMEOUT
23    );
24    if (status != HAL_OK) {
25        printf("Error occurred during reading I2C, REG_KEY_DATA_2\n");
26    }
27    printf("Interrupt on pin (%d).\n", GPIO_Pin);
28 }
```

1.5.3 Code Explanation:

in this code we have implemented callback interrupt ...

1.6 Exercise 2:

1.6.1 Description:

In this exercise we had to extend the code of exercise 1 to handle the keypad interrupt. we printed which keypad button has been pressed.

1.6.2 Code:

Listing 6: Pressed Keypad Button

```
1  const char keypadLayout[4][4] = {
2      {'*', '0', '#', 'D'},
3      {'7', '8', '9', 'C'},
4      {'4', '5', '6', 'B'},
5      {'1', '2', '3', 'A'}
6  };
7  int column, row;
8  char triggeredChar;
9  int columnDir [4]= { 3, 2 ,1 ,0};
10 int getIndex(int value);
11
12 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin){
13
14     HAL_StatusTypeDef status;
15     status = HAL_I2C_Mem_Read(
16         &hi2c1,
17         SX1509_I2C_ADDR2 << 1,
18         REG_KEY_DATA_1,
19         1,
20         &column,
21         1,
22         I2C_TIMEOUT
23     );
24     if (status != HAL_OK) {
25         printf("Error occurred during reading I2C, REG_KEY_DATA_1\n");
26     }
27     status = HAL_I2C_Mem_Read(
28         &hi2c1,
29         SX1509_I2C_ADDR2 << 1,
30         REG_KEY_DATA_2, 1,&row,
31         1,
32         I2C_TIMEOUT
33     );
34     if (status != HAL_OK) {
35         printf("Error occurred during reading I2C, REG_KEY_DATA_2\n");
36     }
37     printf("Interrupt on pin (%d).\n", GPIO_Pin);
38     printf("column.raw (%d)    row.raw (%d).\n", column, row);
39     column = getIndex (column);
40     row = getIndex(row);
41     printf("column (%d)    row (%d).\n", column, row);
42     triggeredChar = keypadLayout[row][column];
43 } //End of Interrupt Callback function
44 int getIndex(int value){
45     switch (value){
46         case 247:
47             return 3;
48         case 251:
49             return 2;
50         case 253:
51             return 1;
```

```
52     case 254:
53         return 0;
54     default:
55         return 99;
56     }
57 }
```

1.6.3 Code Explanation:

1.7 Exercise 3:

1.7.1 Description:

In this exercise we had to write a routine that reads the status of the line sensor and prints it. The routine must check the status with a polling period of 100ms.

1.7.2 Code:

Listing 7: Reading Line Data

```
1 int findBinary(int decimal){
2     int base = 1;
3     int binary = 0;
4     while(decimal > 0){
5         int rem = decimal % 2;
6         binary = binary + rem*base;
7         decimal = decimal / 2;
8         base = base * 10;
9     }
10    printf("Binary: %d\n\r", binary);
11 }//End of findBinary Function
12 int lineData;
13 while (1){
14     HAL_I2C_Mem_Read(
15         &hi2c1,
16         SX1509_I2C_ADDR1 << 1,
17         REG_DATA_B,
18         1,
19         &lineData,
20         1,
21         I2C_TIMEOUT
22     );
23     findBinary(lineData);
24     printf("Decimal is: %d \n\r", lineData);
25     HAL_Delay(100);
26 }//End of While loop
```

1.7.3 Code Explanation:

sadklasjdllkasjdl

1.8 Exercise 4:

1.8.1 Description:

We made one LED blinking. If you use LEDs connected to PE5 or PE6, check the *.ioc to make sure that those pins are set as GPIO_output. The blinking frequency should be set by the user through the keypad. There's two way to do this: Hard way (Bonus): The frequency can be set "dynamically" by the user. For example, if the user press 125# the LED should blink with a frequency of 125Hz. If the user press 250# the LED should blink with a frequency of 250Hz, and so on.

1.8.2 Code:

Listing 8: C code using listings

```
1 //These are Global Variables//
2 int inputUser = 0;
3 int counter = 0;
4 int flag=0;
5 int freq;
6
7 //This part of code has implemented in the Interrupt Callback Function
8 triggeredChar = keypadLayout[row][column];
9 if(flag == 1){
10     freq = inputUser;
11     inputUser = 0;
12     counter = 0;
13     flag = 0;
14 }
15 if((triggeredChar <= '9') && (triggeredChar >= '0')){
16     keypadFreq = (int)(triggeredChar - '0');
17     inputUser = inputUser*(10^counter) + keypadFreq;
18     counter++;
19 }
20 printf("Triggered Char: %c \n\r ", triggeredChar);
21 //End of Interrupt Callback function
22
23
24 //Start infinit loop
25 while (1){
26     if(triggeredChar == '#'){
27         flag = 1;
28         HAL_GPIO_TogglePin(GPIOE, GPIO_PIN_5);
29         if(inputUser != 0){
30             HAL_Delay((1.0/inputUser)*1000);
31         }
32         else{
33             HAL_Delay(1000);
34         }
35     } //End of if(triggerChar ..)
36     else{
37         HAL_GPIO_TogglePin(GPIOE, GPIO_PIN_5);
38         if(freq != 0){
39             HAL_Delay((1.0/freq)*1000);
40         }
41         else{
42             HAL_Delay(1000);
43         }
44     }
45 } //End of While loop
```

1.8.3 Code Explanation:

2 Laboratory 2: Open loop control (camera stabilizer)

In this lab we are going to implement an open loop stabilizer using the IMU and servo motors.

2.1 IMU

IMU stands for Inertial Measurement Unit. It is a device that is used to measure and report an object's specific force, angular rate, and sometimes the magnetic field surrounding it. IMUs are commonly used in various applications, including robotics, navigation systems, virtual reality, augmented reality, and motion capture.

An IMU typically consists of several sensors that work together to provide information about an object's motion and orientation:

- **Accelerometer:** Measures linear acceleration along different axes (typically three axes: X, Y, and Z). It detects changes in velocity and orientation, allowing the IMU to determine the object's acceleration and tilt. Figure (7)
- **Gyroscope:** Measures angular velocity or rate of rotation around different axes. It provides information about the object's rotational motion and helps track changes in orientation.
- **Magnetometer:** Measures the strength and direction of the magnetic field around the object. It is often included in IMUs to provide additional information for orientation estimation, especially when used in combination with accelerometers and gyroscopes.

By combining the data from these sensors, an IMU can estimate the object's orientation, position, and velocity relative to its initial state.

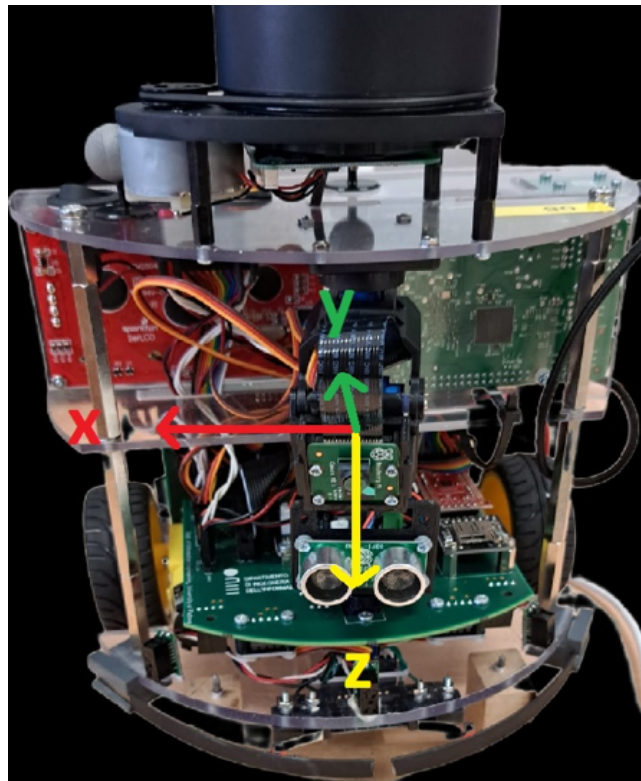


Figure 7: The axis of the accelerometer

- **Tilt** refers to the vertical movement of the camera, where the camera is moved up or down.
- **Pan** refers to the horizontal movement of the camera, where the camera is rotated around its vertical axis to the left or right.

2.2 Servo Motor

A servo motor is a type of motor that is widely used in various applications, particularly in robotics, automation, and control systems. It is designed to provide precise control over angular or rotational movement.

The distinguishing feature of a servo motor is its ability to maintain a specific position or follow a desired trajectory with great accuracy. It achieves this through a closed-loop control system, which continuously compares the actual position of the motor shaft with the desired position and adjusts the motor's output accordingly.

Here are some key components and characteristics of servo motors:

- **DC Motor:** Most servo motors are based on a DC motor as the primary driving mechanism. The DC motor converts electrical energy into rotational motion.
- **Gear Train:** Servo motors often include a gear train that reduces the motor's high-speed, low-torque output to a lower speed with higher torque. This gearing mechanism enables the servo motor to generate more precise and controlled movements.
- **Position Feedback Sensor:** A servo motor typically incorporates a position feedback sensor, such as an encoder or a potentiometer. This sensor provides information about the current position and velocity of the motor shaft. The feedback data is used by the control system to determine if the motor needs to adjust its position.
- **Control Circuitry:** The control circuitry of a servo motor includes a microcontroller or a dedicated servo controller. It receives the control signal or command from an external device, such as a microcontroller or a computer, and generates the appropriate electrical signals to drive the motor.
- **Pulse Width Modulation (PWM) Signal:** Servo motors commonly utilize a PWM signal to control their position and speed. The control signal consists of a series of pulses with varying widths, where the width of each pulse determines the desired position. The control circuitry interprets the pulse width and adjusts the motor's position accordingly. Figure (8)

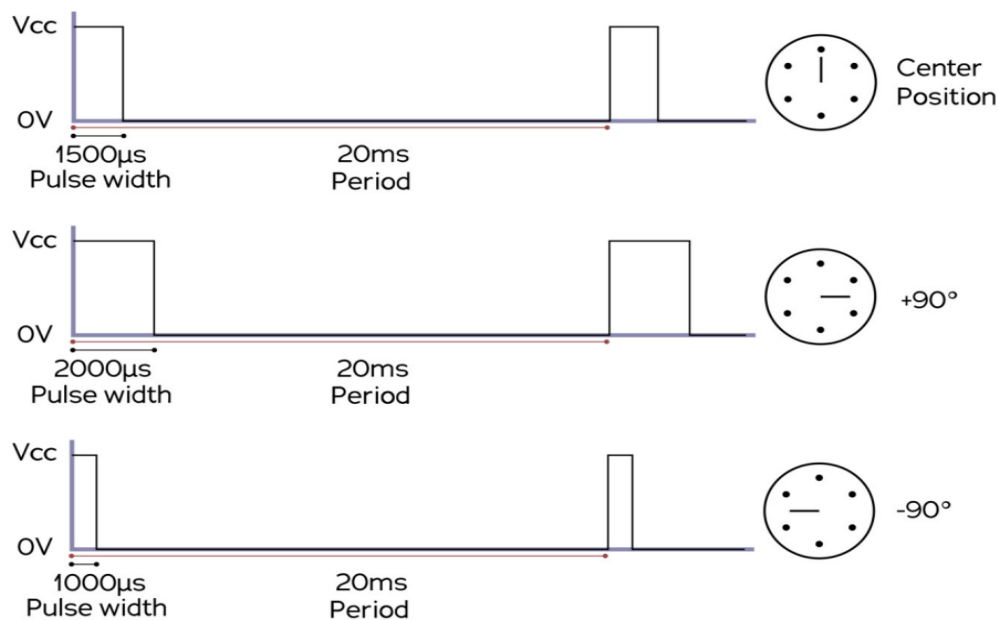


Figure 8: PWM signal and controlling the position of the motor

On the TurtleBot these are two servo that control the tilt and pan of a camera as shown in the figure below.
Figure (9)

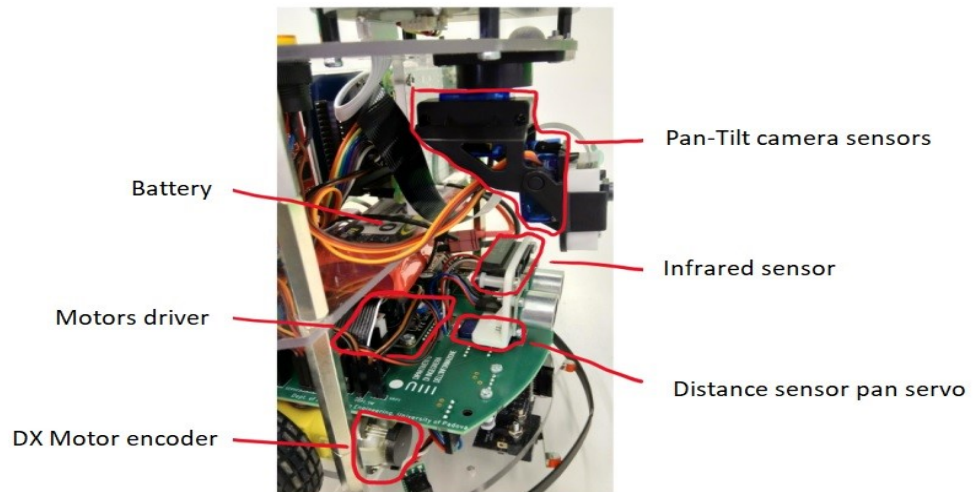


Figure 9: PWM signal and controlling the position of the motor

2.3 Exercise1:

Develop a control law to stabilize the camera's tilt by utilizing data from an IMU.
First of all, we have to keep the camera aligned (0 degree) with the horizon. For this purpose, we should obtain the accelerometer and gyroscope data from the robot.

Listing 9: C code using listings

```

1
2     int8_t tilt = 0;
3     float angle = 0;
4     while (1) {
5
6         HAL_Delay(20);
7         bno055_convert_double_accel_xyz_msq(&d_accel_xyz);
8         angle = (asin(d_accel_xyz.y/ 9.81)) * 180 / 3.14;
9         tilt = -angle;
10        logger_data.u1 = tilt;
11        logger_data.u2 = d_accel_xyz.y;
12        ertc_dlog_send(&logger, &logger_data, sizeof(logger_data));
13        ertc_dlog_update(&logger);
14
15        __HAL_TIM_SET_COMPARE(&htim1,
16            TIM_CHANNEL_3,
17            (uint32_t)saturate((150+tilt*(50.0/5.0)),
18                SERVO_MIN_VALUE,
19                SERVO_MAX_VALUE));
20
21    }
```

As you see, first we defined tilt and angle globally. To obtain the accelerometer data we use “bno055_convert_double_accel_xyz_msq” and “d_accel_xyz” as a pointer to a structure as input.

The formula for finding the tilt of Tbot is: $\theta = \sin^{-1}\left(\frac{a_y}{g}\right)$ which θ is the tilt angle, a_y is the acceleration measured on the y axis and g is gravity acceleration ($=9.81$). To convert θ from radian the equation should be multiplied by $\left(\frac{180}{\pi}\right)$. As we want to put the camera aligned we should put the tilt equal to $(-\theta)$ otherwise the camera turns reverse.

For moving the camera we use “__HAL_TIM_SET_COMPARE” Which is a HAL library macro used for setting the compare value of a specific channel of a hardware timer on a microcontroller.

To record the data, we use datalogger. As all the configuration for using the datalogger is already done, first we define a structure to send the data:

```

1 struct ertc_dlog logger;
2
3 struct datalog {
4     float u1, u2;
5 } logger_data;

```

Then we use logger_data.u1 as the tilt data and logger_data.u2 as the acceleration measured on the y axis data in the while loop.

To plot the data we use Matlab and the serial datalogger is invoked using serial_datalog() function as below:

```

1 data = serial_datalog('COM9',{ '2*single ', '2*single ' }, 'baudrate', 115200)

```

“COM9” is the port of the serial datalogger. “single,single” specifies the data format to be logged. It indicates that two data values will be logged in each iteration of the logging process. In this case, the logged data is expected to be of type single (a single-precision floating-point number).

“baudrate”, “115200” sets the baud rate of the serial communication. The baud rate determines the speed at which data is transferred over the serial port. In this case, the baud rate is set to 115200 bits per second.

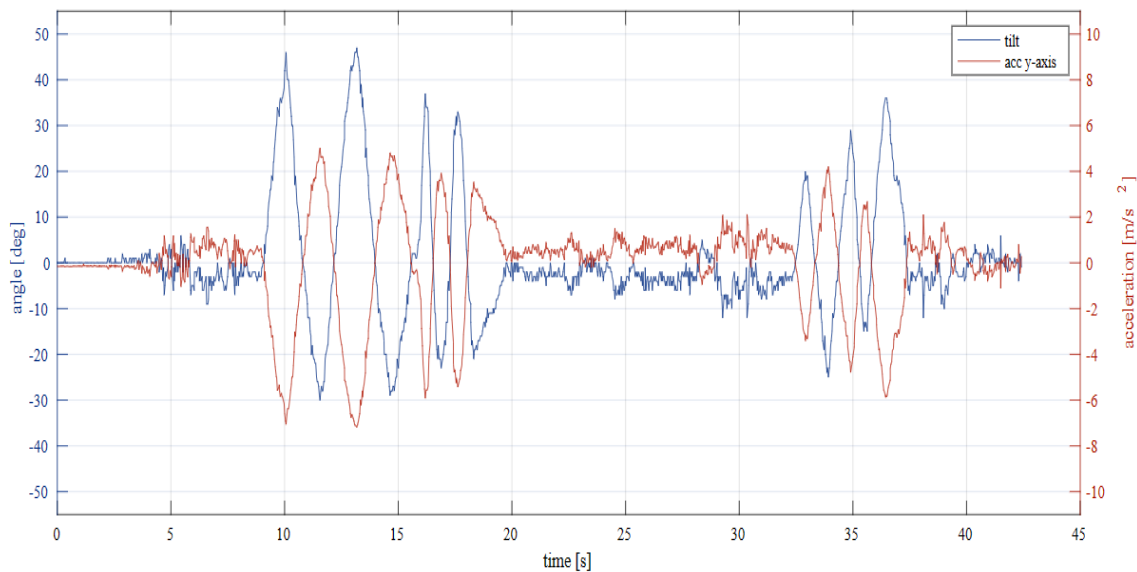


Figure 10: Datalogger Plot

2.4 Bonus:

Implement a control algorithm that implement a “smooth pan” control (i.e. a control system that compensate sharp horizontal rotations of the TBot) using data from the gyroscope.

Process is like exercise 1:

```

1   int8_t pan = 0;
2
3   float angle = 0;
4   while (1) {
5       HAL_Delay(20);
6       bno055_convert_double_gyro_xyz_rps(&d_gyro_xyz);
7       angle = d_gyro_xyz.z * 180 / 3.14;
8       pan = -angle/8;
9       logger_data.u1 = pan;
10      logger_data.u2 = d_gyro_xyz.z;
11      ertc_dlog_send(&logger, &logger_data, sizeof(logger_data));
12      ertc_dlog_update(&logger);
13      __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_3,
14      (uint32_t)saturate((150+pan*(50.0/55.0)),
15      SERVO_MIN_VALUE,
16      SERVO_MAX_VALUE));
17  }

```

As it is clear to obtain the θ we have to put the gyroscope data measured in z axis and we put the pan equal to $(-\frac{\theta}{8})$ which the negative sign is because of turning up and down properly and is divided to 8 which is the proper gain, and inside the “__HAL_TIM_SET_COMPARE()”, as the channel 3 is controlling the pan, we should put “TIM_CHANNEL_3” instead of “TIM_CHANNEL_2” which is controlling the tilt.

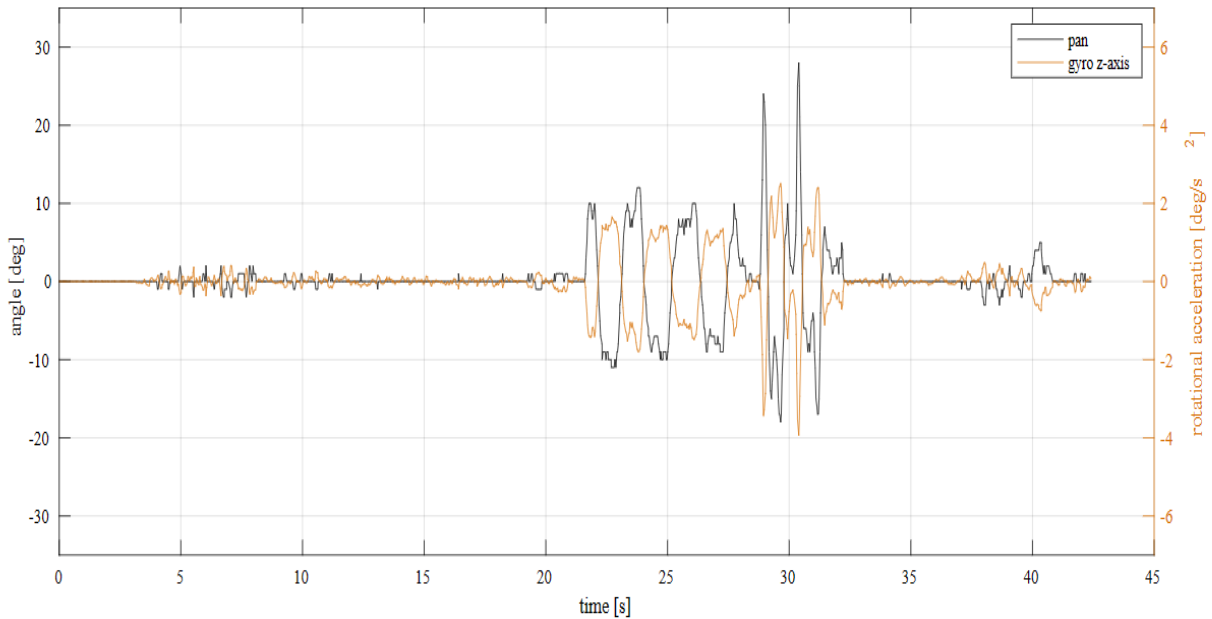


Figure 11: Datalogger Plot2

3 Laboratory 3: Motor Control

3.1 Relevant theoretical notions

3.2 Another sub

3.3 Another sub

4 Laboratory 4: Line Following

4.1 Relevant theoretical notions

4.2 Another sub

4.3 Another sub

A Section in the Appendix

A.1 Subsection in the Appendix

Additional relevant information...