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

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

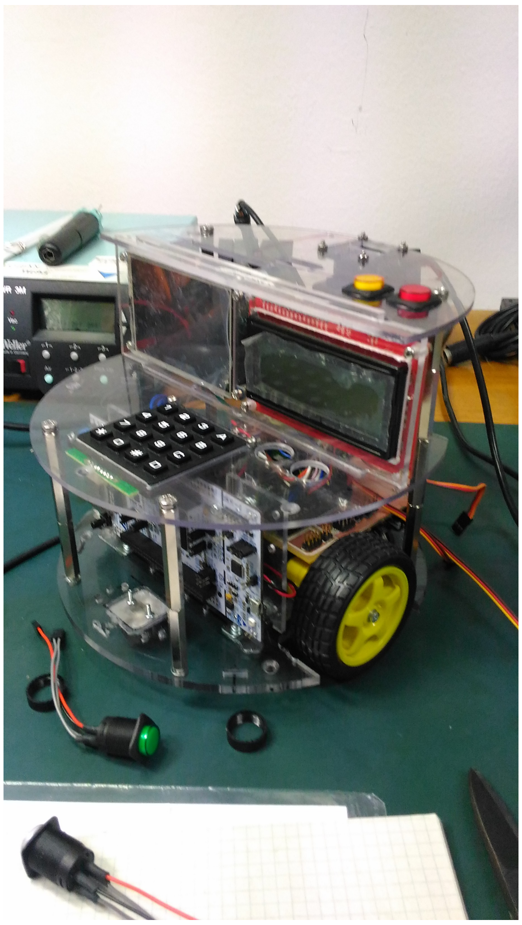


Figure 1: The TurtleBot

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.

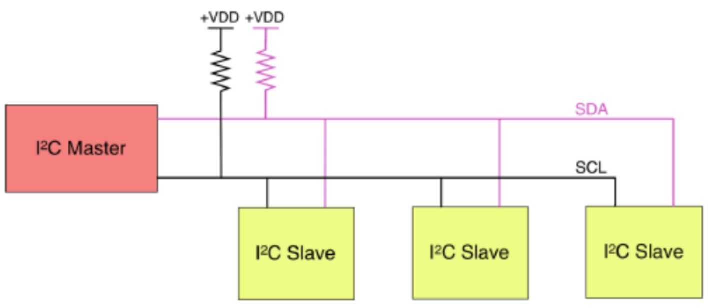
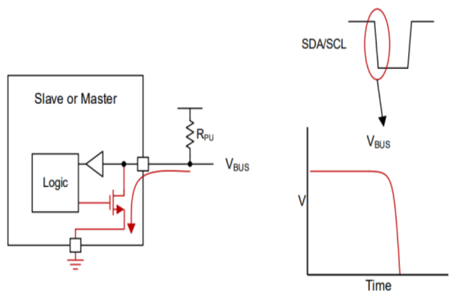
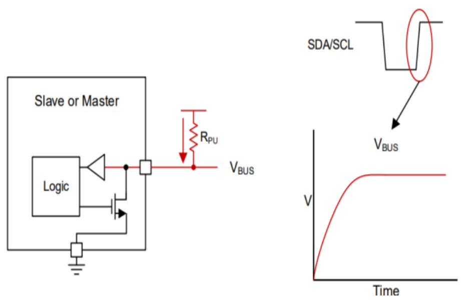


Figure 2: I2C Connection Scheme

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)



(a) Pulling the Bus Low with an open-drain Interface



(b) Releasing the bus with an open-drain interface

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

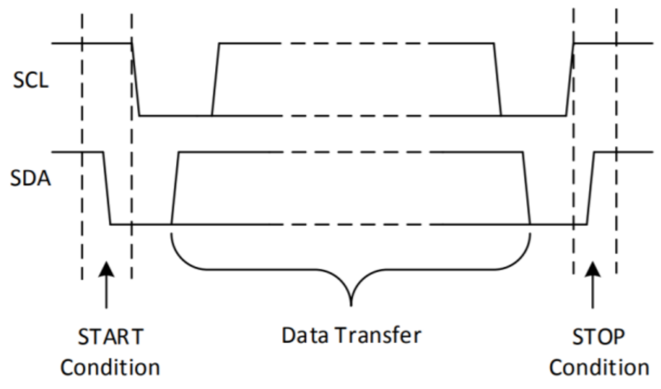


Figure 4: Example of Start and Stop condition

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.

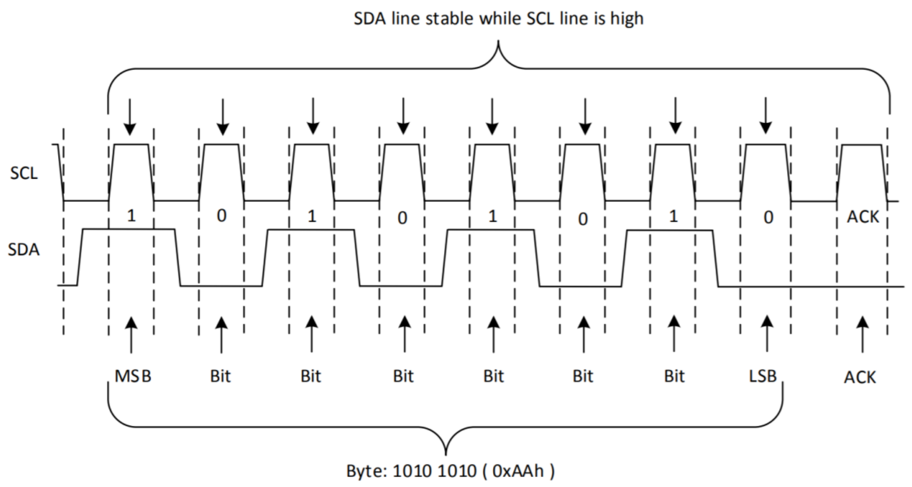

SDA line stable while SCL line is high
SCL
SDA
1
0
1
0
1
0
1
0
ACK
MSB
Bit
Bit
Bit
Bit
Bit
Bit
LSB
ACK
Byte: 1010 1010 (0xAAh)

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

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

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

Listing 2: Keypad Data registers

```
#define REG_KEY_DATA_1  0x27         //RegKeyData1 Key value (column) 1111 1111
#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

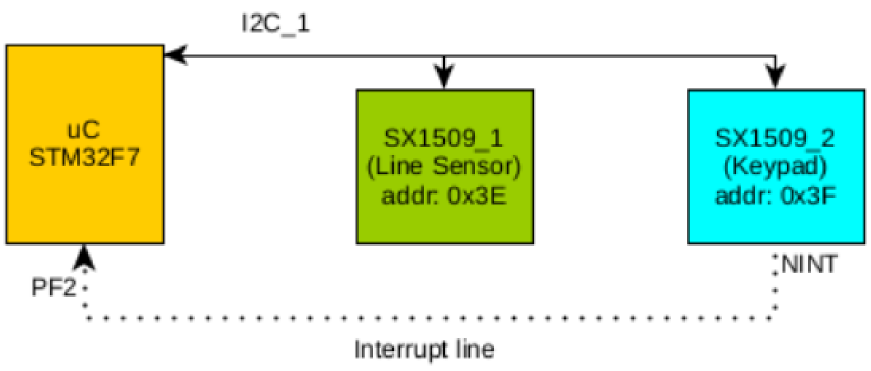


Figure 6: SX1509 connected to Keypad and Line sensor

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

```
HAL_StatusTypeDef HAL_I2C_Mem_Read(I2C_HandleTypeDef *hi2c, uint16_t
    DevAddress, uint16_t MemAddress, uint16_t MemAddSize, uint8_t *pData,
    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

```
HAL_StatusTypeDef HAL_I2C_Mem_Write(I2C_HandleTypeDef *hi2c, uint16_t
    DevAddress, uint16_t MemAddress, uint16_t MemAddSize, uint8_t *pData,
    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

```c
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin){
    /*Reading Data From SX1509_I2C_ADDR2 Device Address and REG_KEY_DATA_1 and
    REG_KEY_DATA_2 Memory Addresses */
    HAL_StatusTypeDef status;
    status = HAL_I2C_Mem_Read(
        &hi2c1,
        SX1509_I2C_ADDR2 << 1,
        REG_KEY_DATA_1,
        1,
        &colum,
        1,
        I2C_TIMEOUT
    );
    if (status != HAL_OK) {
        printf("Error occurred during reading I2C, REG_KEY_DATA_1\n");
    }
    status = HAL_I2C_Mem_Read(
        &hi2c1,
        SX1509_I2C_ADDR2 << 1,
        REG_KEY_DATA_2, 1,&row,
        1,
        I2C_TIMEOUT
    );
    if (status != HAL_OK) {
        printf("Error occurred during reading I2C, REG_KEY_DATA_2\n");
    }
    printf("Interrupt on pin (%d).\n", GPIO_Pin);
}
```

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

```c
const char keypadLayout[4][4] = {
    {'*', '0', '#', 'D'},
    {'7', '8', '9', 'C'},
    {'4', '5', '6', 'B'},
    {'1', '2', '3', 'A'}
};
int colum, row;
char triggeredChar;
int columDir [4]= { 3, 2 ,1 ,0};
int getIndex(int value);

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin){

    HAL_StatusTypeDef status;
    status = HAL_I2C_Mem_Read(
        &hi2c1,
        SX1509_I2C_ADDR2 << 1,
        REG_KEY_DATA_1,
        1,
        &colum,
        1,
        I2C_TIMEOUT
    );
    if (status != HAL_OK) {
        printf("Error occurred during reading I2C, REG_KEY_DATA_1\n");
    }
    status = HAL_I2C_Mem_Read(
        &hi2c1,
        SX1509_I2C_ADDR2 << 1,
        REG_KEY_DATA_2, 1,&row,
        1,
        I2C_TIMEOUT
    );
    if (status != HAL_OK) {
        printf("Error occurred during reading I2C, REG_KEY_DATA_2\n");
    }
    printf("Interrupt on pin (%d).\n", GPIO_Pin);
    printf("colum.raw (%d)    row.raw (%d).\n", colum, row);
    colum = getIndex (colum);
    row = getIndex(row);
    printf("colum (%d)    row (%d).\n", colum, row);
    triggeredChar = keypadLayout[row][colum];
} //End of Interrupt Callback function
int getIndex(int value){
  switch (value){
    case 247:
      return 3;
    case 251:
      return 2;
    case 253:
      return 1;
    case 254:
      return 0;
    default:
      return 99;
  }
}
```

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

```
int findBinary(int decimal){
     int base = 1;
     int binary = 0;
    while(decimal > 0){
        int rem = decimal % 2;
        binary = binary + rem*base;
        decimal = decimal / 2;
        base = base * 10;
    }
    printf("Binary: %d\n\r", binary);
}//End of findBinary Function
int lineData;
while (1){
     HAL_I2C_Mem_Read(
          &hi2c1,
          SX1509_I2C_ADDR1 << 1,
          REG_DATA_B,
          1,
          &lineData,
          1,
          I2C_TIMEOUT
          );
       findBinary(lineData);
       printf("Decimal is: %d \n\r", lineData);
       HAL_Delay(100);
}//End of While loop
```

1.7.3 Code Explanation:

sadklasjdlkasjdl

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

```c
//These are Global Variables//
int inputUser = 0;
int counter = 0;
int flag=0;
int freq;

//This part of code has implemented in the Interrupt Callback Function
triggeredChar = keypadLayout[row][colum];
if(flag == 1){
     freq = inputUser;
     inputUser = 0;
     counter = 0;
     flag = 0;
}
if((triggeredChar <= '9') && (triggeredChar >= '0')){
     keypadFreq = (int)(triggeredChar - '0');
     inputUser = inputUser*(10^counter) + keypadFreq;
     counter++;
}
  printf("Triggered Char: %c \n\r ", triggeredChar);
//End of Interrupt Callback function


//Start infinit loop
while (1){
     if(triggeredChar == '#'){
     flag = 1;
     HAL_GPIO_TogglePin(GPIOE, GPIO_PIN_5);
         if(inputUser != 0){
          HAL_Delay((1.0/inputUser)*1000);
         }
         else{
          HAL_Delay(1000);
         }
     }//End of if(triggerChar ..)
     else{
          HAL_GPIO_TogglePin(GPIOE, GPIO_PIN_5);
          if(freq != 0){
          HAL_Delay((1.0/freq)*1000);
          }
          else{
               HAL_Delay(1000);
          }
     }
}//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.

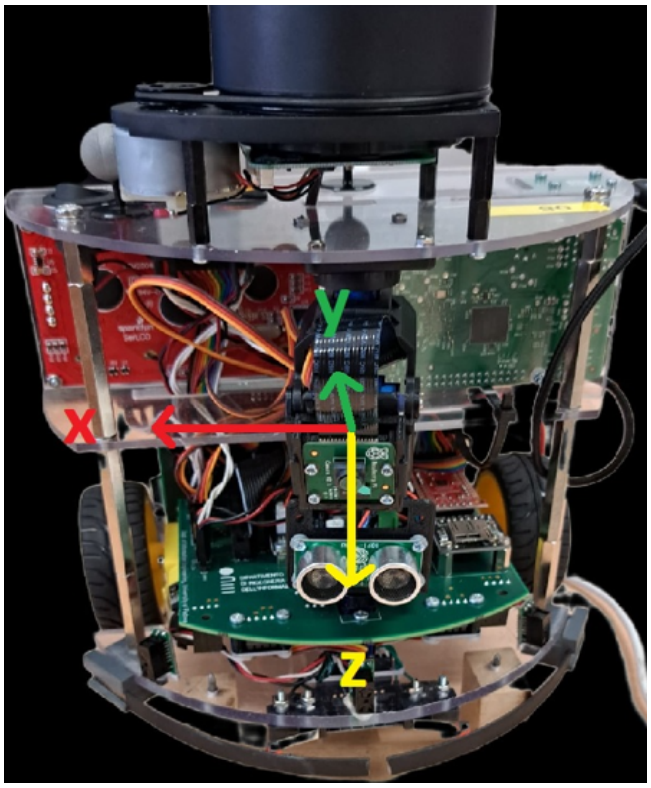


Figure 7: The axis of the accelerometer

- **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)

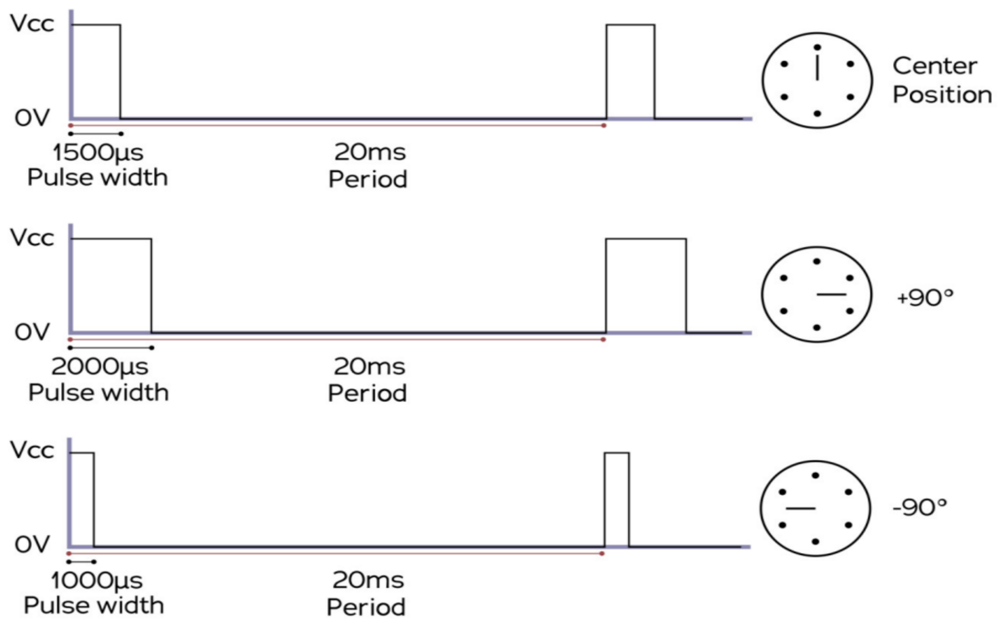


Figure 8: PWM signal and comtroling the position of the motor

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

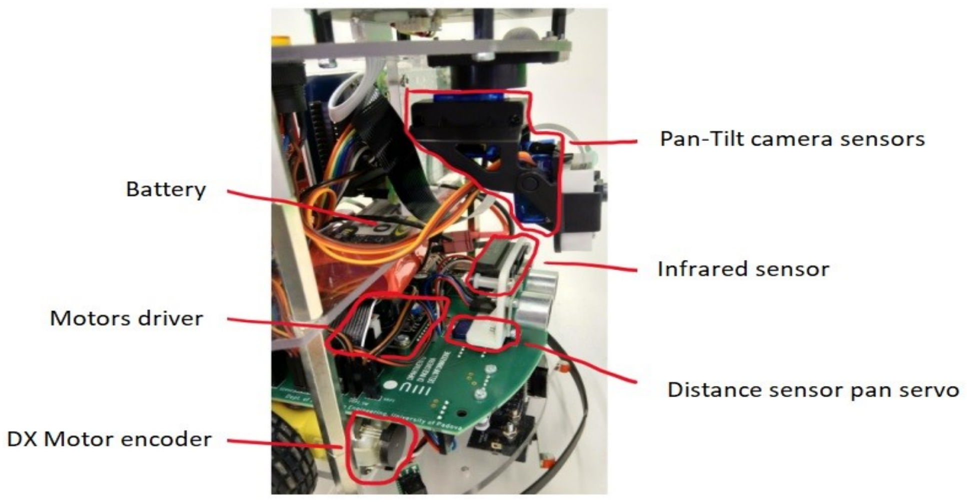


Figure 9: PWM signal and comtroling the position of the motor

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

```

    int8_t tilt = 0;
    float angle = 0;
while (1) {

    HAL_Delay(20);
    bno055_convert_double_accel_xyz_msq(&d_accel_xyz);
    angle = (asin(d_accel_xyz.y/ 9.81)) * 180 / 3.14;
    tilt = -angle;
    logger_data.u1 = tilt;
    logger_data.u2 = d_accel_xyz.y;
    ertc_dlog_send(&logger, &logger_data, sizeof(logger_data));
    ertc_dlog_update(&logger);

    __HAL_TIM_SET_COMPARE(&htim1,
        TIM_CHANNEL_3,
        (uint32_t)saturate((150+tilt*(50.0/5.0)),
         SERVO_MIN_VALUE,
         SERVO_MAX_VALUE));

  }
```

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 radiant the equation should be multiplied by $(\frac{180}{\pi})$ As we want to put the camera aligned we should put the tilt equal to (-θ) 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:

```
struct ertc_dlog logger;

struct datalog {
    float u1, u2;
} 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:

```
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.

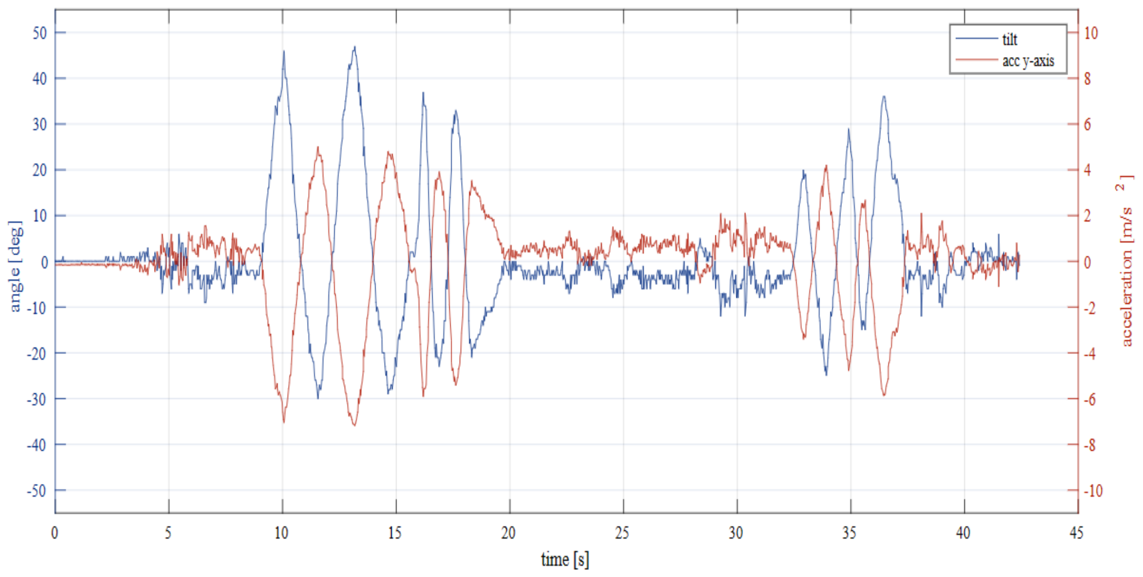


Figure 10: Datalogger Plot

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:

```
    int8_t pan = 0;

    float angle = 0;
while (1) {
    HAL_Delay(20);
    bno055_convert_double_gyro_xyz_rps(&d_gyro_xyz);
    angle = d_gyro_xyz.z * 180 / 3.14;
    pan = -angle/8;
    logger_data.u1 = pan;
    logger_data.u2 = d_gyro_xyz.z;
    ertc_dlog_send(&logger, &logger_data, sizeof(logger_data));
    ertc_dlog_update(&logger);
    __HAL_TIM_SET_COMPARE(&htim1,TIM_CHANNEL_3,
    (uint32_t)saturate((150+pan*(50.0/55.0)),
     SERVO_MIN_VALUE,
    SERVO_MAX_VALUE));
}
```

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.

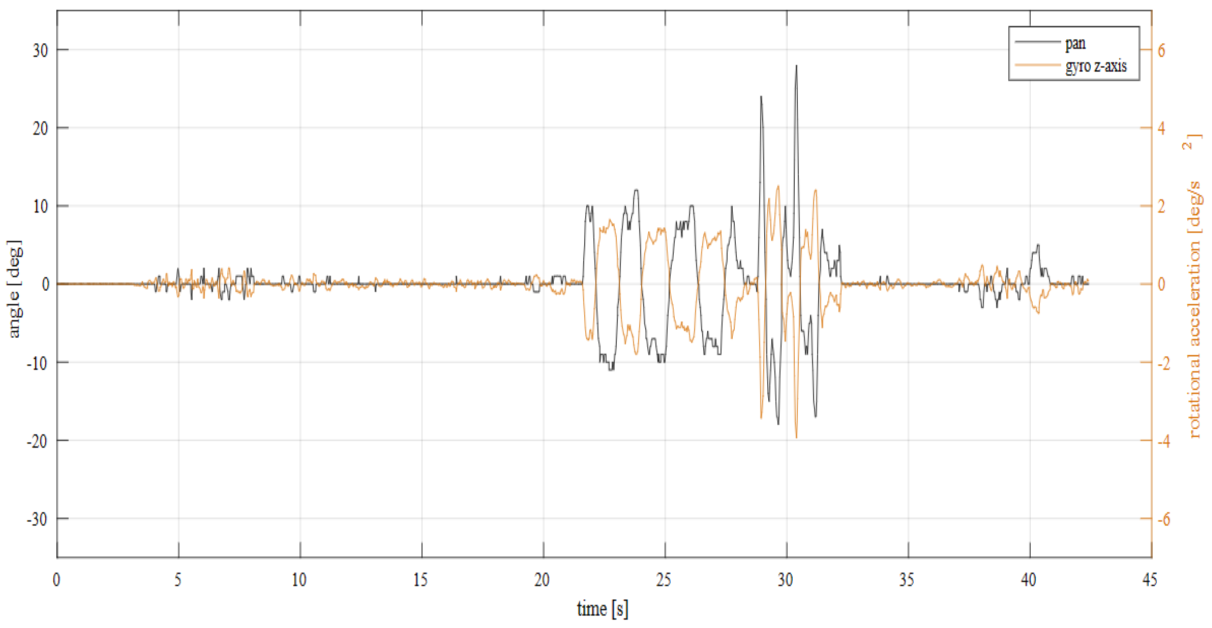


Figure 11: Datalogger Plot2

3 Laboratory 3: Motor Control

3.1 Relevant theoretical notions

3.2 Another sub

3.3 Another sub

4 Laboratory 4: Line Following

The goal of this laboratory was to combine the line sensor from Lab 1 and the motor controller from Lab 3 to make the Turtlebot follow a given line autonomously.

- **The infrared line sensor:**

We are using a Pololu QTR reflectance sensor which can detect how much infrared light has been reflected from a surface. With a certain threshold we can than determine if a surface is dark or light. The installed sensor is located on the lower front of the robot. It is about 10 cm wide and consists of an array of 8 photo diodes with each one photo transistor. The distance between every sensor is 8 mm. This gives us a total of 8 individual sensors that can differentiate between dark and light areas. This data is retrieved by the controller MCU through a port expander board using a I2C connection. The sensor data encoded in a binary format and received as an integer. To retrieve the original data, it is necessary to convert the integer back to a binary number. This binary number has 8 bits, each storing the value of one sensor.

- **Properties of the line:**

The line is assumed to be wide enough to trigger at least one of the infrared sensors and at most two at the same time. The goal is to create a controller algorithm that keeps the line between the two middle sensors 3 and 4. We are implementing a tracking error and set this position to tracking error 0. We then define the tracking error to be greater than zero when the line is on the left side of the sensor. If the tracking error is less than zero, the line is on the right side of the sensor. Figure 12

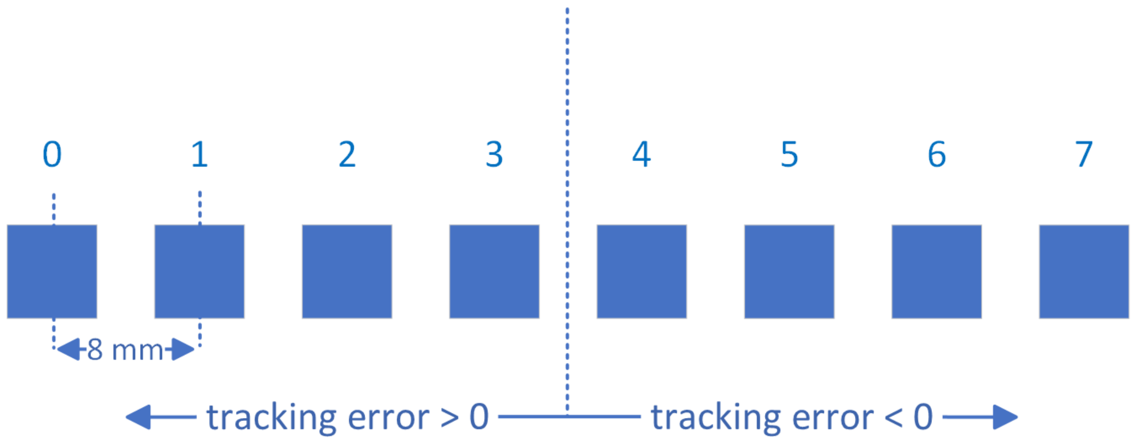


Figure 12: Spacing of the infrared sensors

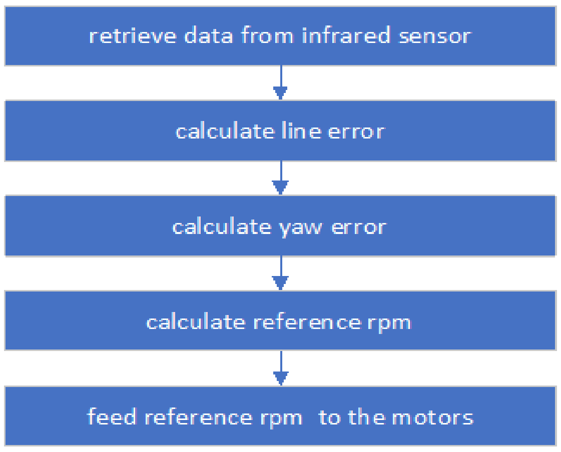

retrieve data from infrared sensor
calculate line error
calculate yaw error
calculate reference rpm
feed reference rpm to the motors

Figure 13: Main structure of the line following code

In LAB1 we already created a function to get the binary values of the integer we received by the line sensor. However, the function would give us an integer again with all the bits in a single number. This was not a very useful implementation. We needed to get the bits in an array. We define an empty array in the callback function (Code 11) and give the pointer to the findBinary function. The basic decimal to binary converter in form of a while function would then store every bit in this array, Code 10.

Listing 10: converts integer to a binary array

```
void findBinary(int decimal, int * binary){
    int i =0;
    while(decimal > 0){
        int rem = decimal % 2;
        binary[i] = rem;
        i++;
        decimal = decimal / 2;
       }
}
```

Listing 11: Top part of the timer callback function

```
    void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    /* Speed ctrl routine */
    if(htim->Instance == TIM6) // check if interrupt came from timer
    {
        // get the line sensor data
        status = HAL_I2C_Mem_Read(&hi2c1,
                            SX1509_I2C_ADDR1 << 1,
                            REG_DATA_B, 1,
                            &lineData, 1,
                            I2C_TIMEOUT);
    int binary[8] = {0};
    findBinary(lineData, binary); // converts int to binary array
```

The calc_error_line function shown in Code 12 does the calculation of the line error. The input of this function is an 8-dimensional array, containing the state of each infrared sensor coded in binary. A logical one means, that the corresponding sensor has detected a dark spot. In the for loop we iterate through every value in the binary[] array, ergo through all sensors data. We calculate the sum of the binary[] array to get the total number of triggered sensors. Furthermore, we are filling the distance_from_middle array with the opposing distance of every sensor from the middle. The distance between every sensor is 4mm. These values get negative to indicate a distance to the left and positive to indicate a distance to the right. The sum_dist value is the sum of every distance from the center if the sensor was activated. Calculating the line_error by dividing the sum of the distance of each activated sensor with the total number of activated sensors. This line_error ten represents the offset of the black line from the center, regardless of how wide the line is. Even if only one sensor is not activated, we still get a useful line error value.

Listing 12: Function to calculate the line error based on a binary array

```
int calc_error_line (int binary[]){
    float distance_from_middle[8]={0};
    float sum_dist = 0;
    int sum_binary = 0;
    for(int n=0;n<8;n++){
        sum_binary += binary[n];
        distance_from_middle[n]=((7.0/2.0)-n)*4;
        sum_dist += binary[n]*distance_from_middle[n];
    }
    float line_error = sum_dist / sum_binary;
    return line_error;
}
```

Listing 13: Conversion from line error to yaw error

```
float calc_yaw_error(float line_error){
    float phi_err = line_error/85;
    float yaw_err = phi_err * (165/2);
    return yaw_err;
}
```

The calc_yaw_error function in Code 13 calculates a much stabler result for our error value. It includes the wheelbase dimension (D) and the distance between the line sensor and wheels (H). This determines the position of the turning point of the bot, which is between the two wheels, shown in Figure (14)

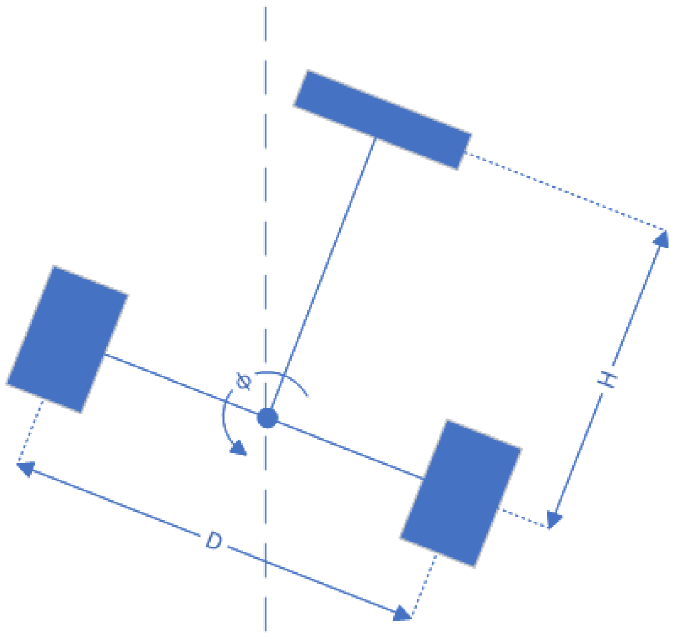


Figure 14: Schematic of the wheelbase and line sensor placement
D=165 mm, H=85 mm

- **Calculating the reference for the motors**:

Based on the yaw error, we can now calculate our reference for the two motors. Since our motor controller is designed to compute rotations per minute, we set a base speed of 100 rpm and depending on the direction of the motor add or subtract our yaw error. To give the yaw error more effect on the base speed, we incorporated an additional gain. Testing different gains, we first found a relay able solution with a base speed of 100 rpm +- yaw_error * 12. By Increasing the gain, we got better performance in sharper corners. The drawback was increased instability on the straight segments. Small changes in the yaw error multiplied with a high gain gave the bot a very shaky ride in straight segments. To furthermore improve the performance a linear or even an exponential gain was necessary. This would increase the impact of the yaw error in tighter corners and make sure to decrease the gain when the line is going straight. The robot would be very stable on straight lines and keep its agile performance in sharp corners. We tried implementing different if statements to check if the yaw error was in a certain range. Then we could set a different gain accordingly. This worked but needed a lot of tuning of all the values. In the end we saw that the values where basically the same as the yaw error itself. A very simple solution was to square the yaw error. This would increase the gain linearly. To keep the sign of the yaw error, the absolute value of the yaw error was multiplied by itself, see Code 14. For tuning this setup to use different base speeds it is possible to also add a gain to increase the impact of the squared yaw error.

Listing 14: Calculating the reference signals for the motors

```
reference_rpm_L = 100 - yaw_err*ABS(yaw_err); // workes great
reference_rpm_R = 100 + yaw_err*ABS(yaw_err);
```

A Section in the Appendix

A.1 Subsection in the Appendix

Additional relevant information...