SAPIENZA
UNIVERSITÀ DI ROMA

Differential Drive and Inertial Navigation Robot

Faculty of Information Engineering, Informatics and Statistics

Degree Program in Computer and Control Engineering

Candidate

Filippo Graziano
ID number 1761694

Thesis Advisor

Prof. Giorgio Grisetti

Academic Year 2018/2019

Differential Drive and Inertial Navigation Robot
Bachelor's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: graziano.1761694@studenti.uniroma1.it

Abstract

In this thesis we will present the robot we have built in the last months. This robot implements two different localization algorithms:

- on one side, it uses two encoders to measure the wheel rotations and estimate position and orientation using differential drive;
- on the other side, it uses an IMU[1] to estimate the position using inertial navigation.

The thesis is structured as follows:

- Chapter 1 introduces the problem of localization and its importance for autonomous robots.
- Chapter 2 describes the sensors used by the robot and how they work.
- Chapter 3 includes a broad description of the robot, explaining the specific hardware used, the communication protocols needed, the firmware and how it interfaces with the sensors, and finally the host (i.e. the PC[2] interfacing with the firmware).
- Chapter 4 explains the two localization algorithms in detail.
- Chapter 5 provides some final considerations and hints possible improvements in the localization algorithms adopted for our robot.
- Some implementation details are provided in the Appendices.

The code for both the firmware running on the robot and the host is available at https://github.com/FilippoGraziano98/Acquisition_Platform.

[1] IMU stands for Inertial Measurement Unit
[2] PC stands for Portable Computer

Acknowledgments

First, I would like to thank my parents, my sisters, my girlfriend and my closest friends for providing me with unfailing support and continuous encouragement throughout my years of study. This accomplishment would not have been possible without them.

I would also like to acknowledge my fellow students for making tough lessons less tiring and for stimulating me whenever my laziness was almost going to win me.

Finally, I express my sincere gratitude to my advisor Prof. Grisetti for his patience, guidance and insightful comments.

Contents

Chapter 1

Introduction

In a world where automation and artificial intelligence are getting everyday more important, one of the fundamental problems for an autonomous robot is to be able to "understand" the environment from the measurements of its sensors. An autonomously navigating robot should know where it stands in the environment and what the environment looks like.

In this thesis we focus on the problem of localization. Localization[1] is the process of determining where a robot is located with respect to its environment, using the measurements up to the current instant. Localization is a fundamental skill for an autonomous robot as the knowledge of the robot's own location is an essential prerequisite to making decisions about future actions. Therefore localization is a starting point for mapping and path planning.
Usually a map of the environment is available and the robot is equipped with sensors that observe the environment as well as monitor its own motion in order to be able to estimate its position in the map. In our scenario the robot does not have knowledge of a map of the environment and it is only equipped with sensors measuring its motion; our goal is to be able to correctly estimate the robot's position in relation to its starting point.

Chapter 2

Sensors

A sensor is a device whose purpose is to detect events or changes in the environment and send the information to a computer processor. The processor will then use the sensor observations to perform more complex tasks.
In our case, we are using position and movement sensors in order to be able to localize the robot and keep track of its movements with respect to its starting position.

Nowadays MEMS[1] sensors are widespread. MEMS is a technology which allows to miniaturize mechanical and electro-mechanical elements; therefore it enables to create very small sensors which can be integrated in circuits.

2.1 Rotary Encoders

A rotary encoder is a position sensor used for determining the angular position of a rotating shaft, it converts the angular position or motion of the shaft to analog or digital output signals.
There are two main types of rotary encoders, differing in the output signal: absolute and incremental.

- Absolute encoders give in output a signal indicating the current shaft position;
- while incremental encoders provide information about the motion of the shaft, which is then processed to calculate the current shaft position.

In the following we will focus on incremental encoders as such are the ones used in the robot in consideration.

[1]MEMS stands for "MicroElectroMechanical System"

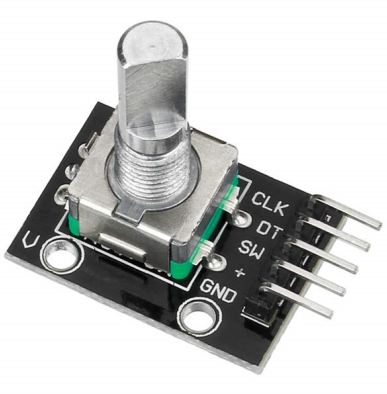


Figure 2.1. Incremental Rotary Encoder

2.1.1 Incremental Rotary Encoders

As previously mentioned, an incremental encoder measure changes in position, but it does not keep track of absolute position.
It is also known as Quadrature Encoder since its A and B output signals are two square waves in quadrature.

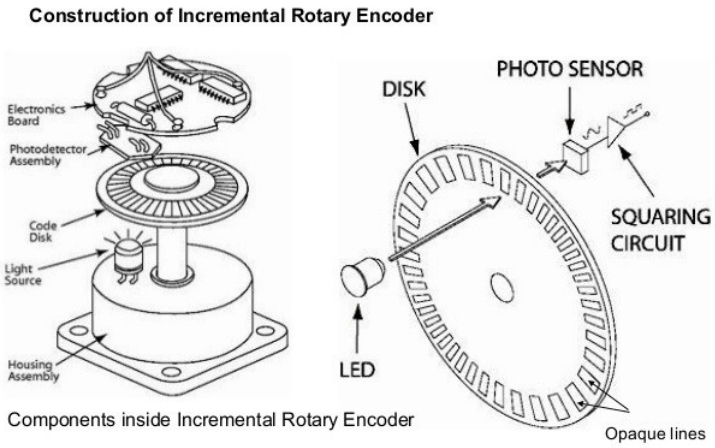


Figure 2.2. Encoder internal structure.

The encoder is composed of a disk with evenly spaced opaque lines. On one side of the disk there is a light source, while on the other side there are two photodetectors, which are generating the signals A and B. The disk rotates together with the shaft, while it rotates the opaque lines will pass between the light source and the photodetectors generating the square waves.

The frequency of the pulses in the A and B output is directly proportional to the shaft rotational velocity; higher frequencies indicate rapid movement, whereas lower frequencies indicate slower speeds. Static, unchanging signals A and B indicate the

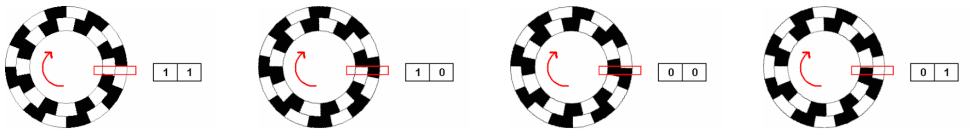


Figure 2.3. Generation of the square wave otuput pulses for clockwise rotations.

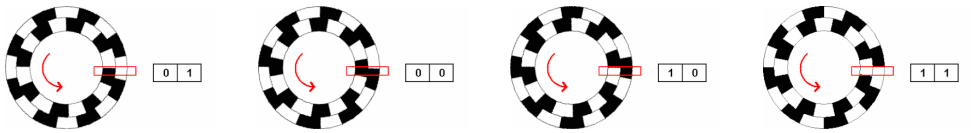


Figure 2.4. Generation of the square wave otuput pulses for anti-clockwise rotations.

encoder is motionless.

2.2 Inertial Measurement Unit

There are mainly two categories of IMUs: stable platform systems and strapdown systems. The main difference between the two is the frame of reference in which the sensor operates.

- In stable platform systems the inertial sensors are mounted on a platform isolated from any external motion; therefore the sensor always keep its alignment with the global frame.
- Strapdown systems instead are mounted rigidly on the platform, therefore the sensor is always aligned with the body frame (i.e. the navigation system's frame of reference).

In the following, we will consider strapdown systems as such is the one used in the robot we are considering.
Since the output is relative to the body frame, to keep track of the orientation gyroscope measurements must be integrated, and to keep track of the position accelerometer measurements are projected onto the global axis (using the orientation determined through the gyroscope) and then they are integrated.[2]

2.2.1 Gyroscope

Gyroscopes are devices that measure rotational motion. While conventional gyroscopes measure orientation, most modern gyroscopes are rate-gyros measuring angular velocity.

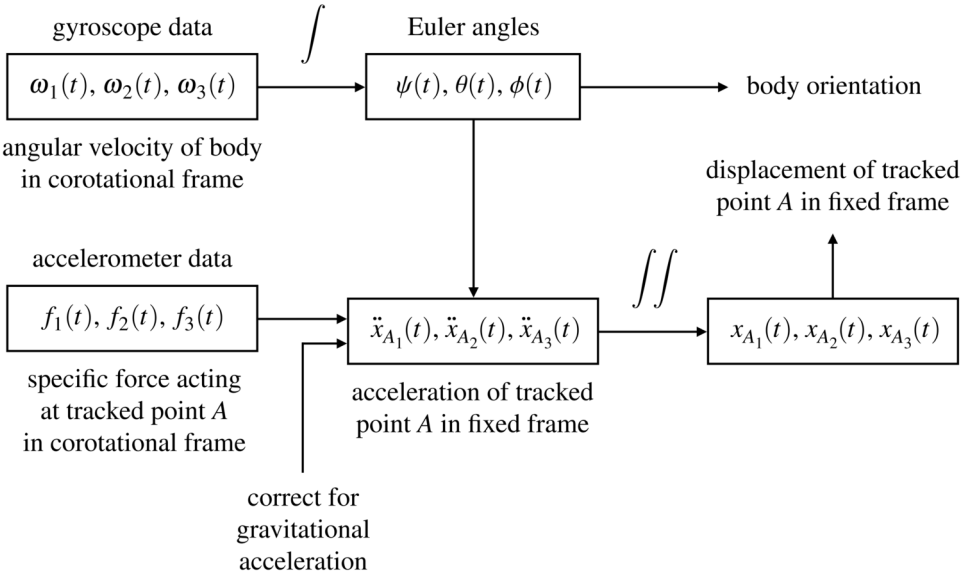


Figure 2.5. Strapdown Inertial Navigation Algorithm.

Mechanical Gyroscope

A mechanical gyroscope consists of a spinning wheel mounted on two gimbals which allow it to rotate in all three axis. As an effect of the conservation of angular momentum, the spinning wheel will resist orientation changes. Hence when a gyroscope is subject to rotation, the wheel will keep rotating around the same rotational axis, while the angles between adjacent gimbals will change. Therefore to measure the orientation we have to measure angles between adjacent gimbals.

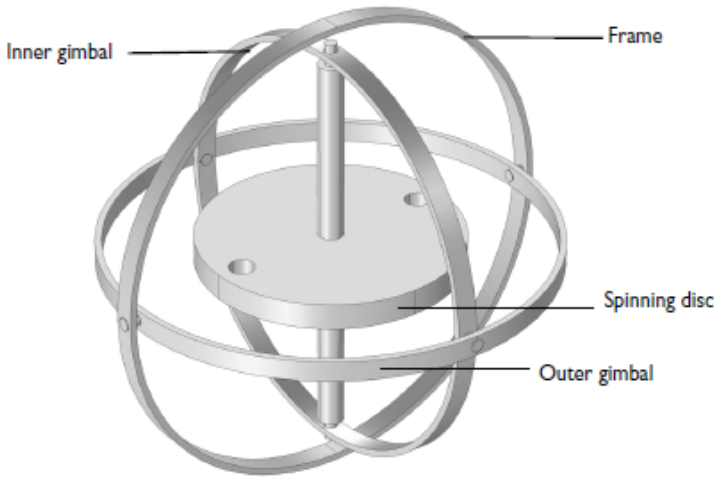


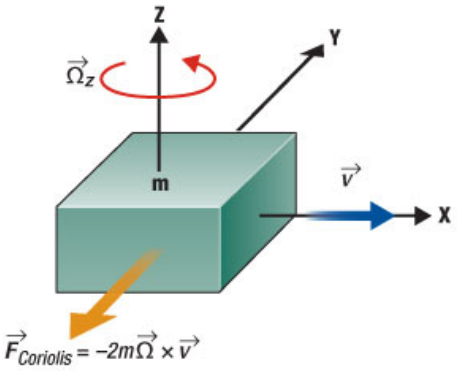
Figure 2.6. Mechanical Gyroscope.

MEMS Gyroscope

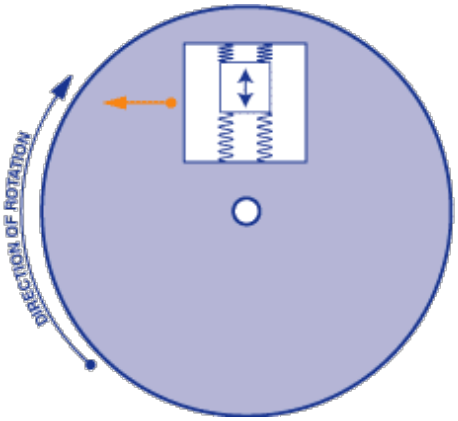
Unlike mechanical gyroscopes, MEMS gyros use the Coriolis effect to measure angular velocities. When a mass m is moving with velocity v, within a reference frame rotating at angular velocity ω, it experiences an apparent force:

$$F_c = -2m(\omega * v)$$

In MEMS gyros, vibrating elements are used to measure the Coriolis effect. In a simple configuration, there is a single mass allowed to vibrate along a drive axis; when the gyro is rotated, a second vibration occurs along the perpendicular sense axis due to the Coriolis effect. The angular velocity can be calculated from this second movement.



(a) The Coriolis Force.



(b) MEMS Gyroscope.

Compared to conventional mechanical gyros, MEMS gyros are smaller and becoming cheaper, but they are not always as accurate as the former ones.

2.2.2 Accelerometer

Accelerometers are devices that measure accelerations (i.e. the rate of change of the velocity of an object).

Mechanical Accelerometer

A mechanical accelerometer consists of a mass suspended by springs. When an acceleration occurs, as an effect of the principle of inertia, the mass will resist to any change in its velocity and the spring stretches with a force proportional to the acceleration (according to Newton's second law $F = ma$). Therefore the distance the spring stretches can be used to measure the acceleration.

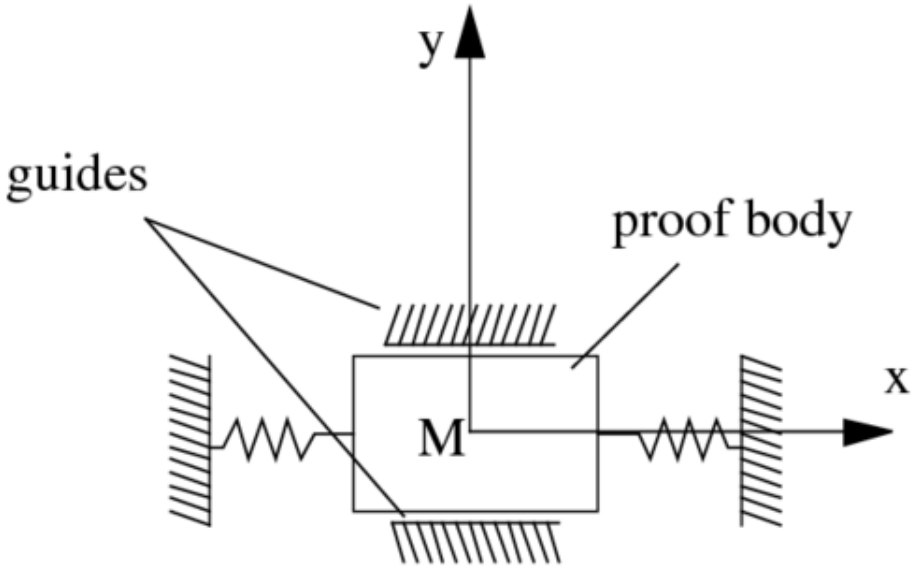
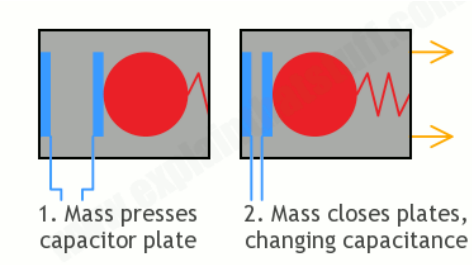


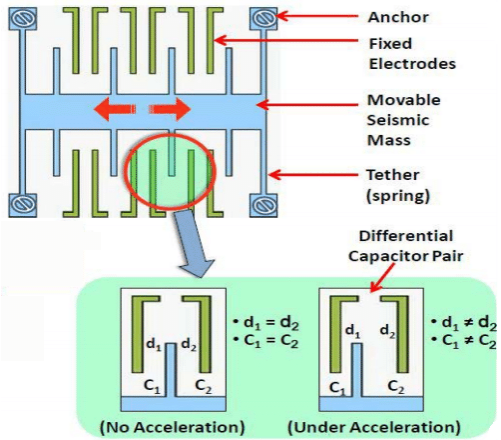
Figure 2.8. Mechanical Accelerometer.

MEMS Accelerometer

MEMS accelerometers usually contain capacitive plates internally. Some of these are fixed, while others are attached to springs and therefore able to move when subject to acceleration forces. As these plates move in relation to each other, the capacitance between them changes. Therefore measuring the capacitance we can determine the acceleration acting on the sensor.



(a) Capacitive Accelerometer.



(b) MEMS Accelerometer.

2.2.3 MEMS Sensors Error Characteristics

MEMS sensors are subject to many possible different errors; in the following we will explain only the ones we took in consideration calibrating our sensors (we will describe the calibration process we used in Section 3.3.2).

- **Constant Biases**. The bias of a MEMS sensor is the average value it outputs in a motionless state.
- **Scale Factor Erros**. The scale factor should convert the digital quantity measured by each sensor into real physical quantities (e.g, accelerations and gyro rates). Unfortunately, MEMS based IMUs are affected by non-accurate scaling; therefore the scale factor tends to be slightly different from the one expected, and the 3-axes of the same sensor may have different scale factors.
- **Thermo-Mechanical White Noise**. The output of MEMS sensors is also affected by a zero-mean thermo-mechanical noise.

Chapter 3

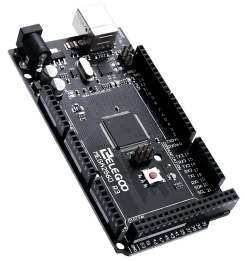
Robot Details

In this chapter we will examine the robot's architecture and details.

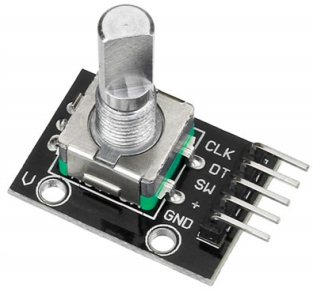
3.1 Hardware

The robot in discussion includes two encoders and an IMU in order to have motion's observation and be able, in the firmware's logic, to compute the odometry of the robot.
We have used :

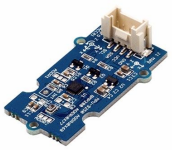
- a Mega2560 board[3],
- two rotary encoders KY-040[4],
- an IMU 10DOF MPU 9250[5].



(a) A Mega2560 board,



(b) two rotary encoders KY-040,



(c) an IMU 10DOF MPU 9250.

Figure 3.1. Hardware used for the robot.

The encoders are connected to the wheels in order to measure the wheels' rotations, while the IMU is mounted in correspondence of the center of the robot

(where as center of the robot we take the mid-point of the wheel axle) at a higher position in order to reduce the electromagnetic-noise given by the CPU computations.

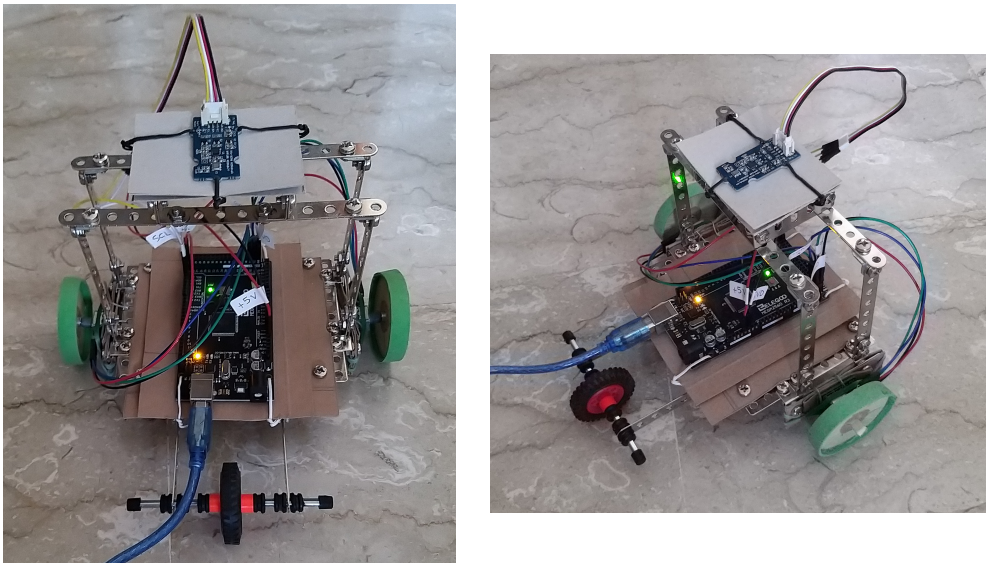


Figure 3.2. Robot used for the tests.

3.2 Communication Protocols

In the development of the firmware we have used two communication protocols:

- the I2C protocol to communicate with the IMU,
- the UART communication to interface with the host.

3.2.1 I2C Protocol

The I2C[1] protocol is a synchronous serial computer bus. It enables multi-slave and multi-master modes, but we have used it in a simple configuration using the MEGA2560 board as master and the IMU as slave.

Each device on the bus has a preset ID so that the master can choose the slave to communicate with.
This bus uses two wires:

- Serial Clock (SCL) for the clock signal (generated by the master) which synchronizes the communication between master and slave,

[1] I2C stands for Inter-Integrated Circuit

- Serial Data (SDA) carrying the data.

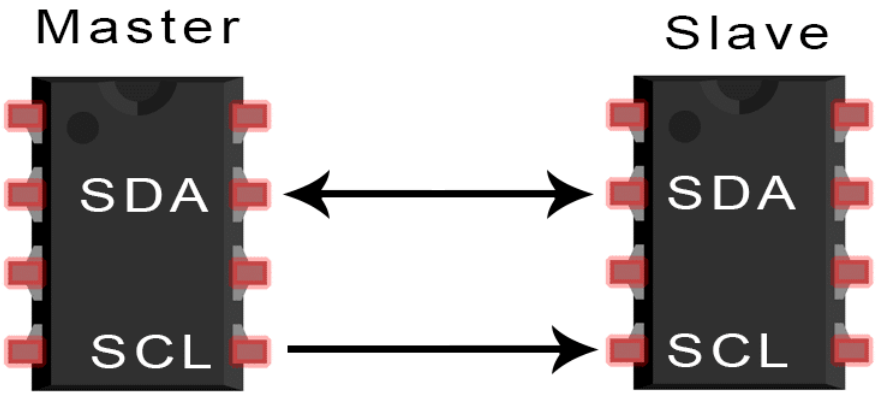


Figure 3.3. I2C wires.

The data signal is transferred in sequences of 8 bits, each of them synchronized with the clock signal.

- The communication starts with the master sending a special START condition over the bus,
- then the master sends an 8-bit sequence indicating the address of the slave device to which the communication is addressed,
- at this point the master waits for the slave to send back an Acknowledgement (ACK).
- After this first Acknowledgement by the slave device, in most cases there is a second addressing sequence indicating the address of the internal register where to read or write.
- Finally the data is sent through the bus,
- and the communication ends with a STOP condition generated by the master.

Let us now dig further in the protocol distinguishing the cases of reading and writing a register. To distinguish between a read and a write operation, in the first addressing cycle only the first 7 bits are actually used to identify the slave device, while the least significant bit is used to discriminate read and write operations [0 for a write operation, 1 for a read].

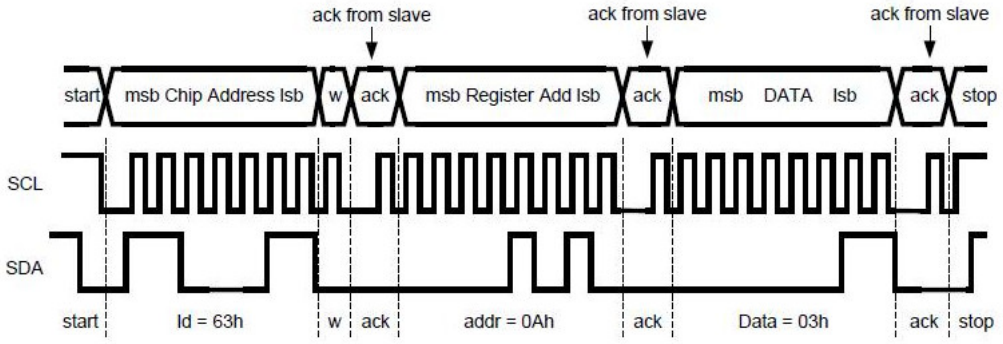


Figure 3.4. I2C general protocol.

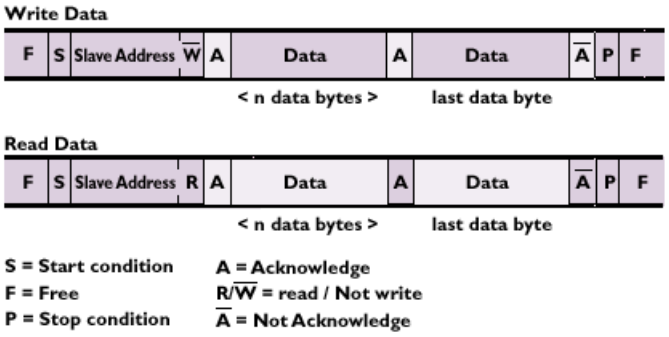


Figure 3.5. I2C protocol: distinguishing read and write operations.

Write Operations A write operation is composed by a single write transaction, where the master sends two data bytes to the slave, the first one is the address of the internal register the master wants to write to, while the second one is the value the master wants to write in it.

1. The master starts the transmission transmitting the start condition.

2. Then the master transmits the 7-bits device address, followed by the write bit (0), and waits for the slave to send an ACK.

3. At this point the master puts the address of the internal device register on the bus, and waits for the slave's ACK;

4. and next the master sends the data to be written, and waits for the slave's ACK. Eventually the master could go on sending more data which will be written in the following registers, waiting for the slave's ACK for each 8-bit data.

5. Finally the master transmits the stop condition to end the communication.

Read Operations A read operation is composed of a write transaction followed by a read transaction. In the write transaction the master sends to the slave the address of the register he wants to read, while in the read transaction the slave sends the value of the requested register to the master.

1. The master starts the write transaction transmitting the start condition.
2. Then the master transmits the 7-bits device address, followed by the write bit (0), and waits for the slave to send an ACK;
3. and next the master puts the address of the internal device register on the bus, and waits for the slave's ACK.
4. At this point the master starts the read transaction retransmitting a start condition (restart).
5. Then the master transmits the 7-bits device address, followed by the read bit (1), and waits for the slave to send an ACK;
6. and next the slave puts the data to be read on the bus. The master will send an ACK only if more data (from the following registers) is expected, when the master does not want the slave to send more data on the bus it sends a NACK (Not-Acknowledgement). Therefore the slave will stop sending data only upon receiving a NACK.
7. Finally the master transmits the stop condition to end the communication.

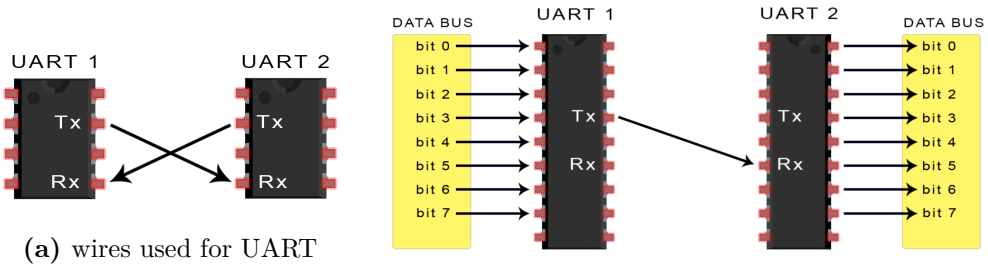
See Appendix A for an implementation of the I2C protocol.

3.2.2 UART Communication

The UART[2] is a computer hardware device for asynchronous serial communication. It is not a communication protocol as I2C but a physical circuit on the microcontroller, whose main purpose is to transmit and receive serial data.

In UART communication, two devices communicate directly with each other. The transmitting UART converts parallel data into serial data, and then transmits it in serial to the receving UART, that is in charge of reconstructing the parallel data. Only two wires are used to transmit data between devices, each of them going from the Transmit (Tx) pin of a device to the Receive (Rx) pin of the other.

[2]UART stands for Universal Asynchronous Receiver-Transmitter



(a) wires used for UART communication

(b) parallel to serial data conversion for UART communication

Figure 3.6. UART communication

1. The transmitting CPU sends the data to be sent to the UART on a data bus, then the data is transferred from the data bus to the transmitting UART in parallel form.

2. The transmitting UART gets the parallel data from the bus and creates a data packet (adding a start bit, a parity bit and a stop bit).

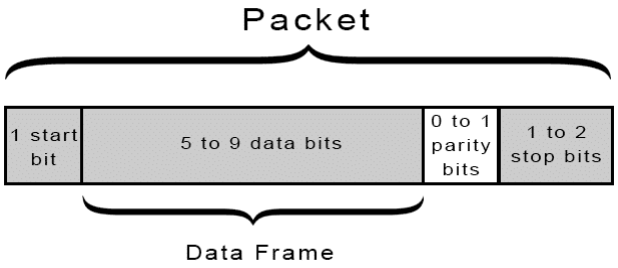


Figure 3.7. UART data packet.

3. The transmitting UART sends the data packet to the receiving UART serially, outputting it bit by bit from the Tx pin.

4. The receiving UART reads the packet bit by bit from the Rx pin,

5. and then converts the data received back into parallel form and removes the start bit, the parity bit and the stop bit.

6. Finally the receiving UART transfers the data to the receiving CPU on the data bus in parallel.

See Appendix B for an implementation of UART communication.

3.3 Firmware

All the firmware is designed to be interrupt driven. In addition to the interrupt driven implementation of the I2C protocol and the UART communication, the firmware's logic is based on timed interrupts. The sampling of the sensors and the update of the odometry is at a rate of 100 Hz, while the communication with the host is done at a rate of 1 Hz.

3.3.1 Interpreting the Encoders Observations

An encoder has two outputs A and B, which we have connected to two pins on the board.
When an encoder detects a rotation its outputs become two square waves (as described in 2.1.1), while when there is no rotation its outputs do not change their value. Therefore we have decided to use an interrupt solution, throwing an interrupt when one of the two pins changed its value.
We have connected the encoder corresponding to the left wheel to the pins 52-53 (PB1 and PB0 respectively[3]) while the one on the right wheel to the pins 50-51 (PB3 and PB2 respectively). Therefore we are using the first 4 pins of port B.

```
1   #define ENC_MASK 0xf //0b1111
2
3   cli();
4
5     //set the encoder pins as input
6   DDRB &= ~ENC_MASK;
7     //enable pull up resistors
8   PORTB |= ENC_MASK;
9     //set interrupt on change, looking up PCMSK0
10  PCICR |= (1 << PCIE0);
11    //tells which bits of the port triggers the interrupt
12  PCMSK0 |= ENC_MASK;
13
14  sei();
```

Listing 3.1. Installing the interrupt for handling the encoders.

The ISR will then be responsible for the update of the state of the encoders. For each encoder we need to store:

- a global counter, counting the total number of rotations the encoder has been subject to. Note that clockwise rotations increment the counter, while anticlockwise rotations decrement it.
- the previous value of the output pins, since in an incremental encoder the

sense of the rotation is encoded in the transition between states.

```
typedef struct Encoder_t {
  uint8_t prev_value; // previous value of the output pins
  int32_t counter;  // global counter of the encoder
} Encoder_t;
```

Listing 3.2. Structure used to store all useful informations about encoders.

Therefore the ISR recovers the actual and the previous values of the output pins and uses them to lookup in a transition table and update the encoder rotation counter.

```
#define NUM_ENCODERS 2

  //global variable storing the current state of the encoders
    //updated by the ISR
static Encoder_t Encs[NUM_ENCODERS];

static const int8_t _transition_table []= {
    0, //0000
   -1, //0001
    1, //0010
    0, //0011
    1, //0100
    0, //0101
    0, //0110
   -1, //0111
   -1, //1000
    0, //1001
    0, //1010
    1, //1011
    0, //1100
    1, //1101
   -1, //1110
    0  //1111
};

// Interrupt Service Routine for PCINT0
ISR(PCINT0_vect) {
  uint8_t port_value = PINB & ENC_MASK;

  uint8_t i;
  for(i=0; i<NUM_ENCODERS; i++){
      //we need the previous state, since in incremental encoders
        //we handle transitions from a state to the next
    Encs[i].prev_value <<=2;

      // port_value stores the new values of all encoders
```

```
37          // with &0x03 we can handle an encoder at the time
38          // and with >>= 2 t the end, we pass to the next encoder
39        Encs[i].prev_value = Encs[i].prev_value | (port_value & 0x03);
40        Encs[i].counter += _transition_table[Encs[i].prev_value & 0x0F
              ↪ ];
41
42        port_value >>= 2;
43      }
44    }
```

Listing 3.3. Installing the interrupt for handling the encoders.

3.3.2 Interpreting the IMU Observations

The IMU MPU9250 provides an I2C interface, through which we can access its control register and its 16-bit digital output registers[5].

Initializing the IMU

Reading the IMU datasheet[5] and the register map[6] we were able to initialize the configuration registers according to our needs.

The gyroscope's full scale range was set to 250 DPS[3], hence it can observe only angular velocities in the range [-250 DPS, +250 DPS]. The accelerometer's full scale range instead was set to 2 G[4] and therefore it can measure accelerations in the range [-2 G, +2 G].
These are a quite small ranges but perfect for our needs since we do not expect the robot to exceed these bounds; moreover the use of small ranges gives us high sensitivity in these ranges.

The gyroscope's bandwidth, which is the highest frequency of angular velocity variation that can be detected by the sensor, was set to 41 Hz. The accelerometer's bandwidth instead was set to 44.8 Hz.
According to the Nyquist Sampling Criterion, the rate at which the sensors are sampled must be at least twice the sensors' bandwidths. Therefore since we want to sample the IMU sensors at 100 Hz, we had to set the sensors' bandwidths below 50 Hz.

```
1   static void IMU_ConfigRegs(void) {
2     //resets the internal registers and restores the default settings
```

[3]DPS stands for Degrees Per Second
[4]1 G corresponds to the gravitational acceleration (i.e. $\sim 9.8 m/s^2$)

```
3     I2C_WriteRegister(ACCELGYRO_DEVICE, PWR_MGMT_1, 0x80);
4     _delay_ms(500);
5
6     //sets Gyroscope Full Scale range to 250dps (degrees/sec)
7       //[low range, high sensitivity]
8     I2C_WriteRegister(ACCELGYRO_DEVICE, GYRO_CONFIG, 0x00);
9     _delay_ms(10);
10
11    //sets Gyroscope bandwidth to 41Hz,
12    I2C_WriteRegister(ACCELGYRO_DEVICE, CONFIG, 0x03);//41 Hz
13    _delay_ms(10);
14
15    //sets Accelerometer Full Scale to 2G
16    I2C_WriteRegister(ACCELGYRO_DEVICE, ACCEL_CONFIG, 0x00);
17    _delay_ms(10);
18
19    //sets Accelerometer bandwidth to 44.8 Hz
20    I2C_WriteRegister(ACCELGYRO_DEVICE, ACCEL_CONFIG2, 0x03);//44.8
          ↪ Hz
21    _delay_ms(10);
22
23    //sets PowerManagement Registers
24      //auto selects the best available clock source
25      //and disables the sleep mode
26    I2C_WriteRegister(ACCELGYRO_DEVICE, PWR_MGMT_1, 0x01);
27    _delay_ms(10);
28
29    //sets all sensors to on
30    I2C_WriteRegister(ACCELGYRO_DEVICE, PWR_MGMT_2, 0x00);
31    _delay_ms(10);
32
33    //i2c_master interface pins(SCL, SDA)
34      //will go into "bypass mode"
35      //when the i2c master interface is disabled
36    I2C_WriteRegister(ACCELGYRO_DEVICE, INT_PIN_CFG, 0x02);
37    _delay_ms(10);
38  }
```

Listing 3.4. IMU initialization of configuration registers.

Reading IMU registers

As mentioned in the previous paragraph, we want to sample the IMU sensors at a rate of 100 Hz. Therefore we defined a timed interrupt with a frequency of 100 Hz and we gave to the corresponding interrupt service routine the task of reading the IMU data registers.

```
1  #define IMU_UPDATE_RATE 100 //Hz
2
3  //from a frequency in Hz, we get a period in millisecs
4  uint16_t period_ms = 1000 / IMU_UPDATE_RATE;
5
6  //configure timer3, prescaler : 256, CTC (Clear Timer on Compare
       ↪ match)
7  TCCR3A = 0;
8  TCCR3B = (1 << WGM12) | (1 << CS12);
9
10 /*
11   * cpu frequency 16MHz = 16.000.000 Hz
12   * prescaler 256
13   * -->> TCNT3 is increased at a frequency of 16.000.000/256 Hz =
         ↪ 62500 Hz
14   * so 1 ms will correspond do 62.5 counts
15   */
16 OCR3A = (uint16_t)(62.5 * period_ms);
17
18  // timer-interrupt enabling must be executed atomically (no other
       ↪ interrupts)
19    // and ATOMIC_FORCEON ensures Global Interrupt Status flag bit in
         ↪  SREG set afetrwards
20 ATOMIC_BLOCK(ATOMIC_FORCEON) {
21  TIMSK3 |= (1 << OCIE3A);
22 }
```

Listing 3.5. Installing the timed-interrupt to periodically read the IMU data registers.

```
1  ISR(TIMER3_COMPA_vect) {
2    //reads the accelerometer and the gyroscope data registers
3    IMU_AccelGyroRaw();
4
5    //increases the time stamp
6    IMU.imu_time_seq++;
7  }
```

Listing 3.6. ISR in charge of reading the IMU data registers.

The output data from the accelerometer and the gyroscope are 16-bit values for each axis, divided in two 8-bit registers. Since the 6 registers for the accelerometer and the six ones for the gyroscope are consecutive, we can read them in a single I2C communication cycle and then reconstruct the 16-bit digital values.

Calibrating the IMU sensors

As described in paragraph 2.2.3, MEMS sensors are subject to errors, some of which can be compensated through a prior calibration process[78].

To compensate constant bias errors for the accelerometer and the gyroscope, we calculate the calibration biases by taking a long term average of the sensors' outputs while the IMU is motionless.
To compensate the scale factor errors of the accelerometer, we use a very simple multi-position scheme. By keeping the IMU with its z-axis upwards (aligned with the vertical gravity field) we expect to measure an acceleration of 1 G along the z-axis; therefore we can use the digital output of the sensor as the scale factor of the z-axis. By aligning the other axes with the gravity field, we calculate the scale factors for all the other axes of the accelerometer.
We have calculated the scale factors of the gyroscope starting from the full scale range. Since we expect the full scale range to correspond to the maximum value of the digital output, we have calculated the scale factor by dividing the maximum digital value by the maximum physically meaningful value than can be measured: $scale\_factor = 2^{15}/250$. We did not compensate the scale factor errors of the gyroscope.

```
1  #define CALIBRATION_SAMPLES 128
2  #define CALIBRATION_SAMPLES_LOG 7
3
4  #define IMU_POS_Z_UP 0 // z-axis upwards
5  #define IMU_POS_X_UP 1 // x-axis upwards
6  #define IMU_POS_Y_UP 2 // y-axis upwards
7  #define IMU_POS_Z_DOWN 3 // z-axis downwards
8  #define IMU_POS_X_DOWN 4 // x-axis downwards
9  #define IMU_POS_Y_DOWN 5 // y-axis downwards
10 #define IMU_N_POS 6 //max number if positions
11
12 int32_t gyro_x_sum[IMU_N_POS]={}, gyro_y_sum[IMU_N_POS]={},
       ↪ gyro_z_sum[IMU_N_POS]={};
13 int32_t accel_x_sum[IMU_N_POS]={}, accel_y_sum[IMU_N_POS]={},
       ↪ accel_z_sum[IMU_N_POS]={};
14
15 uint8_t p, i, orientation;
16
17 for(p=0; p < IMU_N_POS; p++) {
18   //waits for the robot is in a new orientation
19     // ...
20
```

```
21      //sets ths local variable to the current IMU orientation
22      orientation = <imu_orientation>;
23
24      //get samples
25      for(i=0; i<CALIBRATION_SAMPLES; i++) {
26        gyro_x_sum[orientation] += <gyro_raw_x_observation>;
27        gyro_y_sum[orientation] += <gyro_raw_y_observation>;
28        gyro_z_sum[orientation] += <gyro_raw_z_observation>;
29
30        accel_x_sum[orientation] += <accel_raw_x_observation>;
31        accel_y_sum[orientation] += <accel_raw_y_observation>;
32        accel_z_sum[orientation] += <accel_raw_z_observation>;
33      }
34
35      gyro_x_sum[orientation] >>= CALIBRATION_SAMPLES_LOG;
36      gyro_y_sum[orientation] >>= CALIBRATION_SAMPLES_LOG;
37      gyro_z_sum[orientation] >>= CALIBRATION_SAMPLES_LOG;
38
39      accel_x_sum[orientation] >>= CALIBRATION_SAMPLES_LOG;
40      accel_y_sum[orientation] >>= CALIBRATION_SAMPLES_LOG;
41      accel_z_sum[orientation] >>= CALIBRATION_SAMPLES_LOG;
42    }
43
44    // GYROSCOPE
45    int32_t gyro_total_x_sum=0, gyro_total_y_sum=0, gyro_total_z_sum=0;
46    for(p=0; p<6; p++) {
47      gyro_total_x_sum += gyro_x_sum[p];
48      gyro_total_y_sum += gyro_y_sum[p];
49      gyro_total_z_sum += gyro_z_sum[p];
50    }
51
52      //calculates the gyro's constant biases
53    int32_t gyro_x_bias = (int16_t)(gyro_total_x_sum / 6);
54    int32_t gyro_y_bias = (int16_t)(gyro_total_y_sum / 6);
55    int32_t gyro_z_bias = (int16_t)(gyro_total_z_sum / 6);
56
57      //calculates the gyro's scale factor (no calibration)
58    float gyro_sensitivity = (float)((uint16_t)(1<<15) / 250.);
59
60    float gyro_x_scale = gyro_sensitivity;
61    float gyro_y_scale = gyro_sensitivity;
62    float gyro_z_scale = gyro_sensitivity;
63
64    //ACCELEROMETER
65    //asse X
66      //to calculate the bias, we take in consideration the positions
          ↪ where we expect zero-acceleration along the axis
```

```
67   int32_t accel_x_bias_sum = accel_x_sum[IMU_POS_Z_UP]+accel_x_sum[
         ↪ IMU_POS_Y_UP]+accel_x_sum[IMU_POS_Z_DOWN]+accel_x_sum[
         ↪ IMU_POS_Y_DOWN];
68     //to calculate the scale factor, we take in consideration the
           ↪ positions with +1G and -1G acceleration
69   int32_t accel_x_scale_sum = accel_x_sum[IMU_POS_X_UP]-accel_x_sum[
         ↪ IMU_POS_X_DOWN];
70
71   //asse Y
72   int32_t accel_y_bias_sum = accel_y_sum[IMU_POS_Z_UP]+accel_y_sum[
         ↪ IMU_POS_X_UP]+accel_y_sum[IMU_POS_Z_DOWN]+accel_y_sum[
         ↪ IMU_POS_X_DOWN];
73   int32_t accel_y_scale_sum = accel_y_sum[IMU_POS_Y_UP]-accel_y_sum[
         ↪ IMU_POS_Y_DOWN];
74
75   //asse Z
76   int32_t accel_z_bias_sum = accel_z_sum[IMU_POS_X_UP]+accel_z_sum[
         ↪ IMU_POS_Y_UP]+accel_z_sum[IMU_POS_X_DOWN]+accel_z_sum[
         ↪ IMU_POS_Y_DOWN];
77   int32_t accel_z_scale_sum = accel_z_sum[IMU_POS_Z_UP]-accel_z_sum[
         ↪ IMU_POS_Z_DOWN];
78
79     //calculates the accel's constant biases
80   int32_t accel_x_bias = (int16_t)(accel_x_bias_sum >> 2);
81   int32_t accel_y_bias = (int16_t)(accel_y_bias_sum >> 2);
82   int32_t accel_z_bias = (int16_t)(accel_z_bias_sum >> 2);
83
84     //calculates the accel's scale factor
85       // we must not keep track of bias calculating the scale
86         //because we are subtracting two accel_x_sum values ( one
               ↪ positive, one negative )
87           //accel_x_scale_sum = accel_x_sum[IMU_POS_X_UP] -
                 ↪ accel_x_sum[IMU_POS_X_DOWN];
88             //we would have +BIAS the first time, and -BIAS the second
89           //and the two biases goes away
90   float accel_x_scale = (float)(accel_x_scale_sum >> 1);
91   float accel_y_scale = (float)(accel_y_scale_sum >> 1);
92   float accel_z_scale = (float)(accel_z_scale_sum >> 1);
```

Listing 3.7. Accelerometer and Gyroscope Calibration.

Interpreting values read from the registers

Upon receipt of a 16-bit digital value, we must convert it to a physically meaningful value.

First of all we subtract the calibration bias, then we divide the 16-bit digital value

for the scale factor calculated during the calibration.

```
1  //in accel_raw_x, accel_raw_y, accel_raw_z we store the 16-bit
       ↪ values read from the sensor
2  //while in accel_x, accel_y, accel_z we store physically meaningful
       ↪  values
3
4  accel_x = (float)(accel_raw_x - accel_x_bias) /
       ↪ accel_x_scale_factor;
5  accel_y = (float)(accel_raw_y - accel_y_bias) /
       ↪ accel_y_scale_factor;
6  accel_z = (float)(accel_raw_z - accel_z_bias) /
       ↪ accel_z_scale_factor;
```

Listing 3.8. Converting the 16-bit digital accelerometer values to physically meaningful values (i.e. accelerations). [Similarly it can be done for the gyroscope.]

3.3.3 Communication with the Host

The communication with the host is packet-based and performed via UART communication. At a frequency of 1 Hz the firmware sends to the host the estimated position using differential drive and the one estimated using inertial navigation.

```
1  typedef uint8_t PacketType;
2  typedef uint8_t PacketSize;
3  typedef uint16_t PacketSeq;
4
5  #pragma pack(push, 1)
6  typedef struct {
7    PacketType type; // type of the packet
8    PacketSize size; // size of the packet in bytes
9    PacketSeq seq;  // sequential number
10 } PacketHeader;
11
12 typedef struct {
13   PacketHeader header;
14
15   //packet-specific data
16   //...
17 } ExamplePacket;
18 #pragma pack(pop)
```

Listing 3.9. Structure of the packets used for Firmware-Host Communication via UART.

3.4 Host

The host has a multi-thread structure. There is a comunication thread in charge of listening to the serial port and storing the information sent by the firmware and the main thread which is just logging the data received on the screen at a rate of 1 Hz.

3.4.1 Serial Interface

On the host the serial is accessible at */dev/ttyACM0*. We have used the *termios* interface to configure it.

```
1  #define SERIAL_NAME "/dev/ttyACM0"
2
3  int fd = open(SERIAL_NAME, O_RDWR | O_NOCTTY | O_SYNC);
```

Listing 3.10. Opening the Serial as a file.

```
1  #define SERIAL_SPEED 57600
2  #define SERIAL_PARITY 0
3
4  struct termios tty;
5  memset(&tty, 0, sizeof(tty));
6
7  // store in tty the parameters associated to the serial device
       ↪ referred by fd
8  res = tcgetattr(fd, &tty);
9  //... check res >= 0
10
11 //converts speed into one of the speed_t Bnnn constants
12 speed_t speed_bnnn = B0;
13 switch (speed){
14   case 19200:
15     speed_bnnn=B19200;
16     break;
17   case 38400:
18     speed_bnnn=B38400;
19     break;
20   case 57600:
21     speed_bnnn=B57600;
22     break;
23   case 115200:
24     speed_bnnn=B115200;
25     break;
26   default:
```

```
27        printf("[serial_set_interface_attribs] Cannot set baudrate to %
              ↪ d\n", speed);
28        return -1;
29    }
30    // sets the input baud rate stored in the termios structure to
          ↪ speed
31    cfsetispeed(&tty, speed_bnnn);
32    // sets the output baud rate stored in the termios structure to
          ↪ speed
33    cfsetospeed(&tty, speed_bnnn);
34
35    // sets the terminal to a "raw" mode
36      // input is available character by character, echoing is disabled
37      // all special processing of terminal input and output characters
            ↪  is disabled.
38      // serial switchs to noncanonical mode
39    cfmakeraw(&tty);
40
41    // PARENB : Enable parity generation on output and parity checking
          ↪ for input.
42    // PARODD : If set, then parity for input and output is odd;
43                  //otherwise even parity is used.
44    if( parity )
45      tty.c_cflag |= (PARENB | PARODD);
46    else
47      tty.c_cflag &= ~PARENB;
48
49    // sets characters' size to 8-bit
50    tty.c_cflag &= ~CSIZE;
51    tty.c_cflag |= CS8;
52
53    // set blocking mode
54      // MIN > 0; TIME > 0:
55        // TIME specifies the limit for a timer in tenths of a second.
56          // Once an initial byte of input becomes available,
57            //timer is restarted after each further byte is received.
58        // read returns either
59          // when min {number of bytes requested, MIN bytes have been
                ↪ read }
60          // when the inter-byte timeout expires.
61        // Because the timer is only started after the initial byte
62          // becomes available, at least one byte will be read.
63    tty.c_cc[VMIN] = 1;
64    tty.c_cc[VTIME] = 5; // 0.5 seconds read timeout
65
66    // sets the parameters associated with the serial device
67      // from the termios structure tty
```

```
68      // TCSANOW : the change occurs immediately
69    res = tcsetattr(fd, TCSANOW, &tty);
70    //... check res >= 0
```

Listing 3.11. Configuration of the serial interface.

Using this configuration it is possible to perform read and write operations on the serial as if it was a local file on the host.

3.4.2 ROS node

Furthermore we also provide a ROS node interfacing with the ROS environment. This node publishes the two odometries on two different topics so that they can be easily visualized in RViz[5].

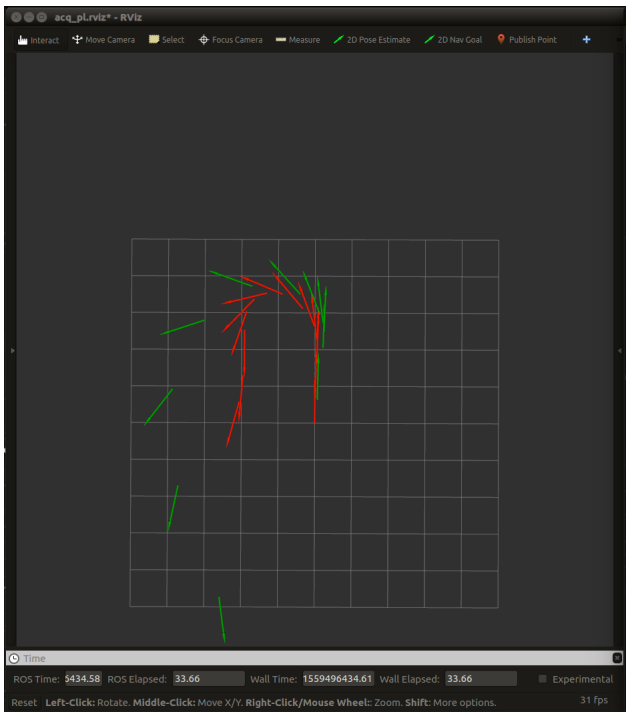


Figure 3.8. Data Visualization in RViz. In red we are plotting the position estimated using differential drive, while in green the one using inertial navigation.

[5] RViz is the ROS visualization tool

Chapter 4

Localization

As mentioned in Chapter 1, localization is the process of determining the current robot position using the measurements up to the current instant.
In the first section we will describe the Differential Drive technique, which uses the encoder measurements; in the second section we will describe the Inertial Navigation technique, which uses the IMU measurements.

4.1 Differential Drive

Differential Drive[9] is a localization technique consisting in calculating the odometry of the robot from the instant rotation of each wheel. In the following we will assume 2 drive wheels mounted on the same axis, and we will take the center of the robot on the mid-point of the wheel axis.
We will call, b the length of the basis of the robot (i.e. the length of the wheel axis) and C the center of the robot.

We use the two rotary encoders to have measurements about the wheels' rotations. We sample the values of the encoders at 100 Hz. Knowing the radius of each wheel, we can then compute the distance traveled by each wheel every 10ms starting from the corresponding measured rotation.
In the following we will call l, r the distance traveled by the left and by the right wheel respectively.

4.1.1 Movement in the Local Reference Frame

We can model each instant movement as a rotation of an angle α around a point O lying on the wheel axis, at a distance R from the center of the robot (note that R

will be the radius of the robots' rotation).

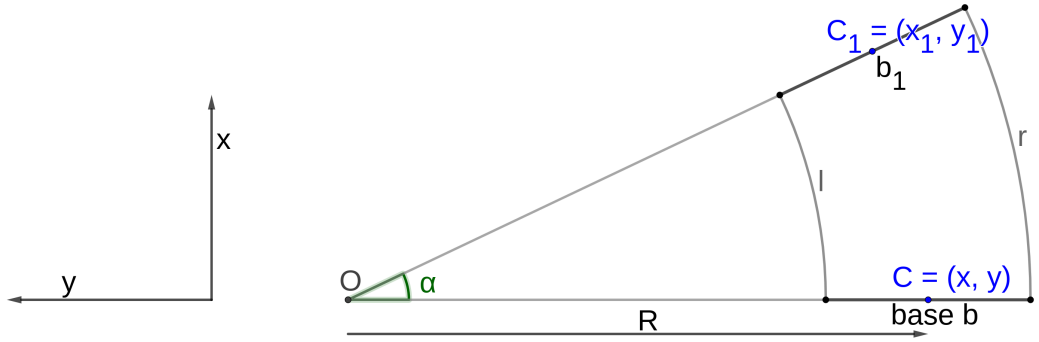


Figure 4.1. Model of an instantaneous movement through a rotation.

Using this model, we can express the local movement of each wheel l, r as a function of the parameters of the instant rotation R, α[1].

$$r = (R + \frac{b}{2})\alpha$$
$$l = (R - \frac{b}{2})\alpha$$

Respectively adding and subtracting these two equations we get:

$$r + l = 2R\alpha$$
$$r - l = b\alpha$$

And therefore we are able to express R, α as a function of l, r:

$$\alpha = \frac{r - l}{b} \tag{4.1}$$

$$R = \frac{r + l}{2\alpha} \tag{4.2}$$

We are now ready to express the local movement in the local reference frame:

[1] The length of an arc of a circle with radius r and subtending an angle α (measured in radians) with the circle center is : $l = \alpha\ r$

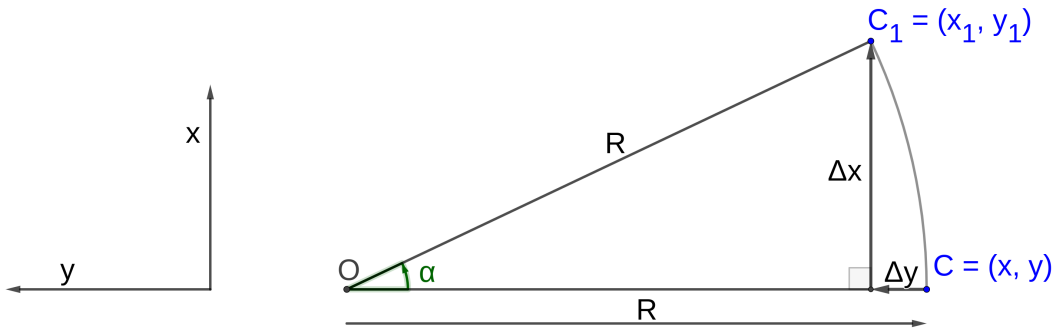


Figure 4.2. Model of an instantaneous movement through a rotation (emphasis on the local movement).

$$\Delta x = R sin\alpha \tag{4.3}$$

$$\Delta y = R - R cos\alpha = R(1 - cos\alpha) \tag{4.4}$$

and finally, using the (4.2):

$$\Delta x = \frac{r+l}{2} \frac{sin\alpha}{\alpha} \tag{4.5}$$

$$\Delta y = \frac{r+l}{2} \frac{1-cos\alpha}{\alpha} \tag{4.6}$$

$$\Delta\theta = \alpha \tag{4.7}$$

Since $\frac{sin\alpha}{\alpha}$ and $\frac{1-cos\alpha}{\alpha}$ have a singularity for $\alpha = 0$, we will use the corresponding Taylor-Maclaurin series[2]:

$$\frac{sin\alpha}{\alpha} = 1 - \frac{1}{3!}\alpha^2 + \frac{1}{5!}\alpha^4 - \frac{1}{7!}\alpha^6 + ... \tag{4.8}$$

$$\frac{1-cos\alpha}{\alpha} = \frac{1}{2}\alpha - \frac{1}{4!}\alpha^3 + \frac{1}{6!}\alpha^5 + ... \tag{4.9}$$

[2]Taylor-Maclaurin series for sine and cosine:

$sin\alpha = \alpha - \frac{1}{3!}\alpha^3 + \frac{1}{5!}\alpha^5 - \frac{1}{7!}\alpha^7 + ...$

$cos\alpha = 1 - \frac{1}{2}\alpha^2 + \frac{1}{4!}\alpha^4 - \frac{1}{6!}\alpha^6 + ...$

4.1.2 Movement in the Global Reference Frame

Having calculated the local movement, we are ready to integrate the latest sensors' measurements in the robot's global odometry.

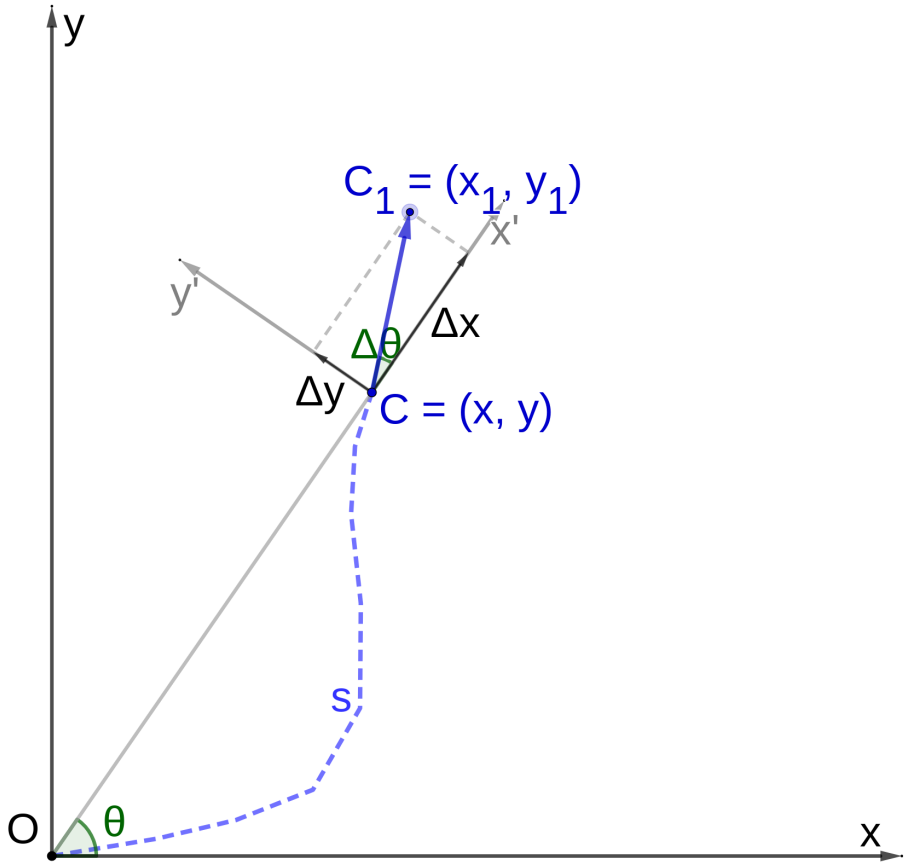


Figure 4.3. Movement in the Global Reference Frame.

$$x+ = \Delta x cos\theta - \Delta y sin\theta \tag{4.10}$$

$$y+ = \Delta x sin\theta + \Delta y cos\theta \tag{4.11}$$

$$\theta+ = \Delta\theta \tag{4.12}$$

4.1.3 Velocities Update

To calculate the velocities, we will consider the movement of its central point C as the movement fo the robot (using the (4.2)).

$$\Delta s = R\alpha = \frac{r+l}{2\alpha}\alpha = \frac{r+l}{2} \tag{4.13}$$

At this point we can also calculate the instant robot's velocities, we will call v_t the translational velocity and v_r the rotational velocity:

$$v_t = \frac{\Delta s}{\Delta t} = \frac{r + l}{2\Delta t} \tag{4.14}$$

$$v_r = \frac{\Delta \theta}{\Delta t} \tag{4.15}$$

For an implementation of Differential Drive see Appendix C.

4.2 Inertial Navigation

Inertial navigation is a navigation technique in which the measurements provided by the accelerometer and the gyroscope are used to update the estimate of the current position and orientation of the robot with respect to a known starting point.[2]

4.2.1 Dead Reckoning

As mentioned in Section 2.2, to keep track of the orientation of the robot we need to integrate the gyroscope measurements, while to keep track of the position we can integrate the accelerometer measurements obtaining the local displacement and then project it on the global axes (using the orientation determined using the gyroscope measurements).

Updating Orientation

At each sampling time interval, the gyroscope gives us the observation of the angular velocity of the robot.
Therefore since we observe $v_r = \frac{\partial \theta}{\partial t}$, the integration is immediate:

$$\theta += v_r \Delta t \tag{4.16}$$

Updating Position

At each sampling time interval, the accelerometer provides observations on the translational accelerations along the x-axis and the y-axis of the local reference frame.

We will first of all calculate the instant movement in the local reference frame:

$$\Delta x = v_x \Delta t + \frac{1}{2} a_x \Delta t^2 \tag{4.17}$$

$$\Delta y = v_y \Delta t + \frac{1}{2} a_y \Delta t^2 \tag{4.18}$$

At this point we can project it onto the global axes, calculating the displacement in the global reference frame (in a similar way as we did for the differential drive in Section 4.1.2):

$$x+ = \Delta x cos\theta - \Delta y sin\theta \tag{4.19}$$

$$y+ = \Delta x sin\theta + \Delta y cos\theta \tag{4.20}$$

4.2.2 High-pass filter

However this basic navigation algorithm is very inaccurate. One of the main reasons is that the IMU is subject to electromagnetic noise and therefore will measure non-zero accelerations and rotations even if the robot is not moving.
This problem is far more evident for the position rather than for the orientation, due to the double integration in the navigation algorithm which increases the error.

The first solution we have adopted is a simple high-pass filter, hence we ignore translational accelerations and rotational velocities below a predefined threshold (∼0.4 DPS[3] for rotational velocities and ∼0.05 G-Forces for translational accelerations).

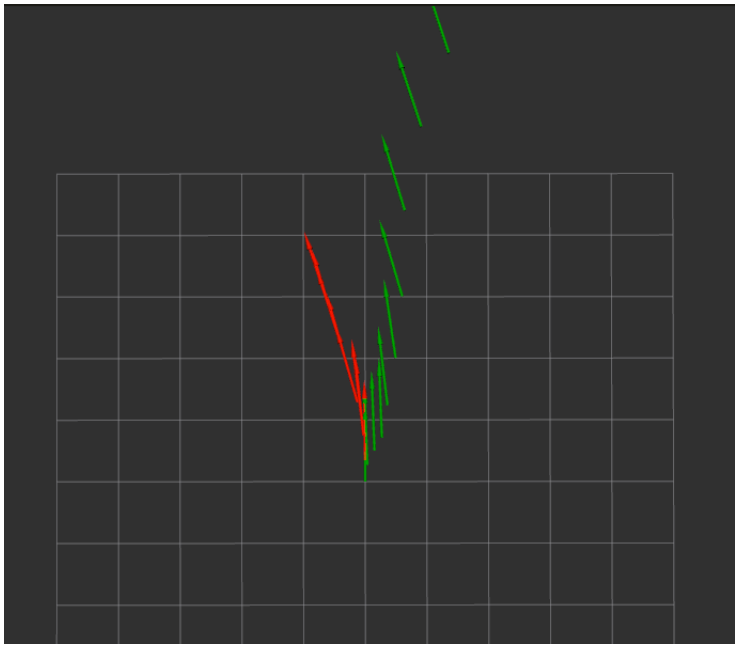


Figure 4.4. Inertial Navigation (with high-pass filter) trajectory (in green) compared to the more accurate Differential Drive trajectory (in red) visualized using Rviz.

[3]DPS stands for Degrees Per Second

4.2.3 Stop Motion detection

Using the described high pass filter we have obtained a more accurate result, however once this localization algorithm has detected the movement of the robot, it is not able to detect when the robot stops. This is because even if the accelerometer no longer detects any acceleration, the previous accelerations have produced a non-zero velocity that will hardly be zeroed when the robot stops.

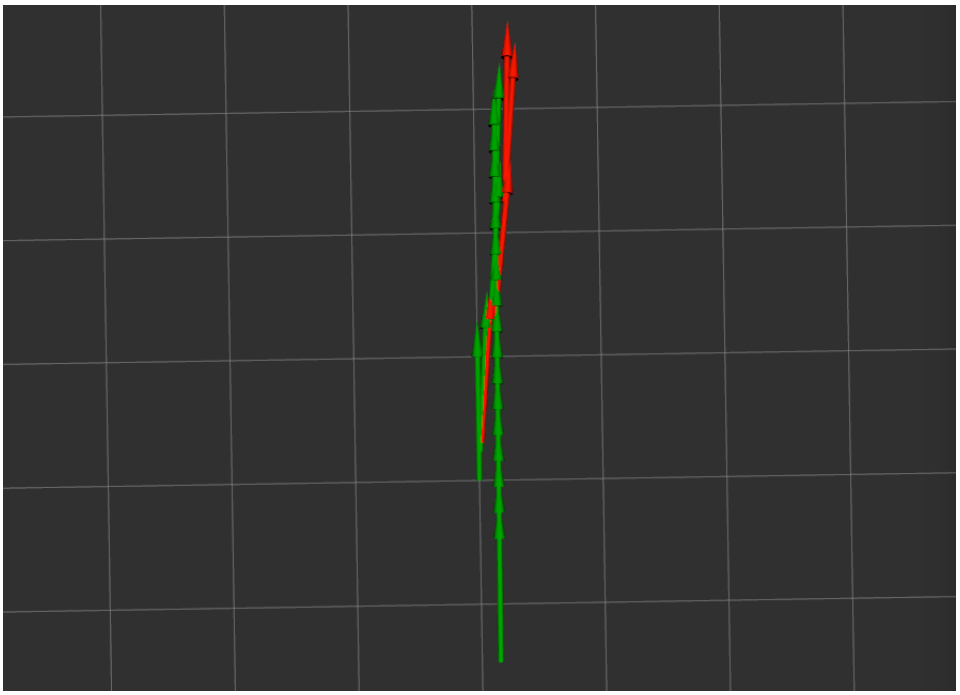


Figure 4.5. Inertial Navigation trajectory (in green) compared to the more accurate Differential Drive trajectory (in red) visualized using Rviz. We can notice how after a first phase in which the Inertial Navigation odometry followed the Differential Drive odometry, the former was unable to detect the motionless state and (having a negative velocity due to the braking phase) brought the odometry back beyond the starting position before we stopped the visualization.

Therefore we designed a *Stop Motion Detection Algorithm* which basically counts the instants with positive and negative acceleration between two motionless states and the consecutive instants with zero acceleration for each axis.
When the number of consecutive instants with no acceleration exceeds the difference between the instants with positive and negative acceleration scaled by a predefined factor, we assume that the robot is motionless due to friction and therefore we zero its velocity.

```
#define STOP_ZERO_ACCEL_OUTLIER_FILTER 5
#define STOP_BREAKING_TRESHOLD 2

if ( translational_acceleration_x_axis > 0. ) {
  total_time_pos_accel_x++;
  curr_time_zero_accel_x = 0;
} else if ( translational_acceleration_x_axis < 0. ) {
  total_time_neg_accel_x++;
  curr_time_zero_accel_x = 0;
} else //accel_x_axis == 0.
  curr_time_zero_accel_x++;

if( curr_time_zero_accel_x > STOP_ZERO_ACCEL_OUTLIER_FILTER &&
    curr_time_zero_accel_x >= abs( total_time_pos_accel_x -
        ↪ total_time_neg_accel_x) * STOP_BREAKING_TRESHOLD) {
  //after it has had approximately
    //the same time of positive and negative acceleration
    //we assume it is in STATIONARY state
  translational_acceleration_x_axis = 0.;
  translational_velocity_x_axis = 0.;

  //we reset the variables
  total_time_pos_accel_x = 0;
  total_time_neg_accel_x = 0;
}
```

Listing 4.1. Implementation of the described Stop Motion Detection algorithm.

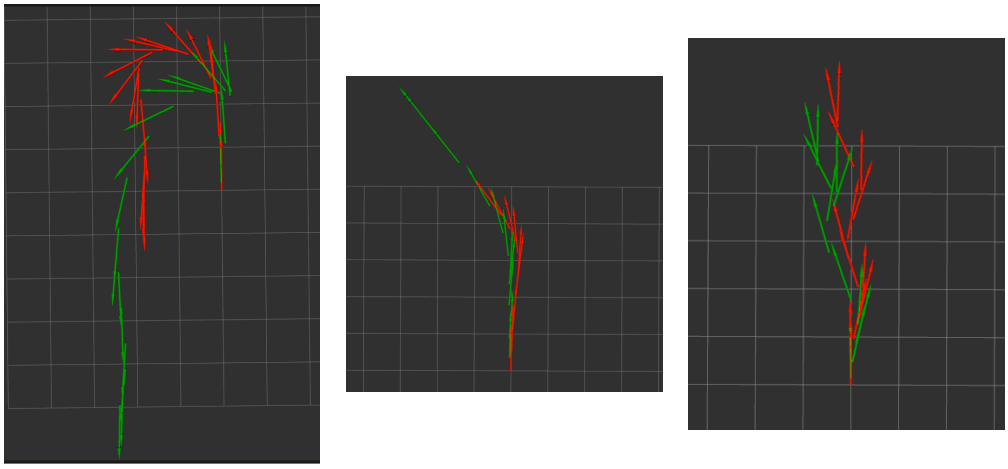


Figure 4.6. Inertial Navigation (with Stop Motion Detection) trajectories (in green) compared to the more accurate Differential Drive trajectories (in red) visualized using RViz.

In the images 4.6, we can see how the Inertial Navigation technique, even with the high-pass filter and the Stop Motion Detection, is far less accurate than the

Differential Drive technique; however it is now able to detect a motionless state, eventually with some latency, without moving the odometry to infinity.
For an implementation of Inertial Navigation see Appendix D.

Chapter 5

Conclusions

In this thesis we have presented and performed some testing of two different localization techniques: Differential Drive and Inertial Navigation.
The former is more accurate than the latter especially in the detection of motionless states, however it can have problems on certain surfaces if the wheels start to slip. Inertial Navigation is affected by electromagnetic noise and calibration errors which have been reduced with our simple and fast calibration process described in Section 3.3.2; results could be further improved with a more accurate calibration.

So far we are only using the accelerometer and the gyroscope sensors of the IMU, while it also includes a magnetometer which provides a long-term North reference and could be used to correct for the drift of the gyroscope[10].

Furthermore, the encoders and the IMU have been used separately, whilst they could be used together in the update of a single global odometry. In this way each sensor would compensate for the weaknesses of the other. This issue has been addressed and demonstrated in the additional work we have performed for the Honours Programme[1], where we have performed sensor fusion based on Kalman filtering.

[1]The Honours Programme stands for "Percorso d'Eccellenza".

Appendix A

I2C Protocol Implementation

In this appendix we present two implementations of the I2C protocol. The first one is a busy-waiting implementation, while the second one is an interrupt-driven implementation. These implementations are specific for the board used[3], and uses the TWI module offered by the board[1].

A.1 I2C Control Registers

On the board used, the TWI module is controlled by the following set of registers.

- TWBR (TWI Bit Rate Register) is used to control the SCL clock frequency.
- TWCR (TWI Control Register) is used to control TWI operations and to poll if the current operation has ended.
- TWSR (TWI Status Register) is used to check the status of the TWI module and of the current transaction.
- TWDR (TWI Data Register) contains the data travelling on the data bus.

A.2 I2C Busy-Waiting Primitives Implementation

In this section we will show a busy-waiting implementation of the primitives needed for the I2C protocol. After the start of each operation in fact the CPU polls the TWCR register until the end of the operation.

```
1  #define FREQ_CPU 16000000L //cpu clock speed at 16MHz
2  #define FREQ_I2C 400000L  //i2c clock speed at 400KHz
3
4
```

[1] TWI is a variant of I2C used by manufacturers like Atmel

```
5   //status indicating START condition successfully sent
6   #define TWSR_START 0x08
7   #define TWSR_REPEATED_START 0x10
8
9   //status codes for Master Transmitter mode (MT)
10  //status indicating address packet successfully sent
11  #define TWSR_MT_SLA_ACK 0x18
12  #define TWSR_MT_SLA_NACK 0x20
13  //status indicating data packet successfully sent
14  #define TWSR_MT_DATA_ACK 0x28
15  #define TWSR_MT_DATA_NACK 0x30
16
17  //status codes for Master Receiver mode (MR)
18  //status indicating address packet successfully sent
19  #define TWSR_MR_SLA_ACK 0x40
20  #define TWSR_MR_SLA_NACK 0x48
21  //status indicating data packet successfully sent
22  #define TWSR_MR_DATA_ACK 0x50
23  #define TWSR_MR_DATA_NACK 0x58
24
25  /*
26   * This function is used to initialize the I2c Module
27   */
28  void I2C_Init(void) {
29    //set te frequency of the Serial Data Clock
30      //no need of prescaler, so TWSR = 0
31    TWSR = 0x00;
32    TWBR = ((FREQ_CPU/FREQ_I2C)-16)/2;
33
34    //TWEN to enable theTWI module
35    TWCR = (1<<TWEN);
36  }
37
38  static void _I2C_Start(void) {
39    //TWINT to start the operation of the TWI
40    //TWEN to start TWI interface
41    //TWSTA to make the microcontroller master on the bus (sending a
        ↪ start condition)
42    TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWSTA);
43
44    //wait until current task has ended
45    while (!(TWCR & (1<<TWINT)));
46  }
47
48  /*
49   * This function is used to generate I2C Start Condition
50   *  Start Condition: SDA goes low when SCL is High
```

```
51     * @return: 1 if ok, 0 else
52     */
53    uint8_t I2C_Start(void) {
54      _I2C_Start();
55
56      //check if status code generated is 0x08
57      return (TWSR&0xF8) == TWSR_START;
58    }
59
60    /*
61     * This function is used to generate a repeated I2C Start Condition
62     *  Start Condition: SDA goes low when SCL is High
63     * @return: 1 if ok, 0 else
64     */
65    uint8_t I2C_Repeated_Start(void) {
66      _I2C_Start();
67
68      //check if status code generated is 0x10
69      return (TWSR&0xF8) == TWSR_REPEATED_START;
70    }
71
72    /*
73     * This function is used to generate I2C Stop Condition
74     *  Stop Condition: SDA goes High when SCL is High
75     */
76    void I2C_Stop(void) {
77      //TWSTO generates a STOP condition
78      TWCR = (1<< TWINT)|(1<<TWEN)|(1<<TWSTO);
79
80       //wait for a short time
81       _delay_us(10) ;
82    }
83
84    static void _I2C_Write(unsigned char data) {
85      //writes data (an 8-bit info) on TWDR
86      TWDR = data;
87
88      //TWINT to start the operation of the TWI
89      //TWEN to start TWI interface
90      TWCR = (1<< TWINT)|(1<<TWEN);
91
92      //wait until current task has ended
93      while (!(TWCR & (1 <<TWINT)));
94    }
95
96    /*
97     * This function is used to send a Device Address over the bus
```

```
98    * @param address_7bit : is the 7bit address of the slave device
99    *             you want to communicate with
100   * @param rw : rw flag to determine if master transmitter/receiver
          ↪ mode
101   *        if rw = 1, read
102   *        else, write
103   * @return: 1 if ok, 0 else
104   */
105  uint8_t I2C_SendAddress(uint8_t address_7bit, uint8_t rw) {
106    uint8_t addr = address_7bit<<1;
107    if(rw) //read
108      addr |= 1; //set rw bit to 1
109
110    _I2C_Write(addr);
111
112    //checks the status code generated
113    if(rw) //read, Master Receiver
114      return (TWSR&0xF8) == TWSR_MR_SLA_ACK;
115     else //write, Master Transmitter
116      return (TWSR&0xF8) == TWSR_MT_SLA_ACK;
117  }
118
119  /*
120   * This function is used to send a byte on SDA line using I2C
          ↪ protocol
121   *  8bit data is sent bit-by-bit on each clock cycle
122   *   MSB(bit) is sent first and LSB(bit) is sent at last
123   *  Data is sent when SCL is low
124   * @return: 1 if ok, 0 else
125   */
126  uint8_t I2C_Write(uint8_t data) {
127    _I2C_Write(data);
128
129    //checks the status code generated
130    return (TWSR&0xF8) == TWSR_MT_DATA_ACK;
131  }
132
133  /*
134   * This function is used to receive a byte on SDA line using I2C
          ↪ protocol
135   *  8bit data is received bit-by-bit each clock and finally packed
          ↪ into Byte
136   *   MSB(bit) is received first and LSB(bit) is received at last
137   */
138  #define ACK 1
139  #define NACK 0
140  uint8_t I2C_Read(uint8_t ack) {
```

```
141     //TWINT to start the operation of the TWI
142     //TWEN to start TWI interface
143     //TWEA controls the generation of the acknowledge pulse
144     // - if 1, then ACK pulse is generated
145     // - else, not
146     TWCR = (1<<TWINT)|(1<<TWEN)|(ack<<TWEA);
147
148     //wait current task ended
149     while (!(TWCR & (1 <<TWINT)));
150
151     return TWDR;
152   }
```

A.3 I2C Busy-Waiting Implementation

In the following section we will use the primitives defined previously, to implement the I2C protocol in busy-waiting mode.

```
1   #define WRITE 0
2
3   static void I2C_WritePreamble(uint8_t device_address, uint8_t
        ↪ device_reg) {
4     uint8_t err;
5
6     //master transmits the start condition
7     err = I2C_Start();
8     check_err(err, "[I2C_WritePreamble] Error transmitting the Start
          ↪ condition");
9
10    //master transmits the (7bits) device address and the write bit
          ↪ (0)
11    err = I2C_SendAddress(device_address, WRITE);
12    check_err(err, "[I2C_WritePreamble] Error transmitting the Device
          ↪  Address");
13
14    //master puts the address of the internal device register on the
          ↪ bus
15    err = I2C_Write(device_reg);
16    check_err(err, "[I2C_WritePreamble] Error transmitting the
          ↪ Internal Device Register");
17  }
18
19  void I2C_WriteRegister(uint8_t device_address, uint8_t device_reg,
        ↪ uint8_t data) {
20    uint8_t err;
```

```
21
22      I2C_WritePreamble(device_address, device_reg);
23
24      //master puts the data to be written on the bus
25      err = I2C_Write(data);
26      check_err(err, "[I2C_WriteRegister] Error transmitting the data
            ↪ to be written");
27
28      I2C_Stop();
29    }
30
31    void I2C_WriteNRegisters(uint8_t device_address, uint8_t
        ↪ device_reg_start, int n, uint8_t* data) {
32      uint8_t err;
33
34      I2C_WritePreamble(device_address, device_reg_start);
35
36      //master puts the data to be written on the bus, one byte at the
            ↪ time
37      int i;
38      for(i=0; i<n; i++) {
39        err = I2C_Write(data[i]);
40        check_err(err, "[I2C_WriteNRegisters] Error transmitting the
              ↪ data to be written");
41      }
42      I2C_Stop();
43    }
```

```
1   #define READ 1
2   #define WRITE 0
3
4   static void I2C_ReadPreamble(uint8_t device_address, uint8_t
        ↪ device_reg) {
5     uint8_t err;
6     /* MASTER TRANSMITTER MODE */
7     //master transmits the start condition
8     err = I2C_Start();
9     check_err(err, "[I2C_ReadPreamble] Error transmitting the Start
            ↪ condition");
10
11    //master transmits the 7-bits device address and the write bit
            ↪ (0)
12    err = I2C_SendAddress(device_address, WRITE);
13    check_err(err, "[I2C_ReadPreamble] Error transmitting the Device
            ↪ Address");
14
```

```
15      //master puts the address of the internal device register on the
            ↪ bus
16      err = I2C_Write(device_reg);
17      check_err(err, "[I2C_ReadPreamble] Error transmitting the
            ↪ Internal Device Register");
18
19      /* MASTER RECEIVER MODE */
20      //master transmits the start condition (restart)
21      err = I2C_Repeated_Start();
22      check_err(err, "[I2C_ReadPreamble] Error re-transmitting the
            ↪ Start condition");
23
24      //master transmits the (7bits) device address and the read bit
            ↪ (1)
25      err = I2C_SendAddress(device_address, READ);
26      check_err(err, "[I2C_ReadPreamble] Error transmitting the Device
            ↪ Address");
27    }
28
29    uint8_t I2C_ReadRegister(uint8_t device_address, uint8_t device_reg
          ↪ ) {
30      I2C_ReadPreamble(device_address, device_reg);
31
32      //master reads from the bus without outputting ACK (=> NACK)
33      unsigned char data = I2C_Read(0);
34
35      I2C_Stop();
36
37      return data;
38    }
39
40    void I2C_ReadNRegisters(uint8_t device_address, uint8_t
          ↪ device_reg_start, int n, uint8_t* data) {
41      I2C_ReadPreamble(device_address, device_reg_start);
42
43      //master reads from the bus outputting an ACK for each byte
44      int i;
45      for(i=0; i<n-1; i++) {
46        data[i] = I2C_Read(ACK);
47      }
48      //master reads from the bus without outputting ACK (=> NACK)
49      data[n-1] = I2C_Read(NACK);
50
51      I2C_Stop();
52    }
```

A.4 I2C Interrupt-Driven Implementation

In the following section we will present an interrupt-driven implementation of the I2C protocol. The TWI module in fact offers interrupt support, enabling the application software to carry on other operations during a TWI byte transfer.

When the TWINT Flag is asserted, the TWI has finished an operation and awaits application response. In this case, the TWI Status Register (TWSR) contains a value indicating the current state of the TWI bus. The application software can then decide how the TWI should behave in the next TWI bus cycle by manipulating the TWCR and TWDR Registers[3].

I2C Operations Handling We have decided to store the operations in a global queue that the ISR of the TWI module will automatically process.
Each operation has a set of attributes:

- the address of the slave device involved in the operation,
- a data buffer containing the data to be sent to the slave for write operations, or the data sent by the slave for read operations,
- the length of the buffer, indicating the number of bytes to write or read,
- the address of a post process function which will be fired at the end of the operation for any further processing to be done on the data in the buffer.
  In the implementation of the firmware, we have used the post process function in order to store the values returned by a read operation in the wanted memory locations.

```
1  typedef void(*PostProcessFunction_t)(uint8_t* buffer, uint8_t
       ↪ buflen);
2
3  typedef struct I2C_Operation {
4    struct I2C_Operation* next; //linked list of remaining ops
5    uint8_t device_address;//7 bits for device_address, 1 bit for
         ↪ read/write
6
7    //note: if write op, device_reg will be first byte in buffer
8      // if read op, buffer starts as empty
9    uint8_t* buffer; //full if write, empty if read
10   uint8_t buflen;  //length of the buffer (#data to read/write)
11   uint8_t bufpos;  //current pos in buffer
12
```

```
13      PostProcessFunction_t post_process_fn; //to store values if it
            ↪ was a read op
14    } I2C_Operation;
```

TWI Module Initialization The main difference between this and the previous (A.2) TWI initialization is that this time we must enable the generation of interrupts by the TWI module.

```
1   #define FREQ_CPU 16000000L //cpu clock spedd at 16MHz
2   #define FREQ_I2C 400000L  //i2c clock spped at 400KHz
3
4
5   //status indicating START condition successfully sent
6   #define TWSR_START 0x08
7   #define TWSR_REPEATED_START 0x10
8   #define TWSR_SLARW_ARB_LOST 0x38
9
10  //status codes for Master Transmitter mode (MT)
11  #define TWSR_MT_SLA_ACK 0x18
12  #define TWSR_MT_SLA_NACK 0x20
13  #define TWSR_MT_DATA_ACK 0x28
14  #define TWSR_MT_DATA_NACK 0x30
15
16  //status codes for Master Receiver mode (MR)
17  #define TWSR_MR_SLA_ACK 0x40
18  #define TWSR_MR_SLA_NACK 0x48
19  #define TWSR_MR_DATA_ACK 0x50
20  #define TWSR_MR_DATA_NACK 0x58
21
22  static volatile I2C_Operation* global_I2C_ops;
23
24  void I2C_Init(void) {
25    cli();
26
27    //set te frequency of the Serial Data Clock
28    TWSR = 0x00;
29    TWBR = ((FREQ_CPU/FREQ_I2C)-16)/2;
30
31    //Active internal pull-up resistors for SCL and SDA
32      //on atmega 2560 they are PD0 and PD1
33    PORTD |= 1|(1<<1);
34
35    // Disable slave mode
36    TWAR = 0;
37
38    //TWEN: TWI Enable, to start TWI interface
```

```
39    //TWIE : TWI Interrupt Enable in TWCR
40    TWCR = (1<<TWEN)|(1<<TWIE);
41    sei();
42  }
```

I2C Exposed Function The following is the primitive exposed to the CPU to enqueue I2C operations.

```
1   void I2C_Enqueue_Operation(uint8_t device_address_7bit, uint8_t rw,
      ↪  uint8_t n, uint8_t* data, PostProcessFunction_t
      ↪ post_process_fn) {
2     if( n <= 0 ) // no empty operations in the queue
3       return;
4
5     I2C_Operation* op = (I2C_Operation*)malloc(sizeof(I2C_Operation))
        ↪ ;
6
7     op->next = NULL;
8
9     op->device_address = (rw) ? (device_address_7bit<<1)|1 : (
        ↪ device_address_7bit<<1);
10
11    op->buffer = (uint8_t*)malloc(n*sizeof(uint8_t));
12    if( data != NULL )
13      memcpy(op->buffer, data, n);
14    op->buflen = n;
15    op->bufpos = 0;
16
17    op->post_process_fn = post_process_fn;
18
19    // updates on global list of I2C_Operations will be executed
        ↪ atomically
20    ATOMIC_BLOCK(ATOMIC_RESTORESTATE){
21      if( global_I2C_ops == NULL ) {
22        //if there was no pending operation,
23          //appends current one
24        global_I2C_ops = op;
25          //and schedules it immediately
26        TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWIE)|(1<<TWSTA);
27      } else {
28        I2C_Operation* aux = global_I2C_ops;
29        while( aux->next != NULL)
30          aux = aux->next;
31        aux->next = op;
32      }
33    }
```

```
34  }
```

Interrupt Service Routine The following ISR reads the value in the TWSR register and according to its value it decides how to proceed. The possible application software responses are based on the board's datasheet[3].

```
1   ISR(TWI_vect) {
2     //if there is no global I2C_Operation, exits
3     if( global_I2C_ops == NULL )
4       return;
5
6     switch( TWSR & 0xF8 ) { //TWI Status
7       //a START (or a repeated START) condition has been transmitted
8       case TWSR_START:
9       case TWSR_REPEATED_START:
10        //send address_7bit + read_bit
11        TWDR = global_I2C_ops->device_address;
12        //resets bufpos (should already be 0)
13        global_I2C_ops->bufpos = 0;
14        TWCR = (1<< TWINT)|(1<<TWEN)|(1<<TWIE);
15        break;
16
17      //arbitration lost in SLA+R/W or NACK bit
18      case TWSR_SLARW_ARB_LOST:
19        //start condition will be retransmitted
20        TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWIE)|(1<<TWSTA);
21        break;
22
23    /* WRITE operation, Master Transmitter Mode */
24
25      //SLA+W has been transmitted; ACK has been received
26      case TWSR_MT_SLA_ACK:
27        //sends first byte (device reg)
28        TWDR = global_I2C_ops->buffer[global_I2C_ops->bufpos];
29        global_I2C_ops->bufpos++;
30        TWCR = (1<< TWINT)|(1<<TWEN)|(1<<TWIE);
31        break;
32
33      // SLA+W has been transmitted; NOT ACK has been received
34      case TWSR_MT_SLA_NACK:
35        //STOP condition followed by a START condition will be
              ↪ transmitted
36          //and TWSTO Flag will be reset
37        TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWIE)|(1<<TWSTA)|(1<<TWSTO);
38        break;
39
```

```
40      // data byte has been transmitted; ACK has been received
41      case TWSR_MT_DATA_ACK:
42        //sends next byte
43        if( global_I2C_ops->bufpos < global_I2C_ops->buflen ) {
44          TWDR = global_I2C_ops->buffer[global_I2C_ops->bufpos];
45          global_I2C_ops->bufpos++;
46          TWCR = (1<< TWINT)|(1<<TWEN)|(1<<TWIE);
47          break;
48        } else {
49          //calls post-processing function
50          if( global_I2C_ops->post_process_fn != NULL )
51            (*(global_I2C_ops->post_process_fn))(global_I2C_ops->
                ↪ buffer, global_I2C_ops->buflen);
52
53          goto next_operation;
54        }
55
56      case TWSR_MT_DATA_NACK:
57        //STOP condition followed by a START condition will be
              ↪ transmitted
58          //and TWSTO Flag will be reset
59        TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWIE)|(1<<TWSTA)|(1<<TWSTO);
60        break;
61
62    /* READ operation, Master Receiver Mode */
63
64      // SLA+R has been transmitted; ACK has been received
65      case TWSR_MR_SLA_ACK:
66        //if just one read, ACK pulse not generated
67        if (global_I2C_ops->buflen == 1)
68          TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWIE);
69        else  //else ACK is generated
70          TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWIE)|(1<<TWEA);
71        break;
72
73      // SLA+R has been transmitted; NOT ACK has been received
74      case TWSR_MR_SLA_NACK:
75        //STOP condition followed by a START condition will be
              ↪ transmitted
76          //and TWSTO Flag will be reset
77        TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWIE)|(1<<TWSTA)|(1<<TWSTO);
78        break;
79
80      // data byte has been received; ACK has been returned
81      case TWSR_MR_DATA_ACK:
82        //read byte
83        global_I2C_ops->buffer[global_I2C_ops->bufpos] = TWDR;
```

```
84          global_I2C_ops->bufpos++;
85          //if this is last operation, ACK pulse not generated
86          if( global_I2C_ops->bufpos+1 == global_I2C_ops->buflen )
87            TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWIE);
88          else //else ACK is generated
89            TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWIE)|(1<<TWEA);
90          break;
91
92        // data byte has been received; NOT ACK has been returned
93        case TWSR_MR_DATA_NACK:
94          //read byte
95          global_I2C_ops->buffer[global_I2C_ops->bufpos] = TWDR;
96          global_I2C_ops->bufpos++;
97          //calls post-processing function
98          if( global_I2C_ops->post_process_fn != NULL )
99            (*(global_I2C_ops->post_process_fn))(global_I2C_ops->buffer,
                ↪ global_I2C_ops->buflen);
100
101         goto next_operation;
102
103       default:
104         //it should never get here, we treated all possible Status
              ↪ Codes
105         exit(1);
106     }
107     return;
108
109     next_operation: {
110       I2C_Operation* ended_op = global_I2C_ops;
111         //forwards to next transaction
112       global_I2C_ops = global_I2C_ops->next;
113
114       if( global_I2C_ops != NULL ) //repeated start
115         TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWIE)|(1<<TWSTA);
116       else  //if there is no next operation, transmits a stop
117         TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWIE)|(1<<TWSTO);
118
119       free(ended_op->buffer);
120       free(ended_op);
121       return;
122     }
123   }
```

Appendix B

UART Communication Implementation

In this appendix we present two implementations of UART communication. The first one is a busy-waiting implementation, while the second one is an interrupt-driven implementation. These implementations are specific for the board used[3].

B.1 UART Control Registers

On the board used, the UART serial interface is controlled by the following set of registers.

- UBRRn (UART Baud Rate Register) is used to control the frequency of the byte transmission.
- UCSRn (UART Control and Status Register) allows to configure other aspects of the port and to poll if the UART is ready to transmit data or has received some.
- UDRn (UART Data Register) is the data register containing the data being sent and received.

B.2 UART Busy-Waiting Implementation

In this first section we show an easy busy-waiting implementation of UART communication. Before starting any operation in fact the CPU polls until the end of the previous transmission.

```
1  //baud rate at which we want the serial to communicate
2  #define UART_BAUD_RATE 38400
```

```
3
4   //baud rate = F_CPU / (16*(UBRR+1))
5   #define UBRR ( (F_CPU/16)/UART_BAUD_RATE - 1 )
6
7   /*
8    * initializes the UART at UART_BAUD_RATE baud rate
9    */
10  void UART_Init(void){
11    //sets baud rate
12    UBRR0H = (uint8_t)((UBRR)>>8);
13    UBRR0L = (uint8_t)UBRR;
14
15    //enables receiver and transmitter
16    UCSR0B = (1<<RXEN0)|(1<<TXEN0);
17
18    //sets data packet format: 8data, 2stop bit
19    UCSR0C = (1<<UCSZ01)|(1<<UCSZ00);
20  }
21
22  /*
23   * transmits a char through the UART module.
24   *  It waits till previous char is transmitted (i.e. until UDRE is
       ↪ set),
25   *  then the new byte to be transmitted is loaded into UDR.
26   */
27  void UART_TxByte(uint8_t data) {
28    //waits until the transmit buffer is empty
29    while ( !( UCSR0A & (1<<UDRE0) ) );
30
31    //puts data into buffer, sends the data
32    UDR0 = data;
33  }
34
35  /*
36   * receives a char from the UART module.
37   *  It waits till a char is received (i.e.till RXC is set),
38   *  then returns the received char.
39   */
40  uint8_t UART_RxByte(void) {
41    //waits for the data to be received
42    while ( !(UCSR0A & (1<<RXC0) ) );
43
44    //gets and returns received data from buffer
45    return UDR0;
46  }
```

B.3 UART Interrupt-Driven Implementation

In the following section we will present an interrupt-driven implementation of UART communication. The UART module in fact offers interrupt support, enabling the application software to carry on other operations during UART byte transfers.

The UART module will fire a Receive Interrupt (RX) when a new byte has been received, and a Data Register Empty Interrupt when the transmit buffer is ready to receive new data.

UART Data Handling We have decided to store the data to be transmitted and the data received in two global buffers automatically populated by the ISRs.

```
1  #define UART_BUFFER_SIZE 255
2
3  typedef struct UART {
4    uint8_t tx_buffer[UART_BUFFER_SIZE];
5    volatile uint8_t tx_start;
6    volatile uint8_t tx_end;
7    volatile uint16_t tx_size;
8
9    uint8_t rx_buffer[UART_BUFFER_SIZE];
10   volatile uint8_t rx_start;
11   volatile uint8_t rx_end;
12   volatile uint16_t rx_size;
13
14   uint16_t baudrate;
15 } UART;
```

UART Module Initialization The main difference between this and the previous (B.2) UART initialization is that this time we must enable the generation of interrupts by the UART module.

```
1  // Baud Rate at which we want the serial to communicate
2  #define UART_BAUD_RATE 57600
3
4  // Baud Rate = F_CPU / (16*(UBRR+1))
5  #define UBRR ( (F_CPU/16)/UART_BAUD_RATE - 1 )
6
7  static UART uart0;
8
9  static void UART_buffers_init(void) {
10   //initialize rx_buffer
11   uart0.rx_start = 0;
12   uart0.rx_end = 0;
```

```
13      uart0.rx_size = 0;
14
15      //initialize tx_buffer
16      uart0.tx_start = 0;
17      uart0.tx_end = 0;
18      uart0.tx_size = 0;
19    }
20
21    /*
22     * UART_Init :
23     *  initializes the UART at UART_BAUD_RATE baud rate
24     */
25    void UART_Init(void){
26      //initializes the global buffers
27      UART_buffers_init();
28
29      uart0.baudrate = UART_BAUD_RATE;
30
31      //sets baud rate
32      UBRR0H = (uint8_t)((UBRR)>>8);
33      UBRR0L = (uint8_t)UBRR;
34
35      //enables receiver and transmitter
36        // RXCIE0 : RX Complete Interrupt Enable. Set to allow receive
              ↪ complete interrupts.
37      UCSR0B = (1<<RXEN0)|(1<<TXEN0)|(1<<RXCIE0);
38      //sets frame format: 8data, 2stop bit
39      UCSR0C = (1<<UCSZ01)|(1<<UCSZ00);
40
41      sei();
42    }
```

UART Exposed Functions The followings are the exposed functions the CPU can use to enqueue data to be transmitted or get data that has been received.

```
1     /*
2      * UART_TxByte :
3      * puts a byte in the transmit buffer
4      *  then as soon as possible it will be transmitted
5      *   through the UART module
6      */
7     void UART_TxByte(uint8_t data) {
8       //loops until there is some space in the buffer
9       while (uart0.tx_size >= UART_BUFFER_SIZE);
10
11      ATOMIC_BLOCK(ATOMIC_RESTORESTATE){
```

```
12        uart0.tx_buffer[uart0.tx_end] = data;
13        uart0.tx_end++;
14
15        //if we reached the end of the circular buffer,
16          //we go back to the beginning
17        if (uart0.tx_end >= UART_BUFFER_SIZE)
18          uart0.tx_end = 0;
19
20        uart0.tx_size++;
21      }
22
23      //activates buffer empty interrupt
24      UCSR0B |= (1<<UDRIE0);
25    }
26
27    /*
28     * UART_RxByte :
29     * gets a byte from the receive buffer
30     *  @return : the received byte.
31     */
32    uint8_t UART_RxByte(void) {
33      //loops until there is some data in the buffer
34      while(uart0.rx_size == 0);
35
36      uint8_t data;
37
38      ATOMIC_BLOCK(ATOMIC_RESTORESTATE){
39        data = uart0.rx_buffer[uart0.rx_start];
40        uart0.rx_start++;
41
42        //if we reached the end of the circular buffer,
43          //we go back to the beginning
44        if (uart0.rx_start >= UART_BUFFER_SIZE)
45          uart0.rx_start = 0;
46
47        uart0.rx_size--;
48      }
49
50      return data;
51    }
```

Interrupt Service Routines The first ISR is fired upon receipt of a byte, while the second one is fired when the transmit buffer is ready to transmit new data.

```
1   //Receive Interrupt Handler
2   ISR(USART0_RX_vect) {
```

```
3     //reads the new byte from UDR0,
4       //and writes it in rx_buffer
5
6     if (uart0.rx_size < UART_BUFFER_SIZE) {
7       uart0.rx_buffer[uart0.rx_end] = UDR0;
8       uart0.rx_end++;
9
10      //if we reached the end of the circular buffer,
11        //we go back to the beginning
12      if (uart0.rx_end >= UART_BUFFER_SIZE)
13        uart0.rx_end = 0;
14
15      uart0.rx_size++;
16    }
17  }
18
19  //Data Register Empty (UDRE0) Flag
20    //indicates whether the transmit buffer
21    //is ready to receive new data
22  ISR(USART0_UDRE_vect) {
23    //if the transmit buffer is empty
24      //disables buffer empty interrupts
25    if( uart0.tx_size == 0 )
26      UCSR0B &= ~(1<<UDRIE0);
27    else {
28      //gets a byte from the tx_buff
29        //adn writes it on UDR0
30      UDR0 = uart0.tx_buffer[uart0.tx_start];
31      uart0.tx_start++;
32
33      //if we reached the end of the circular buffer,
34        //we go back to the beginning
35      if (uart0.tx_start >= UART_BUFFER_SIZE)
36        uart0.tx_start = 0;
37
38      uart0.tx_size--;
39    }
40  }
```

Appendix C

Differential Drive Implementation

In this appendix we present the implementation of the Differential Drive localization technique.

```
1   static const float sin_taylor_coeffs[] = {1., 0., -1./6., 0.,
        ↪ 1./120., 0.};
2   static const float cos_taylor_coeffs[] = {0., 1./2., 0., -1./24.,
        ↪ 0., 1./720.};
3
4   static void computeThetaTerms(float* sin_theta_over_theta, float*
        ↪ one_minus_cos_theta_over_theta, float theta) {
5     *sin_theta_over_theta = 0;
6     *one_minus_cos_theta_over_theta = 0;
7     float theta_exp = 1;
8
9     uint8_t i;
10    for(i=0; i < (sizeof(sin_taylor_coeffs)/sizeof(float)); i++) {
11      *sin_theta_over_theta += sin_taylor_coeffs[i]*theta_exp;
12      *one_minus_cos_theta_over_theta += cos_taylor_coeffs[i]*
            ↪ theta_exp;
13
14      theta_exp *= theta;
15    }
16  }
17
18  void Encoder_OdometryUpdate(void) {
19    int32_t encs_cnt[NUM_ENCODERS];
20    // reads encoders' global counters in encs_cnt
21
22    // calculates how many rotations where done in the last 10 ms
23    int32_t left_ticks = encs_cnt[0] - encs_cnt_previous[0];
```

```
24      int32_t right_ticks = encs_cnt[1] - encs_cnt_previous[1];
25
26      //if encoders didn't detect any move,
27        //robot didn't move
28      if( left_ticks || right_ticks ) {
29        //updates previous values stored
30        encs_cnt_previous[0] = encs_cnt[0];
31        encs_cnt_previous[1] = encs_cnt[1];
32
33        //gets movement in meters
34        float delta_l = left_ticks * (float)METERS_PER_ENCODER_TICK;
35        float delta_r = right_ticks * (float)METERS_PER_ENCODER_TICK;
36
37        float delta_plus = delta_r + delta_l;
38        float delta_minus = delta_r - delta_l;
39
40        float delta_theta = delta_minus / BASE_LEN;
41
42        float sin_dtheta_over_dtheta, one_minus_cos_dtheta_over_dtheta;
43        computeThetaTerms(&sin_dtheta_over_dtheta, &
              ↪ one_minus_cos_dtheta_over_dtheta, delta_theta);
44
45        float delta_x = .5 * delta_plus * sin_dtheta_over_dtheta;
46        float delta_y = .5 * delta_plus *
              ↪ one_minus_cos_dtheta_over_dtheta;
47
48        //update global odometry
49          //odom_theta is the orientation angle until now
50        float sin_theta = sin(odom_theta);
51        float cos_theta = cos(odom_theta);
52
53        odom_x += delta_x*cos_theta - delta_y*sin_theta;
54        odom_y += delta_x*sin_theta + delta_y*cos_theta;
55        odom_theta += delta_theta;
56
57        //update global current volocities
58        translational_velocity = .5 * delta_plus / delta_time;
59        rotational_velocity = delta_theta / delta_time;
60      }
61    }
```

Appendix D

Inertial Navigation Implementation

In this appendix we present the implementation of the Inertial Navigation technique.

```
1   #define IMU_ANG_VEL_THRESHOLD 0.4 //DPS
2   #define IMU_TRASL_ACC_THRESHOLD 0.05 //G-Forces
3   #define G_FORCE_ACCEL_G 1 //g[G] = 1
4   #define G_FORCE_ACCEL_M_S2 9.8 //accel[m/s^2] = accel[G-Force]*9.8
5
6   #define STOP_BREAKING_TRESHOLD 2
7   #define STOP_ZERO_ACCEL_OUTLIER_FILTER 5
8
9   // HIGH-PASS FILTER for translational accelerations
10    // (moreover we also convert from an acceleration in G-forces to
        ↪ an acceleration in m/s^2)
11  translational_acceleration_x_axis = (fabs(accel_x) >
        ↪ IMU_TRASL_ACC_THRESHOLD) ? accel_x*G_FORCE_ACCEL_M_S2 : 0.;
12  translational_acceleration_y_axis = (fabs(accel_y) >
        ↪ IMU_TRASL_ACC_THRESHOLD) ? accel_y*G_FORCE_ACCEL_M_S2 : 0.;
13    //since we are in 2D, we'll always have a gravitational
        ↪ acceleration componenet on the z-axis, for which we must
        ↪ compensate
14  translational_acceleration_z_axis = (fabs(accel_z - G_FORCE_ACCEL_G
        ↪ ) > IMU_TRASL_ACC_THRESHOLD) ? accel_z*G_FORCE_ACCEL_M_S2 :
        ↪ 0.;
15
16  // STOP MOTION DETECTION x axis
17  if ( translational_acceleration_x_axis > 0. ) {
18    total_time_pos_accel_x++;
19    curr_time_zero_accel_x = 0;
20  } else if ( translational_acceleration_x_axis < 0. ) {
21    total_time_neg_accel_x++;
```

```
22      curr_time_zero_accel_x = 0;
23    } else //accel_x_axis == 0.
24      curr_time_zero_accel_x++;
25
26    if( curr_time_zero_accel_x > STOP_ZERO_ACCEL_OUTLIER_FILTER &&
          ↪ curr_time_zero_accel_x >= abs( total_time_pos_accel_x -
          ↪ total_time_neg_accel_x) * STOP_BREAKING_TRESHOLD) {
27      //after it has had approximately
28        //the same time of positive and negative acceleration
29        //we assume it is in MOTIONLESS state
30      translational_acceleration_x_axis = 0.;
31      translational_velocity_x_axis = 0.;
32      total_time_pos_accel_x = 0;
33      total_time_neg_accel_x = 0;
34    }
35
36    // STOP MOTION DETECTION y axis
37    if ( translational_acceleration_y_axis > 0. ) {
38      total_time_pos_accel_y++;
39      curr_time_zero_accel_y = 0;
40    } else if ( translational_acceleration_y_axis < 0. ) {
41      total_time_neg_accel_y++;
42      curr_time_zero_accel_y = 0;
43    } else //accel_x_axis == 0.
44      curr_time_zero_accel_y++;
45
46    if( curr_time_zero_accel_y > STOP_ZERO_ACCEL_OUTLIER_FILTER &&
          ↪ curr_time_zero_accel_y >= abs( total_time_pos_accel_y -
          ↪ total_time_neg_accel_y) * STOP_BREAKING_TRESHOLD) {
47      //after it has had approximately
48        //the same time of positive and negative acceleration
49        //we assume it is in MOTIONLESS state
50      IMU_OdometryController.odometry_status.
            ↪ translational_acceleration_y_axis = 0.;
51      IMU_OdometryController.odometry_status.
            ↪ translational_velocity_y_axis = 0.;
52      IMU_OdometryController.odometry_status.total_time_pos_accel_y =
            ↪ 0;
53      IMU_OdometryController.odometry_status.total_time_neg_accel_y =
            ↪ 0;
54    }
55
56    // integrate in the local motion vector (local reference frame)
57    // dx = v*t + .5*a*t*t = (v + .5*a*t)*t
58    float delta_x_local = (translational_velocity_x_axis + .5*
          ↪ translational_acceleration_x_axis*delta_time) * delta_time;
```

```
59  float delta_y_local = (translational_velocity_y_axis + .5*
        ↪ translational_acceleration_y_axis*delta_time) * delta_time;
60    //since we are in 2D we can assume delta_z to be 0
61  float delta_z_local = 0.0;//(translational_velocity_z_axis + .5*
        ↪ translational_acceleration_z_axis*delta_time) * delta_time;
62
63  // update global odom x, y, z
64  float sin_yaw = sin(imu_yaw);
65  float cos_yaw = cos(imu_yaw);
66
67  imu_odom_x += delta_x_local*cos_yaw - delta_y_local*sin_yaw,
68  imu_odom_y += delta_x_local*sin_yaw + delta_y_local*cos_yaw;
69  imu_odom_z += delta_z_local;// 0.0
70
71  // integrate in translational velocities
72  translational_velocity_x_axis += translational_acceleration_x_axis
        ↪ * delta_time;
73  translational_velocity_y_axis += translational_acceleration_y_axis
        ↪ * delta_time;
74  translational_velocity_z_axis += translational_acceleration_z_axis
        ↪ * delta_time;
75
76  // high-pass filter for rotational velocities
77  rotational_velocity_z_axis = (fabs(gyro_z) > IMU_ANG_VEL_THRESHOLD)
        ↪  ? gyro_z : 0.;
78  rotational_velocity_y_axis = (fabs(gyro_y) > IMU_ANG_VEL_THRESHOLD)
        ↪  ? gyro_y : 0.;
79  rotational_velocity_x_axis = (fabs(gyro_x) > IMU_ANG_VEL_THRESHOLD)
        ↪  ? gyro_x : 0.;
80
81  // integrate in local orientation change
82  float delta_yaw_deg = rotational_velocity_z_axis * delta_time;
83  float delta_pitch_deg = rotational_velocity_y_axis * delta_time;
84  float delta_roll_deg = rotational_velocity_x_axis * delta_time;
85
86  float delta_yaw_rad = delta_yaw_deg * M_PI / M_180;
87  float delta_pitch_rad = delta_pitch_deg * M_PI / M_180;
88  float delta_roll_rad = delta_roll_deg * M_PI / M_180;
89
90  //update global yaw, roll, pitch
91  imu_yaw += delta_yaw_rad;
92  imu_pitch += delta_pitch_rad;
93  imu_roll += delta_roll_rad;
```

References

[1] S. Huang and G. Dissanayake, "Robot localization: An introduction", *John Wiley & Sons, Inc.*, 2016. [Online]. Available: https://onlinelibrary.wiley.com/doi/full/10.1002/047134608X.W8318.

[2] O. J. Woodman, "An introduction to inertial navigation", *Tech. Report UCAM-CL-TR-696, University of Cambridge, Computer Laboratory*, 2007. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-696.pdf.

[3] *Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V Datasheet*. Atmel. [Online]. Available: https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf.

[4] *KY-040 rotary encoder Datasheet*. Handson Technology. [Online]. Available: https://www.handsontec.com/dataspecs/module/Rotary%20Encoder.pdf.

[5] *PS-MPU-9250A-01, MPU-9250 Product Specification*. InvenSense Inc., 2016. [Online]. Available: https://www.invensense.com/wp-content/uploads/2015/02/PS-MPU-9250A-01-v1.1.pdf.

[6] *RM-MPU-9250A-00, MPU-9250 Register Map and Descriptions*. InvenSense Inc., 2015. [Online]. Available: https://www.invensense.com/wp-content/uploads/2015/02/RM-MPU-9250A-00-v1.6.pdf.

[7] A. P. David Tedaldi and E. Menegatti, "A robust and easy to implement method for imu calibration without external equipments", *In IEEE International Conference on Robotics and Automation (ICRA 2014)*, 2014. [Online]. Available: http://www.dis.uniroma1.it/~pretto/papers/tpm_icra2014.pdf.

[8] A. Pretto and G. Grisetti, "Calibration and performance evaluation of low-cost imus", *In Proceedings of the 20th IMEKO TC4 International Symposium and the 18th International Workshop on ADC Modelling and Testing*, 2014. [Online]. Available: http://www.dis.uniroma1.it/~pretto/papers/pg_imeko2014.pdf.

[9] "Differential drive robots", [Online]. Available: http://www.cs.columbia.edu/~allen/F15/NOTES/icckinematics.pdf.

[10] D. P. Anderson, "Imu odometry", 2006. [Online]. Available: http://seattlerobotics.org/encoder/200610/Article3/IMU%20Odometry,%20by%20David%20Anderson.htm.