Smart-Robot Challenge
Final Report
Group A4

June 19, 2019

Jeroen van Ammers *4818369*
David Denekamp *4900561*
Yunus Emre Döngel *4850629*
Reinier van der Leer *4947703*
Aysen Otman *4759044*
Vera Pauptit *4698517*
Raquel Simon *4903501*
Gabriel Yousef *4922085*

TU Delft

Preface

This report has been written for an educational project during the first year of the bachelor Electrical Engineering and covers only the first part of the project. It contains a detailed description of the development of a smart robot, the basic hardware of which was already provided. The aim of this project is to let the robot move on a grid from one point to another as fast as possible, while avoiding mines, which are little pieces of metal located at arbitrary positions on the board.

During this project, a group of eight people will work on the robot twice a week in four-hour sessions over a period of 9 weeks. These sessions take place in the Tellegen Hall. The group will be guided during sessions by TU Delft staff and will use the study material that was put together for this project.

The underlying aim of this project is to give students the opportunity to gain experience with designing, programming, report-writing, team-working and, more importantly, to implement the theory that was gained in the courses Digital Systems A, Amplifiers and Instrumentation and Digital Systems B.

Group A4
Delft, May 2019

Contents

1 Introduction

Smart robots can be found *everywhere*: the health sector, industrial sector, sports and the military and so on; it is almost impossible to imagine a life without them. Electrical engineers are the driving force behind these devices. For EPO 2 all first years students electrical engineering should do a project. During this project such a smart robot will be designed and programmed.
The robot should be able to travel over a grid as fast as possible and avoid mines: little pieces of metal that block the track. While the decision capabilities of the robot are designed in VHDL, it is implemented on a FPGA board (BASYS 2 with SPARTAN 3E). The objective of this project is not only to design a robot that can follow a line and avoid mines, but also to build the sensors for detecting the mines.

1.1. Pojects goals

The goal of the project is being able to program a robot such that it follows a track and avoid mines detected on the track. The programming of the robot should be done in VHDL and in C. Another goal of the project is being able to design a mine sensor that is able to detect mines.

Project Approach

In order to fulfill the learning goals mentioned before and aid the progress of the team, a project plan was composed. This plan defines the structure of the project and will be explained in more detail in the report .
The project plan includes quantifiable milestones that divide the project into several tasks; each week has a specific task assigned to it. The tasks for the first four weeks, are straight-forward and the same for the whole team, because they are guided by the Manual. [1] After the four weeks the group needs to at least fulfill the first two Challenges. The third challenge is optional. These three challenges are:

- Challenge A: Find the shortest route, the robot should be able to follow lines. The robot should also be able to communicate with the PC, since it will be controlled through XBee.
- Challenge B: Avoid mines, the robot should be able to run and detect mines.
- Challenge C: Find all the mines throughout the maze, without running over them, keeping their location, and then finding one additional mine.

The tasks were divided over the weeks as follows:

- Week 4.1: Exploring the hardware of the robot
  This task consists of exploring the different hardware components of the robot and performing several measurements on them to understand their functionality. The knowledge gained in this part will later be implemented in the design procedure (Chapter 2 and 3 from the Manual).
- Week 4.2: Top-level architecture The goal of this week is to design the top-level architecture of the robot. The aim during the first four weeks is to have the robot to follow a line. The top-level architecture of the line-follower describes the entities of the different components and how they are connected (Chapter 4 from the Manual).

- Week 4.3: Time base, FSMs for input buffer and motor driver

  During this week, the time base, input buffer and motor controller will be designed. These are components of the line-follower, each responsible for a different function. The time base offers the system an implicit way to keep track of time; the input buffer makes sure input signals are in sync with the clock and the motor controller controls the servomotors (Chapter 5 and 6 from the Manual).

- Week 4.4: Controller FSM
  The controller has to be designed in the fourth week. This line-follower component receives the sensor's input and based on that, instructs the motor driver to run the motors (Chapter 7 from the Manual).

- Week 4.6 and further:
  From here we divided in three groups. A minesensor group, a VHDL group and a C group. During the last few weeks we try to fulfill all the goals of the project.

The report consists of 12 chapters. The first 7 chapters are the chapters guided by the manual. The 8th chapter descibed the communication between the computer and the FPGA. The 9th chapters explains the design of the mine sensor. Chapter 10 describes the VHDL mine sensor code. Chapter 11 the control software. Eventually the last chapter shows the consludion and recommendations.

2

Measurements on the robot

To design the robot in the optimal way it is important to get to know how the robot works first. This robot has two key elements: the optical sensor and the servo motors. To perform some experiments, the robot is programmed to be used as a range finder, and several measurements are done.

2.1. The optical sensor

The optical sensor has three light-sensitive sensors, which are placed in a row. These sensors consist of an infra-red and a LED phototransistor. When the sensors are hovering above a black surface, which means little to no light reflection, the conductivity of the phototransistor becomes very low. When over a white surface, which means good light reflection, the conductivity becomes high. While converting the signal from the analog to the digital domain, an inverting Schmitt-trigger is used to buffer the output signal. While over a black surface, the circuit will set a '0' as the output and while over a white surface, the output is set to '1'. A way to make sure the correct output is given, there are LED-lights which light up when the output of the sensor is set to '1'. [1]

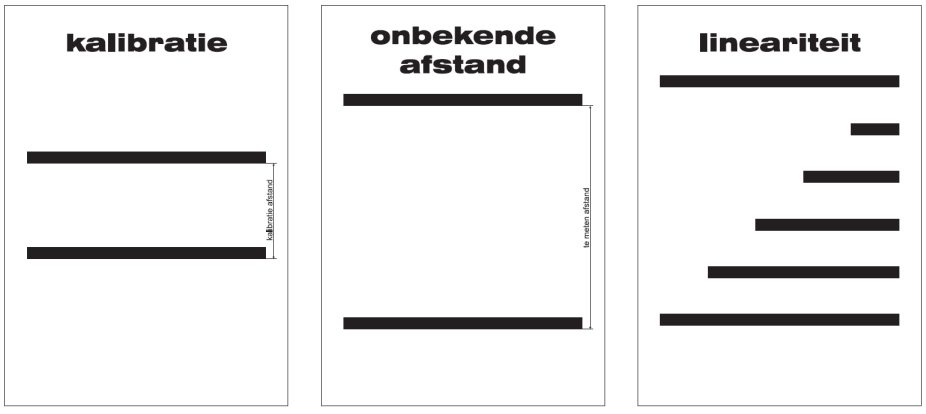

Figure 2.1: Test patterns for the robot-distance meter

2.2. The servo motor

The robot has two wheels driven by servomotors, which are controlled by a pulse-width modulating signal: a PWM signal. PWM works as follows: every period a pulse is sent with a certain width, which determines the behaviour of the motor:

- If the pulse-width is 1.5 ms, the wheel stops rotating
- If it is less than 1.5 ms, the wheel rotates counter-clockwise
- If it is more than 1.5 ms, the wheel rotates clockwise
- The more the pulse-width differs from the 1.5 ms, the faster the wheel rotates

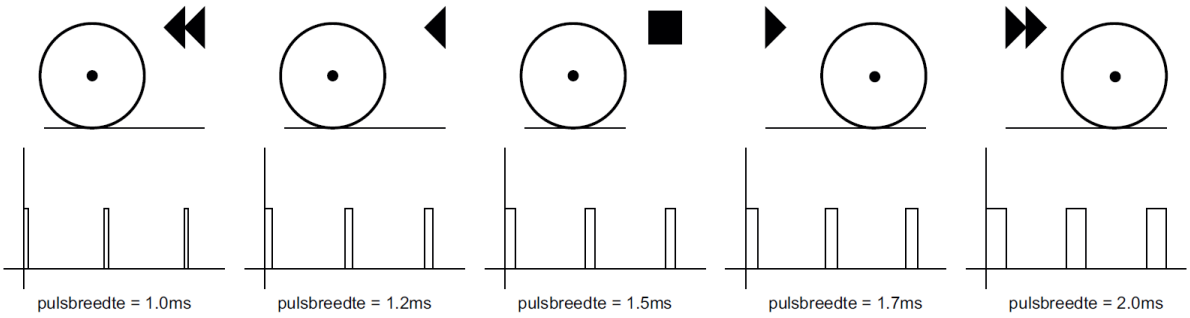

Figure 2.2: pulse width controls motor direction

2.3. The robot range-finder

The robot can also be used as a tool to measure distances. The measurement procedure is indirect, which means a calibration is needed: The system measures the time it takes for the robot to go from the first passing mark, a black stripe, to the second passing mark, also a black stripe, as portayed in Figure 2.1. The measured time is then multiplied with the average speed to get the distance travelled.
First a calibration measurement is needed. The distance between the first and the second passing mark is known to be 9.72 cm, and the time it takes the robot to cross both passing marks is measured. This measurement is repeated ten times. The results are shown in Table 2.1. The average time it takes the robot to go from the first to the second passing mark was calculated to be 0.839 seconds. With a simple formula the average speed of the robot was calculated, as seen in Equation 2.1. If the average speed is known, the robot can "measure" distances.

Measurement	1	2	3	4	5	6	7	8	9	10
Time (s)	0.83	0.83	0.84	0.83	0.85	0.84	0.83	0.86	0.83	0.85

Table 2.1: The time in seconds it takes to travel from the first to the second passing mark

$$\begin{aligned} v_{avg} &= \frac{x}{t_{avg}} \\ v_{avg} &= \frac{9.72}{0.839} \\ v_{avg} &= 11.59\,\mathrm{cm\,s^{-1}} \end{aligned} \tag{2.1}$$

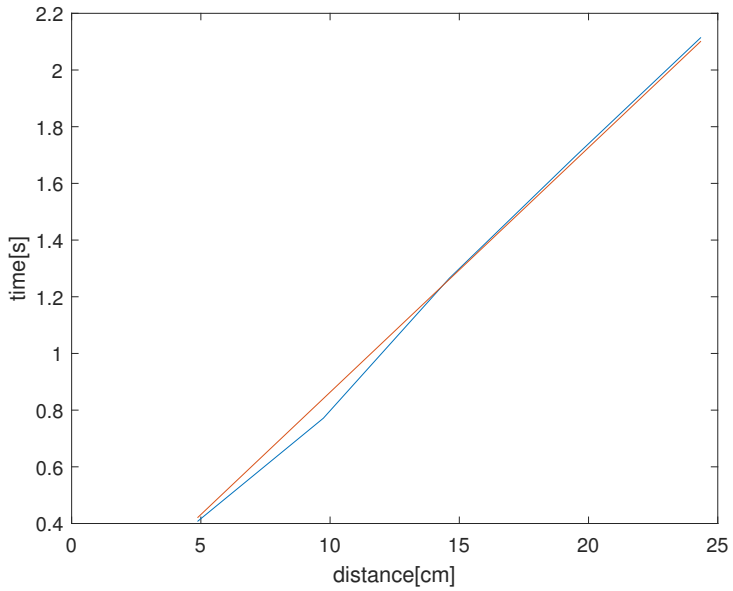

Figure 2.3: Speed of the robot over a distance

The previous measurements were done with a robot that was already moving, but to get the most information, different measurements are needed. The following measurements are done starting with a robot at rest. The test pattern of Figure 2.1(c) was used for these measurements. The results can be seen in Figure 2.3. This figure shows that the speed is not perfectly linear. The robot is not at the average speed right away. This difference in graphs is unpredictable. While it is not a significant difference, it should be taken into consideration when designing the robot.

3

Top-level description

The top level description is the highest programming level in the system of the robot. The input signals of this system are the sensor signals, the mine detector signal, the clock and the reset. The output signals are the PWM signals which control the motor. Between the input and output signals, the top-level is simply a black box that defines where all the signals in the system go. This box contains multiple components that each perform a specific task using these signals. The top-level describes how all these different components need to behave and how they are connected to each other. The global schematic of the top-level is shown in 3.1:

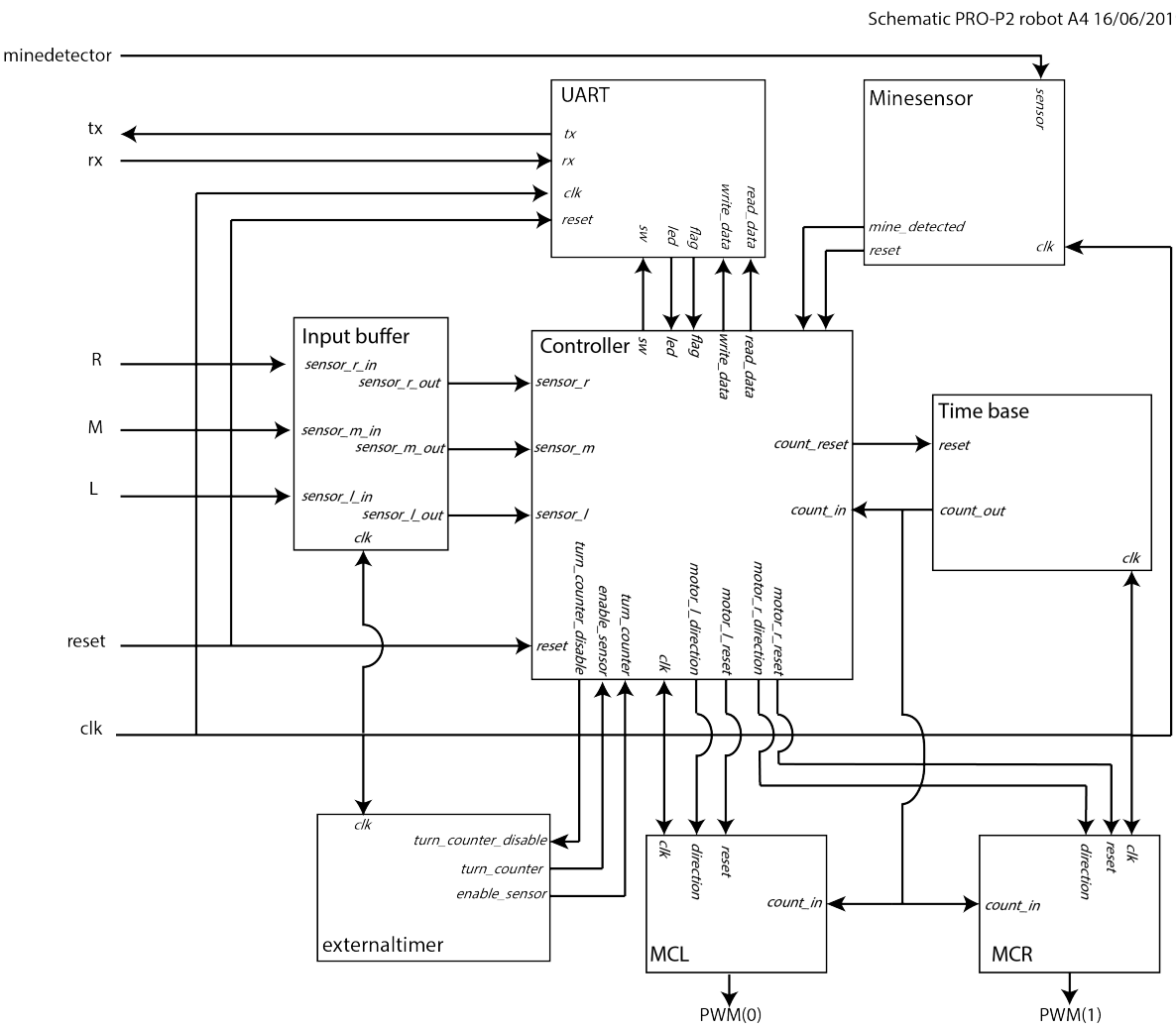

Figure 3.1: Schematics system

As seen in the figure, the left side has multiple inputs. The `R`,`M` and `L` stand for the sensor inputs; respectively the right, middle and left optical sensor. They are used to detect the reflectivity of the path that the robot is navigating on. On the top-left side the `tx` and `rx` input are shown. These inputs come from the uart module and are used to communicate with the computer. The last input signals are the reset and clock signal. The reset can be applied using a button or switch on the FPGA board, whereas the clock signal comes from the FPGA board directly. The outputs are the right motor pulse and the left motor pulse that drive the right and left motor respectively.

The figure above shows that there are multiple intermediate signals in the top-level description. One of these intermediate signals is the count signal between the timebase and controller that needs to be a `std_logic_vector`. To calculate the amount of bits in this vector, Equation 3.1 is used:

$$N_{\text{bits}} = \log_2(t_{\text{period}} * f_{\text{clock}}) \tag{3.1}$$

In this equation, N stands for the number of bits, `f` stands for the frequency and `t` for the time. The count signal has to count the time between two pulses, which is 20 ms. Using the equation for 20 ms results in an amount of 20 bits. Thus the count can be implemented as a `std_logic_vector(19 downto 0)` data type.

3.1. Design of the top-level description

The design of the top-level description is done in VHDL and simulations are made with the program Modelsim. The script starts with an entity, which has the following inputs: tx, rx, right, middle and left infrared

sensor, clock and the reset. The output signals are the left motor and the right motor output signals. The signals are all individually defined as a standard logic type (`std_logic`). If the sensor detects a white surface it will give a logic one and if it detects a black surface it will give a logic zero. The entity script can be seen in Appendix A.1.

After the entity of the top-level, the architecture had to be made. In the first part of the architecture, multiple components are described. All these components are needed for the robot to process signals from input to output. The components are the input buffer, controller, timebase, external timebase, the mine sensor and motor controller. The first component that is needed after the input of the sensors is the input buffer. This component has as inputs: `sensor_right`, `sensor_middle`, `sensor_left` and `clk`. The output signal is the `sensor_output` and is of the logic type (`std_logic_vector (2 downto 0)`). The vector is a three bits vector because the input buffer combines the output of the three sensors together in one vector. The code of the component input buffer can be seen in Appendix A.5.

When the signals are processed by the input buffer they are given as an input to the controller. The controller has as input signals: `reset`, `clk`, sensor inputs, uart inputs, `count_in` from timebase, count signal from external timebase and the input signal from the minesensor. The sensor input is implemented as a `std_logic_vector` because the input exists of 3 sensors which the input buffer puts in a vector. `Count_in` is realised as a `std_logic_vector`. The output signals are: count_reset, motor_l_reset, motor_r_reset -who are resetting the left and right motor respecively, motor_l_direction and motor_r_direction are both implemented as
`std_logic_vector`. The `sw` between the uart and controller is implemented as an 8-bits `std_logic_vector`. The Count_in is a vector of 20 bits according to equation 3.1. The code can be seen in Appendix A.9.

Multiple outputs of the controller are connected to the motor controllers. The motor control entity has generic parameters which specify the pulse length in clock ticks for the four motor "directions". Using two speeds for going forwards and backwards, allows for small directional changes, making the robot drive smoother along the line. Using a generic for the number of ticks, allows reusing the motorcontrol architecture for both motors while still being able to adjust their speed individually. There is a MCL (motorcontrol left) and a MCR (motorcontrol right) system. The MCL sends a signal to left motor and the MCR sends a signal to the right motor. The input signals are respectively: `clk`, `reset`, `direction` and `count_in`. The output signals is the `pwm` signal which is the input of the motor. The controller works with a clock signal and reset signal that needs to be reset every 20 ms. This is done by the timebase.

The time base has inputs: `clk` and `reset` and output `count`. The count is a (`std_logic_vector(19 downto 0)`) according to equation 3.1. The component time-base in the top-level description can be seen at Appendix A.3.

The mine sensor has the inputs: `clk` and `sensor`. The mine sensor will give an output signals to the controller. These signals are `mine_detected` and `reset`. They are standard `std_logic` signals because if a mine is detected, the mine detector component only gives a logic one.

The UART part has six inputs. These are `rx`, `clk`, `reset`, `read_data`, `write_data` and `sw`. The last three signals come from the controller. The `sw` is a (`std_logic_vector(7 downto 0)`)and is sent to `tx` when `write_data` from the controller equals 1. The outputs of the UART are `tx`, as explained previously, `flag`, and `led`. The latter two outputs are sent to the controller.

The external timer has two inputs and two outputs. The two inputs are: `clk` and `turn_counter_disable`. The `turn_counter_disable` comes from the controller. The two outputs are `turn_counter` and `enable_sensor`. Both are sent to the controller. The `turn_counter` is a (`std_logic_vector(27 downto 0)`) and is therefore a 28-bit vector. The output signals are sent to the controller.
After the components were defined, the intermediate signals had to be defined for the top-level system. The code can be seen in Appendix A.1.

The count is a (`std_logic_vector(19 downto 0)`) according to equation 3.1. The sw to the UART is a (`std_logic_vector(7 downto 0)`) because the communication between the robot and computer uses 8

bit vectors. The external timebase is connected with a (`std_logic_vector(27 downto 0)`) because this timer uses 28 bits. The others are normal logic signals because they only need to be high or low. The signals motor signals are intermediate signals between the controller and motorcontrol. For the left and right motors, different intermediate signals are used.

The last task for the top-level description was to connect the intermediate signals to the components. Also the general clock from the uart module had to be connected to each of the components. Every component is described with the connection to the intermediate signal and the input signals. The decision was made to use the motor control as a general component so it can be used for the left and right servo motor. Two motor control entities are mapped, one to the output for the left motor and another for the output for the right motor. The generic parameters which specify the number of ticks per pulse for each speed are also specified here in a generic map: one per motor control entity. The code can be seen in Appendix A.1.

3.2. Simulations top-level description

The simulations are made in the program ModelSim. In this program, the code is implemented and a test bench is added. The test bench of the top-level description can be seen in appendix A.2. In the simulation, the FSM of the top-level is simulated. The simulations resulted in the following graphs:
This is when the line follower is turned on. It can be seen that after 200 ms, two of the sensor inputs changes.

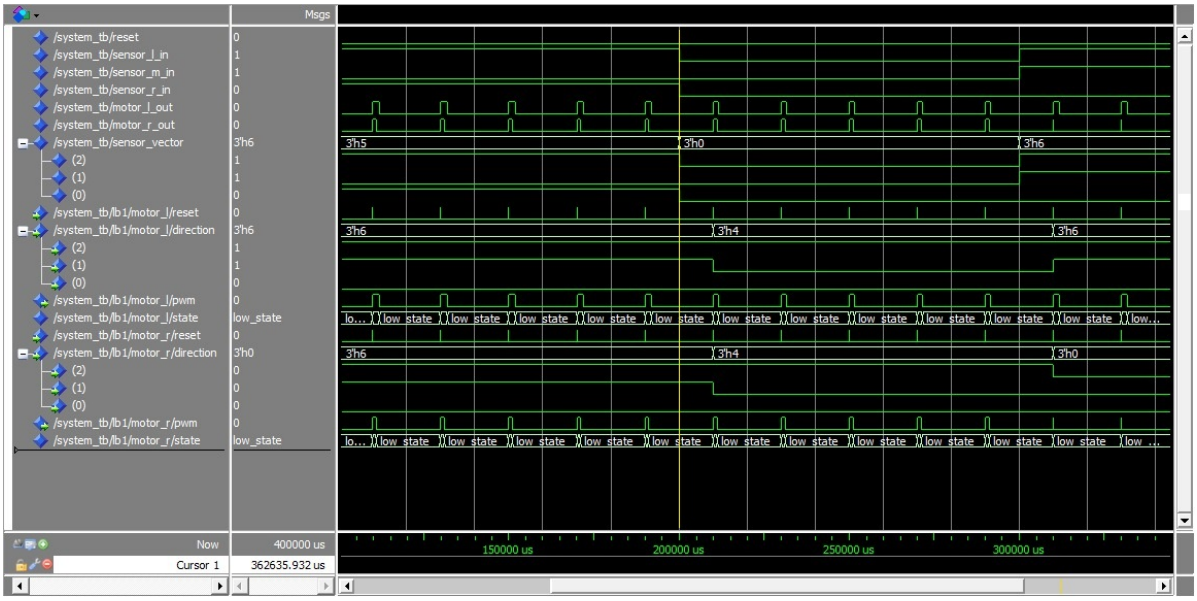

Figure 3.2: simulation top-level with enable_sensor = '1'

The left sensor and the right sensor are switching from a logic one to a logic zero at the yellow line. As a result, multiple other signals are changing. One of the motor direction vectors changes at the new clock cycle. This means that the signals are changing according to a changing input.

4

Time base

The motors of the robot are controlled with a PWM signal. The time base will give a time reference for the motor control. It will count the periods of the 50MHz clock used by the system. The time base is carried out with a counter. Chapter 5 of the manual of epo 2 is called: 'Designing a counter: the time base'. This chapter consists of describing behavioral circuits, describing clock circuits, describing a counter in VHDL, and finally, how to simulate a test-bench. After being able to describe all these thing, a counter can be designed and implemented.

4.1. Implementation

From the figure it can be seen that the counter work as a flip-flop with 4 inputs: `clk`, `reset`, `enable` and the `new_count`. The count gives a signal count, this becomes the new_count, and the new_count is a feedback signal for count. Every count signal is equal to the output signal count_out. The `new_count` is equal to the `count` + 1. This will be excecuted trough a new process in the Time base VHDL description, which can be found in Appendix A.3.

The VHDL code for the time base will be written according to the schematic representation of the time base, as shown in figure 4.1 [1]. The time base will only have the input signals `clk` and `reset`. The counting process is executed every time a rising edge occurs. It will have as output the `count` signal and the `count_out` signal. This signal will go to the 'adder'. The 'adder' will add +1 to the count signal and this will be the `new_count` signal. This signal will be sent to the register again. Essentially, `count` is a measure for the amount of clock-cycles that have passed.

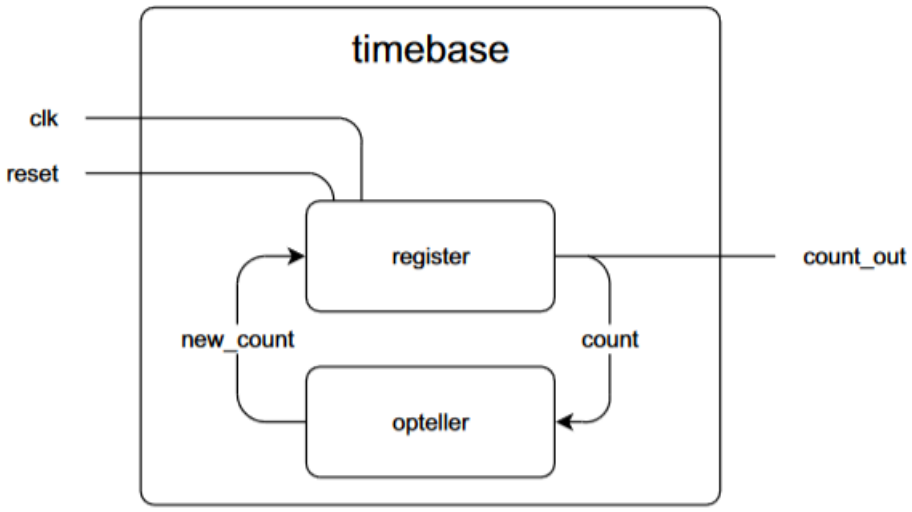

Figure 4.1: Schematic representation of time base

4.2. Code of the time base

The design of the time base in VHDL is as follows: first the the only entity, the entity time base, is defined. The time base has two inputs: the clock and the reset. It has the output `count_out`. The `count_out` has 20 bits. Therefore we use `std_logic_vector(19 downto 0)`.

After the entity is defined, the behavioral architecture is described. To store the current and next count, two signals, `count` and `new_count` are used; unsigned 20-bit vectors, since there won't be a negative count.
When the behaviour is described, the clock counts the rising edges. When the signal `reset` equals 1, `count` will go back to 0. If `reset` equals 0, then the count will take the value of `new_count`.
The signal `new_count` is defined as `count + 1`. Therefore a process is needed in the time base architecture that adds +1 to the value `count` and assigns this value to `new_count`.

4.3. Simulation

A test bench was used to check if the time-base code works. A.4. In the test bench, `clk` and `reset` get the values that are assigned to them in the test bench. After running the simulation, it can be seen that, after every rising-edge of the `clk`, the `count` signal changes. The `count_new` is the same as `count` but is different in phase. It's one phase ahead of `count`.

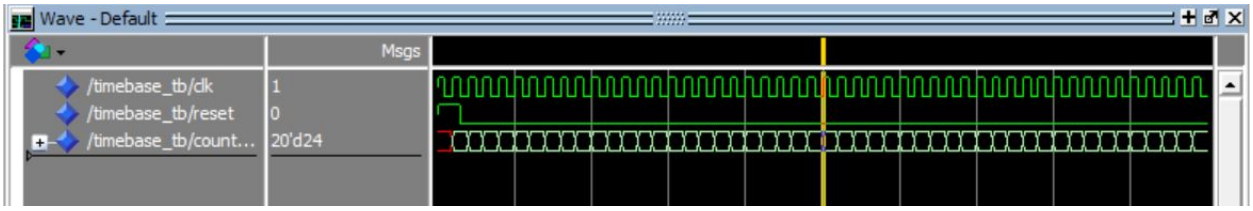

Figure 4.2: simulation of the time base

5

Input buffer

The output of the sensors needs to be read in by the controller FSM. Nevertheless, the sensor signals changes asynchronously, whereas the FSM is in sync with a 50 MHz clock. To solve this discrepancy, an input buffer is used, which ensures that the sensor inputs are stable during an active clock-edge before they are read in by the controller. Only in this way can they be safely propagated through the system.

5.1. On stability

An FSM with more that two states uses multiple bits to encode its present and next states. Unfortunately, each bit is calculated with a separate piece of combinatorial logic, which results in different delay times. It may occur that the input arrives at such a moment that one bit changes during one clock period, but the other more delayed bit, changes during the next period. Since it uses only one clock-cycle to calculate its new state, the FSM may enter into an undesired state. This can be seen in Figure 5.1 from the manual [1].

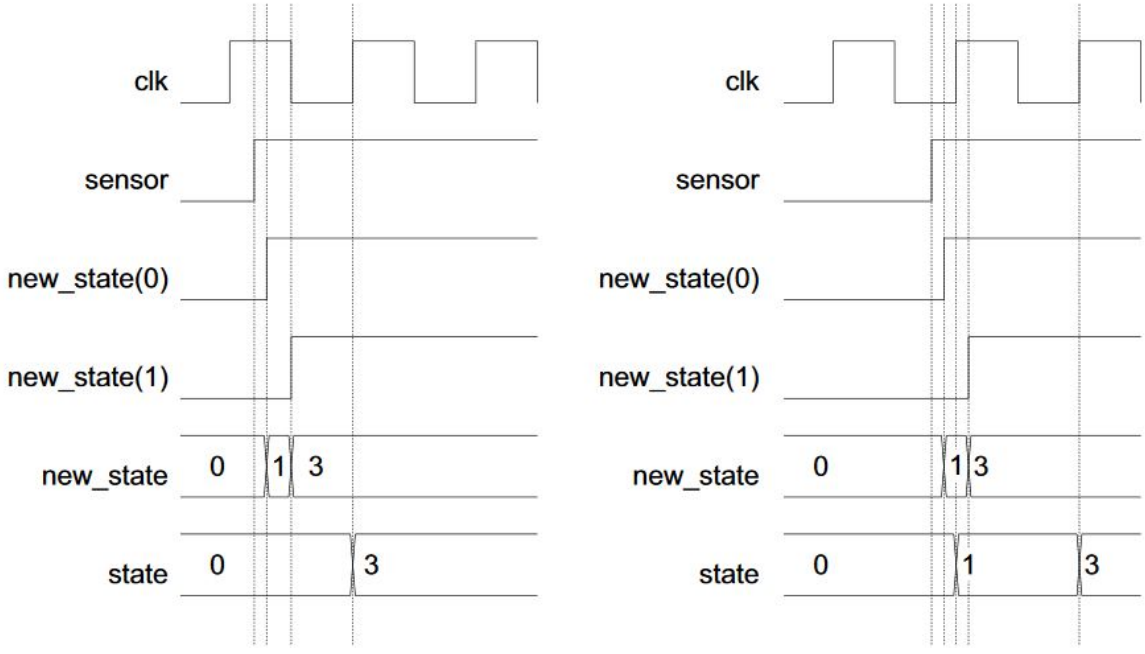

Figure 5.1: The effect of having an ill-timed input event

Here the FSM has two encoding bits: new_state(0) and new_state(1). In the left-hand image, both bits change in the same clock-cycle, resulting in a correct new state. In the right-hand image, new_state(1) has too much delay and changes after the next active clock-edge. This causes the FSM to go into the wrong state (state 1). To avoid this problem, an input-buffer needs to be added.

5.2. Implementation

To ensure that the input events are in sync with the clock of the system, they need to be read in by a flip-flop before they can pass through the system. This flip-flop is in sync with the FSM and produces its output only at the active clock-edge, ensuring a safe input signal.

Nevertheless, adding only one flip-flop can still cause errors due to the fact that a flip-flop needs a certain setup-time to read in the signal. If the signal is not stable during this time, the output is undefined or *meta-stable*. In that case, the output of the flip-flop is not a discreet signal but hovers in between the two voltages, which can cause a lot of problems if this is read in by the FSM. To reduce this risk, a second flip-flop is added to ensure that the meta-stable signal has one more clock period to enter a stable condition. Unfortunately, this doesn't completely eradicate the risk, but for now this can be neglected.

Thh implementation described above is called an input-buffer and will be placed in between the sensor and the controller. Since the sensor input is not just one signal, but three, the buffer needs a 3-bit register (see Figure 5.2). This is in fact three D flip-flops placed in parallel.

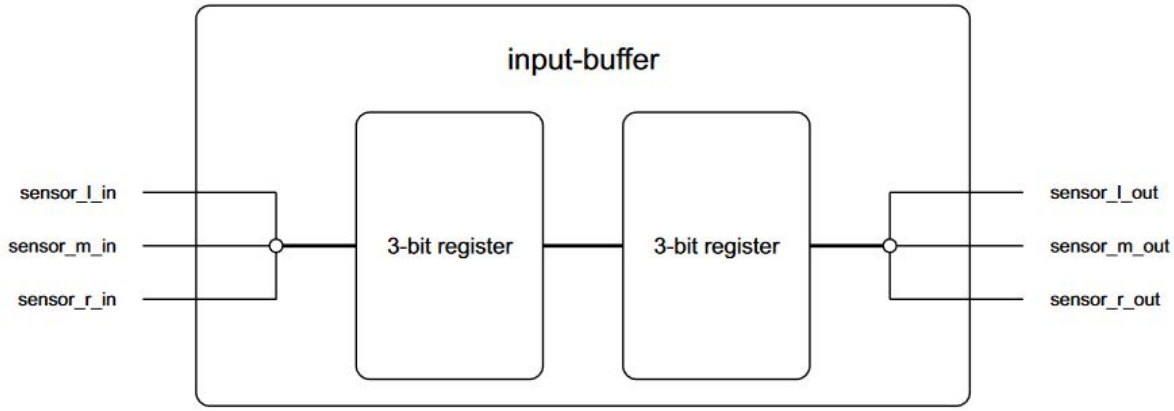

Figure 5.2: Schematic of the lay-out of the input buffer

Like every component of the robot, the input buffer will be described using VHDL. This code can be viewed in appendix A.5. Here the two 3-bit registers are cascaded in a port map.

5.3. Simulation

To verify whether this code functions properly, a testbench is used, whose code can be viewed in appendix A.6). Here, the sensor signals take on random values at random times. After running the simulation, the outputs have the same value as the inputs, but they are in sync with the clock edges, as can be seen in figure 5.3. Note that the red line is due to the fact that the flip-flop needs one clock-cycle to produce its output.

Figure 5.3: Testbench results of the input buffer

Since the simulation results show that the VHDL of the input buffer correctly synchronizes the input signals, it can be safely placed between the sensor and the controller of the line follower.

6

Motor control

To ensure the robot navigates in the right direction, a motor controller is added. This component instructs the wheels to drive backwards, forwards, or stop completely.

6.1. Requirements

The motor control sends a PWM signal to the servomotors, as described in section 2.2. This signal has the following requirements [1]:

- The frequency of the signal has to be 50 Hz (T = 20 ms)
- The duty-cycle must be between 5 and 10 %
- The pulse-signal must be between 3 and 5V
- If the duration of the pulse is shorter than 1.5 ms, the engine turns left
- If the duration of the pulse is longer than 1.5 ms, the engine turns right
- If the engine must stop, there can be no signal coming from the motor control

The PWM signal will then look like Figure 6.1 from the manual[1]. 1 ms constitutes a duty-cycle of $(1/20) * 100\% = 5\%$ and 2 ms constitutes a duty-cycle of $(2/20) * 100\% = 10\%$. Note that in the line-follower setup, the wheels are mirrored, meaning that they turn in opposite directions when given the same PWM signal.

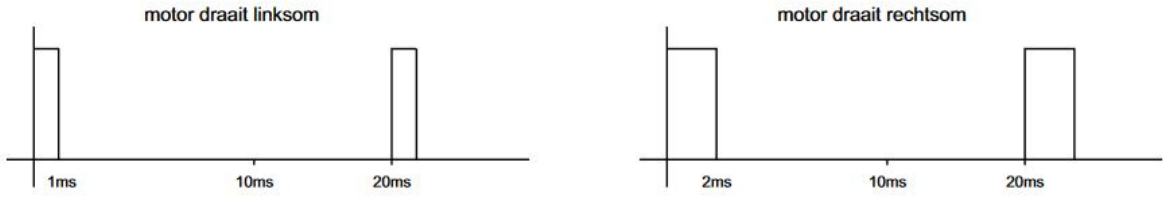

Figure 6.1: Left figure: the motor turns left. Right figure: the motor turns right.

6.2. FSM Implementation

The motor control is implemented as an FSM (see Figure 6.2 with the following three states:

- `reset_state`: The motor control enters this state at the start of every pulse-period. It can only be entered when the controller sends a reset, which happens every 20 ms. The next state is the `high_state`; `reset_state` doesn't need any input to transfer to this state: it happens automatically.
- `high_state`: This state is entered after the `reset_state`. Here the actual pulse is generated, so `PWM` is high. The next state is the `sleep_state`.

- `sleep_state`: The moment at which this state is entered, depends on the pulse length. If the corresponding time-value of `count_in` (see chapter 4) equals that of the desired pulse_width, this state is entered. It is left again only when the FSM is reset.

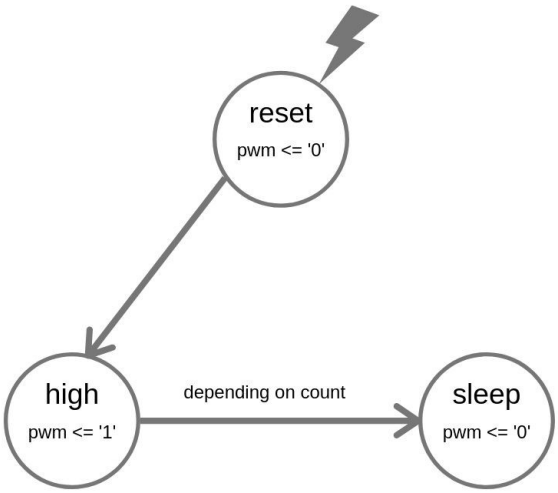

Figure 6.2: Schematic of the FSM of the motor control

6.3. Code

The FSM will be implemented in VHDL, the code of which can be found in appendix A.7. This section highlights the most important aspects of the code.

The entity `motordriver` uses a generic map that defines the widths of the different pulses, measured in number of clock-cycles. Based on the direction, the count_in must be equal to a certain tick-value before entering the `sleep_state`.

The direction is encoded in three bits as follows:

Code	Meaning
"000"	stand still
"011"	fast forward
"001"	slow forward
"110"	fast backward
"100"	slow backward

6.4. Simulation

The FSM is simulated using a test bench, the code of which can be found in appendix A.8. In this case the generics are defined as follows:

Generic	Time	Clock periods
`fast_backward_ticks`	1.0 ms	50,000
`fast_forward_ticks`	2.0 ms	100,000
`slow_forward_ticks`	1.7 ms	85,000
`slow_backward_ticks`	1.2 ms	60,000

Note that these are not necessarily the values used in the line-follower. The corresponding amount of clock_periods are calculated using this formula ($f = 50\,\text{MHz}$):

$$N_{clock\_periods} = f * t \tag{6.1}$$

In the test bench, the input directions are fast forward, slow forward and fast backward. The corresponding pulses are 2.0 ms, 1.7 ms and 1.0 ms, which is in accordance with the simulation results (see Figure 6.3).

Figure 6.3: Simulation results of the motor control testbench

The motor contol was uploaded onto the FPGA to test if it functions properly. A series of random direction were input through the switches on the board, to which the robot reacted according to expectation.

7

Controller FSM

The controller is the final component needed for the basic architecture to make the robot follow a line or execute a command from the computer. The line follower interprets the input signals from the sensors, and determines what the motors should do based on that input. Like the motor controllers,(chapter 6) the controller is an FSM[1], but more complex. The line follower can be enabled and disabled.

7.1. Desired functionality

The controller is the most important component in the circuit, in the sense that it controls all the other components, except for the input buffer. It must reset the timebase, external timebase, mine detector, motor drivers and communicate with the uart every millionth clock cycle so that the system operates at 50 Hz[2] and it determines what the motors should do based on the current and previous input from the IR reflectivity sensors, also 50 times per second.
The result of this process should be that the robot can drive from one destination to another according to the commands given. Also it has to detect mines and communicate via the uart to change direction.

7.2. Input and output

The controller has the following inputs:

- `clk`: 50 MHz clock signal of the FPGA
- `reset`: external reset signal
- `mine_detected`: signal from mine detector which indicates whether a mine is detected
- `enable_sensor`: Sensor which indicates whether the line follower is enabled
- `flag`: Indicates whether a signal has been received from the computer
- `sensor_input`: 3-bit vector with buffered sensor inputs out of the input buffer, originating from the IR reflectivity sensors
- `count_in`: 20-bit count vector from the timebase.
- `in_signal`: Input from computer
- `turn_counter`: 28-bit count vector from external timebase for making smooth turns

The `clk` and `reset` both come directly from ports on the FPGA, the sensor input signals come from the input buffer, the `mine_detected` signal comes from the mine detector and the `flag` signal comes from the uart module.

The controller generates the following outputs:

[1] Finite State Machine
[2] the PWM frequency of the servo motors

- `count_reset`: triggered when `count_in` reaches 999999, resetting the timebase every millionth clock tick, which is 50 times per second with 20 ms per system cycle.
- `turn_counter_disable`: used to disable the turn counter when the turn has been made
- `write_data`: for writing with the uart module
- `read_data`: to read with the uart module
- `out_signal`: 8-bit vector to send with uart which indicates the command that is executed by the robot
- `motor_[l|r]_reset`: triggered together with `count_reset` providing the 50 Hz base clock for the motor drivers and the servo PWM signals
- `motor_[l|r]_direction`: 3-bit vector that tells the corresponding motor driver what to do

The VHDL entity of the controller can be found in Appendix A.9. A graphical representation of the entity and its connections can be found in Figure 3.1.

7.3. Implementation and operation

As mentioned at the beginning of the chapter, the controller operates as an FSM. For stability, a Moore machine was chosen over a Mealy machine. Reading the uart input and determining the robot's direction, is done in the `check_state`, which is entered at the beginning of every 20 ms system cycle and lasts one clock tick or 20 ns. The `check_state` is also the initial state of the FSM, meaning that this state is entered when a checking point on the track has been reached. This also occurs when the system is reset externally or powered on for the first time. In this state, besides reading and interpreting the inputs, the controller also triggers the `reset` of the timebase and sends a character with the uart to the computer.

To make faster turns, the decision was made to cut corners using so-called smooth turns. This means that the robot will leave the line for a short while. As a result, the line follower has to be disabled in smooth turns and enabled when the turn has been completed. To drive straight, the line follower has to be enabled again. As a result the controller has to be written such that the line follower can be enabled and disabled.

In the `check_state` a signal is received from the computer via the uart module. Table 7.1 lists the possible signals inputs and their interpretation:

Occurrence	`turn`	Description
Expected	`"01010010"`	Smooth right turn
	`"01001100"`	Smooth left turn
	`"01111100"`	Forward
	`"01111101"`	Sharp right turn
	`"01111011"`	Sharp left turn
	`"00101000"`	Turn 180 degrees left
	`"00101001"`	Turn 180 degrees right

Table 7.1: Possible sensor inputs

If the robot has to go forward until the next checkpoint, the line-follower has to be enabled because the robot will not go off the line. The robot drives according to its sensor inputs and sends the next signal via the uart module when encountering a `sensor_input` with `"000"`, which indicates that a check point has been reached. With the signal `enable_s` the line follower is enabled which is only changed when the command is received via the uart module that it has to be disabled.

In the case of a smooth turn, the robot has to go off the track for a while. The line follower has to be disabled for this turn. Because the robot must still drive without its sensors, an external timebase, also called `turn_counter` was made. With this counter, it will be possible to drive the robot through the turns with small steps of different directions. In every small step the controller executes a direction state. With the signal `enable_s` the line follower is disabled. The robot automatically enables the line follower again when the turn is completed.

In the case of visits and the 180 degrees turns the robot is just as the smooth turns off the track for some time. Therefore, for this turn the external timebase is also used and it works the same as for the smooth turns but for the 180 degrees turn it takes longer to complete the turn. This makes it that the external timebase has to count longer before it is resetted again.

7.3.1. Forward state

For the forward command, the line follower is enabled. The line follower gets different inputs from the sensors and has to send an output signal to the motor controllers. For the expected command, the subsequent states and their outputs are listed in Table 7.2: The robot has to drive according to its input sensor and received commands. Table 7.3 lists the possible sensor inputs and their interpretation:

Action	new_state	motor_l_output	motor_r_output
Go straight	straight_state	"110"	"110"
Bend slightly right	right_state	"110"	"100"
Bend slightly left	left_state	"100"	"110"
Hard right turn	hard_right_state	"110"	"000"
Hard left turn	hard_left_state	"000"	"110"

Table 7.2: Expected command inputs and subsequent states and output

Occurrence	sensor_input	Description
Expected	"101"	Sensors centered on line
	"100"	Sensors slightly left of center
	"001"	Sensors slightly right of center
	"110"	Sensors left of center
	"011"	Sensors right of center
	"000"	Perpendicular on line or at junction
Unexpected	"111"	Sensors off line
	"010"	Unknown

Table 7.3: Possible sensor inputs

The first six input vectors listed in the table can be expected to occur while following a line, and a single subsequent state (and corresponding driving direction) can be assigned to each of them. The last two listed input vectors are not expected to occur, and the position of the robot relative to the line can not be derived from them, but they have to be handled nonetheless.
This is done by saving the general direction in which the robot is moving, (`straight`, `left`, `right` or `standstill`), in a signal, `last_direction`, which can be used to determine the driving direction when unexpected input occurs during the next `check_state`.

If `sensor_input` is `"111"` or `"010"`, `new_state` is based on the `last_direction` signal:

last_direction	Action	new_state	motor_l_output	motor_r_output
straight	Go straight slowly	slow_straight_state	"100"	"100"
left	Turn left in place	inplace_left_state	"001"	"100"
right	Turn right in place	inplace_right_state	"100"	"001"
standstill	Stand still	standstill_state	"000"	"000"

Table 7.4: Failure-mode states and output

When the controller is initialized or reset externally, the `last_direction` signal is set to `standstill`. After the first system cycle, `last_direction` is determined by the last value of `current_direction` which is set during each of the executive[3] states:

state	current_direction
straight_state slow_straight_state	straight
left_state hard_left_state inplace_left_state	left
right_state hard_right_state inplace_right_state	right
standstill_state	standstill

Table 7.5: General directions associated with the executive states

Figure 7.1 is the Finite State Diagram (FSD) of the controller FSM:

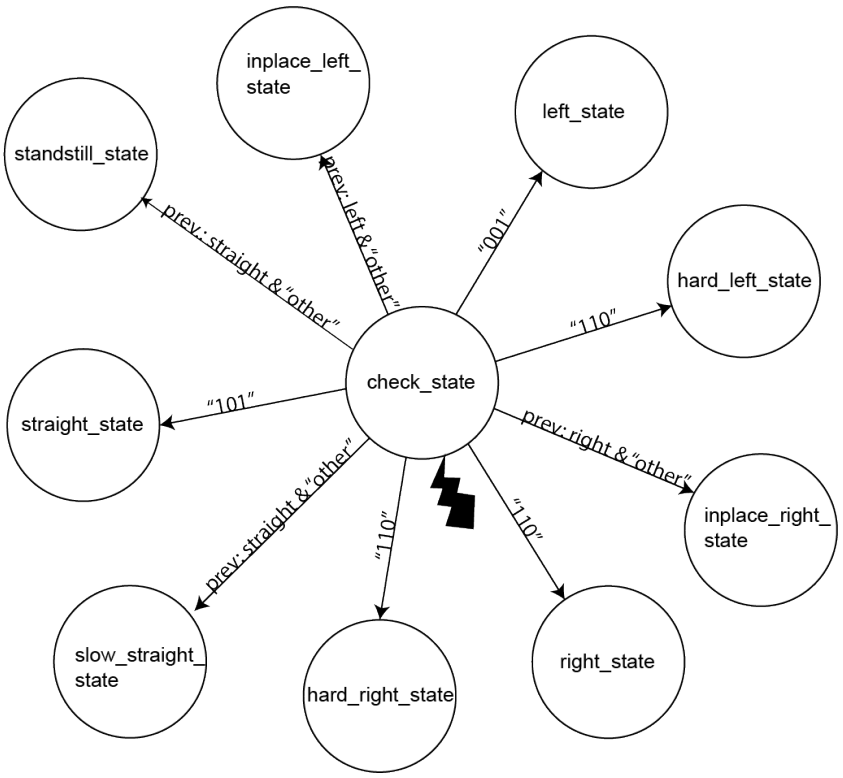

Figure 7.1: Controller FSD when `enable_sensor` = '1'

In this diagram, the `"other"` condition means that the controller receives an input sensor vector other than the *expected* values given in Table 7.3.

The VHDL description of the controller can be found in Appendix A.9.

[3] executive states: states in which the motor driver outputs are set

7.4. The external timebase

In order to allow the robot to navigate through the track as fast as possible, the robot is set to receive turn orders when arriving at a check point, as mentioned before. Due to the fact that the robot receives the order of the turn before arriving at the corner, it is able to start turning before the crossing and cut the corner. In this way time is saved while making turns and the robot will be faster. However a smooth turn requires an inconstant movement of the motors, so a simple smooth turn state,is not an option.

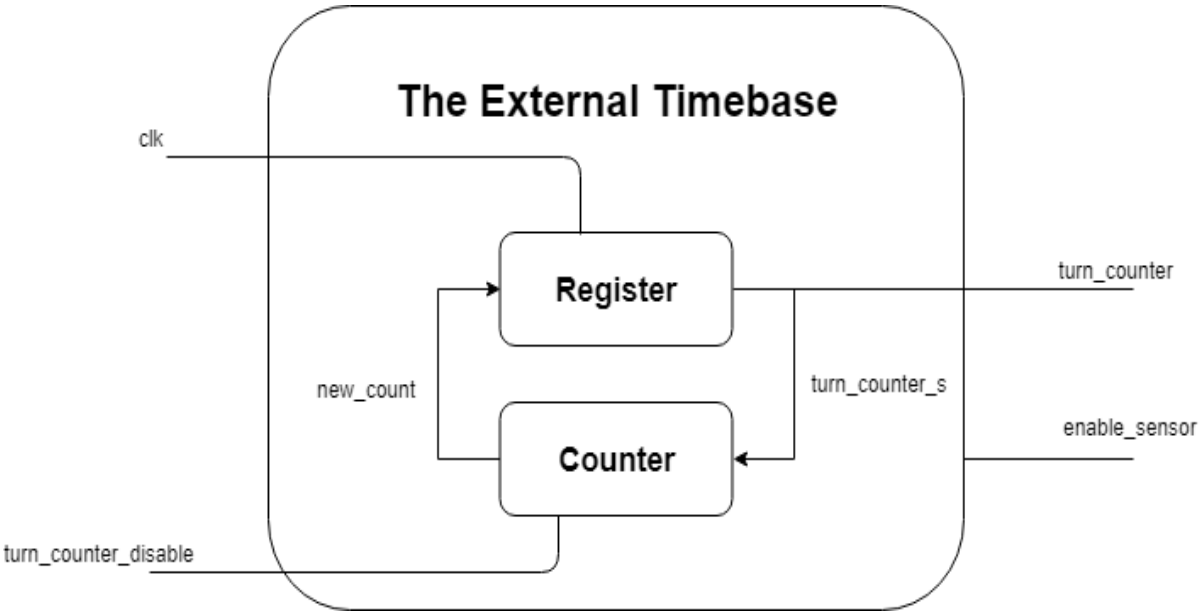

Figure 7.2: Schematic representation of the external timebase

So to implement this method, the robot has to follow a certain procedure in order to cut the turn. First, a new component was developed: an external timer that starts counting when a station is detected. Its design is similar to the design of the timebase but it has a a few more functions. The external timebase has a clock and the disable signal (`turn_counter_disable`) as inputs, which are needed to count and to disable the timer when needed. Disabling the turn counter is needed when the turn has completed and enabling the turn counter is needed when the robot has to start a smooth turn. The outputs of the external timebase are the counter(`turn_counter`) and an `enable_s` signal, which enables the sensors again and therefore the line follower when the external timer is disabled, as mentioned before.

7.4.1. Implementation of the external timebase

Now that the external timer has been designed, a curved turn can be made according to the procedure. When arriving at a station, the robot receives a certain signal as described in 7.1 from the C code when it has to make this turn. The external timer turns off the line follower and starts counting. After tuning the robot and the motors, the curved turn was separated into several steps. When the order is received as in 7.1 the external timer then sets the time-period of each states. An example of curved right turn: The right turn script is divided into two states. The different parts of the script are shown in different highlights. First the robot drives straight for a certain time period (counts)

```
case turn is
                        when "01010010" =>          --Right
            if(turn_counter<to_unsigned(42000000,28)) then --0.5second
                                                disable_s <= new_disable_s;
                                                new_state <= straight_state;
```

after that it takes a slight turn in the preferred direction which is in the example case the straight state. When the robot has finished driving straight, the robot now drives a certain time to the right as shown in the script below.

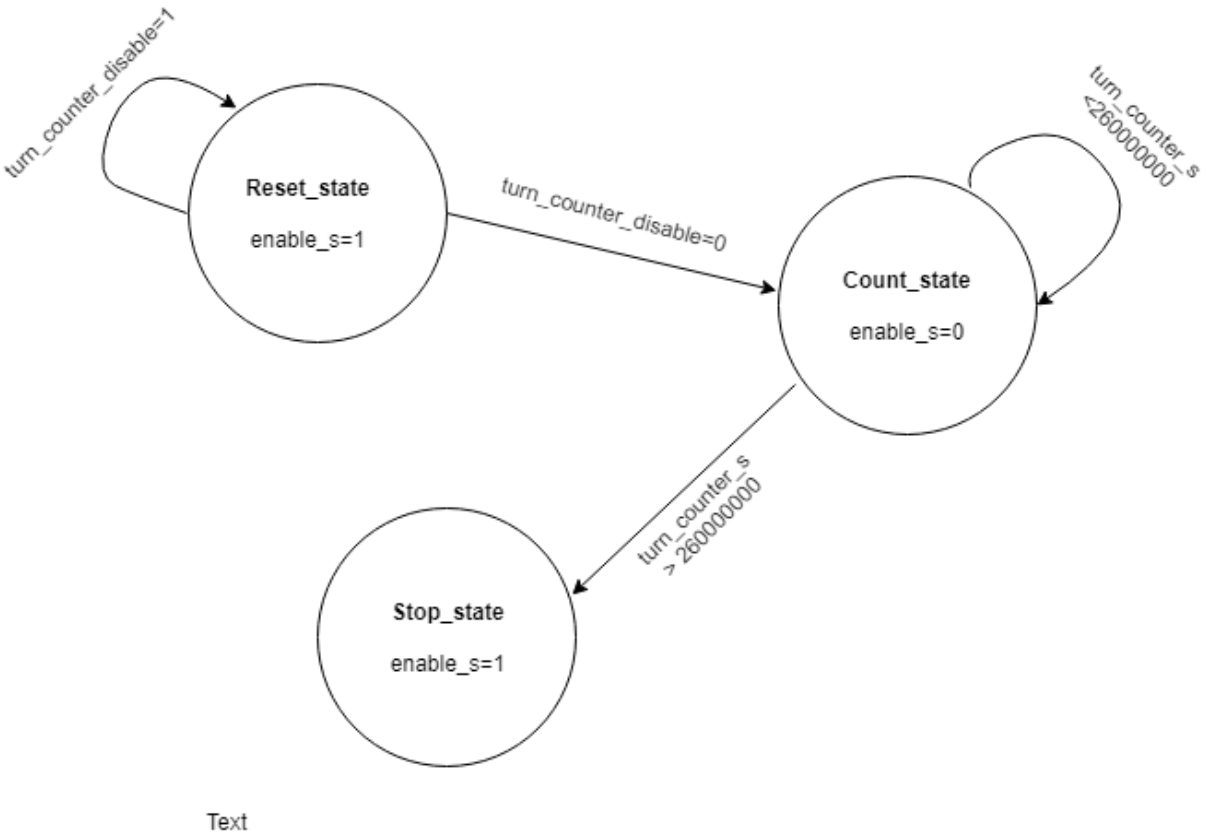

Figure 7.3: The fsm of the external timebase

```
elsif  (turn_counter < to_unsigned(156000000,28)) then ---1sec
                                        disable_s <= new_disable_s;
                                        new_state <= right_state;
```

According to the tests, the robot arrives at the line after finishing step one and two of the procedures. At this point the external timer is turned off and the line follower turned on again. It can be seen in the script below that line follower is turned on again.

```
                    disable_s <= '1';
                new_state <= right_state;
            end if;
end case;
```

The external timebase was initially designed to make curved turns, however, later it was implemented for many other movements of the robots, such as 180 degrees turns and station visits. In all these movements the same procedure is followed: each movement is a part of the controller states for a certain time; the state and the exact timing of it, are chosen based on testing and tuning.

7.5. Simulation and conclusion

Since the controller works together with all the other components, the top-level testbench and its simulation (Section 3.2) suffice to verify its functioning. Additionally, the functioning of the system as a whole was tested, and demonstrated to TA's and tutors during a lab session. It was found to function well, and the addition of the "failure-mode" options with the `last_direction` signal proved useful for preventing the robot from falling off the table when it lost track of the line at a sharp turn on one occasion. (Most groups make the robot go straight ahead when the sensors do not detect the line.)

8

Communication between the computer and the FPGA

The computer decides when the robot should go to left or right, but the exact movement of the robot is controlled by the FPGA. For example: how long the robot should take for a turn. To communicate between the two parts, XBee modules with UART communication are used.

8.1. How UART works

UART is an asynchronous, serial communication protocol. The idea is that to communicate between two devices a series of ones and zeros are sent over a wire and that the receiver can decode this to a multiple of 8 bits. The default value one the line is a '1'. As seen in figure 8.1. But when you want to sent an 8 bit string you start with a starting bit which is always '0'. After this there can be read 8 bits, the bit after these eight is always a '1' which is called the stop bit. Where after this sequence can be done again.

This communication protocol is used to communicate between the computer and the FPGA. This is done wireless with two XBEE modules. Which are PCB's that can be linked and then communicate on a certain frequency. When one sends for example a '1' the other receives this '1'. One XBEE is connected through USB with the PC the other is directly put on the robot. And so directly connected to the FPGA where the receiving and sending code is. Making use of XBee modules the link can be layed wireless. The VHDL description to send and receive bits is given, so only its application will be explained, not the underlying theory.

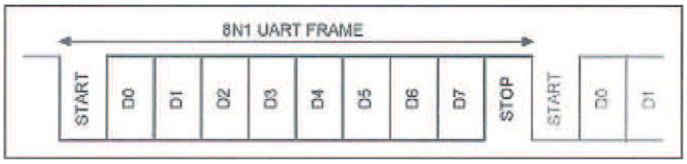

Figure 8.1: UART communication data frame

8.2. How the FPGA sends bits

The VHDL UART description has been used as seen in the appendix. If `write_data` is '1', the `sw` bit vector is sent to the computer on the `tx` output. If `read_data` is '1', `flag` is set to '0' and `led` is filled with the buffer. The flag is '1' if the buffer is filled, otherwise it is '0'. Thanks to this mechanism, a bit can be sent and received when needed.

8.3. How the computer sends bytes

The C program uses a library (`rs232.c`) to interface with the operating system's serial ports, as described in Section 11.4. The library provides functions to open and close connections to serial ports and read from and write to them, and some functions for buffer control.

On top of this library, in `uart.c` the functions `uart_tx()` and `uart_rx_wait()` are defined, which are used to respectively transmit a signal, and wait for a signal to be received. When an XBee module is connected via USB, it can be used as a serial interface to communicate with the robot.

8.4. Communication from computer

In the design, it was decided that movements are to be sent to the FPGA and the VHDL description should be able to interpret them. The possible movement instructions are listed in Table 8.1:

Movement	Character
Right	'}'
Left	'{'
Forward	'|'
Hard turn right	'>'
Hard turn left	'<'
Turn 180 degrees right	')'
Turn 180 degrees left	'('
Visit right station	'/'
Visit left station	'\'
Visit forward station	'^'

Table 8.1: Movement commands to robot

All these characters will be sent with UART in ASCII coding. Another chapter will elaborate on how the robot performs its movements; this chapter will only explain when and how the robot sends these characters.

8.5. Communication to computer

Every 20 ms the robot sends a signal back to the computer from which the status of the robot can be deduced:

Situation	Character
Intersection or edge detected with mine	'M'
Intersection or edge detected without mine	'X'
Intersection or edge detected but no command received yet	'W'
No intersection detected; currently driving	'\0'

Table 8.2: Location characters to computer

8.6. The use of UART in the controller

First it is important to know when the controller sends or receives bits. The robot sends a bit when the robot is at an intersection(also includes the dots in the middle of a path). It will wait on that point till the robot receives a command from the computer. Then the robot will send its message. The robot will ideally receive its message before he enters an intersection and otherwise he will stop at the intersection till he receives one.

There are three scenario's there is accounted for:

- The robot already received a message and there is no mine at the intersection. Then will follow up the command and send a 'X'.
- The robot already received a message but there is a mine at the intersection. Then it will turn automatically and send a 'M'.
- the robot didn't receive a message, therefor will not move and sends a 'W'.

8.7. The simulation

This simulation will show first the third scenario and after that the first. Figure 8.2 contains a few signals of the complete system to explain how communication works. In reality a robot takes 3 seconds for a turn which

cannot be simulated without performance issues. Therefor some values for the timers are changed so that everything takes less time and the computer can simulate it.

- The first signal is `sensor_vector`. There can be seen that at a certain time it gets 000 which means the robot is at an intersection. This is the place where the robot needs to perform the command the computer has sent.
- `tx` is the signal of the outgoing communication line. There can be seen that when `sensor_vector` is 000 a 'W' is sent because the robot did not receive a command from the computer yet.
- `rx` is the signal of the incoming communication line. there can be seen that the complete command is not yet received when `sensor_vector` is 000.
- `mine_detector` is '0' this is just for simulation and says that there is no mine detected.
- `new_state` decides which state the FPGA will enter. When `sensor_vector` is 000, the robot enters the `error_state` because it does not know what to do and waits until it receives a command. Once this command has been received, the robot moves straight forward because that is what the controller commanded with instruction '}'.
- `disable_s` and `enable_sensor` both go 0 when the signal is received, which disables the line follower and make the robot follow the scripted path as explained further on.
- `received_vector` is '}' what the UART component received.
- `send_vector` keeps sending 'W' every 20 ms because the robot did not receive a command yet.

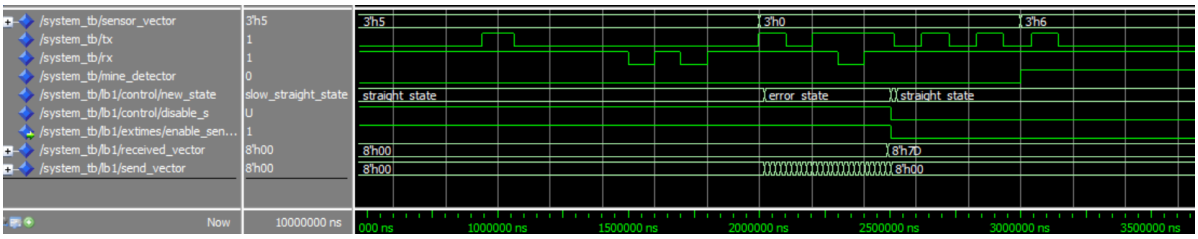

Figure 8.2: simulation of the system

The simulation code can be found at chapter A.10.

8.8. Conclusion

To dictate the movements of the robot, communication between the computer and the FPGA is necessary. This is done using the UART protocol. The computer sends commands and the FPGA sends a X, M or W back if it is on an intersection.

9 Mine sensor

The second challenge of the project is detecting and avoiding the mines. For this challenge a mine sensor is crucial to detect the mines. The mine sensor has to detect a mine on the track and send a signal to the controller that there is a mine located in front of the robot.The mine sensor will have a certain chosen oscillation frequency that will change in the presence of a mine.The design of the mine sensor was a long process with some difficulties who are described in this chapter.

9.1. Designing sensors

For the design of the mine sensor the first step was to choose the type of sensor. For a mine sensor there are two types of sensors that can be used; a capacitive and a inductive sensor respectively.

The capacitive sensor is a proximity sensor. This means that it detects nearby objects by their influence on the electrical field created by the sensor. If there is an object extremely nearby a capacitive sensor, the capacitance of the capacitor will change. When the capacitor is used in a circuit the frequency can be measured. Because of that the capacitance changes when a object is extremely nearby, the frequency will also change. The changed frequency determines if a object is detected.

Inductive sensors are also a type of proximity sensor, that works according to the principles of electromagnetic coupling between a sensor coil and the object that has to be detected. In order to be detected, an object must be conductive, like a metal. Ferrous metals will increase the inductance, while non-ferrous metals will decrease it. When there is a metallic object in the electromagnetic field induced by the coil, some of the circulating electromagnetic energy will be transferred to the metal object. The transfer of this energy will induce a current called *eddy current*. The eddy current flowing in a conductive object will on its turn, induce the electromagnetic field generated by the sensor coil. This interference of the object with the sensor coil's magnetic field will reduce the effectiveness of the sensor coil. There are several factors to keep in mind while designing inductive sensors. Some design factors for an inductive sensor are the distance from the object to the sensor and size of the object that has to be detected. Very small objects will not provide enough electromagnetic losses to be detected, while large objects are easily detected.

9.2. Designing the sensor

The basis for the inductive sensor is an inductor and capacitor. With coupling these two passive circuit elements in parallel, a simple oscillator is made. From the *Linear Circuits B* course we know that the oscillation frequency of this (parallel) circuit is equal to:

$$f_{oscillation} = \frac{1}{2\pi\sqrt{LC}} \tag{9.1}$$

To determine the value of the capacitor, the value of the inductor must first be known. To determine this, several inductors are connected to a RLC-meter and their overall sensibility of absolute change to the presence of a mine is measured, as shown in Table 9.1. The inductor with the highest sensibility (the 0.33 mH inductor) is chosen.

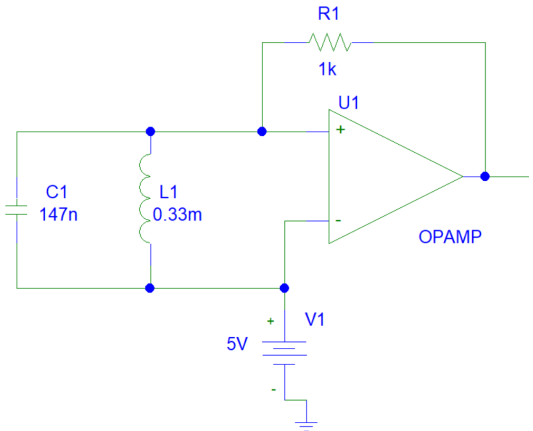

Figure 9.1: First design

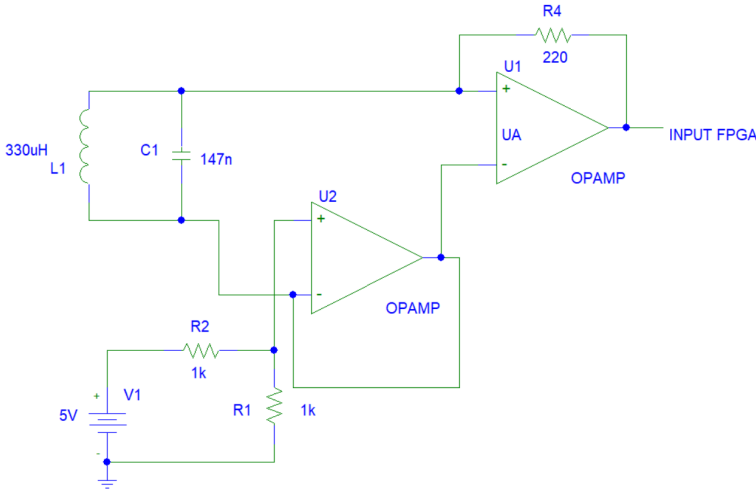

Figure 9.2: Second design, the right opamp is a comparator

number	given value	sensibility	actual value
1	0.351 mH	3.4%	0.33mH
2	0.499mH	2%	0.47mH
3	ovl	ovl	0.15mH
4	0.15mH	3.5%	0.5mH
5	ovl	ovl	2.2mH
6	0.99mH	2.2%	1mH
7	0.23mH	1%	0.22mH
8	0.47mH	2%	0.47mH

Table 9.1: The measured values and sensibility of different inductors

To determine the capacitor Equation 9.1 is rewritten to Equation 9.2 and there is a frequency chosen in the 20 and 25 kHz interval. The choice of the frequency interval is not random, there is an upper limit to this determined by the operational amplifiers output frequency and the decrease of its CMRR with higher frequencies.

$$C = \frac{1}{(2\pi)^2 f^2 L} \tag{9.2}$$

From this calculation and with the parts available in the Tellegen Hall the choice is made to use a 147 nF capacitor who gives with the 0.33 mH inductor an oscillation with frequency 22.8kHz. This frequency is going to be lower when delivered to the FPGA by the intermediate stage board between the FPGA and the sensor circuit board. Also the use of operational amplifiers has an effect on the frequency.

The first design as shown in figure 9.1 was oscillating but did not meet the requirements.

There had to be a 2.5 V baseline where the periodic signal is oscillating around, between 0 V (logic low) and 5 V (logic high). A second design (see Figrure 9.2) was a design with 2 operational amplifiers, where the first opamp has a 2.5 V at its negative terminal, created by a voltage divider. The second opamp amplified the oscillation of the capacitor inductor pair and with the help of the 2.5 V voltage buffer created by the first opamp it generated a periodic swinging around 2.5 V.

According to the simulation this design had to work flawless but when build there was no oscillation. After a lot of discussion and thinking there was only one idea and that was to change the value of the feedback resistor of 1 kΩ to 220 Ω. This worked well and after reasoning a bit it is correct. The problem was the fact that the second opamp had to deliver to much current due to the 1 kΩ resistor and could not perform an amplification on the oscillations. The output signal (as seen in Figure 9.4) of this circuit looked like square wave but the VHDL code could not trigger on this signal.

Although the form, frequency and voltage levels of this signal was appropriate for this project, after the intermediate board with whom the sensor is connected a different signal appeared which was to weak. This

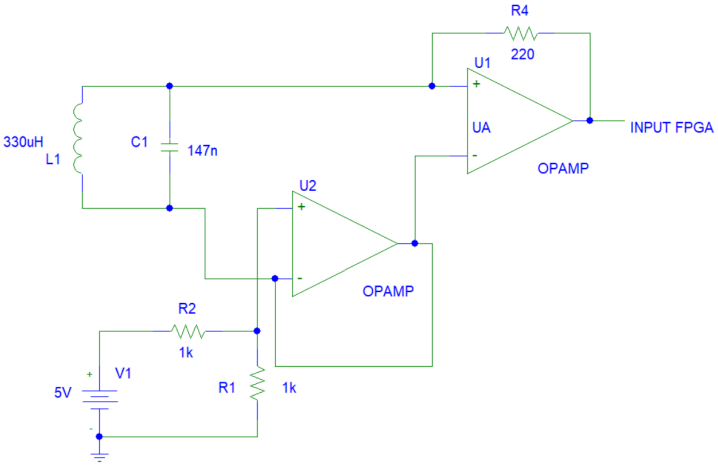

Figure 9.3: Second design, the right opamp is a comparator

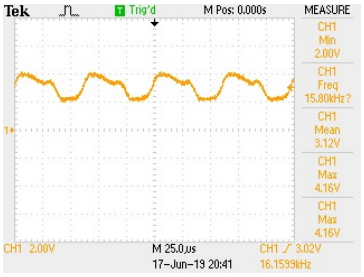

Figure 9.4: The output of the second design

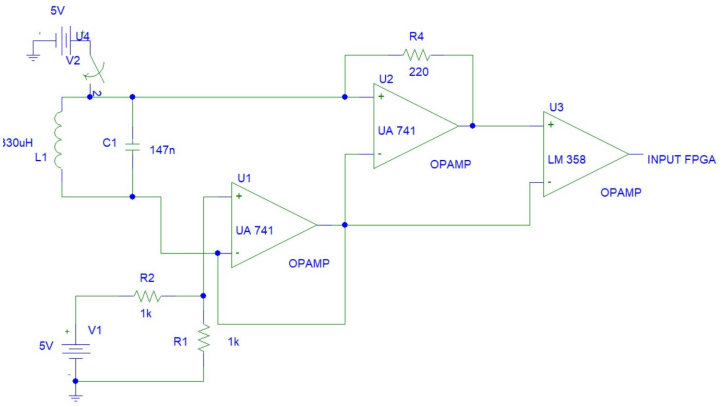

Figure 9.5: Final design with three operational amplifiers.

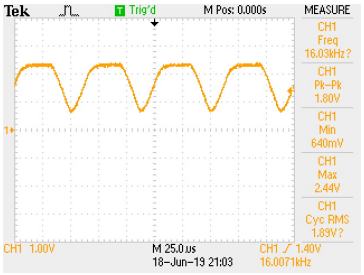

Figure 9.6: Input signal at the FPGA pin.

problem was first tried to solve by changing the VHDL code but this did not work well so a third and final design followed, Figure 9.5.

This design was made with the idea to generate a better square wave with the output of the second amplifier where a "stronger" signal with less losses was planned to get. To do this a third opamp was used in a comparator configuration without a resistor. This third opamp was also from another series(LM358) than the previous ones(UA741). The third opamp was chosen so because it had a different configuration and frequency response. This worked well and the third opamp also created a periodic signal. When the signal was measured at the input of the FPGA after the intermediate board whom the mine sensor is connected a cleaner square wave appeared as seen in Figure 9.6. The only problem with this design was that it could not start to oscillate automatically so it needed a manually performed electric "shock" to get started. This is done by providing the inductor capacitor pair an instantaneous voltage pulse with a button.

9.3. Summary

During this design there where some difficulties. One of them was the fact that the circuit was not oscillating while it had to be according to the simulations. Trying to find out what the solution for this could be took a lot of time. Another problem was to fix the signal power loss by the intermediate board on the robot. As a conclusion it can be said that designing and building a mine sensor was more difficult then expected. After the knowledge gained during this project, it is known that it can be designed faster and what the possible problems can be.

10

VHDL mine sensor code

The VHDL code of the mine sensor has to transform the changed frequency to a logic signal when there is a mine in front of the sensor. Therefore the VHDL consists of three different VHDL files. A sensorcounter, an FSM and the toplevel respectively. The sensorcounter, Appendix A.11, counts the amount of times a clock period fits into one sensor period. The FSM, Appendix A.12 , switches states from `nomine_state` to `mine_state` if the sensor counter shows a mine is present. The top level, Appendix A.13, connects both VHDL codes, as shown in Figure 10.1. The output of the sensorcounter is connected as input to the FSM. The clock and reset are the same for both the sensorcounter and FSM. The other input for the sensorcounter is the sensor, as in the hardware. As output of the total minesensor it has Mine_detected, which will show a '1' when there is a mine and a '0' when not. To test if the codes works a test bench is written which is shown in Appendix A.14.

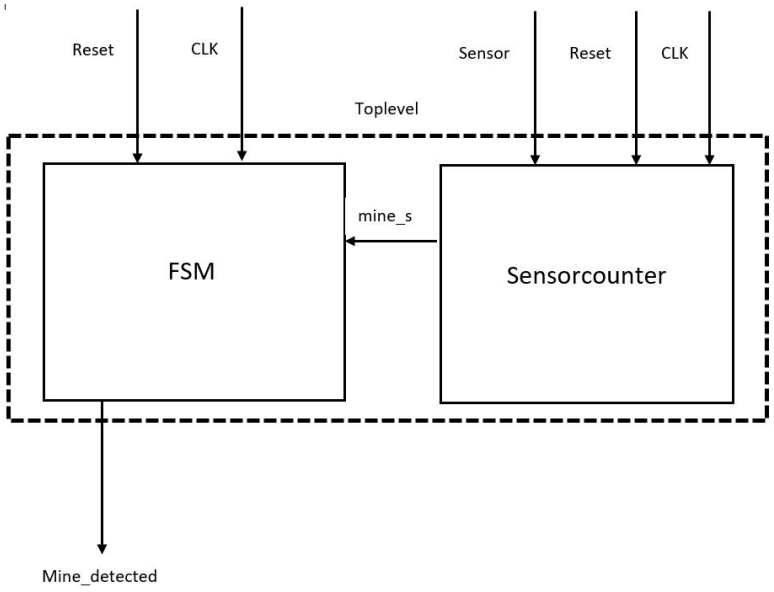

Figure 10.1: The graphic representation of the toplevel of the minesensor

10.1. Function of counter

The sensor counter is a synchronous counter. First the input `sensor` is buffered with two flip-flops. The counter starts when `sensor` = '0'. Then it will add 1 every clock cycle until `sensor` = '1' or `reset` = '1'. Figure 10.2 shows that the pulse of a mine is bigger than when there is no mine. This makes the count higher when there is a mine. When counter > 750, `mine` will be set to '1'. Else it is '0'.

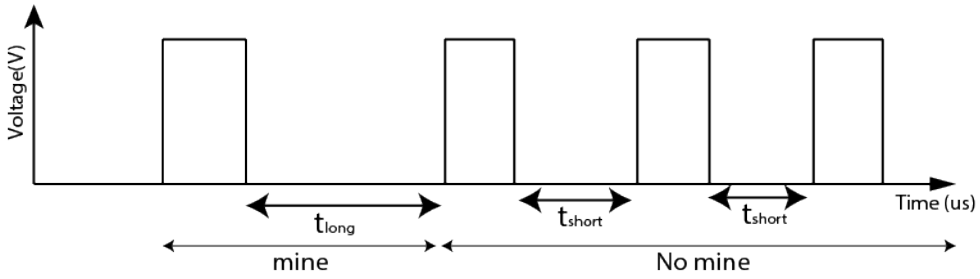

Figure 10.2: difference between pulse of mine and no mine

10.2. Function of FSM

The output of the sensor counter, when a mine is present, only shows a '1' for the duration of one clock period. For the controller to detect the output, a longer signal is needed. To achieve this, an FSM is used. The FSM has two states: `mine_state` and `nomine_state`. Figure 10.3 shows the graphic representation of the FSM. Every 20 ms the FSM is reset by the controller. Otherwise the signal stays in the `mine_state` forever.

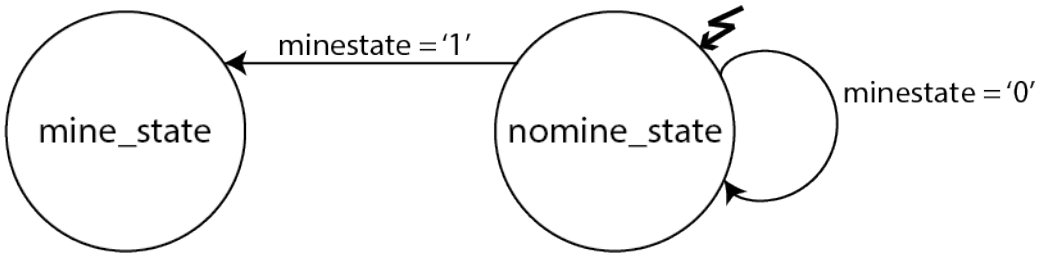

Figure 10.3: FSM for minedetector

10.3. Test bench

Figure 10.4 shows the simulation of the VHDL code. In the beginning of the simulation, the FSM is in the `nomine_state`, but when the input frequency gets lower, it moves into the `mine_state`. This simulation shows a working mine sensor

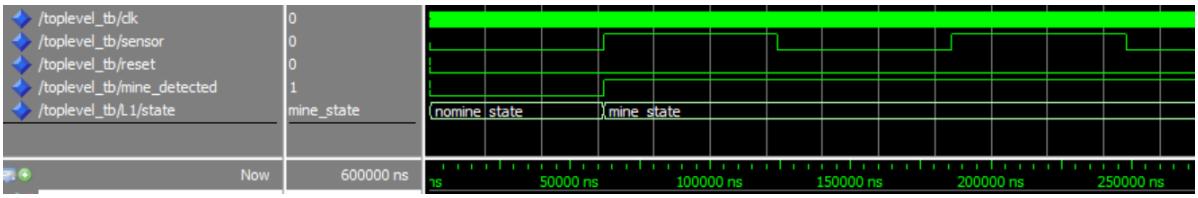

Figure 10.4: The simulation of the minesensor VHDL code

11

Control software

With the robot able to move along the grid and make certain movements, it can be instructed and controlled wirelessly via the XBee link from a computer running a custom C program.

11.1. Program structure

The program is divided into four `.c` source files (and their associated `.h` header files), each having their own task. A library for serial communication is also included:

- `main.c` contains the `main()` function and the code for the three challenges
- `router.c/.h` contains the code to generate and modify a `Grid` (see Section 11.2), and to calculate routes on one.
- `robot.c/.h` contains the `struct Robot` type definition, the declaration of the global `struct Robot robot` and a function to do certain operations on a `struct Robot`
- `utils.c/.h` contains some mathematical and miscellaneous functions that are not strictly related to any of the other source files.
- `uart.c/.h` contains the functions used to communicate with the robot during challenges
- `rs232.c/.h` is a serial communications library which works on both Windows and Linux

All files mentioned above can be found in Appendix B.

11.2. The challenge grid

The three challenges described in Chapter 1 take place on a grid with 25 nodes, 40 so-called edges, and 12 stations. To represent and use this grid in a program, a type `Grid` is defined; a 11x11 matrix of `struct Cells`. All places on the grid plane, that is: grid edges, junctions and white surfaces, are represented by a `struct Cell`. Every `Cell` has a value `Cell.v`, which indicates the type of location on the `Grid`.
In a standard `Grid`, edges, nodes and stations have value 0, while cells which represent a white surface have value -2. The standard `Grid` can either be hard-coded or generated algorithmically. Hard-coding would be easier, but intrinsically does not provide the flexibility of an algorithmic solution. Hence, an algorithmic solution was chosen. The function `generate_stdgrid()` (in `router.c`) consists of two double nested `for`-loops; the first loop determines the value for each cell, the second loop names the nodes, edges and stations. The function can be found in Appendix B.3.

11.3. Routing

During the third-quarter course Digital Systems A, the students were already introduced to the Wave-algorithm or Lee-algorithm. The routing algorithms used by team A4 are based on this algorithm and variations of it.

11.3.1. The Wave-algorithm

The Wave- or Lee-algorithm finds the shortest route between two points by setting the target point to value 1, and subsequently iterating over all cells in the grid with a positive value, setting neighboring cells with a value smaller than or equal to their own value to their own value +1, until the value of the starting point is changed. This algorithm is implemented in `router.c` by the `wave()` function. After applying the wave to the `Grid`, a route is traced back by beginning at the starting point and jumping to a neighboring `Cell` with the value of the current `Cell` -1, repeating this until the target point is reached. This traceback algorithm is implemented in `router.c` by the `traceback_single()` and `traceback_length()` functions.

11.3.2. Optimized routing

When tracing back a route using `traceback_single()`, any point can have multiple neighboring cells with the value of the current cell -1. This means that even though only one route is traced back, there are multiple possibilities. Since taking a turn with the robot is slower than going straight (while covering the same distance), finding a route with as few turns as possible is desirable. In order to do this, a number of extra steps are introduced in the routing algorithm. The steps for calculating a route that is both as short as possible and has as few turns as possible are as follows:

1. `wave()` is applied to the grid.

2. By `isolate_shortest()`, all cells of all possible shortest routes are set to `Cell.v = 0`; cells not on a shortest route are set to `Cell.v = -1`.
   Inside this function, `traceback_set_zero()` is used to recursively trace back all shortest routes.

   A reduced grid is now left with only the nodes, edges and the two stations contained in shortest routes between the two stations.

3. A modified, recursive version of the Wave-algorithm, `turn_wave()` is applied to the reduced grid.

4. `traceback_set_zero()` isolates the paths with fewest possible turns.

5. `wave()` is applied again to the grid, which now only consists of optimal paths which are both as short as possible and have as few turns as possible.

6. A single route is traced back by `traceback_single()` and an array of strings containing the names of all cells on the route (in order) is returned.

This optimized routing algorithm is implemented in `route_optimized()` in `routing.c`.

11.3.3. Multi-point routing

Both Challenge 1 and Challenge 2 require routing from a starting station to three stations which need to be "visited". Finding the shortest route along all three stations is akin to solving the Chinese Postman Problem, which can be complicated. However, since there are only three stations to visit, there are $3! = 6$ possible orders in which they can be visited. This allows for a brute-force solution, which calculates the total length of all six possibilities and picks (one of) the shortest option(s). This method is implemented in `postman_solve()`, which gets its name from the Chinese Postman Problem.
In `route_multipoint()`, which is used for both Challenge 1 and 2, after finding an order for the three stations with the shortest total route distance, the individual route segments between the stations are calculated with `route_optimized()`.
Because the number of possible orders grows factorially with the number of stops, the strategy of brute-forcing a solution is not scalable and will only work with a very small number of stations; while for three stations only six possibilities have to be analyzed, for six stations this number grows to 720, and for 10 stations some $10! = 3.6 \times 10^6$ possibilities.
When the robot encounters a mine during Challenge 2, the robot can use the current route point and the orientation from the global `struct Robot robot` to calculate a new (multi-point) route from its current position.

11.4. Communication and instructions

The C program can communicate with the robot via a serial interface, for which the RS-232 library by Teunis van Beelen[1] (`rs232.c`) is used. On top of this, `uart.c` contains `uart_tx()` and `uart_rx_wait()` which reduce the number of lines of code needed per instance to transmit or receive a signal, as also described in Section 8.3. Especially `uart_rx_wait()` is important, because it continuously polls the OS's RX buffer and returns once a signal is received, allowing for timely responses to the robot's feedback.
The XBee connection can not be trusted to transport 100% of signals reliably, so where possible, redundancy[2] or failsafes are implemented to catch situations where a transmitted signal is not received by the other side.

The instruction set for the robot is described in Section 8.4; the possible status signals in Section 8.5.

11.4.1. Deriving instructions from route

When a route has been calculated, instructions as described in Section 8.4 have to be derived from the route itself. This task is fulfilled by the function `derive_next_instruction()` which is defined in `main.c`. This function returns a `struct Instruction`, containing the following properties:

- `char character`: instruction character
- `int dp`: number of route points which are traversed at the execution of the instruction by the robot
- `struct Robot robot_position_after`: position of robot (which has to be given as parameter) after execution of the instruction by the robot

The function takes a `Grid`, a `struct Robot`, a route (`string` array), the length of the route and the current progress of the route traversal. The function checks the coordinates of the current, first next and, if applicable, second next points on the route, and combined with the orientation of the given `struct Robot` derives the first next needed instruction to make progress on the route. Because the function also looks at the second next route point, if existent, it is possible to detect an upcoming turn and cut the corner, or detect the visiting of a station and use one of the designated visit commands ('\', '^', '/'), saving time.

[1] `https://www.teuniz.net/RS-232/`
[2] e.g. sending a signal multiple times

12

Conclusions and Recommendations

With the end of the 2018-2019 academic year the project EPO 2 is approaching its end. The main goal of this project was to build a robot which could travel, using instructions from a computer, over a grid, as fast as possible, and detect and avoid mines. For completing this goal different control parts were described in VHDL and implemented on a FPGA board. There was also a C program needed to implement the algorithm to find the shortest route in a maze and give instructions to the robot. In the form of hardware a inductive sensor was build.

Some problems occurred during the project, one of them was the design process of the mine sensor. First there was a wrong design chosen, another problem that occurred, and the one who took most of our time, was the fact that the second design, which had to work according to the simulations, did not work. The reason for that was a feedback resistor which was too large. It was quite challenging for the team to figure out why a circuit which had to work according to the simulations did not.

It can be concluded that it was a tough project where the team learned about communication and good teamwork. Besides this a lot of knowledge about the software and hardware was gained by the team. This project was an excellent opportunity to practice the before learned theoretical VHDL and C programming knowledge during the lectures.

All in all did we test and achieved the first challenge(A) and the second one(B) will likely be finished too in time. It is not yet clear if challenge C will be finished. The robot performances quite good. It drives according to the instructions which are given by the computer with the C code. The smooth turns are also well executed. When a mine is detected, the right bit vector is send to the computer. This makes the robot well designed but not all challenged are completed. The team needed more time to complete all the challenges.

All this was doable with flawless teamwork, where the content of the different sub-tasks is known by each member. When everyone knows what had to be done, different subgroups were able to start dividing the project to achieve a result as fast as possible. Even when each subgroup does their work perfectly, it is still possible to fail the project. The reason for that is the fact that dividing a large project is a great solution if and only if the communication between the different subgroups is healthy and efficient. The group had some difficulties when bringing the parts together. A recommendation is to make sure every task and gained information is shared with the rest of the group on time. In the end the team managed to bring the parts together in the right way.

Bibliography

[1] B. Jacobs, X. van Rijnsoever, T. Slats, J. Hoekstra, A.J. van Genderen, M. Pertijs, M. Bartek, A.M.J. Slats, M. Spirito, S. Izadkhast, and T.M. de Rijk. *Lab Courses EE Semester 1 – Student Manual. Course Labs of EE1C11, EE1P11, and EE1M11.* TU Delft, 2018-2019.

A

VHDL code

A.1. Top level description of the system

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity system is
  port( clk                                    : in std_logic;
        reset                                  : in std_logic;
        sensor_l_in, sensor_m_in, sensor_r_in : in std_logic;
        motor_l_out, motor_r_out               : out std_logic
  );
end system;

architecture structural of system is
    component inputbuffer is
        port (  clk   : in  std_logic;

                sensor_l_in : in  std_logic;
                sensor_m_in : in  std_logic;
                sensor_r_in : in  std_logic;

                sensor_vector_out  : out std_logic_vector (2 downto 0)
        );
    end component;

    component controller is
        port (  clk   : in  std_logic;
                reset  : in  std_logic;

                sensor_input  : in  std_logic_vector (2 downto 0);

                count_in      : in  std_logic_vector (19 downto 0);
                count_reset   : out std_logic;

                motor_l_reset     : out std_logic;
                motor_l_direction : out std_logic_vector (2 downto 0);

                motor_r_reset     : out std_logic;
                motor_r_direction : out std_logic_vector (2 downto 0)
```

```
        );
    end component;

    component timebase is
        port (  clk   : in std_logic;
                reset   : in std_logic;

                count_out : out std_logic_vector (19 downto 0)
        );
    end component;

    component motorcontrol is
        generic (   fast_backward_ticks,
                    slow_backward_ticks,
                    slow_forward_ticks,
                    fast_forward_ticks : unsigned (19 downto 0)
        );
        port (  clk     : in  std_logic;
              reset     : in  std_logic;

            direction   : in  std_logic_vector (2 downto 0);
            count_in    : in  std_logic_vector (19 downto 0);
            pwm         : out  std_logic
        );
    end component;

  signal sensor_vector      : std_logic_vector (2 downto 0);
  signal count              : std_logic_vector (19 downto 0);
  signal count_reset,
         motor_l_reset,
         motor_r_reset      : std_logic;
  signal motor_l_direction,
         motor_r_direction  : std_logic_vector (2 downto 0);

BEGIN
input   : inputbuffer  port map (clk, sensor_l_in, sensor_m_in, sensor_r_in,
                                 sensor_vector);

control : controller   port map (clk, reset, sensor_vector, count, count_reset,
                                 motor_l_reset, motor_l_direction,
                                 motor_r_reset, motor_r_direction);
time    : timebase     port map (clk, count_reset, count);
motor_l : motorcontol
              generic map ( fast_backward_ticks => to_unsigned(100000, 20),
                            slow_backward_ticks => to_unsigned(80000, 20),
                            slow_forward_ticks  => to_unsigned(70000, 20),
                            fast_forward_ticks  => to_unsigned(50000, 20) )
              port map (clk, motor_l_reset, motor_l_direction, count, motor_l_out);
motor_r : motorcontrol
              generic map ( fast_backward_ticks => to_unsigned(50000, 20),
                            slow_backward_ticks => to_unsigned(70000, 20),
                            slow_forward_ticks  => to_unsigned(80000, 20),
                            fast_forward_ticks  => to_unsigned(100000, 20) )
              port map (clk, motor_r_reset, motor_r_direction, count, motor_r_out);

END structural;
```

A.2. Testbench of the top level

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY system_tb IS

END ENTITY system_tb;


ARCHITECTURE test OF system_tb IS

    COMPONENT system IS
        port( clk                                  : in std_logic;
            reset                                  : in std_logic;
            sensor_l_in, sensor_m_in, sensor_r_in  : in std_logic;
            motor_l_out, motor_r_out               : out std_logic
        );
    END COMPONENT system;

    SIGNAL clk, reset, sensor_l_in, sensor_m_in, sensor_r_in, motor_l_out, motor_r_out : STD_LOGIC;
    signal sensor_vector : std_logic_vector (2 downto 0);

BEGIN
    lb1: entity work.system(structural) PORT MAP (clk, reset,
                                                  sensor_l_in, sensor_m_in, sensor_r_in,
                                                  motor_l_out, motor_r_out);

    clk  <=  '1' AFTER 0 ns,
             '0' AFTER 10 ns WHEN clk /= '0' ELSE '1' AFTER 10 ns;

    reset <= '1' AFTER 0  ms,
             '0' AFTER 50 ms;

    sensor_vector <= "101" AFTER 0 ms,
                     "000" AFTER 200 ms,
                     "110" AFTER 300 ms,
                     "000" AFTER 400 ms,
                     "100" AFTER 500 ms,
                     "001" AFTER 600 ms,
                     "010" AFTER 700 ms,
                     "111" AFTER 800 ms;

    sensor_l_in <= sensor_vector(2);
    sensor_m_in <= sensor_vector(1);
    sensor_r_in <= sensor_vector(0);

END test;
```

A.3. Time base

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

--timebase has 2 inputs and 1 output.
--The output should be able to count up to 1.000.000 cycles for 20 ms.
--Therefore the count_out should have 20 bits (19 downto 0).
entity timebase is
    port (  clk         : in   std_logic;
            reset       : in   std_logic;
            count_out   : out  std_logic_vector (19 downto 0)
    );
end entity timebase;

architecture behavioral of timebase is
--two new 20-bit signals are described
    signal count, new_count  : unsigned (19 downto 0);

BEGIN
--Whenever there is a rising edge, and the reset is equal to 1,
--the count will be reset and the value '0' will be assigned to count.
    process (clk)
    begin
        if (rising_edge(clk)) then
            if (reset = '1') then
                count <= (others => '0');
            else
                count <= new_count;
            end if;
        end if;
    end process;

    process (count)
    begin
         new_count    <= count + 1;
    end process;

    count_out <= std_logic_vector (count);

END behavioral;
```

A.4. Time base testbench

```
library IEEE;
use IEEE.std_logic_1164.all;

entity testbench is
end entity testbench;

architecture structural of testbench is
    component timebase
        port (  clk       : in   std_logic;
                reset     : in   std_logic;
                count_out : out  std_logic_vector (19 downto 0)
        );
    end component timebase;

    signal clk, reset   : std_logic;
    signal count_out    : std_logic_vector (19 downto 0);

BEGIN
    lbl0: timebase port map (clk, reset, count_out);

    clk  <= '1' after 0 ms,
            '0' after 20 ms when clk /= '0'
            else '1' after 10 ms;

    reset  <=   '1' after 0 ms,
                '0' after 30 ms;

END structural;
```

A.5. Input buffer

```
library IEEE;
use IEEE.std_logic_1164.all;

entity reg is --this is the register
    port (  clk     : in  std_logic; --50 MHz
            input1  : in std_logic;
            input2  : in std_logic;
            input3  : in std_logic;

            output1 : out std_logic;
            output2 : out std_logic;
            output3 : out std_logic
    );
end entity reg;


architecture behavior of reg is --acts like a kind as a flip-flop

BEGIN
    process (clk)
    begin
        if (rising_edge (clk)) then
            output1 <= input1;
            output2 <= input2;
            output3 <= input3;
            end if;
    end process;
end architecture behavior;


-- This is the toplevel of the inputbuffer
library IEEE;
use IEEE.std_logic_1164.all;

entity inputbuffer is --top-level entity
    port (  clk             : in    std_logic; --50 MHz

            sensor_l_in     : in    std_logic;
            sensor_m_in     : in    std_logic;
            sensor_r_in     : in    std_logic;

            sensor_l_out    : out   std_logic;
            sensor_m_out    : out   std_logic;
            sensor_r_out    : out   std_logic
    );
end entity inputbuffer;

architecture structural of inputbuffer is
component reg is
    port (  clk     : in  std_logic; --50 MHz

            input1  : in std_logic;
            input2  : in std_logic;
            input3  : in std_logic;
```

```
            output1 : out std_logic;
            output2 : out std_logic;
            output3 : out std_logic
    );
end component;

signal inter1, inter2, inter3 : std_logic;
begin
    --two registers in series
    register1: reg port map (clk, sensor_l_in, sensor_m_in,
    sensor_r_in, inter1, inter2, inter3);

    register2: reg port map (clk, inter1, inter2, inter3,
    sensor_l_out, sensor_m_out, sensor_r_out);

END structural;
```

A.6. Input Buffer Testbench

```
library IEEE;
use IEEE.std_logic_1164.ALL;

entity inputbuffer_tb is
end entity inputbuffer_tb;

architecture behaviour of inputbuffer_tb is
    component inputbuffer
        port (  clk             : in    std_logic; --50 MHz

                sensor_l_in     : in    std_logic;
                sensor_m_in     : in        std_logic;
                sensor_r_in     : in        std_logic;

                sensor_l_out    : out        std_logic;
                sensor_m_out    : out        std_logic;
                sensor_r_out    : out        std_logic
        );
    end component inputbuffer;

    signal clk                              : std_logic;
    signal input1, input2, input3       : std_logic;
    signal output1, output2, output3    : std_logic;

BEGIN
lbl0: inputbuffer port map (clk, input1, input2, input3,
                                                output1, output2, output3);

    clk <=  '0' after 0 ns,
            '1' after 10 ns  when clk /= '1' else '0' after 10 ns;
    input1 <='0' after 0 ns,
            '1' after 52 ns,
            '0' after 85 ns,
            '1' after 123 ns;

    input2 <='0' after 0 ns,
            '0' after 52 ns,
            '0' after 85 ns,
            '1' after 123 ns;

    input3 <=       '0' after 0 ns,
            '0' after 52 ns,
            '1' after 85 ns,
            '1' after 123 ns;

END behaviour;
```

A.7. Motor control

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity motorcontrol is
    -- This generic specifies how many clock periods the PWM pulse should take
    generic(    fast_backward_ticks,
                slow_backward_ticks,
                slow_forward_ticks,
                fast_forward_ticks : unsigned (19 downto 0)
    );

    port (  clk     : in  std_logic;
            reset     : in  std_logic;
            --encoding of direction is explained in chapter \ref{chapter:motorcontrol}
            direction  : in  std_logic_vector (2 downto 0);
            count_in   : in  std_logic_vector (19 downto 0);

            pwm        : out  std_logic
    );
end entity;

architecture behavioral of motorcontrol is
    --states are explained in chapter \ref{chapter:motorcontrol}
    type motorcontrol_state is (    reset_state,
                                    high_state,
                                    sleep_state
                                );
    signal state, new_state : motorcontrol_state;

BEGIN

    process (state, count_in, direction)
    begin
        case state is
            when reset_state => --before every new pulse
                pwm <= '0';
                new_state <= high_state;
            when high_state => --start new pulse
                pwm <= '1';
                if (direction = "110") and (unsigned(count_in) = fast_backward_ticks)
                --for going backwards quickly
                then new_state <= sleep_state;
                elsif (direction = "100") and (unsigned(count_in) = slow_backward_ticks)
                --for going backwards slowly
                then new_state <= sleep_state;
                elsif (direction = "000")
                -- standstill
                then   new_state <= sleep_state;
                elsif (direction = "001") and (unsigned(count_in) = slow_forward_ticks)
                ----for going forwards slowly
                then new_state <= sleep_state;
                elsif (direction = "011") and (unsigned(count_in) = fast_forward_ticks)
                --for going forwards quickly
                then    new_state <= sleep_state;
```

```
                else --direction unclear, pwm stays high
                    new_state <= high_state;
                end if;
            when sleep_state => --after PWM pulse, still in PWM period
                pwm <= '0';
                            new_state <= sleep_state;
        end case;
    end process;

    process (clk) --adopt new state
    begin
        if (rising_edge (clk)) then
            if (reset = '1') then
                state <= reset_state;
            else
                state <= new_state;
            end if;
        end if;
    end process;

END behavioral;
```

A.8. Motor control testbench

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity motorcontrol_tb is
end entity;

architecture behavioral of motorcontrol_tb is
        component motorcontrol
                generic (   fast_backward_ticks,
                              slow_backward_ticks,
                              slow_forward_ticks,
                              fast_forward_ticks : unsigned (19 downto 0)
        );

                port (      clk         : in  std_logic;
                            reset       : in  std_logic;

                            direction   : in  std_logic_vector (2 downto 0);
                            count_in    : in  std_logic_vector (19 downto 0);

                            pwm         : out  std_logic
            );
        end component;

        component timebase
            port (            clk                   : in        std_logic;
                              reset                 : in        std_logic;
                              count_out        : out        std_logic_vector (19 downto 0)
                );
        end component;

        signal clk, reset               : std_logic;
        signal direction                : std_logic_vector(2 downto 0);
        signal count                       : std_logic_vector(19 downto 0);
        signal pwm                          : std_logic;

BEGIN
        lbl0: motorcontrol        generic map (   --1 ms for fast_backward_ticks:
                                                  to_unsigned(50000, 20),
                                                  --1.2 ms for slow_backward_ticks:
                                                  to_unsigned(60000, 20),
                                                  --1.7 ms for slow_forward_ticks:
                                                  to_unsigned(85000, 20),
                                                  --2.0 ms for fast_forward_ticks:
                                                  to_unsigned(100000, 20)
                                             )
                                              port map(clk, reset, direction, count, pwm);

        lbl1: timebase                         port map(clk, reset, count);

        clk <=                      '0' after 0 ns,
                                   '1' after 10 ns  when clk /= '1' else '0' after 10 ns;

                    --acts as a stand-in for the reset that is sent every 20 ms
```

```
                    -- by the controller
        reset <=            '1' after 0 ns,
                                '0' after 40 ns,
                                '1' after 20 ms,
                                '0' after (20 ms + 40 ns),
                                '1' after 40 ms,
                                '0' after (40 ms + 40 ns),
                                '1' after 60 ms,
                                '0' after (60 ms + 40 ns);


        direction <=         "011" after 0 ms, --fast fwd
                                "001" after 25 ms, --slow fwd
                                "110" after 45 ms; --fast bwd

END behavioral;
```

A.9. Controller

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity controller is
 port (
        clk  : in std_logic;
        reset                                   : in std_logic;
        mine_detected                 : in std_logic; --'1' if mine is detected else '0'
        enable_sensor                 : in std_logic; --'1' to disable line follower
        flag                                    : in std_logic;        --'1' if signal is received
        sensor_input             : in std_logic_vector (2 downto 0); --'1' if white else black
        count_in                 : in std_logic_vector (19 downto 0); --count from timebase
        in_signal                          : in std_logic_vector(7 downto 0); --incomming signal
        turn_counter               : in unsigned(27 downto 0); --count from externaltimer

        mine_reset                          : out std_logic;
        motor_r_reset              : out std_logic;
        motor_l_reset                 : out std_logic;
        turn_counter_disable : out std_logic; --to disable externaltimer
        count_reset              : out std_logic; --to reset timebase
        write_data                          : out std_logic; --to write with UART
        read_data                          : out std_logic; --to read with UART
            motor_r_direction          : out std_logic_vector (2 downto 0);
        motor_l_direction    : out std_logic_vector (2 downto 0);
        out_signal                 : out std_logic_vector (7 downto 0)--a bitvector to send with UART
 );
end controller;

architecture behavioral of controller is

type controller_state is (          check_state,
                                    standstill_state,

                                    straight_state,
                                    slow_straight_state,

                                    hard_right_state,
                                    right_state,
                                    fast_inplace_right_state,
                                    inplace_right_state,

                                    hard_left_state,
                                    left_state,
                                    fast_inplace_left_state,
                                    inplace_left_state,

                                                        anti_hard_left_state,
                                                        anti_hard_right_state,

                                                        backward_state,
                                                        error_state);

type direction is (straight, left, right, standstill, error);
```

```
signal state, new_state : controller_state;
signal last_direction, last_direction_int : direction;
signal turn, out_signal_s, new_signal : std_logic_vector(7 downto 0);
signal count_reset_s : std_logic;
signal disable_s, new_disable_s : std_logic;
signal sensor_input_s: std_logic_vector(2 downto 0);

begin
    process (state, sensor_input, last_direction, turn_counter, enable_sensor, turn, in_signal, n
    begin
        case state is
        when check_state =>         --the reset state where the movement gets decided
                count_reset_s <= '1';         --the time base gets reset
                motor_l_reset <= '1';         --no movement is done in this state
                motor_r_reset <= '1';
                motor_l_direction <= "000";
                motor_r_direction <= "000";
                last_direction_int <= standstill;

                if (last_direction /= error) then
                --to make sure that even if he moves he stays in the error state
                        sensor_input_s <= sensor_input;
                else
                        sensor_input_s <= "000";
                end if;


                if(enable_sensor='1') then --if the linefollower is enabled
                        case sensor_input_s is
                                when "101" =>      -- forward
                                        read_data <= '0';
                                        out_signal_s <= "00000000";
                                        disable_s <= new_disable_s;
                                        turn <= new_signal;
                                        new_state <= straight_state;
                                when "100" =>      -- a bit to left
                                        read_data <= '0';
                                        out_signal_s <= "00000000";
                                        disable_s <= new_disable_s;
                                        turn <= new_signal;
                                        new_state <= right_state;
                                when "001" =>      -- a bit to right
                                        read_data <= '0';
                                        out_signal_s <= "00000000";
                                        disable_s <= new_disable_s;
                                        turn <= new_signal;
                                        new_state <= left_state;
                                when "110" =>      -- more to right
                                        read_data <= '0';
                                        out_signal_s <= "00000000";
                                        disable_s <= new_disable_s;
                                        turn <= new_signal;
                                        new_state <= hard_right_state;
                                when "011" =>      -- more to left
                                        read_data <= '0';
                                        out_signal_s <= "00000000";
```

```
                                disable_s <= new_disable_s;
                                turn <= new_signal;
                                new_state <= hard_left_state;
                            when "000" =>         -- intersection
                                if(flag='1') then -- if a signal is received
                                        read_data <= '1'; -- read the data and reset flag
                                        if(mine_detected = '0')then -- no mine detected
                                                out_signal_s <= "01011000"; --send 'X'
                                                turn <= in_signal;
                                        else --mine deteced
                                                out_signal_s <= "01001101"; --send 'M'
                                                turn <= "00101000";
                                    end if;
                                                disable_s <= '0';
                                                new_state <= standstill_state;

                                        else -- not yet received a signal
                                                out_signal_s <= "01010111"; --send 'W'
                                                read_data <= '0';
                                                disable_s <= new_disable_s;
                                                turn <= new_signal;
                                                new_state <= error_state;
                                        end if;
                            when others =>    -- lost
                                read_data <= '0';
                                out_signal_s <= "00000000";
                                disable_s <= new_disable_s;
                                turn <= new_signal;
                                case last_direction is
                                -- smart new_state assignment when lost
                                        when straight =>
                                            new_state <= slow_straight_state;
                                        when left =>
                                                new_state <= inplace_left_state;
                                        when right =>
                                                new_state <= inplace_right_state;
                                        when standstill =>
                                                new_state <= standstill_state;
                                        when others =>
                                                new_state <= standstill_state;
                                        end case;
                                end case;
        else
-- if the line follower is not enabled the robot will follow a path according to a timer
-- and a given signal by the computer
                read_data <= '0';
                out_signal_s <= "00000000";
                turn <= new_signal;
                case turn is
                    when "01111101" =>        --Right
                                if(turn_counter < to_unsigned(42000000,28)) then
                                        disable_s <= new_disable_s;
                                        new_state <= straight_state;
                                elsif(turn_counter < to_unsigned(156000000,28)) then
                                        disable_s <= new_disable_s;
                                        new_state <= right_state;
```

```
                else
                        disable_s <= '1'; --turn line follower back on
                        new_state <= right_state;

                end if;
        when "01111011" =>          --Left
                if(turn_counter < to_unsigned(42000000,28)) then
                        disable_s <= new_disable_s;
                        new_state <= straight_state;
                elsif(turn_counter < to_unsigned(156000000,28)) then
                        disable_s <= new_disable_s;
                        new_state <= left_state;
                else
                        disable_s <= '1'; --turn line follower back on
                        new_state <= left_state;
                end if;
        when "01111100" =>                  --Forward
                if(turn_counter < to_unsigned(20000000,28)) then
                        disable_s <= new_disable_s;
                        new_state <= straight_state;
                else
                        disable_s <= '1'; --turn line follower back on
                        new_state <= straight_state;
                end if;
        when "00111110" =>                  --Hard turn right fixed
                if(turn_counter < to_unsigned(20000000,28)) then
                        disable_s <= new_disable_s;
                        new_state <= fast_inplace_right_state;
                elsif(turn_counter < to_unsigned(40000000,28)) then
                        disable_s <= new_disable_s;
                        new_state <= hard_right_state;
                else
                        disable_s <= '1'; --turn line follower back on
                        new_state <= hard_right_state;
                end if;
        when "00111100" =>                  --Hard turn left fixed
                if(turn_counter < to_unsigned(20000000,28)) then
                        disable_s <= new_disable_s;
                        new_state <= fast_inplace_left_state;
                elsif(turn_counter < to_unsigned(40000000,28)) then
                        disable_s <= new_disable_s;
                        new_state <= hard_left_state;
                else
                        disable_s <= '1'; --turn line follower back on
                        new_state <= hard_left_state;
                end if;

        when "00101000" =>                  -- Turn 180 degrees left
                if(turn_counter < to_unsigned(110000000,28)) then
                        disable_s <= new_disable_s;
                        new_state <= fast_inplace_left_state;

                else
                        disable_s <= '1'; --turn line follower back on
                        new_state <= straight_state;
                end if;
```

```
                        when "00101001" =>                -- Turn 180 degrees right
                                if(turn_counter < to_unsigned(100000000,28)) then
                                disable_s <= new_disable_s;
                                        new_state <= fast_inplace_right_state;

                                else
                                        disable_s <= '1'; --turn line follower back on
                                        new_state <= straight_state;
                                end if;
                        when "01011110" =>                -- visit forward
                                if(turn_counter < to_unsigned(20000000,28)) then
                                        disable_s <= new_disable_s;
                                        new_state <= straight_state;
                                elsif(turn_counter < to_unsigned(60000000,28)) then
                                        disable_s <= new_disable_s;
                                        new_state <= backward_state;
                                else
                                        disable_s <= '1'; --turn line follower back on
                                        new_state <= straight_state;
                                end if;

                        when "01011100" =>  -- visit left station
                                  if(turn_counter < to_unsigned(80000000,28)) then
                                        disable_s <= new_disable_s;
                                        new_state <= hard_left_state;

                                  elsif( turn_counter < to_unsigned(160000000,28)) then
                                        disable_s <= new_disable_s;
                                        new_state <= anti_hard_left_state;
                                         else
                                        disable_s <= '1'; --turn line follower back on
                                        new_state <= straight_state;
                                  end if;

                        when "00101111" =>  -- visit right station
                                  if(turn_counter < to_unsigned(80000000,28)) then
                                        disable_s <= new_disable_s;
                                        new_state <= hard_right_state;

                                  elsif(turn_counter < to_unsigned(160000000,28)) then
                                        disable_s <= new_disable_s;
                                        new_state <= anti_hard_right_state;
                                        else
                                        disable_s <= '1';
                                        new_state <= straight_state;
                                  end if;

                        when others =>                  -- forward
                                disable_s <= '1'; --turn line follower back on
                                new_state <= standstill_state;
                end case;
        end if;
        write_data <= '1';        --the robot sends a signal

when standstill_state =>
        sensor_input_s <= sensor_input;
```

```
            read_data <= '0';
            out_signal_s <= "00000000";
            write_data <= '0';
            turn <= new_signal;
            disable_s <= new_disable_s;
            count_reset_s <= '0';
            motor_l_reset <= '0';
            motor_r_reset <= '0';

            motor_l_direction <= "000";
            motor_r_direction <= "000";

            last_direction_int <= standstill;
            new_state <= standstill_state;

        when straight_state =>
            sensor_input_s <= sensor_input;
            read_data <= '0';
            out_signal_s <= "00000000";
            write_data <= '0';

            turn <= new_signal;
            disable_s <= new_disable_s;
            count_reset_s <= '0';
            motor_l_reset <= '0';
            motor_r_reset <= '0';

            motor_l_direction <= "110";
            motor_r_direction <= "110";

            last_direction_int <= straight;
            new_state <= straight_state;

        when backward_state =>
            sensor_input_s <= sensor_input;
            read_data <= '0';
            out_signal_s <= "00000000";
            write_data <= '0';

            turn <= new_signal;
            disable_s <= new_disable_s;
            count_reset_s <= '0';

            motor_l_reset <= '0';
            motor_r_reset <= '0';
            motor_l_direction <= "011";
            motor_r_direction <= "011";

            last_direction_int <= straight;
            new_state <= backward_state;

        when slow_straight_state =>
            sensor_input_s <= sensor_input;
            read_data <= '0';
            out_signal_s <= "00000000";
            write_data <= '0';
```

```
            turn <= new_signal;
            disable_s <= new_disable_s;
            count_reset_s <= '0';

            motor_l_reset <= '0';
            motor_r_reset <= '0';
            motor_l_direction <= "100";
            motor_r_direction <= "100";
            last_direction_int <= straight;

            new_state <= slow_straight_state;

        when hard_right_state =>
            sensor_input_s <= sensor_input;
            read_data <= '0';
            out_signal_s <= "00000000";
            write_data <= '0';

            turn <= new_signal;
            disable_s <= new_disable_s;
            count_reset_s <= '0';

            motor_l_reset <= '0';
            motor_r_reset <= '0';
            motor_l_direction <= "110";
            motor_r_direction <= "000";

            last_direction_int <= right;
            new_state <= hard_right_state;

        when right_state =>
            sensor_input_s <= sensor_input;
            read_data <= '0';
            out_signal_s <= "00000000";
            write_data <= '0';

            turn <= new_signal;
            disable_s <= new_disable_s;
            count_reset_s <= '0';

            motor_l_reset <= '0';
            motor_r_reset <= '0';
            motor_l_direction <= "110";
            motor_r_direction <= "100";

            last_direction_int <= right;
            new_state <= right_state;

        when fast_inplace_right_state =>
            sensor_input_s <= sensor_input;
            read_data <= '0';
            out_signal_s <= "00000000";
            write_data <= '0';

            turn <= new_signal;
```

```
            disable_s <= new_disable_s;
            count_reset_s <= '0';

            motor_l_reset <= '0';
            motor_r_reset <= '0';
            motor_l_direction <= "110";
            motor_r_direction <= "011";

            last_direction_int <= right;
            new_state <= fast_inplace_right_state;

        when inplace_right_state =>
            sensor_input_s <= sensor_input;
            read_data <= '0';
            out_signal_s <= "00000000";
            write_data <= '0';

            turn <= new_signal;
            disable_s <= new_disable_s;
            count_reset_s <= '0';

            motor_l_reset <= '0';
            motor_r_reset <= '0';
            motor_l_direction <= "100";
            motor_r_direction <= "001";

            last_direction_int <= right;
            new_state <= inplace_right_state;

        when hard_left_state =>
            sensor_input_s <= sensor_input;
            read_data <= '0';
            out_signal_s <= "00000000";
            write_data <= '0';

            turn <= new_signal;
            disable_s <= new_disable_s;
            count_reset_s <= '0';

            motor_l_reset <= '0';
            motor_r_reset <= '0';
            motor_l_direction <= "000";
            motor_r_direction <= "110";

            last_direction_int <= left;
            new_state <= hard_left_state;

        when left_state =>
            sensor_input_s <= sensor_input;
            read_data <= '0';
            out_signal_s <= "00000000";
            write_data <= '0';

            turn <= new_signal;
            disable_s <= new_disable_s;
```

```
                count_reset_s <= '0';
                motor_l_reset <= '0';
                motor_r_reset <= '0';
                motor_l_direction <= "100";
                motor_r_direction <= "110";

                last_direction_int <= left;
                new_state <= left_state;

    when fast_inplace_left_state =>
                sensor_input_s <= sensor_input;
                read_data <= '0';
                out_signal_s <= "00000000";
                write_data <= '0';

                turn <= new_signal;
                disable_s <= new_disable_s;
                count_reset_s <= '0';

                motor_l_reset <= '0';
                motor_r_reset <= '0';
                motor_l_direction <= "011";
                motor_r_direction <= "110";

                last_direction_int <= left;
                new_state <= fast_inplace_left_state;

        when inplace_left_state =>
                sensor_input_s <= sensor_input;
                read_data <= '0';
                out_signal_s <= "00000000";
                write_data <= '0';

                turn <= new_signal;
                disable_s <= new_disable_s;
                count_reset_s <= '0';

                motor_l_reset <= '0';
                motor_r_reset <= '0';
                motor_l_direction <= "001";
                motor_r_direction <= "100";

                last_direction_int <= left;
                new_state <= inplace_left_state;

        when error_state =>
                sensor_input_s <= sensor_input;
                read_data <= '0';
                out_signal_s <= "00000000";
                write_data <= '0';

                turn <= new_signal;
                disable_s <= new_disable_s;
                count_reset_s <= '0';

                motor_l_reset <= '0';
```

```
motor_r_reset <= '0';
motor_l_direction <= "000";
motor_r_direction <= "000";

last_direction_int <= error;
new_state <= error_state;

when anti_hard_right_state =>
                sensor_input_s <= sensor_input;
                read_data <= '0';
                out_signal_s <= "00000000";
                write_data <= '0';
                turn <= new_signal;
                disable_s <= new_disable_s;
                                count_reset_s <= '0';
                                motor_l_reset <= '0';
                                motor_r_reset <= '0';

                                motor_l_direction <= "011";
                                motor_r_direction <= "000";
                                last_direction_int <= left;
                new_state <= anti_hard_right_state;

 when anti_hard_left_state =>
                                sensor_input_s <= sensor_input;
                                read_data <= '0';
                                out_signal_s <= "00000000";
                                write_data <= '0';
                                turn <= new_signal;
                                disable_s <= new_disable_s;
                                 count_reset_s <= '0';
                                 motor_l_reset <= '0';
                                 motor_r_reset <= '0';

                                 motor_l_direction <= "000";
                                 motor_r_direction <= "011";
                                 last_direction_int <= right;
                                new_state <= anti_hard_left_state;


        when others =>
                sensor_input_s <= sensor_input;
                read_data <= '0';
                out_signal_s <= "00000000";
                write_data <= '0';

                turn <= new_signal;
                disable_s <= new_disable_s;
                                count_reset_s <= '0';

                                motor_l_reset <= '0';
                                motor_r_reset <= '0';
                                motor_l_direction <= "000";
                                motor_r_direction <= "000";

                                last_direction_int <= standstill;
```

```
                                new_state <= standstill_state;
        end case;
    end process;

process (clk, count_in, reset)
begin
if (rising_edge (clk)) then
        if (reset = '1') then --synchronous reset
                state <= check_state;
                mine_reset <= '0';
                last_direction <= standstill;
                new_disable_s <= '1';
                new_signal <= (others=>'0');
        elsif (unsigned(count_in) = to_unsigned(9, 20)) then --at 20 ms the timebase will be reset
                state <= check_state;
                mine_reset <= '1';
                new_signal <= turn;
                new_disable_s <= disable_s;
                last_direction <= last_direction_int;
        else --else just assign the new state at clockpulse
                state <= new_state;
                mine_reset <= '0';
                new_signal <= turn;
                new_disable_s <= disable_s;
                last_direction <= last_direction_int;
        end if;
end if;
end process;

count_reset <= count_reset_s;
turn_counter_disable <= disable_s;
out_signal <= out_signal_s;
end architecture behavioral;
```

A.10. Controller_tb

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity controller_tb is
end entity controller_tb;

architecture testbench of controller_tb is

component controller is
    port ( clk   : in  std_logic;
         reset   : in  std_logic;

        sensor_input            : in  std_logic_vector (2 downto 0);

        count_in                : in  std_logic_vector (19 downto 0);
        count_reset             : out std_logic;

        motor_l_reset               : out  std_logic;
        motor_l_direction           : out  std_logic_vector (2 downto 0);

        motor_r_reset               : out  std_logic;
        motor_r_direction           : out  std_logic_vector (2 downto 0);

        out_signal                    : out std_logic_vector (7 downto 0);
        in_signal                : in std_logic_vector(7 downto 0);

        turn_counter                : in integer;
        enable_sensor                : in std_logic;
        turn_counter_disable        : out std_logic
    );
end component controller;

component timebase is
    port ( clk    : in std_logic;
           reset  : in std_logic;

                count_out : out std_logic_vector (19 downto 0)
            );
end component timebase;

component externaltimer is
        port (
                clk:                        in std_logic;
                turn_counter_disable:         in std_logic;
                turn_counter:                 out integer;
                enable_sensor:                 out std_logic
        );
end component externaltimer;

SIGNAL clk, reset, count_reset : STD_LOGIC;
SIGNAL sensor_input : STD_LOGIC_VECTOR (2 downto 0);
SIGNAL in_signal, out_signal : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL count : STD_LOGIC_VECTOR (19 downto 0);
```

```
-- out
SIGNAL motor_l_reset, motor_r_reset : STD_LOGIC;
SIGNAL motor_l_direction, motor_r_direction : STD_LOGIC_VECTOR (2 downto 0);
SIGNAL turn_counter : INTEGER;
SIGNAL enable_sensor : STD_LOGIC;
SIGNAL turn_counter_disable : STD_LOGIC;

begin

control : controller   port map (clk, reset, sensor_input, count, count_reset, motor_l_reset, motor_l_d
times           : timebase     port map (clk, count_reset, count);
turn_counter_L        : externaltimer port map (clk, turn_counter_disable, turn_counter, enable_sensor)

    in_signal <= "01010010";
    clk  <=  '1' AFTER 0 ns,
             '0' AFTER 10 ns WHEN clk /= '0' ELSE '1' AFTER 10 ns;

    reset <= '1' AFTER 0  ms,
             '0' AFTER 50 ms;

    sensor_input <= "101" AFTER 0 ms,
                    "000" AFTER 200 ms,
                    "110" AFTER 300 ms,
                    "000" AFTER 400 ms,
                    "100" AFTER 500 ms,
                    "001" AFTER 600 ms,
                    "010" AFTER 700 ms,
                    "111" AFTER 800 ms,
                    "000" AFTER 1000 ms;
end testbench;
```

A.11. Minesensor sensor

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity minesensor is
    port (  clk         : in        std_logic;
                        reset               : in        std_logic;
                        sensor      : in        std_logic;

                        mine            : out       std_logic);
end entity minesensor;

architecture behavioural of minesensor is

        signal count, new_count                         : unsigned(19 downto 0);
        signal sensor_between, sensor_out       : std_logic;

--With every rising edge of the clock the input of the sensor is buffered twice.
--If the sensor after the buffers and the reset both give a '1' every value is is set to '0'.
--Otherwise count becomes new_count.
        begin
                process (clk, sensor, reset)
                            begin
                                if (rising_edge (clk)) then
                                        sensor_between <= sensor;
                                    sensor_out <= sensor_between;
                                            if(sensor_out = '1' or reset = '1') then
                                                        count <= (others => '0');
                                              else
                                                    count <= new_count;
                                                      end if;
                                end if;
                end process;

--new_count is set to count+1 to count the
--amount of clockperiods fit in one period of the sensor.
--If count is higher a mine is detected.
--It count is lower no mine is detected.
                process(count)
                        begin
                                new_count <= count + 1;
                                        if (count > 750) then
                                           mine <= '1';
                                          else
                                           mine <= '0';
                                          end if;
                end process;


end behavioural;
```

A.12. Minesensor states

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity minestates is
    port (    clk                 : in    std_logic;
              reset                  : in    std_logic;
              minestate                 : in    std_logic;

              mine_detected    : out   std_logic);
end entity minestates;
architecture behavioural of minestates is

    type detector_state is (mine_state, nomine_state);
    signal state, new_state     : detector_state;

    begin
--When resetted state goes back to nomine_state.
 process(clk,reset)
 begin
   if(reset = '1') then
                      state <= nomine_state;
          elsif(rising_edge(clk)) then
                      state <= new_state;
          end if;
 end process;


 process(clk, reset, state, new_state, minestate)
 begin
  case state is
            when mine_state =>
--When in mine_state, mine_detected has an output of '1'.
          mine_detected <= '1';
                        new_state <= mine_state;
--When in mine_state, the state stays mine_state.

            when nomine_state =>
                        mine_detected <= '0';
--when in nomine_state, mine_detected has an output of '0'
                          if(minestate = '1') then
--When a mine is detected in the sensorcounter the new state is mine_state.
                                    new_state <= mine_state;
                          else
--Otherwise it stays in nomine_state.
                                    new_state <= nomine_state;
                          end if;
  end case;
 end process;

end behavioural;
```

A.13. Minesensor toplevel

```
library IEEE;
use IEEE.std_logic_1164.all;

entity toplevel is

        port (  clk                 : in std_logic;
                        reset               : in std_logic;
                        sensor               : in std_logic;

                        mine_detected       : out std_logic
        );
end entity toplevel;

architecture toplevel of toplevel is

        component minesensor is
        port (  clk                 : in std_logic;
                        reset               : in std_logic;
                        sensor               : in std_logic;

                        mine               : out std_logic
        );
        end component minesensor;

        component minestates is
        port (  clk                 : in std_logic;
                        reset               : in std_logic;
                        minestate        : in std_logic;

                        mine_detected       : out std_logic
        );
        end component minestates;

        signal mine_s: std_logic;


        BEGIN
--The working between the minesensor
--and minestates is shown in Figure \ref{fig:toplevel minesensor}
L1:         minesensor port map( clk => clk,
                                                        reset => reset,
                                                        sensor => sensor,

                                                        mine => mine_s
                                                );
L2:          minestates port map( clk => clk,
                                                           reset => reset,
                                                           minestate => mine_s,
                                                           mine_detected => mine_detected
                                                );

end architecture;
```

A.14. Minesensor toplevel testbench

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity toplevel_tb is
end entity toplevel_tb;

architecture beschrijving of toplevel_tb is
        component toplevel_test1 is
        port (          clk         : in        std_logic;
                        reset         : in        std_logic;
                       sensor    : in         std_logic;
                       mine_detected         : out        std_logic);
        end component toplevel_test1;

        signal clk, sensor, reset, mine_detected                : std_logic;

        begin



                clk <=  '0' after 0 ns,
                              '1' after 10 ns when clk /= '1' else '0' after 10 ns;

                sensor <=         '0' after 0 ns,
                                 '1' after 62 us,
                                '0' after 124 us,
                                 '1' after 186 us,
                                '0' after 248 us,
                                '1' after 312 us,
                                '0' after 376 us,
                                '1' after 440 us,
                                '0' after 504 us;

                reset <=   '1' after 0 ns,
                           '0' after 40 ns,
                           '1' after 20 ms,
                           '0' after 20.00001 ms;
        L1: toplevel_test1 port map(clk, reset, sensor, mine_detected);

end architecture beschrijving;
```

B

C code

B.1. main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "router.h"
#include "robot.h"
#include "utils.h"
#include "uart.h"
#include "rs232.h"

int com_portnr;
int com_baudrate = 9600;
char com_mode[]={'8','N','1',0};

Grid main_grid;

void challenge_1(Grid grid, char* start_station, char stop_names[3][6]);
void challenge_2(Grid grid, char* start_station, char stop_names[3][6]);
void challenge_3(Grid grid, char* start_station);

struct Instruction {
    char character;
    int dp;
    struct Robot robot_position_after;
};

char* route_derive_instructions(Grid grid, struct Robot simrobot, char** route, int length);
struct Instruction derive_next_instruction(Grid grid, struct Robot simrobot, char** route,
                                           int length, int progress);
void send_instruction(int cport_nr, char instruction);

#ifndef _GUI_

int main (void)
{
    int challenge_nr;
    char start_station[6];
    char stop_names[3][6];
```

```
generate_stdgrid(main_grid); /* generate standard grid and copy into cell matrix */

printf("Enter serial port number for XBee module (COM[n-1] or /dev/ttyUSB[n + 16]): ");
scanf("%d", &com_portnr);

/* open serial port: if serial port number 42 is used, this is ignored for test purposes */
if(com_portnr != 42 && RS232_OpenComport(com_portnr, com_baudrate, com_mode))
{
    printf("Can not open serial port\n");

    return(0);
}

printf("Enter start station: ");
scanf("%s", start_station);

/* get coordinates of start station */
if (find_cell(start_station, main_grid, &robot.location.x, &robot.location.y)) {
    printf("main(): cell '%s' not found on grid\n", start_station);
    exit(1);
}

/* set global Robot's orientation based on starting station */
robot.orientation.dx = robot.orientation.dy = 0;
if (robot.location.x == 0)
    robot.orientation.dx = 1;
else if (robot.location.x == 10)
    robot.orientation.dx = -1;
else if (robot.location.y == 0)
    robot.orientation.dy = 1;
else if (robot.location.y == 10) {
    robot.orientation.dy = -1;
}

printf("Choose challenge (1-3): ");
scanf("%d", &challenge_nr);

printf("\n");

switch (challenge_nr) {
    case 1:
        printf("Enter the names of the three stations to visit: ");
        scanf("%s %s %s", stop_names[0], stop_names[1], stop_names[2]);

        printf("\n");

        challenge_1(main_grid, start_station, stop_names);

        break;

    case 2:
        printf("Enter the names of the three stations to visit: ");
        scanf("%s %s %s", stop_names[0], stop_names[1], stop_names[2]);

        printf("\n");
```

```
            challenge_2(main_grid, start_station, stop_names);

            break;

        case 3:
            challenge_3(main_grid, start_station);

            break;

        default:
            printf("Invalid challenge number\n");
            return(1);
    }

    return(0);
}

#endif

/* executes challenge 1 */
void challenge_1(Grid grid, char* start_station, char stop_names[3][6])
{
    int i, p = 0, j;
    int total_length = 0;
    char **route;
    char *instructions, current_instruction[2] = "", rx_buffer[16];
    struct Instruction current_instruction_struct;

    route = route_multipoint(grid, start_station, stop_names, 3, &total_length);

    printf("\n[start] ");
    for (i = 0; i <= total_length; i++) {
        printf("%s -> ", route[i]);
    }
    printf("\b\b\b[end]\n");

    instructions = route_derive_instructions(grid, robot, route, total_length);

    printf("%ld instructions: [ %s ]\n", strlen(instructions), instructions);

    /* calculate and ignore first instruction */
    current_instruction_struct = derive_next_instruction(grid, robot, route, total_length, p);
    p += current_instruction_struct.dp;
    robot = current_instruction_struct.robot_position_after;

    i = 1;  /* skip first instruction */
    current_instruction_struct = derive_next_instruction(grid, robot, route, total_length, p);
    current_instruction[0] = current_instruction_struct.character;

    uart_rx_clear(com_portnr);  /* clear all waiting signals from UART buffer */

    while (p < total_length)
    {
        do {
            uart_rx_wait(com_portnr, rx_buffer, 2);
```

```
            printf("%s", rx_buffer);
        } while (rx_buffer[0] != 'X' && rx_buffer[0] != 'W');
        printf("\n");

        switch (rx_buffer[0])
        {
            case 'W':
                printf("NOTICE: last transmission failed\n");

                current_instruction[0] = current_instruction_struct.character;

                j = 0;
                printf("resending '%s' ", current_instruction);
                do {
                    uart_tx(com_portnr, current_instruction, 0);
                    j++;

                    uart_rx_wait(com_portnr, rx_buffer, 2);
                } while (rx_buffer[0] == 'W');
                printf("%d times\n", j);

                if (rx_buffer[0] != 'X' && rx_buffer[0] != '.') {
                    printf("WARNING: received unexpected response from robot: '%s'\n", rx_buffer)
                    break;
                }

            case 'X':
                /* apply progress delta and robot location delta */
                p += current_instruction_struct.dp;
                robot = current_instruction_struct.robot_position_after;
                i++;
                if (p >= total_length) break;

                /* derive next instruction */
                current_instruction_struct = derive_next_instruction(grid, robot, route,
                                                                     total_length, p);

                send_instruction(com_portnr, current_instruction_struct.character);

                break;

            default:
                printf("WARNING: received unexpected response from robot: '%s'\n", rx_buffer);
        }
    }

    free(route);
    free(instructions);
}

/* executes challenge 2 */
void challenge_2(Grid grid, char* start_station, char stop_names[3][6])
{
    int i, p = 0, j, sx, sy;
    int total_length = 0, length_to_go, n_turns = 0, n_visited = 0;
    char stations_visited[2][6] = {"", ""}, stations_to_visit[3][6] = {"", "", ""}, location[6];
```

```
    char **route, *instructions, current_instruction[2] = "", rx_buffer[16];
    struct Instruction current_instruction_struct;

    for (i = 0; i < 3; i++)
    {
        strcpy(stations_to_visit[i], stop_names[i]);
    }

    route = route_multipoint(grid, start_station, stations_to_visit, 3, &total_length);

    printf("\n[start] ");
    for (i = 0; i <= total_length; i++) {
        printf("%s -> ", route[i]);
    }
    printf("\b\b\b[end]\n");

    instructions = route_derive_instructions(grid, robot, route, total_length);

    printf("%ld instructions: [ %s ]\n", strlen(instructions), instructions);

    length_to_go = total_length;

    /* calculate and ignore first instruction */
    current_instruction_struct = derive_next_instruction(grid, robot, route, total_length, p);
    p += current_instruction_struct.dp;
    length_to_go -= current_instruction_struct.dp;
    robot = current_instruction_struct.robot_position_after;

    i = 1;  /* skip first instruction */
    current_instruction_struct = derive_next_instruction(grid, robot, route, total_length, p);
    current_instruction[0] = current_instruction_struct.character;

    uart_rx_clear(com_portnr);  /* clear all waiting signals from UART buffer */

    while (length_to_go > 0)
    {
        strcpy(location, route[p]);

        do {
            uart_rx_wait(com_portnr, rx_buffer, 2);
            printf("%s", rx_buffer);
        } while (rx_buffer[0] != 'X' && rx_buffer[0] != 'W' && rx_buffer[0] != 'M');
        printf("\n");

        switch (rx_buffer[0])
        {
            case 'W':
                printf("NOTICE: last transmission failed\n");

                current_instruction[0] = current_instruction_struct.character;

                j = 0;
                printf("resending '%s' ", current_instruction);
                do {
                    uart_tx(com_portnr, current_instruction, 0);
                    j++;
```

```
            uart_rx_wait(com_portnr, rx_buffer, 2);
        } while (rx_buffer[0] == 'W');
        printf("%d times\n", j);

        if (rx_buffer[0] != 'X' && rx_buffer[0] != '.') {
            printf("WARNING: received unexpected response from robot: '%s'\n", rx_buffer);
            break;
        }

    case 'X':   /* robot at middle of edge or junction */
        /* check if current move is station visit and add to list of visited stations */
        if (current_instruction_struct.character == '^'
            || current_instruction_struct.character == '\\'
            || current_instruction_struct.character == '/') {
            switch (current_instruction_struct.character) {
                case '^':
                    sx = robot.location.x + robot.orientation.dx;
                    sy = robot.location.y + robot.orientation.dy;
                    break;
                case '\\':
                    sx = robot.location.x + robot.orientation.dy;
                    sy = robot.location.y - robot.orientation.dx;
                    break;
                case '/':
                    sx = robot.location.x - robot.orientation.dy;
                    sy = robot.location.y + robot.orientation.dx;
                    break;
            }
            strcpy(stations_visited[n_visited++], grid[sx][sy].name);

            /* remove visited station from stations_to_visit array */
            for (j = 0; j < 3 - n_visited; j++)
            {
                strcpy(stations_to_visit[j], stations_to_visit[j + 1]);
            }
            strcpy(stations_to_visit[3 - n_visited], "");
        }

        /* apply progress delta and robot location delta */
        p += current_instruction_struct.dp;
        length_to_go -= current_instruction_struct.dp;
        robot = current_instruction_struct.robot_position_after;
        i++;
        if (p >= total_length) break;

        current_instruction_struct = derive_next_instruction(grid, robot, route,
                                                             total_length, p);

        send_instruction(com_portnr, current_instruction_struct.character);

        break;

    case 'M':
        printf("\nMine encountered at %s; rerouting...\n\n", route[p + 1]);
```

```
                /* robot turns around and drives back to last junction */
                robot_rotate(&robot, 2);
                grid[current_instruction_struct.robot_position_after.location.x]
                    [current_instruction_struct.robot_position_after.location.y].v = -2;

                /* recalculate route from current location */
                if (n_visited < 2) {
                    route = route_multipoint(grid, location, stations_to_visit, 3 - n_visited,
                                                                                &total_length);
                } else {
                    route = route_optimized(grid, location, stations_to_visit[0], &total_length,
                                                                                  &n_turns, 1);
                }

                /* derive and show instruction sequence */
                instructions = route_derive_instructions(grid, robot, route, total_length);
                printf("%ld instructions: [ %s ]\n", strlen(instructions), instructions);

                /* reset distance left to endpoint */
                length_to_go = total_length;

                /* set new instruction */
                current_instruction_struct =
                        derive_next_instruction(grid, robot, route, total_length, p);

                break;

            default:
                printf("WARNING: received unexpected response from robot: '%s'\n", rx_buffer);
        }
    }

    free(route);
    free(instructions);
}

void challenge_3(Grid grid, char* start_station)
{
    printf("Challenge 3 scanning algorithm not implemented yet\n");
    exit(0);
}

char* route_derive_instructions(Grid grid, struct Robot simrobot, char** route, int length)
{
    int i = 0, j = 0;
    char* instructions;
    struct Instruction next_instruction;

    instructions = calloc (length + 1, sizeof (char));

    //instructions[0] = 'D';      /* activate robot */

    while (i < length)
    {
        next_instruction = derive_next_instruction(grid, simrobot, route, length, i);
        instructions[j++] = next_instruction.character;
```

```
        i += next_instruction.dp;
        simrobot = next_instruction.robot_position_after;
    }

    //instructions[j++] = 'P';    /* park/stop */
    instructions[j] = 0;

    return instructions;
}

struct Instruction derive_next_instruction(Grid grid, struct Robot simrobot, char** route,
                                           int length, int progress)
{
    int x1, y1, x2, y2, dx, dy, x3, y3, lookahead_dx, lookahead_dy;

    struct Instruction instruction;

    //instructions[0] = 'D';    /* activate robot */

    /* get coordinates of current and next cell on route */
    if (find_cell(route[progress], grid, &x1, &y1)) {
        printf("route_derive(): cell 1 '%s' not found on grid\n", route[progress]);
        exit(1);
    }
    if (find_cell(route[progress + 1], grid, &x2, &y2)) {
        printf("route_derive(): cell 2 '%s' not found on grid\n", route[progress + 1]);
        exit(1);
    }
    dx = x2 - x1;
    dy = y2 - y1;

    /* if exists, get coordinates of second next cell on route */
    if (progress + 1 < length && find_cell(route[progress + 2], grid, &x3, &y3)) {
        printf("route_derive(): cell 3 '%s' not found on grid\n", route[progress + 2]);
        exit(1);
    } else {
        lookahead_dx = x3 - x2;
        lookahead_dy = y3 - y2;
    }

    /* intermediate station visit */
    if (progress + 1 < length && strcmp(route[progress], route[progress + 2]) == 0)
    {
        /* station straight ahead */
        if ((dy * simrobot.orientation.dy == 1) ^ (dx * simrobot.orientation.dx == 1))
        {
            instruction.dp = 2;

            /* drive forward a few cm and reverse until back on junction */
            instruction.character = '^';
        }

        /* station on the left */
        else if ((dx * simrobot.orientation.dy == 1) ^ (dy * simrobot.orientation.dx == -1))
        {
            instruction.dp = 2;
```

```
            instruction.character = '\\';   /* turn onto station and turn back */
        }

        /* station on the right */
        else if ((dx * simrobot.orientation.dy == -1) ^ (dy * simrobot.orientation.dx == 1))
        {
            instruction.dp = 2;

            instruction.character = '/';    /* turn onto station and turn back */
        }
        else {
            /* debug output */
            printf("route_derive_instructions(): station visit b0rk\n");
            printf("n1: %s; n2: %s; n3: %s; (x1,y1): (%d,%d); (x2,y2): (%d,%d); (x3,y3): (%d,%d)\n",
                    route[progress], route[progress + 1], route[progress + 2],
                    x1, y1, x2, y2, x3, y3);
            printf("robot: location: (%d,%d); orientation (dx,dy): (%d,%d)\n",
                    simrobot.location.x, simrobot.location.y,
                    simrobot.orientation.dx, simrobot.orientation.dy);
        }
    }

    /* straight ahead */
    else if ((dy * simrobot.orientation.dy == 1) ^ (dx * simrobot.orientation.dx == 1))
    {
        /* left turn lookahead */
        if ((lookahead_dx * simrobot.orientation.dy == 1) ^ (lookahead_dy * simrobot.orientation.dx ==
            && strlen(route[progress]) > 2 && strlen(route[progress + 2]) > 2)  /* exclude stations */
        {
            simrobot.location.x += dx + lookahead_dx;
            simrobot.location.y += dy + lookahead_dy;

            robot_rotate(&simrobot, -1);    /* turn simrobot left 90 degrees */

            instruction.dp = 2;

            instruction.character = '{';    /* turn left, cut corner */
        }

        /* right turn lookahead */
        else if ((lookahead_dx * simrobot.orientation.dy == -1) ^ (lookahead_dy * simrobot.orientation.
            && strlen(route[progress]) > 2 && strlen(route[progress + 2]) > 2)  /* exclude stations */
        {
            simrobot.location.x += dx + lookahead_dx;
            simrobot.location.y += dy + lookahead_dy;

            robot_rotate(&simrobot, 1);     /* turn simrobot right 90 degrees */

            instruction.dp = 2;

            instruction.character = '}';    /* turn right, cut corner */
        }

        else {
            simrobot.location.x += dx;
```

```
            simrobot.location.y += dy;

            instruction.dp = 1;

            instruction.character = '|';    /* just go straight */
        }
    }

    /* left turn */
    else if ((dx * simrobot.orientation.dy == 1) ^ (dy * simrobot.orientation.dx == -1))
    {
        simrobot.location.x += dx;
        simrobot.location.y += dy;

        robot_rotate(&simrobot, -1);    /* turn simrobot left 90 degrees */

        instruction.dp = 1;

        instruction.character = '<';    /* hard left turn */
    }

    /* right turn */
    else if ((dx * simrobot.orientation.dy == -1) ^ (dy * simrobot.orientation.dx == 1))
    {
        simrobot.location.x += dx;
        simrobot.location.y += dy;

        robot_rotate(&simrobot, 1);     /* turn simrobot right 90 degrees */

        instruction.dp = 1;

        instruction.character = '>';    /* hard right turn */
    }

    /* need to go back: turn around */
    else if ((dx * simrobot.orientation.dx == -1) ^ (dy * simrobot.orientation.dy == -1))
    {
        simrobot.location.x += dx;
        simrobot.location.y += dy;

        robot_rotate(&simrobot, 2);     /* turn simrobot right 180 degrees */

        instruction.dp = 1;

        instruction.character = ')';    /* rotate 180 degrees */
    }
    else {
        /* debug output */
        printf("route_derive_instructions(): instruction compilation b0rk\n");
        printf("n1: %s; n2: %s; n3: %s; (x1,y1): (%d,%d); (x2,y2): (%d,%d); (x3,y3): (%d,%d)\n",
               route[progress], route[progress + 1], route[progress + 2], x1, y1, x2, y2, x3, y3)
        printf("robot: location: (%d,%d); orientation (dx,dy): (%d,%d)\n",
               simrobot.location.x, simrobot.location.y, simrobot.orientation.dx, simrobot.orient
    }

    instruction.robot_position_after = simrobot;
```

```
    return instruction;
}

void send_instruction(int cport_nr, char instruction)
{
    int i;
    char instr_buf[2] = "";
    instr_buf[0] = instruction;

    printf("sending new instruction: ");
    /* send next instruction multiple times to ensure transmission */
    for (i = 0; i < 3; i++) {
        usleep(1000);
        uart_tx(cport_nr, instr_buf, 0);
        printf("%s ", instr_buf);
    }
    printf("\b\n");
}
```

B.2. router.h

```
typedef struct Cell {
    int v;
    char name[8];
} Grid[11][11];

void manual_route(char* start, char* target, Grid grid);
int distance(char* start, char* target, Grid grid);
void generate_stdgrid(Grid stdgrid);
void wave(Grid grid, char* start, char* target);
int traceback_length(Grid grid, char* start, char* target, int std_output);
int traceback_single(Grid grid, char* start, char* target, char **route, int std_output);
void isolate_shortest(Grid grid, char* start);
void traceback_set_zero(Grid grid, int x, int y, int allow_equal);
void turn_wave(Grid grid, char* start);
void trace_count_turns(Grid grid, int x, int y, int dx, int dy, int i);
char** route_optimized(Grid grid, char* start, char* target, int* length,
                       int* n_turns, int std_output);
void postman_solve(Grid grid, char* entrypoint, char (*node_names)[6], int n_nodes,
                   char (*route)[6]);
char** route_multipoint(Grid grid, char start_station[6],
                        char (*stops)[6], int n_stops, int *total_length);

int find_cell(char* name, Grid grid, int* cell_x, int* cell_y);
void copy_grid(Grid source_grid, Grid target_grid);
void print_grid(Grid grid);
int in_grid(int x, int y);
```

B.3. router.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "router.h"
#include "robot.h"
```

```c
#include "utils.h"

#define min(a, b) (((a) < (b)) ? (a) : (b)) /* macro to get smallest of two values */

/* algorithmic functions */

/* calculates and returns shortest route on grid
 *
 * char* start      string with name of starting point
 * char* target     string with name of destination point
 * Grid grid        grid to use for routing
 *
 * @returns (int)               shortest path length
 */
int distance(char* start, char* target, Grid grid)
{
    Grid gridbuf;
    copy_grid(grid, gridbuf);

    wave(gridbuf, start, target);          /* apply wave algorithm to grid */

    return traceback_length(gridbuf, start, target, 0);  /* trace back a shortest route*/
}

/* generate standard grid
 *
 * Grid stdgrid     cell matrix to write grid to
 */
void generate_stdgrid(Grid stdgrid)
{
    int x, y;
    int x_a, y_a, n, s;
    int c, r;

    for (y = 0; y < 11; y++)
    {
        for (x = 0; x < 11; x++)
        {
            /* the grid is symmetrical: x_a and y_a are symmetrical coordinates
             * and mirrored in all 4 quarters of the grid (e.g. [0 1 2 3 4 5 4 3 2 1 0] in both d
            x_a = min(x, 10 - x);
            y_a = min(y, 10 - y);
            n = x > y;

            /* set nodes and edges to 0, other cells to -3 */
            if (((x_a % 2 == 1 && x_a > 0) || (y_a % 2 == 1 && y_a > 0)) && (x_a + y_a >= 2)) {
                stdgrid[x][y].v = 0;
            } else {
                stdgrid[x][y].v = -3;
            }

            /* calculate station number from coordinates */
            if (y_a == 0 && x_a % 2 == 1 && x_a >= 2) {
                s = abs(10*n - x/2);
            }
            else if (x_a == 0 && y_a % 2 == 1 && y_a >= 2) {
```

```
                s = abs(y/2 + 9 - 16*n);
            } else continue;

            /* convert station number to string and copy to cell name */
            num_to_char(s, stdgrid[x][y].name);
            //printf("(%d, %d): station \"%s\" (%d)\n", x, y, stdgrid[x][y].name, s);
        }
    }

    /* assign names to all nodes and edges */
    for (r = 0; r < 5; r++)
    {
        for (c = 0; c < 5; c++)
        {
            x = c*2 + 1;    /* node row number -> row number on grid */
            y = r*2 + 1;    /* node column number -> column number on grid */
            stdgrid[x][y].name[0] = 'c';
            stdgrid[x][y].name[1] = '0' + r;
            stdgrid[x][y].name[2] = '0' + c;
            stdgrid[x][y].name[3] = '\0';

            if (c < 4) {    /* assign name to edge right of current node _if_ not rightmost node column
                stdgrid[x + 1][y].name[0] = 'e';
                stdgrid[x + 1][y].name[1] = '0' + r;
                stdgrid[x + 1][y].name[2] = '0' + c;
                stdgrid[x + 1][y].name[3] = '0' + r;
                stdgrid[x + 1][y].name[4] = '0' + c + 1;
                stdgrid[x + 1][y].name[5] = '\0';
            }
            if (r < 4) {    /* assign name to edge below current node _if_ not bottom node row */
                stdgrid[x][y + 1].name[0] = 'e';
                stdgrid[x][y + 1].name[1] = '0' + r;
                stdgrid[x][y + 1].name[2] = '0' + c;
                stdgrid[x][y + 1].name[3] = '0' + r + 1;
                stdgrid[x][y + 1].name[4] = '0' + c;
                stdgrid[x][y + 1].name[5] = '\0';
            }
        }
    }
    //print_grid(stdgrid);
    //exit(0);
}

/* applies wave algorithm to given grid
 *
 * Grid grid        grid to apply wave to
 * char* start      string with name of starting point
 * char* target     string with name of destination point
 */
void wave(Grid grid, char* start, char* target)
{
    int x, y, dx, dy;
    int start_x, start_y, target_x, target_y;

    /* find coordinates of start and end stations */
    if (find_cell(start, grid, &start_x, &start_y)) {
```

```
        printf("wave(): start cell '%s' not found on grid\n", start);
        exit(1);
    }
    if (find_cell(target, grid, &target_x, &target_y)) {
        printf("wave(): target cell '%s' not found on grid\n", target);
        exit(1);
    }

    grid[target_x][target_y].v = 1; /* set end station value to 1 to start "wave" */

    /* iterate through grid until start station cell value is changed */
    while (grid[start_x][start_y].v == 0)
    {
        for (y = 0; y < 11; y++)
        {
            for (x = 0; x < 11; x++)
            {
                /* if cell value > 0, set adjacent cells (with value 0) to current cell value + 1
                if (grid[x][y].v > 0) {
                    for (dy = -1; dy <= 1; dy++) {
                        for (dx = -1; dx <= 1; dx++) {
                            if (abs(dx) != abs(dy)
                                && in_grid(x + dx, y + dy))
                                if (grid[x + dx][y + dy].v == 0)
                                    grid[x + dx][y + dy].v = grid[x][y].v + 1;
                        }
                    }
                }
            }
        }
    }
}

/* traces back length of shortest route on grid to which wave has been applied
 *
 * Grid grid        grid with wave
 * char* start      string with name of starting point
 * char* target     string with name of destination point
 *
 * int std_output   boolean int (0|1) determines if output is written to stdout
 *
 * @returns (int)   path length
 */
int traceback_length(Grid grid, char* start, char* target, int std_output)
{
    int i = 0, x, y, dx, dy, start_x, start_y, target_x, target_y, path_length = 0;

    /* find coordinates of start and end stations */
    if (find_cell(start, grid, &start_x, &start_y)) {
        printf("traceback_length(): start cell '%s' not found on grid\n", start);
        exit(1);
    }
    if (find_cell(target, grid, &target_x, &target_y)) {
        printf("traceback_length(): target cell '%s' not found on grid\n", target);
        exit(1);
    }
```

```
    x = start_x;
    y = start_y;

    if (std_output) printf("%s ", grid[x][y].name);
    while (!(x == target_x && y == target_y))   /* while target cell not reached */
    {
        /* find adjacent cell which has [current cell value - 1]; go to that cell */
        for (dy = -1; dy <= 1; dy++) {
            for (dx = -1; dx <= 1; dx++) {
                if (abs(dx) != abs(dy)
                    && in_grid(x + dx, y + dy)
                    && grid[x + dx][y + dy].v == grid[x][y].v - 1) {
                    x = x + dx;
                    y = y + dy;

                    /* print names of nodes and stations on route */
                    if (strlen(grid[x][y].name) > 0 && strlen(grid[x][y].name) <= 3) {
                        if (std_output) printf("%s ", grid[x][y].name);
                    }

                    path_length++;

                    dx = dy = 2;    /* break out of loops */
                }
            }
        }

        if (++i > 121) {
            printf("traceback_length(): route cannot be traced back, check grid:\n\n\n");
            print_grid(grid);
            exit(1);
        }
    }
    return path_length;
}

/* traces back a shortest route on grid to which wave has been applied
 *
 * Grid grid        grid with wave
 * char* start      string with name of starting point
 * char* target     string with name of destination point
 *
 * char (*route)[6] string array to store route in
 * int std_output   boolean int (0|1) determines if output is written to stdout
 *
 * @returns (int)   path length
 */
int traceback_single(Grid grid, char* start, char* target, char **route, int std_output)
{
    int i = 0, n = 0, x, y, dx, dy, start_x, start_y, target_x, target_y, path_length = 0;

    /* find coordinates of start and end stations */
    if (find_cell(start, grid, &start_x, &start_y)) {
        printf("traceback_length(): start cell '%s' not found on grid\n", start);
        exit(1);
```

```
    }
    if (find_cell(target, grid, &target_x, &target_y)) {
        printf("traceback_length(): target cell '%s' not found on grid\n", target);
        exit(1);
    }

    x = start_x;
    y = start_y;

    if (std_output) printf("%s ", grid[x][y].name);
    strcpy(route[n++], grid[x][y].name);

    while (!(x == target_x && y == target_y))    /* while target cell not reached */
    {
        /* find adjacent cell which has [current cell value - 1]; go to that cell */
        for (dy = -1; dy <= 1; dy++) {
            for (dx = -1; dx <= 1; dx++) {
                if (abs(dx) != abs(dy)
                    && in_grid(x + dx, y + dy)
                    && grid[x + dx][y + dy].v == grid[x][y].v - 1) {
                    x = x + dx;
                    y = y + dy;

                    /* print names of nodes and stations on route */
                    if (std_output) printf("%s ", grid[x][y].name);
                    strcpy(route[n++], grid[x][y].name);

                    path_length++;

                    dx = dy = 2;     /* break out of loops */
                }
            }
        }

        if (++i > 121) {
            printf("traceback_length(): route cannot be traced back, check grid:\n\n\n");
            print_grid(grid);
            exit(1);
        }
    }
    if (std_output) printf("\n");

    return path_length;
}

/* isolates all shortest paths on grid to which wave has been applied
 *
 * Grid grid        grid to operate on
 * char* start      name of start point
 */
void isolate_shortest(Grid grid, char* start)
{
    int x, y, start_x, start_y;

    /* find coordinates of start station */
    if (find_cell(start, grid, &start_x, &start_y)) {
```

```
        printf("isolate_shortest(): start cell '%s' not found on grid\n", start);
        exit(1);
    }

    for (y = 0; y < 11; y++)
    {
        for (x = 0; x < 11; x++)
        {
            if (grid[x][y].v == 0) grid[x][y].v = -1;
        }
    }

    traceback_set_zero(grid, start_x, start_y, 0);

    for (y = 0; y < 11; y++)
    {
        for (x = 0; x < 11; x++)
        {
            if (grid[x][y].v > 0) grid[x][y].v = -1;
        }
    }
}

/* clears all shortest paths recursively
 *
 * Grid grid    grid to operate on
 * int x,y      x and y coordinate for cell to inspect
 */
void traceback_set_zero(Grid grid, int x, int y, int allow_equal)
{
    int v, dx, dy;

    v = grid[x][y].v;
    grid[x][y].v = 0;

    for (dy = -1; dy <= 1; dy++) {
        for (dx = -1; dx <= 1; dx++) {
            if (abs(dx) != abs(dy)
                && in_grid(x + dx, y + dy)
                && (grid[x + dx][y + dy].v == v - 1 || (grid[x + dx][y + dy].v == v && allow_equal == 1
                && grid[x + dx][y + dy].v != 0)
            {
                traceback_set_zero(grid, x + dx, y + dy, allow_equal);
            }
        }
    }
}

/* modified wave algorithm which only counts turns
 *
 * Grid grid        grid to apply wave to
 * char* start      string with name of starting point
 */
void turn_wave(Grid grid, char* start)
{
    int x, y, dx, dy;
```

```
    int start_x, start_y;

    /* find coordinates of start and target points */
    if (find_cell(start, grid, &start_x, &start_y)) {
        printf("turn_wave(): start cell '%s' not found on grid\n", start);
        exit(1);
    }

    x = start_x;
    y = start_y;

    if (start_x == 0) {
        trace_count_turns(grid, x, y, 1, 0, 1);
    } else if (start_x == 10) {
        trace_count_turns(grid, x, y, -1, 0, 1);
    } else if (start_y == 0) {
        trace_count_turns(grid, x, y, 0, 1, 1);
    } else if (start_y == 10) {
        trace_count_turns(grid, x, y, 0, -1, 1);
    } else {
        for (dy = -1; dy <= 1; dy++) {
            for (dx = -1; dx <= 1; dx++) {
                if (abs(dx) != abs(dy)
                    && in_grid(x + dx, y + dy)
                    && grid[x + dx][y + dy].v == 0)
                {
                    trace_count_turns(grid, x, y, dx, dy, 1);
                }
            }
        }
    }
}

/* traces through all available paths, counting turns
 *
 * Grid grid    grid to trace in
 * int x,y      x and y of starting point
 * int dx,dy    direction to trace in from starting point
 * int i        value of current trace; used for recursion
 */
void trace_count_turns(Grid grid, int x, int y, int dx, int dy, int i)
{
    int d = 0;

    while (in_grid(x, y) && (grid[x][y].v == 0 || grid[x][y].v > i))
    {
        grid[x][y].v = i;
        x += dx;
        y += dy;
        d++;
    }

    x -= d * dx;
    y -= d * dy;

    while (d >= 0)
```

```
    {
        if (in_grid(x + dy, y + dx) && (grid[x + dy][y + dx].v == 0 || grid[x + dy][y + dx].v > i)) {
            trace_count_turns(grid, x + dy, y + dx, dy, dx, i + 1);
        }
        if (in_grid(x - dy, y - dx) && (grid[x - dy][y - dx].v == 0 || grid[x - dy][y - dx].v > i)) {
            trace_count_turns(grid, x - dy, y - dx, -dy, -dx, i + 1);
        }
        x += dx;
        y += dy;
        d--;
    }
}

char** route_optimized(Grid grid, char* start, char* target, int* length, int* n_turns, int std_output)
{
    int i = 0, x, y, target_x, target_y;
    char** route;
    Grid gridbuffer;
    copy_grid(grid, gridbuffer);

    /* find coordinates of end station */
    if (find_cell(target, gridbuffer, &target_x, &target_y)) {
        printf("traceback_optimized(): target cell '%s' not found on grid\n", target);
        exit(1);
    }

    if (std_output) printf("applying wave...\n");
    wave(gridbuffer, start, target);
    if (std_output) print_grid(gridbuffer);

    if (std_output) printf("isolating shortest paths...\n");
    isolate_shortest(gridbuffer, start);
    if(std_output) print_grid(gridbuffer);

    if (std_output) printf("applying turn tracer...\n");
    turn_wave(gridbuffer, start);
    if (std_output) print_grid(gridbuffer);
    *n_turns = gridbuffer[target_x][target_y].v - 1;
    if (std_output) printf("%d turns in best route(s) from %s to %s\n\n", *n_turns, start, target);

    if (std_output) printf("isolating paths with minimum turns...\n");
    traceback_set_zero(gridbuffer, target_x, target_y, 1);
    if(std_output) print_grid(gridbuffer);

    if (std_output) printf("disabling all other cells...\n");
    for (y = 0; y < 11; y++)
    {
        for (x = 0; x < 11; x++)
        {
            if (gridbuffer[x][y].v > 0) gridbuffer[x][y].v = -1;
        }
    }

    wave(gridbuffer, start, target);
    *length = traceback_length(gridbuffer, start, target, 0);
    if (std_output) printf("shortest route is %d hops long\n", *length);
```

```
    route = calloc(*length + 1, sizeof (char*));
    for (i = 0; i < *length + 1; i++)
    {
        route[i] = calloc(6, sizeof (char));
    }
    traceback_single(gridbuffer, start, target, route, std_output);
    if (std_output) print_grid(gridbuffer);

    return route;
}

/* solve Chinese Postman Problem: shortest route to visit all nodes
 *
 * Grid grid                grid to use for routing
 * char* entrypoint         string with name of starting point
 * char (*node_names)[6]    string array with names of nodes to visit
 * int n_nodes              number of nodes to visit
 *
 * char (*route)[6]         string array to write route to
 */
void postman_solve(Grid grid, char* entrypoint, char (*node_names)[6], int n_nodes, char (*route)
{
    int numbers[n_nodes];
    int perm[n_nodes];
    int results[fact(n_nodes)][3];
    int n_results = 0;
    int i, j;
    int route_length, shortest_route_number = -1, shortest_route_length = 121;

    for (i = 0; i < n_nodes; i++) {
        numbers[i] = i;
    }

    permutations(numbers, 3, 3, perm, 0, results, &n_results);

    for (i = 0; i < fact(3); i++) {
        route_length = distance(entrypoint, node_names[results[i][0]], grid);

        for (j = 0; j < (3 - 1); j++) {
            route_length += distance(node_names[results[i][j]], node_names[results[i][j + 1]], gr
        }

        if (route_length < shortest_route_length) {
            shortest_route_length = route_length;
            shortest_route_number = i;
            printf("postman_solve(): new shortest route found: (%s -> %s -> %s -> %s) with length
                   node_names[results[i][0]], node_names[results[i][1]], node_names[results[i][2]
        }
    }

    if (shortest_route_number < 0) {
        printf("postman_solve(): route could not be found on grid");
        exit(1);
    }
```

```
    strcpy(route[0], entrypoint);
    for (i = 0; i < n_nodes; i++) {
        strcpy(route[i + 1], node_names[results[shortest_route_number][i]]);
    }
}

char** route_multipoint(Grid grid, char start_station[6],
                        char (*stops)[6], int n_stops, int *total_length)
{
    int i, j, length = 0;
    int segment_length[n_stops], current_length = 0, segment_turns[n_stops];
    char shortest_route[n_stops + 1][6], **route_segment, **route;

    postman_solve(grid, start_station, stops, n_stops, shortest_route);

    /* copy shortest route station order to input station list */
    for (i = 1; i < n_stops + 1; i++)
    {
        strcpy(stops[i - 1], shortest_route[i]);
    }

    printf("\n");

    printf("Order of stations on route: ");
    printf("[start] %s -> ", shortest_route[0]);
    for (i = 1; i < n_stops + 1; i++) {
        printf("%s -> ", shortest_route[i]);
    }
    printf("\b\b\b[end]\n");

    *total_length = 0;
    for (i = 0; i < n_stops; i++)
    {
        *total_length += segment_length[i] = distance(shortest_route[i], shortest_route[i+1], grid);
    }

    route = calloc(*total_length, sizeof (char*));

    for (i = 0; i < n_stops; i++)
    {
        route_segment = route_optimized(grid, shortest_route[i], shortest_route[i + 1],
                                        &length, &segment_turns[i], 0);

        if (segment_length[i] != length)
            printf("length mismatch: %d vs %d\n", segment_length[i], length);

        for (j = 0; j < length + (i == n_stops - 1); j++)
        {
            route[current_length + j] = calloc(6, sizeof (char));
            strcpy(route[current_length + j], route_segment[j]);
        }
        current_length += length;

        free(route_segment);
    }
```

```
    return route;
}


/* proprietary functions */

/* finds cell with specified name on grid
 *
 * char* name               string with name of cell to find
 * Grid grid                grid to search in
 *
 * int *cell_x              pointer to integer to write x coordinate of cell to when found
 * int *cell_y              pointer to integer to write y coordinate of cell to when found
 *
 * @returns (boolean int)   0 if cell is found; 1 if cell is not found
 */
int find_cell(char* name, Grid grid, int *cell_x, int *cell_y)
{
    int x, y;
    for (y = 0; y < 11; y++)    /* loop through all cells until match is found or end of grid rea
    {
        for (x = 0; x < 11; x++)
        {
            if (strcmp(name, grid[x][y].name) == 0)
            {
                *cell_x = x;
                *cell_y = y;
                return 0;
            }
        }
    }
    return 1;
}

/* copies grid into a second cell matrix
 *
 * Grid source_grid     grid to copy
 * Grid target_grid     cell matrix to write to
 */
void copy_grid(Grid source_grid, Grid target_grid)
{
    int x, y;
    for (y = 0; y < 11; y++)
    {
        for (x = 0; x < 11; x++)
        {
            target_grid[x][y] = source_grid[x][y];
        }
    }
}

/* prints grid to stdout
 *
 * Grid grid    grid to print
 */
void print_grid(Grid grid)
```

```
{
    int x, y;

    printf("\n");
    printf("|\t\t\t\t\t\t\t\t\t\t\t\t|\n");

    for (y = 0; y < 11; y++)
    {
        printf("|\t");
        for (x = 0; x < 11; x++)
        {
            if (grid[x][y].v >= 0) {
                printf("%d", grid[x][y].v);
            } else if (grid[x][y].v == -1) {    /* disabled edges */
                printf("·");
            } else if (grid[x][y].v == -2) {    /* mines */
                printf("X");
            }
            /* -3 is nothingness and is not printed */

            if (x == robot.location.x && y == robot.location.y) {
                printf("\033[1;32m");
                if (robot.orientation.dx != 0)
                {
                    printf("%c", 0x3D + robot.orientation.dx);  /* < when -1, > when 1 */
                }
                else if (robot.orientation.dy != 0)
                {
                    printf("%c", 0x6A + 12 * robot.orientation.dy); /* ^ when -1, v when 1 */
                }
                else {
                    /* TODO: output error */
                }
                printf("\033[0m");
            }

            printf("\t");
        }
        printf("|\n");
        printf("|\t\t\t\t\t\t\t\t\t\t\t\t|\n");
    }
    printf("\n");
}

int in_grid(int x, int y)
{
    return (0 <= x && x <= 10 && 0 <= y && y <= 10);
}
```

B.4. robot.h

```
struct Robot {
    struct {
        int x,y;
    } location;
    struct {
```

```
        int dx,dy;
    } orientation;
} robot;

void robot_rotate(struct Robot*, int direction);
```

B.5. robot.c

```
#include "robot.h"

/* changes the orientation of a struct Robot
 *
 * struct Robot* robot  pointer to Robot to modify
 * int direction        indicates direction and amount to rotate:
 *                          positive is right, negative is left, in steps of 90 degrees
 */
void robot_rotate(struct Robot* robot, int direction)
{
    int tmp;
    for (int i = 0; direction < 0 ? i > direction : i < direction; direction < 0 ? i-- : i++)
    {
        tmp = robot->orientation.dx;
        robot->orientation.dx = direction < 0 ? robot->orientation.dy : -robot->orientation.dy;
        robot->orientation.dy = direction < 0 ? -tmp : tmp;
    }
}
```

B.6. utils.h

```
void permutations(const int* numbers, int n_numbers, int size, int* permutation, int current_size
long fact(int k);
void num_to_char(int num, char* str);
int string_in_array(char **array, char *string, int array_size);
```

B.7. utils.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "utils.h"


/* returns all possible orders for a list of numbers
 *
 * const int* numbers   pointer to array with (current) set of numbers to permutate (also used fo
 * int n_numbers        amount of numbers in current available set                  (used for rec
 * int size             amount of numbers in set (total)
 * int* permutation     pointer to integer array with current (partial) permutation (used for rec
 * int current_size     size of current permutation                                (used for rec
 *
 * int (*result)[size]  array with pointers to write resulting permutations to
 * int* n_result        pointer to integer used to keep track of number of results already in res
 */
void permutations(const int* numbers, int n_numbers, int size, int* permutation, int current_size
                  int (*result)[size], int* n_result)
{
```

```
    int i, j, n, available_numbers[n_numbers > 0 ? n_numbers - 1 : 0];

    if (current_size == size) {
        for (i = 0; i < current_size; i++) {
            result[*n_result][i] = permutation[i];
        }
        (*n_result)++;
    }

    for (i = 0; i < n_numbers; i++) {
        permutation[current_size] = numbers[i];

        /* build array with all numbers except those already in the permutation */
        n = 0;
        for (j = 0; j < n_numbers; j++) {
            if (j == i) continue;

            available_numbers[n] = numbers[j];
            n++;
        }

        permutations(available_numbers, n_numbers - 1, size, permutation, current_size + 1, result, n_r
    }
}

/* returns factorial for number k */
long fact(int k) {
    long result = 1;

    if (k > 12) {
        printf("fact(): %d! will not fit in 'long' datatype\n", k);
        exit(1);
    }

    while (k > 0) {
        result *= k--;
    }

    return result;
}

/* generates a string from given integer
 *
 * int num      number to convert to string
 * char* str    pointer to write generated string to
 */
void num_to_char(int num, char* str)
{
    int p = 0;

    if (num > 9) {          /* number >= 10 needs 2 digits */
        str[0] = '0' + 1;
        num -= 10;
        p++;
    }
    str[p++] = '0' + num;  /* number to corresponding character */
```

```
    str[p] = '\0';          /* terminate string with null character */
}

/* returns whether given string exists in string array
 *
 * char **array     array to search in
 * char *string     string to search for
 * int array_size   size of string array
 */
int string_in_array(char **array, char *string, int array_size)
{
    for (int i = 0; i < array_size; i++)
    {
        if (strcmp(string, array[i]) == 0)
            return 1;
    }
    return 0;
}
```

B.8. uart.h

```
void uart_tx(int cport_nr, char* str, int std_output);
int uart_rx_wait(int cport_nr, char* strbuf, int buf_size);
void uart_rx_clear(int cport_nr);
```

B.9. uart.c

```
#include "rs232.h"

void uart_tx(int cport_nr, char* str, int std_output)
{
    if (cport_nr != 42)
        RS232_cputs(cport_nr, str);

    if (std_output || cport_nr == 42)
        printf("sent: \"%s\"\n", str);
}

int uart_rx_wait(int cport_nr, char* strbuf, int buf_size)
{
    int length, i, j = 0;

    if (cport_nr != 42) {
        while (1) {
            length = RS232_PollComport(cport_nr, strbuf, buf_size - 1);

            if (length > 0) {
                strbuf[length] = 0;   /* == '\0' -> null character string terminator */

                for (i = 0; i < length; i++) {
                    if (strbuf[i] < 32)  /* replace unreadable control-codes by dots */
                    {
                        strbuf[i] = '.';
                    }
                }

                //printf("uart_rx_wait(): received transmission '%s' at cycle %d\n", strbuf, j);
```

```
                return length;
            }
            j++;
        }
    } else {
        printf("\nGive fake XBee response (max length %d): ", buf_size - 1);
        scanf("%s", strbuf);
        return strlen(strbuf);
    }
}

void uart_rx_clear(int cport_nr)
{
    char buffer[17];
    int length;

    do {
        length = uart_rx_wait(cport_nr, buffer, 17);
    } while (length > 8);
}
```

B.10. rs232.h

```
/*
***************************************************************************
*
* Author: Teunis van Beelen
*
* Copyright (C) 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017 Teunis van
*
* Email: teuniz@gmail.com
*
***************************************************************************
*
* This program is free software: you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation, either version 3 of the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program.  If not, see <http://www.gnu.org/licenses/>.
*
***************************************************************************
*/

/* Last revision: August 5, 2017 */

/* For more info and how to use this library, visit: http://www.teuniz.net/RS-232/ */


#ifndef rs232_INCLUDED
```

```
#define rs232_INCLUDED

#ifdef __cplusplus
extern "C" {
#endif

#include <stdio.h>
#include <string.h>



#if defined(__linux__) || defined(__FreeBSD__)

#include <termios.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <limits.h>
#include <sys/file.h>
#include <errno.h>

#else

#include <windows.h>

#endif

int RS232_OpenComport(int, int, const char *);
int RS232_PollComport(int, char *, int);
int RS232_SendByte(int, unsigned char);
int RS232_SendBuf(int, unsigned char *, int);
void RS232_CloseComport(int);
void RS232_cputs(int, const char *);
int RS232_IsDCDEnabled(int);
int RS232_IsCTSEnabled(int);
int RS232_IsDSREnabled(int);
void RS232_enableDTR(int);
void RS232_disableDTR(int);
void RS232_enableRTS(int);
void RS232_disableRTS(int);
void RS232_flushRX(int);
void RS232_flushTX(int);
void RS232_flushRXTX(int);
int RS232_GetPortnr(const char *);

#ifdef __cplusplus
} /* extern "C" */
#endif

#endif
```

B.11. rs232.c

```
/*
**************************************************************************
*
* Author: Teunis van Beelen
*
* Copyright (C) 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017 Teunis van
*
* Email: teuniz@gmail.com
*
**************************************************************************
*
* This program is free software: you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation, either version 3 of the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program.  If not, see <http://www.gnu.org/licenses/>.
*
**************************************************************************
*/


/* Last revision: November 22, 2017 */

/* For more info and how to use this library, visit: http://www.teuniz.net/RS-232/ */


#include "rs232.h"


#if defined(__linux__) || defined(__FreeBSD__)   /* Linux & FreeBSD */

#define RS232_PORTNR  38


int Cport[RS232_PORTNR],
    error;

struct termios new_port_settings,
       old_port_settings[RS232_PORTNR];

char *comports[RS232_PORTNR]={"/dev/ttyS0","/dev/ttyS1","/dev/ttyS2","/dev/ttyS3","/dev/ttyS4","/dev/tt
                        "/dev/ttyS6","/dev/ttyS7","/dev/ttyS8","/dev/ttyS9","/dev/ttyS10","/dev/ttyS11",
                        "/dev/ttyS12","/dev/ttyS13","/dev/ttyS14","/dev/ttyS15","/dev/ttyUSB0",
                        "/dev/ttyUSB1","/dev/ttyUSB2","/dev/ttyUSB3","/dev/ttyUSB4","/dev/ttyUSB5",
                        "/dev/ttyAMA0","/dev/ttyAMA1","/dev/ttyACM0","/dev/ttyACM1",
                        "/dev/rfcomm0","/dev/rfcomm1","/dev/ircomm0","/dev/ircomm1",
                        "/dev/cuau0","/dev/cuau1","/dev/cuau2","/dev/cuau3",
                        "/dev/cuaU0","/dev/cuaU1","/dev/cuaU2","/dev/cuaU3"};
```

```
int RS232_OpenComport(int comport_number, int baudrate, const char *mode)
{
  int baudr,
      status;

  if((comport_number>=RS232_PORTNR)||(comport_number<0))
  {
    printf("illegal comport number\n");
    return(1);
  }

  switch(baudrate)
  {
    case      50 : baudr = B50;
                   break;
    case      75 : baudr = B75;
                   break;
    case     110 : baudr = B110;
                   break;
    case     134 : baudr = B134;
                   break;
    case     150 : baudr = B150;
                   break;
    case     200 : baudr = B200;
                   break;
    case     300 : baudr = B300;
                   break;
    case     600 : baudr = B600;
                   break;
    case    1200 : baudr = B1200;
                   break;
    case    1800 : baudr = B1800;
                   break;
    case    2400 : baudr = B2400;
                   break;
    case    4800 : baudr = B4800;
                   break;
    case    9600 : baudr = B9600;
                   break;
    case   19200 : baudr = B19200;
                   break;
    case   38400 : baudr = B38400;
                   break;
    case   57600 : baudr = B57600;
                   break;
    case  115200 : baudr = B115200;
                   break;
    case  230400 : baudr = B230400;
                   break;
    case  460800 : baudr = B460800;
                   break;
    case  500000 : baudr = B500000;
                   break;
    case  576000 : baudr = B576000;
                   break;
```

```
  case  921600 : baudr = B921600;
                 break;
  case 1000000 : baudr = B1000000;
                 break;
  case 1152000 : baudr = B1152000;
                 break;
  case 1500000 : baudr = B1500000;
                 break;
  case 2000000 : baudr = B2000000;
                 break;
  case 2500000 : baudr = B2500000;
                 break;
  case 3000000 : baudr = B3000000;
                 break;
  case 3500000 : baudr = B3500000;
                 break;
  case 4000000 : baudr = B4000000;
                 break;
  default      : printf("invalid baudrate\n");
                 return(1);
                 break;
}

int cbits=CS8,
    cpar=0,
    ipar=IGNPAR,
    bstop=0;

if(strlen(mode) != 3)
{
  printf("invalid mode \"%s\"\n", mode);
  return(1);
}

switch(mode[0])
{
  case '8': cbits = CS8;
            break;
  case '7': cbits = CS7;
            break;
  case '6': cbits = CS6;
            break;
  case '5': cbits = CS5;
            break;
  default : printf("invalid number of data-bits '%c'\n", mode[0]);
            return(1);
            break;
}

switch(mode[1])
{
  case 'N':
  case 'n': cpar = 0;
            ipar = IGNPAR;
            break;
  case 'E':
```

```
    case 'e': cpar = PARENB;
              ipar = INPCK;
              break;
    case 'O':
    case 'o': cpar = (PARENB | PARODD);
              ipar = INPCK;
              break;
    default : printf("invalid parity '%c'\n", mode[1]);
              return(1);
              break;
  }

  switch(mode[2])
  {
    case '1': bstop = 0;
              break;
    case '2': bstop = CSTOPB;
              break;
    default : printf("invalid number of stop bits '%c'\n", mode[2]);
              return(1);
              break;
  }

/*
http://pubs.opengroup.org/onlinepubs/7908799/xsh/termios.h.html

http://man7.org/linux/man-pages/man3/termios.3.html
*/

  Cport[comport_number] = open(comports[comport_number], O_RDWR | O_NOCTTY | O_NDELAY);
  if(Cport[comport_number]==-1)
  {
    perror("unable to open comport ");
    return(1);
  }

  /* lock access so that another process can't also use the port */
  if(flock(Cport[comport_number], LOCK_EX | LOCK_NB) != 0)
  {
    close(Cport[comport_number]);
    perror("Another process has locked the comport.");
    return(1);
  }

  error = tcgetattr(Cport[comport_number], old_port_settings + comport_number);
  if(error==-1)
  {
    close(Cport[comport_number]);
    flock(Cport[comport_number], LOCK_UN);  /* free the port so that others can use it. */
    perror("unable to read portsettings ");
    return(1);
  }
  memset(&new_port_settings, 0, sizeof(new_port_settings));  /* clear the new struct */

  new_port_settings.c_cflag = cbits | cpar | bstop | CLOCAL | CREAD;
  new_port_settings.c_iflag = ipar;
```

```
  new_port_settings.c_oflag = 0;
  new_port_settings.c_lflag = 0;
  new_port_settings.c_cc[VMIN] = 0;      /* block untill n bytes are received */
  new_port_settings.c_cc[VTIME] = 0;     /* block untill a timer expires (n * 100 mSec.) */

  cfsetispeed(&new_port_settings, baudr);
  cfsetospeed(&new_port_settings, baudr);

  error = tcsetattr(Cport[comport_number], TCSANOW, &new_port_settings);
  if(error==-1)
  {
    tcsetattr(Cport[comport_number], TCSANOW, old_port_settings + comport_number);
    close(Cport[comport_number]);
    flock(Cport[comport_number], LOCK_UN);  /* free the port so that others can use it. */
    perror("unable to adjust portsettings ");
    return(1);
  }

/* http://man7.org/linux/man-pages/man4/tty_ioctl.4.html */

  if(ioctl(Cport[comport_number], TIOCMGET, &status) == -1)
  {
    tcsetattr(Cport[comport_number], TCSANOW, old_port_settings + comport_number);
    flock(Cport[comport_number], LOCK_UN);  /* free the port so that others can use it. */
    perror("unable to get portstatus");
    return(1);
  }

  status |= TIOCM_DTR;    /* turn on DTR */
  status |= TIOCM_RTS;    /* turn on RTS */

  if(ioctl(Cport[comport_number], TIOCMSET, &status) == -1)
  {
    tcsetattr(Cport[comport_number], TCSANOW, old_port_settings + comport_number);
    flock(Cport[comport_number], LOCK_UN);  /* free the port so that others can use it. */
    perror("unable to set portstatus");
    return(1);
  }

  return(0);
}


int RS232_PollComport(int comport_number, char *buf, int size)
{
  int n;

  n = read(Cport[comport_number], buf, size);

  if(n < 0)
  {
    if(errno == EAGAIN)  return 0;
  }

  return(n);
}
```

```
int RS232_SendByte(int comport_number, unsigned char byte)
{
  int n = write(Cport[comport_number], &byte, 1);
  if(n < 0)
  {
    if(errno == EAGAIN)
    {
      return 0;
    }
    else
    {
      return 1;
    }
  }

  return(0);
}


int RS232_SendBuf(int comport_number, unsigned char *buf, int size)
{
  int n = write(Cport[comport_number], buf, size);
  if(n < 0)
  {
    if(errno == EAGAIN)
    {
      return 0;
    }
    else
    {
      return -1;
    }
  }

  return(n);
}


void RS232_CloseComport(int comport_number)
{
  int status;

  if(ioctl(Cport[comport_number], TIOCMGET, &status) == -1)
  {
    perror("unable to get portstatus");
  }

  status &= ~TIOCM_DTR;    /* turn off DTR */
  status &= ~TIOCM_RTS;    /* turn off RTS */

  if(ioctl(Cport[comport_number], TIOCMSET, &status) == -1)
  {
    perror("unable to set portstatus");
  }
```

```
  tcsetattr(Cport[comport_number], TCSANOW, old_port_settings + comport_number);
  close(Cport[comport_number]);

  flock(Cport[comport_number], LOCK_UN);  /* free the port so that others can use it. */
}

/*
Constant  Description
TIOCM_LE        DSR (data set ready/line enable)
TIOCM_DTR       DTR (data terminal ready)
TIOCM_RTS       RTS (request to send)
TIOCM_ST        Secondary TXD (transmit)
TIOCM_SR        Secondary RXD (receive)
TIOCM_CTS       CTS (clear to send)
TIOCM_CAR       DCD (data carrier detect)
TIOCM_CD        see TIOCM_CAR
TIOCM_RNG       RNG (ring)
TIOCM_RI        see TIOCM_RNG
TIOCM_DSR       DSR (data set ready)

http://man7.org/linux/man-pages/man4/tty_ioctl.4.html
*/

int RS232_IsDCDEnabled(int comport_number)
{
  int status;

  ioctl(Cport[comport_number], TIOCMGET, &status);

  if(status&TIOCM_CAR) return(1);
  else return(0);
}


int RS232_IsCTSEnabled(int comport_number)
{
  int status;

  ioctl(Cport[comport_number], TIOCMGET, &status);

  if(status&TIOCM_CTS) return(1);
  else return(0);
}


int RS232_IsDSREnabled(int comport_number)
{
  int status;

  ioctl(Cport[comport_number], TIOCMGET, &status);

  if(status&TIOCM_DSR) return(1);
  else return(0);
}
```

```
void RS232_enableDTR(int comport_number)
{
  int status;

  if(ioctl(Cport[comport_number], TIOCMGET, &status) == -1)
  {
    perror("unable to get portstatus");
  }

  status |= TIOCM_DTR;    /* turn on DTR */

  if(ioctl(Cport[comport_number], TIOCMSET, &status) == -1)
  {
    perror("unable to set portstatus");
  }
}


void RS232_disableDTR(int comport_number)
{
  int status;

  if(ioctl(Cport[comport_number], TIOCMGET, &status) == -1)
  {
    perror("unable to get portstatus");
  }

  status &= ~TIOCM_DTR;    /* turn off DTR */

  if(ioctl(Cport[comport_number], TIOCMSET, &status) == -1)
  {
    perror("unable to set portstatus");
  }
}


void RS232_enableRTS(int comport_number)
{
  int status;

  if(ioctl(Cport[comport_number], TIOCMGET, &status) == -1)
  {
    perror("unable to get portstatus");
  }

  status |= TIOCM_RTS;    /* turn on RTS */

  if(ioctl(Cport[comport_number], TIOCMSET, &status) == -1)
  {
    perror("unable to set portstatus");
  }
}


void RS232_disableRTS(int comport_number)
```

```
{
  int status;

  if(ioctl(Cport[comport_number], TIOCMGET, &status) == -1)
  {
    perror("unable to get portstatus");
  }

  status &= ~TIOCM_RTS;    /* turn off RTS */

  if(ioctl(Cport[comport_number], TIOCMSET, &status) == -1)
  {
    perror("unable to set portstatus");
  }
}


void RS232_flushRX(int comport_number)
{
  tcflush(Cport[comport_number], TCIFLUSH);
}


void RS232_flushTX(int comport_number)
{
  tcflush(Cport[comport_number], TCOFLUSH);
}


void RS232_flushRXTX(int comport_number)
{
  tcflush(Cport[comport_number], TCIOFLUSH);
}


#else  /* windows */

#define RS232_PORTNR  16

HANDLE Cport[RS232_PORTNR];


char *comports[RS232_PORTNR]={"\\\\.\\COM1",  "\\\\.\\COM2",  "\\\\.\\COM3",  "\\\\.\\COM4",
                              "\\\\.\\COM5",  "\\\\.\\COM6",  "\\\\.\\COM7",  "\\\\.\\COM8",
                              "\\\\.\\COM9",  "\\\\.\\COM10", "\\\\.\\COM11", "\\\\.\\COM12",
                              "\\\\.\\COM13", "\\\\.\\COM14", "\\\\.\\COM15", "\\\\.\\COM16"};

char mode_str[128];


int RS232_OpenComport(int comport_number, int baudrate, const char *mode)
{
  if((comport_number>=RS232_PORTNR)||(comport_number<0))
  {
    printf("illegal comport number\n");
    return(1);
```

```
}

switch(baudrate)
{
  case     110 : strcpy(mode_str, "baud=110");
                 break;
  case     300 : strcpy(mode_str, "baud=300");
                 break;
  case     600 : strcpy(mode_str, "baud=600");
                 break;
  case    1200 : strcpy(mode_str, "baud=1200");
                 break;
  case    2400 : strcpy(mode_str, "baud=2400");
                 break;
  case    4800 : strcpy(mode_str, "baud=4800");
                 break;
  case    9600 : strcpy(mode_str, "baud=9600");
                 break;
  case   19200 : strcpy(mode_str, "baud=19200");
                 break;
  case   38400 : strcpy(mode_str, "baud=38400");
                 break;
  case   57600 : strcpy(mode_str, "baud=57600");
                 break;
  case  115200 : strcpy(mode_str, "baud=115200");
                 break;
  case  128000 : strcpy(mode_str, "baud=128000");
                 break;
  case  256000 : strcpy(mode_str, "baud=256000");
                 break;
  case  500000 : strcpy(mode_str, "baud=500000");
                 break;
  case 1000000 : strcpy(mode_str, "baud=1000000");
                 break;
  default      : printf("invalid baudrate\n");
                 return(1);
                 break;
}

if(strlen(mode) != 3)
{
  printf("invalid mode \"%s\"\n", mode);
  return(1);
}

switch(mode[0])
{
  case '8': strcat(mode_str, " data=8");
            break;
  case '7': strcat(mode_str, " data=7");
            break;
  case '6': strcat(mode_str, " data=6");
            break;
  case '5': strcat(mode_str, " data=5");
            break;
  default : printf("invalid number of data-bits '%c'\n", mode[0]);
```

```
            return(1);
            break;
  }

  switch(mode[1])
  {
    case 'N':
    case 'n': strcat(mode_str, " parity=n");
            break;
    case 'E':
    case 'e': strcat(mode_str, " parity=e");
            break;
    case 'O':
    case 'o': strcat(mode_str, " parity=o");
            break;
    default : printf("invalid parity '%c'\n", mode[1]);
            return(1);
            break;
  }

  switch(mode[2])
  {
    case '1': strcat(mode_str, " stop=1");
            break;
    case '2': strcat(mode_str, " stop=2");
            break;
    default : printf("invalid number of stop bits '%c'\n", mode[2]);
            return(1);
            break;
  }

  strcat(mode_str, " dtr=on rts=on");

/*
http://msdn.microsoft.com/en-us/library/windows/desktop/aa363145%28v=vs.85%29.aspx

http://technet.microsoft.com/en-us/library/cc732236.aspx
*/

  Cport[comport_number] = CreateFileA(comports[comport_number],
                      GENERIC_READ|GENERIC_WRITE,
                      0,                          /* no share  */
                      NULL,                       /* no security */
                      OPEN_EXISTING,
                      0,                          /* no threads */
                      NULL);                      /* no templates */

  if(Cport[comport_number]==INVALID_HANDLE_VALUE)
  {
    printf("unable to open comport\n");
    return(1);
  }

  DCB port_settings;
  memset(&port_settings, 0, sizeof(port_settings));  /* clear the new struct  */
  port_settings.DCBlength = sizeof(port_settings);
```

```
  if(!BuildCommDCBA(mode_str, &port_settings))
  {
    printf("unable to set comport dcb settings\n");
    CloseHandle(Cport[comport_number]);
    return(1);
  }

  if(!SetCommState(Cport[comport_number], &port_settings))
  {
    printf("unable to set comport cfg settings\n");
    CloseHandle(Cport[comport_number]);
    return(1);
  }

  COMMTIMEOUTS Cptimeouts;

  Cptimeouts.ReadIntervalTimeout         = MAXDWORD;
  Cptimeouts.ReadTotalTimeoutMultiplier  = 0;
  Cptimeouts.ReadTotalTimeoutConstant    = 0;
  Cptimeouts.WriteTotalTimeoutMultiplier = 0;
  Cptimeouts.WriteTotalTimeoutConstant   = 0;

  if(!SetCommTimeouts(Cport[comport_number], &Cptimeouts))
  {
    printf("unable to set comport time-out settings\n");
    CloseHandle(Cport[comport_number]);
    return(1);
  }

  return(0);
}


int RS232_PollComport(int comport_number, unsigned char *buf, int size)
{
  int n;

/* added the void pointer cast, otherwise gcc will complain about */
/* "warning: dereferencing type-punned pointer will break strict aliasing rules" */

  ReadFile(Cport[comport_number], buf, size, (LPDWORD)((void *)&n), NULL);

  return(n);
}


int RS232_SendByte(int comport_number, unsigned char byte)
{
  int n;

  WriteFile(Cport[comport_number], &byte, 1, (LPDWORD)((void *)&n), NULL);

  if(n<0)  return(1);

  return(0);
```

```
}


int RS232_SendBuf(int comport_number, unsigned char *buf, int size)
{
  int n;

  if(WriteFile(Cport[comport_number], buf, size, (LPDWORD)((void *)&n), NULL))
  {
    return(n);
  }

  return(-1);
}


void RS232_CloseComport(int comport_number)
{
  CloseHandle(Cport[comport_number]);
}

/*
http://msdn.microsoft.com/en-us/library/windows/desktop/aa363258%28v=vs.85%29.aspx
*/

int RS232_IsDCDEnabled(int comport_number)
{
  int status;

  GetCommModemStatus(Cport[comport_number], (LPDWORD)((void *)&status));

  if(status&MS_RLSD_ON) return(1);
  else return(0);
}


int RS232_IsCTSEnabled(int comport_number)
{
  int status;

  GetCommModemStatus(Cport[comport_number], (LPDWORD)((void *)&status));

  if(status&MS_CTS_ON) return(1);
  else return(0);
}


int RS232_IsDSREnabled(int comport_number)
{
  int status;

  GetCommModemStatus(Cport[comport_number], (LPDWORD)((void *)&status));

  if(status&MS_DSR_ON) return(1);
  else return(0);
}
```

```
void RS232_enableDTR(int comport_number)
{
  EscapeCommFunction(Cport[comport_number], SETDTR);
}


void RS232_disableDTR(int comport_number)
{
  EscapeCommFunction(Cport[comport_number], CLRDTR);
}


void RS232_enableRTS(int comport_number)
{
  EscapeCommFunction(Cport[comport_number], SETRTS);
}


void RS232_disableRTS(int comport_number)
{
  EscapeCommFunction(Cport[comport_number], CLRRTS);
}

/*
https://msdn.microsoft.com/en-us/library/windows/desktop/aa363428%28v=vs.85%29.aspx
*/

void RS232_flushRX(int comport_number)
{
  PurgeComm(Cport[comport_number], PURGE_RXCLEAR | PURGE_RXABORT);
}


void RS232_flushTX(int comport_number)
{
  PurgeComm(Cport[comport_number], PURGE_TXCLEAR | PURGE_TXABORT);
}


void RS232_flushRXTX(int comport_number)
{
  PurgeComm(Cport[comport_number], PURGE_RXCLEAR | PURGE_RXABORT);
  PurgeComm(Cport[comport_number], PURGE_TXCLEAR | PURGE_TXABORT);
}


#endif


void RS232_cputs(int comport_number, const char *text)  /* sends a string to serial port */
{
  while(*text != 0)   RS232_SendByte(comport_number, *(text++));
}
```

```
/* return index in comports matching to device name or -1 if not found */
int RS232_GetPortnr(const char *devname)
{
  int i;

  char str[32];

#if defined(__linux__) || defined(__FreeBSD__)   /* Linux & FreeBSD */
  strcpy(str, "/dev/");
#else  /* windows */
  strcpy(str, "\\\\.\\");
#endif
  strncat(str, devname, 16);
  str[31] = 0;

  for(i=0; i<RS232_PORTNR; i++)
  {
    if(!strcmp(comports[i], str))
    {
      return i;
    }
  }

  return -1;  /* device not found */
}
```