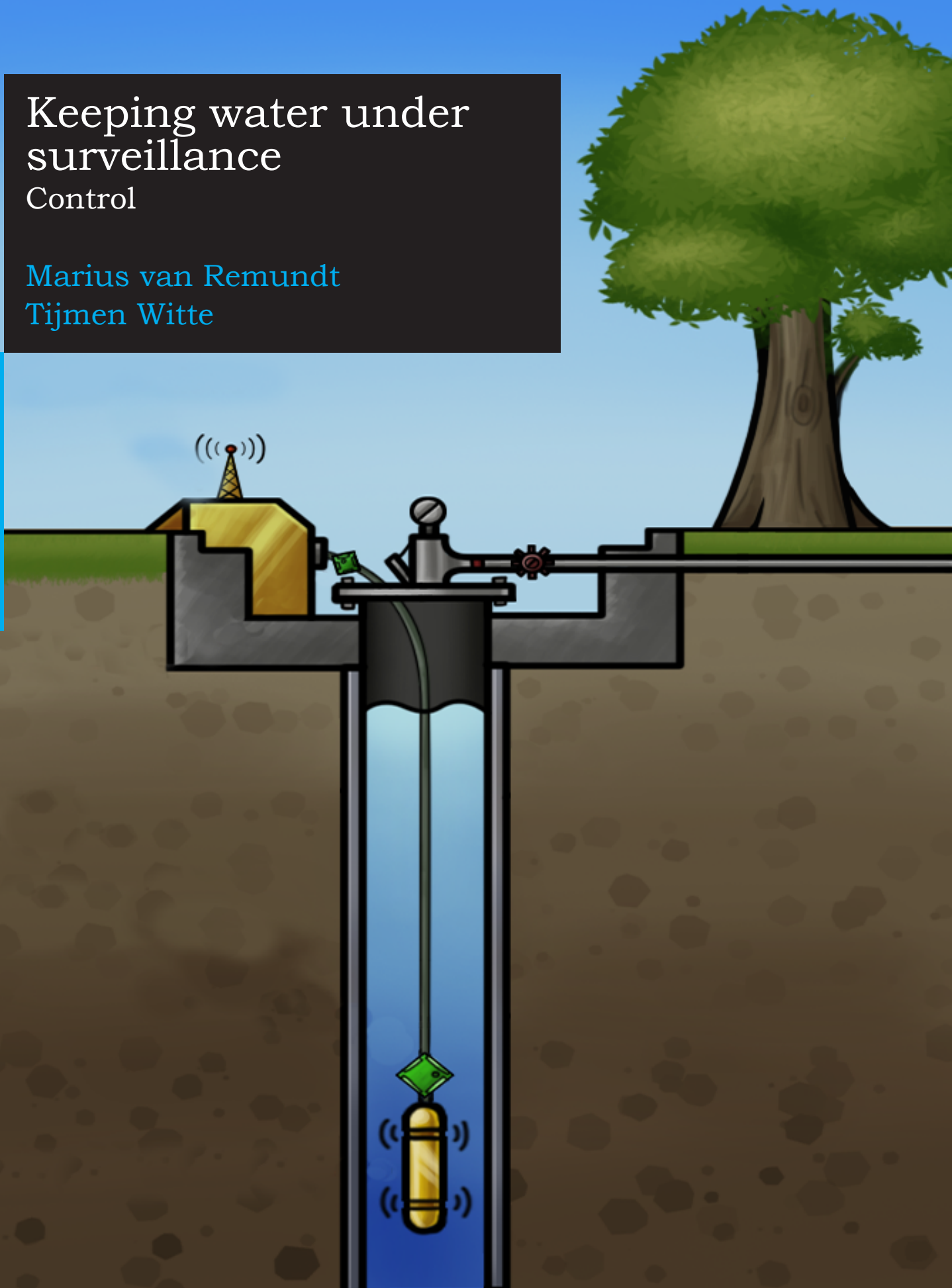


# Keeping water under surveillance

## Control

Marius van Remundt  
Tijmen Witte

Technische Universiteit Delft



# Keeping water under surveillance

Control

by

**Marius van Remundt  
Tijmen Witte**

In partial fulfilment of the requirements for the degree of

**Bachelor of Science**

in Electrical Engineering

at Delft University of Technology,

to be defended on June 27, 2016

Marius van Remundt: 4260767

Tijmen Witte: 4259041

Project duration: April 18, 2016 – Juli 1, 2016

External proposer: Msc. E. van der Putte

Supervisor: Dr. Ir. A. Bossche

Daily supervisor: Ing. J. Bastemeijer

Thesis committee: Dr. N. Llombart Juan  
Dr. Ir. A. van Genderen  
Msc. E. van der Putte

An electronic version of this thesis is available at <http://repository.tudelft.nl/>

# Preface

This thesis is written as part of the Electrical Engineering Bachelor programme, provided by Delft University of Technology located in the Netherlands. To successfully obtain the degree of Bachelor of Science and by that completing the Bachelor programme, a group project must be executed and judged to be satisfactory. One of the requirements of the project is to divide the group, consisting of six people, into subgroups where each subgroup writes their own thesis about their part of the total concept. This thesis is about the control subsystem of the total concept for *Keeping water under surveillance*.

This thesis and research could not have been completed without the help provided by several people. As a token of gratitude, we especially would like to thank the following persons:

- Dr. Ir. A. Bossche
- Ing. J. Bastemeijer
- Msc. E. van der Putte

*Marius van Remundt  
Tijmen Witte  
Delft, June 2016*

# Abstract

This thesis describes the work of two students who designed the control system of a stand-alone water well measuring system, named SubDowser. The project was done by order of a proposal from the start-up company SkyDowser. The total thesis group, consisting of three subgroups, was asked to build a working prototype which could measure the water level, conductivity and temperature within a water well. Subsequently, this data needs to get uploaded to a server located in Delft.

When designing the control system clear requirements and interfaces were set. From these demands numerous considerations were taking into account resulting into using two LPC1768 microcontrollers to control the whole system. One stationed underground acting as slave and one stationed above ground acting as master. These two microcontrollers communicate with each other by two differential drivers using the RS-485 standard.

The development and implementation of the written programs were done on the mbed IDE, which was provided by the used developers board. From the result can be stated that a decent working prototype has been developed in fulfilment of the requirements. However, there is still room for improvement.

# Contents

<b>Preface</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Total concept</b>	<b>2</b>
2.1 Function analysis . . . . .	2
2.2 Requirements . . . . .	3
2.3 Interfaces . . . . .	3
2.3.1 Interfacing with sensor subsystem . . . . .	3
2.3.2 Interfacing with telecommunication subsystem . . . . .	3
2.3.3 Interfacing with the power subsystem . . . . .	4
2.4 Design of the total concept . . . . .	4
<b>3 Requirements</b>	<b>6</b>
3.1 Function analysis . . . . .	6
3.2 Functional requirements . . . . .	7
3.3 Non-functional requirements . . . . .	7
3.3.1 Product requirements . . . . .	7
<b>4 Conceptual Design</b>	<b>9</b>
4.1 Morphological table . . . . .	10
4.1.1 Data collection . . . . .	10
4.1.2 Data processing . . . . .	11
4.1.3 Data saving . . . . .	11
4.1.4 Subsystem communication . . . . .	12
4.2 Design choices . . . . .	13

---

4.3	Final concept . . . . .	14
<b>5</b>	<b>Implementation</b>	<b>15</b>
5.1	Sub tasks and components. . . . .	15
5.1.1	ADCs . . . . .	15
5.1.2	Digital . . . . .	16
5.1.3	RTC . . . . .	16
5.2	Data collection . . . . .	16
5.3	Data saving . . . . .	17
5.3.1	Settings . . . . .	17
5.3.2	Measurement . . . . .	18
5.4	Data sending . . . . .	19
5.5	Modular . . . . .	20
5.6	First run . . . . .	20
5.7	Master microcontroller . . . . .	20
5.8	Slave microcontroller . . . . .	22
5.9	Serial communication with RS-485 . . . . .	23
5.10	Lookup table . . . . .	23
<b>6</b>	<b>Testing</b>	<b>25</b>
6.1	Test plan . . . . .	25
6.2	Results . . . . .	26
<b>7</b>	<b>Conclusion</b>	<b>28</b>
<b>8</b>	<b>Discussion and recommendations</b>	<b>29</b>
8.1	Discussion . . . . .	29
8.2	Recommendations . . . . .	30
8.2.1	Data Packages reduction . . . . .	30
8.2.2	C++ code can be improved . . . . .	30
8.2.3	More efficient microcontroller . . . . .	30
8.2.4	Data needs to get flushed . . . . .	30

8.2.5	Adding digital sensors . . . . .	30
<b>A</b>	<b>Appendix</b>	<b>31</b>
A.1	Programme of requirements . . . . .	31
A.1.1	Functional requirements . . . . .	31
A.1.2	Non-functional requirements . . . . .	31
A.2	Layout LPC1768 . . . . .	34
A.3	Results . . . . .	35
A.3.1	Additional measurement results . . . . .	36
A.3.2	LocationCheck() result . . . . .	37
A.4	Code . . . . .	37
A.4.1	Master microcontroller . . . . .	37
A.4.2	Slave microcontroller . . . . .	44
A.5	Cost analysis . . . . .	47
	<b>Bibliography</b>	<b>48</b>

# 1

## Introduction

Clean water is one of the most essential things in life. Unfortunately, it is not everywhere easily accessible. The start-up company SkyDowser tries to create awareness of the state of the groundwater level for people. Currently, SkyDowser uses a drone to map the contents of the subsoil in water stressed regions. While the drones gives excellent results the cost of creating real time water surveillance at specific places is too high compared to the utility it provides. Therefore, SkyDowser searched for students to design a diver to make real time water surveillance possible at low cost. These divers could then be used to monitor the water levels and quality of water in water wells.

SkyDowser wants to start deploying their diver throughout Africa because real-time water surveillance is not yet available in that region of the world. This is due to the fact that other products with the same purpose are too expensive. If farmers could get insight in their water resources they could set up their irrigation in a more sustainable way and could possibly save great parts of their yield. The first prototype will be built for Kenya because groundwater control is a major issue there [1]. This specifies a certain area to develop the prototype for.

The diver, further named SubDowser, which must be developed should be able to measure the ground water level, the conductivity and temperature. These quantities will be used to judge if the water from the well is suitable for irrigation. Furthermore, it needs to be possible for customers to add new sensors in case better sensors are developed or, for example, if the customer wants other information about their water. In order for the customer to use the data of the measurements, the data will be made available to them through the SkyDowser servers, so SubDowser must be able to send the data to this server. Besides that, the system needs to work autonomously and be modular.

The main topic of this thesis will be the control subsystem of SubDowser. The thesis will thereby consist of the following sub-parts. Firstly, the total concept and the requirements of SubDowser will be explained. Then the requirements for the control subsystem and the design choices are handled. After this the implementation and test results will be discussed. At last a conclusion and discussion will be stated.



# 2

## Total concept

### 2.1. Function analysis

In this chapter the total concept of SubDowser will be discussed. This total concept is designed by the the whole group and in collaboration with the external proposer, so the external demands are met. A function analysis is done to get a better understanding of what the functions of SubDowser are. The function analysis is shown in figure 2.1.

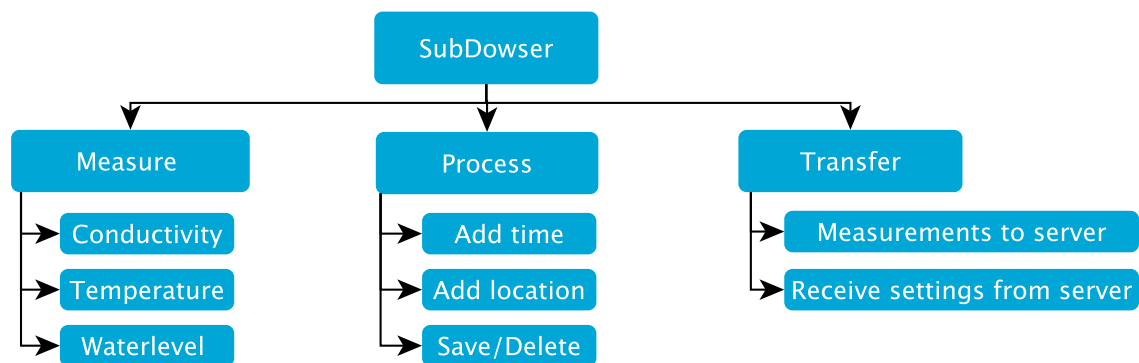


Figure 2.1: Function analysis of the complete system

The main function of SubDowser is divided into three different functions. First of all, the SubDowser needs to measure the ground water level compared to the ground surface, the conductivity and the temperature of the water. Secondly, the measured data needs to be processed, other data (time and location) needs to be added and saved, and thirdly the measurements data needs to be sent to the SkyDowser server according to the imposed setting. To divide the whole assignment into three parts one group of students will focus on designing a way to do the measurements, the other group will do the control of the system and the last group will focus on sending the measurement data to the servers of SkyDowser and designing the powercontrol of the system.

## 2.2. Requirements

For the total system to function properly and to meet the external proposer demands, the thesis group set-up a programme of requirements. The most important requirements of the total concept are shown below, the full list of requirements can be seen in appendix [A.1](#).

1. The product needs to measure the groundwater level, conductivity and temperature of the water
2. The product needs to send the measurement data to the server of SkyDowser
3. The product needs to be able to adjust settings, such as measuring and sending frequency, according to changes given by the SkyDowser servers
4. The product needs to be Plug & Play for the customer
5. The product needs to be modular
6. The cost price of the product needs to be lower than 50 Euros
7. The product needs to function autonomously for at least two successive years
8. The product needs to fit in a water well with at least a diagonal of 6 centimetres
9. The product needs to be able to work in a water well with a maximum depth of 60 metres

## 2.3. Interfaces

Each of the three groups must keep the requirements in mind when designing their subsystems. To separate the total concept into three parts clear interfaces need to be agreed upon to mark the boundaries of each subsystem.

### 2.3.1. Interfacing with sensor subsystem

The subsystem that needs to measure the groundwater level, conductivity and temperature of the water will do this by using sensors. Both analogue and digital sensors will be used. The task of the control subsystem will be to digitise the voltages, 0 - 3.3V, of the analogue sensors using ADCs with a resolution of 12 bits and furthermore to communicate with the digital sensors using I<sup>2</sup>C.

Further details about the sensor subsystem can be read in the thesis "*Sensorsysteem voor SubDowser*" written by R.C. van Beelen and S.C.A. de Groot.

### 2.3.2. Interfacing with telecommunication subsystem

Another subsystem is responsible for sending the measurement data to the server of SkyDowser. This will be done through a telecommunication system, a SIM5320 GPS module[2]. The telecommunication system is in charge of sending the measured data to the SkyDowser server. Furthermore, the telecommunication system will also be responsible for providing the location of the measurement by returning the location in coordinates to the control subsystem. Through this subsystem any changes made on the SkyDowser servers, regarding the settings,

will be sent to the control subsystem. The control subsystem will be responsible for powering up/down the telecommunication subsystem. Therefore, a keypin of the control system needs to be put high for at least six seconds to turn the module on and off. Further details about the telecommunication subsystem can be read in the thesis *"Keeping water under surveillance - Telecommunication"* written by L.B. Kunkels and J.H. Belier.

### 2.3.3. Interfacing with the power subsystem

The last subsystem is responsible for powering the total system. The subsystem above ground will be powered continuously while the whole subsystem below ground will only be powered when needed. The control system will set a pin to 3.3V to signal the power subsystem to power on the subsystems below ground. As long as this pin is high the subsystem below ground needs to be provided with power. The power supply for the subsystems below ground is done by a power over bus system. More information concerning the power subsystem can be read in the thesis *"Keeping water under surveillance - Telecommunication"* written by L.B. Kunkels and J.H. Belier.

## 2.4. Design of the total concept

To get a better understanding of what the total concept will look like in practice, the design is described and illustrated below. The design of SubDowser will look as follows:

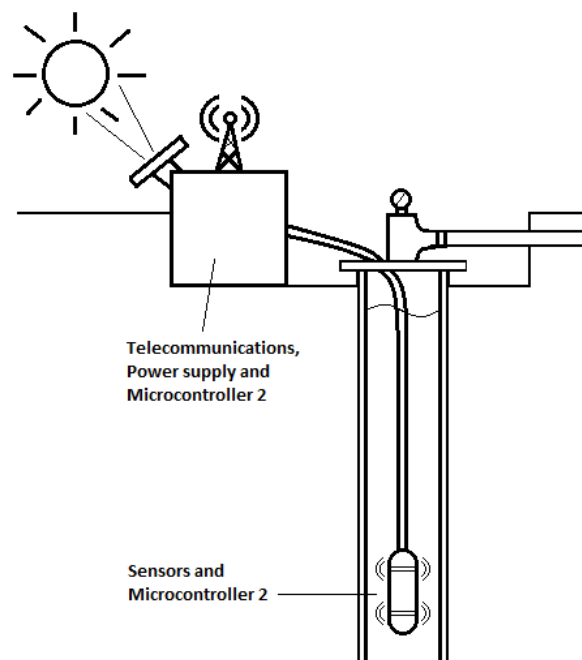


Figure 2.2: Total concept of the SubDowser

As said in the previous section, the other subgroups chose to use respectively sensors and a telecommunication module to meet their requirements. To get the data from the sensors to the telecommunication module two microcontrollers are used. One will be attached to the sensors below ground, while the other will be above ground connected to the telecommunication module.

The part of the system that is located in the water well will be made waterproof. Besides that, it should be possible to add new sensors to this part of the system. In this way the system becomes modular. The enclosure of the part above ground will be resistant against high temperatures, rain and dust.

The main power supply will be placed in the part above ground. From there both microcontrollers and sensors will be powered. To power the bottom part of the system, the power will be transferred over the communication link between the two microcontrollers, making it a power over bus system. A more detailed diagram of the interfaces is shown in figure 2.3.

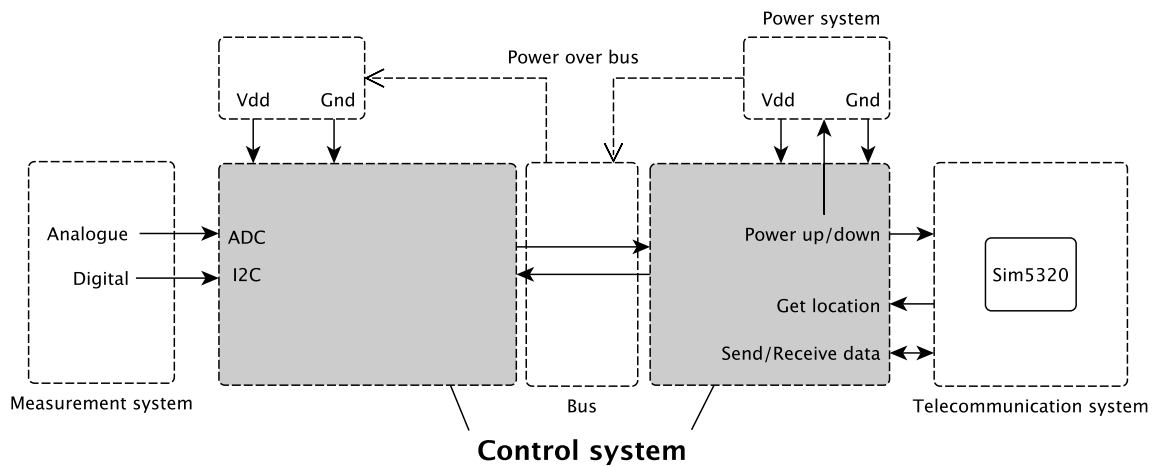


Figure 2.3: Interface diagram

The design choices that are named here concerning the control subsystem, such as why microcontrollers are used or why this particular communication protocol was chosen, this can be read in chapter 4. Other design choices concerning the other subsystems such as the power supply and sensors are explained in the previously stated theses.

# 3

## Requirements

### 3.1. Function analysis

In order to build the complete SubDowser a control subsystem is needed. As mentioned in the previous chapter this subsystem must properly interact with the other subsystems. Incoming data from the sensors needs to be processed in an organised manner and directed to the telecommunication subsystem. To get a better understanding of the control system a function analysis was done. This analysis can be seen in figure 3.1.

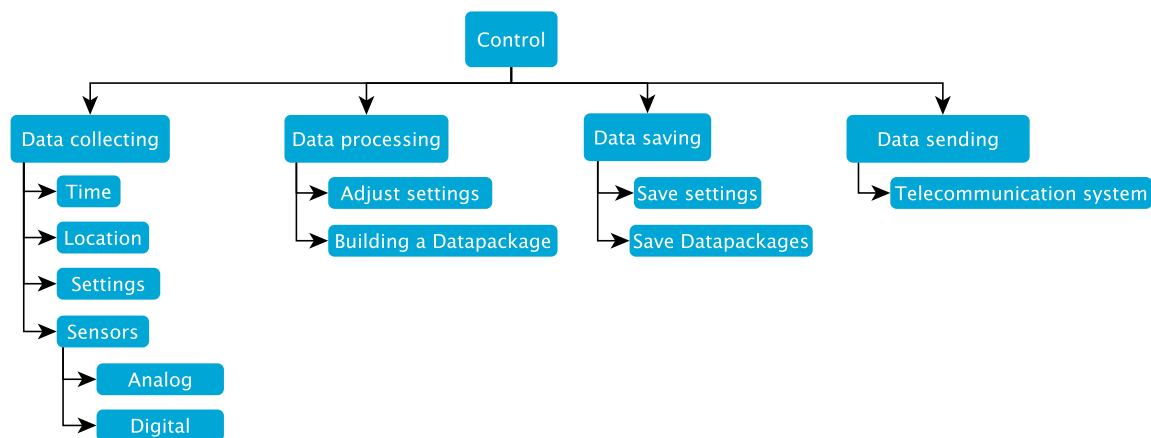


Figure 3.1: Function analysis of the control subsystem

The control system will consist of four sub-functions. The first function is to collect the different forms of data. This data includes all measurement values from the sensors, but also a timestamp, the location in coordinates and potential changes in the settings. The second function will be processing the data. The obtained data will be put together in one data-package. If there are changes in the settings according to the incoming data, the control system must adjust these settings. Furthermore, another function of the control system is to save the acquired data packages. The last function of the control subsystem will be to give the data to the telecommunication system, from where the data will be sent to the SkyDowser server.

A set of requirements was made in order to let the control subsystem meet the functional description. These requirements are for the control subsystem only, but the total concept requirements must also be kept in mind. This set of requirements can be seen below.

### 3.2. Functional requirements

1. The subsystem must read and process the sensor data
2. The subsystem must save the data in data packages
3. The subsystem must be able to update settings according to incoming data
4. The subsystem must be able to send the saved data packages to the telecommunication module

### 3.3. Non-functional requirements

#### 3.3.1. Product requirements

##### Usability

5. The subsystem must be powered with a 3.3 V power supply
6. The subsystem must have at least 3 ADC input pins with a range of 0 - 3.3 V and a minimum input-impedance of 2k ohm
7. The subsystem must have at least 2 digital input pins using I<sup>2</sup>C communication protocol
8. The subsystem must be able to read out the sensors according to the measure frequency
9. The subsystem must be able to build a data package consisting of a time stamp, location stamp, sensor data, unique product ID and measurement number
10. The location stamp must be updated every two weeks
11. The subsystem must be able to save data packages until there has been an acknowledgement that the packages have been received by the telecommunication module
12. The subsystem must be able to sent the saved data packages to the telecommunication console according to the send frequency using an asynchronous communication protocol
13. The subsystem must be able to delete data packages from the memory
14. It must be possible to add new sensors to the subsystem. Analogue sensors must have an output ranging from 0V to 3.3V. Digital sensor must be I<sup>2</sup>C protocol compatible
15. Settings must be adjustable, namely the measure and send frequency, according to incoming data
16. The subsystem must put a GPIO pin high (3.3 V) to power up the subsystem below ground. This subsystem will be powered as long as the GPIO pin is high
17. The subsystem must put a GPIO pin high (3.3 V) for at least six seconds to power up/down the telecommunication module
18. The subsystem must not cost over 12.50 Euros

#### Dimensions

19. If used underground the subsystem must fit in a 6 cm diagonal wide tube

#### Reliability

20. The subsystem must work autonomously for at least 2 years before any maintenance is required

#### Safety

21. The subsystem needs to meet the European General Product Safety Directive 2001/95/EC
22. If the subsystem is above ground it needs to be resistant to temperatures up to 70 °C
23. If the subsystem is below ground it needs to be resistant to temperatures up to 30 °C

#### Operational

24. The subsystem needs to be able to be activated without the assistance of SkyDowser, so by the customer himself

# 4

## Conceptual Design

This chapter will discuss the process of designing the control system. The control system is the subsystem that communicates with the measurement and the telecommunication subsystems. The requirements of the subsystem, stated in chapter 3, describe what this subsystem exactly needs to do but not how. As stated before, the control system has four sub-functions: Data collecting, Data processing, Data saving and Data sending.

**Data collection** Starting with data collection, the control system needs to be able to read out the sensor. So reading voltages and communicating with digital sensors using I<sup>2</sup>C is necessary. Furthermore, it needs to collect the time and the location of the measurements and receive settings from the telecommunication module.

**Data processing** To process everything in an organised manner and put all the measurements in a data-package some sort of processing is needed, so the control unit needs to consist of a processor like component.

**Data saving** As stated in the programme of requirements of the total system, the send frequency is equal to or a multitude of the measure frequency. For example, if a measurement is done each 10 minutes and the send frequency is 40 minutes, it means that the system will measure each 10 minutes, but only send these measurements every 40 minutes. So in this case the first time the system will send its data is after 40 minutes. In total four measurements have been done before the first send opportunity. These four measurements must be sent all together. Therefore, all data from each measurement must be saved until the next send opportunity occurs. The send and measure frequency settings also need to be saved, together with how many measurements SubDowser has already done.

**Data sending** The last function of the control system is to give or send the data, destined for the SkyDowser servers, to the telecommunication system. So the data location or file name needs to be given to this system.



## 4.1. Morphological table

To give an overview of which techniques and technologies are available to fulfil the functions stated above, a morphological table is made. This morphological table is shown in table 4.1. The table shows a selection of available options per sub-function.

Functions:		Options:			
<b>Data collection:</b>	Time:	RTC	GPS	Internet	Customer
	Location:	Skydowser	GPS	Customer	
	Setting:	Telecom subsystem			
	Sensor: Analogue:	ADC			
	Sensor: Digital:	I <sup>2</sup> C compatible			
<b>Data processing:</b>		Microcontroller	Microprocessor	FPGA	
<b>Data saving:</b>		Flash	RAM	SSD	Hard Disk
<b>Subsystem com:</b>		I <sup>2</sup> C/SPI	UART:RS-485/422/232	CAN	Wireless

Table 4.1: Morphological table

### 4.1.1. Data collection

**Time** There are a few options available to collect the time of a measurement. The system needs to be Plug & Play and work autonomously, as described in the requirements, so letting the customer write down the time would not be a suitable solution. The time could be collected from the Internet through the telecommunication system. However, a built-in Real Time Clock (RTC[3]) or through a GPS module would be better options. Getting the time from a GPS module, however, demands a significant amount of power: booting up, connecting to a tower, etc. But a single RTC is not really accurate when using it for two successive years, RTC typically differ a few seconds per day.

**Location** The location of the measurements is demanded as well. See appendix A.1. This location could be communicated by the customer itself through, for example, the in the future completed SkyDowser smartphone application. Another option is that SkyDowser keeps track of where SubDowser are used, for example, by letting an employee install and mark the location of SubDowser. however, the previous stated options influence how Plug & Play SubDowser is. Another option is adding a GPS module to the system which can accurately tell the measurement location. This will slightly increase the cost price of SubDowser.

**Settings** The measure and send frequency will be needed in the control system at all times and should be stored internally. Changes in these frequencies will be communicated by the telecommunication system to the control system. How this is done, can be read in chapter 5 Implementation.

**Sensor: Analogue** Voltages coming from the analogue sensors need to be digitised by the control system. ADC will be used for this task because ADCs are cheap, low powered and easily available on the electronics market. Larger systems also often have ADCs already built-in.

**Sensor: Digital** To communicate with the digital sensors the I<sup>2</sup>C protocol will be used. So the control system needs to be compatible with this protocol.

#### 4.1.2. Data processing

To process the collected data, build data packages, save and delete those packages and give the packages to the telecommunication system a processing unit is necessary. Three different processing units were found interesting and will be discussed.

**Microcontroller** A microcontroller is easy to use for developers, program environments are widely available and free to use. Furthermore, microcontrollers have lots of I/O pins, built-in counters, ADCs, clocks and sometimes even a RTC. In addition, microcontrollers are often cheap compared to other processing units. Most of the time microcontrollers have power saving modes which come in handy when the system should work autonomously for at least two years.

**Microprocessor** A microprocessor, like a Paspberry Pi zero or a BeagleBone, definitely has enough processing power to fulfil the above stated tasks. Unfortunately, this processing power comes with some significant downsides. A microprocessor is far less integrated than a microcontroller and needs to run an OS, which demands a lot of power. Microprocessor are most often used in more complex systems.

**FPGA** The last option would be an FPGA. However, FPGAs are pretty hard to use for developers. They often come in BGA packages, need explicit memory and are power inefficient. Most of the FPGAs are quite expensive too.

#### 4.1.3. Data saving

To save the data packages some kind of memory is needed. There are different kind of memories available for usage.

**Flash** Flash memory is non-votile memory, it is relatively cheap and available in lots of different sizes[4].

**RAM** RAM (random access memory) is fast and has votile memory. Since SubDowser needs to be an energy efficient system votile memory is not the best option[5].

**SSD** SSD (solid-state disk) memory is fast and offers lots of disk space. It comes in packages of GBs which is too much for SubDowser's purposes[6].

**Hard Disk** The hard disk is non-volatile memory and also comes in GBs packages. It is cheaper than SSD but more power inefficient and offers too much space as well[7].

#### 4.1.4. Subsystem communication

The SubDowser electronic system, which consists of two individual systems, must have a common communication protocol in order to exchange data. There are hundreds of communication protocols of which each can be separated into two main categories: parallel or serial communication.

Parallel interfaces transfer multiple bits at the same time. They usually require buses of data transmitting across eight, sixteen, or even more wires. While serial interfaces stream their data, one single bit at a time[8]. Because a distance of 60 metres needs to be reached, a parallel interface would be too expensive.

A serial communication protocol sounds as the best option. Serial communication protocols can be divided into two types: asynchronous and synchronous.

In asynchronous serial communication data is transferred without an external clock signal. In synchronous serial communication data lines are paired with a clock signal. This is often a faster way of data transfer, but it requires at least one more extra wire between the communicating devices.

**RS-232** RS-232 is a standard for serial communication, it is capable of moderate distances at speeds up to 20 Kbps. Cables of 9 metres are common. A RS-232 bus is an unbalanced bus that is capable of full-duplex communication between two receiver/transmitter pairs[8].

**RS-422 and RS-485** RS-422 and RS-485 are twisted pair standards which cancel out any electromagnetic interferences. This interface is capable of speeds up to 10 Mbps and could have a cable length of 1 kilometre. The difference between RS-422 and RS-485 is that RS-422 is full-duplex, meaning that data can be transferred in both directions at the same time. RS-485 is half-duplex, data can be transferred in both directions but not at the same time[9]

**SPI** The Serial Peripheral Interface (SPI) is a synchronous serial bus. SPI could achieve data rates over 1Mbps in full duplex mode and distance of 10 metre. However, SPI is synchronous and requires a clock line. [10].

**I<sup>2</sup>C** The Inter-Integrated Circuit bus (I<sup>2</sup>C) is a patented interface by Philips. It is a half-duplex, synchronous, multi-master bus requiring only two signal wires. With I<sup>2</sup>C it is possible to have three different kind of speeds, slow (under 100Kbps), fast (400Kbps), and high-speed (3.4Mbps). Cable lengths up to 25 metres are possible[11].

**CAN** CAN (Controller Area Network) is a serial two-wire full-duplex communication interface. With CAN it is possible to have a variable transfer speed. The cable length of a CAN-bus could be at most 1 kilometre. However, to use CAN, a CAN controller and a transceiver defined by ISO 11898-2/3 Medium Access Unit [MAU] standards are necessary [12].

**Wireless** A wireless communication link could be applied between the systems. However, wireless communication antennas require more power than a wired link. Furthermore, it is hard to communicate wirelessly through water, because of the water's high permittivity and electrical conductivity [13].

## 4.2. Design choices

To design the first concept of the subsystem the following design choices were made. First of all, a microcontroller was chosen instead of a FPGA or a microprocessor. A microcontroller compared to the other options, offers the all in one package, consisting of ADCs, possible RTCs and enough I/O pins. The low cost and a power saving mode make it ideal for the purpose of our subsystem.

To measure the time and still keep the system Plug & Play, a RTC in combination with a GPS module was chosen. A RTC is a cheap and easily integrated option to collect the current time. However, RTCs are not really accurate and can differ a few seconds a day. This could result in significant errors in measurement time when used in two successive years. Therefore, also a GPS module will be used to update the RTC time once in a while to overcome this time difference and to save power. Besides that, to make the system Plug & Play and collect accurate location coordinates, the GPS will also be used to get a location stamp.

To make a choice on how the communication will be organised the distance to overcome is most important, as speed does not have the highest priority in our system. I<sup>2</sup>C, SPI and RS-232 are not able to work at distances of 60 metres so are not suitable for SubDowser's purposes. Using a CAN bus could be an option, but is perhaps too costly and complicated for the subsystem. To make sure data is transferred easily and correctly the RS-485 standard will be used in combination with the UART protocol. Two differential transceiver RS-485 drivers will be used, namely the SN75HVD12P[14]. These drivers were chosen because they come in dip-8 packages, have a 3V - 3.3V supply voltage which is in the same range as the microcontrollers, they can easily overcome the distance required and are within the subsystem budget, see appendix A.5.

To make a decision of which microcontroller to use several characteristics were looked at. Those are:

- Available I/O pins, ADCs, I<sup>2</sup>C and RTC
- Enough memory
- Does it have a power saving mode?
- Software tool and open source
- Availability
- Is it scalable?

By evaluating the above points and considering the students experience with a certain microcontroller, the LPC1768 microcontroller was chosen [15]. The microcontroller has internal ADCs, flash memory, RAM, an RTC and the right I/O pins which are needed to connect to the other subsystems. Furthermore, it has multiple power saving modes and a large community to help with any development problems. The LPC1768 is a comprehensive controller and offers a broad variety of properties. Possibly too much for the purpose of SubDowser. This is done to proof the concept and can be downscaled in the future to save costs. This microcontroller can be easily downscaled, because there are simpler versions of the same family available. This will be further discussed in section 8.

### 4.3. Final concept

The final concept of the control system for SubDowser is shown in figure 4.1. Two microcontrollers will be used, one will be put in the water well next to the measurement system and the other one above ground next to the telecommunication module. These two microcontrollers will communicate with each other using two RS-485 drivers. Both power and data will be transmitted over the communication lines, making it a power over bus. The voltages of the measurement system will be read by on-board ADCs on the LPC1768 microcontroller. This microcontroller will also communicate with the digital sensors using I<sup>2</sup>C. The measurement time will be collected from the RTC. However, to correct differences in the RTC, the RTC will have to be synchronised to the GPS with a predefined timing.

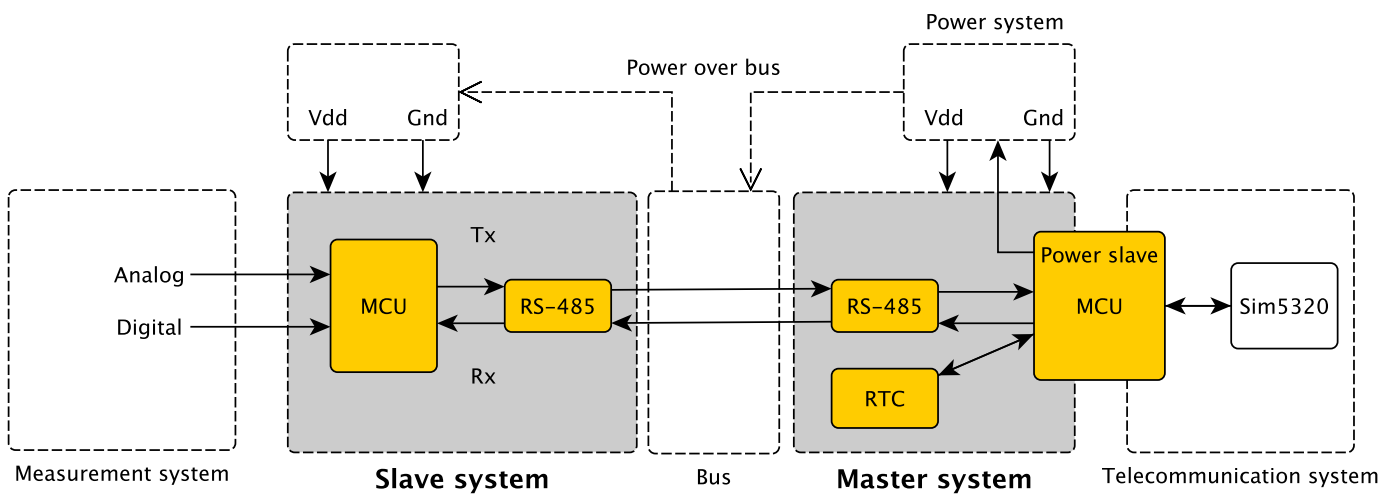


Figure 4.1: Block diagram control system

# 5

## Implementation

The microcontroller that will be used for the conceptual design is the mbed LPC1768 development board. The mbed LPC1768 is a 32-bit ARM Cortex-M3 based design manufactured by NXP. It is packaged as a small DIP form board, which makes it easy to use on breadboards. The LPC1768 chip must be powered with a 3.3 supply voltage. The board consists of a built-in RTC, 70 General Purpose I/O (GPIO) pins and six 12-bit ADC (Analog-to-Digital Converter) pins. Furthermore, the chip supports up to three serial communication ports, two I<sup>2</sup>C protocols and two SPI protocols. The LPC1768 has four power saving modes, of which the Deep power-down will be used to reduce the power consumption when the microcontroller is not used.

To program the microcontroller the C++ language will be used within the mbed programming environment [16][17]. This environment is especially designed for mbed microcontrollers developers and is an online based C++ IDE compiler. This compiler is very user-friendly, programs can easily be compiled and downloaded from the web, while libraries are largely available.

As said in the design concept, two LPC1768 microcontrollers will be used. The microcontroller that is above ground will be the master controller, the one below ground will be the slave. The master microcontroller will control the slave circuit, the power to the slave circuit and the telecommunication system.

Below each sub task or sub component will be explained and how it is implemented in one of the microcontrollers. Furthermore, it will be explained how the communication between both microcontrollers is done.

### 5.1. Sub tasks and components

#### 5.1.1. ADCs

The LPC1768 has an 8 channel 12-bit ADC, so that the incoming voltages, 0 - 3.3V, can be read with a 12 bit resolution. Reading out these voltages is done by using the function `read.u16()` in the `mbed.h` header while declaring the pin by using `AnalogIn`[18]. The `read.u16()` function returns 16 bits from which the 12 most significant bits represent the voltages. Therefore, this 16 bit result must be shifted 4 bits to the right to get a proper 12 bit value from a 12-bit ADC.

### 5.1.2. Digital

The I<sup>2</sup>C protocol is integrated in the LCP1768 controller so digital sensors that use the I<sup>2</sup>C protocol can be added to the assigned pins. Two I<sup>2</sup>C protocols are available. One is already occupied by the temperature sensor but more can be added to this bus. An address is needed to read out the I<sup>2</sup>C sensors. So when digital sensors are added these addresses need to be pre-coded in the programs. To convert the temperature sensor data from 16-bit data to 12-bit data, the 16-bit result must be shifted 4 bits to the right.

### 5.1.3. RTC

The integrated real time clock (RTC) will be used to keep track of the current time and generate an interrupt when needed[19]. The RTC time is set by using the `set_time(int seconds)` function. The argument needs to be the number of seconds since 1 January 1970 00:00:00, this is the UNIX timestamp. To set the next interrupt the function `SetNextAlarm(int second)` was created with a header file from the mbed environment [20]. This function needs the time till the next interrupt as an argument in seconds. The function will then add those seconds to the current time and set the upcoming RTC interrupt. As said in chapter 4 the RTC time will be updated once in a while with the GPS time, to overcome the time difference.

## 5.2. Data collection

Besides the data that is collected from the sensors, there is also data received from the telecommunication subsystem. This data consists of the GPS location coordinates, the GPS time, the send frequency and the measure frequency.

The GPS data is collected as a string[2], this is done with function `getGPSData()`. From this string the coordinates and time can be separated. The coordinates are located on positions 10-37 of the string and will be saved as a string of characters, see next section. The time is located on positions 39-51. The time is first converted to the UNIX time stamp, before it is saved. An example of a GPS data string is shown in figure 5.1.

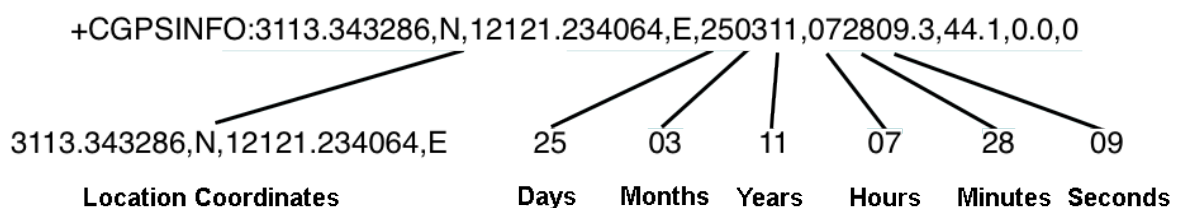


Figure 5.1: Example of a GPS string

The GPS location coordinates uses the degree:minute format. First the latitude of the current position is given in the following format: `ddmm.mmmmmmm` followed by the north or south indicator. Then the longitude position is given in the following format: `dddmm.mmmmmmm` followed by the east or west indicator.

The send and measure frequency are updated through the SkyDowser servers, so that it is possible for each customer to adjust these frequencies in the way that is best suited for their water well. These settings will be communicated through the telecommunication system. Every time the measurement data is sent to the SkyDowser servers, the send and measure

frequency are returned by the SkyDowser server. These settings will then be saved in the settings.txt file, see section 5.3.1. Both frequencies will be sent from the SkyDowser server as 9 bits data. To have variable frequencies ranging from 10 minutes to 48 hours with a 10 minute adjust ability an integer from 0-288 is needed. Therefore, at least 9 bits are needed.

### 5.3. Data saving

The LPC1768 has 512 Kb internal flash memory which will be used to save measurement data and settings. This data will be saved into two different text files, namely the settings.txt and the measurement.txt file.

#### 5.3.1. Settings

The settings.txt will consist of the following data:

- SubDowser ID
- Measurement number
- Location coordinates
- Measure frequency
- Send frequency
- Send counter
- Time of last location update

**SubDowser ID** The SubDowser ID will be implemented in the first stage of the manufacturing process. With this unique ID it is possible for the customer to request data for his SubDowser from the SkyDowser server. The SubDowser ID range will be from 0 - 65536, representing 16 bits. This ID will be represented as a number in the file.

**Measurement number** The measurement number is a number that keeps track of how many measurements a certain SubDowser has done. It has a range of 0 - 1048576, representing 20 bits. A measurement every 10 minutes for two successive years results in 105408 measurements. Because SubDowsers can operate more than two years 20 bits were chosen. This measurement number will be represented as a number in the file.

**Location coordinates** The location coordinates as mentioned previously will be given as a string of characters. In this string the coordinates are represented. The entire string will be saved in the file.



**Measure frequency** The measure frequency tells the system after how many minutes it should measure again. The measure frequency starts at a rate of 30 minutes and can be adjusted after that in multiplies of 10 minutes to a maximum of 48 hours. The measure frequency, however, can never be larger than the send frequency. For example, it is not possible to have a send frequency of 40 minutes and a measure frequency of 60 minutes. The measure frequency will be represented as a number from 0-288. To get the actual measure frequency in minutes this number must be multiplied by 10. For example, a 12 in the file will correspond to a measure frequency of 120 minutes.

**Send frequency** The send frequency tells the microcontroller after how much time the data package must be sent to the SkyDowser server. As stated in the programme of requirements in appendix A.1, this setting can be set to send the data package up from each 10 minutes up to every 48 hours. This number is represented in the file in a same way as the measure frequency.

**Send counter** The send counter keeps track of the timing to send the data package. This counter is increased every time the data packaged is not sent. The microcontroller will know when to send the data package to the SkyDowser server, when the send counter is larger than the result of the division of the send frequency by the measure frequency.

**Time last location update** As said before, the location and RTC time will be updated once every two weeks. To let the microcontroller know when to update the location and RTC time, the time of the last update is saved in the settings file. This number will be compared to the actual time, when two weeks have passed, so 1209600 seconds later, the microcontroller will update the time and location.

### 5.3.2. Measurement

All measurements will be saved in the measurement.txt file and will be sent to the SkyDowser servers. When an acknowledgement has been received from the SkyDowser server, the text file will be deleted. Each new line represents a new measurement, represented as follows:

- SubDowser ID
- Measurement number
- Time in seconds using UNIX time stamp
- Location coordinates
- First analogue (ADC) sensor 12 bits
- Second analogue (ADC) sensor 12 bits
- Third analogue (ADC) sensor 12 bits
- First digital (I<sup>2</sup>C) sensor
- Second digital (I<sup>2</sup>C) sensor

**SubDowser ID** This is the SubDowser ID as explained earlier.

**Measurement number** This measurement number is the same as described above. After each measurement it will be increased by 1.

**Time** The time of the measurement will be written in the textfile in the UNIX time stamp. For example, 20 June 2016 at 17:00:00 is represented in the UNIX time stamp as 1466442000 seconds.

**Location coordinates** The location coordinates saved in the settings.txt file will be added to measurement.txt.

**ADC** All measurement values from the ADC sensors are 16 bits, therefore these will be represented in the text file as 4 hexadecimal numbers.

**I<sup>2</sup>C** All measurement values from the I<sup>2</sup>C sensors are 16 bits, therefore these will be represented in the text file as 4 hexadecimal numbers.

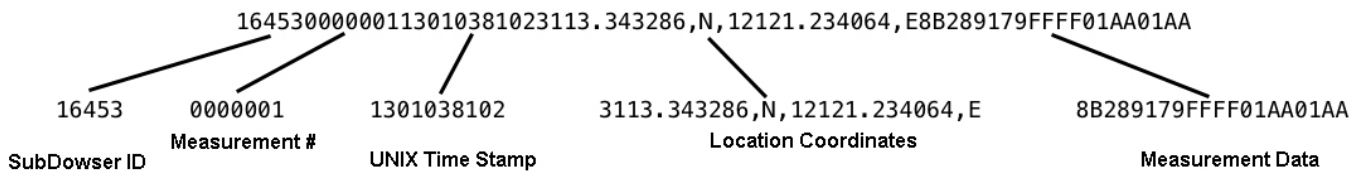


Figure 5.2: Example of a measurement string

An example of one string of data in the measurement.txt file is shown in figure 5.2. In figure 5.3 the measurement data is zoomed in, to show how the sensor data is represented in the string.



Figure 5.3: Example of the measurement data zoomed in

It is possible that a measurement.txt file consists of more than one measurement. If it is not yet time to send the data package, the new measurements are added to the measurement.txt file on a new line. This will add up until the data package is sent.

## 5.4. Data sending

The whole measurement.txt file will be sent to the SkyDowser server. As said before, this will be done through the telecommunication module. In the master microcontroller the function `TransmitData()` will be called when it is time to send the data package. The measurement.txt file will be given to this function as an argument. In the function the string of data from the text file will be sent to the server. How this is done exactly can be read in the thesis *"Keeping water under surveillance - Telecommunication"*.

When the data package has been received on the SkyDowser server, the server will return a package consisting of the measure and send frequencies. These will be updated to the settings.txt file.

## 5.5. Modular

To make the system modular, so to make it possible for the customer to add new sensors to his SubDowser, the microcontroller and the SkyDowser server must know when a new sensor is added. To come up with a solution the following concept was designed for the extra analogue sensor input. The empty spot for a new sensor will be connected to the Vdd of the microcontroller. In this way the obtained data will have a value of FFFF in the measurement data string. Active sensors almost never have this output. In this way the SkyDowser server can distinguish when a new sensor is attached to a SubDowser.

When digital sensors are added to the system they can be read with the I<sup>2</sup>C protocol. However, addresses of added digital sensors should correspond with pre-programmed addresses in the slave code. Another solution to this problem is given the recommendation chapter 8.

## 5.6. First run

When the customer first activates his/her SubDowser a FirstRun() function is called. In this function the RTC time is set according to the time coming from the GPS module. Furthermore, the GPS location is requested from the GPS module. The results of the time and location are then stored in the settings.txt file. The first measure and send frequency are set in the settings.txt file to an initialised value, namely 30 minutes for the measure frequency and 24 hours for the send frequency. The measurement number and the send counter will be set to 1 in the settings.txt file. The last act the FirstRun() function does is to SetNextAlarm(), in which the next RTC interrupt is set to 10 seconds to start the mastercontrollers state diagram. After this is done the system will go in Deep power down mode and wait for the next interrupt to occur.

## 5.7. Master microcontroller

The master controller will be stationed above ground and is responsible for the following tasks:

- Communicating with the telecommunication system
- Collecting the measurement data from the slave controller
- Processing, saving and deleting the measurement data
- Updating the settings file
- Controlling the power

In the following figure the state diagram of the master microcontroller is shown. The states and functions will be described below.

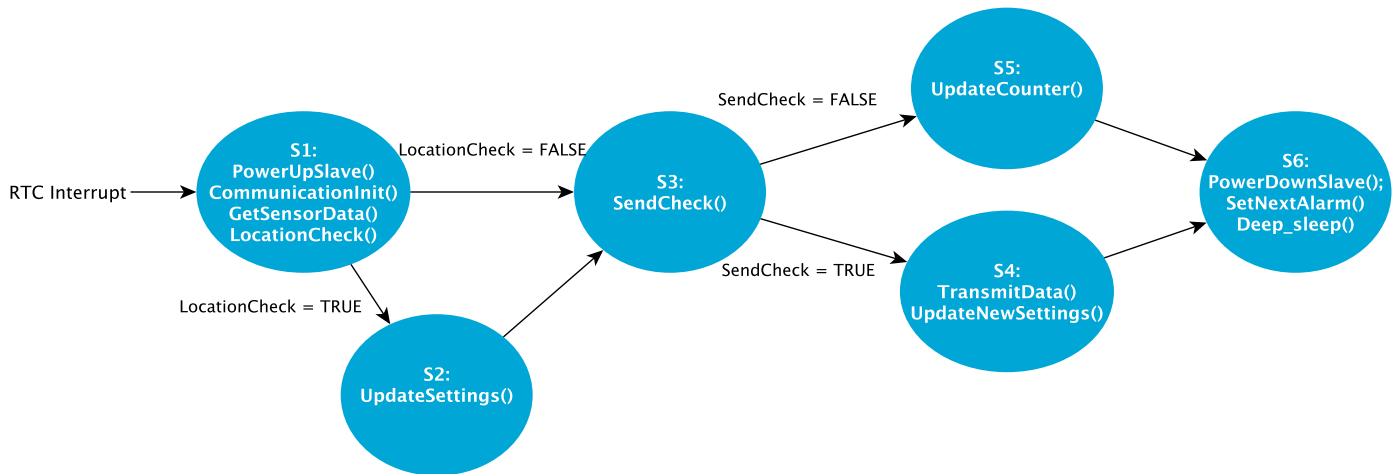


Figure 5.4: State diagram of the master microcontroller

**State 1** The master microcontroller will be in Deep power-down mode most of the time to reduce power consumption. The integrated RTC will generate an interrupt to wake up the system at a predefined time. When waking up, the system will know it needs to measure again. To start the measuring the first thing the master microcontroller has to do, is to wake up the slave microcontroller and to initialise the communication link. This is done in functions `PowerUpSlave()` and `CommunicationInit()`. The function `GetSensorData()` will acquire the measurement data from the slave and save this data to the `measurement.txt` file. The last step in this state is to check whether it is time to update the location or not, this is done in the function `LocationCheck()`. If this location check is true the microcontroller will go to state 2, if false it will go to state 3.

**State 2** In this state the location and RTC time must be updated. In `UpdateSettings()` first the GPS data from the telecommunication module will be obtained. With this GPS data the new RTC time will be set and stored in the `settings.txt` file, as well as the newly obtained location coordinates. If this is accomplished the following state will be done.

**State 3** In state 3 will be looked if it is time to send the data package to the SkyDowser server. The function `SendCheck()` will check if the send counter is larger than the result of the division of the send frequency by the measure frequency. If this is true the microcontroller will go to state 4, else state 5 will be the next state.

**State 4** In this state the `measurement.txt` file must be sent to the SkyDowser server. As mentioned in section 5.4 the function `TranmistData()` is part of the telecommunication module. The function returns two values for, respectively, the measure and send frequency. In `UpdateNewSettings()` the `settings.txt` file will be updated with these two new frequencies. Furthermore, the send counter will be reset and the `measurement.txt` file will be cleared.

**State 5** If the `SendCheck()` is false the fifth state will be executed. In this state the measurement number and the send counter will be increased by 1. The new values will be stored in the `settings.txt` file.

**State 6** In the last state, all the systems excluding the master microcontroller will be powered down. The time in which the next interrupt must occur will be set in the function `SetNextAlarm()`, after this has been done the master microcontroller will enter the Deep power down mode and wait for the next RTC interrupt to start a new cycle. To enter the Deep power down mode the header `PowerControl.h` was used [21]. The c++ code of the master microcontroller can be seen in appendix A.4.1.

## 5.8. Slave microcontroller

The slave microcontroller is stationed in the part of SubDowser that is below ground. The slave is connected to the master microcontroller with the RS-485 protocol, see section 5.9.

The state diagram of the slave microcontroller is shown in the following figure:



Figure 5.5: State diagram of the slave microcontroller

The slave is powered up by the master microcontroller as stated in the previous section. When the slave is started, the sensors are started as well. To get a correct measurement of the sensors, the slave microcontroller should wait at least 0.2 seconds before actually measuring the sensors. This 0.2 seconds is set up in communication with the measurement system group, to let the sensors stabilise.

**State 1** The slave sets up a communication link with the master in `CommunicationInit()`.

**State 2** It then checks in function `CheckReadable()` if the communication with the master microcontroller is alive. If this is the case it will go further to state 3.

**State 3** In this state the slave collects data from the sensors and send it to the master microcontroller. Firstly, the amount of characters that will be sent, will be communicated to the master controller. After that the data will be transmitted to the master microcontroller. The c++ code of the slave microcontroller can be seen in appendix A.4.2.

## 5.9. Serial communication with RS-485

The RS-485 drivers are differential line drivers, which operate in half-duplex mode. So both transmitting and receiving happen on the same line. Both the UARTs Rx and Tx will be connected to the drivers, namely to R and D respectively. Two GPIO pins will control the drivers at input /RE and DE. For transmitting both pins need to be high, while for receiving both pins need to be low [14]. The output of the drivers will be connected to each other. A block diagram of the communication set up is shown in figure 5.6.

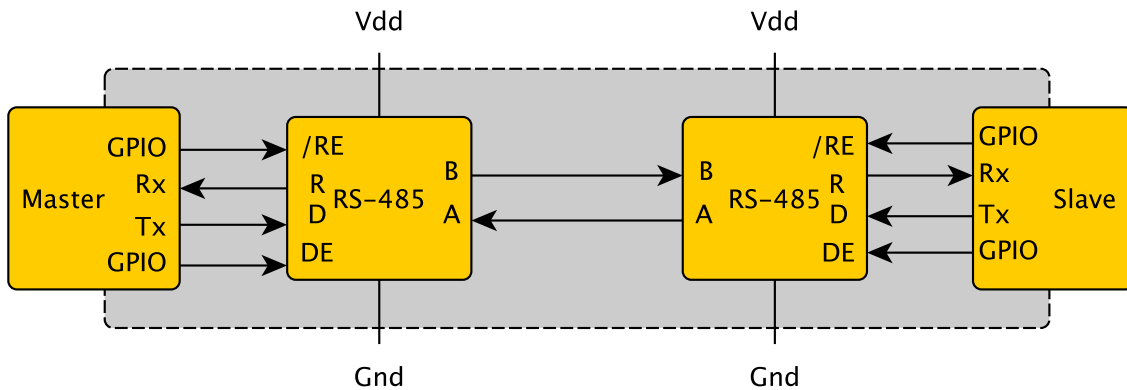


Figure 5.6: Block diagram communication set up

## 5.10. Lookup table

To read out the data on the SkyDowser server the following look up table is provided, see table 5.1. With this table it is easy to find where in the data string what kind of information is stored. Also provided is how to convert this data string to real data.

	Start location in string	Length of characters	How
SubDowser ID	0	5	Number
Measurement number	5	7	Number
Time	12	10	UNIX time stamp
Location	22	28	Degree:minutes format
First ADC sensor	50	4	Hexadecimal number
Second ADC sensor	54	4	Hexadecimal number
Third ADC sensor	58	4	Hexadecimal number
First digital sensor	62	4	Hexadecimal number
Second digital sensor	66	4	Hexadecimal number

Table 5.1: Look up table

**SubDowser ID and Measurement number** are represented as a decimal number.

**Time** is a decimal number, and represents the seconds since 1 January 1970 00:00:00. To get a real time these seconds must be converted to a UTC time. With the location the local time can be calculated.

**Location** The GPS location coordinates uses the degree:minute format. First the latitude of the current position is given in the following format: ddmm.mmmmmm followed by the north or south indicator. Then the longitude position is given in the following format: dddmm.mmmmmm followed by the east or west indicator.

**Sensor data** The sensor data is represented in hexadecimal numbers. To get from these hexadecimal numbers to real values of the sensors they must be converted. As stated before, only the 12 most significant bits represent the 0 to 3.3 voltage range for the analogue sensors. The conversion of the measurement data to usable information can be read in thesis *"Sensorsysteem voor SubDowser"*.

# 6

## Testing

In order to verify the correctness of the implementation a test plan was developed. The goal of this test plan is to check if every event or change in events is dealt with accordingly to the implementation. To get a better understanding of how they function all the sub tasks and components were first tested separately. In order not to fall into too much detail a test plan was established which deals with every possible situation.

### 6.1. Test plan

To check if the `FirstRun()` function works correctly a `settings.txt` file is created in the microcontroller memory. This `settings.txt` file only consists of SubDowser's ID on line 1, namely 19413. This simulates the final design, because the SubDowser ID would be implemented when the microcontrollers are programmed, so before selling it to the customer. In this simulation the first run measure and send frequency are set to 30 seconds and 60 seconds respectively.

After the first run the control system will go into Deep power-down mode for 30 seconds. Then an interrupt activates the subsystem and the master microcontroller follows the state diagram in chapter 5.7. After sending the first measurement data to the telecommunication system new settings are received. In order to simulate the whole system in a short time span the measure and send frequency are then set to 30 seconds and 90 seconds respectively.

In this test plan the slave microcontroller will already be provided with energy and will just wait on the signal to start sending the measurement data. start the state diagram shown in 5.5. Furthermore, the functions `TransmitData()` and `GetGPSData()` are not yet filled in with the telecommunication code, see A.4.1, but rather with testing data. The RS-485 drivers will be connected during the entire test. The length of the communication bus between the two microcontrollers will be around 20 centimetres. Two digital temperature sensor will be connected to the slave controller using the 1-wire protocol.



```

Arrived in RTC Interrupt
Communication initialized, time is: 1301038284
Measurement: 5 measuring now!
Slave MCU is alive
Amount of Character to receive: 10, characters received: 8D789339FFFF19118E
Sensor Data received and saved
Now check if location needs to be updated
Location check is false, go further
Now check if data needs to be send
Send check is true, sending data to telecommunication
MeasurementFile successfully deleted
Data send, measurement file deleted and settings updated
Setting upcoming interrupt

MCU slave alive and ready to send data
First ADC sensor gives data: 88C8
Second ADC sensor gives data: 91E9
Third ADC sensor gives data: FFFF
Temperature sensor 1 returns 25.12500 oC
Temperature sensor 2 returns 25.06250 oC
Temperature in hexadecimal numbers of sensor 1: 0192
Temperature in hexadecimal numbers of sensor 2: 0191
Response sent:
88C891E9FFFF01920191

```

Figure 6.1: Master log example

Figure 6.2: Slave log example

For testing the `LocationCheck()` function, the program is again run from the beginning. After the `settings.txt` file is initialised, the saved time will be changed to at least two weeks before the initial time by using a computer. If the `LocationCheck()` works correctly, the program should update the settings.

To test the functionality of the two microcontrollers, a debug code is developed. This code prints data to a PC terminal while the programs on the microcontrollers are running. Two examples representing the logs of the master controller and slave controller are shown in figure 6.1 and figure 6.2.

## 6.2. Results

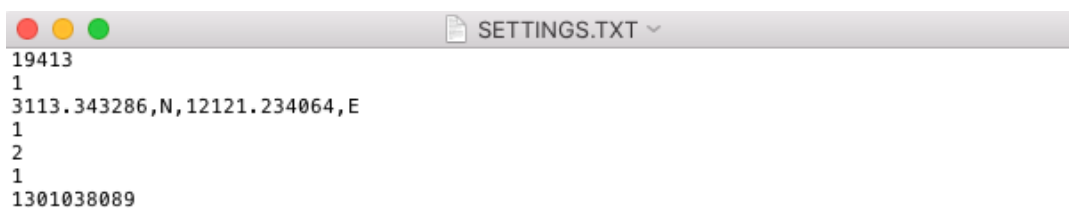
To execute the test plan, first the `setting.txt` file was created and put in the root environment of the microcontrollers memory. This file consisting of the SubDowser ID is shown in figure 6.3.



A screenshot of a file explorer window. The title bar shows three colored circles (red, yellow, green) and the text 'SETTINGS.TXT'. Below the title bar, the file name 'SETTINGS.TXT' is displayed, followed by a dropdown arrow. Below the file name, the SubDowser ID '19413' is shown.

Figure 6.3: Setting.txt file with SubDowser ID

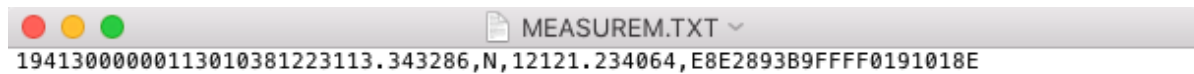
After loading the c++ code on the master and slave controllers the two reset buttons were pushed, an picture of the microcontrollers layout is shown in A.1. This automatically pushes the slave controller into its main function. The master controller starts by running the `FirstRun()` function and sets up the first `settings.txt` file shown in figure 6.4. Observable is that all the settings are set up correctly according to the list of settings stated in section 5.3.1.



A screenshot of a file explorer window. The title bar shows three colored circles (red, yellow, green) and the text 'SETTINGS.TXT'. Below the title bar, the file name 'SETTINGS.TXT' is displayed, followed by a dropdown arrow. Below the file name, the contents of the file are shown: '19413', '1', '3113.343286,N,12121.234064,E', '1', '2', '1', and '1301038089'.

Figure 6.4: Setting.txt file after `FirstRun()` function

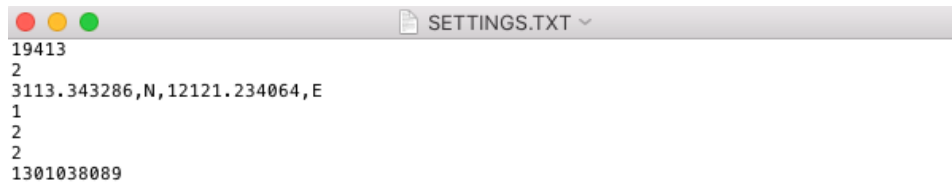
The first data package is shown in figure 6.5. Consisting of all the required data as described in section 5.3.2. The total log of the test plan of the master controller is shown in appendix A.3



```
MEASUREM.TXT
19413000000113010381223113.343286,N,12121.234064,E8E2893B9FFFF0191018E
```

Figure 6.5: Measurement.txt file after first measurement

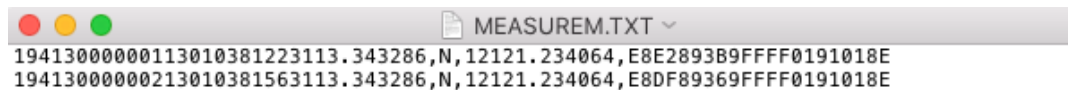
In figure 6.6 it shows that the measurement number and send counter are increased by one.



```
SETTINGS.TXT
19413
2
3113.343286,N,12121.234064,E
1
2
2
1301038089
```

Figure 6.6: Setting.txt file before second measurement

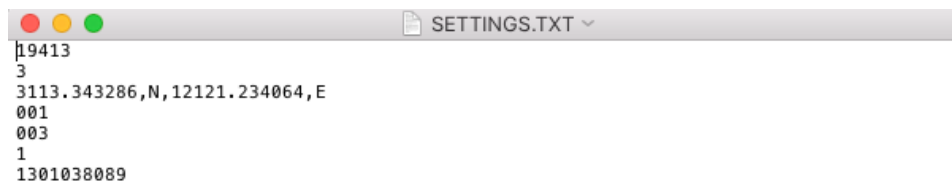
In figure 6.7 the measurement.txt file after two measurements is shown. There can be seen that the data package of the first measurement is retained in a correct way and that the second measurement is added to the file.



```
MEASUREM.TXT
19413000000113010381223113.343286,N,12121.234064,E8E2893B9FFFF0191018E
19413000000213010381563113.343286,N,12121.234064,E8DF89369FFFF0191018E
```

Figure 6.7: Measurement.txt file after second measurement

After the second measurement, as stated in the test plan, the data is sent to the telecommunication module, while new setting are received. These new setting are directly changed in the setting.txt file as shown in figure 6.8.



```
SETTINGS.TXT
19413
3
3113.343286,N,12121.234064,E
001
003
1
1301038089
```

Figure 6.8: Setting.txt file before third measurement

In addition, after sending the data, the measurement.txt file will be cleared and new measurement data can be added to the file. Three more measurement were done to check the functionality of the program. The setting.txt, measurement.txt file and the debug logs of this processes are shown in appendix A.3

In figure A.8 the results of the LocationCheck() function is shown. After initialisation the time of the last location update was set to 1200000000 seconds. Because this time corresponds to an update more than two weeks ago the LocationCheck() return a true, which results in updating the RTC and location coordinates in the settings.txt file.

# 7

## Conclusion

*Keeping water under surveillance*, a project to design a stand alone water well measuring system for water stressed regions, developed by a group of Electrical Engineering bachelor students. The subgroup of this thesis focused on designing and developing the control system. After several weeks of research and designing, the set up of using two LPC1768 microcontrollers with two RS-485 drivers was developed. This developed control system is capable of collecting all the required data: measurement data, location stamp and time stamp. Furthermore, the subsystem is capable of processing all the required data into readable data packages/strings and can save this data for a short period of time.

Looking back on the programme of requirements it can be concluded that all the requirements are met. In addition, even the total cost of the subsystem is significantly lower than the budget limit. However, the wiring between the two microcontrollers is not yet accounted for in any of the three subsystems, so this remaining budget can be used for this purpose.

Looking at the results there can be concluded that a working prototype has been developed. However, to develop a final product there are still some improvements to be done in both used implementation code and hardware. These recommendation are discussed in the following section.

# 8

## Discussion and recommendations

In this chapter a discussion about the project and the recommendations for a future design are outlined. Some of these recommendations are done to save cost of the final design, others are more based on improvements for the implementation of the concept.

### 8.1. Discussion

Due to time constraints the power over bus has not been tested yet. Therefore, it is not completely sure if the data communication over the same line will work correctly. As there is still budget left, see appendix [A.5](#), this can be used for connecting the two microcontrollers with a 60 metre wire. This wire needs to have specific characteristics, due to the fact that the wire will be used for power over bus, measuring the water level and transmitting data.

There was one downside to the usage of the used microcontroller, namely the online programming environment. There have been some occasions that it was not possible to work on the code, due to the fact that the website was under heavy load. For future work it should be handy to use an offline compiler.

During the project there were some problems with the C++ code. After a certain function was performed the program stopped working. As the compiler did not give any errors, it was hard to find where these problems lay. Luckily, after intensive searching on the mbed forums the solution to this problem was found. Unfortunately, this confiscated some valuable time.

## 8.2. Recommendations

### 8.2.1. Data Packages reduction

To reduce the data package size, by organising the data packages in a more efficient way, the memory usage of the microcontroller can be reduced. Furthermore, by reducing the data package size, the telecommunication module will transmit less data to the SkyDowser server, in this way power can be saved as well. In the current design the data packages consists of numbers, hexadecimal and characters while being saved as one big string. This could be improved by saving everything as bits and designing a proper lookup table.

### 8.2.2. C++ code can be improved

Due to the lack of experience of the students in programming, there are probably more efficient ways in programming the C++ code. This could save memory, in which case a smaller microcontroller could be chosen to function as the master microcontroller.

### 8.2.3. More efficient microcontroller

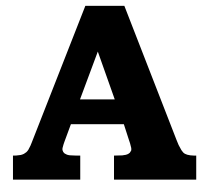
To save cost, the slave microcontroller can be downscaled a lot as it only has to get data from a maximum of five sensors, many pins are unused now. Also, this microcontroller has too much memory for its purpose. A better microcontroller for the slave microcontroller will be the microcontroller LPC1124, this will save up to 3 Euros for the final design.

### 8.2.4. Data needs to get flushed

To make sure there will be enough space on the master microcontroller's memory, the memory must be flushed once in a while. It was noticed that the memory of the microcontrollers stores all deleted files in a trash map. These deleted data will add-up during the process, which in the end will make it impossible to save new data. If this is the case a thorough check should be done to find out if this is also the case if a different microcontroller for a future design is used.

### 8.2.5. Adding digital sensors

If new I<sup>2</sup>C are added to the system the address is needed. This needs to be implemented in the program of the slave controller. In the current design pre-defined addresses are chosen. In the future these addresses could be send through the server to the control subsystem. A new function could be designed for this task.



# Appendix

## A.1. Programme of requirements

### A.1.1. Functional requirements

1. The product needs to measure the groundwater level, conductivity and temperature of the water
2. The product needs to send the measurement data to the server of SkyDowser

### A.1.2. Non-functional requirements

#### Product requirements

#### Usability

3. The product needs to be able to adjust settings, such as measuring and sending frequency, according to changes given by the SkyDowser servers
4. The product needs to be Plug & Play for the customer
5. The product needs to be modular
6. The cost price, without assembly cost of the product, needs to be lower than 50 Euros at an order of 10000 pieces
7. The annual operating cost may not exceed 10 Euros
8. The product shall first be used in Kenya and Tanzania

#### Performance

9. The measure frequency must be adjustable up to once every 10 minutes. From that point it should be possible to adjust this setting in steps of 10 minutes, with a maximum measure frequency of 48 hours

10. The send frequency must be adjustable up to once every 10 minutes. From that point it should be possible to adjust this setting in steps of 10 minutes, with a maximum send frequency of 48 hours
11. The measured data must be labeled with time at an accuracy of **1 second** and must use the UNIX timestamp format
12. The location coordinates of SubDowser must be sent to the SkyDowser servers at least once, with an accuracy of **100 m**
13. All the measured data must be sent correctly to the SkyDowser servers
14. The product must also be capable of operating in hilly area (0-1000m)
15. The product must work autonomously for at least **2 years** before any maintenance is required. Manufacturing failures are excluded
16. The conductivity should be able to be measured within a reach of **0 to 15000 uS/cm** at a resolution of **1uS/cm** and a maximum deviation of **±1.5%**
17. The temperature of the water in the well should be able to be measured within a reach of **5 to 20 °C** at a resolution of **0.5 °C** and a maximum deviation of **±1°C**
18. The water level should be able to be measured within a reach of **0 to 60 m** with a maximum water column of **20 m** and a accuracy of **±5 cm** and a resolution of **1 cm**

## Dimensions

19. The SubDowser must fit in a **6 cm** diagonal wide tube
20. The SubDowsers have an average density of **2-3** SubDowser per square kilometre
21. The average distance between two SubDowsers in a work area is a maximum of 1 kilometre

## Safety

22. The subsystem needs to meet the European General Product Safety Directive 2001/95/EC
23. The measurement data may not be intercepted while being transferred to the SkyDowser servers
24. If the product is above ground it needs to be resistant to temperatures up to **70 °C**
25. If the product is below ground it needs to be resistant to temperatures up to **30 °C** when operational. During transport it needs to be resistant to temperatures up to **70 °C**
26. If a part of the SubDowser is underground and underwater it should meet the IP-68 encryption
27. If a part of the SubDowser is above ground it should meet the IP-64 encryption
28. The part above ground must be resistant to normal mechanical impact, minimum of IK-07.

29. The part underground must be resistant to mediate mechanical impact, minimum of IK-05.

### **Operational**

30. The product must be easy to use by the customer
31. The product needs to be able to be activated without the assistance of SkyDowser, so by the customer himself

### **Environmental**

32. The product must not contaminate the groundwater as specified in the law for groundwater of the European Union



A.2. Layout LPC1768

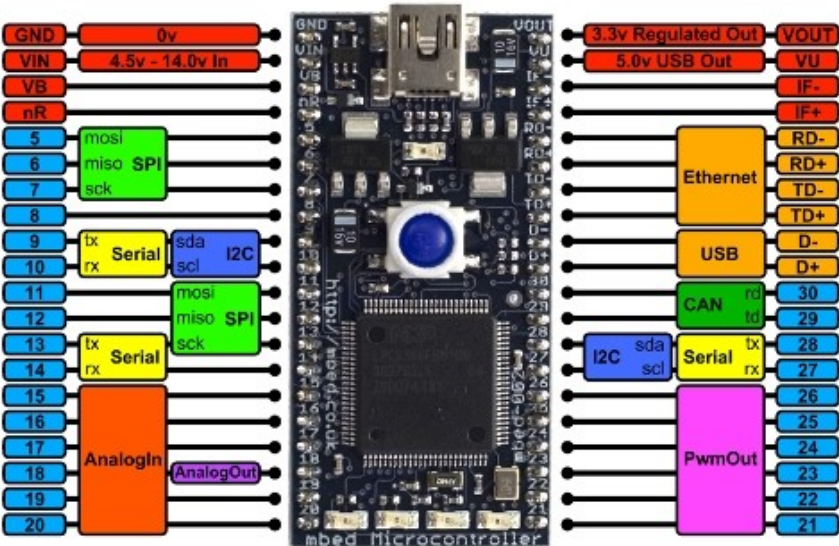


Figure A.1: Layout LPC1768

## A.3. Results

First Run succeeded

Arrived in RTC Interrupt  
Communication initialized, time is: 1301038119  
Measurement: 1 measuring now!  
Slave MCU is alive  
Amount of Character to receive: 10, characters received: 8E2893B9FFFF19118E  
Sensor Data received and saved  
Now check if location needs to be updated  
Location check is false, go further  
Now check if data needs to be send  
Send check is false  
Setting upcoming interrupt

Arrived in RTC Interrupt  
Communication initialized, time is: 1301038153  
Measurement: 2 measuring now!  
Slave MCU is alive  
Amount of Character to receive: 10, characters received: 8DF89369FFFF19118E  
Sensor Data received and saved  
Now check if location needs to be updated  
Location check is false, go further  
Now check if data needs to be send  
Send check is true, sending data to telecommunication  
MeasurementFile successfully deleted  
Data send,measurement file deleted and settings updated  
Setting upcoming interrupt

Arrived in RTC Interrupt  
Communication initialized, time is: 1301038217  
Measurement: 3 measuring now!  
Slave MCU is alive  
Amount of Character to receive: 10, characters received: 8D78939FFFF19118E  
Sensor Data received and saved  
Now check if location needs to be updated  
Location check is false, go further  
Now check if data needs to be send  
Send check is false  
Setting upcoming interrupt

Arrived in RTC Interrupt  
Communication initialized, time is: 1301038251  
Measurement: 4 measuring now!  
Slave MCU is alive  
Amount of Character to receive: 10, characters received: 8E189389FFFF19118E  
Sensor Data received and saved  
Now check if location needs to be updated  
Location check is false, go further  
Now check if data needs to be send  
Send check is false  
Setting upcoming interrupt

Arrived in RTC Interrupt  
Communication initialized, time is: 1301038284  
Measurement: 5 measuring now!  
Slave MCU is alive  
Amount of Character to receive: 10, characters received: 8D789339FFFF19118E  
Sensor Data received and saved  
Now check if location needs to be updated  
Location check is false, go further  
Now check if data needs to be send  
Send check is true, sending data to telecommunication  
MeasurementFile successfully deleted  
Data send,measurement file deleted and settings updated  
Setting upcoming interrupt

Figure A.2: Total log of test master state diagram

A.3.1. Additional measurement results

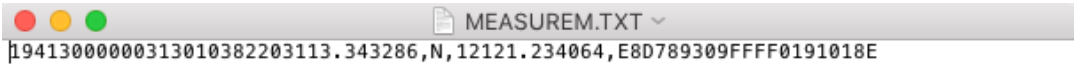


Figure A.3: Measurement.txt file after third measurement



Figure A.4: Setting.txt file before fourth measurement

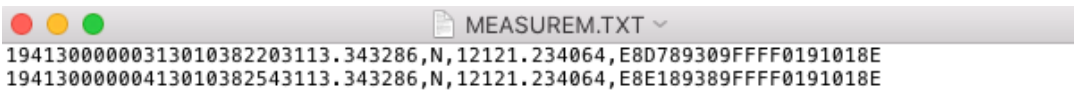


Figure A.5: Measurement.txt file after fourth measurement



Figure A.6: Setting.txt file before fifth measurement

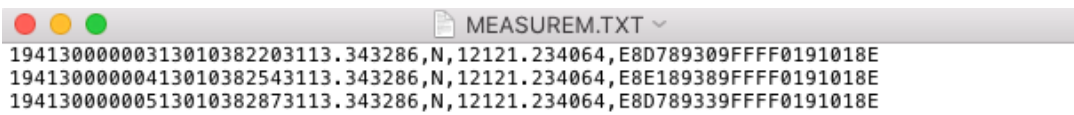


Figure A.7: Measurement.txt file after fifth measurement

### A.3.2. LocationCheck() result

```

Arrived in RTC Interrupt
Communication initialized, time is: 1301038119
Measurement: 1 measuring now!
Slave MCU is alive
Amount of Character to receive: 10, characters received: 8F5894F9FFFF199192
Sensor Data received and saved
Now check if location needs to be updated
Location check is false, go further
Now check if data needs to be send
Send check is false
Setting upcoming interrupt

Arrived in RTC Interrupt
Communication initialized, time is: 1301038153
Measurement: 2 measuring now!
Slave MCU is alive
Amount of Character to receive: 10, characters received: 8B589199FFFF199192
Sensor Data received and saved
Now check if location needs to be updated
Location check is true, update settings now
Settings updated
Now check if data needs to be send
Send check is true, sending data to telecommunication

```

Figure A.8: Master log, during LocationCheck() function

## A.4. Code

### A.4.1. Master microcontroller

#### Main code

---

```

1  #include <masterHeader.h>

3
4  int main()
5  {
6      FirstRun(); // Function to set first settings and RTC
7      pc.printf("First Run succeeded\n\n");

9      while(1) {
10         DeepPowerDown(); // After each interrupt the microcontroller goes into
11         deep power down
12     }
13

15 void RTCInterrupt(void) // Control function
16 {
17     // Begin state1
18     pc.printf("Arrived in RTC Interrupt\n");
19     CommunicationInit();
20     pc.printf("Communication initialized , time is: %d\n", time(NULL));

21     GetSensorData();
22     pc.printf("\nSensor Data received and saved\n");

```

```

23 pc.printf("Now check if location needs to be updated\n");
   // End state1
25 if(LocationCheck()) {
    pc.printf("Location check is true, update settings now\n");
27    UpdateSettings (); // state2
    pc.printf("Settings updated\n");
29 } else {
    pc.printf("Location check is false, go further\n"); // state 3
31 }
pc.printf("Now check if data needs to be send\n");
33 if(SendCheck()) { // state 4
    string Settings;
35    pc.printf("Send check is true, sending data to telecommunication\n");
    PowerControlSim = 1; // power up sim module by setting GPIO pin high for 6
    seconds
37    wait(6);
    PowerControlSim = 0;
39    Settings = TransmitData ();
    PowerControlSim = 1; // power down sim module by setting GPIO pin high for
    6 seconds
41    wait(6);
    PowerControlSim = 0;
43    UpdateNewSettings(Settings);
    pc.printf("Data send,measurement file deleted and settings updated\n");
45 } else {
47    UpdateCounters(); // state 5
    pc.printf("Send check is false\n");
49 }
   // state 6
51 pc.printf("Setting upcoming interrupt\n\n");
    SetNextAlarm(); // Sets next alarm
53 }

```

---

#### Header file of master

---

```

1 #include "mbed.h"
  #include "RTC.h"
3 #include <string>
  #include "PowerControl.h"
5 #include <fstream>
  #include <iostream>
7 #include <iomanip>
  #include <sstream>
9
   // declare personal functions
11 void RTCInterrupt(void); // Control function
   void SetNextAlarm (void); // Sets next RTC Interrupt
13 void SetFirstAlarm (void); // Set first alarm after first run
   string TransmitData (void); // Calls telecommunication system to send measurement
   data
15 void UpdateSettings (void); // Updates settings file with new location and update
   time
   void UpdateNewSettings(string Settings); // Updates settings with new Frequencies
   and reset SendCounter
17 void UpdateCounters(void); // Updates SendCounter and MeasurementNumber
   bool SendCheck(void); // Checks if data needs to be send
19 bool LocationCheck(void); // Checks if location needs to get updated

```

```

void GetSensorData(void); // Get measurement data from slave MCU
21 void CommunicationInit(void); // Initialise baudrate and communication format
void FirstRun(void); // Initialise RTC clock, Setting file and sets first alarm
23 string GetGPSData (void); // Get GPS Data from telecom, coordinates and time
void SetRTCtime (string GPSData); // Set RTC time using GPSData
25
DigitalOut led1(LED1);
27 DigitalOut led2(LED2);
DigitalOut pin5(p5);
29 DigitalOut pin6(p6);
DigitalOut PowerControlSlave(p7);
31 DigitalOut PowerControlSim(p8);
LocalFileSystem local("local"); // Create the local filesystem under the name "
    local"
33 Serial pc(USBTX, USBRX); // Set up link with computer
Serial MCU1(p9, p10); // Set up link with slave MCU
35
void SetNextAlarm (void) // Sets next RTC Interrupt
37 {
    ifstream input("/local/settings.txt"); // Gets measurement frequency out of
    settings.txt
39     int line = 4;
    string line4;
41     for (int i = 1; i <= line; i++)
        getline(input, line4);
43     input.close();

45     int NextInterrupt = (std::atoi (line4.c_str())) * 600; //Converts string to
    integer

47     time_t rawtime;
    rawtime = (time (NULL) + NextInterrupt); // adds current time to time in
    seconds till next interrupt
49     struct tm * timeinfo;
    tm t;

51     //Put time in t tm struct ready to set alarm
53     timeinfo = localtime (&rawtime);
    t.tm_sec = timeinfo->tm_sec;
55     t.tm_min = timeinfo->tm_min;
    t.tm_hour = timeinfo->tm_hour;
57     t.tm_mday = timeinfo->tm_mday;
    t.tm_wday = -1; //timeinfo->tm_wday;
59     t.tm_yday = -1; //timeinfo->tm_yday;
    t.tm_mon = -1; //(timeinfo->tm_mon + 1);
61     t.tm_year = -1; //(timeinfo->tm_year + 1900);

63     RTC::alarm(&RTCInterrupt, t); // sets next alarm in the RTC and assigns
    function to jump to (Interrupt)
    }
65
void SetFirstAlarm (void)
67 {
    time_t rawtime;
69     rawtime = (time (NULL) + 30); // adds 30 seconds to current time
    struct tm * timeinfo;
71     tm t;

73     //Put time in t tm struct ready to set alarm
    timeinfo = localtime (&rawtime);
75     t.tm_sec = timeinfo->tm_sec;

```

```

    t.tm_min = timeinfo->tm_min;
77    t.tm_hour = timeinfo->tm_hour;
    t.tm_mday = timeinfo->tm_mday;
79    t.tm_wday = -1; //timeinfo->tm_wday;
    t.tm_yday = -1; //timeinfo->tm_yday;
81    t.tm_mon = -1; //(timeinfo->tm_mon + 1);
    t.tm_year = -1; //(timeinfo->tm_year + 1900);
83
    RTC::alarm(&RTCInterrupt, t); // sets next alarm in the RTC and assigns
    function to jump to (Interrupt)
85 }

87
string TransmitData (void) // Calls telecommunication system to send measurement
    data
89 {
    string Data = "001003";
91    return Data;
    }
93
void UpdateSettings (void) // Updates settings file with new location and update
    time
95 {
    string GPSData = GetGPSData();
97    SetRTCtime(GPSData); // Set RTC using the GPSData string

99    ifstream input("/local/settings.txt"); // Copy text and create settings
    string line1, line2, line3, line4, line5, line6, line7; // could be improved
    further
101    getline( input, line1 );
    getline( input, line2 );
103    getline( input, line3 );
    getline( input, line4 );
105    getline( input, line5 );
    getline( input, line6 );
107    getline( input, line7 );
    input.close();
109
    ofstream SettingsFile;
111    SettingsFile.open ("/local/settings.txt");
    SettingsFile << line1 << endl; // SubDowser ID
113    SettingsFile << line2 << endl; // Measurement number set to 1
    SettingsFile << GPSData.substr(10, 28) << endl; // GPS location in
    coordinates
115    SettingsFile << line4 << endl; // Measurment frequency 3 * 10 minutes = 30
    minutes
    SettingsFile << line5 << endl; // Send frequency 144 * 10 minutes = 24 hours
117    SettingsFile << line6 << endl; // Sendcounter needs to when to send next
    SettingsFile << time(NULL) << endl; // Time last location check
119    SettingsFile.close();
    }
121
void UpdateNewSettings(string Settings) // Updates settings file with new location
    and update time
123 {
    if( remove("/local/measurement.txt") != 0 ) // Delete all measurements
125        pc.printf( "Error deleting file\n" );
    else
127        pc.printf( "MeasurementFile successfully deleted\n" );

129    ifstream input("/local/settings.txt"); // Copy text and set new settings

```

```

    string line1, line2, line3, line4, line5, line6, line7;    // could be improved
    further
131  getline( input, line1 );
    getline( input, line2 );
133  getline( input, line3 );
    getline( input, line4 );
135  getline( input, line5 );
    getline( input, line6 );
137  getline( input, line7 );
    input.close();

139
    int MeasurementNumber = std::atoi (line2.c_str());
141  MeasurementNumber = MeasurementNumber + 1; // Adds measurement number with one

143  ofstream SettingsFile;
    SettingsFile.open ("/local/settings.txt");
145  SettingsFile << line1 << endl; // SubDowser ID
    SettingsFile << MeasurementNumber << endl; // set measurement number
147  SettingsFile << line3 << endl; // GPS location in coordinates
    SettingsFile << Settings.substr (0, 3) << endl;    // Measurment frequency 3 *
    10 minutes = 30 minutes
149  SettingsFile << Settings.substr (3, 3) << endl; // Send frequency 144 * 10
    minutes = 24 hours
    SettingsFile << "1" << endl;    // Sendcounter needs to when to send next
151  SettingsFile << line7 << endl;    // Time last location check
    SettingsFile.close();
153 }

155 void UpdateCounters(void)
{
157     ifstream input("/local/settings.txt"); // Copy text and set new counters
    string line1, line2, line3, line4, line5, line6, line7;    // could be improved
    further
159     getline( input, line1 );
        getline( input, line2 );
161     getline( input, line3 );
        getline( input, line4 );
163     getline( input, line5 );
        getline( input, line6 );
165     getline( input, line7 );
        input.close();

167
        int MeasurementNumber = std::atoi (line2.c_str());
169         int SendCounter = std::atoi (line6.c_str());
        MeasurementNumber = MeasurementNumber + 1; // Adds Measurement number with one
171         SendCounter = SendCounter + 1; // Adds SendCounter number with one

173     ofstream SettingsFile;
        SettingsFile.open ("/local/settings.txt");
175     SettingsFile << line1 << endl; // SkyDowser ID
        SettingsFile << MeasurementNumber << endl; // Measurement number set to 1
177     SettingsFile << line3 << endl; // GPS location in coordinates
        SettingsFile << line4 << endl;    // Measurment frequency 3 * 10 minutes = 30
        minutes
179     SettingsFile << line5 << endl; // Send frequency 144 * 10 minutes = 24 hours
        SettingsFile << SendCounter << endl;    // Sendcounter needs to when to send
        next
181     SettingsFile << line7 << endl;    // Time last location check
        SettingsFile.close();
183 }

```



```

185 bool SendCheck(void) // Checks if data needs to be send
186 {
187     ifstream input("/local/settings.txt"); // Get settings and counter out of
188     settings.txt
189     int line = 4;
190     string line4, line5, line6;
191     for (int i = 1; i <= line; i++)
192         getline(input, line4);
193
194     getline(input, line5);
195     getline(input, line6);
196     input.close();
197
198     double MeasureFrequency = std::atof (line4.c_str());
199     double SendFrequency = std::atof (line5.c_str());
200     double SendCounter = std::atof (line6.c_str());
201
202     return (SendCounter >= SendFrequency/MeasureFrequency); // Check if its time
203     to send
204 }
205
206 bool LocationCheck(void) // Checks if location needs to get updated
207 {
208     ifstream input("/local/settings.txt"); // Get last location time out of
209     settings
210     int line = 7;
211     string line7;
212     for (int i = 1; i <= line; i++)
213         getline(input, line7);
214     input.close();
215
216     int LastLocationCheck = std::atoi (line7.c_str());
217
218     return (time(NULL) >= LastLocationCheck + 1209600) ; // Check if its time to
219     get new location
220 }
221
222 void GetSensorData(void) // Get measurement data from slave MCU
223 {
224     PowerControlSlave = 1; // Power up slave circuit underground
225     int size_dataIN;
226     unsigned char b = 0xAA; // just an character
227
228     ifstream input("/local/settings.txt");
229     string line1, line2, line3, line4, line5, line6;
230
231     getline( input, line1 ) ; // SubDowser ID
232     getline( input, line2 ) ; // Measurement number
233     getline( input, line3 ) ; // Location coordinates
234     input.close();
235
236     int ID = std::atoi (line1.c_str());
237     int MeasurementNumber = std::atoi (line2.c_str());
238
239     pc.printf("Measurement: %d measuring now!\n", MeasurementNumber);
240
241     while(true) { // Check if slave is online and writeble
242         pin6 = 1;
243         pin5 = 1;
244         if (MCU1.writeable()) {
245             MCU1.putc(b);

```

```

243         pc.printf("Slave MCU is alive\n");
           break;
       }
245     }
    while(true) { // Read how much characters need to be recieved
247         pin5 = 0;
           pin6 = 0;
249         if (MCU1.readable()) {
           size_dataIN=MCU1.getc();
251             break;
           }
253     }
    char dataIN[size_dataIN];

255     pc.printf("Amount of Character to receive: %d, characters received: ",
size_dataIN);
257     for (int i=0; i<size_dataIN; i++) {
        while (true) { // Get characters and save them in array
259             if (MCU1.readable()) {
                dataIN[i]= MCU1.getc();
261                 dataIN[i+1] = '\0';
                pc.printf("%X", dataIN[i]);
263                 break;
            }
265         }
    }
267     PowerControlSlave = 0; // Power down slave circuit underground

269     stringstream append;
    string result; // Create string with all the data of a measurement
271     append << setfill ('0') << setw (5) << ID;
    append << setfill ('0') << setw (7) << MeasurementNumber;
273     append << time(NULL);
    append << line3;
275     for (int i = 0; i<size_dataIN; i++) {
        append << hex << uppercase << setw (2) << setfill ('0') << (int)dataIN[i
];
277     }
    append << '\n';
279     result = append.str();

281     ofstream MeasurementFile; // Put string in measurement.txt
    MeasurementFile.open("/local/measurement.txt", ofstream::out | ofstream::app);
283     MeasurementFile << result;
    MeasurementFile.close();
285 }

287 void CommunicationInit(void) // Initialise baudrate and communication format
{
289     MCU1.format(8, Serial::None, 1);
    MCU1.baud(9600);
291 }

293 void FirstRun (void)// Initialise RTC clock, Setting file and sets first alarm
295 {
    string GPSData = GetGPSData();
297     SetRTCTime(GPSData);

299     ifstream input("/local/settings.txt"); // Get ID out of text to merge with
other settings

```

```

    string line1;                                // could be improved further
301  getline( input, line1 );
    input.close();
303
    ofstream SettingsFile;
305  SettingsFile.open ("/local/settings.txt");
    SettingsFile << line1 << endl; // SkyDowser ID
307  SettingsFile << "1" << endl; // Measurement number set to 1
    SettingsFile << GPSData.substr (10, 28) << endl; // GPS location in
    coordinates
309  SettingsFile << "1" << endl; // Measurment frequency 3 * 10 minutes = 30
    minutes
    SettingsFile << "2" << endl; // Send frequency 144 * 10 minutes = 24 hours
311  SettingsFile << "1" << endl; // Sendcounter needs to when to send next
    SettingsFile << time(NULL) << endl; // Time last location check
313  SettingsFile.close();
    SetFirstAlarm(); // Set next alarm will be in 10 seconds
315
}
317
string GetGPSData () // Get GPS Data from telecom, coordinates and time
319 {
    string Data = "+CGPSINFO:3113.343286,N,12121.234064,E
    ,250311,072809.3,44.1,0.0,0";
321  return Data;
}
323
void SetRTCTime (string GPSData) // Set RTC time using GPSData
325 {
    time_t rawtime;
327  struct tm *timeinfo;
    //
329  time ( &rawtime);
    timeinfo = localtime ( &rawtime );
331  timeinfo->tm_sec = (GPSData[50] - '0')* 10 + (GPSData[51] - '0'); // Setting
    time values corresponding to GPS string
    timeinfo->tm_min = (GPSData[48] - '0')* 10 + (GPSData[49] - '0');
333  timeinfo->tm_hour = (GPSData[46] - '0')* 10 + (GPSData[47] - '0');
    timeinfo->tm_mday = (GPSData[39] - '0')* 10 + (GPSData[40] - '0');
335  timeinfo->tm_mon = (GPSData[41] - '0')* 10 + (GPSData[42] - '0') - 1;
    timeinfo->tm_year = (GPSData[43] - '0')* 10 + (GPSData[44] - '0') + 2000 -
    1900;
337  //
    rawtime = mktime (timeinfo); // Converts dates to seconds since 1 january
    1970
339  set_time(rawtime); // set time in RTC
}

```

---

### A.4.2. Slave microcontroller

#### Main code

```

2  #include "Slave_header.h"

4  int main()
    {
6      CommunicationInit(); // State 1
    }

```

```

8   while(1) {
        CheckReadable(); //State 2
10       wait(1); // wait for sensors to get stable
12       LoadSensorData(); //State3
14   }
}

```

---

#### Header file of slave

---

```

1   #include "mbed.h"
3   #include "DS1820.h"

5   #define MAX_PROBES      16
   #define DATA_PIN      p8

7   DS1820* probe[MAX_PROBES];

9   Serial micro(p9, p10); //tx, rx
11  Serial pc(USBTX, USBRX); //tx, rx

13  DigitalOut dout(LED1); // debug leds
   AnalogIn depth(A2); // analog sensor depth
15  AnalogIn cond(A1); // analog sensor conductivity
   AnalogIn extra(A0); // extra analog sensor
17  DigitalOut pin1(p5); // connected to DE of rs485 driver chip
   DigitalOut pin2(p6); // connected to /RE of rs485 driver chip
19  I2C i2c(p28, p27); // Setting up I2C bus

21  void CommunicationInit(void); // Initialise baudrate and communication format
   void CheckReadable(void); // Check if Master is transmitting
23  void LoadSensorData(void); // Do the measuring and send data to master

25  void CommunicationInit(void) // Initialise baudrate and communication format
   {
27       micro.format(8, Serial::None, 1);
        micro.baud(9600);
29       pc.format(8, Serial::None, 1);
        pc.baud(9600);
31  }

33  void CheckReadable(void) // Check if Master is transmitting
   {
35       while(true) {
37           pin1 = 0; // both pins low to disable driver and activate reciever
           pin2 = 0;
39           if (micro.readable()) {
               int start = micro.getc();
41           pc.printf("MCU slave alive and ready to send data\n"); //Binnenkomend
               signaal checken
               break;
43       }
   }
45 }

```

```

47 void LoadSensorData(void) // Do the measuring and send data to master
{
49     int PredefinedAdress1 = 70, PredefinedAdress2 = 71; // Adresses for the I2C
        protocol
        char result1[2], result2[2];
51     char measurement[10];
        uint16_t temp_depth;
53     uint16_t temp_cond;
        uint16_t temp_extra;
55
        temp_depth = depth.read_u16(); // Not connected returns random values
57     temp_cond = cond.read_u16(); // Not connected random values
        temp_extra = extra.read_u16(); // extra adc connected to v_out 3.3v so returns
        FFFF
59
61     measurement[0] = temp_depth >> 8; // Splitting total data into seperated hexa
        numbers
        measurement[1] = temp_depth & 0xFF; // Ready to be send as characters
63     measurement[2] = temp_cond >> 8;
        measurement[3] = temp_cond & 0xFF;
65     measurement[4] = temp_extra >> 8;
        measurement[5] = temp_extra & 0xFF;
67
        pc.printf("First ADC sensor gives data: %.2X%.2X\n", measurement[0],
        measurement[1]);
69     pc.printf("Second ADC sensor gives data: %.2X%.2X\n", measurement[2],
        measurement[3]);
        pc.printf("Third ADC sensor gives data: %.2X%.2X\n", measurement[4],
        measurement[5]);
71
        i2c.read(PredefinedAdress1, result1, 2); // Reading out I2C sensors using
        predefine address
73     i2c.read(PredefinedAdress2, result2, 2); //Reading out I2C sensors using
        predefine address
75
        measurement[6] = result1[0]; // MSB first
        measurement[7] = result1[1];
77     measurement[8] = result2[0]; // MSB first
        measurement[9] = result2[1];
79
        // This code was used in the test plan.
81
        /// Initialize the probe array to temperature sensor
83     // int num_devices = 0;
        // while(DS1820::unassignedProbe(DATA_PIN)) {
85     //     probe[num_devices] = new DS1820(DATA_PIN);
        //     num_devices++;
87     //     if (num_devices == MAX_PROBES)
        //         break;
89     // }
        //
91 //     probe[0]->convertTemperature(true, DS1820::all_devices); // Start
        temperature conversion, wait until ready
        //     probe[1]->convertTemperature(true, DS1820::all_devices); // Start
        temperature conversion, wait until ready
93 //     pc.printf("Temperature sensor 1 returns %3.5f oC\r\n", probe[0]->temperature
        ());
        //     pc.printf("Temperature sensor 2 returns %3.5f oC\r\n", probe[1]->temperature
        ());

```

```

95 //
96 //   measurement[6] = probe[0]->giveRam1(); // MSB first
97 //   measurement[7] = probe[0]->giveRam0();
98 //   measurement[8] = probe[1]->giveRam1(); // MSB first
99 //   measurement[9] = probe[1]->giveRam0();
100 //
101 //
102 //   pc.printf("Temperature in hexadecimals numbers of sensor 1: %.2X%.2X\n",
103 //   measurement[6],measurement[7]);
104 //   pc.printf("Temperature in hexadecimals numbers of sensor 2: %.2X%.2X\n",
105 //   measurement[8],measurement[9]);
106
107 size_t string_length= strlen(measurement);
108 int err;
109 pin1 = 1; // both pins high to activate driver and disable reciever
110 pin2 = 1;
111 while(true) {
112     if (micro.writeable()) {
113         err=micro.putc(string_length);
114         pc.printf("i 'm writeable\n");//Outgoing signal
115         break;
116     }
117 }
118
119 pc.printf("Response sent:\r");
120 for (int i=0; i<string_length; i++) {
121     while (true) {
122         if (micro.writeable()) {
123             err= micro.putc(measurement[i]); // Sending data per character
124             pc.printf("%.2X", err);
125             break;
126         }
127     }
128 }
129 pc.printf("\r\n\n");
130 }

```

## A.5. Cost analysis

A cost analysis was made for the final concept, the prices are based on a production of 10000 SubDowsers. As can be seen a total cost of €9.30 for the control subsystem is achieved. The rest of the budget €3.20 will be used for wiring the total system.

What	Quantity	Price	Total cost
LPC1768 chip	1	€4.45	€4.45
LPC1114 chip	1	€1.45	€1.45
Rs-485 driver chips	2	€1.70	€3.40
Total			€9.30

Table A.1: Total cost of control subsystem

# Bibliography

- [1] e. a. Albert Mumma, *Kenya groundwater governance case study*, Tech. Rep. (Water Unit, Transport, Water and ICT Department, 2011).
- [2] *SIM5320 AT Command Set*, SIMCom (2014), v2.02.
- [3] *Industry's First Temperature-Compensated RTC*, Maxim Integrated Product (2015), dS3231M.
- [4] V. Beal, *flash memory*, (), [http://www.webopedia.com/TERM/F/flash\\_memory.html](http://www.webopedia.com/TERM/F/flash_memory.html).
- [5] V. Beal, *Ram memory*, (), <http://www.webopedia.com/TERM/R/RAM.html>.
- [6] V. Beal, *Ssd*, (), [http://www.webopedia.com/TERM/S/solid\\_state\\_disk.html](http://www.webopedia.com/TERM/S/solid_state_disk.html).
- [7] V. Beal, *Hard disk*, (), [http://www.webopedia.com/TERM/H/hard\\_disk.html](http://www.webopedia.com/TERM/H/hard_disk.html).
- [8] J. Axelson, *Serial Port Complete COM Ports, USB Virtual COM Ports, and Ports for Embedded Systems*, second edition ed. (Lakeview Research LLC Madison, WI 53704, 2007) pp. 43–68 and 79–130, ISBN 978-1931448-07-9.
- [9] *Using RS-485/RS-422 Transceivers in Fieldbus Networks*, Maxim Integrated Products (2010).
- [10] F. Leens, *An introduction to i2c and spi protocols*, IEEE Instrumentation & Measurement (2009), 1094-6969.
- [11] *I2C-bus specification and user manual*, NXP Semiconductors N.V. (2014).
- [12] *Introduction to the Controller Area Network (CAN)*, Texas Instruments (2008).
- [13] M. A. Jaime Lloret, Sandra Sendra and J. J. P. C. Rodrigues, *Underwater wireless sensor communications in the 2.4 ghz ism frequency band*, Sensors (Basel) (2012), 4237–4264.
- [14] *SNx5HVD1x 3.3-V RS-485 Transceivers*, Texas Instruments (2014), rev. 3.1.
- [15] *32-bit ARM Cortex-M3 microcontroller*, NXP Semiconductors N.V. (2015), IPC1768.
- [16] F. B. Brokken, *C++ Annotations version 9.0.0*, 9th ed., ISBN 90 367 0470 7.
- [17] B. Eckel, *Thinking in C++*, second edition ed. (MindView, 2013).
- [18] Mbed, *mbed.h*, (2015), [https://developer.mbed.org/users/mbed\\_official/code/mbed-src/file/42ba2fb9673a/api/mbed.h/](https://developer.mbed.org/users/mbed_official/code/mbed-src/file/42ba2fb9673a/api/mbed.h/).
- [19] *LPC 1768 User Manual*, NXP Semiconductors N.V. (2014), rev. 3.1.
- [20] E. Olieman, *Rtc.h*, (2012), <https://developer.mbed.org/users/Sissors/code/RTC/>.
- [21] M. Wei, *Powercontrol.h*, (2010), <https://developer.mbed.org/users/no2chem/code/PowerControl/>.