Instituto Superior de Engenharia do Porto
Departamento de Engenharia Electrotécnica
Rua Dr. António Bernardino de Almeida, 431, P-4200-072 Porto

Environmental/Regatta Buoy Sensing System

Erasmus Project/Internship Thesis
MSc. in Electrical Engineering and Computers

Mathias Van Flieberge

Supervisors:
Mrs Benedita Malheiro
Mr Manuel Silva
Mr Paulo Ferreira
Mr Pedro Guedes

Academic Year: 2013-2014

Resumo

Um dos principais objetivos da ciência é perceber a natureza, i.e., descobrir e explicar o funcionamento do mundo que nos rodeia. Para tal, os cientistas precisam de coligir dados e monitorar o meio ambiente. Em particular, considerando que cerca de 70% da Terra é coberta por água, a coleta de parâmetros de caracterização da água de grandes superfícies é uma prioridade.

A monitorização das condições da água é feita principalmente através de bóias. No entanto, as bóias disponíveis no mercado não satisfazem as necessidades existentes. Esta é uma das principais razões que levaram o Laboratório de Sistemas Autónomos (LSA) do Instituto Superior de Engenharia do Porto a lançarem um projeto para o desenvolvimento de uma bóia reconfigurável e com dois modos de funcionamento: monitorização ambiental e baliza ativa de regata. O segundo modo é destinado a regatas de veleiros autónomos.

O projeto começou há um ano com um projeto do European Project Project [1] (EPS), realizado por quatro estudantes internacionais, destinado à construção da estrutura da bóia e à seleção dos componentes mais adequados para o sistema de medição e controlo. A arquitetura que foi definida para este sistema é do tipo mestre-escravo e é composta por uma unidade de controlo mestre para a telemetria e configuração e uma unidade de controlo escrava para a medição e armazenamento de dados. O desenvolvimento do projeto continuou com dois estudantes belgas que trabalharam na comunicação e no armazenamento de dados.

Este projeto, que prossegue com o desenvolvimento da medição e do armazenamento de dados do lado da unidade de controlo escrava, tem os seguintes objetivos: (*i*) implementar o protocolo de comunicação na unidade de controlo escrava; (*ii*) coligir e armazenar os dados dos sensores no cartão SD em tempo real; (*iii*) fornecer dados em tempo útil; e (*iv*) recuperar dados do cartão SD em tempo diferido.

As contribuições anteriores foram estudadas e foi feito um levantamento dos projetos congéneres existentes. O desenvolvimento do projeto atual começou com o protocolo de comunicação. Este protocolo, que foi projetado pelos alunos an-

teriores, foi um bom ponto de partida. No entanto, o protocolo foi atualizado e melhorado com novas funcionalidades. Esta última componente foi um trabalho conjunto com Laurens Allart, que esteve a trabalhar no subsistema de telemetria e de configuração durante este semestre. O protocolo foi implementado do lado da unidade de controlo escrava através de uma estrutura de múltiplas atividades paralelas (multithreaded). Esta estrutura recebe as mensagens da unidade mestre, executa as ações solicitadas e envia de volta o resultado.

A bóia é um dispositivo reconfigurável multimodo que pode ser expandido com novos modos de operação no futuro. Infelizmente, sofre de algumas limitações: suporta uma carga máxima de 40 kg e tem uma área de implantação limitada pela distância máxima à estação base.

Abstract

One of the main targets of science is to understand how the world functions, i.e., scientists want to discover how and explain why things occur. In order to do so, they need to collect data and monitor the environment. Considering that around 70% of Earth is covered by water, the collection of water-related information is a priority.

The monitoring the water conditions is mostly done by using buoys. However, the buoys available on the market do not fulfil existing needs. This is one of the main reasons why the Autonomous Systems Lab (LSA) and the Instituto Superior de Engenharia do Porto launched a project for the development of a reconfigurable dual mode buoy for environmental monitoring as well as regatta beacon. The second mode is intended for autonomous sailing boat regattas.

This project started one year ago, as an European Project Semester [Jor14] (EPS) project involving four international students. At this stage, the objectives were to design and construct a buoy structure and select the most suitable components for the sensing and control system. The architecture defined for the sensing and control system was a master-slave with two microprocessors: a master control unit for telemetry and configuration and a slave control unit for sensing and data logging. The project development continued with two other Belgian students that worked on the communication and data logging.

This current project continues the development of the sensing and data logging subsystem on the slave control unit side. The current goal is to: (*i*) implement the communication protocol on the slave control unit; (*ii*) collect and store sensor data in the SD card in real time; (*iii*) provide data in near real; and retrieve data from the SD card in deferred time.

The previous contributions were studied and a survey on related projects was done. The current project development started with the communication protocol. This protocol, that was designed by the previous students, was a good starting point. But, in order to get a good implementation and some new functions, the protocol was updated. This component was a joint work with Laurens Allart,

who worked on the telemetry and configuration subsystem during the current semester. The protocol was implemented on the slave control unit side using a multithreaded structure. This structure receives the messages from the master, executes the requested actions and sends back the results.

The buoy is a reconfigurable multi-mode device that can be further expanded with new operation modes and functions in the future. Unfortunately, it suffers from the following limitations: the payload cannot exceed 40 kg and its deployment area is limited to maximum distance allowed to the base station.

Abstract

Een van de grootste doelstelling van de wetenschap is begrijpen hoe de wereld functioneert. Dit wil zeggen dat wetenschappers willen ontdekken hoe en waarom iets gebeurt. Om dit te kunnen verwezenlijken, moeten ze de omgeving waarnemen en bijhorende data verzamelen. Wanneer men acht neemt van het feit dat de oppervlakte van de Aarde voor meer dan 70% uit water bestaat, is het vergaren van water gerelateerde data in prioriteit.

Het waarnemen van het water wordt veelal uitgevoerd door gebruik te maken van boeien. Toch is de boei markt nog niet verzadigd, er zijn nog steeds bepaalde noden die niet vervuld worden. Dit was een van de belangrijkste redenen waarom het Autonomous Systems Lab en het Instituto Superior de Engenharia do Porto een project gestart hebben waarin een herconfigureerbare dual-mode boei wordt ontwikkeld.

Het project begon vorig jaar, als een European Project Semester project. Binnen dit project was het doel om een boei structuur te ontwerpen en de juiste componenten te verzamelen voor het controle en data opname systeem. Dit controle en data opname systeem werd ontworpen als een master-slave structuur opgebouwd uit 2 microprocessoren. De master controle unit verzorgt de telemetrie en de configuratie, terwijl de slave controle unit instaat voor de data opname en het bewaren van de opgenomen gegevens. Het project werd nadien verder gezet door 2 andere studenten die werkten aan de communicatie en het loggen van de data.

Dit project zet de ontwikkeling verder van het data-opname subsysteem. Het doel op dit moment is: (*i*) implementeer het communicatie protocol in de slave controle unit; (*ii*) verzamel en bewaar sensor data op een SD kaart in real time; (*iii*) verstrek data van op dit ogenblik (broadcast) en haal data, van in het verleden, op van de SD kaart.

De vorige bijdragen aan het project werden bestudeerd en een studie van gerelateerde projecten werd uitgevoerd. De huidige ontwikkelingen in het project startten met het communicatie protocol. Dit protocol, waarvan het ontwerp

gemaakt werd door de vorige studenten, was een goed vertrekpunt. Om een goede implementatie te krijgen in de twee microprocessoren moesten er echter enkele wijzigingen doorgevoerd worden. Deze taak werd uitgevoerd in samenwerking met Laurens Allart, die werkte aan het telemetrie en configuratie subsysteem gedurende dit semester. Het protocol werd geïmplementeerd op de slave controle unit door gebruik te maken van een structuur opgebouwd uit meerdere threads. Deze structuur ontvangt berichten van de master, voert de gevraagde acties uit en stuurt de resultaten terug naar de master.

De boei is een herconfigureerbaar multi-mode toestel dat kan uitgebreid worden met nieuwe operatie modes en extra functies in de toekomst. Jammer genoeg zijn er ook enkele beperkingen: de massa van de boei, samen met alle geïmplementeerde elektronica, is beperkt tot 40 kg. Ook is het toepassingsgebied van de boei beperkt tot de maximale afstand die wordt toegelaten door het base station.

Contents

List of Figures

List of Tables

Glossary

Abbreviation	Description
μGFX	micro Graphics
μIP	micro Internet Protocol
ADC	Analog-Digital Convertor
API	Application Programming Interface
ARM	Acorn RISC Machine
ASCII	American Standard Code for Information Interchange
BBB	BeagleBone Black
BBB/Pi	BeagleBone Black/Raspberry Pi
BS	Base Station
CAN	Controller Area Network
CPU	Central Processing Unit
CRC-32	32 bit Cyclic Redundancy Check
CSRC	C Source
CTD	Conductivity, Temperature and Depth sensor
DGPS	Differential Global Positioning System
DMA	Direct Memory Acces
EGNOS	European Geostationary Navigation Overlay Service
EOH	End of Header
EPS	European Project Semester
EXT	Extended file system
FatFs	File allocation table File system
FIFO	First In, First Out
FPGA	Field-Programmable Gate Array
GNSS	Global Navigation Satellite System
GPIO	General Purpose Input/Output
GPRS	General Packet Radio Service
GPS	Global Positioning System
GPT	Globally unique identifier Partition Table
HAL	Hardware Abstraction Layer

Abbreviation	Description
HS	High-Speed
I2C	Inter-Integrated Circuit
ICU	Instruction Cache Unit
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IRQ	Interrupt Request
ISEP	Instituto Superior de Engenharia do Porto
ISP	Image Signal Processing
KaHo	Katholieke Hogeschool
KU Leuven	Katholieke Universiteit Leuven
LQFP100	Low Profile Quad Flat Package with 100 pins
LSA	Laboratório de Sistemas Autónomos / Autonomous Systems Laboratory
lwIP	ligthweigth IP
MAC	Media Access Control
MCX	Micro Coaxial
MEMS	MicroElectroMechanical Sensor
MMC	Multi-Media Card
NMEA	National Marine Electronics Association Standard for Interfacing
OPC	Optical Particle Counter
OpenOCD	Open On-Chip Debugger
OS	Operating System
PC	Personal Computer
PCB	Printed Circuit Board
PT	Platinum RTD
PT100	Platinum RTD that has a resistance of 100Ω at 0°C
PVT	Position, Velocity and Time
PWM	Pulse-Width Modulation
RAM	Random Acces Memory
RF	Radio Frequency
RISC	Reduced Instruction Set Computer
RS232	Recommended Standard 232
RTC	Real-Time Clock
RTD	Resistance Temperature Detector
RTOS	Real Time Operating System
SBAS	Satellite Based Augmentation System
SBDS	Short Burst Data Service
SDC	SD-card
SD	Secure Digital
SOH	Start of Header
SPI	Serial Peripheral Interface
STM32F3	STM32F3DISCOVERY
SWD	Serial Wire Debug Interface

Abbreviation	Description
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
USB-CDC	USB Communication Device Class
UV	Ultraviolet
WA	Wide Area
WET	Western Environmental Technologies
ZDA	Date and Time used on NMEA

Agradecimentos/Acknowledgements

I would like to take this opportunity to thank all the people who where involved in this project. Firstly I am very thankful for receiving the opportunity, by KU Leuven, KaHo Sint-Lieven and the Instituto Superior de Engenharia do Porto (ISEP), to fulfil my thesis at the Laboratório de Sistemas Autónomos (LSA) of ISEP. This project/internship was a great experience. I have learned a lot and had a great time.

A special thank should be offered to Mr De Strycker Lieven. He introduced me to Mrs. Malheiro Benedita and ISEP. I also would like to thank Mrs. Roelandts and Mr. Naessens, they helped me at KaHo Sint-Lieven to arrange all the necessary paperwork. The next person I would like to thank is Ms. Gudrun Vandenabeele of the International Admissions and Mobility Unit at KU Leuven. This for the help to get the necessary documents to the university during my stay in Portugal.

I am also immensely grateful to Mrs. Benedita Malheiro, Mr. Paulo Ferreira, Mr. Manuel Silva and Mr. Pedro Barbosa Guedes. Thank you for holding the weekly meetings and the advices. Without these meetings and their knowledge, it would not be possible to have realized what I have now. Next to this useful technical information, they also shared some great information about the Portugese traditions and pointed me at some great places in Porto and Lisbon.

A particular thank should go to Mrs. Benedita Malheiro. She guided me from the start of my Erasmus exchange until the end, and it would not be such a great experience without her drive and efforts in this project.

Additionally I would like to thank the people at LSA who helped me. Especially Mr. Pedro Rocha to help me with some problems, provide me the com-

ponents/tools I needed and share his knowledge. At last, I would like to thank the people of the International office at ISEP to help me with the neccesary documents.

Chapter 1

Introduction

1.1 Presentation

My name is Mathias Van Flieberge, I am in the final year of the Master of Science in Electronics Engineering from KU Leuven Campus Gent in Belgium. I arrived at the Instituto Superior de Engenharia do Porto (ISEP) in February 2014 to make my Master thesis under the Erasmus programme. This thesis was a major project that lasted a full semester. As an addition to that project, I have followed two courses: " Control of Autonomous Systems" and " Advanced Vision Topics for Robotics".

1.2 Motivation

One of the main reasons why I chose the Instituto Superior de Engenharia do Porto to do my Erasmus project is that it has a very good connection with KaHo Sint-Lieven/KU Leuven. The projects that were available focussed on robotics and had some maritime influences. This made these thesis assignments very attractive and coherent with my own interests.

I have chosen for the " Environmental/Regatta Buoy: Sensing System" project because it was an already running project. This gave the opportunity to do real implementations instead of theoretical research. The project mainly consisted of programming and developping software. This was a big challenge for me, because I am an Electronics Engineer, not a programmer. However, I like to have a challenge and to learn new things.

Another thing that made me choose Portugal as a destination is the culture of the country. I wanted to learn about the way of living and working in another environment than Belgium.

1.3 General Information

A bouy can be described as a floating device, that can have multiple purposes. When you divide them on structural difference, you can distinguish two types: a stationary buoy and a drifting buoy. A stationary boy is anchored to the ground (or another solid structure), it will not change its location. A drifting buoy has no fixed connection, it will drift with the currents. The buoys can be found on the ocean, at sea, in a river or a lake.

Another way to distinguish different buoy types is based on their function. A buoy can have many purposes. Most of the time they are used as markers, or to collect data. Depending on the function, the size of the buoy will differ. Markers are often much smaller than the data collecting buoys, this because of the fact that they dont́ have to carry electronic equipment. The more equipment a buoy has to carry, the bigger it is. Also the location where it will be used has its influence on the size. The buoys that will be used in the open ocean are bigger than the ones used on a river or in shallow water. One of the most known examples of a buoy are the ones that can be found on the beach. They are used to indicate that the depth of the water is limited. These buoys are from the stationary type. An example of this type of buoy is depicted in Figure 1.1.

Figure 1.1: Buoy used on the beach [Pau14]

1.4 Project Description

This project is a continuation of the work of six previous students. The first four students started this project in order to fulfil their European Project Semester. This group designed and constructed the autonomous buoy, depicted in Figure 1.2 on the next page. It consists of a hull and a stainless steel structure. This structure was made to attach the sensors, antenna and a blinkling lamp to the buoy. One aspect about the buoy that stands out is its size. It is relatively small in comparison with other measuring buoys. An example of a measuring

buoy is shown in Figure 1.3. Another aspect about this buoy is that it is rather complex because it is intended to carry a number of electronic devices. All of the necessary equipment to realise this autonomous buoy is present at the Laboratório de Sistemas Autónomos (LSA).

Figure 1.2: LSA buoy

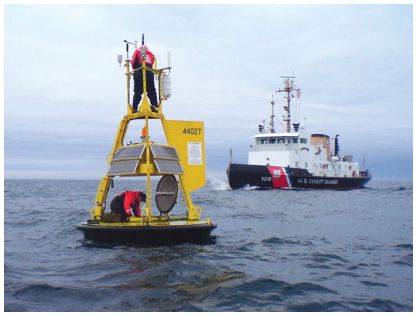


Figure 1.3: Example of a measuring buoy [Dat14]

After the EPS group, the project was divided in two main blocks: a data logging part and a telemetry and configuration part. These two elements were handled by two other students. They worked on this project the previous semester. In the data logging part a lot of progress was made on the storage of the data on the SD-card, and the connection of one of the sensors. The telemetry part of the project focussed on the communication between a base station and the buoy and a lot of research was done on this. Also some tests were ran with the microprocessors that are used.

This means that there were still some objectives remaining, mainly assembling

and programming. On the telemetry side, the communication between the base station and the buoy should be implemented.

This paper handles about the sensing system side of the project, which will be described in detail in section 1.5.2 below.

1.5 Objectives

1.5.1 General Objective

The general objective of the autonomous buoy is to monitor its environment. This should be done by measuring, storing and sending data from its location at sea, near the shore, or in a river. The buoy is intended to have two main functions in the future:

1. Assist an autonomous sailing boat that will be constructed at LSA in the future
2. Collect environmental data

The assistance of the boat in a regatta requires that the buoy is able to perform the following two tasks:

- make sure that the sailing boats travel in the correct direction by informing them about the buoy's own location
- provide judges in the races with the necessary weather data to increase each boat's performance

The collecting function for the buoy is supposed to make measurements of the selected weather conditions at a chosen site and for a specified period of time. Although this final goal will occur at sea, the buoy is for the moment limited to work at sea near the shore or in a river because of the absence of a long distance communication unit.

1.5.2 Objectives within the Sensing System Part

A sensing system can be described as a series of sensors that monitors certain parameters in their surroundings. In this case, the parameters are:

- Wind: speed and direction
- Water: conductivity, temperature and depth
- Location of the buoy: Longitudenal and lattitude coordinates, velocity, heading, etc.

The objective in the sensing sytem part is to develop a system that is able to read the sensors, that is reconfigurable (sensors can be added and deleted via software), that communicates with the master, that is able to store data and that sends the data to the master/base station.

The buoy, and so the sensing system, has two main operation modes that must be implemented:

- An environmental data logging mode
- A regatta mode

When the system is running in environmental mode, it should be able to receive requests/commands and transmit data via the Universal Serial Bus (USB) interfaces. The requests can be data retrieval related, or can be about the configuration of the buoy. This configuration, or reconfiguration, command can be include/exclude sensors, change the operation mode, etc.

The regatta mode differs from the environmental mode. In this mode, the buoy must act as a beacon and broadcast data via the wireless interface. It has to send Position, Velocity and Time (PVT), wind and water related data. This to help the autonomous boats located in the vicinity. This mode can run on its own, but another option is that it runs at the same time as the environmental mode.

1.6 Requirements

Since the project uses a master-slave architecture, it uses two separate microprocessors. One is used for the telemetry and configuration part. And another one is used for the Sensing System.

The target of the telemetry and configuration part is to run on a BeagleBone Black [Bea14]. These were unavailable at the start of this semester. Due to this a Raspberry Pi [Ras14] was used to develop the communication software on the master. This communication part asks the slave for specific data or gives configuration commands. These commands can ask for data from a sensor from a longer period, this can be something like this: "Give me all the temperature measurements from the month May."

The slave takes care of the data logging and sensing part. To do so, a STM32F3DISCOVERY [STM14] (STM32F3) development board is used. This microprocessor board will communicate with the sensors, will coordinate the data storage, etc. The functions of this board will be discussed in the Project Development section (section 3.4).

Chapter 2

State of the Art

2.1 Introduction

Scientists want to have a look on what happens around the globe. In order to do that, they have to collect data at sea. This is because of the fact that the globe is 70% water, as can be seen in Figure 2.1. This can be done by using manned ships that sail the ocean and collect data. Another option is to use an autonomous system that is able to send the measured data to the coast.

Figure 2.1: Earth [Vol14]

During the last 10 to 15 years, there was a huge amount of research on this topic. Several autonomous buoys and autonomous measuring boats were developed and a lot of research was done on these items. Especially, the autonomous environmental buoys are interesting in this case. What can be found on the market at this moment? And what is still under construction? What is the scoop used in the development? Which sensors-types are available? All these questions will be answered in this chapter.

2.2 Purpose of the Buoy

2.2.1 Introduction

The ISEP approach of an autonomous buoy has a master-slave structure and wants two implemented operation modes. These operation modes have their benefits, but what are these benefits? What function will the implemented sensing system have? Will it differ in the different operation modes?

2.2.2 Operation Modes

The eventual goal of the buoy project can be divided in two main targets: an environmental part and the regatta part, as depicted in Figure 2.2.

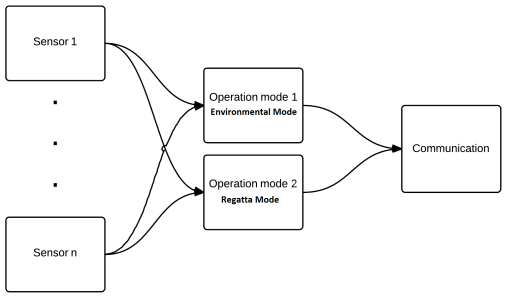


Figure 2.2: Operation modes

- **Environmental** The environmental operation of the buoy is focussed on collecting data from the surroundings of the buoy. This data, captured by the sensors, is logged to make sure it is available for a longer time.

  The captured data can be location information, water or wind features. These options must be available in the first buoy model, but there are possibilities to add more sensors in a later stage. These measuring components can for example capture video, radiation, etc.

- **Regatta Mode** The regatta mode is an operational mode that was requested by LSA, because their research focusses on autonomous platforms. One of those projects handles an autonomous sailing boat, that has as a goal to participate in the World Robotic Sailing Championship. The regatta mode will have an important task in such sailing races. When a regatta, a series of boat races, is raced, the buoy will work as a beacon and provide the sailing boats information. The final target is to define a race course on the water with a number of these buoys. Figure 2.3 on the next page depicts a sailing regatta.

Figure 2.3: Sailing regatta [Sar01]

The main differences between these two modes are the implemented sensors and the data sending rate. The environmental mode will typically use more sensors than the Regatta mode. But in Regatta mode the data sending rate must be much higher then in environmental mode. During a regatta, the buoy will send data at a rate of some seconds, this will continue for a few hours (the duration of the race). While in environmental mode, the data will be captured for a longer period (days, weeks) but data will only be sent when it is requested. Because this difference in operation, it can be possible to run both modes at the same time.

2.3 Related Projects

In the development of an autonomous buoy, there are different studies and opinions under construction. In this section we will take a brief look at some of these developments at other institutions besides ISEP. All the projects present some similar characteristics, but there is no exact copy or equal project.

2.3.1 Aerial Wireless Sensor Network for Oceanographic Monitoring

The work presented in [Cri10] describes the available techniques for wireless sensor Networks.

Wireless Sensor Networks give a new opportunity to oceanography. They are autonomous, self-organized Wireless Personal Area Networks composed of multiple (up to 1000 and more) sensor nodes. Every Sensor network usually consists of a processor, a radio module, a power supply and one or more sensors mounted on a floating device. The processor controls the sensors and the data collection. An improvement to this idea is adding some remote base stations on land. These are reached by adding some longer-range connections to some floating stations. This can be done via satellite, General Packet Radio Service (GPRS), etc. This network is different from the network that is used to connect all the floating nodes. A general description scheme of a sensor node can be found in Figure 2.4 on the following page.

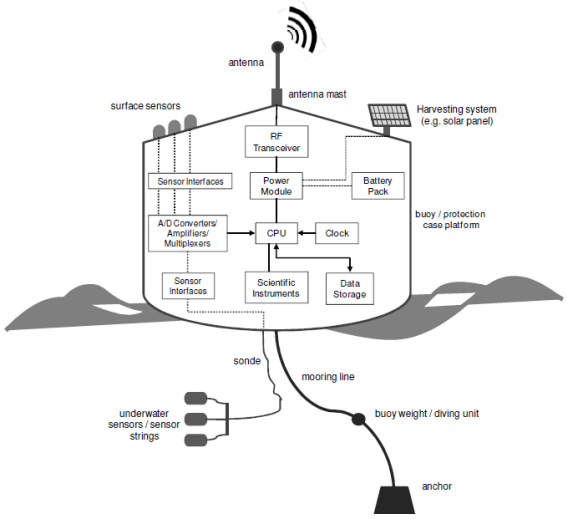


Figure 2.4: Scheme of sensor node [Cri10]

Between the different approaches to the environmental buoy, there are also several differences in the communication section. Some of them use Wi-Fi, others use WiMAX or Bluetooth. On the sensor level there are also differences. The number of sensors varies from buoy to buoy, from goal to goal. There are buoys that test on chemical components, while others test the wind velocity and direction, Global Positioning System (GPS) location, etc. The most common sensors can be found in Table 2.1 on the next page.

Table 2.1: Popular sensors

Measured Parameter	***Unit***
Surface sensors	
Temperature	°C, °F
Pressure	mmHg, Pa
Wind velocity	m/s
Wind direction	°
Relative humidity	%RH
In water sensors	
Water temperature	°C, °F
Salinity/Conductivity	g/l
Water velocity	m/s
Turbidity	mg/l SiO2
Chlorophyll	μg/l
Dissolved oxygen	mg/l
Nitrate	mg/l
pH	/
Swell	Height: m, Direction: °
Other chemical components	mg/l
Hydrocarbons	ppm
Precipitation	mm

2.3.2 Embedded System for an Autonomous Buoy

In this buoy project [Kyu11], they composed an embedded system that works with an Optical Particle Counter (OPC) and a Conductivity, Temperature and Depth (CTD) sensor. The equipment is used to observe the water and the underwater environment. It is based on a floating vehicle, called an autonomous profiling float. This is a vehicle that is able to move horizontally by using a buoyancy engine while it moves up and down due to the waves. This project focusses at the marine ecosystem. There are possibilities to add other observation equipment, which are more specific to the underwater environment. They can add different camera?s to check the fauna and flora under the sea level. The shape of the buoy itself is a cylindrical hull, that can be seen in Figure 2.5 on the following page.

The buoy will move as the water moves. During this movement it will collect data from the OPC, CTD and flow sensors. The collected data is processed by a Central Processing Unit (CPU). The data management structure is based on a field-programmable gate array (FPGA) module that sends data on a First In,

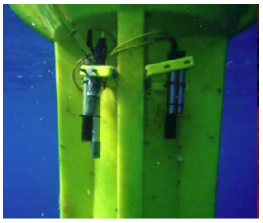
Figure 2.5: Cilinndrical autonomous buoy [Kyu11]

First Out (FIFO) order to the CPU. After some correction and Image Signal Processing (ISP) the data can be transmitted. This is done when the buoy is back at the sea level. GPS position information is used and the data is sent via ORBCOMM satellite communication. In general this buoy works with images to watch the environment and collect data. Afterwards this is sent to a receiver by use of a satellite connection/transmission.

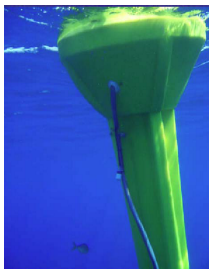
2.3.3 Hidroboya

Hidroboya [Xul11] is project that already runs/works at sea. The purpose is to provide a database system on an "on land" server with ocean and continental water data. The main goal of the data collecting is checking the water quality. In order to collect this data, the buoy is fitted with a water filtering system and various sensor systems. The main concern in this project was to make it durable. They have found a solution for the sensor pollution when they are in the seawater during a long time. The idea of the Hidroboya developers was to bring the sensors to the inside of the buoy. By achieving this, the sensors are in a dry environment when they aren't in use. This means less pollution. A backside of this approach was that they had to collect water in the buoy. This was achieved by using a pump that can collect water at multiple depths. In Figure 2.6 on the next page there is an image of the difference between a normal buoy and the Hidroboya approach.

In general this buoy measures some components that are also present in the ISEP approach of an autonomous buoy. However, the two projects focus on a different aspect. Here the main goal is durability.



(a) Classical Buoy



(b) Hidroboya

Figure 2.6: Sensor arrangement [Xul11]

2.3.4 Autonomous River Drifting Buoys

This project [Lis14] uses a similar formed buoy, but with smaller dimensions, and the goal of this project is river based. This brings some extra conditions, as there are changing current velocities, changing water level, presence of vegetation, etc. The QinetiQ North America Technology Solutions group has developed a buoy that can work under these influences. It is called a Lagrangian river drifter and can be seen in Figure 2.7.

Figure 2.7: Lagrangian river drifter [Lis14]

The equipment/sensors used by this project are a GPS receiver, satellite and radio frequency (RF) communication system, temperature transducer and depth sounder, accelerometer and a Gumstix single-board computer running a Linux Operating System (OS). It is powered by a battery pack that provides power for 24 hours.

2.3.5 Autonomous Buoy System in the Western English Channel

The buoy in this project [T. 10] gathers information by measuring a range of bio-geochemical and physical parameters that are transmitted every hour to two long-term monitoring sites on shore. The biggest problem in this project was the English Channel. It is a very wild area that suffers from large waves, winds, storms, etc. An image of the Channel Buoy can be found in Figure 2.8 on the next page.

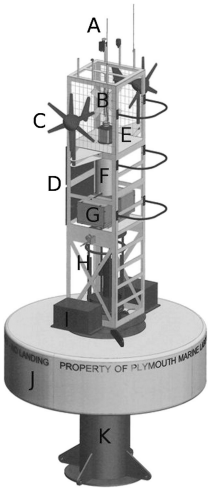


Figure 2.8: Design of Western English Channel Buoy [T. 10]

The data is collected by using different instruments. There is a Western Environmental Technologies (WET) Labs-Sea-Bird Water Quality Monitor (measures temperature, presence of some chemical components). There is also a camera installed.

All data that is collected is processed by an on-board personal computer (PC). There is also a Satlantic Stor-X. This is a data acquisition and logging system that controls the scheduling and data management for the sensors. This board has digital and analog ports that can be used for expansion. The communication type that is used in this project is a radio frequency transmission.

2.3.6 Autonomous Meteorological Buoy

This project [Car09] aims to develop an autonomous buoy that can operate in the Mediterranean Sea. The acquired data will be sent to a base station on land. The sensor network is based on the 1451.4 standard of the Institute of Electrical and Electronics Engineers (IEEE). This means that all sensors are Plug and Play. The implemented sensors are used for measuring wind, air temperature, air pressure, water temperature, pressure and salinity. The number of sensors can be adjusted too. A graphical representation of the buoy can be found in Figure 2.9 on the following page.

All the collected data is processed and stored by use of a Reneses 16 bit microcontroller. The iridium Short Burst Data Service (SBDS) is used to communicate with the base station.

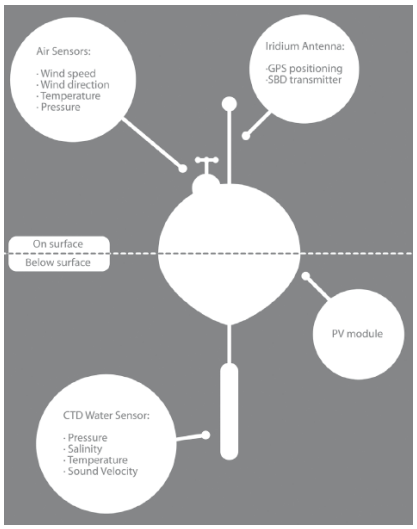


Figure 2.9: Autonomous meteorological buoy [Car09]

2.4 ISEP Buoy

When the different buoy's or sensor systems are compared, similarities can be found. Every buoy uses wind sensors and sensors to measure depth, temperature and location. Some have equipment to measure chemical parameters or take pictures and capture video. These last sensors aren't present, at this time, in the ISEP program but may be implemented in later versions of the buoy.

The ISEP approach has two key differences with all other buoy projects:

1. The data processing and managing is done by two processors that work as a master and a slave. The slave collects and saves the data and makes it available to the master. The master is the communication link to land and controls the receiving and sending of the messages. In most other projects there is just one CPU that coordinates the communication and the data collection. In order to control the data acquisition they often use a FPGA or a similar programmable component.

2. The regatta mode. There are two possible working modes in the ISEP approach. It can work as a environmental buoy, but there is a possibility to use the buoy in a sailing regatta too. This makes it possible to send location parameters and actual CTD or wind parameters to sailing boats in the surrounding area. This isn't implemented in the other buoys.

2.5 Available Equipment

The Regatta and environmental buoy project was already running when this part of the project started. Because of this, a lot of equipment was already present in the lab. In what follows, a short description is given about all these parts.

2.5.1 Physical Structure

The non-electrical parts that are present in the lab are mainly the buoy structure. This consists of:

- Hull
- Steel Structure
- Anchoring system

2.5.1.1 Hull

This was the starting point of the first group on this project. The hull, displayed in Figure 2.10, was available at LSA, but not the supporting structure.

Figure 2.10: Buoy hull

The main characteristics of the hull are:

- Build with fibreglass → lightweight
- Empty inside → access via top (unscrew some bolts first)
- Waterproof → electrical components must be inside
- Extra waterproof box inside the hull for electrical components
- Rubber washers for screws and rubber skirt on lit to make waterproof connections

2.5.1.2 Steel Structure

The steel structure was a development of European Project Semester (EPS) students that were at ISEP last year. The structure is needed to attach the sensors. The form of the structure was chosen to make the buoy taller, so that it is easier to find it. On top of the structure a blinking lamp will be attached, this to make it visible over a larger distance. The structure had some requirements due to the sea/water environment in which the buoy will perform:

- Materials must bee resistant to ultraviolet (UV) radiation and salt water effect
- Must be big enough to attach all sensors
- Must be wind and wave resistant

To achieve this requirements, the EPS students made a stainless steel structure. The drawings of this structure can be found in Figure 2.11.

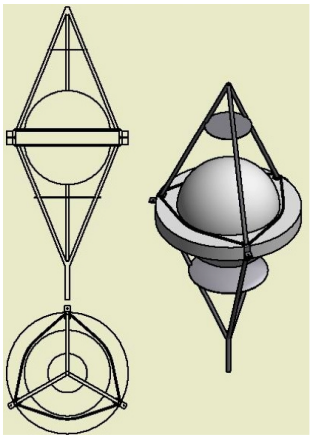


Figure 2.11: Stainless steel structure [Ben13]

Figure 2.12 on the next page shows a picture of the buoy hull and support structure.

2.5.2 Electronic Components

There are also some electrical parts available at the LSA lab. These are mainly the sensors and the microprocessor boards. Another part that is present is the Secure Digital (SD) card for the data capture.

Figure 2.12: Buoy

2.5.2.1 Microprocessor Boards

The project works with a master-slave hierarchy. The previous EPS students had chosen two microprocessor boards. A BeagleBone Black that will work as the master and a STM32F3DISCOVERY that works as the slave. Because there was a problem with the availability of the BeagleBone Black (BBB) [Bea14], a Raspberry Pi is used to replace it.

- **Raspberry Pi** The Raspberry Pi [Ras14] (Figure 2.13) is a single board computer that uses Acorn RISC Machine (ARM)-Processors. It was developed at the Cambridge University and had educational goals. As a result of this educational goal, the production costs are reduced as much as possible, to get a cheaper end product.

Figure 2.13: Raspberry Pi [Wik14d]

  It is possible to run multiple operating systems on the Raspberry Pi. It can run Android, some linux distributions, Raspbian, etc. The Rasberry Pi used in this project runs on Raspbian [Ras12].

- **STM32F3 DISCOVERY** There are two STM32F3's available in the lab. A standard board with no extra extensions and one with an extra printed circuit board (PCB) to attach sensors and the SD card (a picture of this board can be found in Figure 2.14 on the next page).

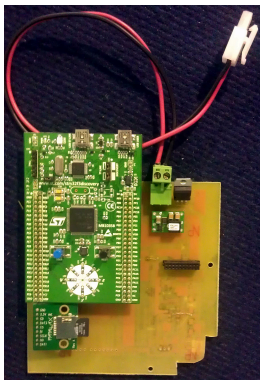


Figure 2.14: STM32F303-DISCOVERY with extra PCB

2.5.2.2 Sensors

The available sensors are:

- CTD package
- SUPERSTAR II Global Navigation Satellite System (GNSS) Sensor
- Davis anemometer

More information about these sensors can be found in section 2.6.

2.5.2.3 SD Card

The available SD card is the following:

- Kingston(R) 8 GB microSDHC class 10

2.6 Sensors

In order to monitor the environment, the buoy needs devices to check certain parameters. After checking the desired parameter, the device has to communicate its state or value to the system. The tools that can do this are called sensors. A sensor can be described as: "a converter that measures a physical quantity and converts it into a signal which can be read by an observer or by an (today mostly electronic) instrument." [Wik14c]

If the concept of the sensor is connected with the target object of the buoy project, it is easy to find which sensors will be used. There are sensors that will measure parameters of the water: conductivity, temperature, depth, etc. Other sensors will check physical quantities outside the water. Examples for this type

are wind velocity, wind direction, location, humidity, air pressure, etc. Not all of the suggested parameters are checked in the buoy at the start of the project, but they can be added later.

The sensors that will be added at the start of the project will be discussed now.

2.6.1 Types of Sensors

The already bought/available sensors can be found in Table 2.2.

Table 2.2: Sensors for the buoy

Name	***Type of sensor***
GNSS receiver	GPS: position and time
Wind sensor	Wind velocity & direction
CTD sensor	Conductivity, Temperature, Depth

2.6.1.1 GNSS Receiver

A Global Navigation Satellite System (GNSS) sensor is a stand alone card. The sensor that is available for this project is a SUPERSTAR II [Nov09]. Figure 2.15 shows an image of the SUPERSTAR II.

Figure 2.15: SUPERSTAR II GNSS [Nov09]

The SUPERSTAR II is a Global Positioning System (GPS) receiver for embedded applications that offers good quality. It provides signal tracking capability even under weak signal conditions. SUPERSTAR II uses 12 channels to track in-view satellites. This is done on a full autonomous way, from the moment when power is applied. There is also a Satellite Based Augmentation System (SBAS). This offers wide-area or regional augmentation. By using additional base systems, additional satellite-broadcast messages are created. Examples where SBAS

is implemented are Wide Area (WA) Augmentation System and the European Geostationary Navigation Overlay Service (EGNOS).

- **Main Features** The main features of the SUPERSTAR II are:
  - 12 channel correlator for all-in-view satellite tracking
  - Input voltage of 3.3 V
  - Single chip RF front end
  - SBAS support
  - Active, and passive, antenna support
  - Complete L1 GPS receiver and navigator on a single compact board
  - Operating temperature range of -30°C to +75°C

  A block diagram of the implemented system on the SUPERSTAR II board can be found in Figure 2.16.

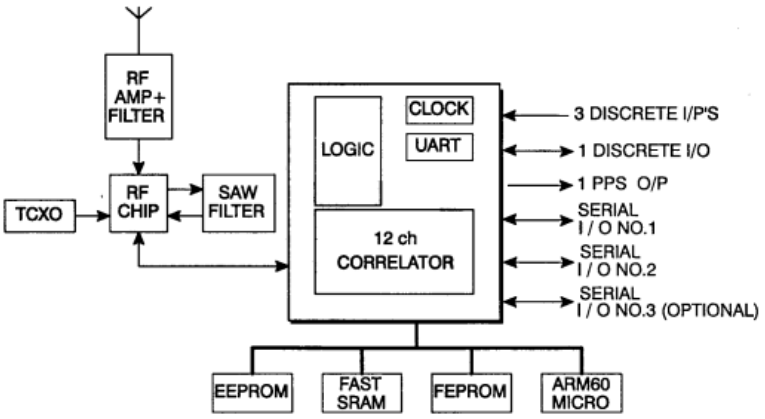


Figure 2.16: SUPERSTAR II GNSS scheme [Nov09]

- **Connectors and Connector Pins Assignment** The receiver has two standard type connectors.
  - J1: 20-pin connector for general input/output interfaces and power input
  - J2: Micro coaxial (MCX) type RF connector.

  The minimum number of connections on J1 required for the system to operate is shown in Table 2.3 on the next page.

Table 2.3: Minimum J1 Connections for SUPERSTAR II

Signal Name	***J1 Pin #***	***Description***
VCC	2	Primary power (3.3 V -0.5 V/+0.25 V)
GND	10, 13, 16 & 18	Ground
TX_No_1	11	Serial port Tx #1
RX_No_1	12	Serial port Rx #1

If Differential Global Positioning System (DGPS) corrections are required for the application, the SUPERSTAR II can receive them on COM2:

Signal Name	**J1 Pin #**	**Description**
RX_No_2	15	Serial port Rx #2

If an active antenna is used:

Signal Name	**J1 Pin #**	**Description**
PREAMP	1	Power for active antenna (50 mA max)

The GNSS sensor can use different communication modes: National Electronics Association Standard for Interfacing (NMEA) of Binary.

- **NMEA:** a standard defined by the National Marine Electronics Association for interfacing navigation-related equipment. The version used by the SUPERSTAR II is the third version. This one is also known as NMEA 0183. At this moment there is also a newer version available, NMEA 2000. NMEA 0183 Interface Standard defines signal requirements, data transmission protocol and time, and specific sentence formats for a 4800 baud serial data bus. The goal of this standard is to support a transmission from one sender to multiple receivers. The data transmitted by the SUPERSTAR II that follows NMEA 0183 uses a printable American Standard Code for Information Interchange (ASCII) form. It may include the following information: position, velocity, frequency allocation, etc. There is also an addendum made to NMEA 0183. In V4.10 a high-speed (HS) option is added. This version is called NMEA 0183-HS V1.01. This has a different baudrate. It works at 38.4 kbaud [Nov09].
  The package used by the NMEA 0183 has always the same form:
  - Starts with ”$”
  - Five characters: first two identify sender (GP OR GPS) and others define the type of message

  - Data, separated by ",". The field remains blank when the data is not available.
  - When a checksum is used, a "*" must be used next
  - Two digit checksum
  - "<CR><LF>" ends the message

  More information on the ID and the format of the data can be found in the SUPERSTAR II firmware reference manual [Nov05].

- **Binary:** The binary protocol used in the SUPERSTAR II is typical for this sensor. It is a series of bytes that is sent by/to the sensor. Every byte in the message has a specific meaning. This meaning can vary on the type of the message. There are also a lot of different message lengths, as a result of the need to specify more parameters when defining different modes/messages. More information on this binary protocol can be found in the SUPERSTAR II firmware reference manual [Nov05].

2.6.1.2 Wind Sensor

The wind is measured by using an anemometer, depicted in Figure 2.17. This anemometer uses a propeller and a vane to measure the wind velocity and the wind direction. These parts are not sufficient to get the correct data in the STM32F3. There must be a conversion made to electrical signals. The wind velocity propeller generates a pulse per revolution. The number of pulses per time unit makes it possible to determine the wind velocity, see Table 2.4 on the next page. To get an electrical signal that represents the direction, a precision potentiometer is used. When the wind changes direction, the resistance of the potentiometer also changes.

Figure 2.17: Wind sensor [Ben13]

The vane itself isn't capable of giving the exact wind direction. This is due to the fact that the buoy can turn itself, and change direction. To get an accurate direction, a compass is used in addition to the wind sensor. The specifications of the anemometer can be found in Table 2.4 on the following page. Figure 2.18 on the next page shows some pictures of the Davis anemometer that can be found in the LSA lab.

(a) Full (b) Propellor (c) Vane

Figure 2.18: Davis anemometer (LSA) [Ben13]

Table 2.4: Specifications Wind Sensor

Product name	***Davis anemometer***
Dimensions	470 mm x 191 mm x 121 mm
Weight	1.332 kg
Connector	Modular connectoir (RJ-11)
Cable length	12 m
Cable type	4-conductor 26AWG

Measurement of	**Wind velocity**	**Wind direction**
Sensor type	Solid state magnetic sensor	Wind vane and potentiometer
Accuracy	+/- 3 km/h	+/-7°
Resolution	0.1 m/s	1° (0° to 355°)
Range	1 to 322 km/h	0° to 360°
Sample period	2.25 s	1 s
Output	1600 rev/h = 1 mph V=P(2,55/T) V: velocity in mph P: # of pulses per sample period T: sample period in seconds	Variable resistance 0 - 20 kΩ 10 kΩ = south = 180°

2.6.1.3 CTD Package

The CTD package [Ben13] includes Conductivity, Temperature and Depth sensors and was developed by LSA. An image of the CTD sensor can be found in Figure 2.19 on the following page.

The CTD developed by LSA has the following specifications (see Table 2.5 on the next page).

Figure 2.19: CTD Sensor (LSA) [Ben13]

Table 2.5: Specifications of the CTD Sensor

Product name	***CTD***
Size	366 x ⌀95 mm
Power Supply	8-30 V
Power Consumption	1.5 W/12 V
Communication	RS232 or I2C

Quantity	**Conductivity**	**Temperature**	**Pressure**
Sensor	7 electrodes	PT100	Load cell
Unit	mS/cm	°C	dbar
Range	0-70 mS/cm	-5°C to 35°C	0-100 dbar
Resolution	+/- 0.01 mS/cm	+/- 0.001 °C	0.2% full scale

- **Conductivity Measurement** The sensor measures the conductivity by using four electrodes. These electrodes work in pairs of two. An alternating current is applied between the outer pair of electrodes, and the conductivity is determined by measuring the potential between the inner pair. With this potential, the conductivity can be found using the distance between the electrodes and their surface. With these values and Ohm's law the conductivity can be determined.
  Ohm's law (with resistance R, potential difference U and current I):
  $U = I \cdot R \leftrightarrow R = \frac{U}{I}$
  together with the relation between conductivity (ρ) and resistance:
  $R = \frac{l}{A} \cdot \rho$
  where l is the distance between the electrodes and A is the surface of those elctrodes, leads to:

$$\rho = \frac{A}{l} \cdot \frac{U}{I}$$

To get a better accuracy, a calibration is done by using electrolytes of well known conductivity, see Table 2.6.

Table 2.6: Conductivity of some solutions

Solution	***Conductivity***
Absolute pure water	0.055 μS/cm
Good city water	50 μS/cm
Ocean water	63 mS/cm

- **Temperature Measurement**
  The temperature is measured with a Resistance Temperature Detector (RTD). The RTD usid in the CTD to measure the temperature is a Platinum RTD (PT). This PT100 has a resistance of 100 Ω at 0°C. It also has a predictable change in resistance due to temperature changes. This change is used to determine the temperature. The RTD has a higher accuracy and repeatability than thermocouples. There are two sorts:
  - Wire wound elements: they consist of some platinum wires coiled around a ceramic or glass core. The core can make them fragile and susceptible to vibration. To protect the cores, a probe sheath is used.
  - Thin film elements: these are manufactured using processes that are developed for integrated circuits. The platinum film is deposited on a ceramic substrate that is encapsulated. Due to this method, the production of small, fast and accurate sensors is possible.
- **Pressure Measurement**
  Pressure is measured by the use of a load sensor. A load cell is a transducer that is used to convert force into an electrical signal. In this case, the conversion is indirect. This means that it is done in two stages. At first, a mechanical arrangement is used. The force that is applied deforms a strain gauge. This strain gauge is able to measure the deformation as an electrical signal, since the strain changes the effective electrical resistance of the wire.

  A normal load cell usually consists of four strain gauges in a Wheatstone bridge configuration. In some cases, a quarter bridge (one strain gauge) or a half bridge (two strain gauges) are available. The signal generated by these load cells is usually in the order of a few mV and requires amplification by an instrumentation amplifier before it can be used. The output from the

load cell gives a measure for the pressure. This can be scaled to calculate the force that was applied.

2.7 Conclusion

Although there are already several autonomous buoys projects running, the ISEP/LSA approach manages to bring something different. This is mainly the result of two big aspects:

- Master-slave architecture
- Different operation modes

To be able to run both operation modes the development will be started with an anemometer, a CTD and a GNSS sensor. With these the basic task should be executed. In a later stage extra sensors can be added.

Chapter 3

Project Development

3.1 Introduction

The architecture of the total project is depicted in Figure 3.1. This shows all main components of the working buoy and the base station and regatta boats.

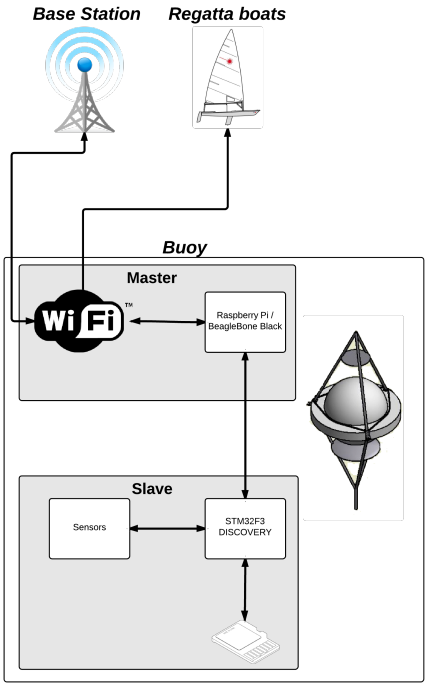


Figure 3.1: Architecture of buoy project

The complete buoy project from the STM32F3's view can be separated in

four main elements:

- The STM32F3DISCOVERY
- The communication with the BeagleBone Black / Raspberry Pi (BBB/Pi)
- The attached sensors
- The SD card.

These elements are shown in Figure 3.2.

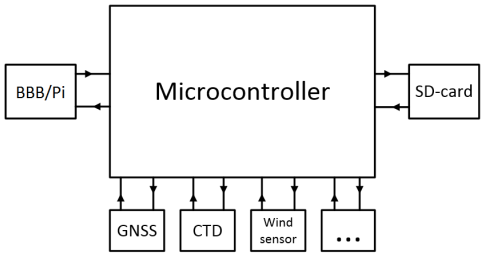


Figure 3.2: Project block diagram [Jer14]

This report focusses on the communication between the BBB/Pi and the STM32F3. In order to understand why the STM32F3 is used, a brief explanation is given. The next section will give an overview of the total project architecture.

3.2 Architecture of the Sensing System

The intended architecture for the STM32F3 can be found in Figure 3.3 on the next page. This diagram shows the main building blocks that must be implemented.

The STM32F3 contains seven main blocks at this moment. The number of blocks must be expanded when new sensors are added. The minimal structure that has to be implemented is the send, receive, executor and one sensor.

The first block is the receiving block. This structure must coordinate the reception of the command messages. It has to read the incoming datastream and detect new messages that are coming in. When a message has arrived, it should be passed to the next block. This block is the executor block. When the message is passed, the Receive structure restarts monitoring the incoming stream for new messages.

The executor block reads the new message and acts according to the content of the message. The executor will contact sensors or the SD-card, will start a

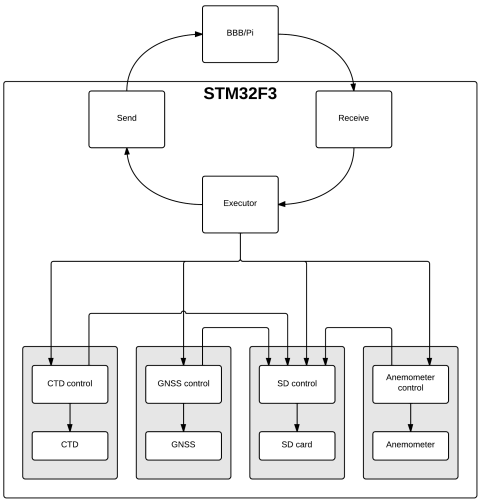


Figure 3.3: STM32F3

mode, etc. When a the message contains a request for specific data, the executor must get that data and transfer it to the send block.

The send block takes care of the transaction of messages to the BBB/Pi. It receives messages from the executor and must transfer them to the BBB/Pi. This must be done in a structured way. To do so, a protocol is used.

The SD-card block is a very important part of the STM32F3 architecture. It is used for the data logging and it stores all previously captured data. The SD-card has its own block that is used to write and read the SD-card. When data must be written to the SD-card, the SD block must be provided with the correct data.

Another important block is the GNSS. The GNSS is the only block that has time awareness. Therefore it will be used for each datapackege that is captured. This because all captured data has to have a timestamp.

The anemometer and CTD structures are two examples of implemented sensors. These two will be implemented at the start of the project and are available in LSA. The sensors are controlled by a structure that is able to send the correct data requests and that is able to read the replies of the sensors.

3.3 Components

Because this project was already runnig for two semesters, a lot of components were available at the LSA laboratory. All available equipment is listed in Table 3.1. The changes in comparison with last year are shown in Bold.

Table 3.1: Components

Part	Product	Price [€]	Quantity	Source
Fibreglass hull	Fibreglass hull	/	1	ALTO
Steel structure	Stainless steel structure	600-700	1	Metal workshop
Anchor	Danforth anchor	40.70	1	Dismotor
Rope	Nylon rope 50 m x 12 mm	28.23	1	Dismotor
Chain	Stainless steel chain 3.5 m x 6 mm	43.19	1	Dismotor
Eye bolt	Eye bolt INOX 12 x 120 mm	11.65	1	Dismotor
Thimble	Thimble INOX 12 mm	1.45	1	Dismotor
Swivel v 1.0	Swivel INOX 10 mm	24.00	1	Dismotor
Swivel v 2.0	Swivel v 2.0	28.00	1	Dismotor
Rope clamp	Rope clamp INOX 10 mm	2.60	2	Dismotor
Shackle	Shackle INOX 10 mm	3.20	1	Dismotor
Wind speed and Direction sensor	Davis Anemometer	100.00	1	LSA
CTD sensor	Developed by a professor at ISEP	/	1	LSA
Battery	Lithium-polymer battery 12 V 22 Ah (option 1)	270.00	1	Deben.com
	NiMH battery 6 V 4,8 Ah (option 2)	35.00	?	Overlander.co.uk
Microcontroller	STM32F3 Discovery	13.50	1	LSA
Data storage	Flash memory card, microSD	25.00	1	www.sandisk.com
	Parallax microSD card adapter	13.00	1	LSA
Communication Module	RN-V WiFly Module	35.00	1	LSA
GNSS module	Novatel Superstar II receiver	/	1	LSA
	Novatel GPS-701-GG antenna	/	1	www.novatel.com
Data storage	2 x Kingston microSD cards + adapters	13.80	1	www.kingston.com
External Hard-drive	WD elements 1 TB		1	www.wdc.com
Second Control Unit	**RASPBERRY PI**		**1**	**LSA**
Communication Module	SR71 USB/ Wi-Fi stick		1	Ubiquitni network

Table 3.2 on the next page shows some more data about some of the electrical components that will be added to the buoy. The dimensions and weight of the components is also added to this table because the total weight is limited due to the buoy size.

3.4 STM32F3DISCOVERY

The previous groups that were involved in this work had chosen the STM32F3DISCOVERY [STM13] to work as a slave and save the data to the SD card.The choice to use this specific board to develop the slave was made based on the following requirements:

- Several interfaces and General Purpose Input/Output (GPIO)
- At least one Serial Peripheral Interface (SPI) for the SD card
- At least one Universal Asynchronous Receiver/Transmitter (UART) connection for the GNSS sensor
- One Universal Serial Bus (USB) connection (to connect to the other part)
- Includes an e-compass on it, useful to define the wind direction

Table 3.2: Electrical components

Component	Voltage [V]	Current [mA]	Power Consumption [W]	Dimensions: length-width-height [mm]	Mass [g]
NiMH battery 4.8 Ah	6	-	-	100-50-10	360
Davis Anemometer	3.3	0.197	0.0007	470-121-191	523
STM32F3 Discovery	5	13.536	0.068	96-65-20	irrelevant
Flash memory card, microSD	3.3	50	0.165	irrelevant	irrelevant
Parallax microSD card adapter	3.3	0.5	0.002	29-26-12	irrelevant
RN-XV WiFly Module	3.3	38	0.125	34-24-10	irrelevant
Novatel Superstar II GNSS receiver	3.3	151.515	0.5	70-45-10	irrelevant
Novatel GPS-701-GG GNSS antenna	5	35	0.175	185-185-69	500
CTD sensor	12	125	1.5	365-100-100	1937
Blinking LED lamp	12	250	3		200

- LSA has the board and has usage experience

The programming work is made easier by using an operating system. Most drivers are already predefined and the OS can handle simultaneous processes. The operating system that is used in this project is ChibiOS. More about ChibiOS can be found in section 3.4.2.

3.4.1 Information

The STM32F3DISCOVERY [STM13], depicted in Figure 3.4 on the following page, is a microcontroller that has 256 kB on-chip flash memory and 48 kB Random Acces Memory (RAM) in a Low Profile Quad Flat Package (LQFP100) with 100 pins. It is based on the STM32F303VCT6, and it offers an ST-Link/V2 embedded debug tool, accelerometer, gyroscope and e-compass, ST MicroElectroMechanical Sensors(MEMS), USB connection LED and pushbuttons. There is also a large number of free applications and firmware examples. All these features provide a good support and quick evaluation.

Features

- STM32F303VCT6 microcontroller featuring 256 kB Flash, 48 kB RAM in an LQFP100 package

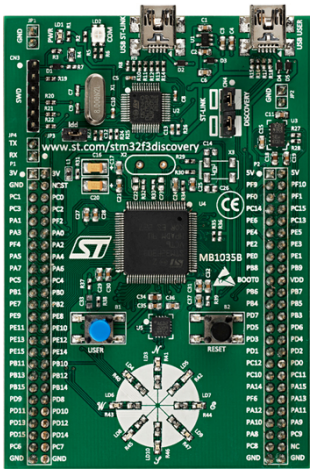


Figure 3.4: STM32F3DISCOVERY board [STM13]

- On-board ST-LINK/V2 with selection mode switch to use the kit as a standalone ST-LINK/V2 (with a Serial Wire Debug (SWD) Interface connector for programming and debugging)
- Board power supply: through USB bus or from an external 3 V or 5 V supply voltage
- External application power supply: 3 V and 5 V
- L3GD20, ST MEMS motion sensor, 3-axis digital output gyroscope
- LSM303DLHC, ST MEMS system-in-package featuring a 3D digital linear acceleration sensor and a 3D digital magnetic sensor
- Ten LED:
  - LD1 (red) for 3.3 V power on
  - LD2 (red/green) for USB communication
  - Eight user LED, LD3/10 (red), LD4/9 (blue), LD5/8 (orange) and LD6/7 (green)
- Two pushbuttons (user and reset)
- USB USER with Mini-B USB-connector
- Extension header for all LQFP100 I/Os for quick connection to prototyping board and easy probing

3.4.2 ChibiOS

3.4.2.1 Introduction

ChibiOS/RT [Chi14a] is a real time operating system (RTOS), and it is designed for deeply embedded real time applications who require efficient execution and a compact code. To fullfill these requirements the ChibiOS/RT RTOS is characterized by its high portability, compact size and its architecture optimized for extremely efficient context switching. Some features presented by the developers are:

- Efficient and portable preemptive kernel.
- Best in class context switch performance.
- Many supported architectures and platforms.
- Static architecture, everything is statically allocated at compile time.
- Dynamic extensions, dynamic objects are supported by an optional layer built on top of the static core.
- Rich set of primitives: threads, virtual timers, semaphores, mutexes, condition variables, messages, mailboxes, event flags, queues.
- Support for priority inheritance algorithm on mutexes.
- Hardware Abstraction Layer (HAL) component supporting a variety of abstract device drivers: Port, Serial, Analog-Digital Convertor (ADC), Controller Area Network (CAN), extended file system (EXT), Globally unique identifie Partition Table (GPT), Inter-Integrated Circuit(I2C), Instruction Cache Unit (ICU), Media Access Control (MAC), Multi-Media Card (MMC), Pulse-Width Modulation (PWM), Real-Time Clock (RTC), SD Card (SDC), SPI, UART, USB, USB Communication Device Class (USB-CDC).
- Support for external components micro Internet Protocol (μIP), ligthweigth IP (lwIP), File allocation table file system (FatFs), micro graphics (μGFX).
- Extensive test suite with benchmarks.

The version of ChibiOS used during this project was version 2.6.4. There will be a newer release somewhere in Q2/2014, the next maintenance release in the 2.6.x branch, ChibiOS v2.6.5

3.4.2.2 Architecture

The general architecture of ChibiOS is shown in Figure 3.5 on the next page. The parts represented in the figure are:

- Application: the high level application code that is abstracted from hardware details.
- HAL: high level, cross platfrom, drivers Application Programming Interface (API) layer. Holds definition of component drivers.
- Kernel: porable RTOS kernel layer. The drivers for the OS structures(threads, ...) are located here.
- Platform: low level, platform-specific drivers layer.
- Port: kernel port layer for a specific architecture.
- Board: the board-specific description and initialisation code.
- Hardware: the hardware platform.

  When the development of an application starts, a minimum of files has to be present in the development working space. These files (see Figure 3.6 on the following page) are:

- ”main.c”: the main application that will be implemented.
- ”chconf.h”: the kernel configuration file
- ”halconf.h”: this file determines the implemented functions by including or excluding device drivers.
- ”mcuconf.h”: This file has several functions. Some of these functions are:
- Setting up the periphals parameters
- Interrupt request (IRQ) priority
- Direct Memory Acces (DMA) priority
- Change hardware related settings
- Build file/folder
- ”.dep” folder: this folder contains the ”.o” object files that were made during the compilation of the project.

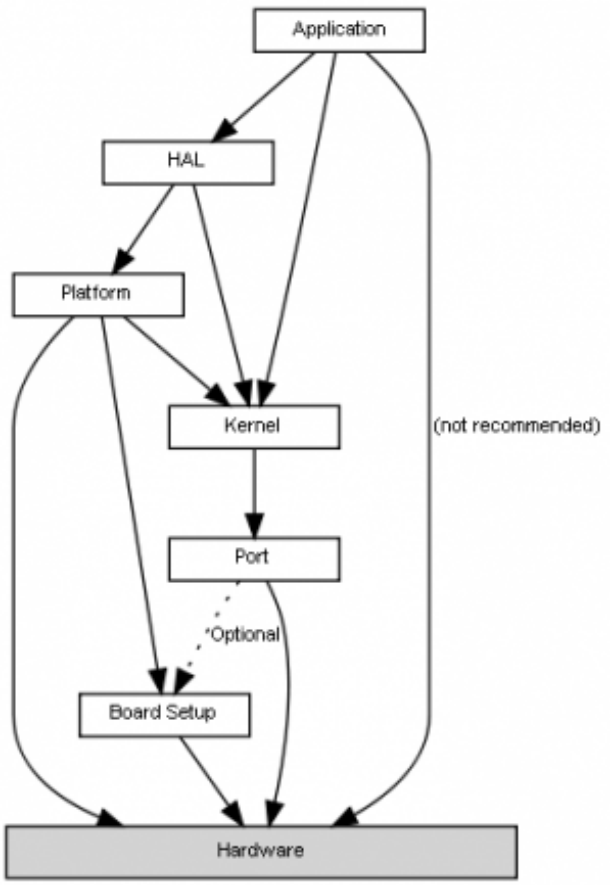


Figure 3.5: ChibiOS architecture [Mar11]

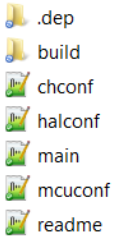


Figure 3.6: Minimal files that are needed to create ChibiOS project

3.4.2.3 Threads

Threads are a really important part of the development of this project, this because a lot of simultanious actions must happen in the STM32F3. Those actions are, for example, the data requests from all sensors at the same time, or save data from a sensor to the SD card. Each structure, or possible simultanious action,

(sensor, SD card, etc.) used in the STM32F3 will be monitored by a thread. Luckily there is a possibility to implement threads in ChibiOS [Chi11]. When a thread has to be implemented, a working area must be created first for this thread. This can be done via the macro ”static WORKING_AREA(waThread, 128);” where 128 is the wanted size of the working area. Now a thread can be created with: ”chThdCreateStatic(void wsp, size t size, tprio t prio, tfunc t pf, void arg);”. The parameters used in this function refer to:

- ”wsp”: pointer to a working area that is dedicated to the thread stack
- ”size”: represents the size of the chosen working area
- ”prio”: this defines the priority level that is selected for the new thread
- ”arg”: this can be either an argument that is passed to the thread function or NULL

When the thread is created, multiple actions can be done. This will force the thread in a defined states. A state diagram for a thread can be found in Figure 3.7

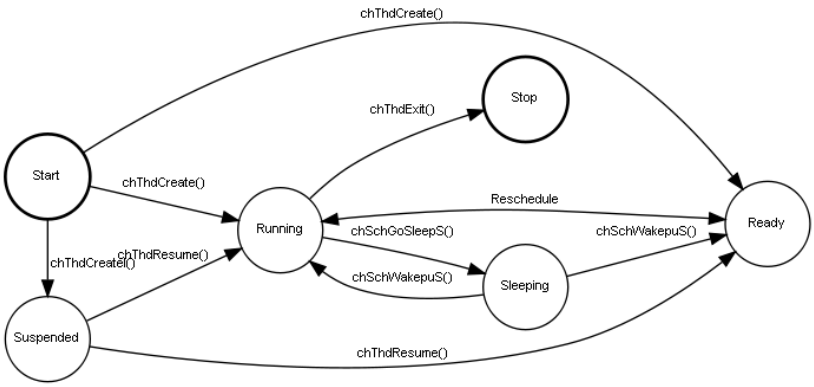


Figure 3.7: Thread State Diagram [Chi14a]

3.5 Programming Environment

In order to develop software for the STM32F3, a software environment is needed. The predecessors in this project used several Integrated Development Environments (IDE). The EPS group used the IAR Embedded Workbench and afterwards the Eclipse IDE was used. Both these solutions offered some problems in the beginning of this project. Due to this, there was chosen to use no IDE, everything is done on a basic/manual way.

In order to do so, some components are needed:

- text editor
- compiler
- Program to flash the STM32F3

3.5.1 Development in Linux

The development of the code was done in a Linux environment. The distribution used was peppermint Four [Pep13], shown in Figure 3.8. This is an OS based on Ubuntu 13.04. It offers some advantages:

- Built for speed
- Ligthweight

Figure 3.8: Peppermint Four [Pep13]

These advantages are very usefull when the OS is ran virtually, what was done in this case. The main OS on the used computer is Windows 8.1, and peppermint Four ran in VMwarePlayer [vmw14], a screenshot of the starting screen can be found in Figure 3.9.

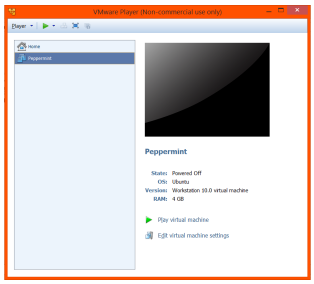


Figure 3.9: VMware Player [vmw14]

3.5.1.1 Guidelines

To edit a file from a project the following steps should be followed:

- Navigate to the correct folder (with the terminal, or with the mouse)
- Open the file that must be edited. During development gedit was used as the editor for the code. This is the standard text editor in peppermint. To open a file with gedit, you can click it, or do "gedit filename" in a terminal.
- When the editting is finished, the file must be saved.
- Go to a terminal, navigate to the correct folder (for example "/home/user /ChibiOS_2.6.3/demos/ARMCM4-STM32F303-DISCOVERY/")
- Now a make must be done. In order to do so, arm-non-eabi-gcc must be installed. A good tutorial to do this can be found in [GNU14].
- When this toolchain is installed, do a "make". The files will now compile. When this is finished without mistakes, you can proceed to the next step. If the compilation did not finish, there were errors. These problems should be solved first to finish the compilation. After editting the errors, another make should be done. Repeat this proces until completion.
- During this make, a new folder, "build", will be created (or replaced when it already existed).
- The last step to get the software loaded in the STM32F3 is flashing its memory. To do so, use this command in the terminal (the fastest way): "st-flash write build/ch.bin 0x80000000". This command can only be executed when the st-link drivers [Git14] are installed on the used machine.

3.6 Application Protocol

3.6.1 Principle

In order to get good communication between two entities a protocol is needed. A protocol can be explained as: "In computer science, a set of rules or procedures for transmitting data between electronic devices, such as computers. In order for computers to exchange information, there must be a preexisting agreement as to how the information will be structured and how each side will send and receive it. This agreement is called a protocol" [Enc14].

This general explanation can be transferred to the master/slave structure of this project. There are multiple protocols used for the data exchange between the different components. When the base station sends a request for an action

or specific data from the SD card, it will send a message to the BBB/Pi. This message uses a specified format or protocol, and is first handled by the BBB/Pi. At this point the priority of the message will be checked and when there are no higher priority commands available, the command will be transferred to the STM32F3.

From the STM32F3 point of view:

The data to be transmitted is requested to the BeagleBone Black (or Raspberry Pi) and not to the Discovery. The base station does interact directly with the STM32F3, *i.e.*, messages are handled only by the master (the BBB/Pi) control unit. The data exchange between the STM32F3 and BBB/Pi is coordinated by a dedicated application protocol. When the base station sends a request to the buoy, the BBB/Pi will perform the actions required to provide the data. These actions are different depending on the working mode of the buoy and on the on-going actions. This means that an on-going data download can be interrupted by a higher priority command. This implicates that the data transfer can be stopped and restarted after a period of time.

3.6.1.1 Protocol Messages

A first approach to a binary protocol was designed by our predecessors. They defined a basic form for each message to be exchanged between the BBB/Pi and the STM32F3. The defined command package is described in Figure 3.10:

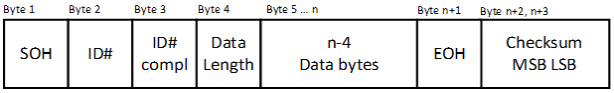


Figure 3.10: Protocol command package [Jer14]

The data message structure is similar, but has some difference – Figure 3.11. A date and time field (ZDA), based on NMEA, is added and a chunk of data is added after the date. Each piece of data is preceded by the ID of the sensor. At the end of the message a carriage return and line feed is sent.

Figure 3.11: Protocol data message [Jer14]

More information about the defined fields in the standard protocol message can be found in Table 3.3 on the following page:

Table 3.3: Byte function

Byte	***Name/abbreviation***	***Function***
1	SOH	This is an indicator that shows the start of a package. In this case the '$' sign is used. The '$' is transmitted as a hexadecimal 24.
2	ID#	This is a number that represents the commands and errors. These commands and errors can be found in Table 3.4 on the next page
3	ID# Complement	The 1st complement of the ID number. This is used as a verification of the ID
4	Data Length	This number indicates how long the package will be. It can vary from 0 to n.
5 to n - 3	Data Bytes	This is the actual data that is sent
n - 2	EOH	This is an indicator for the end of the data. In this case, the '*' is used as End Of Header (EOH).
n-1 to n	Checksum	This is a 16 b sum (2 B) of the entire package.

The most important field in the message is the ID field. This ID defines which function or command is called by the base station or the BBB/Pi. There are already several ID/command defined. Table 3.4 on the following page gives a summary of those ID.

Table 3.4: Protocol ID

ID Function 1st digit	***ID 2nd digit***	***Function***
1: Mode selection	0	Regatta Mode
	4	Environmental
2: GPS	3	Coordinates
3: Wind	1	Wind velocity
	6	Wind direction
4: Water/CTD	1	Water depth
	5	Water conductivity
	9	Water temperature
6: Operation	2	Combination
8: Errors	2	SDC not connected
	4	GNSS not connected
	6	Wrong command
	8	Wrong data
9: Data OK	9	Data OK

3.6.1.2 Protocol Diagrams

In order to detail the protocol, additional protocol diagrams are presented. They provide the order of the messages between the BS, BBB/Pi and the STM32F3.

The diagrams have to be read from top to bottom. The arrows indicate a message between two devices. When you start from the first arrow/message, you will get a second arrow that is a result of the first one. This means that it will show how the different devices communicate.

- **Data Request** The protocol diagram, shown in Figure 3.12 on the next page, represents a data request from the BS to the BBB/Pi, and the BBB/Pi will at its turn make a request for data to the STM32F3.

  Some more information regarding each arrow/step in the protocol diagram about the Data Command, shown in Figure 3.12 on the following page:

  1. BS sends *Data Command* to BBB/Pi.
  2. BBB/Pi receives the command and replies with a *MessageOK* to the BS. The BBB/Pi also transfers the command to the STM32F3.
  3. An error occured during the sending action of the *Data Command*. The BBB/Pi did not receive a MessageOK as a confirmation of the request. As a reaction to this, the BBB/Pi resends the *Data Command*.

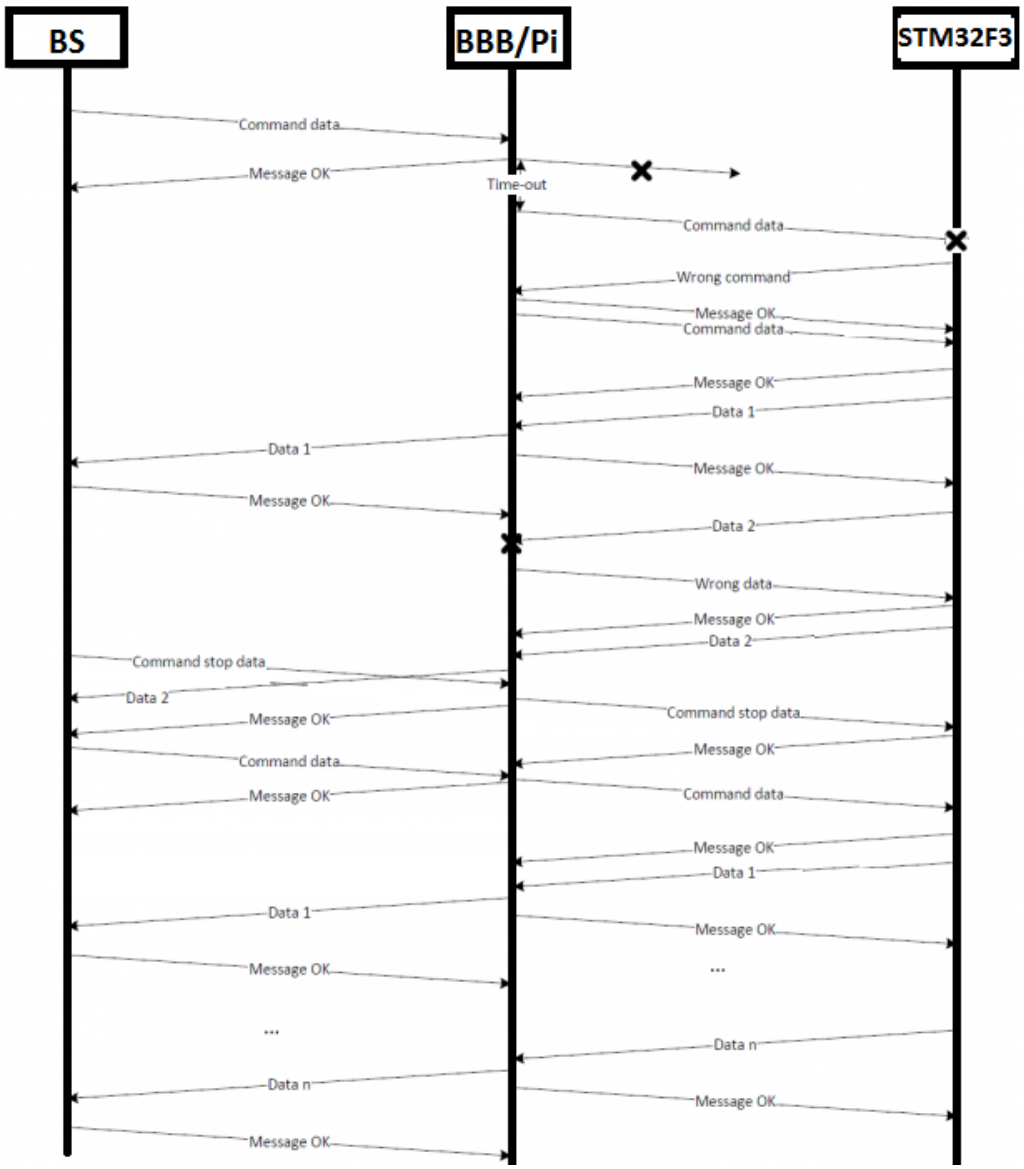


Figure 3.12: Protocol diagram: data command [Jer14]

4. The STM32F3 now receives a command, but there is something wrong with it (for example a checksum that is incorrect). The incoming command is not known by the STM32F3, therefore a *Wrong Command* message is sent to the BBB/Pi.
5. BBB/Pi receives the *Wrong Command*, replies with a *MessageOK* and resends the *Data Command* for the second time.
6. The STM32F3 receives the command without mistakes and replies to the BBB/Pi with a *MessageOK*.

7. After reception of the *MessageOK*, the BBB/Pi knows that the STM32F3 received the message without mistakes.
8. After the MessageOK is sent, the STM32F3 will start transferring the data that was requested (*Data1*).
9. BBB/Pi receives the data (*Data1*) and transfers it to the BS. It also sends a *MessageOK* to the STM32F3 to show that the data transfer did succeed.
10. BS sends a *MessageOK* to the BBB/Pi after reception of the errorless data package.
11. STM32F3 send the next data package (*Data2*).
12. BBB/Pi receives an incorrect datapackage, therefore it sends a *Wrong Data* command to the STM32F3.
13. After receiving the *Wrong Data*, the STM32F3 replies with a *MessageOK* and it will resend the datapackage(*Data2*).
14. BBB/Pi receives an errorless datapackage, and transfers it to the BS.
15. Before receiving *Data 2*, the BS has sent a *Stop Command*for the data. This because it has to send a new Data Command to the BBB/Pi.
16. After receiving this *Stop Command*, the BBB/Pi replies with a *MessageOK* to the BS. The BBB/Pi also transfers the *Stop Command* to the STM32F3.
17. When the BS receives the *MessageOK*, it can send the new *Data Command*.
18. The STM32F3 sends a *MessageOK* as a result of the incoming *Stop Command*.
19. The BBB/Pi receives the *MessageOK* from the STM32F3.
20. The BBB/Pi receives the new *Data Command* from the BS and sends the *Data Command* to the STM32F3.
21. The STM32F3 reacts with a *MessageOK*.
22. The STM32F3 now sends the new asked data (*Data1*) to the BBB/Pi.
23. After reception, the BBB/Pi sends a *MessageOK* to the STM32F3 and transfers the datapackage to the BS.
24. The last communication two steps/actions keeps repeating untill all the datapackages are sent by the STM32F3.

- **Command Mode** This protocol diagram, presented in Figure 3.13 on the next page, represents a command message that has been sent from the BS to the BBB/Pi, the BBB/Pi will at his turn forward the command to the STM32F3.

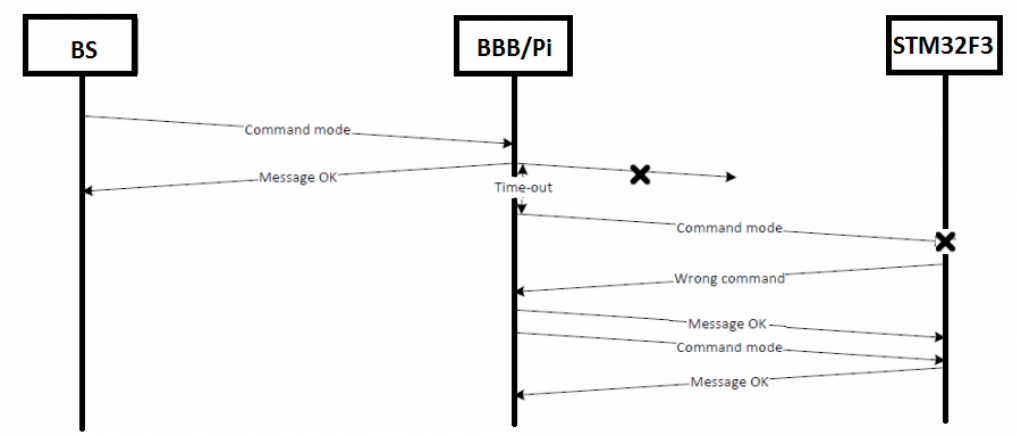


Figure 3.13: Protocol diagram: Normal Command [Jer14]

The situtation that is shown in Figure 3.13 represents the following:

1. BS sends a *Command* to the BBB/Pi.
2. The BBB/Pi receives the *Command*, and reacts with a*MessageOK* to the BS. At the same time it will send the *Command* to the STM32F3.
3. The STM32F3 does not react, before the time-out, to the *Command* with a MessageOK, so the BBB/Pi resends the *Command*.
4. The STM32F3 receives a *Command* with errors. Because of this, a *Wrong Command* message is sent to the BBB/Pi.
5. The BBB/Pi receives the *Wrong Command*, answers with a *MessageOK* and resends the requested *Command*.
6. STM32F3 receives a flawless *Command* and replies to the BBB/Pi with a *MessageOK*.

- **Errors** The first error related with the protocol is depicted in Figure 3.14 on the next page and represents a Data request that has been sent from the BS to the BBB/Pi and, then, from the BBB/Pi to the STM32F3. After receving this message, the STM32F3 is unable to reply because there is no data available from the specified sensor.

  Figure 3.14 on the following page represents the following:

  1. The BS request some data from the BBB/Pi. The BBB/Pi replies to that with a *MessageOK* and also transfers the *Command* to the STM32F3.
  2. As a result of the incoming request, the STM32F3 replies with a *MessageOK*.

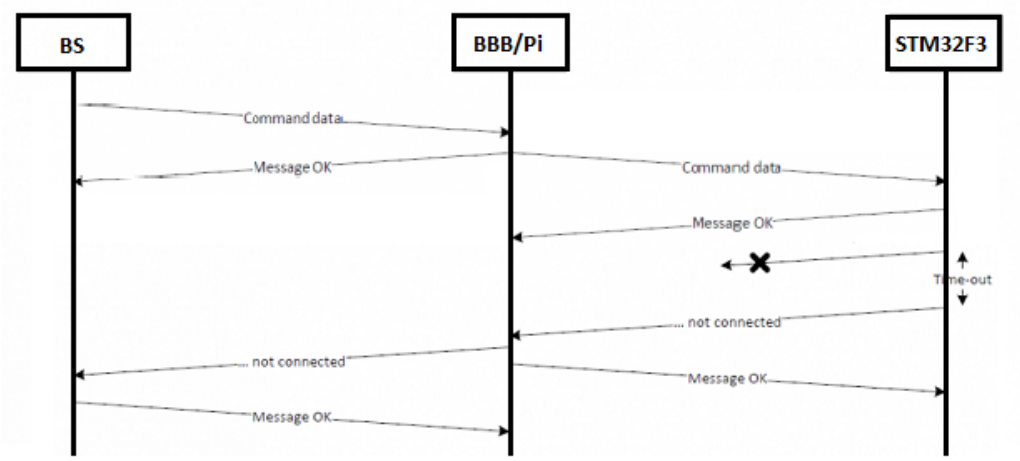


Figure 3.14: Protocol diagram: Unavailable data [Jer14]

3. The STM32F3 checks the incoming request but there is no data available to be able to reply to that specific message. The STM32F3 will wait for data for a specified time interval.
4. When there is no new data before the time-out, a ... *Not Connected* error is sent to the BBB/Pi.
5. After reception of the ... *Not Connected* error, the BBB/Pi transfers the error to the BS and sends a *MessageOK* to the STM32F3.
6. At last, the BS replies to the error with a *MessageOK* to the BBB/Pi.

The last protocol diagram, shown in Figure 3.15 on the next page, represents the implemented time-out in the BBB/Pi. When a call for an action is made from the BS and the BBB/Pi is not able to contact the STM32F3, it will retry two times. If during these three trie no connection is established, an STM32F3 not connected message will be sent back to the BS. This error, however, is not included in this protocol because it occurs between the BS and BBB/Pi.

Figure 3.15 on the following page represents the following:

1. BS sends a *Command* to the BBB/Pi.
2. The BBB/Pi tries to transfer this message to the STM32F3.
3. The STM32F3 does not react to this command, after a time-out the BBB/Pi sends the *Command*again.
4. After three time-outs, there is still no reaction of the STM32F3.

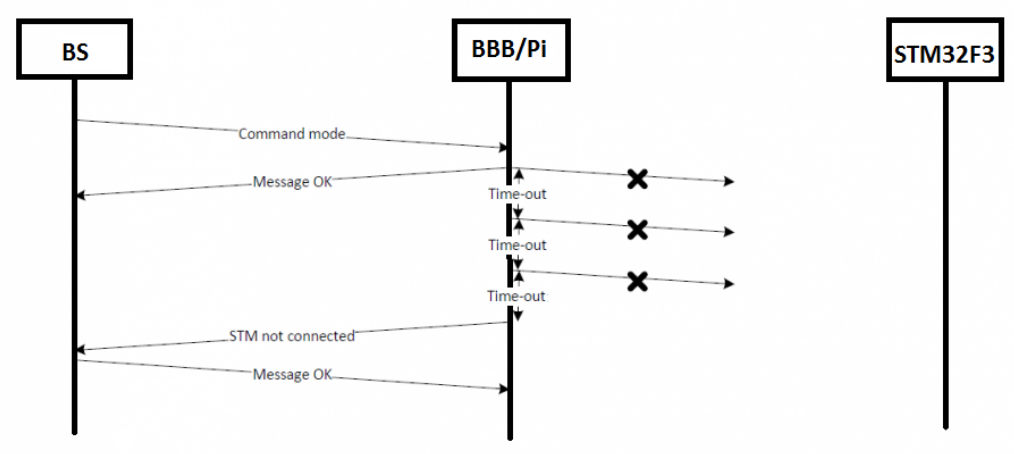


Figure 3.15: Protocol diagram: No connection between BBB/Pi and STM32F3 [Jer14]

5. The BBB/Pi sends a *STM32F3 Not Connected* error to the BS due to this inactivity of the STM32F3.
6. When the BS receives the message, it sends a *MessageOK* to the BBB/Pi.

3.6.2 Protocol Update

A few correction and new functionalities were added to the protocol. The list of updates includes:

- In the former protocol model a mode selection was possible between environmental mode and regatta mode. This was changed to a more general Mode On and Mode Off form. This makes it possible to implement more modes with different sensors and timerates.
- A second addition to the protocol is a message that shows that the data transfer is finished. The reason why this is added is to make it clear to the BBB/Pi when a full data request is handled. When all the datapackages of a certain request are finished, they will be followed by this DataFinished command. The ID of this DataFinished message is ”90h”. A graphical representation of the function of this message can be found in the protocol diagram, depicted in Figure 3.16 on the next page. After all datapackages are sent, a last command is exchanged between the STM32F3 and the BBB/Pi. To confirm this last command, the BBB/Pi replies with a MessageOK. When this is done, the data command is terminated.

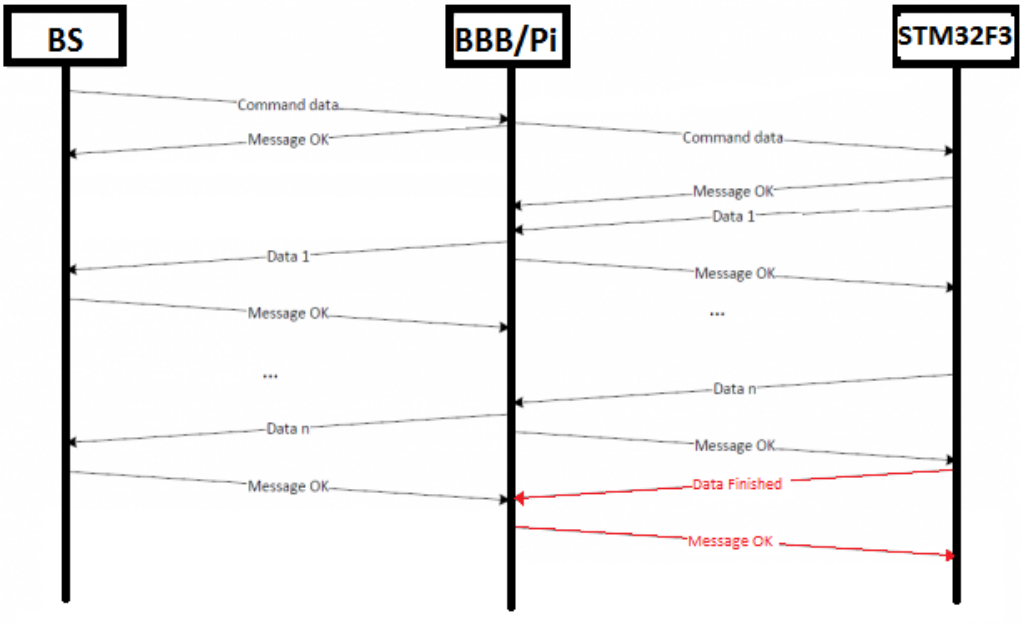


Figure 3.16: Protocol diagram: data finished

Figure 3.17 on the following page displays the overall protocol and Table 3.5 on page 49 shows the complete list of implemented messages.

- A third change made to the protocol relates to the checksum. In the original protocol, the checksum was defined as ”count all ones in the message from ”$” to ”*” (both included) and send this as a checksum. This method was really hard to implement in python on the BBB/Pi. Because of this the checksum was changed to CRC-32, a 32 b cycle redundancy check. More information about this checksums can be found in chapter 3.6.

  This presents as a result that the message format grew 2 b, and now can be described as can be seen in Figure 3.18 on the following page.

- The data package was changed too. Instead of using a ZDA field, it was changed to a mode field. This byte will show the type of message, broadcast or data. At the end there is no longer a carriage return and line feed send. The checksum used in the data package also changed to CRC-32. The new data message can be found in Figure 3.19 on page 49.

- The data chunks used in the data message will have a fixed length. This to make it easier to send and receive those messages. To get to that length all messages will be completed with a certain specified byte, for instance a ”0x01” or ”0xAA”.

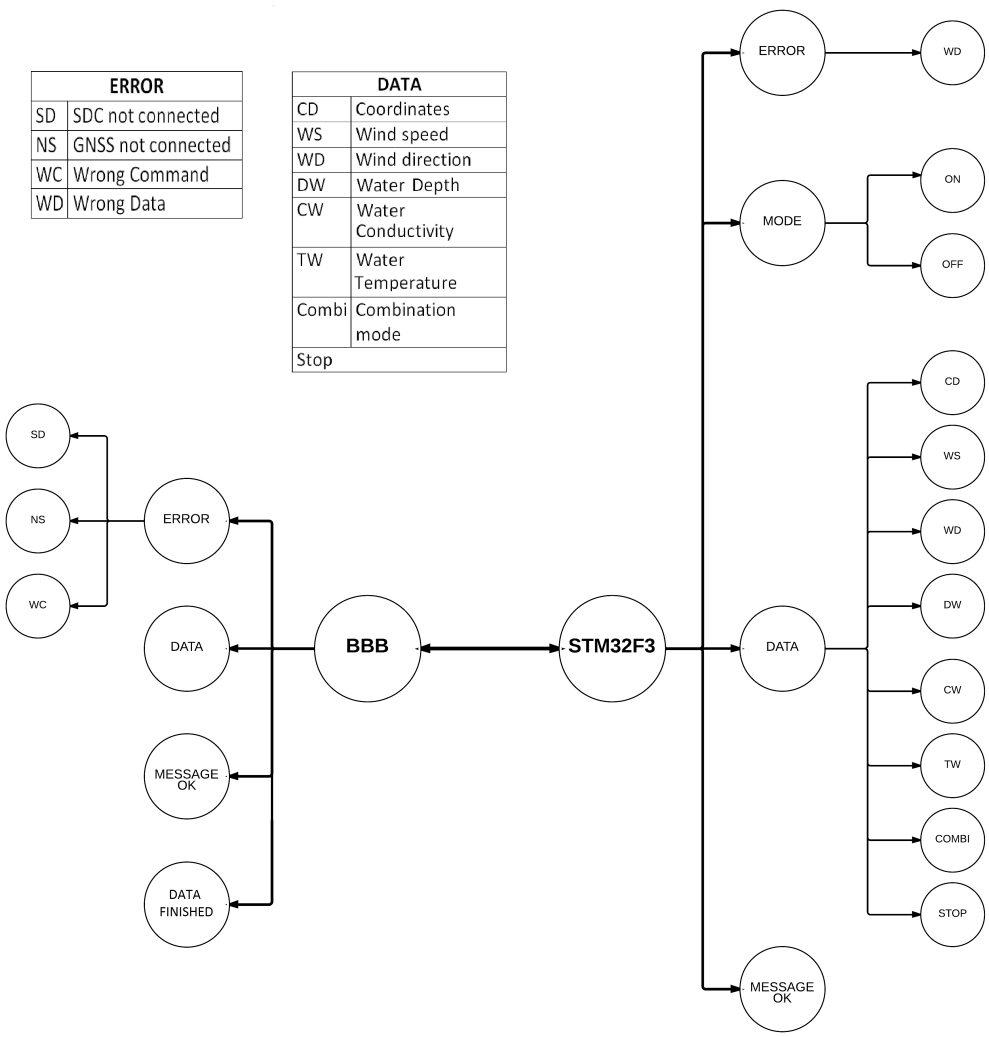


Figure 3.17: Protocol overview

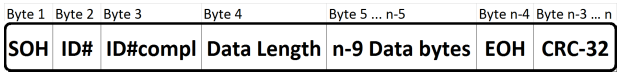


Figure 3.18: New protocol format

- The ID that can be used in the combination command to describe the sensor data can be the same as the normal command ID for those sensors. These ID can be found in Table 3.7 on page 50. These are used instead of the previous 3 byte-length ID.
- A final change acts on the presentation of the numbers that have to be sent. This was never clearly stated in the previous report, but all numbers that

Table 3.5: Possible Protocol ID (v.2)

ID Function 1st digit	***ID 2nd digit***	***Function***
1: Mode selection	0	On
	4	Off
2: GPS	3	Coordinates
3: Wind	1	Velocity
	6	Direction
4: Water/CTD	1	Depth
	5	Conductivity
	9	Temperature
6: Operation	2	Combination
	7	Stop
8: Errors	2	SDC not connected
	4	GNSS not connected
	6	Wrong command
	8	Wrong data
9: Data	0	Data finished
	9	Message OK

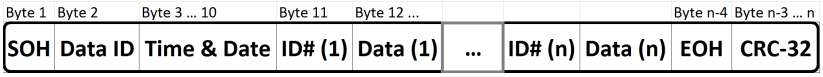


Figure 3.19: Protocol data message

are used in the protocol fields are sent as hexadecimal numbers.

3.6.2.1 New Messages

All new implemented messages use the same structure as the original commands as can be seen in Figure 3.20.

SOH	ID	IDcomplement	Datalength	Data	EOH	CRC-32

Figure 3.20: New protocol format

- **Mode On:** This command is used to add an extra mode to the buoy. The Mode On message defines which sensors are required for this mode. It will also define at which timerate the data must be captured, which protocol that must be used and if the mode requires broadcast or not.

- **SOH:** ”$”
- **ID#:** ”0x10”
- **ID# compl:** ”0xEF”
- **Data Length:** The number of databytes used to transmit the data payload
- **Data:** The data payload
- **EOH:** ”*”
- **Checksum:** CRC-32 checksum of the message
  The data that is send in the command has a specific form represented in Figure 3.21:

Figure 3.21: Mode On data

ModeID	***#sensors***	***Sensor info***	***Broadcast***	***Timer ΔT***	***Protocol***
3 B	1 B	#sensors x 1 B	1 B	4 B	1 B

- **ModeID:** Every mode will get a specific ID that is given by the base station. An example of a mode ID is shown in Table 3.6.

Table 3.6: Mode ID examples

Mode ID	***Definition***
00	Environmental mode
01	Regatta mode

- **# Sensors:** Indicates the number of sensor functions that will be used
- **Sensor information:** Specifies the exact functions that are needed. Uses the same ID as in the normal protocol ID, these ID are listed in Table 3.7:

Table 3.7: Possible Protocol/Sensor ID

Sensor ID	***Definition***
23	Geodetic Coordinates
31	Wind Velocity
36	Wind Direction
41	Water Depth
45	Water Conductivity
49	Water Temperature

The ID values from the standard commands are used here because they describe the same sensors/functions. The ID values that are used are the same as the ones used in a combination command. This similarity is a result of their function. The combination message and Mode On command have the same structure. Their data chunk contains ID values that represent certain sensor functions. That is the reason why the ID values are the same.

  - **Broadcast:** Specifies if there is a real time broadcast to the base station or not. (1 or 0)
  - **Timer interval:** Specifies the time between two data captions
  - **Protocol Type:** Can be used to implement multiple protocols (binary and NMEA, etc.)

- **Mode Off:** Uses the same form as Mode On. The main difference is the ID and the fixed data length of 3 B. This is chosen to get a clear difference between the mode commands and the normal commands that have a datalength of 0 B, 1 B or 2 B. Due to the fact that the ID itself is only one byte long, two zeros are added.
  - **SOH:** ”$”
  - **ID#:** ”0x14”
  - **ID# compl:** ”0xEB”
  - **Data Length:** ”3”
  - **Data:** The ID of the mode that has is to stop and two zeros: 0xID0000
  - **EOH:** ”*”
  - **Checksum:** CRC-32 checksum of the message
- **Data Finished:** This new command shows the end of a data request. It is a small command that contains no data.
  - **SOH:** ”$”
  - **ID#:** ”0x90”
  - **ID# compl:** ”0x6F”
  - **Data Length:** ”0”
  - **Data:**
  - **EOH:** ”*”
  - **Checksum:** ”0xD713DAB6”

3.6.3 Priorities

3.6.3.1 Message Priorities

The BBB/Raspberry receives messages from the base station in a random sequence. Every call to the buoy can be done on every moment of time. Because of this possible variety of the messages, a priority scheme is needed. This should be implemented when a new command arrives due to the already running commands. When this new command arrives, it will request an action from the BBB/STM32F3. Without a priority, the incoming message has to wait untill all the previous commands are executed. This however, is not the correct approach on this kind of application .

Some messages in the protocol are more important than others. One example are the error messages. When an error occurs, this has to be mentioned/sent immediately to the upper level (the master in case of the slave, or the base station in case of the BBB/Pi). To resolve this issue, some priority levels are introduced and implemented.

3.6.3.2 Priority Levels

The proposal for the different levels is based on the earlier developments in the project/protocol. The levels are basically based on the message of the protocol. This means that each type of action or sensor gets its own priority level. At that level there is no further contradistinction implemented. The proposal for the priority levels, based on the protocol messages, can be found in Table 3.8 on the next page. The messages that are related to the data flow, MessageOk and Data Finished, have no priority. This is a result of their function. They are just used as a handshake or acknowledgement of arrival (MessageOk) or a finished request (Data Finished).

A result of these priorities is that the data transfer can be stopped due to a higher priority message. When there is a higher priority message arriving at the BBB/Pi, it will send a stop command to the STM32F3. Afther that, it will generate a new request for the new incoming, higher priority, message. Another example is an occuring error somewhere in a sensor. This error message should immediatly be transferred to the BBB/Pi. This means the data transfer will be halted at the next possible moment to send the error to the BBB/Pi. When the error has been sent and arrival is confirmed by a message ok, the previously stopped data transfer will continue.

Table 3.8: Priority levels

Priority level	***Type of message***
1	Operation
2	Errors
3	Mode selection
4	GPS
5	Wind
6	Water/CTD

During the detailed theoretical implementation and brainstorming some changes were made to the priority levels. In general, the prior suggested form for the priority is well chosen. But there are some small issues. The protocol messages in the sixth message class, called operation, can be separated. This is done because of the difference in working principle between the stop message and the combination message. The stop message is a high priority message to stop the transaction. This means that this message deserves the highest priority. In the contrary, the combination message is a normal data request. In principle, there is no difference between a data request from a sensor and the combination message, since the combination message is just a combination of different sensor data collected in one package. Based on these thoughts, the combinations message priority is changed, according to what is presented in Table 3.9.

Table 3.9: Priority levels v.2

Priority level	***Type of message***
1	Stop
2	Errors
3	Mode selection
4	Combination
5	GPS
6	Wind
7	Water/CTD

Table 3.10 on the next page shows the protocol messages and their assigned priority.

Table 3.10: Priority levels (detailed)

Message ID 1st digit	***Message ID 2nd digit***	***Definition***	***Priority level***
1: Mode selection	0	On	3
1	4	Off	3
2: GPS	3	Coordinates	5
3: Wind	1	Wind velocity	6
3	6	Wind direction	6
4: Water/CTD	1	Water depth	7
4	5	Water conductivity	7
4	9	Water temperature	7
6: Operation	2	Combination	4
6	7	Stop	1
8: Errors	2	SDC not connected	2
8	4	GNSS not connected	2
8	6	Wrong command	2
8	8	Wrong data	2
9: Data:	0	Data finished	/
9	9	Data OK	/

Table 3.10 shows that different protocol messages have the same priority level. When this occurs, the requests are dealt on FIFO basis. This FIFO way of working is implemented because it is very simple, but provides a good way of controlling the requests. If needed or desired, another method can be implemented. But it is sufficient for now, to start the priority implementation.

3.6.4 Priorities with the STM32F3 and ChibiOS

The STM32F3 is a part of a master-slave architecture that is built with a BeagleBone Black/Raspberry Pi and the STM32F3. In this architecture the BBB/Pi is the master control unit and the STM32F3 is the slave control unit. This means that the STM32F3 must listen to the BBB/Pi and act as requested. This suggests that the STM32F3 itself does not specify priority levels, which is correct in most cases. Only in case of errors, the slave control unit assigns priorities. This will be discussed later.

When the BBB/Pi requests data, it sends a protocol message with the wanted data request. At arrival of the request, the STM32F3 checks the message, and decides which data has to be sent. When this has happened, the STM32F3 starts

transferring the data to the BBB. This goes on until the transaction is finished, or until a stop sending message is sent by the BBB. The last option is only possible in environmental mode. More about that will be presented in a later paragraph.

3.6.4.1 Priority on the STM32F3

As stated earlier, there is no need to develop a complex priority level scheme for the STM32F3 side. The only thing that can happen on the slave side that is not requested are the errors. In case of an error, this becomes priority number one. At this point the slave will overrule the BBB/Pi and send the error message as the next possible package to the BBB. This means that the BBB/Pi must check every incoming message.

Another option is that there is an error flag/hardware connection that is activated when an error occurs. This, however is not that easy to implement due to the chosen connection between the STM32F3 and the BBB/Pi.The STM32F3 will be connected to the BBB/Pi via a USB connection. This implicates that they will exchange data as a stream of bytes. When tests show that there is need for an error flag, then an extra hardware connection between the BBB/Pi and the STM32F3 can be used.

3.6.4.2 Modes

The buoy can work in different modes. These modes are based on the same structure, with a defined number of implemented tasks. This pattern is identical in all modes. This provides a simple way to add new modes afterwards. The first target of the buoy project is to implement two modes. These modes are the regatta mode, and the environmental mode. In reality this means three different working principles, because all modes must be able to work at the same time. In order to provide this, a good priority structure must be implemented and all data must be controlled very efficiently.

When the buoy is collecting and sending data, there is an almost continuous data transfer between the BBB/Pi and the STM32F3. However, there is a difference depending on the chosen working mode: environmental, regatta or dual mode.

- **Environmental Mode:** When the buoy is running in an environmental mode, the data from the sensors will be saved to the SD card. This proces will continue until the mode is stopped. During this process and afterwards, specific data can be downloaded/requested from the SD card. These requests are done via protocol commands and will be handled by the STM32F3.

When the buoy is in unicast communication mode, the data transfer is a sequential transmission of the stored data from the specified period of time and sensors and may take a long time, depending on the size of the data requested. An example of a higher priority request can be a GPS request when the STM32F3 is sending data from the CTD. When a request with a higher priority is sent by the base station, the BBB/Pi has to react to it. This causes the BBB/Pi to send a stop command to the STM32F3. At this point the STM32F3 stops transmitting. Afterwards, when a new data request arrives from the BBB/Pi, the STM32F3 starts transmitting the data. After finishing this transfer, the BBB/Pi restarts the previously paused transfer. To restart a previously halted data transfer, the BBB/Pi has saved the date stamp from the last arrived package. It will send a new command specifying as date the date stamp from the last arrived package.

- **Regatta Mode:**

  The regatta mode transmits bursts of data to the nearby listeners using multicast communication. The data is defined when the regatta mode is started. When the mode is started with a Mode On command, all desired sensors are chosen/selected. During the whole regatta/multicast cycle there is no change in the selected collection of sensors. All sent messages are based on the same structure.

  This mode uses the same message structure, only the data content is updated at a predefined rate. This will continue until a Mode Off is sent to the STM32F3.

- **Dual Mode:** This provides a possibility to run multiple modes at the same time. In this mode the BBB/Pi and STM32F3DISCOVERY have to take care of the implemented modes and their priority. In the case of only two modes, environmental and regatta, the regatta mode has a higher priority than the environmental mode. This means that, when a regatta message is ready in the STM32F3, the environmental data download has to be stopped. After that the regatta data will be sent. If more modes are implemented, the higher priority levels will have precedence over the lower levels. Afterwards, the lower priority tasks can continue.

3.6.4.3 Errors

When an error occurs in the STM32F3 or with one of the connected components/sensors, an immediate reaction should follow. This reaction will generate an error in the STM32F3 and an error command will be sent to the BBB/Pi, who will, in turn (and when there is a connection), send the error message to the base station. The fact that the error must be sent immediately adds some

disturbance to the system when it is transferring large chunks of data. This data transfer will be paused for one instant to send the error message. This happens due to the priority scheme implemented. On the STM32F3 side, errors have the highest possible priority. Due to that, they have to be sent on the first possible moment. This means that the STM32F3 should check for available errors before sending every data packet.

3.6.5 CRC-32 Checksum

The first protocol description mentioned a checksum method that was very easy on paper but hard to implement. An other thing about that preposition, the checksum counted all the ones in the message, is that it was rarely used in other projects. Because of these two points, the used checksum was changed to the CRC-32 checksum.

The 4 B Cyclic Redundancy Check (CRC) is an error-detecting code that is often used in digital networks and storage devices. It can be used to detect accidental changes to raw data. A cyclic redundancy check code uses a defined generator polynomial. This polynomial is used as a divisor in a polynomial long division. This division takes the message as the dividend and the the quotient is discarded. The remainder becomes the result. There are different CRC algorithms available. Each of them are called an-bit CRC, where the check value is n-bits. In this case a 32-bit CRC is used, the IEEE CRC-32 [Chr14].

When the CRC is used in a program or device, a check value of a fixed length is calculated for each block of data that is sent. This binary sequence is appended to the data and forms a codeword. When a message that uses CRC is received, the codeword is read and, meanwhile, the receveir recalculates the CRC with the incoming data. When the two don't match, a data error occured.

In more sophisticated programs a corrective action can be performed. But in this case nothing happens, except that the sender has to send the message again. When the two codes are identical, the message is error free. Although this method is not 100% certain, there is a small probability that the message may contain an undetected error.

A small example [Wik14b] of the CRC method will be given, more examples about the computation of cyclic redundancy checks can be found in [Wik14a]. This example shows a 3 bit-length polynomial, $x^3 + x + 1$, or written in binary "1011" ($1 \cdot x^3 + 0 \cdot x^2 + 1 \cdot x + 1$). At the end, a 3 bit-length result will be found. The CRC calculation is executed on a 14 bit-length word.

Binary Representation	**Explanation**
11 0100 1110 1100	Start Message
11 0100 1110 1100 000 1011	Add n-bits for n-CRC,in this case, 3 b for 3-CRC Divisor (4) = $x^3 + x + 1$
01 1000 1110 1100 000	Result

The algorithm acts on the bits directly above the divisor in each step. The result for that iteration is the bitwise XOR of the polynomial divisor with the bits above it. The bits not above the divisor are simply copied directly below for that step. The divisor is then shifted to the next 1 bit on the right, and the process is repeated until the divisor reaches the right-hand end of the input row. Figure 3.22 shows the entire calculation:

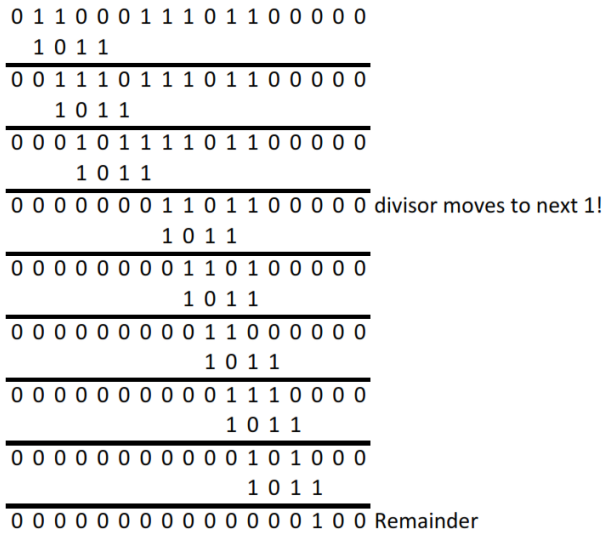


Figure 3.22: CRC-32 calculation

In the end the CRC codeword is ”100”.

3.7 Board with GNSS and SD card

In the end of the project, some test were ran to see if there could be an implementation of the GNSS and the SD card. To to this, an interface board must be used.

3.7.1 Interface Board

Figure 3.23 on the following page shows the lay-out of this PCB that was made during last semester. And some test were ran with that board. However, this

board was damaged. There were conductive traces that were scratched. And some corrections were made by use of wires. After a good look, the board showed also missing components, this was verified with the scheme shown in Figure 3.24. Because these mistakes were discovered in the last week of the project, there was no time left to draw a new layout.

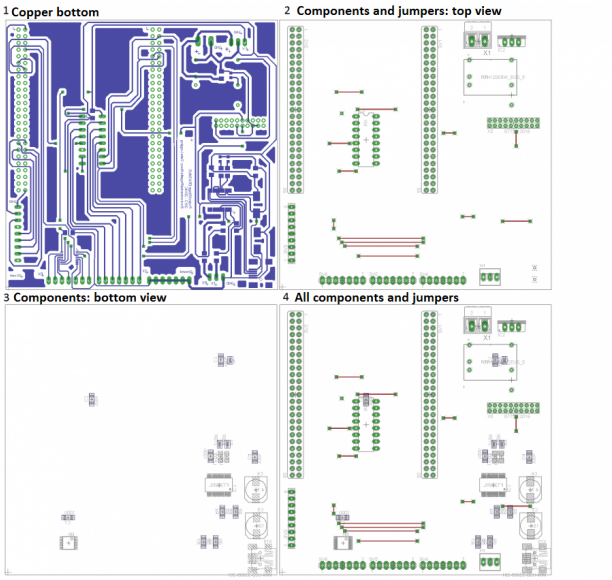


Figure 3.23: Layout PCB [Jer14]

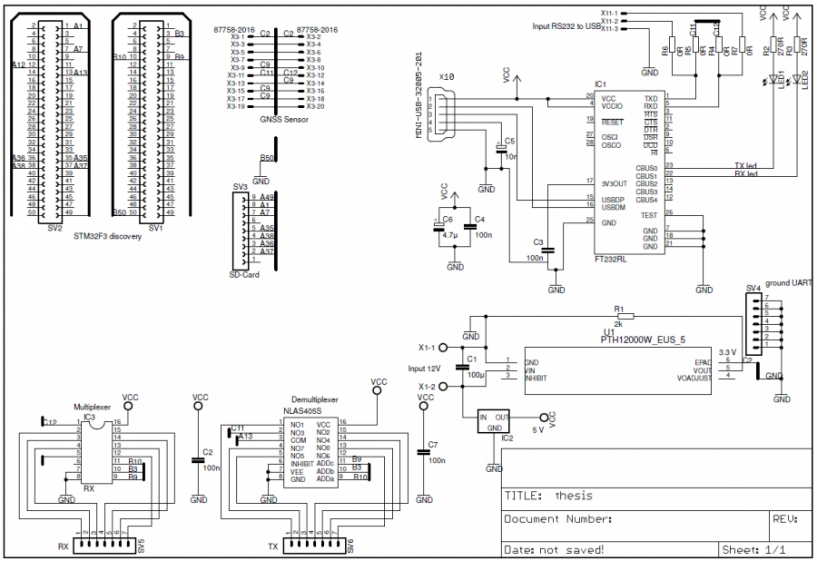


Figure 3.24: PCB Scheme [Jer14]

In order to be able to run some test without a costum made PCB, a special development board for the STM32F3DISCOVERY was ordered. This Waveshare Open 32F3-D Standard [Wav14] was used to run some small tests. An image of this board can be found in Figure 3.25.

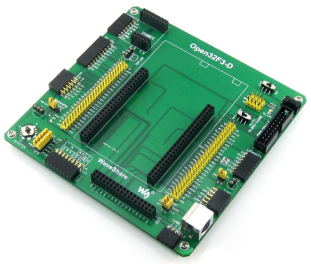


Figure 3.25: Open 32F3-D Standard Board [Wav14]

3.7.2 SUPERSTAR II GNSS

The GNSS was configured in order to try and run a test. This test did not work, due to the problems with the board. The pins of the connector were not connected to the correct STM32F3 pins. There were some scratched conductive traces too.

The configuration of the GNSS was executed by connecting it to a converter that converted the pins of the GNSS to a serial connector, this is depicted in Figure 3.26. This serial connector can be used to connect the SUPERSTAR II to a computer. In order to do so, a serial-to-USB cable was used.

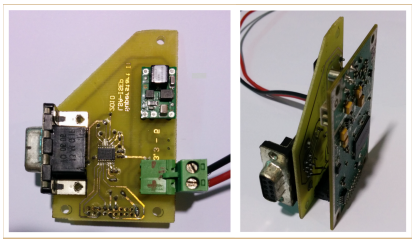


Figure 3.26: GNSS with serial convertor

When the connection is made to the computer, Starview can be used to make the configuration. The procedure to do this configuration was described by Jeroen Vervenne in [Jer14].

3.7.3 SD Card

The SD card used in this project is a micro-SD. A Parallax micro-SD Card Adapter [Par12], as shown in Figure 3.27, is used to connect the card to the board.

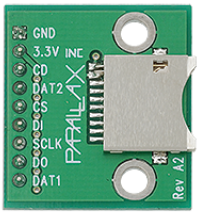


Figure 3.27: Parallax micro-SD Card Adapter [Par12]

This adapter has nine connections. But only seven of them are used in this case. This is a result of the use of a SPI. This is explained in [Jer14]. In order to get the SD card working with the STM32F3 the used pins must be connected to pins on the STM32F3. The way that the pins must be connected is shown in Table 3.11.

Table 3.11: Pin connection of STM32F3 and SD card adapter

Function	***Parallax adapter pin***	***STM32F3 pin***
Ground	GND	one of the GND pins
Vcc	3.3 V	one of the 3 V pins
Card Detect	CD	PC2
	DAT2	/
Slave Select	CS	PB12
Master-Out Slave-In	D1	PB15
Serial Clock	SCLK	PB13
Master-In Slave-Out	D0	PB14
	DAT1	/

The SD card adapter was connected to the Open 32F3-D board with jumper cables. This can be seen in Figure 3.28 on the following page.

The next step was to test the connection. A test was ran with the code for the SD card that was developed last semester. In order to get this code running on another machine, some actions must be done. The plugins to use FatFs must be installed. To do so, navigate to the /ChibiOS_2.6.4/ext/ folder

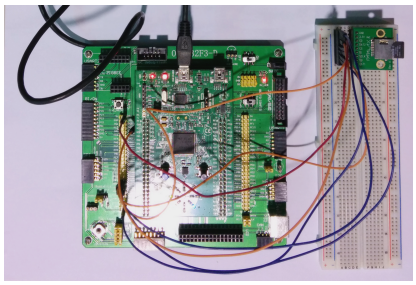


Figure 3.28: Open 32F3-D and SD adapter

and unzip the ”fatfs-0.9-patched” folder. The files in this folder must be copied to /ChibiOS_2.6.3/os/various/FatFs/ to get the SD card code working. Now the code can be compiled to test the SD-card connection. As a test, to see if the connections to the SD card adapter work, some words were written to the SD card.

3.8 Conclusion

This project uses a STM32F3DISCOVERY to implement the data collection. To run this board and have some usefull functions ChibiOS is used to implement a RTOS. ChibiOS is based on basic C-code and in order to edit those files a Linux environment is used with a basic text editor and some commands to program the STM32F3.

In order to get a good communication between the STM32F3 and the BBB/Pi, a protocol is used. This protocol was defined by the previous students, but was changed in order to implement extra functions and to make implementation easier in both the BBB/Pi and the STM32F3. Probably the biggest change to the protocol is the checksum. It now uses a standard CRC-32 checksum as it was defined by IEEE. This checksum was easier to implement and makes it possible to implement error correction in the future.

Chapter 4

Implementation and Test

4.1 Introduction

The main idea to solve the buoy problem is to implement multiple threads in the STM32F3. A thread can be described as: "In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by an operating system scheduler." [Wik14e]. All the implemented threads should take care of a specific task. This task can be sending, receiving, controlling a sensor, executing takst, etc.

4.2 Implemented Threads

There are three main threads in this solution. A receiving thread, a controlling thread and a sending thread. Next to those main threads there are several more. One for each implemented side structure. Those structures are the SD card and the implemented sensors. This means that there will be at least four or five always running threads. Those are the receiving thread, controlling thread, sending thread and the GNSS thread. Eventually you can add the SD card thread, but this might be deactivated when there is no need to save the received data from the GNSS. When we take a closer look on the main threads, that are graphical represented in Figure 4.1 on the next page, you can see the functions of the three most important threads.

Figure 4.1 on the following page shows how the threads are connected. The controlling thread is named executor here. All the threads are connected via queues. And only the receiving and sending thread are connected to the serial in/output.

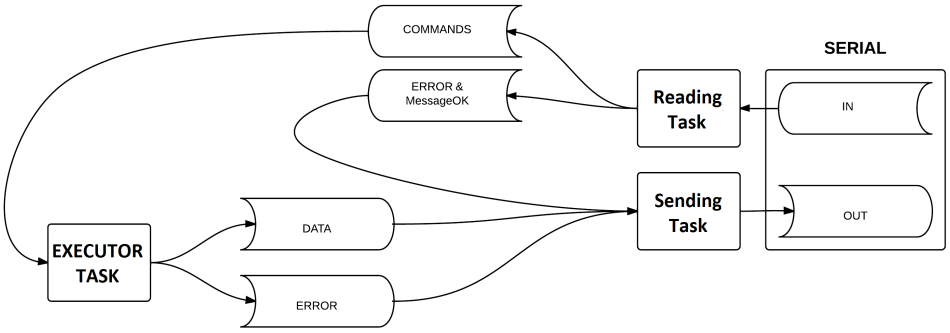


Figure 4.1: Basic thread functions: main actions

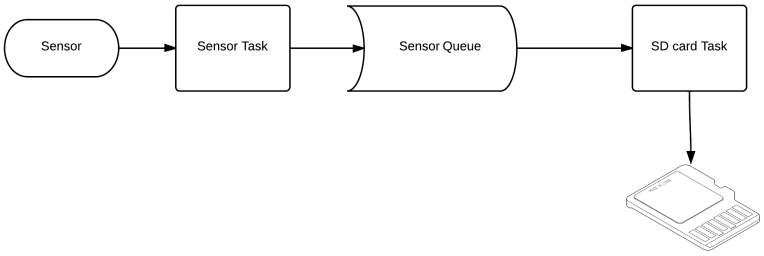


Figure 4.2: Basic thread functions: sensor actions

On the input side, the only possible arrivals are commands, errors or MessageOK, the handshake. The errors and MessageOK should be handled immediately. That is the reason of the different queue. The incoming commands should be queued and can continue. After the Command queue, the command arrives at the executer task. This task will execute the requested actions by the command. This can go from switching sensors on/off, start broadcasting, fulfil a data request, etc.

If the command is not a data request, it will be handled in the executor task and there will be no further actions. When the incoming command asks for data, the executor will prepare the needed data and pass it to a data queue. When data is passed to the queue, it can be sent. The sending part is handled by the sending thread. This thread should check for available errors or if there are MessageOK messages that have to be sent. When those sending needs are handled the prepared datapackages will be sent via the serial connection.

The other part of the threads, sensor threads and SD card threads work via the principle that is shown Figure 4.2.

Figure 4.2 shows two threads. One of them is connected to a sensor, and the

other is connected to the SD card. The sensor thread's first task is to get data from the sensor at the right/wanted moment. After this is done, it has to make sure the data is translated to the wanted datastructure. When this is done, the data can be transferred to a sensor queue. This queue is the direct road to thread that executes SD tasks. This SD card thread interacts with the SD card. It reads requested data, and it writes data to the SD card when there is incoming data in the SD card thread.

4.3 Communication Threads

The implementation of the communication threads is based on existing ChibiOS functions and consist of multiple C-files. In the main file, the USB connection gets described first. And all parameters get set to have a working connection. After this part all needed threads are described and the functions are implemented there. These threads will be discussed later on. After the thread description (receiving thread, controlling thread, sending thread and the blinker thread) the main routine of the program is started. This main routine starts with initiating and implementing some of the used functions. Also the threads are started. All these parts will be described in the next paragraphs. At last the main action of the main file is described. This is in fact just a sleep command in a while loop. The main function does not do anything else than initiating the used structures and after that go to sleep.

4.3.1 ChibiOS Structures

The initialization of the ChibiOS structures is located in the "main.c" file. The STM32F3 and BBB/Pi will be connected via USB. To make this possible the STM32F3 has to be configured in the correct way. This is done in the upper part of the "main.c" file and also in the "halconf.h" file. The first thing that has to be done to make this possible is edit the "halconf.h" file. In this file, all possible ChibiOS structures are defined. This project uses the USB, to do this the HAL_USE_USB setting/parameter must be set to TRUE.

```
/**
 * @brief    Enables the USB subsystem.
 */
#if !defined(HAL_USE_USB) || defined(__DOXYGEN__)
#define HAL_USE_USB                         TRUE
#endif
```

When this option is enabled a USB driver is implemented that supports device-mode operations.

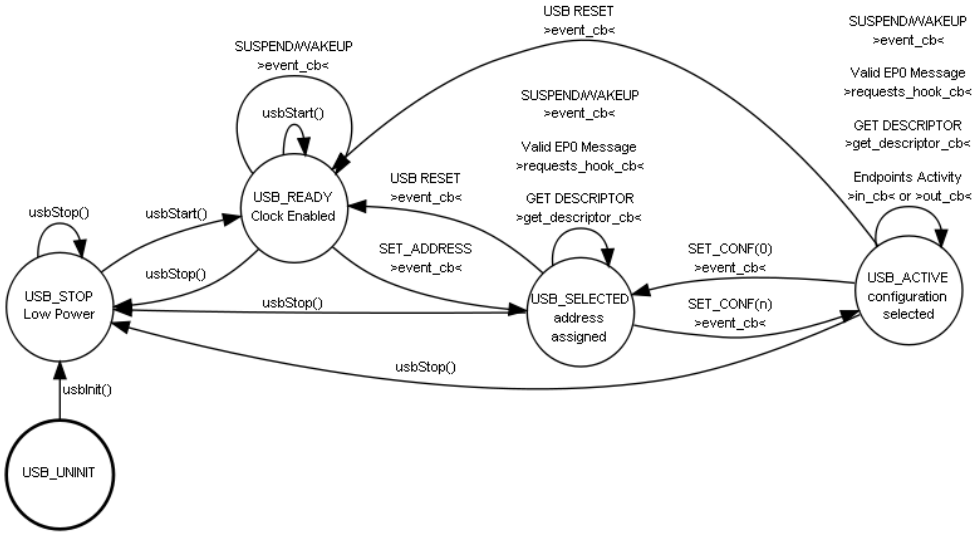


Figure 4.3: USB state machine [Chi14c]

- **Driver State Machine** When the driver is enabled, an internal state machine is implemented. This state machine can be found in figure 4.3. This state machine shows all possible actions for the USB driver.

  The first thing that has to be done to use these structures is to describe the USB device. This is done by setting the USB_DESC_DEVICE structure:

```
/*
 * USB Device Descriptor.
 */
static const uint8_t vcom_device_descriptor_data[18] = {
  USB_DESC_DEVICE (0x0110, /* bcdUSB (1.1). */
                           0x02, /* bDeviceClass (CDC).
*/
                           0x00,/* bDeviceSubClass. */
                           0x00, /* bDeviceProtocol.*/
                           0x40, /* bMaxPacketSize.*/
                           0x0483, /* idVendor (ST).*/
                           0x5740, /* idProduct.*/
                           0x0200,    /* bcdDevice.*/
                           1,/* iManufacturer.*/
                           2, /* iProduct.*/
                           3, /* iSerialNumber.*/
                           1) /* bNumConfigurations.*/
};
```

```
/*
 * Device Descriptor wrapper.
 */
static const USBDescriptor vcom_device_descriptor = {
  sizeof vcom_device_descriptor_data,
  vcom_device_descriptor_data
};
```

After this, the used USB connection is configured. In this part all used functions need to be initiated/get a correct value:

```
/* Configuration Descriptor tree for a CDC.*/
static const uint8_t vcom_configuration_descriptor_data[67] = {
  /* Configuration Descriptor.*/
  USB_DESC_CONFIGURATION(67,/* wTotalLength. */
                         0x02,/* bNumInterfaces.*/
                         0x01,/* bConfigurationValue*/
                         0,/* iConfiguration.*/
                         0xC0,/* bmAttributes (self powered)*/
                         50),/* bMaxPower (100 mA).*/
  /* Interface Descriptor.*/
  ...
  /* Call Management Functional Descriptor. */
  ...
  /* ACM Functional Descriptor.*/
  ...
  /* Union Functional Descriptor.*/
  ...
  /* Endpoint 2 Descriptor.*/
  ...
  /* Interface Descriptor.*/
  ...
  /* Endpoint 3 Descriptor.*/
  ...
  /* Endpoint 1 Descriptor.*/
  ...
};

/*
 * Configuration Descriptor wrapper.
 */
static const USBDescriptor vcom_configuration_descriptor = {
  sizeof vcom_configuration_descriptor_data,
```

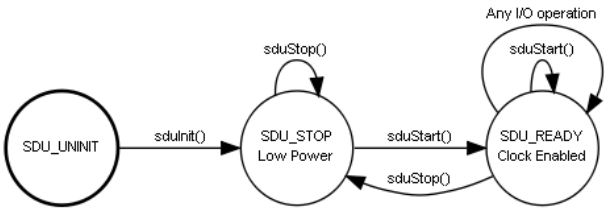


Figure 4.4: SDU state machine [Chi14b]

```
    vcom_configuration_descriptor_data
};
```

Note that not all of the settings are included in the code snippet above, to see them all, the main.c file should be checked. What follows in the main.c file are a lot more configurations, and the configuration ends with:

```
/*
 * Serial over USB driver configuration.
 */
static const SerialUSBConfig serusbcfg = {
  &USBD1,
  USBD1_DATA_REQUEST_EP,
  USBD1_DATA_AVAILABLE_EP,
  USBD1_INTERRUPT_REQUEST_EP
};
```

This configures the used Serial over USB that is used to send the serial data. After activating the driver, the state machine of Figure 4.4 is implemented.

The next part in the ”main.c” file describes the implemented threads and functions. They will be explained in the next section.

4.3.2 Main

The structure of the main file can be found in Figure 4.5 on the next page. This figure shows the used files and the basic structure. On top you have the main.c file that consists of three threads (at this moment) and a blinker thread that is not in the picture. The three threads are:

- Receiving thread
- Controlling thread
- Sending thread

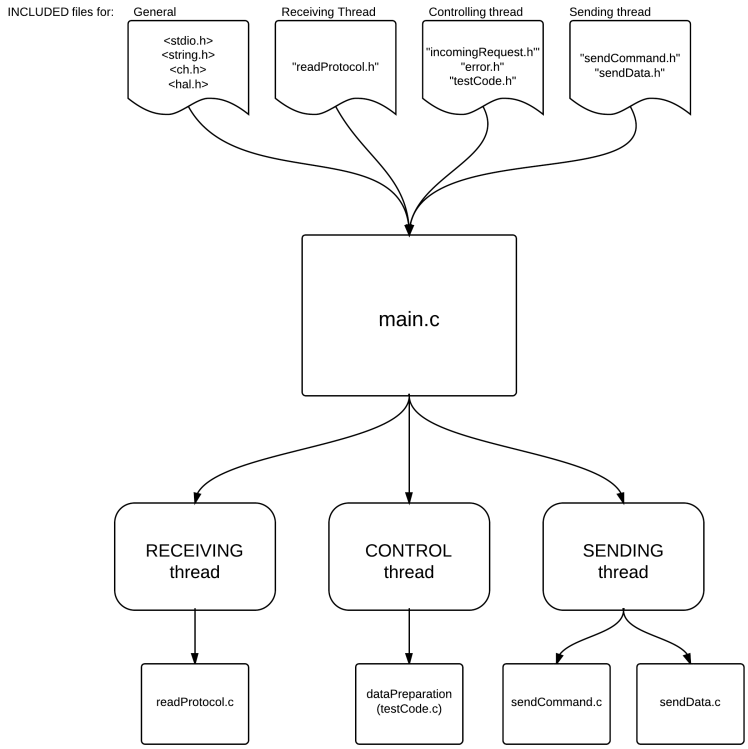


Figure 4.5: Structure of main.c

Every thread calls one or more functions. The Receiving thread just keeps executing the readProtocol() function. The controlling thread keeps executing the testCode() function, or later the function that will execute the data preparation and will handle the incoming requests. The sending thread uses two big functions, the sendCommand() and the sendData(). All these files/functions are included in the main.c file via their header files, as can be seen on Figure 4.5.

The real main() method is used to initiate several functions and implemented structures. It starts with some System initializations and the initialisation of the serial-over-USB CDC driver

```
*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        System initializations.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/* - HAL initialization, this also initializes the configured
        device drivers and performs the board-specific
        initializations.
   - Kernel initialization, the main() function becomes a
        thread and the RTOS is active.
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

  halInit();
  chSysInit();

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        Initialisation of a serial-over-USB CDC driver.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

  sduObjectInit(&SDU1);
  sduStart(&SDU1, &serusbcfg);

  /*
   * Activates the USB driver and then the USB bus
        pull-up on D+.
   * Note, a delay is inserted in order to not have to
        disconnect the cable after a reset.
   */
  usbDisconnectBus(serusbcfg.usbp);
  chThdSleepMilliseconds(1500);
  usbStart(serusbcfg.usbp, &usbcfg);
  usbConnectBus(serusbcfg.usbp);
```

After this part, all other ChibiOS structures that are used are initiated. This is already present in the file but not yet used during the development of the communication part.

The next command in the code makes the incomingRequest file work. And at last, the threads are created. As an example, the creation of the Blinker thread:

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        Thread activation
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

//.........................
// Blinker thread
//-------------------------
// Is used to be able to see if the STM32F3 is working
//.........................

chThdCreateStatic(waThread1, sizeof(waThread1), NORMALPRIO,
                  Thread1, NULL);
```

At the very last of the main.c file, you have the normal main activity. This is in fact just a loop that keeps sleeping:

```
*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        Normal Main Activity
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

  while (TRUE) {

    chThdSleepMilliseconds(1000);
  }
```

After this description of the main file, the threads will be discussed in the next paragraphs.

4.3.3 Receiving Data from the Raspberry Pi/BeagleBone Black

When a command is given by the Raspberry Pi or the BBB, the STM32F3 has to react accordingly. This must be done by decrypting the protocol message to find the wanted action. This will happen in the receiving thread. The receiving thread itself is really small. It just calls the function readProtocol():

```
static WORKING_AREA(waRxThread, 4096);
/*creates working area for this thread.
4096 bytes of memory are allocated */
static msg_t rxThread(BaseSequentialStream *chp, void *arg) {

  (void)arg;
  chRegSetThreadName("Receive");

  while (TRUE) {
        //the green light on the board indicates
        //the running thread
        palSetPad(GPIOE, GPIOE_LED7_GREEN);

        /* readProtocol() call: needs the BaseSequentialStream
        to be able to read incoming data from the usb,
        the other parameters are flags for specific messages
        that arrive or have to be sent.*/
        readProtocol((BaseSequentialStream *)&SDU1, &sendMessageOK,
        &receiveMessageOK, &incomingWrongData, &incomingWrongCommand);
  }
}
```

The implemented function, readProtocol(), is shown as a flowchart in Figures 4.6 on the next page and 4.7 on page 74.

The function starts with reading an incoming byte, this is handled as follows:

```
chSequentialStreamRead(bss, bufRead, 1);
        incByte = (uint8_t) bufRead[0];
        newByte=1;
```

The sequential read reads from the "bss" channel, what was linked to the main serial-over-USB structure in the main file. The incoming byte, a uint8_t number, is saved in the first byte of the bufRead array. When this is done, the newByte flag is set. This flag is used to determine of an incoming byte already is used in a certain action or not.

Before further explanation is done, it can be handy to repeat the basic message/command structure, this structure is shown in Figure 4.8:

"$"	ID	ID complement	Datalength	Data	"∗"	CRC-32

Figure 4.8: Command structure

This is a serial structure, all bytes will arrive one by one and must arrive in this order. This means the message has to start with '$'.

When there is a new byte available, the StartCommand flag is checked. This flag is used to see if there is already an incoming command running through the method, did a '$' arrive before. This is not the case when a new command is arriving at the STM32F3. This brings it to the next step.

The incoming byte is checked, is it equal to 24h? When this is the case, the StartCommand flag is set. Also the readB1 flag is set. This flag shows that the next byte that arrives will be the second byte of a message. When the incoming byte wasn't equal to *24h*, the process starts again when a new byte arrives and is read by the STM32F3. Is it equal to *24h*? Or not? This continues until at last a '$' arrives.

After the readB1 and StartCommand are set, the new incoming byte can proceed to next check. At the next point, if readB1 is 1, the ID of the message has arrived at the STM32F3. So the incoming byte will be transferred to the ID byte in the memory. When this is done, the readB1 is cleared, and readB2 is set. ReadB2 points out that the next byte can transfer to the readB2 check. After those flags are set/cleared, the ID is checked for two specific values. When a Mode On or Off command arrives, a specific action has to be done. To catch this, a modOnOff flag is used. When the ID is *10h* or *14h*, this flag will be set. After these actions a new byte can be read.

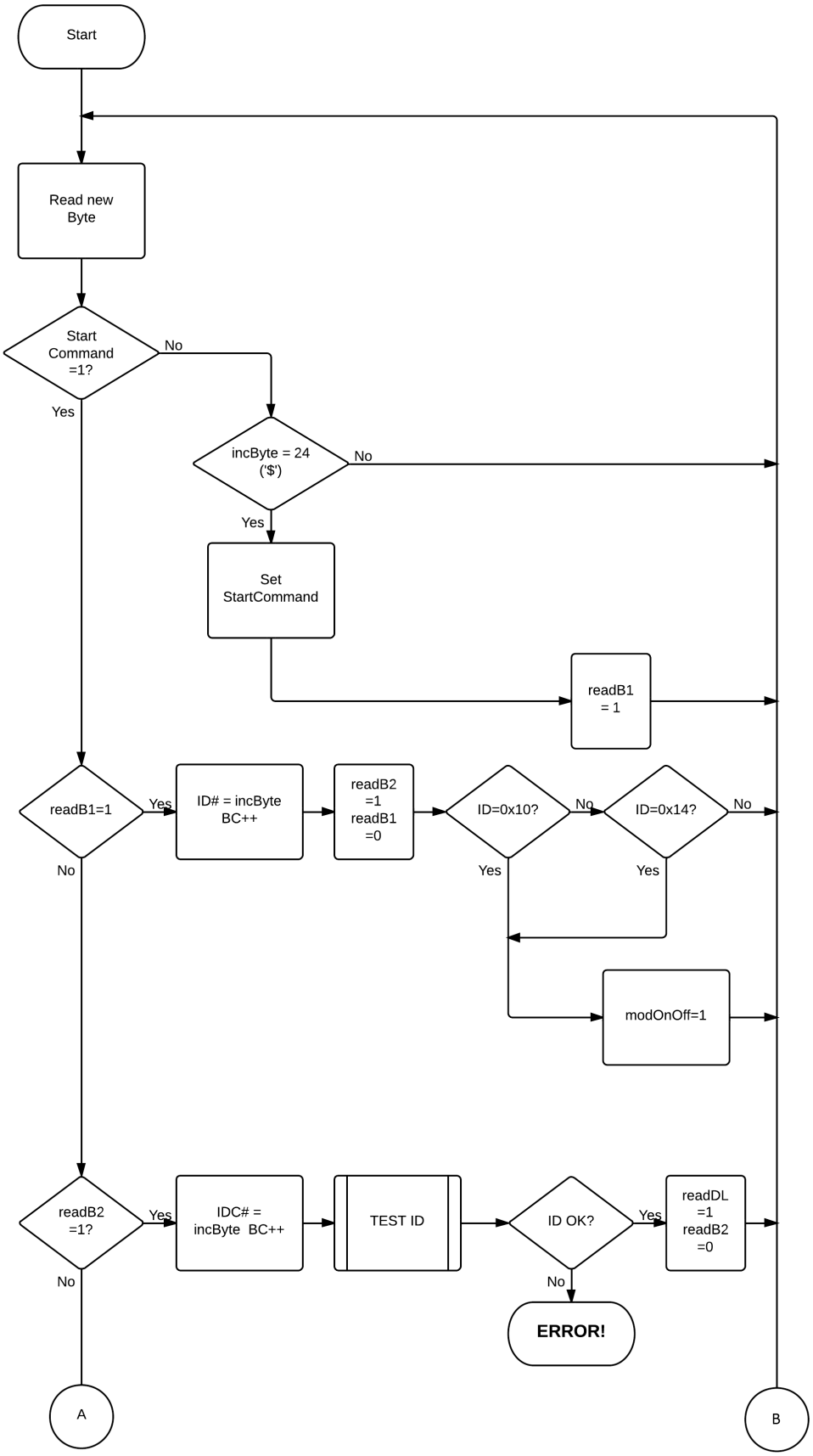


Figure 4.6: Read command (binary protocol), part 1

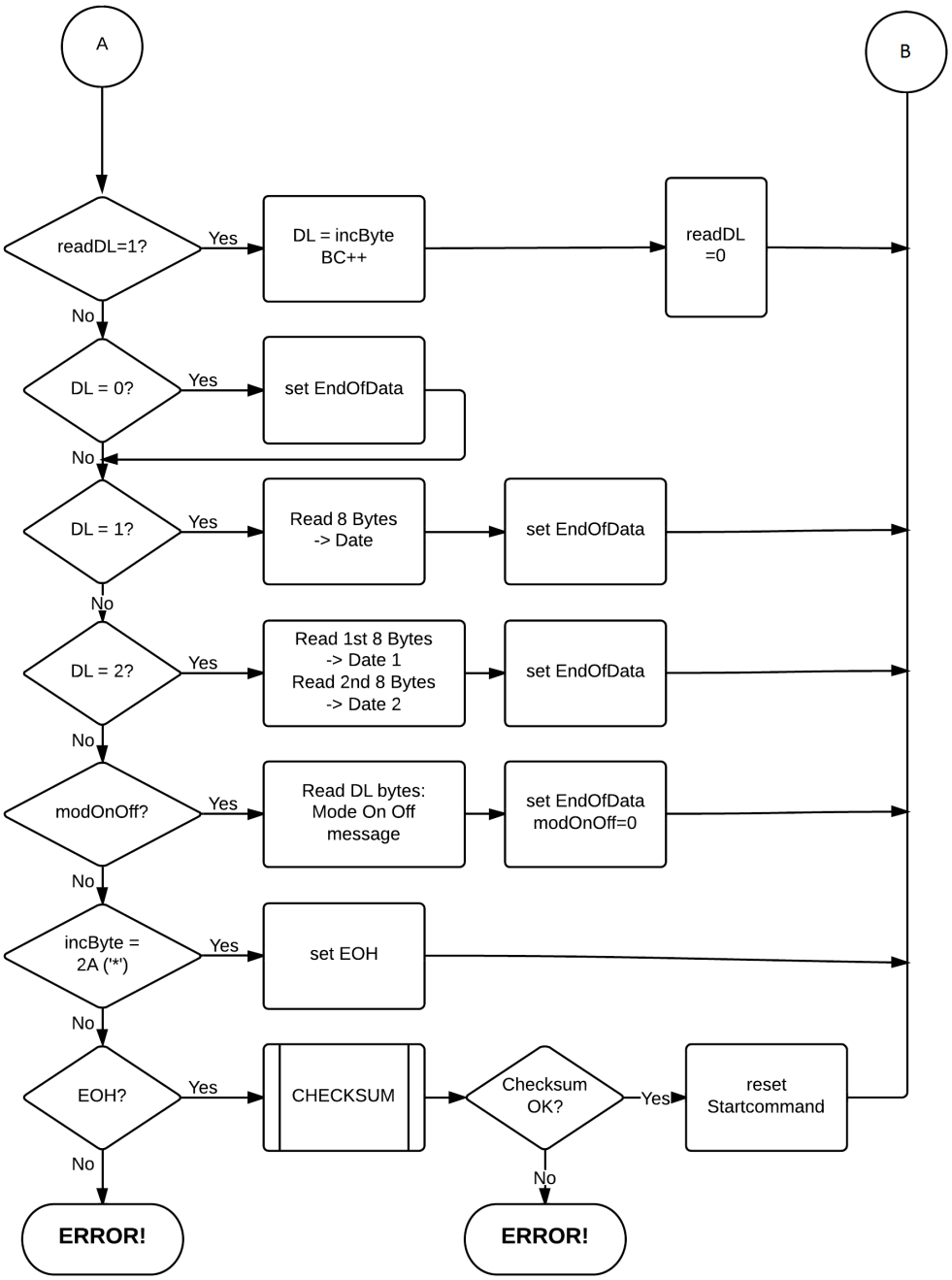


Figure 4.7: Read command (binary protocol), part 2

This new byte can run through the program to the readB2 check. Since readB2 was set, it will pass this check and the program enters the ID complement part. The incoming byte should be the complement of the previously saved ID. At first, the incoming byte will be saved in the memory as IDcompl. When this is done a check is ran to see if the ID and ID complement match. This is done by calling a small function idCheck():

```
int idCheck(BaseSeqStream *bss, uint8_t id, uint8_t idCompl){
int idTest = 0xFF-(int)id; //calculate complement in hex
int ret=0;
if ((int)idCompl==idTest){ret=1;}
return ret;
}
```

When this test fails, an error occurred during the sending process. This is an error and should be transferred to the BBB/Pi. Some more information about this Wrong Command Error will follow at the end of this paragraph.

When the ID and ID complement pass the test, the readDL flag is set, and readB2 is cleared. Now a new byte is read.

By setting the readDL the next incoming byte will go in the datalength part of the code. This because the fourth byte of the incoming message represents the length of the data that still has to come. The incoming byte will be saved in memory. At last the readDL flag is cleared.

After setting the datalength, there are four possible solutions. Either the datalength is 0, 1, 2 or more than 2. This indicates the four possibilities for the bytes that will arrive after this point.

The first possibility, dataLength = 0, means that there will be no actual data or dates attached to this message. The message is a bare command, error or message ok. When this is the case the EndOfData flag is set immediately.

The second possibility, dataLength = 1, indicates that there will follow one date/time structure. This is an 8 byte structure (hh:mm:ss:hh, dd/mm/yyyy) that has to be read. It will be saved to the memory as date1. When the complete date arrived at the STM32F3, the EndOfData flag can be set.

When dataLength = 2 it announces that two dates will follow. This means 16 bytes will follow. These two dates will be saved to the memory as date1 and date2. When the complete date transfer is completed, EndOfData will be set.

The last option based on dataLength is when it is more than 2. This actually isn't used in the code, because it points at a mode On/Off command. This command was captured earlier when the ID was read and it was flagged with the modOnOff flag. This flag is used in the next possibility. If the command

is an actual mode On/Off command it will enter a reading action specific for this command. It will read all bytes that belong to the messages (the value of dataLength). When this process finished, EndOfData is set.

Because a counter is used in the reading method, the program knows exact which byte is arriving. When one of the possibilities above is handled, the data part of the incoming command is handled. When you take a look at the command structure, it is clear that now a '*' must arrive at the STM32F3. This is checked in the function. When this is the case, the eoh flag will be set. If not an error occurred and the wrongCommand() will be executed. If the incoming byte was 0x2A a new byte can be read.

After the '*' in the message, the checksum arrives. These 4 bytes will be read and saved. To check the message the checksum of the incoming message has to be calculated too. To do so, all incoming bytes were saved from the moment a '$' arrived until this point. Now the CRC-32 will be calculated. If the calculated checksum differs from the checksum that has arrived, an error has occurred in the sending process. If the checksums are equal the message was correct and further actions can be taken. The first thing to do is to let the sender, the BBB/Pi, know that the message arrived. The actions to be sure the BBB/Pi will be contacted, are made by the MessageOK() function. After this the actionId() function is ran:

```
void actionId(BaseSeqStream *bss,int IDin, byte modOO[],
        byte d1[], byte d2[], int *incMesOk, int* wrongData,
        int* wrongCommand){

if((IDin==0x10))modeOn(bss, modOO);

        if(IDin==0x99)*incMesOk=1;
        if((IDin==0x14))modeOff(bss, modOO);
        if(IDin==0x86)*wrongCommand=1;
        if(IDin==0x88)*wrongData=1;
        //chprintf(bss,"%c",(char)0xdd);
        //chprintf(bss,"%c", (char) *incMesOk);
        if((IDin!=0x10)&&(IDin!=0x14)&&(IDin!=0x99)&&
                (IDin!=0x86)&&(IDin!=0x88)){

                createIncomingRequest(IDin, d1, d2);
        }
}
```

The ID of the incoming message will be checked and then a new function will be called or a flag will be set. The functions that are used here will be explained in the next chapter.

4.3.4 Communication Structures and Functions

In order to get a clear code, new structures were created. These files are the ”error.c”, ”mode.c”, ”sensor.c” and ”incomingRequest.c”. The function of these files and methods will be explained now.

4.3.4.1 Error

This is used as a buffer for the errors that have appeared while running. When an error has happened it can simply be added to the waiting queue by calling the addError() function. When an error has to be taken from the queue, the getError() function can be used.

4.3.4.2 Sensor

The first thing that is done in the sensor file is a definition of a new struct, it is called SensorFunction:

```
typedef struct SensorFunction {
        uint8_t sensorID;
        int timeInterval[MAX_MODES_PER_SENSOR];
        int modes[MAX_MODES_PER_SENSOR];
        int broadcast[MAX_MODES_PER_SENSOR];
        //1 for broadcast, 0 if not
        int usedInModes;
        //defines how many modes use this function
} SensorFunction;
```

This struct holds every setting that can be requested by the base station from any sensor. It remembers the number of modes the sensor is used in, which modes broadcast, the time interval for collecting new data. There are functions to add a new running function, to delete one, to delete a mode and to check if a sensor is running already.

4.3.4.3 Mode

This file contains all running modes. In general it contains some arrays that remember the mode ID, which sensors are used for which mode, the number of running modes, etc. The two biggest functions to add or delete modes from the arrays are also located in this file. When a mode is added with the modeOn() function it will check if the requested sensor function is already running. When it is not running it will be started. Otherwise the function will add an extra mode to the sensorfunction data. When a mode has to stop, this can be done with the modeOff() function, all sensors in the selected mode will be stopped, or when it is running in multiple modes, the mode will be deleted from the sensor data. The

local arrays will also be changed, because an element will be deleted somewhere in the array.

4.3.4.4 Incoming Request

This is a new defined structure. Every incoming request/command from the BBB/Pi has a number of parameters. Those will be transferred to the incoming request. This incoming request can be seen as a representation of the protocol message in the STM32F3. When a new message arrives that requests data, a new incoming request is generated. This incoming request holds the dates and the ID. Now you can call these dates and the ID from every file. This is done by use of the correct function.

4.3.5 Sending Data to the Raspberry Pi/BeagleBone Black

On the sending side of the STM32F3 and ChibiOS there are two different message types. The STM32F3 can not only send data to the Raspberry Pi (or BeagleBone Black), but also some commands. These commands can be errors or the messageOK acknowledgement. An image that represents the global working principle of sending something to the Rapsberry Pi (or BeagleBone Black) can be found in Figure 4.9 on the following page.

Figure 4.9 on the next page show the flowchart for the code that is located in the sending thread description in the main file. This part of code decides what will be sent. To do this the priority of the messages should be met.

The first thing that is done, is check if there is an error available. When there is an error available, the ID will be collected from the queue, and the error will be removed from the queue. This ID will be passed to the correct sending function. If there is no error, the MessageOK will be checked.

If there is need to send an outgoing MessageOK, the ID of MessageOK will be passed to the send function. This ID is ”99h”. When there is no need to send a MessageOK, the next parameter to check is DataFinished.

If a stream of data is finished, the DataFinished command will be sent. To do this, the sending method needs the DataFinished ID (=90h). At last, when DataFinished isn't activated, there will be a check for available data.

If there is available data, the data will be collected together with the other needed structures (ID, Date). And eventually, the data will be sent. When there is no data available at this moment, the cycle restarts.

Figure 4.10 on page 80 shows some more about the Send structure in Figure 4.9 on the next page. The sending structure will wait for a confirmation of arrival, good or bad, and then continue as a result of this. When there is no MessageOK

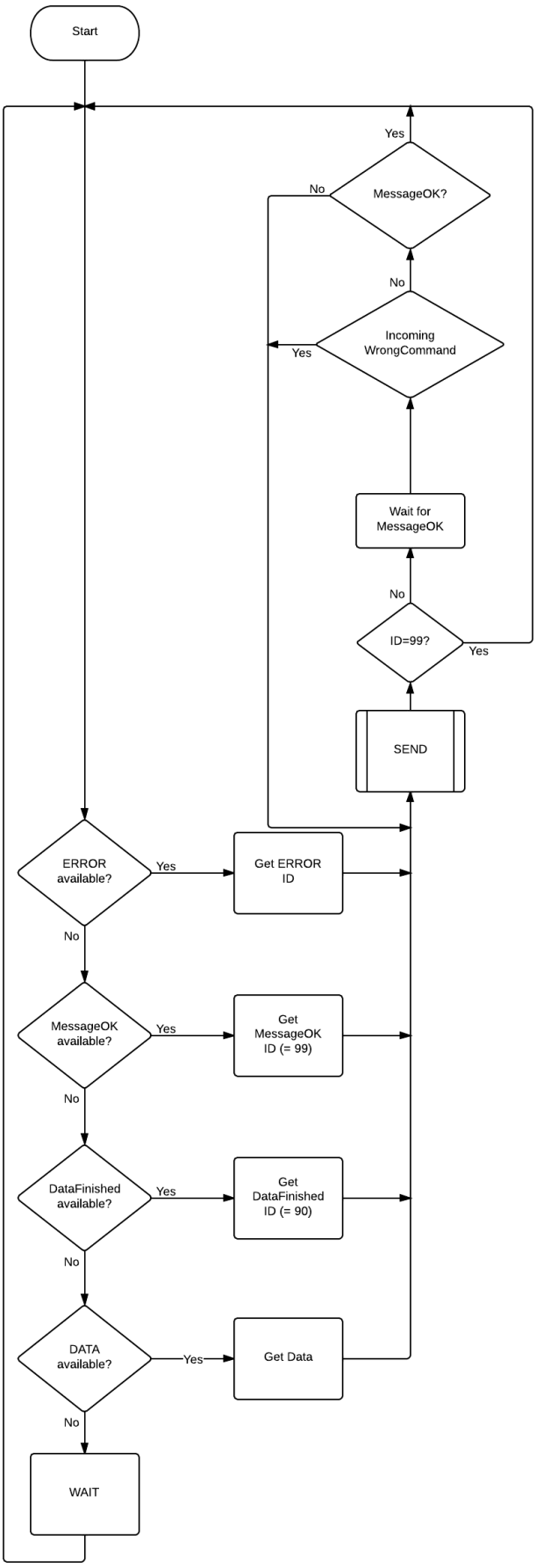


Figure 4.9: Main send action

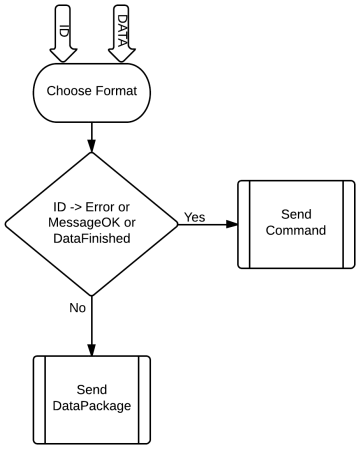


Figure 4.10: 2 possible send methods

before timeout, or there is an incoming WrongCommand, the message will be resent. When there is a MessageOK, the algorithm continues and can send another message.

There are two different sending functions implemented. One of them sends a command. The other is the data. The mode choice for one or the other function is made on the simple fact: "Do I need to send a command or a data structure?" This can be done by using the ID and passing it to an extra function. The implementation however uses immediately the correct sending function when the queues or flags are checked to decide what has to be sent. This can be changed later to simplify and shorten the code.

4.3.5.1 Send Command

When a command has to be sent a basic form can be used. Every command has the same structure. It can be represented as shown in Figure 4.11:

1st byte	**2nd byte**	**3rd byte**	**4th byte**	**5th byte**	**6th to 10th byte**
"$"	ID	ID complement	Datalength	"*"	4 CRC-32 bytes

Figure 4.11: Command structure

The method for sending this types of messages can be represented by Figure 4.12 on the next page.

It is a simple byte by byte sending algorithm. The first step is sending the '$', after this the incoming ID and its complement can be sent. To do so, the

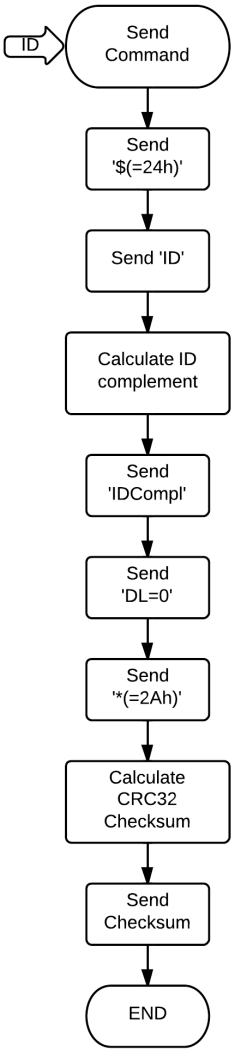


Figure 4.12: Send Command

complement has to be calculated first. The fourth byte that will be send is a 00h. This because the datalength of all commands is 0. After this the EOH will be sent. This is the '*'. All previously sent bytes were stored somewhere in the memory, because the CRC-32 checksum must be calculated. When this is done, the checksum will be sent. This happens byte by byte too. To achieve this a special method writeCheck() is used.

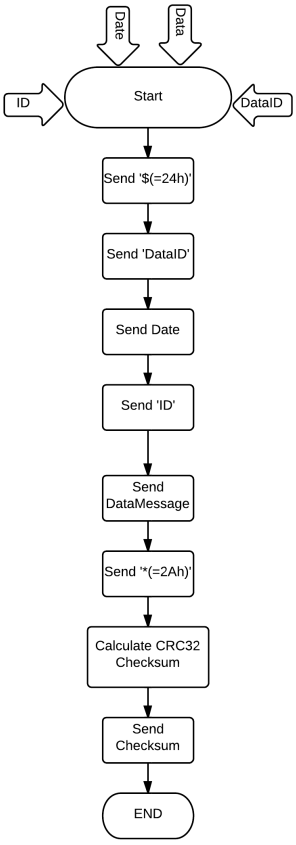


Figure 4.13: Send Data

4.3.5.2 Send Data

When data has to be sent, a different function is used. This although, does not differ that much from the sendCommand() function. The flowchart of the sendData() is shown in figure 4.13.

The structure of this flowchart can be related to the form of a data message, limited to one ID. This is depicted in Figure 4.14.

”$”	Data ID	Date & time	ID	Data	”∗”	4 CRC-32 bytes

Figure 4.14: Structure of data messages with 1 ID

This limitation is implemented at this moment because a bigger combination

message has some drawbacks. The main drawback has to do with the timing. When the data is requested for a sensor, and the data capture rate of the sensors are different, there will be dates/times where one sensor has data, and the other doesn't. This means that the data length would change constantly. When the data length has to stay the same, there will be a lot of duplicate data. This due to the resending of data when there is no new entry available. Another thing that lead to this decision is the combination request. All data request combined in this combination can be asked separately. This will not cause that much of time difference, so it is still acceptable to do so.

To run the send data command, it needs the following: ID, Date, DataID (broadcast or not) and the actual data. After entering the sendData() function, a '$' is send. The second byte that will be sent is the DataID. After this the date of the data in the package is sent. The next byte defines the ID of the data. When all these bytes are sent, the time has come to send the actual data. To make this easier, a fixed length for the datapackage will be used. The message will be sent byte by byte. When the total message is sent, a '*' will be added. The next step to get a successful send, is calculating the checksum. All previously sent bytes were saved in memory, and are now used to calculate the checksum. After calculating, it will be sent.

When, in the further development of the buoy, there is need for a combination message, it can easily be implemented by running the ID send, and Data send in a loop.

4.3.6 Communication Test

4.3.6.1 Discussion

In order to be able to test the communication of the STM32F3 with the BBB/Pi, a test file was made. This file simulates the SD card en sensors. This is done by saving some pre-defined messages in the memory of the STM32F3, together with some dates. When a request for a certain sensor arrives at the STM32F3, the controlling thread will act accordingly. A flowchart of what will happen is depicted in Figure 4.15 on the following page.

At first, there is a check to see if there is a new request. When there is, a next check is done. Is the previously made packet already sent? If not, this the function will wait until it is sent. As a next step, the dates and ID of the new request will be fetched. When this is done, the new request can be deleted from the queue. To continue, a separation must be made between two possibilities: one or two dates.

- Request with one date:

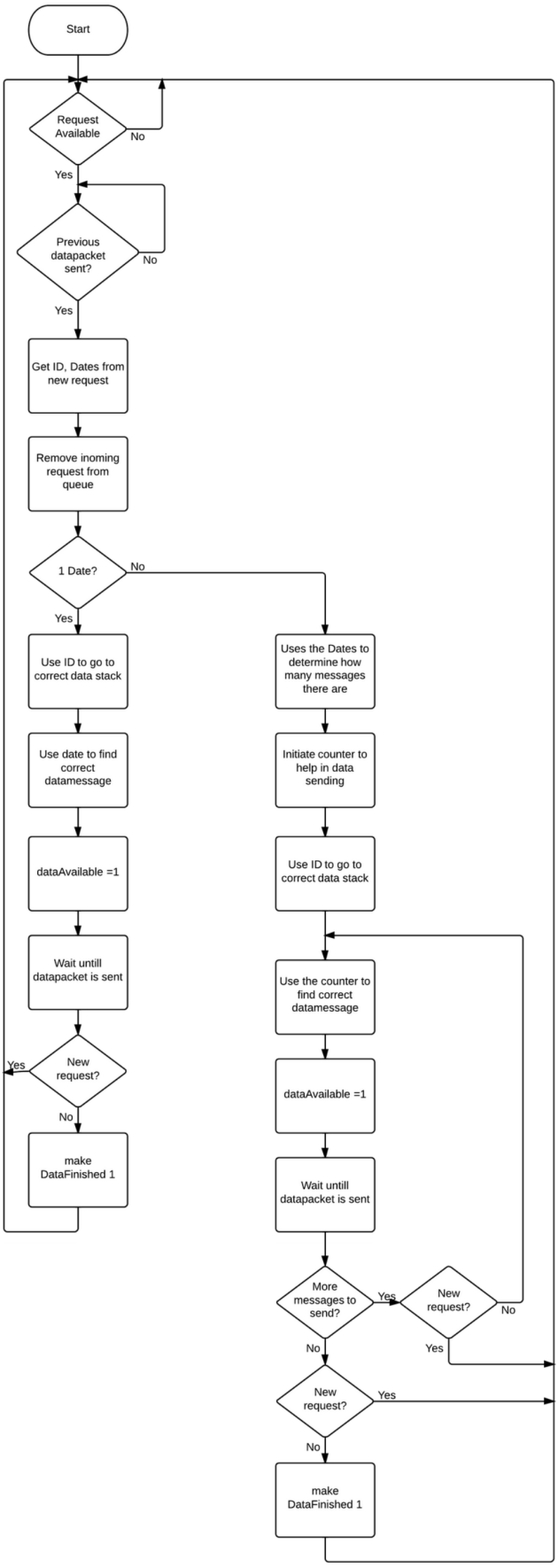


Figure 4.15: Flowchart for test

When there is only one date in the request, the message that eventually will be sent is the one with the first timestamp after the requested date. To do this, the ID is used to go to the correct data stack. With the date, the correct message can be searched. When this message is found, the dataAvailable flag is set. When this data is sent, a check will be made for a new request. The DataFinished flag is made one when there is no new request.

- Request with two dates:

  When there are two dates for the incoming request, multiple messages will have to be sent. The first thing that is done is determine how many messages. When this number is known, a counter will be set. After setting this counter the correct data stack will be selected. The correct datamessage and its date/timestamp will be taken from the stack by use of the counter. When this is done, the dataAvailable flag will be set. This means that the datapacket will be sent. When there are more messages, the counter is not finished, and there is no new request, the program will fetch another datapacket and date, and will send it too. When all messages are sent, there is a check for a new request. When there is no new request, the DataFinished flag is made 1 to send the DataFinished command. After this, the requested datacommand is fully handled.

Selecting the asked datapackage happens a little bit different in the implemented code. Because there is a previously defined array of dates, that is the same for all sensors, the date selection can be done up front. This is done by finding the location of the incoming date in the dates-array. The array elements that follows on the incoming date will be selected and used to get the correct data.

4.3.6.2 Further Development

The structure of the test file can be used in the further development of the software for the STM32F3. The main controlling file has to do similar actions, this to collect the requested data from the SD card. The biggest difference between the test and the real controlling thread will be the location where they get their data. The real controller will get it from the SD card thread via a local buffer, where the test can directly reach to the local array that contains the data.

The second difference will be the date selection. The incoming date will have to be compared with all the saved dates in the file on the SD card. This will take some time because the file on the SD card, a comma separated values (.csv) file, must be read first. When this is done, all messages that pass the date test must

be transferred to the sending thread. To do this, all the files must be copied from the SD to the local memory.

Another change that has to be implemented is a better algorithm to find a date in a large file. The one used in the test is just a basic solution that will not be the fastest in larger files. Some more research must be done on this algorithm.

4.3.7 Preparing Compilation

In order to get all the functions of the newly defined h-files, all the c-files describing those headers must be included in the project. This must be done by changing the Makefile. To do so, the files must be added under "CSRC":

```
# C sources that can be compiled in ARM or THUMB
# mode depending on the global setting.
CSRC = $(PORTSRC) \
       $(KERNSRC) \
       $(TESTSRC) \
       $(HALSRC) \
       $(PLATFORMSRC) \
       $(BOARDSRC) \
       $(CHIBIOS)/os/various/shell.c \
       $(CHIBIOS)/os/various/chprintf.c \
        main.c \
        sendCommand.c \
        ...\
        testCode.c
```

4.4 Testing

The implemented code was tested by using an STM32F3DISCOVERY board that was connected to a pc via a mini USB to USB cable. The Discovery board has two mini USB ports. One of them is connected to the ST-Link programmer, the other one is the USER USB. This USER USB can be usad as a virtual COM port.

4.4.1 Programming the STM32F3

To program the STM32F3, it has to be connected to the PC from the mini USB port in at the middle of the board. The port is labeled with "ST-Link".

The programming is done under a Linux environment. The first thing to do is to navigate to the folder with the c-files. In this case it is the /ChibiOS_2.6.3/testhal/STM32F30x/Buoy folder. When you are located in the folder you can

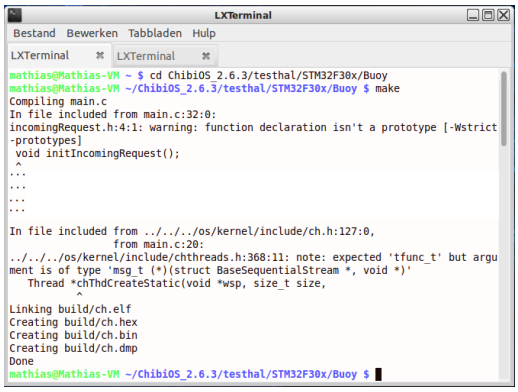


Figure 4.16: Make command

start editing your code. This can be done by a text editor in the terminal, Nano for example, or by using an editor program. During the development of the code the editor gedit was used. When the wanted changes are made to the code, the first action to get it downloaded to the STM32F3 is a make, as can be seen in Figure 4.16.

When this make command has executed without errors, which means that there were no errors during the compilation of the files, the binaries are ready to flash the STM32F3. The warnings that are shown must be checked too. Most of them do not require actions, but sometimes there is need to change something.When this is done, everything is ready to download the code to the STM32F3. Two small scripts were written to do so. The first step that had to be done after connecting the STM32F3 to a USB poort was the ”stm32f3” command, the batch file executes the following:

```
#!/bin/bash
#openocd STM32F3 scripts
openocd -f /usr/local/share/openocd/scripts/
        board/stmf3discovery.cfg
```

It lets openocd, Open On-Chip Debugger [Sou], run with the correct board script. OpenOCD will make a connection to the STM32F3 and check some parameters. The command execution can be found in Figure 4.17 on the following page.

When the command stopped processing, after the end and breakpoints appeared at the screen, the command can be paused by using Ctrl-z. Now the STM32F3 is ready to be flashed. This is done by entering the ”stmflash” command, this command calls the next code:

```
#!/bin/bash
#Download to STM32F3 memory
```

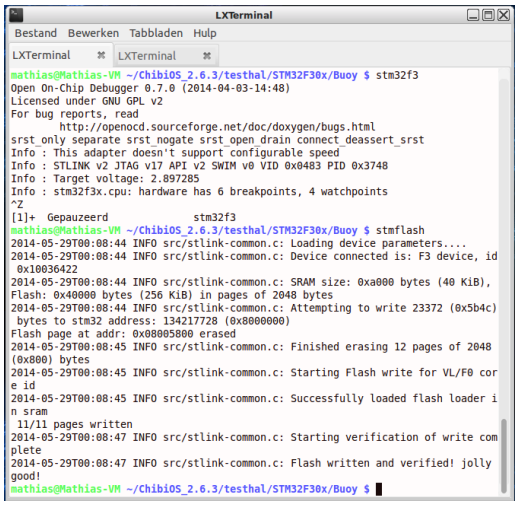


Figure 4.17: STM32F3 scripts

```
st-flash write build/ch.bin 0x8000000
```

After finishing this command, at the end of Figure 4.17, the STM32F3 is programmed and ready for use.

4.4.2 Serial Connection Logging

In order to check the output of the STM32F3 a serial logger has to be used on the PC. Contrary to the programming, the testing is done in a Windows environment. The first thing that has to be done to make this possible is to install the latest ST-Link drivers [ST] from the ST website. This has to be done to make sure that the STM32F3DISCOVERY board will be recognized as a virtual com port. After this is done, a serial port reader can be used to check the output of the STM32F3.

During this test CoolTerm v1.4.3 [Rog14] was used. Before making a connection to the STM32F3, all settings have to be set to the correct values. This can be found in Figure 4.18 on the following page.

When the settings are applied, the testing can be started. To get a clear view of the incoming bytes, use the view Hex button in the top row. Selecting this gives all incoming bytes, and the ASCII characters they represent in a smaller window on the right side. When this is done you can connect, by clicking on the connect option.

When the connection is made, a window is needed to provide input to this STM32F3. To get this window, go to the Configuration menu and select the ”Send String” option, or use the Ctrl-t shortcut. This action results in an input

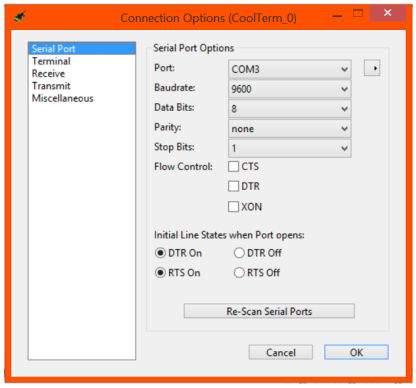


Figure 4.18: CoolTerm: settings

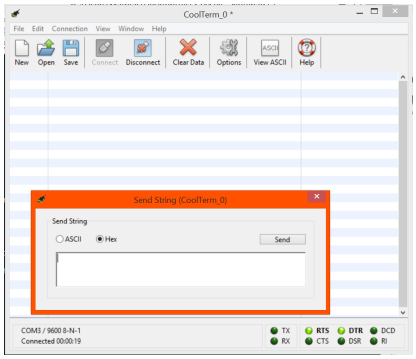


Figure 4.19: Coolterm: ready to send data

window. After selecting hex, the window is ready to send bytes to the STM32F3. This can be seen in Figure 4.19.

4.4.3 Communication

In order to see of the implemented code works, the following test setup was used. The STM32F3 was connected to a laptop where CoolTerm was running. The connection was made with a mini-USB to USB cable (male to male) from the USB USER port on the STM32F3 to one of the USB ports on the PC. This setup can be found in Figure 4.20 on the following page.

In order to test the communication, several requests were simulated/tested. We will discuss some of these tests in detail to explain what has been simulated.

- **Normal Command (Mode On):**

  This situation, depicted as a diagram in Figure 4.21 on the next page, is one that will most of the times appear right after the startup.

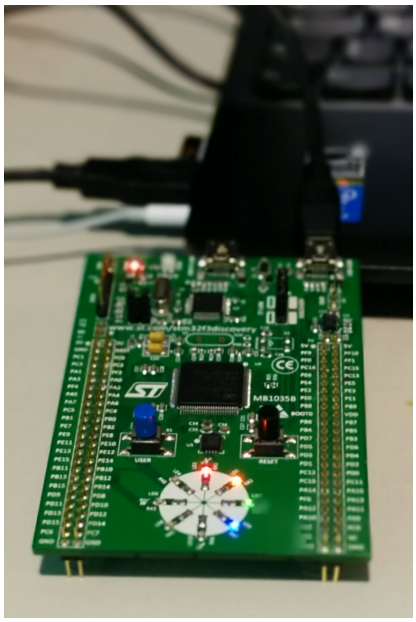


Figure 4.20: Setup used to test communication of STM32F3DISCOVERY

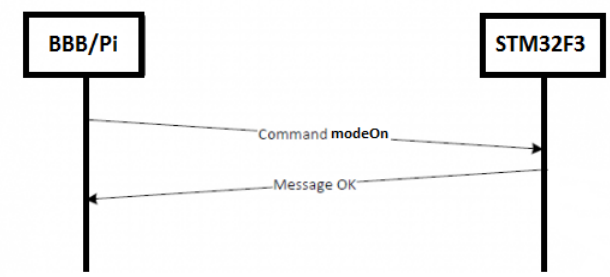


Figure 4.21: Diagram of test situation 1

A ModeOn command will be sent to start mode 1. It will request 3 sensor functions, for simplicity called A1, A2 and A3. It requests no broadcast and the data should be captured every minute. Te requested protocoltype is 0. This leads to the fillowing binary message:

Name	**SOH**	**ID**	**IDc**	**DatL**	**Data**	**EOH**	**Check**
Mode On Wrong Check	24	10	EF	0B	...	2A	99B2699C

Where the data part is the following:

Mode ID	**#sensors**	**Seensor ID**	**Timer ΔT**	**Protocol**
01	03	A1A2A3	0000010000	00

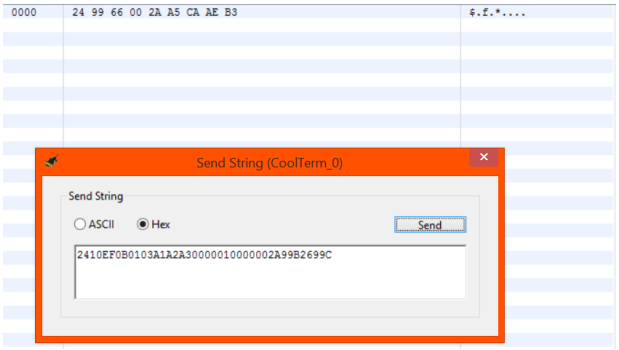


Figure 4.22: Test situation 1

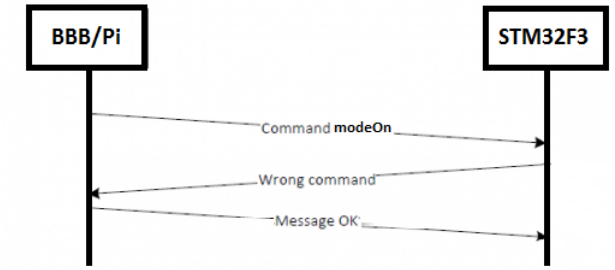


Figure 4.23: Diagram of test situation 2

Figure 4.22 shows the reply of the STM32F3.

The STM32F3 answers with the following message:

Name	SOH	ID	IDc	DatL	EOH	Check
MessageOK	24	99	66	00	2A	A5CAAEB3

This is the MessageOK command. This means the request for the ModeOn has been read by the STM32F3 and there were no mistakes.

- **Wrong Command:**

  This situation is similar to situation 1 and is shown in Figure 4.23.

  The goal was to send the same message/request. But an error has occured and the last number of the checksum has changed. This leads to the following message:

Name	SOH	ID	IDc	DatL	Data	EOH	Check
Mode On Wrong Check	24	10	EF	0B	...	2A	99B2699**0**

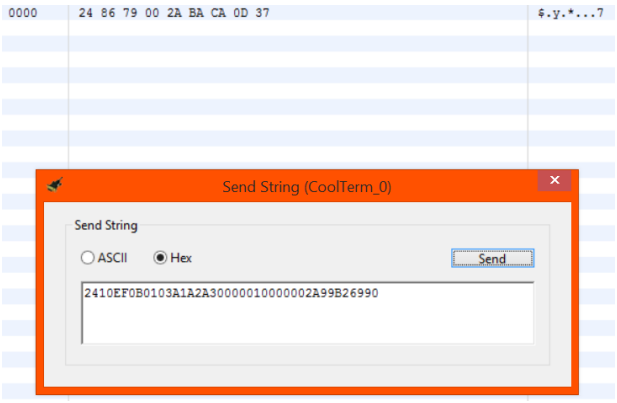


Figure 4.24: Test situation 2 (1)

Where the data part is the following:

Mode ID	#sensors	Seensor ID	Timer ΔT	Protocol
01	03	A1A2A3	0000010000	00

Due to this mistake, the STM32F3 answers with a WrongCommand message: case the PC replies with a MessageOK.

Name	SOH	ID	IDc	DatL	EOH	Check
Wrong Command	24	86	79	00	2A	BACA0D37

This is depicted in Figure 4.24.

This WrongCommand error will be sent by the STM32F3 until the BBB/Pi, or in this case the PC replies with a MessageOK.

Name	SOH	ID	IDc	DatL	EOH	Check
MessageOK	24	99	66	00	2A	A5CAAEB3

These last actions are depicted in Figure 4.25 on the next page.

- **Data Request with one date:**

  The third situation, depicted in Figure 4.26 on the following page, that will be explained is a data request.

  The BBB/Pi asks for data from the CTD about the water depth. This on a certain day/time: 12h00 24/05/2014. In order to fulfil this request the available data will be checked. However, this timestamp is not available in

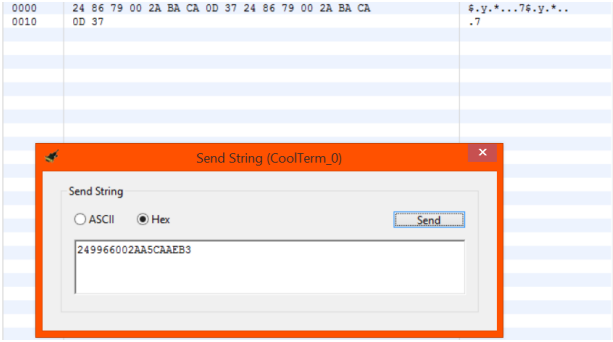


Figure 4.25: Test situation 2 (2)

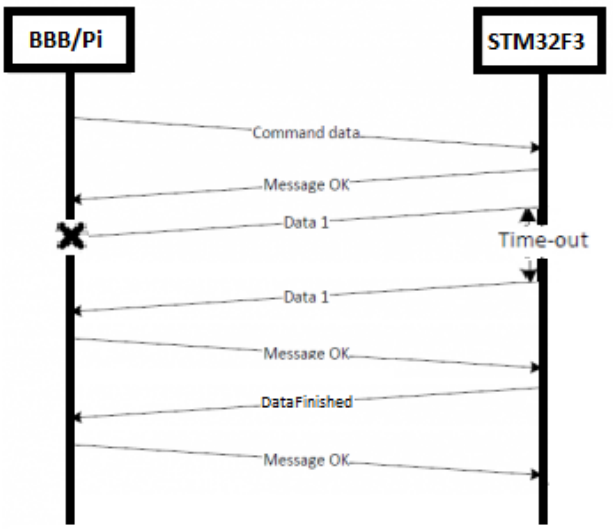


Figure 4.26: Diagram of test situation 3

the data, as a result of this, the STM32F3 will sent the first datainput after this timestamp.

The binary message is the following:

Name	SOH	ID	IDc	DatL	Date	EOH	Check
Water depth data request	24	41	BE	01	12000000 24052014	2A	79BB E9A7

The reply of the STM32F3 can be found in Figure 4.27 on the next page.

It sends a MessageOK first (the first nine bytes) to acknowledge the arrival of the message, and afterwards replies with the data:

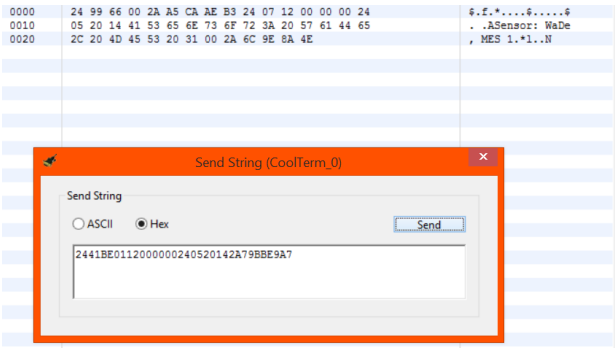


Figure 4.27: Test situation 3 (1)

Name	SOH	Data ID	Date & Time	Sensor ID	Data	EOH	Check
Water Depth Data	24	07	12000000 24052014	41	53656E736F 723A205761 44652C204D 4553203100	2A	6C9E 8A4E

The data packet starts with a '$' and is followed by the dataID. In this case it is 7, what means the data was stored (dataId= 0 means broadcast). The next eight bytes show the date of the data. The following byte shows which sensor captured the following data, in this case the CTD and it is data about the water depth (which has ID 41). The fifth chunk of the message contains the actual data. In this test it contains "Sensor: WaDe, MES 1 " in ASCII.

The message itself will be sent untill there is a MessageOK that confirms the arrival of the packet. This is depicted in Figure 4.28 on the following page, where the MessageOK message is shown in the send string window.

- **Data Request with two dates:**

  The fourth situation, shown in Figure 4.29 on the next page, that will be explained request the following data: the wind direction between two dates/times. These dates are 13h00 24/05/2014 and 10u00 25/05/2014. This leads to the following binary message:

Name	SOH	ID	IDc	DatL	Date 1	Date 2	EOH	Check
Wind direction data	24	36	C9	02	13000000 24052014	10000000 25052014	2A	EEB3 30AE

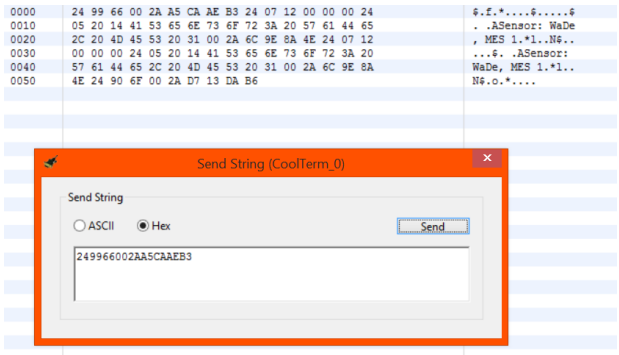


Figure 4.28: Test situation 3 (2)

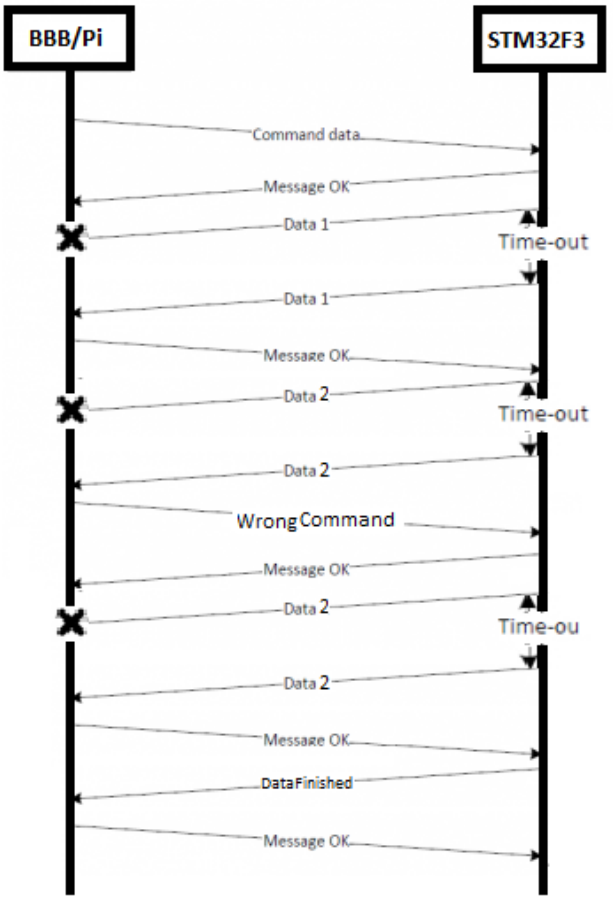


Figure 4.29: Diagram of test situation 4

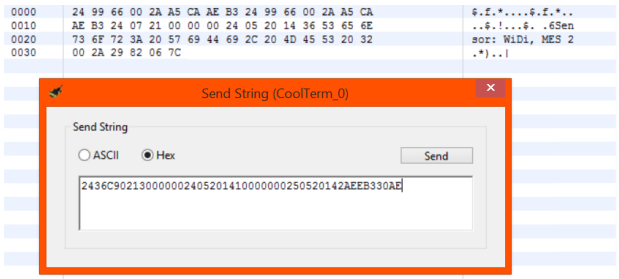


Figure 4.30: Test situation 4 (1)

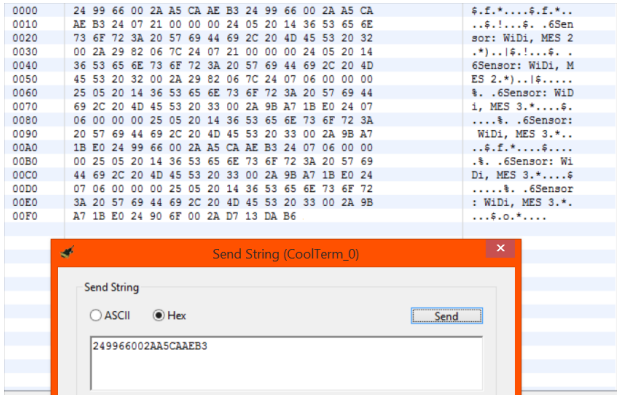


Figure 4.31: Test situation 4 (2)

The internally saved data contains two timestamps between these dates. The STM32F3 should reply with the two date messages. Figure 4.30 shows that the STM32F3 replies with a MessageOK, the second one in the picture. The first one is one from a previous scenario. After this MessageOk, the first available datapacket (”Sensor: WiDi Mes2”) is sent.

The first datapacket is sent two times before a MessageOK was replied. After this the second datapacket (”Sensor: WiDi Mes3”) is sent. The packet was sent two times, but then a WrongData command was sent. The STM32F3 replies with a MessageOK and restarts sending the second dataPacket. After two more tries the acknowledgement of arrival arrives at the STM32F3, which means the DataFinished packet must be sent. This message is also confirmed with a final MessageOK. These messages are shown in Figure 4.31.

- **Normal Command (Mode Off):**

  This last situation, depicted as a diagram in Figure 4.32 on the following page, shows an example of finishing a running mode.

  It represents a sent ModeOff command. Mode 1 will be stopped:

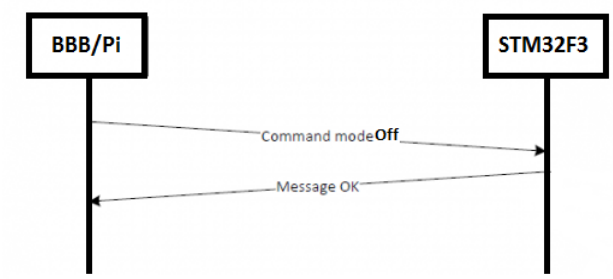


Figure 4.32: Diagram of test situation 5

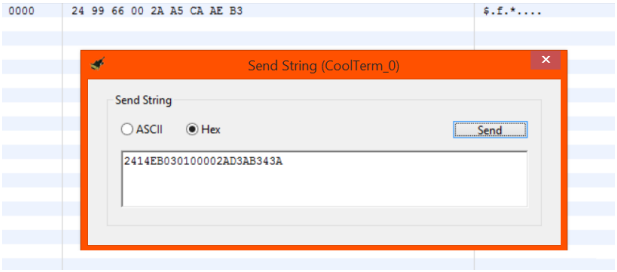


Figure 4.33: Test situation 5

Name	SOH	ID	IDc	DatL	Mode ID	2x 00h	EOH	Check
Mode 1 Off	24	14	EB	03	01	0000	2A	D3AB 343A

The STM32F3 answers to this message with a MessageOK, this is depicted in Figure 4.33.

4.5 Conclusion

This chapter introduced a method to fulfil the communication needs of the STM32F3. It also implements the described protocol and its priorities. This method is based on a USB-CDC connection and is supported by ChibiOS. The communication unit is a combination of three threads on the STM32F3.

A receiving thread that reads the incoming data and creates request for an incoming message. Or flags the arrival of a MessageOK or WrongCommand message. Some checks are ran during the description to discover eventual errors, if they occur they will be passed to the error queue. If everything runs according to plan, a notification will be sent to the sending thread, in order to get an outgoing MessageOK.

The second thread is the controlling thread. This threads executes the incoming requests, prepares data to be sent, etc. Now this thread is limited to a test code. It is able to send some predefined messages.

The sending thread will send the messages to the BBB/Pi. This is done in the order of the priorities. At every point, it will first check for errors, or an outgoing MessageOK before it will send data or DataFinished.

The described structrures were tested too. This test was executed on a manual base to be sure the code reacts to errors too. The test showed no errors, so the implemented code works as it should be.

Chapter 5

Conclusions

5.1 Discussion

Four months of work on the Environmental/regatta buoy project have fulfilled my expectations. The work was a mix between research and implementation. This research, however, took substantially more time than was expected at the beginning of the project. This was a result of the multiple problems interface with, programming for and testing the STM32F3. Initially, he software did not run on Windows 8.1 nor on a virtual Windows XP machine. At last, a Linux distribution was successfully used to develop for, upload to and run on the STM32F3. This development set up included a basic text editor, a compiler and the ST-Link tools. The next task that took more than expected was the learning ChibiOS. This RTOS is relatively new, which means that there are few code examples available. After some trial and errors with the available demos, ChibiOS working principle became clear.

Once these obstacles were surpassed, the communication on the slave unit side was developed. First, the protocol was upgraded together with Laurens Allart. The protocol defined by Hendrik Verschelde [Hen14] and Jeroen Vervenne [Jer14] was improved with a more robust message structure – a new CRC-32 checksum field – and new commands. Then, this protocol was implemented on the slave control unit to process the incoming master requests, *e.g.*, mode, sensor configuration and data request messages, and send back messages with the requested information or detected errors.

The next goal was to develop the GNSS sensor data acquisition and the data storage in the SD card. The writing into the SD card was successfully tested with Jeroen Vervenne's code [Jer14] and the GNSS sensor was configured. The next step was the assembling of the GNSS sensor, SD card and slave control unit

together with the PCB available at LSA. However, the PCB was damaged and a new development board was ordered, but there was no time for implementations.

The implementation of the protocol on the slave control unit was done, but the implementation data acquisition was not achieved due to the lack of time and the problems with the PCB.

5.2 Future Developments

5.2.1 Data Acquisition

As mentioned in the previous section, the implementation of the sensor data acquisition is not done. Threads should be added to the STM32F3 software to control the data collection. There are structures implemented in the code for this purpose. Test were ran with the GNSS during this semester, that were based on Jeroen Vervenne's [Jer14] tests. Jeroen Vervenne's implementation can be used as a basic structure for all the other sensors that need to be implemented.

5.2.2 SD Card Data Storage and Retrieval

The SD card structure is present in the project [Jer14] but was not yet implemented because of the damaged PCB. This should be added to the communication code to allow data storage and retrieval.

5.2.3 Testing

The last thing that should be done when all functions are implemented is testing. The tests run during the implementation of the communication were limited due to the absence of the sensors. Once sensors are integrated, the complete set of functional tests can be executed.

Bibliography

[Bea14] Beagleboard. BeagleBone Black, 2014. [cited at p. 5, 17]

[Ben13] Bennet Möller, Emil Konrad Wlazlo, Mateusz Tarkowski and Toon Van den Bleeken . Wiki, Autonomous Environmental Buoy/Regatta Beacon, 2013. [cited at p. iv, 16, 22, 23, 24]

[Car09] Carrilo Garcier M., Popp H. J., Toma D. M., Stützle M.; Escola Ppolitècnica Superior d'Enginyeria de Vilanova i la Geltrú and Universitat Politècnica de Catalunya. Autonomous Meteorological Buoy, 2009. [cited at p. iv, 13, 14]

[Chi11] ChibiOS. Threads, 2011. [cited at p. 36]

[Chi14a] ChibiOS. ChibiOS/RT Homepage, 2014. [cited at p. v, 33, 36]

[Chi14b] ChibiOS. Serial over USB Driver, 2014. [cited at p. v, 68]

[Chi14c] ChibiOS. USB driver, 2014. [cited at p. v, 66]

[Chr14] Chris Borrelli. IEEE 802.3 Cyclic Redundancy Check, 2014. [cited at p. 57]

[Cri10] Cristina Albaladejo, Pedro Sánchez, Andrés Iborra, Fulgencio Soto, Juan A. López and Roque Torres; Technical University of Cartagena. Wireless Sensor Networks for Oceanographic Monitoring: A Systematic Review, 2010. [cited at p. iv, 8, 9]

[Dat14] Data Buoy Cooperation Panel. Data Buoy Types, 2014. [cited at p. iv, 3]

[Enc14] Encyclopaedia Britannica. Protocol, 2014. [cited at p. 38]

[Git14] GitHub. stm32 discovery line linux programmer, 2014. [cited at p. 38]

[GNU14] GNU ARM Eclipse. Toolchain install, 2014. [cited at p. 38]

[Hen14] Hendrik Verschelde. Environmental/Regatta Buoy: Telemetry and Configuration, 2014. [cited at p. 99]

[Jer14] Jeroen Vervenne. Environmental/Regatta Buoy: Data logging, 2014. [cited at p. iv, v, 28, 39, 42, 44, 45, 46, 59, 60, 61, 99, 100]

[Jor14] Jorgen Hansen and Duane Abata. European Project Semester, 2014. [cited at p. iv]

[Kyu11] Kyung Woon Lee, Ui-seok Jeong, Joon Young Yang, Ho Kyung Jun, Jung Ho Park; Korea University and Shinyang Techology. Implementation of Embedded System for Autonomous Buoy, 2011. [cited at p. iv, 10, 11]

[Lis14] Lisa Emery, Richard Smith, Rebecca McQuary, Bill Hughes, David Taylor; Technology Solutions Group, Qinetiq North America. Autonomous River Drifting Buoys: Applications and Improvements, 2014. [cited at p. iv, 12]

[Mar11] Martino Migliavacca. ChibiOS components, 2011. [cited at p. iv, 35]

[Nov05] Novatel. SUPERSTAR II Firmware Reference Manual, 2005. [cited at p. 22]

[Nov09] Novatel. SUPERSTAR II manual, 2009. [cited at p. iv, 19, 20, 21]

[Par12] Parallax. micro-SD Card Adapter, 2012. [cited at p. v, 61]

[Pau14] Paul Pitteljon. Ambleteuse Buoy, 2014. [cited at p. iv, 2]

[Pep13] Peppermint media group. peppermint FOUR, 2013. [cited at p. v, 37]

[Ras12] Raspbian. Raspbian OS, 2012. [cited at p. 17]

[Ras14] Raspberry Pi Foundation. Raspberry Pi, 2014. [cited at p. 5, 17]

[Rog14] Roger Meier. CoolTerm, 2014. [cited at p. 88]

[Sar01] Sarah Elengorn. Sailing regatta copa mexico, 2001. [cited at p. iv, 8]

[Sou] SourceForge. Openocd, . [cited at p. 87]

[ST] ST. ST-Link Drivers, . [cited at p. 88]

[STM13] STMicroelectronics. STM32F30x DISCOVERY board manual, 2013. [cited at p. iv, 30, 31, 32]

[STM14] STMicroelectronics. STM32F3DISCOVERY: Discovery kit for STM32F303xx microcontrollers, 2014. [cited at p. 5]

[T. 10] T. J. Smyth, R. Fishwick, C. P. Gallienne, J. A. Stephens, A. J. Bale; Plymouth Maine Laboratory. Technology, Design and Operation of an Autonomous Buoy System in the Western English Channel, 2010. [cited at p. iv, 12, 13]

[vmw14] vmware. VMware Player Download, 2014. [cited at p. v, 37]

[Vol14] Volkssterrenwacht Urania. Map of earth surface, 2014. [cited at p. iv, 6]

[Wav14] Waveshare. Open32F3-D Standard, 2014. [cited at p. v, 60]

[Wik14a] Wikipedia. Computation of cyclic redundancy checks, 2014. [cited at p. 57]

[Wik14b] Wikipedia. CRC32, 2014. [cited at p. 57]

[Wik14c] Wikipedia. Generec defenition sensor, 2014. [cited at p. 18]

[Wik14d] Wikipedia. Raspberry Pi Image, 2014. [cited at p. iv, 17]

[Wik14e] Wikipedia. Thread defenition, 2014. [cited at p. 63]

[Xul11] Xulio Fernándze-Hermida, Carlos Durán-Neira, Manuel D. Lago-Reguera, Carlos Rodriguez-Alemparte, Fernando Martin-Rodriguez; University of Vigo. Hidroboya: An Autonomous Buoy for Real Time High Quality Sea and Continental Water Data Retrieval, 2011. [cited at p. iv, 11, 12]