David Fernández

Develement of calibration and limit checking modules for a satellite's ground control software

Aalto Universiy School of Electrical Engineering

Department of Radio Science and Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Otaniemi, Espoo, 31.08.2013

Thesis supervisor and instructor: D.Sc. (Tech.) Jaan Praks

Aalto University
School of Electrical Engineering

Abstract of the
Final Project

Author: David Fernández

Title: Development of calibration and limit checking modules for a satellite's ground control software

Date: 31.08.2013 **Language:** English **Number of pages:** xxx

School of Electrical Engineering

Deparment of Radio Science and Engineering

Professorship: Space Technology **Code:** S-92

Supervisor: D.Sc. (Tech.) Jaan Praks

Keywords: nanosatellite, ground station, calibration, limits

All our dreams can come true, if we have the courage to pursue them.

Walt Disney

Note: blablabla

Preface

Acknowledgements

Otaniemi, Espoo, August 2013.

David.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The relationship between human beings and space has existed since the beginning of time. It has always been a source of mystery, something we want to understand. More than 4000 years ago, the Egyptians and the Babylonians were influenced by the movements of the sun and the planets and, based on those movements, developed calendars for their crops. Later, the ancient Greeks developed the concept of *astronomy*, the science of the heavens. Subsequently, we can find philosophers such as Nicolaus Copernicus, Johannes Kepler, who explained the motion of the planets, and Galileo Galilei. In the 17th century, Sir Isaac Newton invented calculus, developed his law of gravitaion and performed important experiments in optics.

The technological advancements of the 20th century, specially accelerated by the World War II, made physical exploration of space become possible. This thesis is oriented towards one of those advancements, artificial satellites.

A *satellite* is a natural or artificial object moving around a bigger body. This motion is defined as an orbit, enabled by the dominant force of gravity from the bigger body. In early 1945, the United States started the Vanguard Rocket development to launch a satellite. However, after several failed launches, it was de Soviet Union who took the advantage by launching the Sputnik-1 (Figure 1.1) on October 4, 1957. During the Cold War, the space race between the Sovient Union and the United States was a hard fight which made space technology advance quite rapidly.

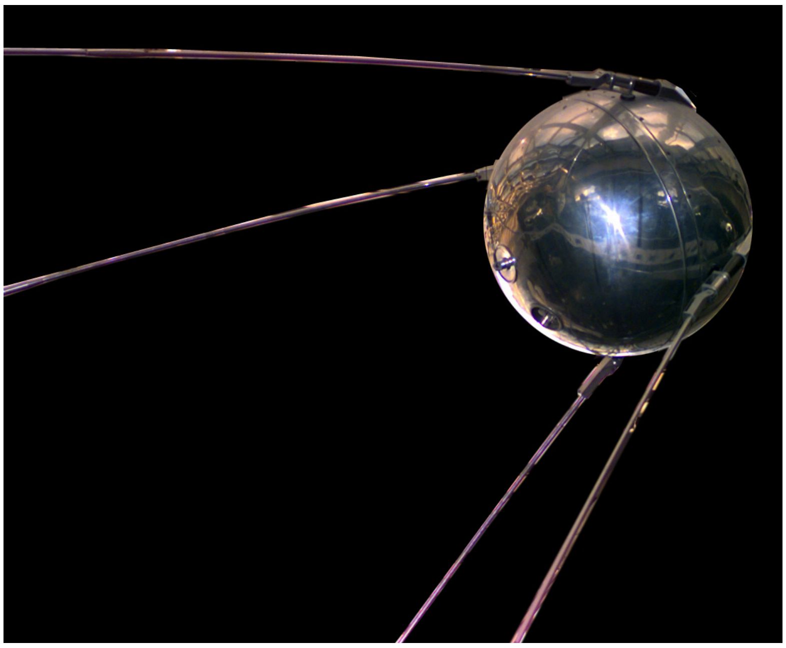


Figure 1.1: Sputnik 1 (NASA Public Domain)

Finally, on January 31, 1958 the United States managed to launch their first artificial satellite, the Explorer-1 (Figure 1.2).

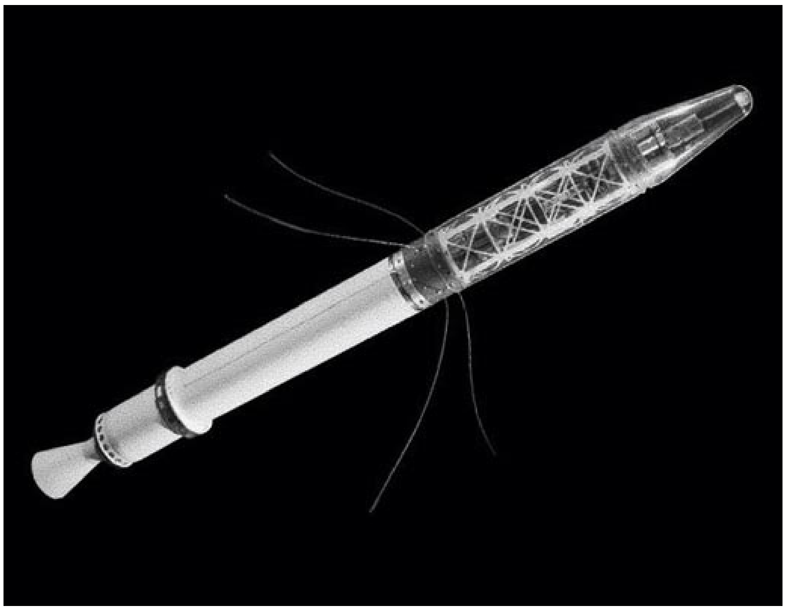


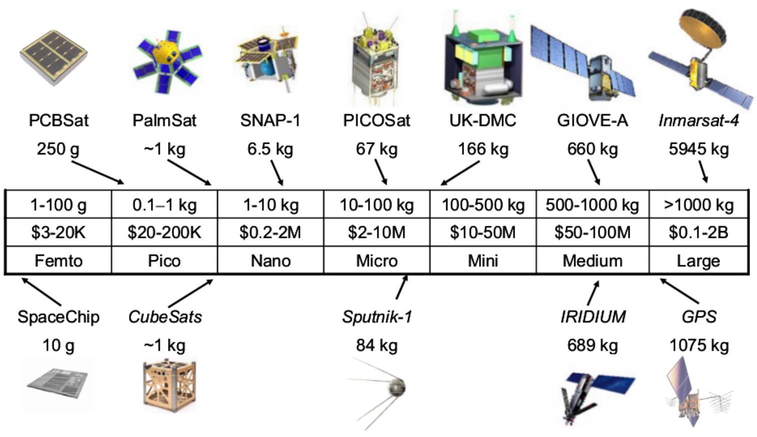
Figure 1.2: Explorer 1 (NASA Public Domain)

As of October 1, 2011 there were 966 operating satellites in orbit. About two-thirds of these were owned by the United States, Russia and China[1]. Their major types are:

- Communications: used for television, radio, Internet and telephone services.
- Navigation: using radio time signals, this satellites allow mobile receivers on the ground to determine their exact position. They are also used to determine the location of satellites situated in lower orbits.
- Exploration: used to observe distant planets, galaxies and other outer space objects by using telescopes and other sensors.
- Remote sensing: Remote sensing satellites are used to gather information about the nature and condition of Earth. The sensors in this kind of satellites receive electromagnetic emissions in several spectral bands and can detect the object's composition and temperature. Also, environmental conditions and so on. These satellites have also been widely used as military "spy satellites".

The constant evolution of technology and the growth of human needs have me the mission requirements rise throughout the last decades, thus, satellite mass has grown from Sputnik's 84 kg. and Explorer-1's 14k kg. to over 6,000 kg in 2007[2]. The main consequence of this, amongst others, has been an increment in mission costs.

To counter this trend, the small satellite movement was created by the academic community and it has shown how mission costs can be cut dramatically to a point in which a university can build a launch their own satellite. (Figure 1.3). Due to its success, it has become vigorous industry. Aalto-1 and ESTCube-1, projects with which this thesis is related, are nanosatellites, and the perfect example of this new concept.



1-100 g	0.1–1 kg	1-10 kg	10-100 kg	100-500 kg	500-1000 kg	>1000 kg
$3-20K	$20-200K	$0.2-2M	$2-10M	$10-50M	$50-100M	$0.1-2B
Femto	Pico	Nano	Micro	Mini	Medium	Large

Figure 1.3: Satellite Mass and Cost Classification [2]

1.1 Background

As it was stated before, this thesis is closely related to two different projects. Aalto-1 and ESTCube-1. These two projects are based on the most common standard use by universities, CubeSat[3]. An open standard developed by the California Polytechnic State University and Stanford University.

1.1.1 Aalto-1

Led by Aalto University, Aalto-1 project aims to build a multi-payload remote sensing nanosatellite (Figure 1.4). The size of the satellite is approximately 34 cm x 10 cm x 10 cm with a mass of less than 4 kg[4]. Aalto-1 will also be the first Finnish satellite.

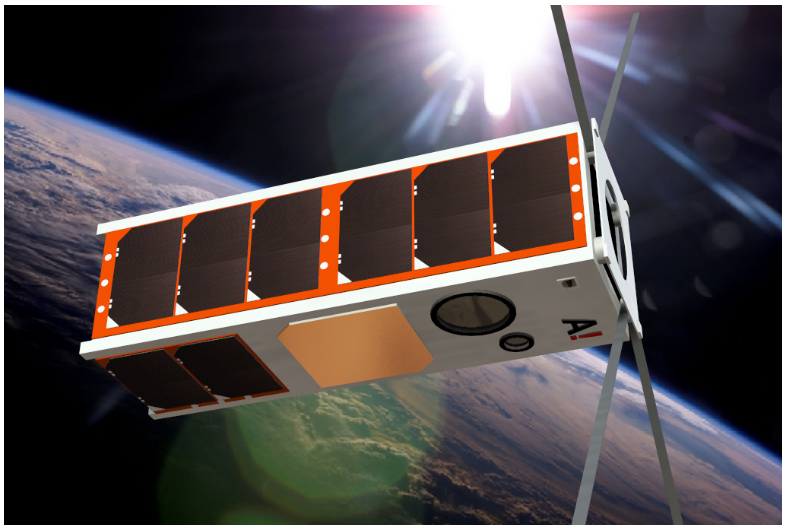


Figure 1.4: Aalto-1

There are different institutions cooperating to make this possible. The main payload, the imaging spectrometer, has been designed and built by VTT Technical Research Centre of Finland. The Radiation Monitor (RADMON) has been designed by the Universities of Helsinki and Turku in cooperation with the Finnish Meteorological Institute (FMI). The Plasma Brake has been designed by a consortium including the FMI, the Department of Physics of the University of Helsinki, the Departments of Physics and Astronomy and Information Technology of the University of Turku, the Accelerator laboratory of the University of Jyväskylä, Aboa Space Resarch Oy, Oxford Instruments Oy and other Finnish companies. Meanwhile, Aalto University is responsible for designing and building the satellite platform and the day-to-day operation of the project.[5]

Aalto-1's mission is to validate the technologies used by the payloads in space environment and measure their performance. In addition, it is also an educational project. Students are the main workforce towards its success. Being the the first Finnish student satellite mission, is a good tool to improve Finnish space teaching and also allow students to be in touch with prominent partners, both domestic and international, in the space technology field.

1.1.2 ESTCube-1

ESTCube-1 is a single-unit CubeSat (Figure 1.5). The size of the satellite is approximately 10 cm x 10 cm x 10 cm with a maximum mass of 1.33 kg. It has been build by students of the universities of Tartu and Tallinn, in Estonia, and it is the first Estonian satellite [6]. It was launched from the Guaiana Space Centre on May 7, 2013 as one of the three payloads of the Vega VV02 rocket [7].

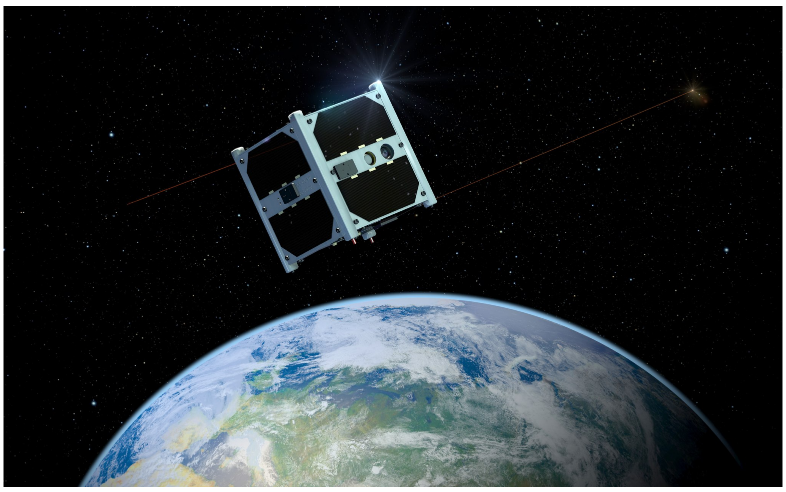


Figure 1.5: ESTCube-1

ESTCube-1's main goal is to observe and measure the E-sail effect for the first time. It has been placed into a polar low Earth orbit (LEO) and will deploy a single 10 m long and 9 mm wide tether[6]. This has relation with Aalto-1's Plasma Brake, which will deploy a 100 m long tether. The require duration of ESTCube-1's mission is a few weeks and it can be extended to about a year.

1.2 Problem statement

1.3 Research objectives and scope

The purpose of this thesis is to develop a calibration system for a ground station which will solve the problem mentioned above. In addition, it is necessary to check if the received values are within a expected range, so a different system will be developed for that purpose.

Using ESTCube-1 project as a baseline, the author aims to develop such systems which are generic enough to be used by any number of satellite missions. The modules developed will be integrated into *Hummingbird*, an open source platform for monitoring and control which will be explained later in this work.

1.4 Motivations

1.5 Outline of the thesis

This thesis is structured as follows: the second chapter gives a general overview about satellite communications, how orbits affect these and what are the different protocols, hardware and software components to do so. The third chapter explains briefly what Hummingbird is and what is behind it. The fourth chapter goes through the requirements set for the software project whereas the fifth one focuses on the design and implementation. Chapter six explains what are the next steps in the work. Finally, chapter seven summarizes the conclusions of this thesis.

Chapter 2

Satellite Communications

This chapter covers the topic of how the satellite communicates with Earth and back. One of the most important constraints that needs to be approached when working with ground station is the fact that the satellite is orbiting around Earth. To understand this, the first section of this chapter will cover what orbits are, what elements describe them and how that information can be delivered by a computer. Following, the different types of data exchanged by the satellite and the ground station will be covered. The last section of the chapter goes through what a ground station is, its hardware and software components and how it communicates with a satellite in space.

2.1 Orbits

An orbit is the **gravitationally curved path of an object around a center of mass**. Examples of orbits can be the Earth around the Sun or artificial satellites around the Earth.

2.1.1 Kepler's Laws

Planetary movements were first mathematically defined by the German mathematician, astronomer and astrologer Johannes Kepler in the 17th century. He concentrated his observations into three simple laws[8]:

- The orbit of each planet is an ellipse with the Sun occupying one focus.
- The line joining the Sun to a planet sweeps out equal areas in equal intervals of time.
- A planet's orbital period is proportional to the mean distance between the Sun and the planet, raised to the power of 3/2.

These laws generally apply to every celestial body. When analysing to bodies, if one is much bigger than the other, it conforms the "two-body problem". It assumes that both bodies are spherical and they are modelled as if they were point particles. This means that influences from any third body are discarded. The analysis of this problem has resulted in six elements that completely define an orbit, which will be explained in the next section.

2.1.2 Classical Orbital Elements

The Classical Orbital Elements are six parameters which uniquely identify an orbit. They also can be used to predict future positions of the satellite.[9]

The first two elements, the orbit's size and shape are defined based on a 2D representation on an ellipse (Figure 2.1).

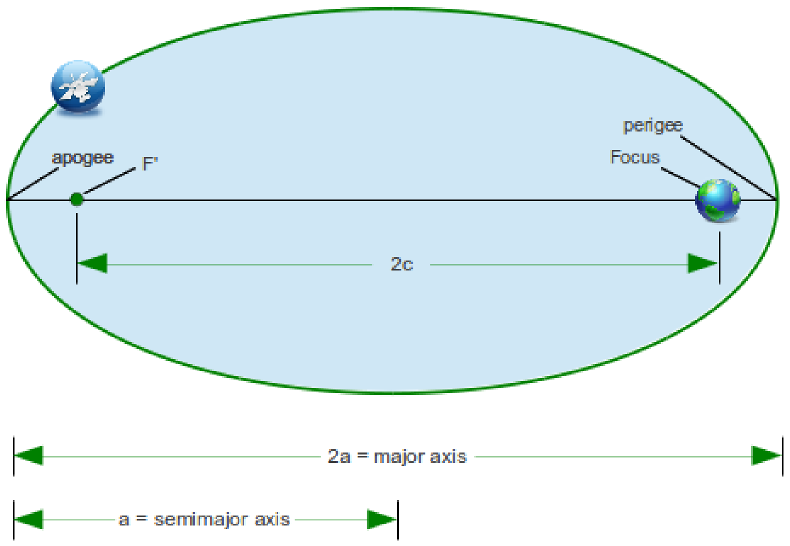


Figure 2.1: Semimajor Axis

- The *semimajor axis* (a) is one half the distance across the long axis of the orbit, and it represents the orbit's size.
- The *eccentricity* represents the shape of the orbit. It describes how much the ellipse is elongated compared to a circle. Based on the latter, the orbit can have the following shapes, as shown in Figure 2.2

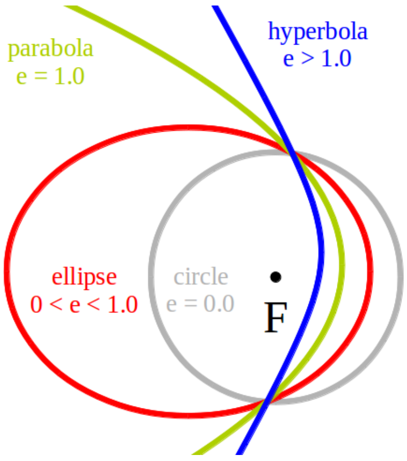


Figure 2.2: Eccentricity

Before jumping onto the next Orbital Elements it is necessary to point out that the Geocentric-equatorial Coordinate System will be used. It is now a 3D representation, where the fundamental plane is Earth's equatorial plane and the principal direction is in the vernal equinox direction (see Figure 2.3).

The following orbital elements define the orientation of the orbital plane:

- The inclination (i) describes the tilt of the orbital plane with respect to the reference plane. It is measured at the ascending node. This is, where the orbit crosses with the reference plain when moving upwards.
- The right ascension of the ascending node (Ω) represents the angle between the principal direction and the point where the orbital plane crosses the reference plane from south to north measured eastward.

Based on this two elements, the orbits can be classified as shown in Table 2.1

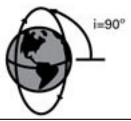
Inclination (i)	**Orbital Type**	**Diagram**
0° or 180°	Equatorial	
90°	Polar	
$0^\circ \leq i < 90^\circ$	Direct or Prograde (moves in the direction of Earth's rotation)	ascending node
$90^\circ < i \leq 180^\circ$	Indirect or Retrograde (moves against the direction of Earth's rotation)	ascending node

Table 2.1: Types of Orbits and Their Inclination [9].

Although it is not part of the COEs, orbits can also be sorted by their altitude. NASA's classification divides orbits in three groups (Table 2.2).

Orbit	**Altitude (a)**	**Uses**
Low Earth Orbit (LEO)	$a < 2000Km$	Scientific and weather satellites
Medium Earth Orbit (MEO)	$2000Km \leq a < 36000Km$	GPS
High Earth Orbit (HEO) or Geosynchronous (GSO)	$36000Km$	Communications (phones, television, radio)

Table 2.2: NASA's classification of orbits. [10]

It is now time to go through the last two COEs:

- The argument of perigee (ω) is the angle between the ascending node and the perigee, measured in the direction of the satellite's motion.
- The true anomaly (v) specifies the location of the satellite within the orbit. Amongst all the CEOs, this is te only one which changes over time. It is the angle between the perigee and the satellite's position vector measured in the direction of its motion.

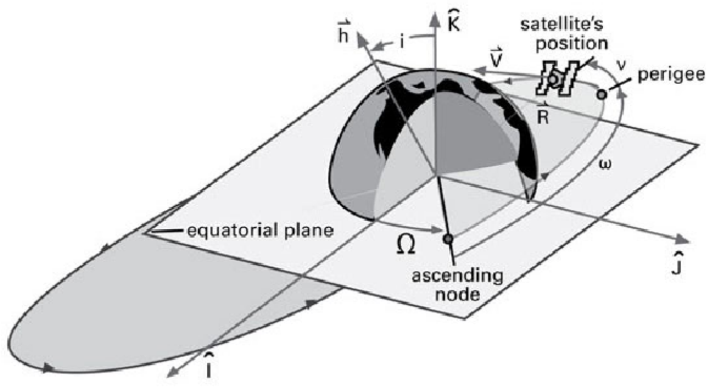


Figure 2.3: Classical Orbital Elements [9]

2.1.3 Ground Tracks

The satellite ground tracks are the projection of its orbit onto Earth. An example of this can be Figure 2.4.

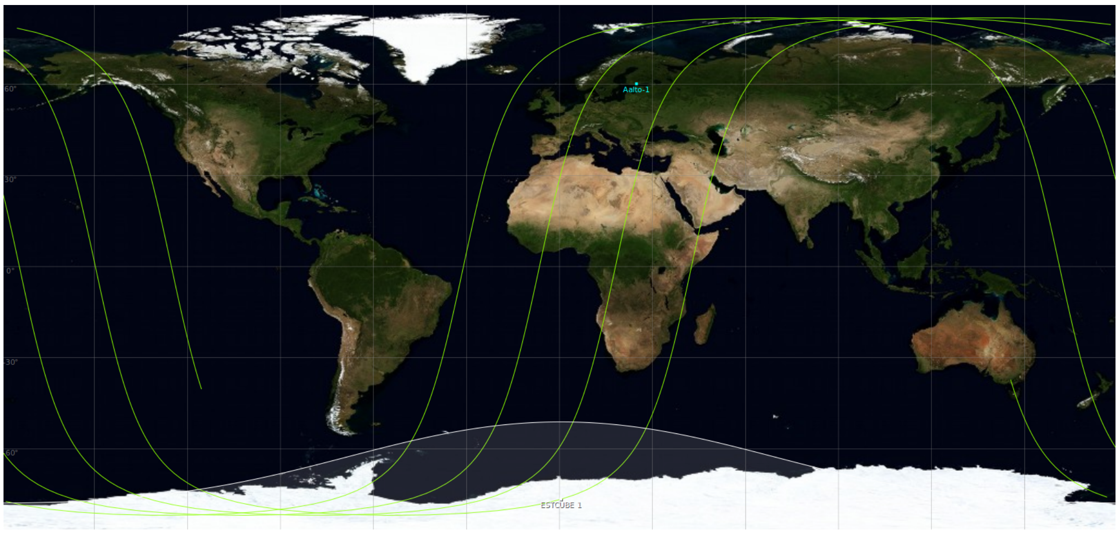


Figure 2.4: Ground Tracks of ESTCube-1

Since the orbital plane does not move in inertial space, the satellite's orbit will always be the same. If Earth did not move the representation of the orbit would be a single line, as the ground track would continuously repeat. However, Earth rotates at 1600 km/hr. Thus, even if the orbit does not change, from the Earth-based observer's point of view it appears to shift to the west.

2.1.4 Two Line Elements

A two line element set (TLE) is a data format created by the North American Aerospace Defense Command (NORAD) and NASA to transport sets of orbital elements describing satellite orbits around Earth. These TLEs can be later processed by a computer to calculate the position of a satellite at a particular time and are usually used by ground stations in order to track them.

The following snippet shows an example of a TLE for the International Space Station.

```
ISS (ZARYA)
1 25544U 98067A 13166.62319444 .00005748 00000-0 10556-3 0 120
2 25544 51.6483 116.0964 0010829 73.3727 265.7013 15.50799671834453
```

Field	**Columns**	**Content**	**Example**
1	01	Line number	1
2	03-07	Satellite number	25544
3	08	Classification (U=Unclassified)	U
4	10-11	International Designator (Last two digits of launch year)	98
5	12-14	International Designator (Launch number of the year)	067
6	15-17	International Designator (Piece of the launch)	A
7	19-20	Epoch Year (Last two digits of year)	08
8	21-32	Epoch (Day of the year and fractional portion of the day)	264.51782528
9	34-43	First Time Derivative of the Mean Motion	-0.00002182
10	45-52	Second Time Derivative of Mean Motion (decimal point assumed)	00000-0
11	54-61	BSTAR drag term (decimal point assumed)	-11606-4
12	63	Ephemeris type	0
13	65-68	Element number	292
14	69	Checksum (Modulo 10) (Letters, blanks, periods, plus signs = 0; minus signs = 1)	7

Table 2.3: Two-Line Element Set Format Definition, Line 1

Field	**Columns**	**Content**	**Example**
1	01	Line number	1
2	03-07	Satellite number	25544
3	09-16	Inclination [Degrees]	51.6416
4	18-25	Right Ascension of the Ascending Node [Degrees]	247.4627
5	27-33	Eccentricity (decimal point assumed) (Launch number of the year)	0006703
6	35-42	Argument of Perigee [Degrees]	130.5360
7	44-51	Mean Anomaly [Degrees]	325.0288
8	53-63	Mean Motion [Revs per day]	15.72125391
9	64-68	Revolution number at epoch [Revs]	56353
10	69	Checksum (Modulo 10)	00000-0

Table 2.4: Two-Line Element Set Format Definition, Line 2

2.2 Data

The data exchanged between a satellite and the ground stations on Earth can be divided into three different categories: the beacon, the telemetry and the telecommands.

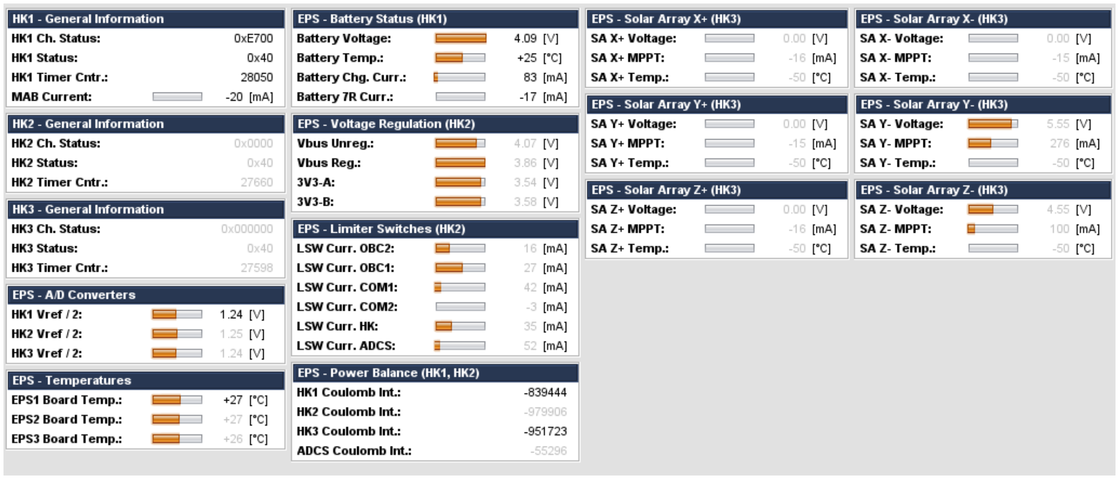
2.2.1 Beacon

A *beacon* is a radio signal transmitted continuously or periodically over a specified radio frequency. It provides a small amount of information such as identification or location, but it can have more applications. Examples of these are: adjust the power of the ground station signal based on the beacon's strength or tune the ground station to compensate the doppler shift.

2.2.2 Telemetry

Telemetry data is sent from the satellite to the ground station and can also be divided in three sub-categories.

The *housekeeping data* provides information abouth the health and operating status of the satellite. Examples of this data can be pressure, voltages and currents, or also bits representing the operational status of all the components as it is shown in Figure 2.5. The size of this data is usually quite small, so a bit rate on only a few hundreds of bits per second is enough to complete the transmission successfully.



HK1 - General Information		
HK1 Ch. Status:	0xE700	
HK1 Status:	0x40	
HK1 Timer Cntr.:	28050	
MAB Current:	-20	[mA]

HK2 - General Information	
HK2 Ch. Status:	0x0000
HK2 Status:	0x40
HK2 Timer Cntr.:	27660

HK3 - General Information	
HK3 Ch. Status:	0x000000
HK3 Status:	0x40
HK3 Timer Cntr.:	27598

EPS - A/D Converters		
HK1 Vref / 2:	1.24	[V]
HK2 Vref / 2:	1.25	[V]
HK3 Vref / 2:	1.24	[V]

EPS - Temperatures		
EPS1 Board Temp.:	+27	[°C]
EPS2 Board Temp.:	+27	[°C]
EPS3 Board Temp.:	+26	[°C]

EPS - Battery Status (HK1)		
Battery Voltage:	4.09	[V]
Battery Temp.:	+25	[°C]
Battery Chg. Curr.:	83	[mA]
Battery 7R Curr.:	-17	[mA]

EPS - Voltage Regulation (HK2)		
Vbus Unreg.:	4.07	[V]
Vbus Reg.:	3.86	[V]
3V3-A:	3.54	[V]
3V3-B:	3.58	[V]

EPS - Limiter Switches (HK2)		
LSW Curr. OBC2:	16	[mA]
LSW Curr. OBC1:	27	[mA]
LSW Curr. COM1:	42	[mA]
LSW Curr. COM2:	-3	[mA]
LSW Curr. HK:	35	[mA]
LSW Curr. ADCS:	52	[mA]

EPS - Power Balance (HK1, HK2)	
HK1 Coulomb Int.:	-839444
HK2 Coulomb Int.:	-979906
HK3 Coulomb Int.:	-951723
ADCS Coulomb Int.:	-55296

EPS - Solar Array X+ (HK3)		
SA X+ Voltage:	0.00	[V]
SA X+ MPPT:	-16	[mA]
SA X+ Temp.:	-50	[°C]

EPS - Solar Array X- (HK3)		
SA X- Voltage:	0.00	[V]
SA X- MPPT:	-15	[mA]
SA X- Temp.:	-50	[°C]

EPS - Solar Array Y+ (HK3)		
SA Y+ Voltage:	0.00	[V]
SA Y+ MPPT:	-15	[mA]
SA Y+ Temp.:	-50	[°C]

EPS - Solar Array Y- (HK3)		
SA Y- Voltage:	5.55	[V]
SA Y- MPPT:	276	[mA]
SA Y- Temp.:	-50	[°C]

EPS - Solar Array Z+ (HK3)		
SA Z+ Voltage:	0.00	[V]
SA Z+ MPPT:	-16	[mA]
SA Z+ Temp.:	-50	[°C]

EPS - Solar Array Z- (HK3)		
SA Z- Voltage:	4.55	[V]
SA Z- MPPT:	100	[mA]
SA Z- Temp.:	-50	[°C]

Figure 2.5: Housekeeping data of the satellite Masat-1

Attitude data is generated by different sensors, such as magnetometers, gyroscopes, accelerometers and Sun, Earth and star sensors.

Payload data changes with every mission and needs to be considered individually. Scientific or Earth-observing mission normally generate very large data volumes, specially in the form of images. An example of this can be Figure 2.6, the first picture taken by the Hungarian nanosatellite Masat-1[13].

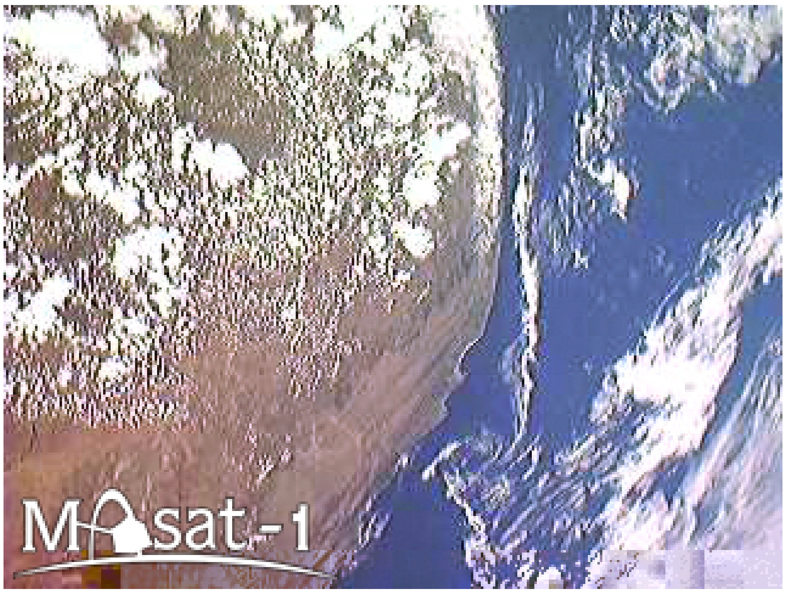


Figure 2.6: Picture of South Africa taken from the nanosatellite Masat-1

2.2.3 Telecomands

The telecommands are sent from the ground station to the satellite. They are used to remotely control its functions and are divided in three basic types [8]:

- *Low-level on-off commands.* These are logic-level pulses used to set or reset log flip-flops.
- *High-level on-off commands.* Higher-powered pulses, capable of operating a latching relay or RF waveguide switch direcly.
- *Proportional commands.* Digital words. Used for purposes such as reprogramming memory locations on the on-board computer or setting up registers in the attitude control subsystem.

2.3 Ground Station

One integral part of every satellite mission is the ground station. It works as the first and final piece of the communication link. Its main functions are the following:

- Tracking the satellite to determine its position in orbit.
- Gather data to keep track of the satellite's data and status.
- Command operations to control the different functions of the satellite.
- Process the received engineering and scientific data to present it in the required formats.

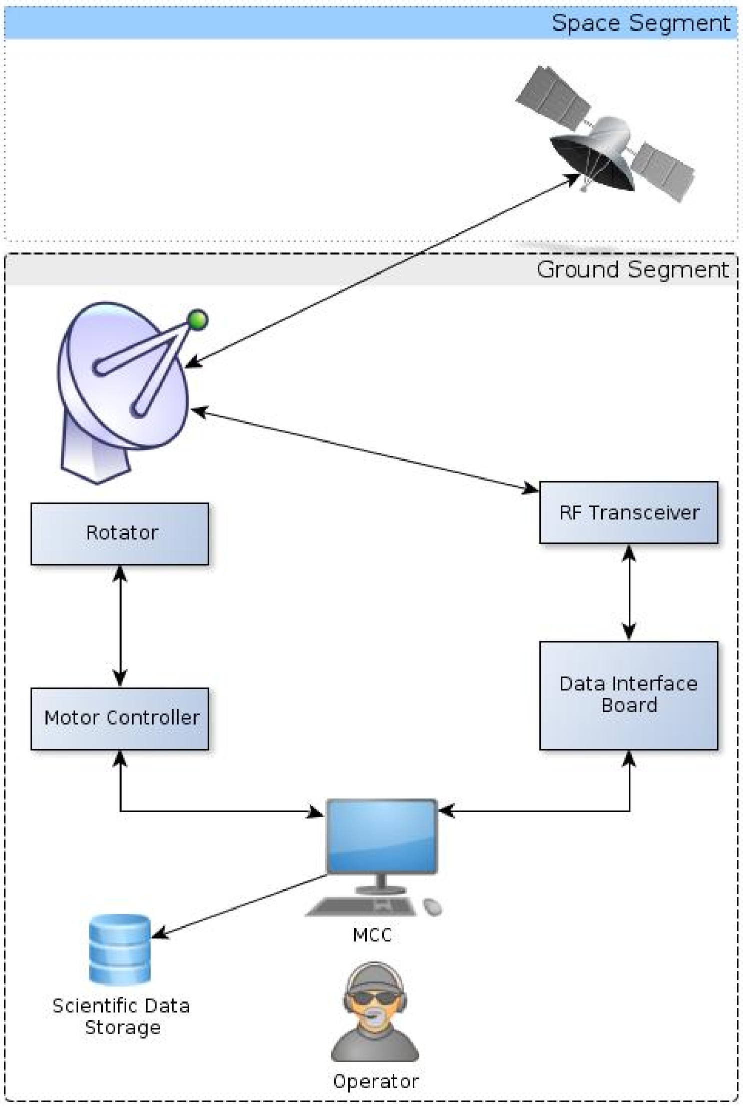


Figure 2.7: Diagram of a ground station

It is important to remember that university satellites are usually classified as amateur satellites. This means that they use amateur radio frequencies and the usage of the ground station is bound to each country's amateur radio regulations.

2.3.1 Hardware

The main components of a ground station are the antenna, the transceiver, the data recorders and the computers and their peripherals.

Antennas

The main hardware component of a ground station is the antenna. Its functions may include tracking, receiving telemetry, sending telecommands, etc.

The frequencies most commonly used for amateur satellites are shown in Table 2.5.

Table 2.5: TODO: Table with Frequencies.

Transceiver

A transceiver is a hardware unit containing both a transmitter and a receiver. It acts as an intermediary between the antenna and a computer, changing the radio frequency into bytes and viceversa.

2.3.2 Software

The activity in the ground station does not start when the satellite is passing over it and does not end once it is gone. There are certain tasks that need to be done before, during and after the pass.

Before the satellite arrives it is necessary to determine and predict its orbit. Based on this prediction the software will schedule future passes and generate the command list which will be sent during the pass.

The real-time software comes into operation when the satellite is visible from the ground station. It is in charge of controlling the antenna rotor to follow it across the sky; it will also send telecommands to the satellite and verify their correct reception. In addition, it will receive the data being transmitted from the satellite, which will be processed later.

Once the satellite is not visible any more the post-pass software comes into play. The data received during the pass is now processed and stored so the specialists can analyse it.

There are many different kinds of software oriented towards its use by amateur satellite missions. Examples of this are **GPredict**[14] and **Orbitron**[15], which are used for tracking and prediction, or **Carpcomm**[16], which amongst other functionalities aims to build a network of ground stations.

There is also professional software in use aiming to build ground station networks. One example is **GENSO**, a project of the European Space Agency (ESA) coordinated by its Education Office. The University of Vigo in Spain hosts the European Operations Node and coordinates the access to the network[17]. At the same time, and as a cooperation with the ESTCube-1 project, CGI is supervising the development of similar solution, **Hummingbird**[18], which will be explained more deeply in the following chapters, as this thesis is part of the mention project.

2.3.3 Protocols

A protocol is an agreement between the communicating parties on how communication is to proceed[19]. This section will be focused on the OSI Reference model as well as on some of the most popular protocols for amateur radio communications.

The OSI Reference Model

The Open Systems Interconnection (OSI) Reference Model was developed in 1983 and revised in 1995. This model deals with connecting systems that are open for communication with other systems. It consists on seven layers which are explained in Table 2.6.

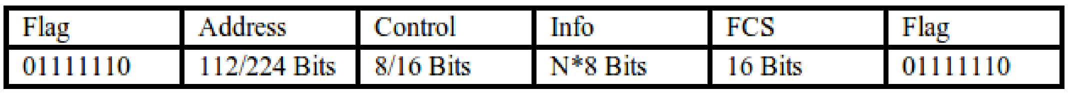
OSI Model			
	Data Unit	**Layer**	**Function**
Host Layers	Data	7. Application	Network process to application.
		6. Presentation	Data representation, encryption and decryption, convert machine dependent data to machine independent data
		5. Session	Interhost communication, managing sessions between applications
	Segments	4. Transport	End-to-end connection, reliability and flow control
Media Layers	Packet/Datagram	3. Network	Path determination and logical adressing
	Frame	2. Data link	Physical addressing
	Bit	1.Physical	Media, signal and binary transmission

Table 2.6: OSI Model[19].

AX.25

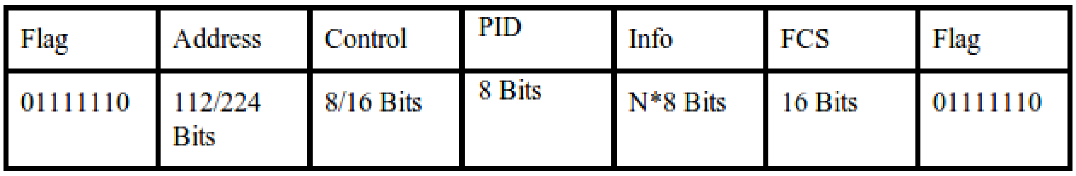
AX.25 is a data link layer protocol designed for use by amateur radio operators. It occupies the first, second and third layers of the OSI model. However, AX.25 was developed before the model came into action, so its specification was not written to separate into OSI layers.

The link-layer packet radio transmission takes place in small blocks of data called frames. Those frames are represented in the Figures 2.8 and 2.9.



Flag	Address	Control	Info	FCS	Flag
01111110	112/224 Bits	8/16 Bits	N*8 Bits	16 Bits	01111110

Figure 2.8: Supervisory and Unnumbered frames [20]



Flag	Address	Control	PID	Info	FCS	Flag
01111110	112/224 Bits	8/16 Bits	8 Bits	N*8 Bits	16 Bits	01111110

Figure 2.9: AX.25 Information frame [20]

FX.25

FX.25 is an extension to the AX.25 protocol. It has been created to complement the AX.25 protocol, providing an encapsulation mechanism that does not alter the AX.25 data or functionalities. AX.25 packets are easily damaged, and this extension intends to remedy the situation by providing a Forward Error Correction (FEC) capability at the bottom of Layer 2.

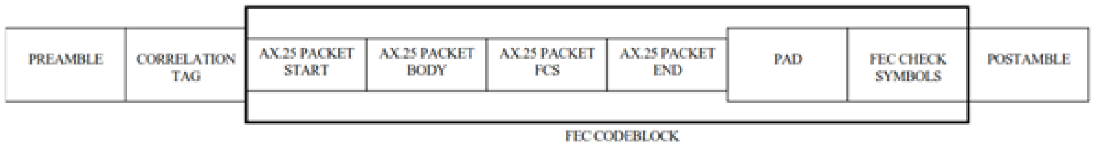


Figure 2.10: FX.25 frame structure [21]

Chapter 3

Hummingbird: the open source platform for monitoring and controlling small satellites

The continuous growth of the popularity of small satellites has been accompanied by an increased interest in creating more a better solutions for the ground segments of the missions. These missions are more complex every time and the use of software oriented for amateurs is starting to prove insufficient. For this reason there are nowadays several projects to build professional-like software and ground station networks to control small satellites. This chapter introduces one of them, *Hummingbird*; the project into which the work carried on in this thesis will be integrated.

Chapter 4

Requirements

4.1 Calibration module

4.1.1 Introduction

Scope

This software is intended to serve as an independent calibration module for *Hummingbird*. As such, it will receive parameters with raw values, calibrate those values and generate new parameters which will be available for other modules in the system to use. The system must be flexible and allow users to define their own calibration scripts.

Definitions

- **Raw values**: values received from the satellite, before going through the calibration process.
- **Engineering values**: result of the calibration process.
- **Hummingbird:** see Chapter 3.

4.1.2 General Description

Product Perspective

This module is part the Hummingbird project based on the advise and needs dictated by the ESTCube-1 team members. For more information about Hummingbird see Chapter 3 and for more information about ESTCube-1 see Chapter 1.

Product Functions

- Information input
  - Allow the user to input the calibration information as an XML file.
  - Parse the XML configuration to generate the calibration scripts.
- Calibration process
  - Receive one raw parameter and return one calibrated parameter.
  - Receive one raw parameter and return several calibrated parameters.
  - Receive several raw parameters and return one calibrated parameter.
  - Receive several raw parameters and return several calibrated parameters.

User Characteristics

- Specialists/Scientists
  - Frequency of use: at the moment of inserting the calibration information.
  - Functions used: XML file to insert the calibration information. Other than that, the process is automated.
  - Technical expertise: Comfortable with XML and shell scripting. Also with simple Java programming.

General Constraints

- The module must be licenced under **Apache License v2.0**[22].
- The use of open source tools is recommended.
- The main programming language must be Java[23].

User Documentation

- Manual for specialist/scientists who will be writing the calibration scripts. The manual must contain examples of the XML format and the way of representing the calibration scripts.

4.1.3 External Interface Requirements

Software Interfaces

Parameter
The module will receive and generate Parameters. The *Parameter* type is part of Hummingbird and is represented as follows.

- Numeric value can be any type).
- Unit of the value.
- Description.
- Timestamp: date and time when the parameter was created.

Apache Camel[24]
Since Hummingbird uses *Apache Camel* for the communication between modules, the parameters for calibration are received and sent back using this system. In addition, Hummingbird has a heartbeat service to check if the module is responding properly.It is necessary to configure the module so it sends and receives messages through Camel.

Communications Interfaces

JMS[25]
The communication interface with the other components in the system is the Java Message Service using Apache Camel. The module is a **JMS client** in a **publish/subscribe model**.

4.1.4 Functional Requirements

Read configuration

Introduction
The first thing the software should do is parse the configuration files to generate the calibration information.

Inputs

XML files with calibration information for the different subsystems.

Processing

1. Find XML files in the selected location.
2. Find calibration information available in each file.
3. Generate calibration table.

Outputs

The process will generate a table with the calibration information for all the different parameters.

Listen to incoming parameters

Introduction
The module will be waiting for new parameters to arrive. When a parameter is ready for calibration it will be sent to the calibrator.

Inputs

Parameters received through Camel.

Processing

1. Receive a parameter.
2. If the parameter is ready for calibration send it to the calibrator.
3. If the parameter needs more parameters to be calibrated wait for those parameters.

Outputs

The output will be one or several parameters which will be sent back to the message queue using Camel.

Error Handling

- If no calibrator is found for the parameter log the error and ignore it. No data return to Camel is expected.
- If there is a problem with the calibrator log the error do not return any data through Camel.

Calibrate

Introduction

Inputs

Parameter to be calibrated plus all extra parameters needed to do so.

Processing

1. Receive parameter(s) needed for calibration.
2. Receive all the calibration information.
3. Use the script to generate the new value
4. Return the new parameter

Outputs

The output will be one or several parameters.

Error Handling
If there is an error it must be sent upwards.

4.1.5 Non-Functional Requirements

Reliability
The software should handle unexpected values correctly. Eg. the value of the parameter is *null* or *NaN*.

Availability
Hummingbird setup can work without the module. However, it must run for days without problems.

Security
Handled by Hummingbird.

Maintainability
XML configuration at startup.

Portability
Since it is written in Java it should work wherever a JVM is available.

4.2 Limit checking module

4.2.1 Introduction

Scope

This software is intended to serve as an independent limit checking module for *Hummingbird*. It will receive a parameter and return information about the state of that parameter in relation to the limits.

Definitions

- **Hummingbird:** see Chapter 3.
- **Parameter**: contains the value.
- **State**: a boolean value reporting the state of the parameter.

4.2.2 General Description

Product Perspective

This module is part the Hummingbird project based on the advise and needs dictated by the ESTCube-1 team members. For more information about Hummingbird see Chapter 3 and for more information about ESTCube-1 see Chapter 1.

Product Functions

- Information input
  - Allow the user to input the limit checking information as an XML file.
  - Parse the XML configuration to generate the limits.

User Characteristics

- Specialists/Scientists
  - Frequency of use: at the moment of inserting the limit checking information.
  - Functions used: XML file to insert the limit checking information. Other than that, the process is automated.
  - Technical expertise: Comfortable with XML.

General Constraints

- The module must be licenced under **Apache License v2.0**[22].
- The use of open source tools is recommended.
- The main programming language must be Java[23].
- There must be two options.
  - Sanity limits, soft limits and hard limits. (See Figure 4.2

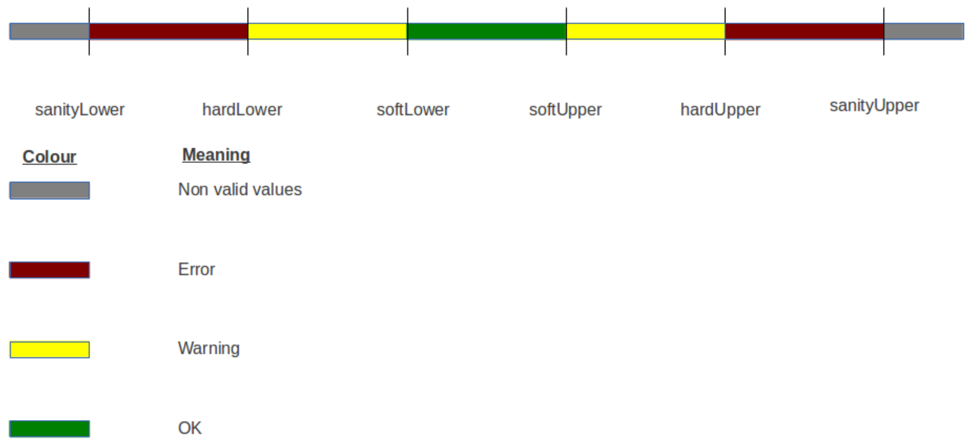


Figure 4.1: Limit Checker with sanity limits available

– Soft limits and hard limits only.

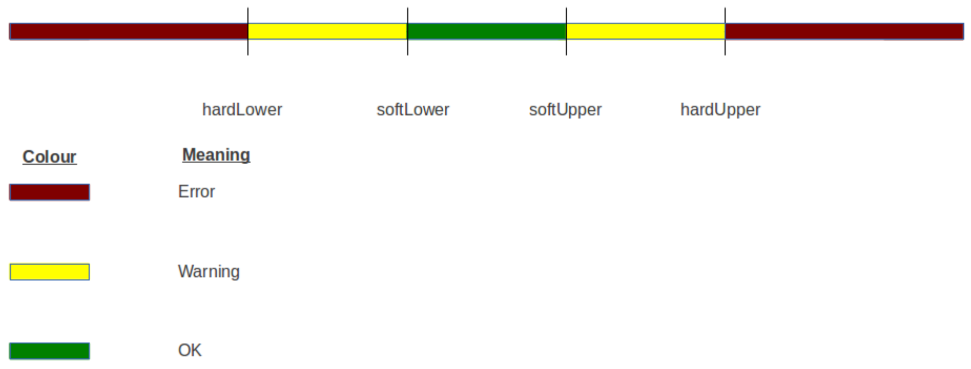


Figure 4.2: Limit Checker only with soft and hard limits

User Documentation

- Manual for specialist/scientists who setting up the limits. The manual must contain examples of the XML format.

4.2.3 External Interface Requirements

Software Interfaces

Parameters

The module will receive Parameters. The *Parameter* type is part of Hummingbird and is represented as follows.

- Numeric value (can be any type).
- Unit of the value.
- Description.
- Timestamp: date and time when the parameter was created.

State

The module will return States. The *State* type is part of Hummingbird and is represented as follows.

- value of the state (boolean).

Apache Camel[24]
Since Hummingbird uses *Apache Camel* for the communication between modules, the parameters for calibration are received and sent back using this system. In addition, Hummingbird has a heartbeat service to check if the module is responding properly.It is necessary to configure the module so it sends and receives messages through Camel.

Communications Interfaces

JMS[25]
The communication interface with the other components in the system is the Java Message Service using Apache Camel. The module is a **JMS client** in a **publish/subscribe model**.

4.2.4 Functional Requirements

Read configuration

Introduction
The first thing the software should do is parse the configuration files to generate the calibration information.

Inputs

XML files with calibration information for the different subsystems.

Processing

1. Find XML files in the selected location.
2. Find limits information available in each file.
3. Generate limits table.

Outputs

The process will generate a table with the limits information for all the different parameters.

Listen to incoming parameters

Introduction
The module will be waiting for new parameters to arrive. Those parameters will be sent to the limit checker.

Inputs

Parameters received through Camel.

Processing

1. Receive a parameter.
2. Send parameter to limit checker.

Outputs

The output will be a list of State elements which will be sent back to the message queue using Camel.

Error Handling

- If no limit checking information is found for the parameter log the error and ignore it. No data return to Camel is expected.
- If there is a problem with the limit checker log the error do not return any data through Camel.

Check limits

Introduction

The software must check the limits, depending on the levels of limits chosen (2 or 3) the comparison will be different.

Inputs

Parameter to be calibrated plus all extra parameters needed to do so.

Processing

1. Receive parameter.
2. Receive the limit checking information.

3. Compare the parameter against the limits.
4. Return the result of the comparison.

Outputs
List of State elements.

Error Handling
If there is an error it must be sent upwards.

4.2.5 Non-Functional Requirements

Reliability
The software should handle unexpected values correctly. Eg. the value of the parameter is *null* or *NaN*.

Availability
Hummingbird setup can work without the module. However, it must run for days without problems.

Security
Handled by Hummingbird.

Maintainability
XML configuration at startup.

Portability
Since it is written in Java it should work wherever a JVM is available.

Chapter 5

Implementation

5.1 Technologies used

As it has been explained previously, the two modules developed as part of the work for this thesis have been designed to be integrated with the Hummingbird project. To do so, Java[23] has been chosen as the main programming language, as it is the language used for the development of Hummingbird. In the same way, Apache Camel[24] and ActiveMQ [26] are used for the communication with the rest of the modules. XStream[27] has been chosen as the library used to parse the XML[28] files used to specify the scientific information.

The final piece of technology used is BeanShell[29], a Java-like scripting language and interpreter which runs in the Java Runtime Environment. The calibration module has been designed to be generic, adaptable to every mission. Also, the goal was to make the calibration information input easy for the scientists, meaning this no need to any difficult Java programming, compilation and so on. BeanShell integrates with the Java code and allows to run those scripts at runtime.

5.2 Implementation of the calibration module

For the sake of clarity, the approach to the explanation will be in small pieces. However, before jumping onto every small piece, it is interesting to look at the package organization of the module.

The software is divided into several Java packages being those the following:

- **eu.estcube.calibration**: provided by the *Hummingbird* software architect, it contains the Apache Camel integration. It also contains the main class of the module.

- **eu.estcube.calibration.calibrate**: contains the implementation of the calibrator.
- **eu.estcube.calibration.constants**: contains several constants used throughout the module.
- **eu.estcube.calibration.domain**: contains the data structures used to represent the calibration information and the calibration units.
- **eu.estcube.calibration.processors**: contains the main algorithm for receiving, calibrating and sending the parameters back. Also, the interface implemented by the calibrator can be found here.
- **eu.estcube.calibration.utils**: contains additional utilities. In this case, the tools to manage finding and reading files.
- **eu.estcube.calibration.xmlparser**: contains the tools to parse the information contained in XML format into data which can be used in the module.

5.2.1 Camel Integration

The integration with Camel is a crucial part of the module since it is where it will take the parameters from. There are several classes related to this, the class diagram of this part can be seen in Figure 5.1.

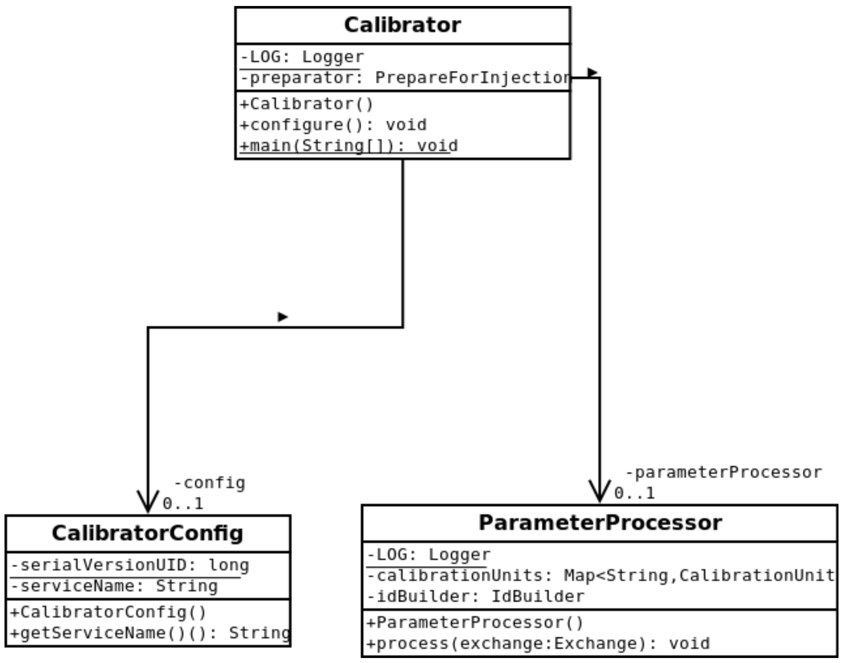


Figure 5.1: Class diagram of the Camel integration

Amongst the three classes represented in Figure 5.1 it is important to highlight two of them: *Calibrator* and *ParameterProcessor*.

Calibrator is the main class of the module. As it is shown in the class diagram there are two methods, the **main** method, where everything starts and the **configure** method. The latter is where the integration with Camel happens, what configures where to get the messages from, what to do with them and where to send them afterwards. In addition, *Hummingbird* utilizes a heartbeat system to check if the modules are working correctly, this is also carried on here. To see a snippet of the code see Table 5.1.

```java

    @Override
    public void configure() throws Exception {

        // @formatter:off
        from(StandardEndpoints.MONITORING)
            .filter(header(StandardArguments.CLASS)
            .isEqualTo(Parameter.class.getSimpleName()))
            .process(parameterProcessor)
            .split(body())
            .process(preparator)
            .to(StandardEndpoints.MONITORING);

        BusinessCard card = new ↙
            ↳ BusinessCard(config.getServiceId(), ↙
            ↳ config.getServiceName());
        card.setPeriod(config.getHeartBeatInterval());
        card.setDescription(String.format("Calibrator; version: ↙
            ↳ %s", config.getServiceVersion()));
        from("timer://heartbeat?fixedRate=true&period=" + ↙
            ↳ config.getHeartBeatInterval())
            .bean(card, "touch")
            .process(preparator)
            .to(StandardEndpoints.MONITORING);
        // @formatter:on

    }
```

Table 5.1: Camel integration Java code

Focusing on the part where the module receives and sends the parameters we can see that:

- Where it gets the messages from –> *from(StandardEndpoints.MONITORING)*
- It only gets messages containing parameters –> *.filter(header(StandardArguments.CLASS) .isEqualTo(Parameter.class.getSimpleName()))*
- What to do with the received message –> *.process(parameterProcessor)*
- Since the result of the processed message is a list and it is necessary to send the parameters one by one, that result must be split. –> *.split(body())*
- Send it back to the messaging service –> *.process(preparator) .to(StandardEndpoints.MONITORING);*

The second part - from line 14 and below - contains the heartbeat system.

ParameterProcessor contains the following:

- Part of the Camel integration.
- Code to load the calibration information from the configuration files.
- Algorithm to process the received parameters.

It will fully explained in the next two subsections.

!!!!REWRITE!!!!!

5.2.2 Loading the calibration information

The first action point when the module is initiated is loading the calibration information. This will read the information from the configuration files written by the specialists and will create a series of calibration units which will later be used to calibrate the incoming parameters.

Figure 5.2, there are several classes involved in this process. They can be organised as follows:

- Main class: *ParameterProcessor*
- Data structures:
  - *CalibrationUnit*
  - *InfoContainer*
- Utils:
  - *InitCalibrationUnits*
  - *InitCalibrators*
  - *FileManager*

- Parser:
  - *Parser*
  - *HashMapConverter*

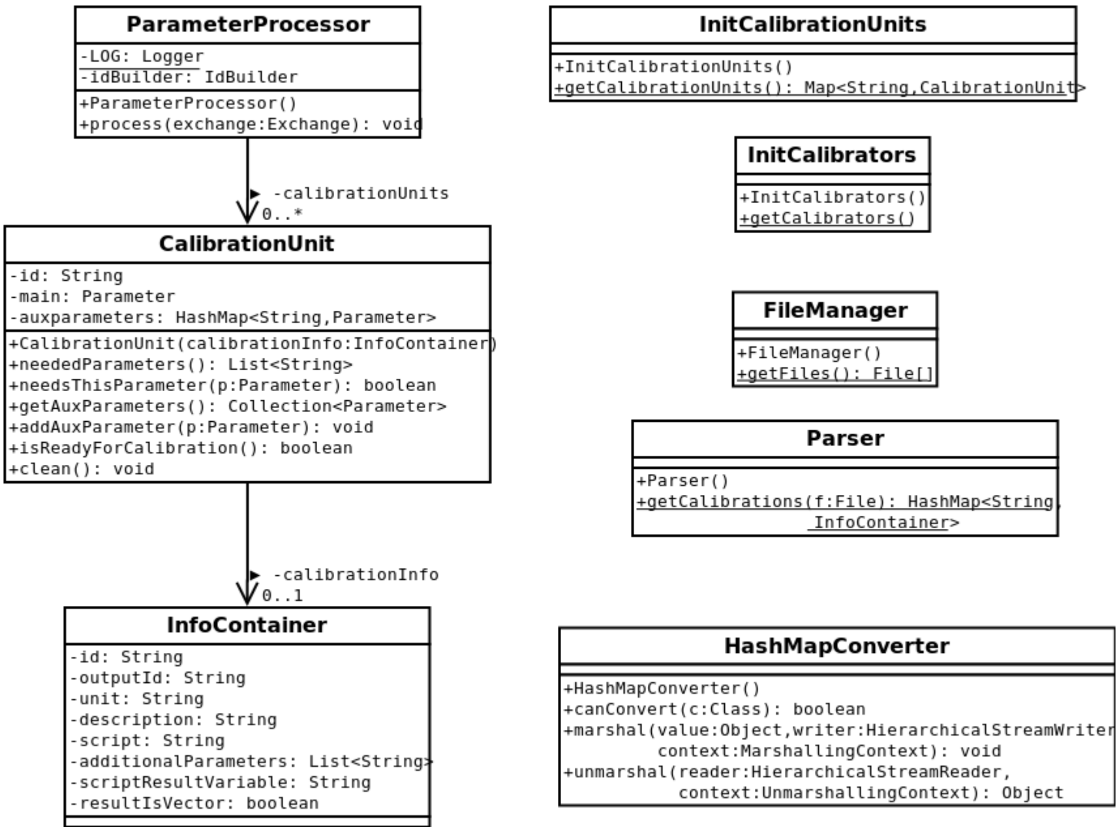

ParameterProcessor
-LOG: Logger
-idBuilder: IdBuilder
+ParameterProcessor()
+process(exchange:Exchange): void
-calibrationUnits
0..*
CalibrationUnit
-id: String
-main: Parameter
-auxparameters: HashMap<String,Parameter>
+CalibrationUnit(calibrationInfo:InfoContainer)
+neededParameters(): List<String>
+needsThisParameter(p:Parameter): boolean
+getAuxParameters(): Collection<Parameter>
+addAuxParameter(p:Parameter): void
+isReadyForCalibration(): boolean
+clean(): void
-calibrationInfo
0..1
InfoContainer
-id: String
-outputId: String
-unit: String
-description: String
-script: String
-additionalParameters: List<String>
-scriptResultVariable: String
-resultIsVector: boolean
InitCalibrationUnits
+InitCalibrationUnits()
+getCalibrationUnits(): Map<String,CalibrationUnit>
InitCalibrators
+InitCalibrators()
+getCalibrators()
FileManager
+FileManager()
+getFiles(): File[]
Parser
+Parser()
+getCalibrations(f:File): HashMap<String, InfoContainer>
HashMapConverter
+HashMapConverter()
+canConvert(c:Class): boolean
+marshal(value:Object,writer:HierarchicalStreamWriter, context:MarshallingContext): void
+unmarshal(reader:HierarchicalStreamReader, context:UnmarshallingContext): Object

Figure 5.2: Class diagram. Loading the calibration information.

The best way to see how these classes interact with each other is by looking at its sequence diagram (Figure 5.3). The main class is *ParameterProcessor*, where the calibration information will be stored in the form of a of calibration units. The reason for choosing a HashMap is its efficiency as it provides constant-time performance for the input and retrieval of information[30]. The process begins by *ParameterProcessor* asking *InitCalibrationUnits* to initialize them. To do so the calibration units need the calibration information (calibrators), which is initialized by *InitCalibrators*. *InitCalibrators* is the class which manages the access to information in the files; first it finds the corresponding XML files to later parse each of them, creating the calibrators. With this calibrators, *InitCalibrationUnits* creates them, and then returns them to the main class where they are ready to receive the incoming parameters.

There is one class present in the class diagram in Figure 5.2 which is not present in Figure 5.3: *HashMapConverter*. This is because this class is managed by *XStream*, the external library used to simplify XML access and parsing.

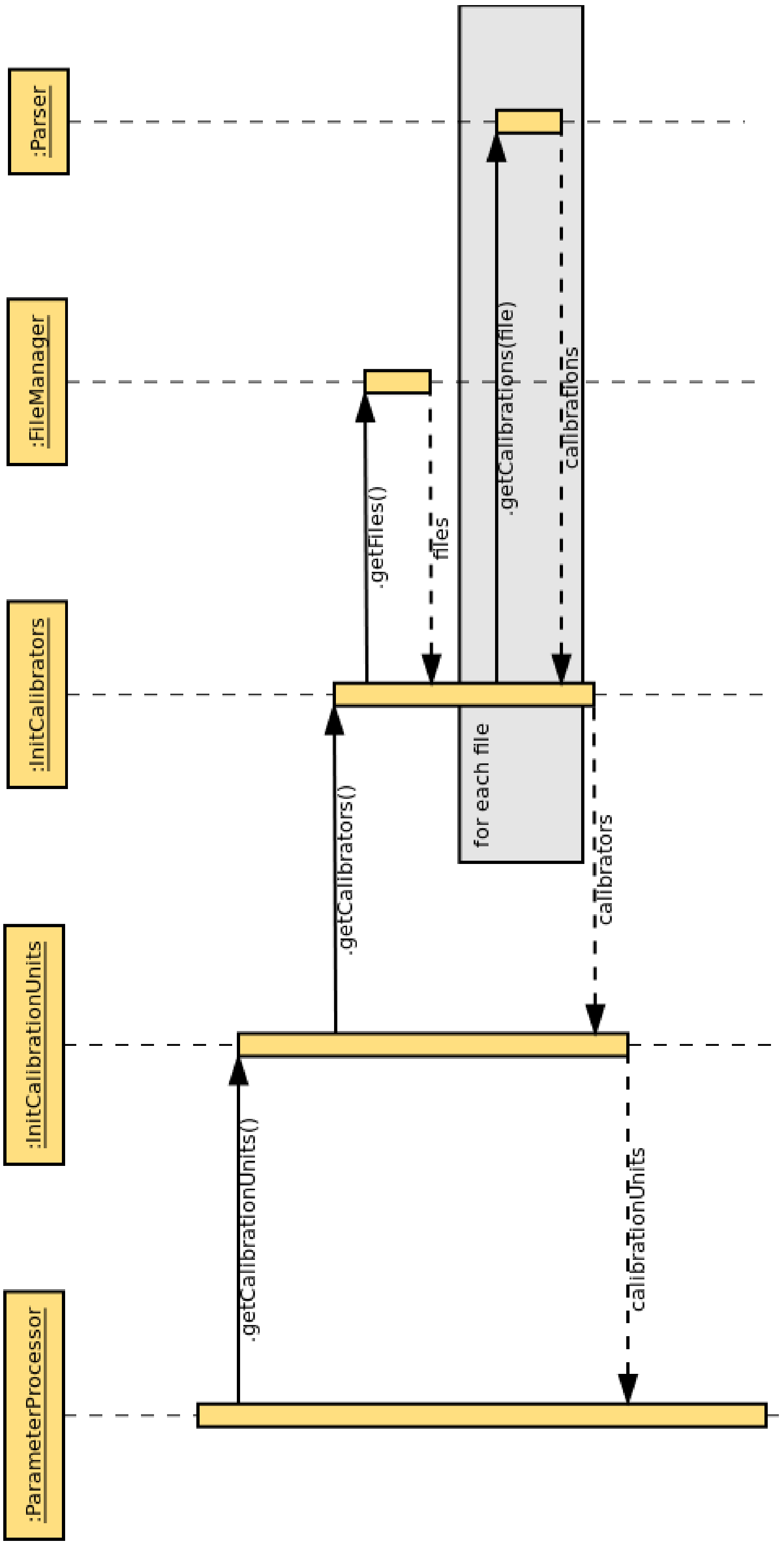


Figure 5.3: Sequence diagram. Loading the calibration information.

For each entry in the XML file, one instance of the above data structure is created and then stored in a **HashMap**.

5.3 Implementation of the limit checking module

5.4 User manual

5.4.1 Calibration module

This short user manual covers the use of the calibration module. The process is fully automated, so the user only needs to configure the pertinent XML file containing the information related to all the parameters which need to be calibrated.

There can be as many files as needed, although it is recommended to have one file per subsystem. This way, the person who is making the changes will not have to be worried about modifying some other parts they do not understand. It is advisable that each file is called as the corresponding subsystem.

The location of the folder where the XML files are stored is fully configurable by a system property. It can be set like this: **-Dpath="/path/to/the/folder"**.

The XML file has the following format:

```xml

<calibration>
    <entry>
        <id></id>
        <description></description>
        <outputId></outputId>
        <unit></unit>
        <scriptInfo>
            <isVector></isVector>
            <resultVariable></resultVariable>
            <auxParameters></auxParameters>
            <script></script>
        </scriptInfo>
    </entry>
</calibration>
```

Table 5.2: blablabla

- **id**: name of the parameter to calibrate.
- **Description**: description of the parameter.

- **outputId**: name of the parameter generated after the calibration. If left blank, it will be the same as **id**. Please note that all calibrated parameters names end in **_cal**.
- **unit**: Units in which the value is represented.
- **isVector**: **true** if the result of the calibration is a vector with several values (which generates several new parameters) or **false** if the calibration returns a single value.
- **resultVariable**: variable in the script in which the result will be stored.
- **auxParameters**: if the are extra parameters needed for the calibration process it is necessary to list them here separated by commas (','). Please note that if the extra parameters also needs to be calibrated the parameters needed for its calibration also must be included here insted of the original one (See Figure **??**).
- **script**: script to generate the calibrated value. Please note that if extra parameters are needed, their calibration script must be included here, not the parameter name (See Figure **??**).

```

<calibration>
    <entry>
        <id>parameterA</id>
        <description>Example of parameter for
         simple calibration</description>
        <outputId>generatedA</outputId>
        <unit>E</unit>
        <scriptInfo>
            <isVector>false</isVector>
            <resultVariable>result</resultVariable>
            <auxParameters></auxParameters>
            <script>result = ↙
                ↳ (parameterA*779.09823)/3145.2839</script>
        </scriptInfo>
    </entry>
</calibration>
```

Table 5.3: Example of simple calibration

Figure **??** represents the simplest example of calibration information. **parameterA** is the parameter to be calibrated and the user has chosen that the name of the calibrated parameter will be **generatedA**. The software will automatically append _cal, so the final output name will be **generatedA_cal**. The information about the calibration script states that the result will not be a vector and the value after the calculations will be stored in a variable called **result**. There are no extra parameters needed ant the calibration script is $(parameterA * 779.09823)/3145.2839$.

```
<calibration>
    <entry>
        <id>parameterA</id>
        <description>Example of parameter which depends
         on others to be calibrated</description>
        <outputId></outputId>\section{Design}
        <unit>E</unit>
        <scriptInfo>
            <isVector>false</isVector>
            <resultVariable>result</resultVariable>
            <auxParameters>parameterB,parameterC</auxParameters>
            <script>result = (parameterA*(parameterB*2345/37))
            /3145.2839 + (parameterC*2)</script>
        </scriptInfo>
    </entry>
</calibration>
```

Table 5.4: Example of calibration depending on other parameters

Figure **??** shows an example of a parameter which depends on others for calibration. Again, **parameterA** is the name of the parameter to be calibrated. In this case the user has not selected an output ID, so it will by default be **parameterA_cal**. The result of the calibration will not be a vector and it needs **parameterB** and **parameterC** to be calibrated. The calibration script can be explained as follows:

- Calibration script for **parameterA**: $result = ((parameterA*(parameterB\_cal))/3145.2839) + (parameterC\_cal)$
- Instead of just stating that **parameterB_cal** is needed to carry on the calibration, the user must specify its calibration script: $parameterB * 2345/37$
- Same thing with **parameterC_cal**: $parameterC * 2$
- The final result is what can be seen in the example: $result = (parameterA * (parameterB * 2345/37))/3145.2839 + (parameterC * 2)$

```

<calibration>
    <entry>
        <id>parameterA</id>
        <description>Example of parameter for simple ↙
            ↳ calibration</description>
        <outputId>generatedA</outputId>
        <unit>E</unit>
        <scriptInfo>
            <isVector>false</isVector>
            <resultVariable>result</resultVariable>
            <auxParameters></auxParameters>
            <script>result = ↙
                ↳ (parameterA*779.09823)/3145.2839</script>
        </scriptInfo>
    </entry>
</calibration>
```

Table 5.5: Example of simple calibration

5.4.2 Limit checking module

This subsection covers the user manual for the limit checking module. The process is fully automated, so the user only needs to configure the pertinent XML file containing the information related to the limits of every parameter.

There can be as many files as needed, although it is recommended to have one file per subsystem. This way, the person who is making the changes will not have to be worried about modifying some other parts they do not understand.It is advisable that each file is called as the corresponding subsystem.

The location of the folder where the XML files are stored is fully configurable by a system property. It can be set like this: **-Dpath="/path/to/the/folder"**.

The XML file has the following format:

```

<limitChecking>
    <entry>
        <id></id>
        <limits>
            <sanityLower></sanityLower>
            <hardLower></hardLower>
            <softLower></softLower>
            <softUpper></softUpper>
            <hardUpper></hardUpper>
            <sanityUpper></sanityUpper>
        </limits>
    </entry>
</limitChecking>
```

Table 5.6: blablabla

- **id**: name of the parameter to calibrate which limits are to be checked.
- **Sanity limits**: Optional. If the value is below the lower limit or above the upper limit it is discarded.
- **Hard limits**:
  - If the sanity limits are available anything between these limits and the sanity limits is considered an error.
  - If the sanity limits are disabled anything below the lower limit or above the upper limit is considered an error.
- **Soft limits**:
  - Anything between the lower and upper soft limits is considered an OK value.
  - Anything between the soft limits and the hard limits is considered OK, but with a warning.

The following two examples show the two ways in which the limit checking module can be configured:

```
<limitChecking>
    <entry>
        <id>parameterA</id>
        <limits>
            <sanityLower>-100</sanityLower>
            <hardLower>-75</hardLower>
            <softLower>-20</softLower>
            <softUpper>20</softUpper>
            <hardUpper>75</hardUpper>
            <sanityUpper>100</sanityUpper>
        </limits>
    </entry>
</limitChecking>
```

Table 5.7: Limit checking with sanity limits available

```
<limitChecking>
    <entry>
        <id>parameterA</id>
        <limits>
            <hardLower>-75</hardLower>
            <softLower>-20</softLower>
            <softUpper>20</softUpper>
            <hardUpper>75</hardUpper>
        </limits>
    </entry>
</limitChecking>
```

Table 5.8: Limit checking without sanity limits available

Chapter 6

Conclussions

Chapter 7

Future work

This chapter describes the next steps to be taken towards full integration of the two modules developed in this work with the rest of the system. The process is described here. The future work will be carried out by the author and also by the ESTCube-1 team.

7.1 Unit testing

Even though the software has been fully tested by the author it is intended to also implement unit tests to cover every possible scenario. This will be done with two different tools:

- JUnit[31]
- Mockito[32]

7.2 Test the integration with ESTCube-1's development build

The Camel code for the modules to be connected with the rest of the system has already been implemented. However, there has been no tests involving the whole system with the two modules working. This work should be carried out before going on any further.

7.3 Integrate with ESTCube-1's live build and with Hummingbird

After the modules have been fully tested in the development environment and have been approved by the people responsible of ESTCube-1's ground segment development they should be integrated into the live build, where they will work with real parameters received from ESTCube-1. In addition, it should also be integrated into Hummingbird when the people in charge of the project considers it convenient.

Bibliography

[1] C. Robert Welti. *Satellite Basics for Everyone.* iUniverse, first edition. United States, 2012. ISBN 978-1-4759-2593-7.

[2] D.J. Barnhart. *Very Small Satellite Design for Space Sensor Networks.* Ph.D. thesis. Faculty of Engineering and Physical Sciences, University of Surrey. United Kingdom, 2008.

[3] *CubeSat Design Specification.* Revision 12, Cal Poly. United States, 2009.

[4] J. Praks. A.Kestilä, M. Hallikainen, H. Saari, J. Antila, P. Janhunen, V. Rami. *Aalto-1 - An experimental nanosatellite for hyperspectral remote sensing.* In Geoscience and Remote Symposium (IGARS). IEEE International, 2011.

[5] A. Näsila, A. Hakkarainen, J. Praks, A. Kestilä, K. Nordling, R. Modrzewski, H. Saari, J. Antila, R. Mannila, P. Janhunen, R. Vainio, M. Hallikainen. *Aalto-1 - A Hyperspectral Earth Observing Nanosatellite.* 2011.

[6] P. Janhunen, P.K. Toivanen, J. Polkko, S. Merikallio, P. Salminen, E. Haeggström, H. Seppänen, R. Kurppa, J. Ukkonen, S. Kiprich, G. Thornell, H. Kratz, L. Richter, O. Krömer, R. Rosta, M. Noorma, J. Envall, S. Lätt, G. Mengali, A.A. Quarta, H. Koivisto, O. Tarvainen, T. Kalvas, J. Kauppinen, A. Nuottajärvi, A. Obraztsov *Electric solar wind sail: Towards test missions (Invited article).* Rev. Sci. Instrum., 81, 111301, 2010.

[7] *Second Vega launch by Arianespace a success: Proba-V, VNREDSat-1 and ESTCube-1 in orbit.* URL `http://www.arianespace.com/news-press-release/2013/5-7-2013-VV02-launch.asp`. [Online; accessed 2013-08-07].

[8] P. Fortescue, J. Startk, G. Swinerd. *Spacecraft Systems Engineering.* Wiley, third edition. United Kingdom, 2003. ISBN 0-470-85102-3

[9] Jerry J. Sellers *Understanding Space, An Introduction to Astronautics.* McGraw-Hill, third edition. United States, 2005. ISBN 978-0-07-340775-3.

[10] NASA *Catalog of Earth Satellite Orbits.* URL `http://earthobservatory.nasa.gov/Features/OrbitsCatalog/`. [Online; accessed 2013-04-17].

[11] T.S. Kelso *Two-Line Element Set Format.* CelesTrak, 2006. URL `https://celestrak.com/columns/v04n03/`. [Online; accessed 2013-04-17].

[12] B. Paige *Satellite Beacons.* The Radio Amateur Satellite Corporation, 2006. URL `http://www.amsat.org/amsat-new/information/faqs/houston-net/beacons.php`. [Online; accessed 2013-04-22].

[13] Masat-1 URL `http://cubesat.bme.hu/en`.

[14] GPredict URL `http://gpredict.oz9aec.net/`.

[15] Orbitron URL `http://www.stoff.pl/`.

[16] Carpcomm URL `http://carpcomm.com/`.

[17] GENSO URL `http://www.genso.org/`.

[18] *CGI provides support for the launch of Estonia's first satellite.* URL `http://www.cgi.com/en/CGI-provides-support-launch-Estonia-first-satellite`. [Online; accessed 2013-08-08].

[19] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, fifth edition. United States, 2010. 978-0132126953.

[20] *AX.25 Link Access Protocol for Amateur Packet Radio.* Tucson Amateur Packet Radio. 1997. URL `http://www.tapr.org/pdf/AX25.2.2.pdf`. [Online; accessed 2013-05-03].

[21] *FX.25: Forward Error Correction Extension to AX.25 Link Protocol For Amateur Packet Radio.* Stensat Group, 2006. URL `http://www.stensat.org/Docs/FX-25_01_06.pdf`. [Online; accessed 2013-05-06].

[22] *Apache License 2.0.* The Apache Software Foundation, 2004. URL `http://www.apache.org/licenses/LICENSE-2.0`. [Online; accessed 2013-08-19].

[23] *Java.* Oracle, 2013. URL `http://www.java.com/en`. [Online; accessed 2013-08-13].

[24] *Apache Camel.* The Apache Software Foundation, 2013. URL `http://camel.apache.org/`. [Online; accessed 2013-05-28].

[25] *Java Message Service.* Oracle, 2013. URL `http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html`. [Online; accessed 2013-08-19].

[26] *ActiveMQ.* The Apache Software Foundation, 2013. URL `http://activemq.apache.org/`. [Online; accessed 2013-05-28].

[27] *XStream.* URL `http://xstream.codehaus.org/`. [Online; accessed 2013-07-05].

[28] *Extensible Markup Language (XML).* URL `http://www.w3.org/XML/`. [Online; accessed 2013-07-05].

[29] Pat Niemeyer. *BeanShell.* URL `http://www.beanshell.org/`. [Online; accessed 2013-07-05].

[30] *Class HashMap<K,V>*. Oracle, 2013. URL `http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html`. [Online; accessed 2013-07-05].

[31] *JUnit*. URL `http://www.junit.org/`. [Online; accessed 2013-08-20].

[32] *Mockito*. URL `https://code.google.com/p/mockito/`. [Online; accessed 2013-08-20].