Universitat Politécnica de Catalunya
Escola Técnica Superior d'Enginyeria de Telecomunicacions de Barcelona
Degree in System Telecommunications Engineering

Marc Justicia Mayoral

Telecommand and Telemetry Implementation of Aalto-2 CubeSat Project

Bachelor's Thesis
Espoo, September 2016

Supervisors:	Professor Jaan Praks, Aalto University
	Professor Adriano Camps, Universitat Politécnica de Catalunya
Advisor:	Nemanja Jovanović M.Sc. (Tech.)

Universitat Politécnica de Catalunya
Escola Técnica Superior d'Enginyeria de Telecomunicacions de Barcelona
Degree in System Telecommunications Engineering

ABSTRACT OF BACHELOR'S THESIS

Author:	Marc Justicia Mayoral		
Title: Telecommand and Telemetry Implementation of Aalto-2 CubeSat Project			
Date:	September 2016	**Pages:**	vi + 40
Major:	Space Science and Technology	**Code:**	S-91
Supervisors:	Professor Jaan Praks Professor Adriano Camps		
Advisor:	Nemanja Jovanović M.Sc. (Tech.)		

This thesis work concentrates on the development of telecommand and telemetry handler software for a 2 kg Aalto-2 nanosatellite, currently scheduled for launch in December 2016. The satellite is part of the international QB50 termosphere mission and it is developed by Aalto University in Espoo, Finland. The telecommand and telemetry (TC/TM) handler, in charge of communications, is one of the most important systems of satellite software, which is executed by On Board Computer (OBC) software. In this thesis, the TC/TM handler subsystem is designed, giving it a special attention in maintaining simplicity and reliability. The design process is started with the derivation of requirements and constraints.

The software is implemented for FreeRTOS, an open-source real-time operating system, which is selected as operation environment of the satellite main OBC software. The designed software uses a Concatenative Language approach and complies with ECSS-PUS standard. It features different libraries that provide capabilities for on-board data handling needs, the most notable of which is the Dyncall library. The library provides functions of dynamic function call in C. The UHF driver library handles incoming and outgoing low-level communications protocols, and the Coffee File System implements storage management.

The work gives also overview of small satellites communication architectures, relevant standards and most important interfaces.

Keywords:	Aalto-2, TC/TM Handler, dynamic dispatch, dyncall, PUS, scripting, C, CubeSats, FreeRTOS, concatenative language
Language:	English

Acknowledgements

I wish to thank professor Jaan Praks who allowed me to come to Finland and work on this amazing project. I like to thank Nemanja Jovanović, who guided me and taught me many things about space engineering and programming. I also wish to thank the friendship and partnership of the people I met at Aalto: Sami, Janne, Erick, Amin, Bagus and Tommy.

On the other hand, I must thank all the people I have met the last years, starting with the fantastic people from Distorsió and Revistes that made these years worth: Alejandro López, Alejandro Nespereira, Albert Navarro, Alberto Gutiérrez, Albert Aparicio, Cristian Cuadrado, Sergi Delgado, Antoni Isern, Eric Martori, Eric Marín, Laia Igelmo, David Molins, Nicolas Díaz, Imanol Barba, Daniel Gallego, Marc Benito, Miquel Piñol, Daniel Armengod, Juan Manuel Piñol, Ferran Olid, Ferran Quer, Ignacio Bosch, Magí Andorrà, Marc Ramos, Gonzalo Gómez, Arantxa Casanova, Eduard Seoane, Ricard Zarco, Javier Coleto, Alvaro Navas, Albert Gonzalez, Josep Ardiaca, Lourdes Extremera, Elena Alonso, Núria Sumalla, Elisabet Valle, David Igon and Hao Luo Wang.

I cannot forget either the people who spent hundreds of hours studying in the library and working with me in the lab, sharing laughs and tears: Josep Zaragoza, Ricard Castellà, Gisela Muxella, Joan Pallarès, Lidia Garrucho, Lucas Rodés, Sergi Liesegang, Yaiza Barrientos, Irene Hernández, Eva Julian and Noe Miguélez. Furthermore, I thank my friends from Girona: Cristina, Jaume, Juli, Rafa, Ester, Laura, Sergi, Victor, Jordi, Alex Barragan, Alex Martin, Cristian, Núria, Aina, Andrea and Alba.

Last but not least; I want to thank my parents, my brother, my grandmother, my grandfather and my uncle for being there always.

Espoo, September 2016

Marc Justicia Mayoral

Abbreviations and Acronyms

ADCS	Attitude, Determination and Control Subsystem
API	Application Process Interface
BPF	Band-Pass Filter
CAN	Controller Area Network
CCS	Code Composer Studio
COMMS	Comunications Subsystem
CRC	Cyclic Redundancy Check
CW	Carrier Wave (modulation)
DC	Dyncall
DSL	Domain Specfic Language
ECSS	European Cooperation for Space Standarization
EM	Engineering Model
FSM	Finite States Machine
GFSK	Gaussian Frequency Shift Keying
GPIO	General-Purpose Input Output
I2C	Inter-Integrated Circuit
ISS	International Space Station
LNA	Low Noise Amplifier
PA	Power Amplifier
PCB	Printed Circuit Board
PEC	Packet Error Control
PUS	Packet Utilization Standard
SPI	Serial Peripheral Interface
TC	Telecommand
TI	Texas Instruments
TM	Telemetry
UHF	Ultra High Frequency
VHF	Very High Frequency
WOD	Whole Orbit Data

Contents

Chapter 1

Introduction

The communication subsystem is an essential element of every spacecraft. During the early days of space technology, spacecrafts had very primitive communication subsystem with simple periodical science and housekeeping data telemetry transmissions. Research and development was needed in order to gain more control and reliability, which is vital for missions such as space exploration, global position systems or weather survey. Decades have passed and technology has developed, so today the ground station gives full control over satellites. Nowadays, new ambitious missions are being planned that require more advanced, smarter, faster and more autonomous designs of hardware and software.

The data handling software usually runs in the On Board Computer (OBC). It is responsible for decoding incoming messages, called telecommands (TC), and encoding outgoing messages, which are called telemetry (TM) packets. The whole software element is called the Telecommand and Telemetry (TC/TM) Handler. Usually, on board software is developed for a specific platform and architecture, which means it is difficult to reuse in other spacecrafts. Therefore, this TC/TM software (as most of the satellite software) is tailor-made for certain spacecraft in order to give better design and implementation. The following work develops new approaches for communications software to achieve the requirements of a specific spacecraft mission. This new approach is done with concatenative programming language definitions and characteristics.

The goal of this work is to design and implement a software architecture for telecommand and telemetry handling, as part of the on board software of the Aalto-2 satellite, by utilising concatenative programming model. The software shall fulfill the requirements and restrictions set by satellite modules hardware, mission specifications and international standards.

The proposed solution is innovative and different from other approaches seen in past satellite projects. It is designed in a FreeRTOS environment, combines different libraries, uses concatenative programming language features and complies with Packet Utilization Standard (PUS). Moreover, PUS standard is a application level interface between satellite and ground that is being adapted to the satellite software and extended to concatenative language.

The thesis is structured as follows; Chapter 1 introduces the topic and the goals of this thesis, Chapter 2, explains different sources applied for designing and implementing this work, Aalto-2 communications requirements and language analysis, Chapter 3 shows TC/TM design and implementation, and its analysis. Finally, in Chapter 4 testing procedures are explained, and in Chapter 5, discussion and conclusion of the work are given along with future work plans.

Chapter 2

Background

The work is done for the Aalto-2 CubeSat satellite, which belongs to the international QB50 satellite constellation. The next chapter gives a short overview of CubeSat satellites, their communication implementations and the QB50 project and the Aalto-2 satellite.

2.1 Small satellites and CubeSats

Small satellites, are defined as those spacecrafts that weigh below 500 kg. The concept was proposed to reduce the building, developing and launching cost. Among the categories in small satellites, the nanosatellite have become the most popular, due to affordable cost and the release of the CubeSat standard.
The CubeSat standard was proposed by professor Jordi Puig-Suari from California Polytechnic State University and professor Bob Twiggs from Standford University in 1998. Their goal was to grant graduated students the opportunity to design, implement, build and operate spacecrafts similar to the first one of history: the *Sputnik*. In 1999 the CubeSat standard was released and it become a popular platform to develop satellites. Nowadays, the standard is used widely by university teams across the world. CubeSats weigh between 1 kg to 10 kg and are shaped as cubes (10 x 10 x 10 cm^3), with a typical weight of 1,3 kg per unit (1U-CubeSat). This new satellite class was embraced due to its low developing cost, simplified design and cheap launching cost, which is normally performed along side other larger satellites. CubeSats usually are similar with each other, because they follow the same standard, often using similar (even identical) components (commercial components adapted or certified for space) and designs. CubeSats also usually use amateur radio bands (Very High Fre-

quency, VHF; and Ultra High Frequency, UHF) for communication. As development cost for nanosatellites is relatively low, developing CubeSats allows hardware and software experimentation that is too expensive for larger satellites. However, new problems are always found as well as solutions that led to new discoveries and techniques. [13]

2.2 CubeSats Projects

In the following section, a few CubeSats projects are presented in order to acknowledged the work performed.

2.2.1 Aalto-1

Aalto-1 is the first Finnish satellite that will reach space by the end of 2016. It is built by students from Aalto University and had the support and collaboration from several space-related Finnish companies and institutions. It has been a five-year project involving a hundred students from different countries and majors. The Aalto-1 is a multi-payload remote-sensing CubeSat, carrying three different payloads and experiments: a spectral Fabry-Perot imager, designed by VTT, a miniaturised radiation monitor (RADMON), developed by the University of Turku and University of Helsinki, and an electrostatic plasma brake, created and built by the Finnish Meteorological Institute (FMI).[18, p.2-3]

The Aalto-1 mission is divided into two phases: the first part consists of remote sensing using the spectral imager and the radiation monitor to perform orbital science observations and to transmit this science data to the ground station. The second phase, will activate the plasma brake and, due to the drag force, it is expected to deorbit the satellite making it fall into the atmosphere and disintegrating it.[18, p.1-2]

Taking a closer look at its On Board Computer, it is an in-house double-redundant design (two computers on one Printed Circuit Board or PCB) with a real-time Linux operating system plus a flash memory for data storage. Moreover, the UHF module uses three different communication frequency channels at different rates: VHF and UHF to uplink and downlink at low rate and S-band to high speed rate downlink. The TC/TM Handler consists on a set of tasks that are being executed concurrently. Scheduling and automatic reporting are implemented, but the user can request a byte dump at any moment during a satellite link coverage window. [18]

2.2.2 CubeCat-1

This is the first initiative from Universitat Politécnica de Catalunya (UPC) to develop nanosatellites. The goal was to design and build a CubeSat capable of carrying different scientific payloads and gave the opportunity to students of participating in space technology development. This satellite is carrying seven different payloads with experiments designed by professors and students from UPC and foreign universities. It is being launched in the same rocket that carries Aalto-1 CubeSat. [11]

The OBC operating system of the satellite is a Linux 2.6, patched with Xenomai 2.6 to provide real-time capabilities. Its architecture has four hierarchical layers of abstraction. The system is based on a Finite State Machine (FSM) with six states: Init, Idle, Contingency, Standby, Payloads and COMMS. The last one is entirely dedicated to send telemetry source packets and receive new telecommands. This does not mean that the satellite cannot receive or send outside this state. It will reach this state if the satellite has coverage with the ground station and enough power to transmit and receive. While in this phase the satellite cannot perform any payload task. [11, p.111-122]

Summarising: both CubeSats have similarities and differences. They share an akin mechanical structure, because they both use the CubeSat standard, on the other hand, modules are made in-house and use different commercial components. At OBCs level, both have different architecture and use different electronics components. Although, both use open-source real-time operating systems and CAN, I^2C and SPI buses.

2.3 QB50 project

The QB50 mission is a European project the goal of which is to launch a satellite network of fifty satellites and study the lower termosphere. The QB50 project grants technical support to university teams to build and operate a satellite, granting the chance for students and professors to perform first-class space science.

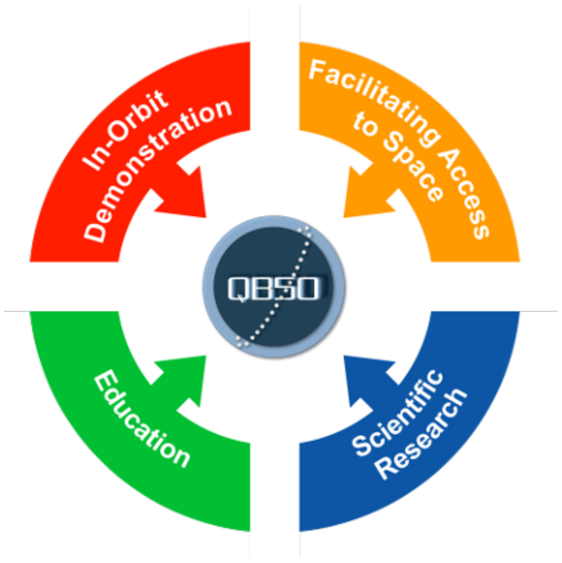


Figure 2.1: QB50 Mission Objectives [21, 1]

As depicted in figure 2.1, the four mission objectives are:

In-orbit demonstration (IOD). The first objective of QB50 project is to test the new technologies developed for nanosatellites.

Scientific Research. The second objective is to carry out atmospheric research of the lower thermosphere, between 200 to 380 km altitude, which is the least explored layer of the atmosphere. For that purpose, three experiments where designed, and each satellite is carrying at least one of them: Ion-Neutral Mass Spectometer (INMS), Flux-Φ-Probe Experiment (FIPEX) and the multi-Needle Langmuir Probe (m-NLP). Additionally, all three experiments are provided with temperature measuring sensors.

Facilitating Access to Space. The third goal, is to achieve a sustained and affordable access to space, through the publication of new standards and improvement of those already existing.

Education. Finally, developing and designing of satellites are left to university teams. That allows students to get in touch with space and be able to learn about satellite manufacturing and space engineering. [10] [21]

2.4 Aalto-2 Overview

The Aalto-2 is, as seen in figure 2.2, a 2U-CubeSat designed for the QB50 constellation. Its mission is to gather science data from a multi-Needle Langmuir Probe and download this data to the ground station for further processing. The satellite will be launched from the International Space Station; therefore, it will have a similar orbit. The ISS is orbiting at a perigee of 409 km and an apogee of 416 km, and an orbital inclination of 51.65°. [17]
Because of the atmospheric drag effect, the satellite orbit will decay in time. The mission is expected to last three months before satellite deorbiting.

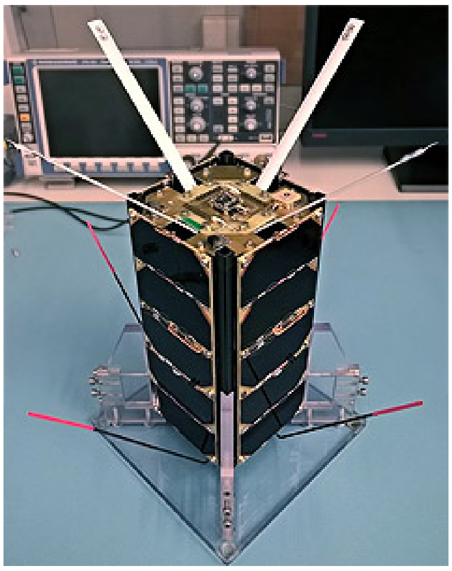


Figure 2.2: Aalto-2 Engineering Model

The mission can be divided into three phases:

1. Satellite first phase is bringing up all the systems, making sure they are working and deploy antennas and probes.

2. Second phase is the nominal phase which lasts until the end of the mission. The m-NLP is deployed along the ram side (the minor inertial axis of the body) of the satellite. During this stage, an amount of 2 MB of science data is going to be gathered and will be downloaded every day. All systems are working during this phase.

3. Finally a contingency phase or safe mode will be triggered if conditions are met, e.g. battery falls below 60% of capacity. This mode turns off the Attitude, Determination and Control System (ADCS), OBC and payload while leaving on the UHF communications module and beacon signal alive. [10, p.3]

2.4.1 Aalto-2 Computer Subsystem and Software environment

The Aalto-2 OBC board is designed as a double-redundant system (two OBCs in one PCB), managed by an arbiter. Each OBC has a Texas Instrument (TI) RISC flash micro-controller RM48L952PGE. The Central Processing Unit (CPU) is a 32-bit ARM-Cortex R4F with single/double precision floating point unit. The micro-controller has integrated 3 MB of program flash memory, 256 kB of RAM memory and 64 kB EEPROM of flash memory. It is also equipped with timer modules, General Purpose Input/Output (GPIO) and various communications modules. Communication with different modules of the satellite is carried out via CAN, SPI and I^2C buses, as depicted in figure 2.3. [10, p.43]

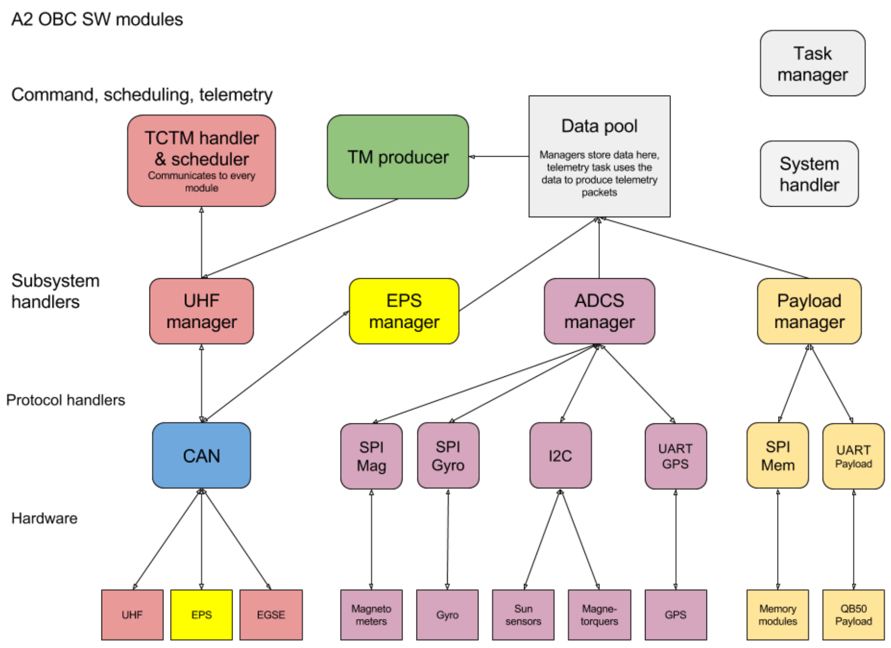


Figure 2.3: Aalto 2 OBC Software Architecture

The OBC software has a master-slave relation with other satellite modules, with the exception of the Electrical Power System (EPS). If the safe mode is reached, then EPS shall have privileges to send a shutdown signal to the OBC.

2.4.2 Aalto-2 Communication Subsystem

The UHF communication subsystem is a half-duplex 2-GFSK (Gaussian Frequency Shift Keying), modulated at 9600 bauds (which can be translated directly to 9600 bits per second or 1200 bytes per second) operating at 437.335 MHz (Amateur-Satellite Band) for transmission and reception. It uses AX.25 protocol with G3RUH modem as default protocol, with a more reliable option for a custom packet format. On the other hand, the UHF module also transmits a modulated Carrier Wave (CW) Morse code followed by Whole Orbit Data (WOD) every 1 minute. Additionally, the UHF is able to upload and download at least 2 MB of data. Its velocity will depend on the coverage area and ground station location at the time of transmission.

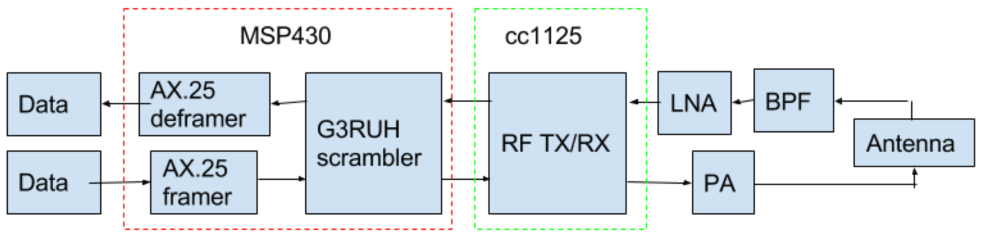


Figure 2.4: Aalto-2 UHF functional diagram [12]

The UHF uses a MSP430FR5739 low-powered micro-controller to switch between transmit and received signals, frame/deframe and scramble/descramble packets, and interface with the OBC as seen in the figure 2.4. This interface is conducted through CAN bus. Also, it uses a TI cc1125 sub-1 GHz radio module to facilitate baseband processing, modulation, mixing and RF functionalities. The subsystem is designed to transmit at 1.2 Watts. [12]

2.5 Aalto-2 Telecommunications General Requirements

The next paragraph present the requirements for building the Aalto-2 TC/TM Handler software.

Aalto-2 will be deployed in a Low Earth Orbit (LEO), where high satellite velocity (Doppler effect) and long period of no ground station contact (which depends on the number of ground stations). The communication software restrictions are: small communication period between ground station and satellite, and a large amount of data to be transmitted (up to 2 MB per day).

Other boundaries are set due to OBC architecture definition:

- The OBC has to be implemented within a real time control software,
- it must allow both interactive spacecraft remote control and automated control,
- the software concept shall use a service based architecture covering control and input/output, such as data input/output handlers and data bus protocols, control routines for payloads, thermal and power subsystems, which are up to failure detection, isolation and recovery routines.

The operations concept for the TC/TM of Aalto-2 must concern the command and control of payloads and platform via telecommands, and send generated data via telemetry messages. This software should be implement within international spacecraft communications standard, such as Packet Utilization Standard.
Finally, the mission software operation concept also has to be elaborated concerning ground station (GS) visibility, the ground station network, link budgets and time-line commanding from GS.

2.6 On Board Computer Software

Some concepts about OBC should be defined to understand the differences between these space micro-computers and normal computers as known on Earth.

The OBC is a vital component of every spacecraft. They should meet higher requirements than ordinary computers on Earth, such as high radiation resistance and high reliability. The hardware limitations are set by physical parameters (e.g. power supply) and satellite processing requirements (e.g. satellite management processes). Meanwhile, software shall fulfill all mission functionalities.

The steps to develop a complete OBC Software are:

First; the **Software functional analysis** considers the functions required for the spacecraft OBC software and the allocation of these inside the spacecraft modules. Among these functions definitions, there are the TC processing function and TM generating functions.

Second; the **Software requirements definition** includes the specification of all OBC software features, such as OBC software structure, data handling, scheduling, spacecraft modes, etc. These definitions may not force any software implementation solution.

Third; the **Software design** is greatly influenced by the chosen programming language.

Fourth; the **Software implementation and coding**.

Finally, the **Software verification and testing** is the last step for validating implementation performance and verifying required functionalities. [7, p.79-168]

This work does not focus on developing a full OBC software but an important piece of it.

2.7 Recommendations for Satellite Communications

The European Committee for Space Standardisation (ECSS) is an organisation created by several European space agencies and multiple space-related companies. Its goal is to create and distribute standards that help companies, agencies, and other organisations to develop spacecrafts.

2.7.1 Packet design and routing

Spacecraft-ground communication is based on an international command packet transmission, which is widely used in spacecrafts around the world. This international packet definition has been exposed by the Consultative Committee for Space Data Systems (CCSDS) in the [4] and in [5] standards recommendations.

2.7.1.1 Telecommand packet (TC)

The CCSDS TC packet, as seen in the figure 2.5, consists of a 6 byte long packet header and a packet data field with a minimum length of 6 bytes and a maximum length of 65526 bytes, that carries encoded information from the ground station to the spacecraft.

The Packet Header key fields are:

- **Application Process Identifier (APID).** The APID defines the routing or destination of the packet on board. Each computer or packet terminal has its own APID, which allows routing of packets by the main on board software (e.g. UHF, EPS, GPS, storage management, etc).

- **Packet Sequence Control.** This field helps the user to solve critical situations might be encountered during telecommand transmission such as having multiple sequentially commands issues or commands belonging to the same request. Two bytes are defined so on board software is able to differentiate them.

The Packet Data key fields are:

- **Data Header Field.** This packet field contains Service Type and Service Subtype, identifying which function is executed within a specific APID. It also encodes four acknowledge bits, that helps to make execution stage reports of the function as can be seen in figure 2.6.

- **Data Source.** This field contains the data for executing each service command. Its format depends on which service and subservice the command it is meant for.

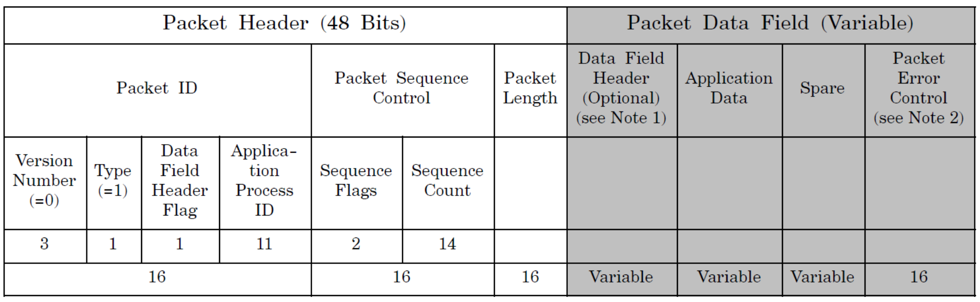
- **Packet Error Control (PEC).** The packet error control field encodes in two bytes an error detection code (e.g. Cyclic Redundancy Check (CRC)) used by the Telecommand and Telemetry handler to make validation of a telecommand source packet. [5]

2.7.1.2 Telemetry packet (TM)

The CCSDS TM packet, as seen in figure 2.7, is very similar to the TC packet. It differs only in a few fields.

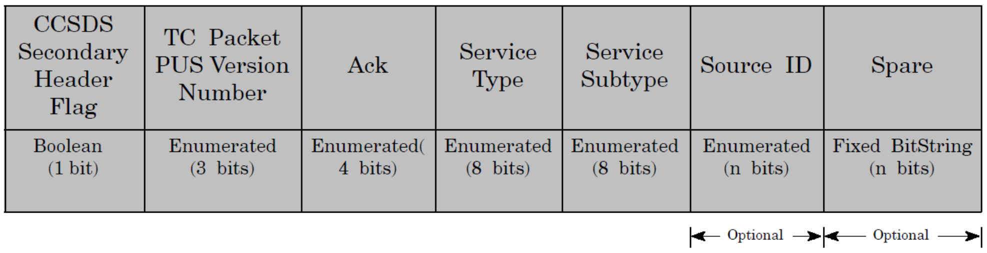
The Packet Header key fields are:

Exactly the same as telecommand. It differs in one bit, the *type* of Packet ID, which is 0 in the telemetry packet.



Packet Header (48 Bits)							Packet Data Field (Variable)			
Packet ID				Packet Sequence Control		Packet Length	Data Field Header (Optional) (see Note 1)	Application Data	Spare	Packet Error Control (see Note 2)
Version Number (=0)	Type (=1)	Data Field Header Flag	Applica-tion Process ID	Sequence Flags	Sequence Count					
3	1	1	11	2	14					
16				16		16	Variable	Variable	Variable	16

Figure 2.5: Telecommand packet fields as described by standard [5]



CCSDS Secondary Header Flag	TC Packet PUS Version Number	Ack	Service Type	Service Subtype	Source ID	Spare
Boolean (1 bit)	Enumerated (3 bits)	Enumerated(4 bits)	Enumerated (8 bits)	Enumerated (8 bits)	Enumerated (n bits)	Fixed BitString (n bits)
					Optional	Optional

Figure 2.6: Telecommand data header fields as described by standard [5]

The Packet Data key fields are:

- **Data Header Field.** There are a few differences from TC Data header and can be seen in the next figure 2.8. The most notable is the Time field which carries the time stamp value of the outgoing packet.
- **The other fields** are exactly the same as the TC packet data fields. The data source field can change depending on the number of optional fields that are used. An important difference is that PEC is not compulsory. [4]

Summarising, telecommand and telemetry packet definitions have a lot of fields in common, diverging mainly in the packet data fields. These standards definitions also limits the size of each field and the overall packet.

2.7.2 Packet Utilization Standard

The Packet Utilization Standard (PUS), complements [5] and [4] standards. PUS standard defines an application layer interface between satellite and ground station. It does not define mission-specific concepts, like payload application functions, therefore can be used in any spacecraft mission, no matter which its payload goal, orbit or ground station characteristics are.

Packet Header (48 Bits)							Packet Data Field (Variable)			
Packet ID				Packet Sequence Control		Packet Length	Data Field Header (Optional) (see Note 1)	Source Data	Spare (Optional)	Packet Error Control (Optional)
Version Number (=0)	Type (=0)	Data Field Header Flag	Application Process ID	Grouping Flags	Source Sequence Count					
3	1	1	11	2	14					
16				16		16	Variable	Variable	Variable	(see Note 2)

Figure 2.7: Telemetry packet fields as described by standard [4]

Spare	TM Source Packet PUS Version Number	Spare	Service Type	Service Subtype	Packet Sub-counter	Destination ID	Time	Spare
Fixed BitString (1 bit)	Enumerated (3 bits)	Fixed BitString (4 bits)	Enumerated (8 bits)	Enumerated (8 bits)	Unsigned Integer (8 bits)	Enumerated	Absolute Time	Fixed BitString (n bits)
					Optional	Optional	Optional	Optional

Figure 2.8: Telemetry data header fields as described by standard [4]

This standard is structured as a set of services functions to be provided by an onboard software. It was chosen for Aalto-2 due to QB50 recommendations. [16]

2.7.2.1 PUS Concept Operations

This section summarises the concept descriptions from PUS, which form the basis of the services definition.

- **Device commands.** A special group of telecommands set to be executed directly in the on board hardware. These requests grants the user a way to manipulate hardware registers, relays and electronic components directly. Device commands shall be used when regular telecommands are not being processed.
- **Telecommand verification.** Satellite shall be able to provide feedback on the execution of a received telecommand, so ground station knows telecommands execution status. The number of telecommands verification checkpoints depends on telecommand execution time.
- **Housekeeping data reporting.** Every spacecraft implements a

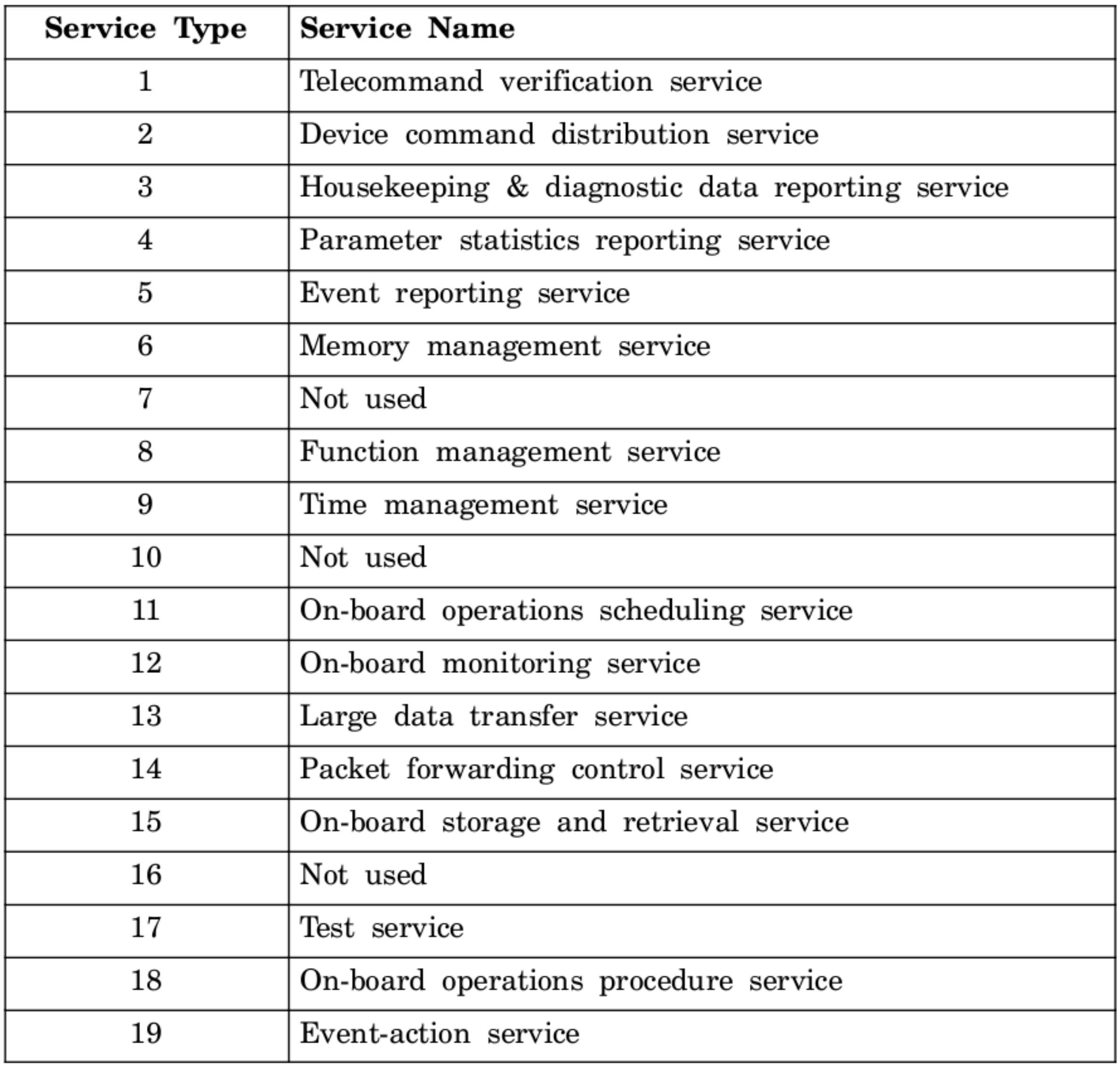
housekeeping data telemetry, so the user may be able to check the health of electronic components and check the status of the satellite.

- **Event reporting.** In addition to periodical reporting, spacecraft shall be able to send events of operational significance (e.g. notify the finishing of a payload experiment or critical problems detection).
- **Statistical data reporting.** Another method which reduces the quantity of housekeeping data gathered by evaluating their statistics values (mean, min, max, standard deviation) and reporting these to the ground station. This mechanism is appropriate for these spacecraft with long periods of no link with a ground station.
- **Time information.** Time is important in space. It is used for tracking the spacecraft along its orbit or scheduling payload experiments. PUS standard defines a time packet which is sent periodically to the ground station.
- **Software management.** PUS identifies a number of standard services that can be applied by different application processes (APIDs). These services are related to on board activities that are generic use by several space missions.
- **On Board operations scheduling.** It is a set of software functions used for storing execution operations that are being automatic executed in the future. The simplest form of on board scheduling is storing the set of functions to be executed and release them once on board time is reached.
- **On Board Monitoring.** Those capabilities of self-monitoring spacecraft parameters in order to reduce the amount of housekeeping data sent to the ground station.
- **On Board operations procedures.** These procedures are a set of other standard or mission-specific functions which are executed stepwise.
- **Attaching actions to on board events.** As an extension of event reporting, an action can be defined to be automatically executed whenever an event occurs.
- **On Board storage and retrieval.** The set of capabilities to store any data in the flash memory and dump it on request to the ground.
- **Telemetry generation and forwarding.** Communication network and downlink bandwidth are not unlimited resources. Therefore, the generation and forwarding of telemetry source packets shall be controlled and managed in order to use rationally these limited re-

sources.

- **Memory management.** Direct modification of memory is not a routine operation of a spacecraft. These procedures are used for troubleshooting support and on board software maintenance only.
- **Diagnostic mode.** A set of capabilities used for investigating troubleshooting situations and operations that differ from nominal procedures.
- **Off-line testing.** A number of functions set for troubleshooting and pre-operational validation purposes. Each application process shall provide a test function that can exercise under ground control.

The figure 2.9, depicts the services names of the explained concepts.



Service Type	Service Name
1	Telecommand verification service
2	Device command distribution service
3	Housekeeping & diagnostic data reporting service
4	Parameter statistics reporting service
5	Event reporting service
6	Memory management service
7	Not used
8	Function management service
9	Time management service
10	Not used
11	On-board operations scheduling service
12	On-board monitoring service
13	Large data transfer service
14	Packet forwarding control service
15	On-board storage and retrieval service
16	Not used
17	Test service
18	On-board operations procedure service
19	Event-action service

Figure 2.9: Service Table Summary [3, p.50]

The PUS standard covers a large set of situations and environment, giving a recommendation for dealing with each of them.

2.8 The Chomsky classification

The Chomsky hierarchy is a classification method for formal grammars. It was develop by the linguist Noam Chomsky in 1956.

2.8.1 Formal languages/grammars

A formal language or grammar is defined as a set of strings and symbols which can be constrained to a set of production rules.

As depicted in [6, 141], a grammar *G* is a semi-group under concatenation with strings in a finite set *V* (vocabulary of *G*), with symbols as its elements, and *I* as the identity element. *V* is conformed by the union of a vocabulary of terminal symbols, V_T, and a vocabulary of non-terminal symbols, V_N; $V = V_T \cup V_N$. V_T contains the identity symbol *I* and a boundary $\#$. V_N contains an element *S* (sentence). A two-place relation $\rightarrow$ is defined on elements of *G*, read "can be rewritten as". This relation satisfies the following conditions:

1. $\rightarrow$ is irreflexive.
2. $A \in V_N$ if and only if there are φ, ψ, ω such that $\varphi A \psi \rightarrow \varphi \omega \psi$.
3. There are no φ, ψ, ω such that $\varphi \rightarrow \psi \# \omega$.
4. There is a finite set of pairs $(x_1, \omega_1)...(x_n, \omega_n)$ such that for all $\varphi, \psi, \varphi \rightarrow \psi$ if only if there are φ_1, φ_2, and $j \leq n$ such that $\varphi = \varphi_1 x_j \varphi_2$ and $\psi = \varphi_1 \omega_j \varphi_2$.

For convention, the capital letters are used for strings in V_N; small Latin letters for strings in V_T; Greek letters for arbitrary strings; early letters of all alphabets for single symbols (members of V); late letters of all alphabets for arbitrary strings. [6, 141]

2.8.2 Chomsky hierarchy

The Chomsky hierarchy is divided in four different types:

Type 3 languages or regular grammars, are those languages that can be depicted by a finite state automaton machine. A finite state machine is defined as a set of constant states that change from one to another one following specific relations. These states cannot be modified and by themselves they cannot store or remember new states. All syntax of formal languages can be depicted as finite state machines.

Type 2 or context-free grammars, include those languages that can be pictured by a non-deterministic pushdown automaton. The pushdown automaton can be defined as a FSM employing a stack. Now it can remember new data but until the stack is filled. In other words, its operations are limited by the size of the stack. Its the most common type among programming languages.

Type 1 or context-sensitive grammars, include the languages that can be recognised by a linear bounded automaton (a non-determinative Turing machine bounded between two symbols, an starting symbol and end symbol). The linear bounded automaton is a FSM with a limited memory from which can be read and written. Overwritten is not allowed, therefore it has not unlimited memory.

Type 0 or unrestricted grammars, includes all previous grammars and can be recognised by a Turing machine. Turing machines have no restrictions. These machines theoretically have infinite size memory available. Of course, there is not a machine capable of that, but by adding mechanism it can be achieved infinite virtual memory (e.g. allow overwriting). [6, 143-152]

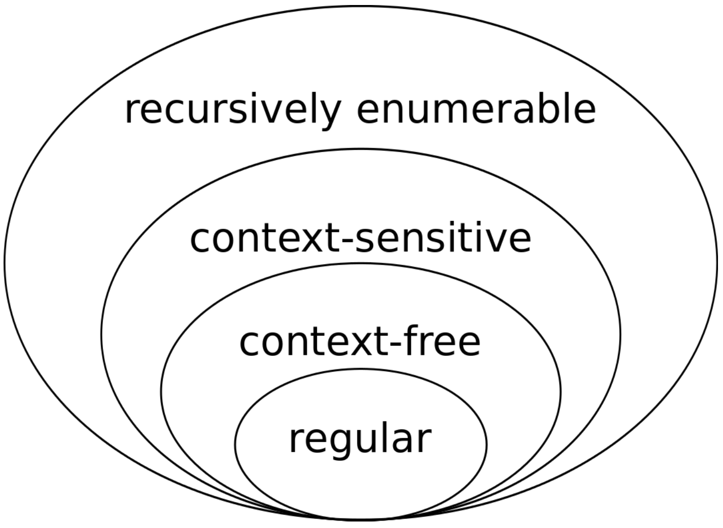


Figure 2.10: Set of inclusions of the Chomsky hierarchy [6]

2.8.3 Which is the standard TC/TM?

The most common TC/TM software structure that it is found in spacecrafts is the finite state machine automaton or object/agent oriented machine. Telecommands can be sent, then these are processed/executed by the satellite, and finally getting a reply according to a define output. [7, 136-147] In other words, it emulates a regular grammar that can send regular expressions and that applies to a certain production rules, which at the same time, generate other regular expressions that are sent back. This is usually modelled as a FSM with no memory, because the states are defined in the program and cannot change. The TCs are only capable of changing from one state to another. However, most of the spacecraft have scheduling capabilities, which means that they implement deferred execution. Forcefully, this means that now memory is needed to stored schedule TCs. But what usually happens is that instructions are programmed as a state to be reach periodically in time. So, this repetition is done with a state and there is no need for memory to do it. Summarising, depending on the type of scheduling implemented the TC/TM will be one type or the other. However, the most common are type 3, type 2 and type 1.

2.9 Concatenative Languages

The most common languages used in satellite software development are C/C++, Assembler and Ada95. These languages are chosen depending on the requirements and constraints of the space project (e.g. high architecture versus machine like architecture versus object-oriented architecture). Also, a lot of space project teams rely on previous successful missions; the vast majority of them uses one (or more) languages. [7, p.135-136] Firstly, these programming languages are used for firmware level programming, in other words, to program the actions, functions. On second matter is the execution flow level, meaning how can be depicted the whole TC/TM process. As said in 2.8.3, most of the satellites use FSM machine with or without memory. At this level of abstraction is where the concatenative language is going to be applied instead of a simple FSM. The main reason concatenative language was selected for the TC/TM is to do experimentation and find new mechanisms for space communications. Also, the approached model allows to simplified the TC/TM software to a single module that usually will need two or three modules to perform the different functions. Finally, it is imperative to keep the safety and robustness as typical models do.
Concatenative programming languages are those that denote every expression as a function (e.g. number values are functions that have no inputs and return themselves), and the juxtaposition of functions as function composition, therefore a concatenative program can be always simplified to a single function. [19] [14]

The most significant characteristics of a concatenative language are:

- A concatenative language is able to return multiple values from a function, not just tuples,
- function composition is implicit,
- data flows in the order the functions are written in,
- ”quotations” are deferred composition of functions (e.g. ”[2 >]” is a function that returns something whether the argument is greater

than two or not),

- these languages are usually implemented using a stack.

Factor is a well known concatenative language that employs a stack for execution. It is a programming language that uses a shell interface and scripts. It uses prefix notation (from left to right), and quotations are surrounded by two square brackets ([]). For the following example in figure 2.11, the word ”USING” is used for importing different word libraries into the current shell, and the word ”bl” means the comparison ”below than” or ”¡”. [8] [14]

```
USING: kernel random math io ;
 [ "hi" write bl 10 random zero? not ] loop
hi hi hi
```

Figure 2.11: Example of a concatenative ”loop” in Factor [8]

The previous figure 2.11 shows an example of the ”loop” using a concatenative language.

```
IN: kernel
: loop ( ... pred: ( ... -- ... ? ) -- ... )
    [ call ] keep [ loop ] curry when ; inline recursive
```

Figure 2.12: ”loop” word description in Factor [8]

The figure 2.12 pictures the description of the program (or word) ”loop” in Factor. The definition depicts that the quotation is going to be call until the result returns ”false”. It has a quotation as an input that must look like the description between parenthesis from figure 2.12. The first element of this parenthesis will be the function (word) to run (in figure 2.11 is ”write ’hi’”) if the condition is true. The second part, is the condition itself (in the example figure 2.11, it is the quotation ”bl 10 random zero? not”; which translates to ”while a random generate number below than zero, return true, otherwise do nothing”).

2.10 Ground Station Domain Specific Language

At the lately stages of this work, it was needed a software interface between the workstation and the OBC to perform more reliable and accurate testing. Therefore, a ground station software was programmed as an internal Domain Specific Language (DSL) in Python. This DSL translates shell commands to the PUS telecommands which are sent to the satellite. Then, the satellite deciphers these commands, executes them and replies back. This software was used for satellite software testing and it will be used for future satellite monitoring operations.

This ground station DSL machine works in parallel with the TC/TM software. It means that only one can run while the other awaits for the other to finish. It is programmed following the same design as the TC/TM Handler, as both use the same functionalities.

Chapter 3

Implementation of Communication Handler

The next chapter explains the TC/TM architecture design that uses concatenative language features, combined with native C and FreeRTOS. The designed architecture is divided in two structures: the *Listener* and the *Interpreter*. The first one, the *Listener*, is designed and programmed as finite state machine that communicates with the *Interpreter*, which is programmed as Turing machine.

3.1 TC/TM Handling Implementation

The telecommunication handler of the Aalto-2 is built in two different parts: the *Listener* and the *Interpreter*.

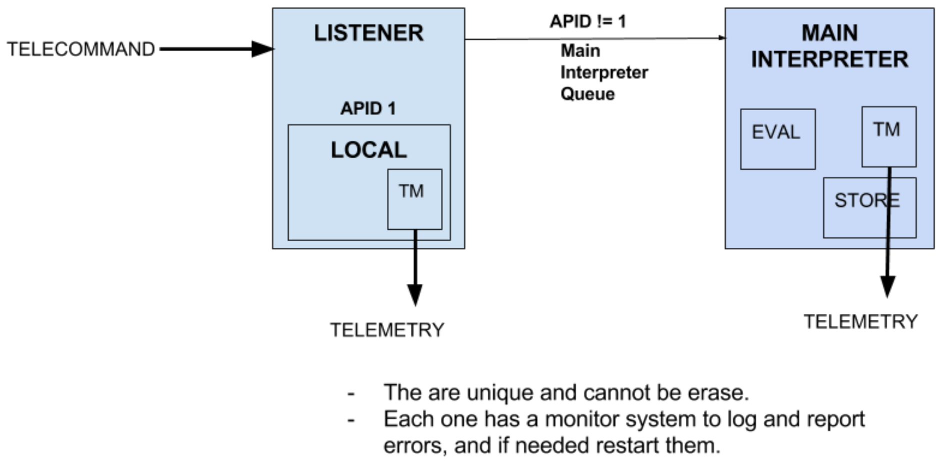

TELECOMMAND
LISTENER
APID 1
LOCAL
TM
TELEMETRY
APID != 1
Main Interpreter Queue
MAIN INTERPRETER
EVAL
TM
STORE
TELEMETRY
- The are unique and cannot be erase.
- Each one has a monitor system to log and report errors, and if needed restart them.

Figure 3.1: TC/TM Handler Architecture

In figure 3.1, the two software structures are being represented: the first one is the *Listener*, whose function is to receive and check incoming telecommands. Additionally, *Listener* implements a few services procedures. The second, is the *Interpreter*, which interprets a command or group of commands, routing them to the targeted module to be executed.

Both pieces use external C libraries that implement different key procedures. The most relevant one is the Dyncall library. The library is a group of files which implements run-time function calls in C. Another important library is the Stack library. It implements C functions and structures to create and manage a byte stack. Additionally, more libraries where build in order to meet the requirements, among those libraries the most relevant are, the System Monitor, which is a set of functions to that implements task watchdog. Finally, the file system support libraries, the Coffee File System (CFS) and the Raw File System (RAW); they both implement methods to manage the OBC file system.

Furthermore, there are some concepts/variables that are needed to be exposed before *Listener* and *Interpreter* function explanation.

First of all, the **SatFunction** variable and **Format** variable: the first one contains the satellite C function address, meanwhile, the second one stores the signature arguments. Both variables are stored in an encoded dictionary-like array that has all the satellite functions a specific APID can execute. A second important variable is the **StackByte**. The variable stores the location and position of the SatFunction arguments. When a command is called, its arguments can be located either inside the telecommand application data field that encoded that command or inside the satellite stack. Additionally, the last bit defines if ground station request immediately telemetry of the result after telecommand execution.

Finally, the C function *evaluate_single* takes SatFunction, Format and a stack as arguments and it is tasked to evaluate and execute the encoded telecommand, returning a result of this execution as shown in figure 3.2.

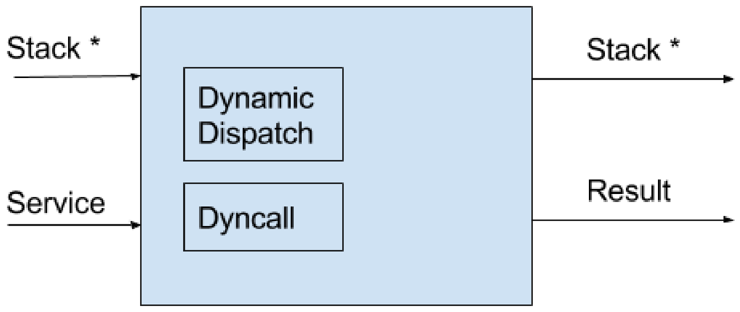


Figure 3.2: *evaluate_single* block diagram

3.1.1 The *Listener*

This first piece of the TC/TM handler is the receiving of a packet and checking if it is a correct telecommand. These procedures are performed by a FreeRTOS FSM task called *Listener*.

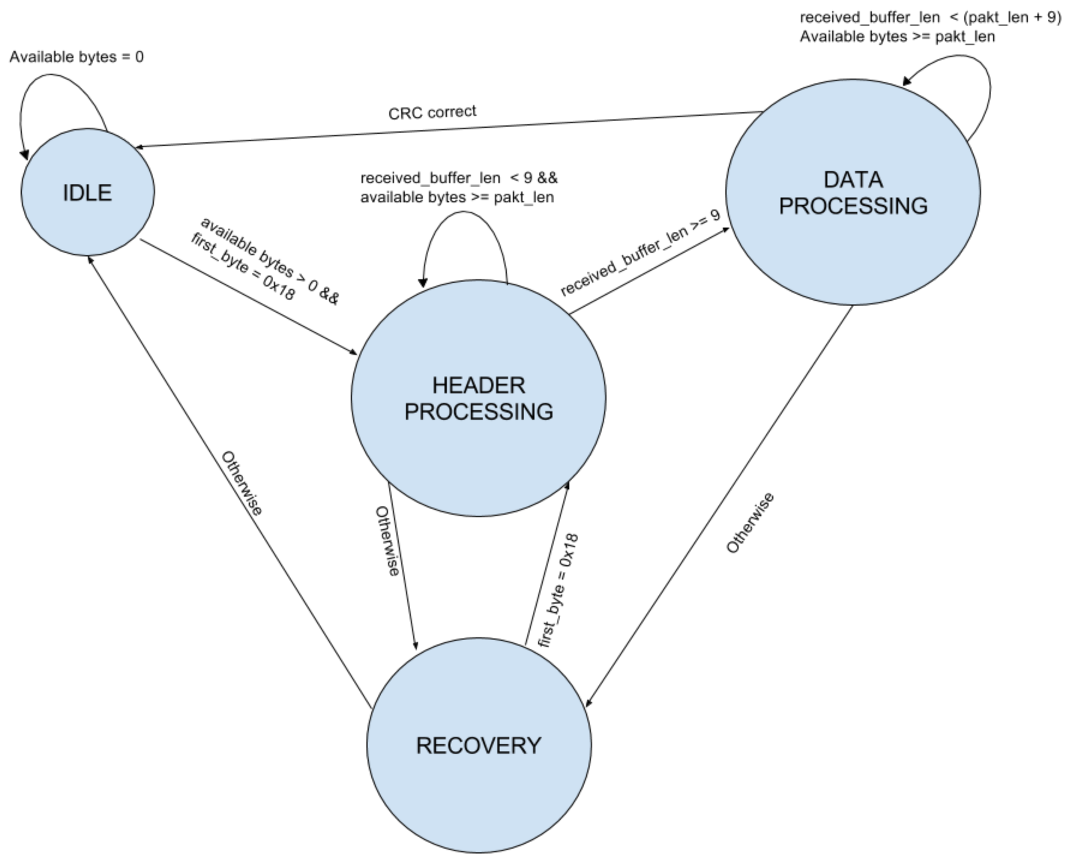


Figure 3.3: *Listener* Finite State Machine

The designed software can be depicted, as seen in figure 3.3, as a state machine with four basic states:

Idle: is the starting point where the *Listener* expects a command from the GS. The satellite receives byte per byte; therefore, the *Listener* remains at this state until there is at least one byte encoded as the hexadecimal value 0x18 (which corresponds with the first byte of the Packet ID format field). Finally, the *Listener* understands that this first byte may be the beginning of a TC. Then it changes the state to HeaderProcessing. Otherwise, it remains at Idle state.

HeaderProcessing: receives the rest of the header bytes and creates a command structure with APID, service, subservice and packet length. Then it checks if these parameters are inside the correct boundaries of the satellite definitions (e.g. maximum packet length, APID maximum number, etc). If all of them are correct, the *Listener* knows that the TC has a correct Header structure, and state is set to DataProcessing. Otherwise is set to Recovery state.

DataProcessing: is reached when the *Listener* has a correct PUS Header. While at this state, the *Listener* receives the rest of the packet (depending on the packet length value) and sets the following variables: Stackbyte, application data and PEC. Then it calculates a new checksum, and checks it with the one stored in PEC variable. If both variable are identical, then the *Listener* is sure that the message is a correct PUS format TC. Once a TC is correctly identified, the command can follow two different paths depending on APIDs value. If APID is 1, it means this command is meant for the *Listener* and will be execute immediately after. On the other hand, if APID is different from 1, the TC is meant for other application processes inside the satellite, and telecommand message is sent to the Main Interpreter. Either way, state variable is set to Idle. If the *Listener* has found that this message PEC is different from the calculated one, or an error happened during variable assigning process (e.g. *malloc* failed), the message is discarded and state is set to Recovery.

Recovery: is the state triggered when a problem has occurred in HeaderProcessing or DataProcessing. There are many different situations that triggered this state (e.g. wrong value in TC, timeout reached, run out of RAM, etc). Despite the different situations, the *Listener* Recovery has one execution direction: first discard the processed TC (if wrong), then, if there is more byte data waiting to be taken, tries to find if there is a second TC by searching the 0x18 hexadecimal value. If this value is found, HeaderProcessing state is set, so the *Listener* can process a possible TC. Otherwise it reaches the end of the buffer. Despite the encounter situation, the *Listener* generates a failure report (an error happened), and at the end, it resets local variables and state to Idle.

A simple procedure has been fixed to catch incoming telecommands, divided and checked them. All telecommands go through the *Listener* task, which will not be deleted or stopped at any point.

Finally, the *Listener* task can execute some telecommands locally to manage the satellite beacon, state reporting tasks and Interpreter management.

3.1.2 The *Interpreter*

The *Interpreter* is also designed to be a FreeRTOS task, which carries command evaluation and execution. This task is designed following a concatenative programming approach and implemented in native C.
C is an imperative language with some characteristics opposed to concatenative languages. For example, C native functions cannot return multiple output, therefore functions with multiple output cannot be implemented. Another feature is that C is typed-check language and concatenative stack does not care about type, so it must be solved in a way that variables have and do not have types. This paradigm forces a decision to keep the satellite functions native C, while the program flow is concatenative. Despite the antonymous characteristics, the final *Interpreter* design follows a concatenative programming approach and execution flow. Additionally, it was taken advantage of using PUS scheme and implement those services repeated in several different APIDs just once. Therefore, all APIDs will share these services without the need to be code repeated. Overall, this is quite natural implementation, because PUS naturally inherits dynamic dispatch procedure similar to the one from object-oriented programming languages.
Also, the software is coded within a FreeRTOS operating system (written in C), which implements useful mechanisms such as Queues, Semaphores, task control functions and kernel control functions, among others.

Among the different implemented elements, the deferred execution and quotations are two that required more effort. The next two paragraph explains its implementation.

Deferred execution means to postpone the execution of an element inside the stack. It is needed to use the scheduling services. The way it is implemented is by using functions that delay in time and in the stack a quotation function (using quotations makes available the possibility of calling any group functions). This function creates a new task that sleeps until the required time moment is reached, then it pushes (if available) the quotation at the top of the stack to be executed immediately.

The other elements are the quotations. Quotations are used to create a function made of other functions and place it on the stack. When the "interpreter" function (*evaluate_single*) takes a quotation command, the quotation calls another "interpreter" function which will call the real function stored in the command and the next one, as wrapping functions inside a function.

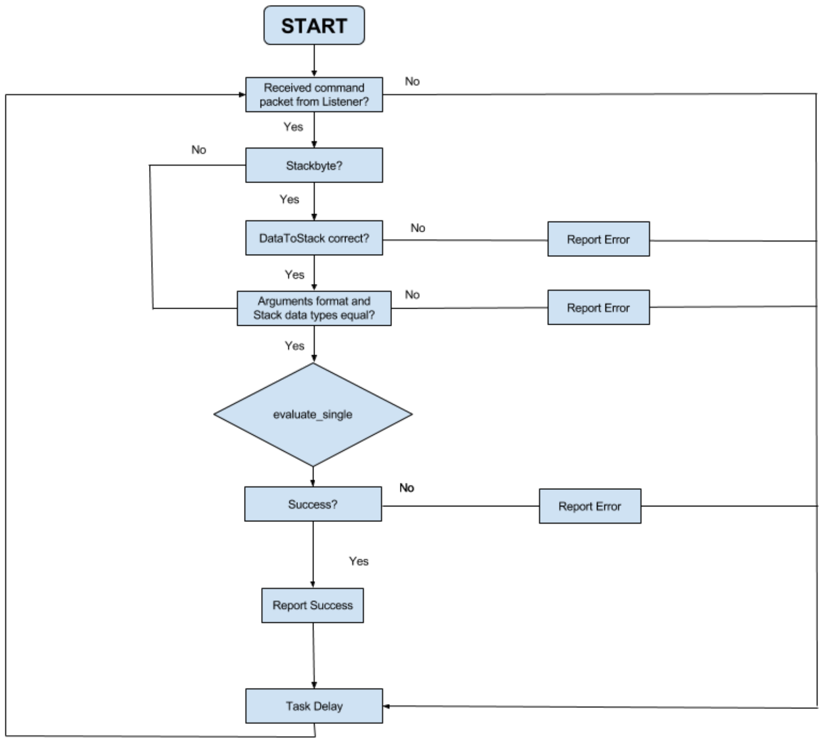


Figure 3.4: *Interpreter* flow chart

As shown in figure 3.4, the *Interpreter* task performs as follows:

1. The first step is receiving the telecommand packet structure from the receiving Queue. If this action is not successful, it means there is no packet waiting in the Queue. Then the *Interpreter* sleeps for a few milliseconds and tries again.

2. If a packet was received, the *Interpreter* uses APID, service and subservice to get a SatFunction address and Format. If combining these three values there is not an encoded Function, it will return NULL and discard the packet.

3. Before encoded function executing, the *Interpreter* passes the function arguments (if any) from telecommand application data field to the stack in the same order as they are given set by Stackbyte value.

4. Now that the *Interpreter* has the SatFunction address, its Format and arguments inside the stack, it calls an evaluation function that runs the requested SatFunction using Dyncall library.

5. Finally, once evaluation returns a Result (an updated stack and status), checks if execution was successful or not. If it was, then reports a successful telemetry and writes the same in the log file. Otherwise, it generates an error telemetry report which is sent to the GS.

3.2 Aalto-2 TC/TM Chomsky classification

In section 2.8.3, general TC/TM software was classified using the Chomsky hierarchy. Now, the same analysis should be performed with the TC/TM implementation of Aalto-2. As explained in previous section 3.1, the TC/TM is composed by two structures: the *Listener* behaves as a regular grammar, that receives TC that can change the state, in other words, a type 3 machine. On the other hand, the *Interpreter* employs memory, rising its category to type 2. This memory is structured as a stack-byte which stores elements of any type.Additionally it can do composition of functions by employing quotations. As this stack can have multiple modification from functions, other TCs and the interpreter, it can be seen as a random access memory. Therefore, type 1 is reached, because the memory can be modified. Finally, if overwritten is allowed, TCs can delete and write in the memory, theoretically having infinite virtual memory and reaching type 0. Summarising, the TC/TM software is a finite state machine that communicates with a Turing machine.

But the overall TC/TM structure is the combination of both models. Each TC is sent as a message that is first process by the *Listener*, then this command is processed and execute by the *Interpreter* and then it replies back. The process could be seen as FSM of three states (receive, interpret and reply). But that is a short description, because it does not take into count that interpreter has memory. Watching the big picture, the *Listener* FSM is just a function executed at the start whenever a TC is received, meaning that the TC/TM is categorised as the *Interpreter* model, which is type 0.

Chapter 4

Testing

The next chapter introduces the testing methodologies used to detect problems and bugs as well as the modifications made. Also, these procedures are validating and verifying the implemented code, checking that all required functionalities are implemented.
The two chosen techniques are: the Software in the loop (SIL) testing and the Overall testing.

4.1 Software-in-the-loop testing

A designed software will not be qualified until implementation and testing is concluded. Therefore, software in the loop testing techniques, grants testing and correction at many levels. The first tests were performed while implementation was not yet finished, in order to detect problems and correct them in small parts of the code. It detected issues that API or compiler could not. Finally, there was used to verified pieces of the implemented software.

The test was performed connecting a real OBC module to the workstation using a serial RS-232 connector. Communication with the development board is performed with a software program called Hercules SETUP utility. The Hercules is a HW-Group serial terminal port (RS-232) which can send raw hexadecimal commands. Also, an extra bus connection was needed between the CCS program (at the workstation) and the OBC to flash and debug the software. [9] Unfortunately, the Hercules program has not a very good interface and was used for small and simple tests that required very little pre-processing.

The figure 4.1, depicts the configuration.

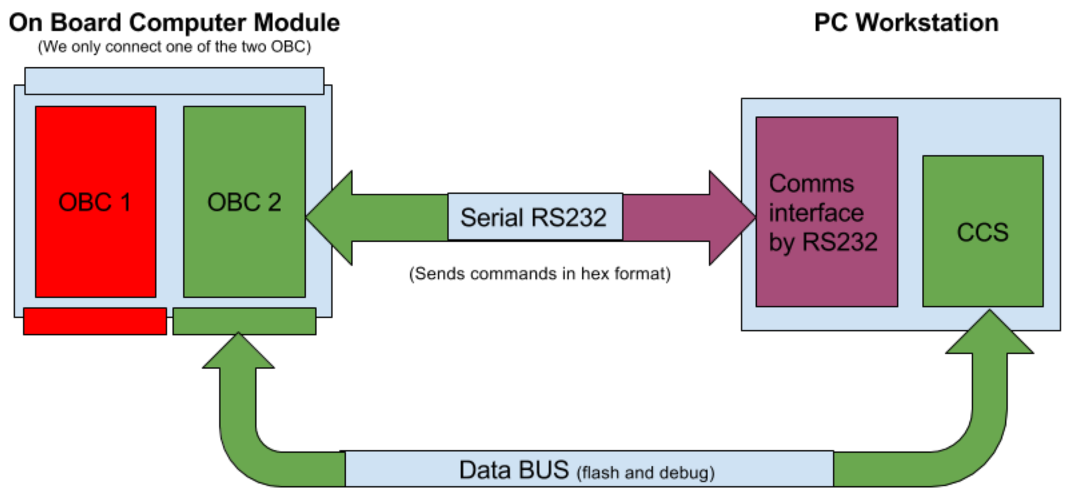

On Board Computer Module
(We only connect one of the two OBC)
PC Workstation
OBC 1
OBC 2
Serial RS232
(Sends commands in hex format)
Comms interface by RS232
CCS
Data BUS (flash and debug)

Figure 4.1: Software in the Loop Setup

The testing software utility used as substitution of the Hercules SETUP program was a Python Domain Specific Language (DSL) program, that simulates a ground station software. Moreover, the Monitor System was enabled for the *Listener* so it could be used for detecting execution problems as it will do once deployed. It uses log files to record any violation of the state machine rules and the level of seriousness of it.

4.2 Overall test

The On Board Computer software manages several tasks at the same time. These functions are synchronised through the task scheduler, which regulates the running tasks depending on the priority and the readiness of each. This test is performed using the maximum number of task/procedures should be able to manage at the same time by the satellite software, checking the adaptability and performance of the developed software with the rest of the system.

For this testing was chosen critical situations. Short commands are unlikely to encounter major problems, on the other hand, sending large commands and files can become a quite serious problem, e.g. sending OBC bootloader command or payload scripts. Therefore, the testing procedure consisted in sending consecutive big dummy files through the radio link and verification of the files were not corrupted. The inverse testing was also performed; instead of uploading a large file, a short telecommand made a request to download a large dummy file. The telecommand generation and telemetry reception was left to the Ground Station software, which was implemented, as explained in 2.10, using a Python DSL.

The following paragraph shows the script codes performed tests using the GS software and their results:

The different tests performed using the GS module are: uploading a file (*FileUpload(slot,filename)*), download some bytes from a file (*FileDownload(slot,size,filename)*) and check for errors from the monitor system (*check_monitor()*).

Options:

TC = Telecommand.

TM = Telemetry requested.

SB = StackByte.

IN = Input.

OUT = Output.

RET = allows to connect the script result values with the Python environment.

I32, UIN8, etc = variable type.

FileUpload(slot,filename) creates a set of commands to upload a file into the satellite storage system. The ground station shell will look like this:

TC ToStackString SB[1 2] IN[ARRAY[FILE "FILENAME" UI32 SIZE]] TM;

TC MakePointerToLast TM;

TC AppendRawFile SB[1 2 16] IN[ARRAY[UI32 SLOT UI32 SIZE]] TM;

TC DROP SB[16] TM;

The first command puts the input data in the stack. The second command creates a pointer to the beginning of the previous string. Third one sends the *slot* and *size* of the file, writing into *filename* location. Finally, it sends a request to drop the string address left. Figure 4.2 represents the stack movement:

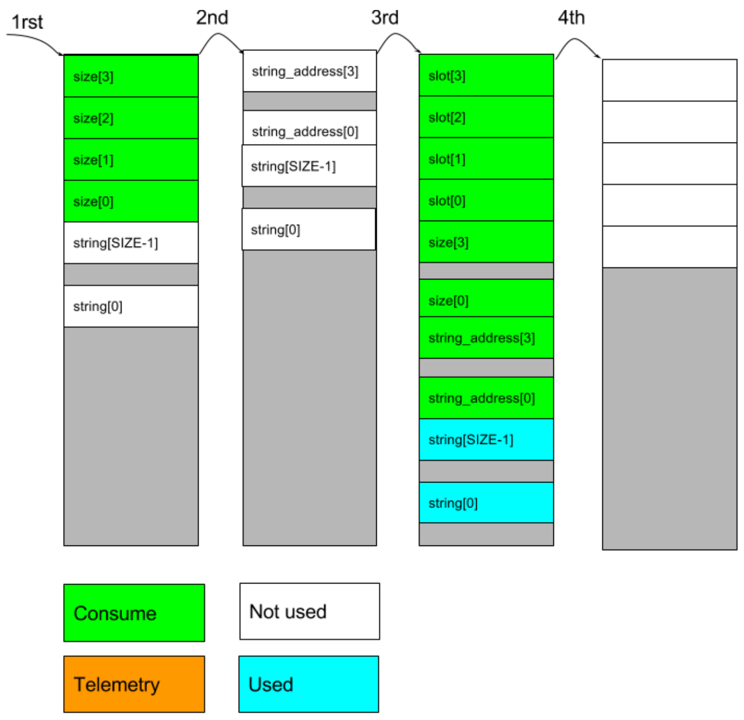


Figure 4.2: Stack variation per command executed

FileDownload(slot,size,filename) creates a set of commands to download a file from the satellite storage system.

TC ReserveStackSpace SB[1] IN[UI16 SIZE] TM;

TC ReadRawFile SB[1 2 3 16] IN[ARRAY[UI32 SLOT UI32 OFFSET UI32 SIZE]]TM;

TC DROP SB[16] OUT[FILE FILENAME] TM;

The first command sends a request to reserve space at the stack. The second command, reads data from *slot* (starting at *offset*) of size *size*. The last command, frees the variable left in the stack.

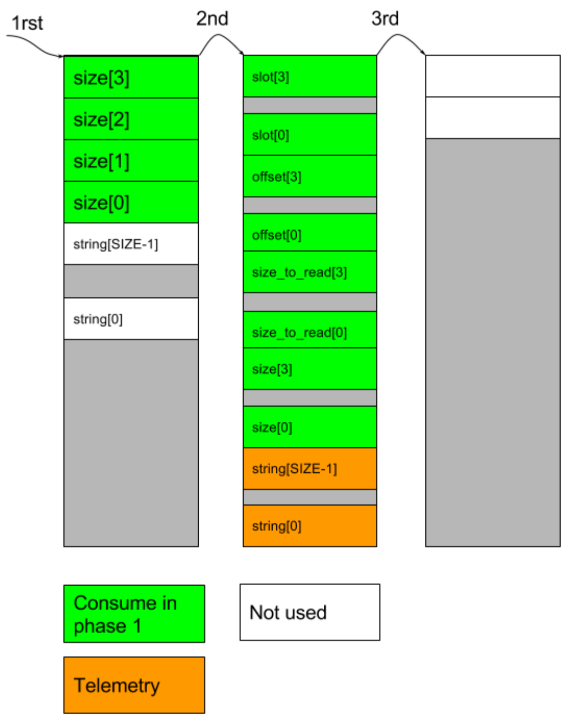


Figure 4.3: Stack variation per command executed

Finally, *check_monitor()* checks the log of the system monitor.
First part:

TC smBufferAvailable SB[16] OUT[UI32] TM RET;

Second Part:

TC ReserveStackSpace SB[1] IN[UI16 SIZE] TM;
TC smBufferRead SB[2] IN[UI32 SIZE] TM;
TC DROP SB[16] TM;
TC Nop SB[16] TM;

The first part sends a command to check the available space at the monitor system and sends back a telemetry of it. Second part, sends a command to reserve space at the stack for reading. The third one reads the information inside the monitor and pushes into the stack and then sends it to the GS. It is important to look at the SB option setting the position where the input variable is going. The last two commands drop the data left in the stack.

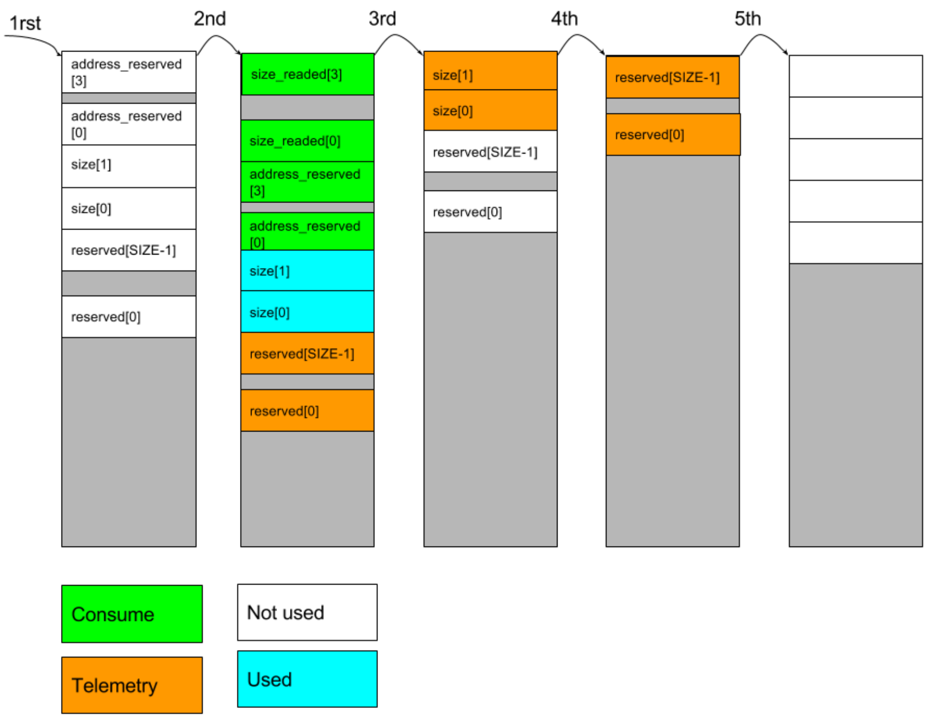


Figure 4.4: Stack variation per command executed

From the test results, it was found issues related to the timeouts. The *Listener* Recovery state timeout takes really long time ($\tilde{2}$ minutes) to recover when the received packets are large, because timeout depends on the number of received bytes (max. 65536 bytes). Therefore, if the maximum packet length is limited to 256 bytes, the top recover time falls ($\tilde{2}$ seconds) which is acceptable. However, larger packets are going to be divided in pieces of 256 bytes or less before sending them.

Testing goals for validating the implemented code (Software in the loop testing) and testing it within the OBC module in real cases (Overall testing), considered usage of hardware resources.

Chapter 5

Conclusions

Among all possible telecommand and telemetry handler software architecture design approaches, an unconventional one was chosen for the Aalto-2 satellite. The goal was to implement and test validation of the telecommunication requirements for Aalto-2 satellite and verification of TC/TM Handler software. The TC/TM Handler was programmed for the Aalto-2 Satellite OBC as Chapter 3 explains. The software in the loop testing and the overall testing were performed to verify and validate the code.

Firstly, a novel approach was developed for processing telecommands (evaluate), which uses a concatenative programming oriented design implemented in C/FreeRTOS. Analysing these implementation, it is found belonging to a Type 0 Chomsky grammar.

Secondly, a simple finite state machine was build to support the Interpreter structure call Listener, which function is to received, check and route incoming commands. The analysis of these structure reveals that it is a Type 3 Chomsky grammar.

Combining both structures, a TC/TM Handler is build, which satisfies the Aalto-2 communication requirements and functionalities.

Finally, some recommendations and suggestions for further work are made after the latest testing results:

The TC/TM software is being executed inside Aalto-2 OBC software and it will fulfill its purpose once the satellite operates and performs its mission successfully in the real environment.

The ground station equipment needs the interaction of a person-user to sent and received most of the commands. Therefore, further work in automatising the satellite-ground station communication could be done.

The satellite autonomy system is simple. The user can upload scripts and schedule them in time. Improvements (such as implementing other PUS services or enabling quotations for scheduling) can be done to achieve more satellite autonomy and better overall performance.

Scheduling are not implement. There is a simple implementation that performs scheduling but it is needed a more complete function. However, building this functions are not a big problem, because the TC/TM model facilitates the scheduling execution. In other words, the prime materials are there and only needs someone to do it.

Bibliography

[1] *CCSDS 100.0-G-1, Green Book, Telemetry Summary of Concept and Rationale*. CCSDS, Frascati, Italy, 1987.

[2] *CCSDS 200.0-G-6, Green Book, Telecommand Summary of Concept and Rationale*. CCSDS, Frascati, Italy, 1987.

[3] *ECSS-E-70-41A*. ESA Publications Division, Noordwijk, The Netherlands, 2003.

[4] *CCSDS 132.0-B-2: TM Space Data Link Protocol. Blue Book. Issue 2*. CCSDS, National Aeronautics and Space Administration, Washington DC, USA, September 2015.

[5] *CCSDS 232.0-B-3: TC Space Data Link Protocol. Blue Book. Issue 3*. CCSDS, National Aeronautics and Space Administration, Washington DC, USA, September 2015.

[6] CHOMSKY, N. On certain formal properties of grammars. *Information and Control 2* (1959).

[7] EICKHOFF, J. *OnBoard Computers, OnBoard Software and Satellite Operations*. Springer, Institute of Space System, University of Stuttgart, Germany, 2011.

[8] FACTOR-COMMUNITY. Factor wikipedia. `http://factorcode.org/`, (accessed on March, 2016) 2016.

[9] HW-GROUP. Hercules setup utility. `http://www.hw-group.com/`, (accessed on March, 2016) 2015.

[10] JOVANOVIC, N. Aalto-2 satellite attitude control system. Master's thesis, Department of Electrical Engineering and Automation, School of Electrical Engineering, Aalto University, Espoo, Finland, 2014.

[11] JOVÉ, R. *Contribution to the development of pico-satellites for Earth observation and technology demonstrators*. PhD thesis, Department of Signal Theory and Communications, Universitat Politécnica de Catalunya, Barcelona, Spain, 2015.

[12] KUHNO, J. Aalto-2 uhf module report. Project Documentation, 2016.

[13] LEE, S., HUTPUTANASIN, A., TOORIAN, A., WENSCHEL, L., AND MUNAKATA, R. *CubeSat Design Specification*, revision 10 ed., August 2007.

[14] LLC, B. *Concatenative Programming Languages: Forth, Postscript, Factor, Cat, Hartmann Pipeline, Joy, Colorforth, Concatenative Programming Language*. General Books LLC, 2010.

[15] LTD., R. T. E. Freertos api. `http://www.freertos.org/`, (accessed on January, 2016) 2004.

[16] MASSON, L. Recommendation for flight software implementation. Tech. rep., QB50, 2014.

[17] NASA. Reference guide to the international space station: Utilization edition. *`http://www.nasa.gov/`* ((accessed on September, 2016) 2015).

[18] PRAKS, J., KESTILÄ, A., HALLIKAINEN, M., SAARI, H., ANTILA, H., JANHUNEN, P., AND VAINIO, R. Aalto-1, an experimental nanosatellite for hyperspectral remote sensing. *IGARSS* (2011).

[19] PURDY, J. The big mud puddle: Why concatenative programming matters. *`http://evincarofautumn.blogspot.com`* (2012).

[20] RIWANTO, B. A. Cubesat attitude system calibration and testing. Master's thesis, Department of Electrical Engineering and Automation, School of Electrical Engineering, Aalto University, Espoo, Finland, 2015.

[21] THOEMEL, J., SINGARAYAR, F., SCHOLZ, T., MASUTTI, D.AND TESTANI, P., ASMA, C., REINHARD, R., AND MUYLAERT, J. Status of the qb50 cubesat constellation mission. *International Astronautical Congress* (2014).