Book Title

Under the Supervision of Prof. Dr. ...

Authors

MemberOne

MemberTwo

MemberThree

MemberFour

MemberFive

MemberSix

MemberSeven

MemeberEight

TABLE OF CONTENTS

List of Acronyms 3

CHAPTER ONE: Introduction 4

What is a CubeSat 4

Time-Triggered Architecture #

Other CubeSats: SwissCube and UPSat #

Our System Structure #

CHAPTER TWO: EPS Subsystem #

Introduction #

Functional Description #

Static Software Design #

Dynamic Software Design #

Hardware Implementation #

CHAPTER THREE: ADCS Subsystem #

ADCS Mission #

How it works #

ADCS Software Static Design #

ADCS Software Dynamic Design #

Hardware Design #

CHAPTER FOUR: COMMS Subsystem #

Mission #

System Design #

The Hardware #

System Realization #

CHAPTER FIVE: OBC Subsystem #

Requirements #

Architecture #

Hardware #

CubeSat Modes of Operation #

Communications Protocol #

On-Board Computer Software #

LIST OF ACRONYMS

OBC	On-Board Computer
GS	Ground Station
ADCS	Attitude Determination and Control Subsystem
COMMS	Communications Subsystem
EPS	Electrical Power Subsystem
GPIO	General Purpose Input/Output
SPI	Serial Peripheral Interface
I^2C	Inter-Integrated Circuit
CSP	CubeSat Space Protocol
OS	Operating System
UART	Universal Asynchronous Receiver/Transmitter
GPIO	Graphical User Interface

CHAPTER ONE
INTRODUCTION

What is a CubeSat:

CubeSats began as a collaborative effort in 1999 between Jordi Puig-Suari, a professor at California Polytechnic State University (Cal Poly), and Bob Twiggs, a professor at Stanford University's Space Systems Development Laboratory (SSDL). The original intent of the project was to provide affordable access to space for the university science community, and it has successfully done so. Using CubeSats, many major universities now have a space program. But it's not just big universities; smaller universities, high schools, middle schools, and elementary schools have also been able to start CubeSat programs of their own.

Let's explain what the "CubeSat" designation means, compared to other small satellites. A small satellite is generally considered to be any satellite that weighs less than 300 kg (1,100 lb). A CubeSat, however, must conform to specific criteria that control factors such as its shape, size, and weight. The very specific standards for CubeSats help reduce costs. The standardized aspects of CubeSats make it possible for companies to mass-produce components and offer off-the-shelf parts. As a result, the engineering and development of CubeSats becomes less costly than highly customized small satellites. The standardized shape and size also reduces costs associated with
transporting them to, and deploying them into, space. CubeSats come in several sizes, which are based on the standard CubeSat "unit"—referred to as a 1U. A 1U CubeSat is a 10 cm cube with a mass of approximately 1 to 1.33 kg. In the years since the CubeSat's inception, larger sizes have become popular, such as the 1.5U, 2U, 3U, 6U, and 12U. Examples of a 1U and 3U are shown in figure 1.

Figure 1.1. 1U CubeSat (Left), 3U CubeSat (Right)

The CubeSat provides the following opportunities:

- Improved science, technology, engineering, and math (STEM) education: CubeSats provide students with hands-on learning opportunities in aerospace engineering. In addition to being a professional development opportunity, Dr. Zac Manchester described the benefit as "a different level of engagement [from students] when the thing you build is going into space" (Building Blocks for Better Science).
- New types of research: Standard satellites are expensive and take a long time to build. For this reason, they typically are used for low-risk missions. CubeSats, on the other hand, can be used for exploratory, high-risk research—such as NASA's studies of bacteria genetics in space and deep space exploration—because of their inexpensive and quick development.
- Accelerated innovation: CubeSats enable new users from across different disciplines to contribute their ideas and unique skill sets to small satellite design. With both more participants and more diverse participants, solutions to challenges in space and on Earth can be faster and more creative.
- Public engagement: CubeSats' accessibility allows members of the public more autonomy over the research questions they address. For example, citizen scientists use CubeSats to conduct experiments of interest in space, including those that broaden our understanding of Earth. The opportunity for the public to engage with and shape research agendas can strengthen the connection between science and society.

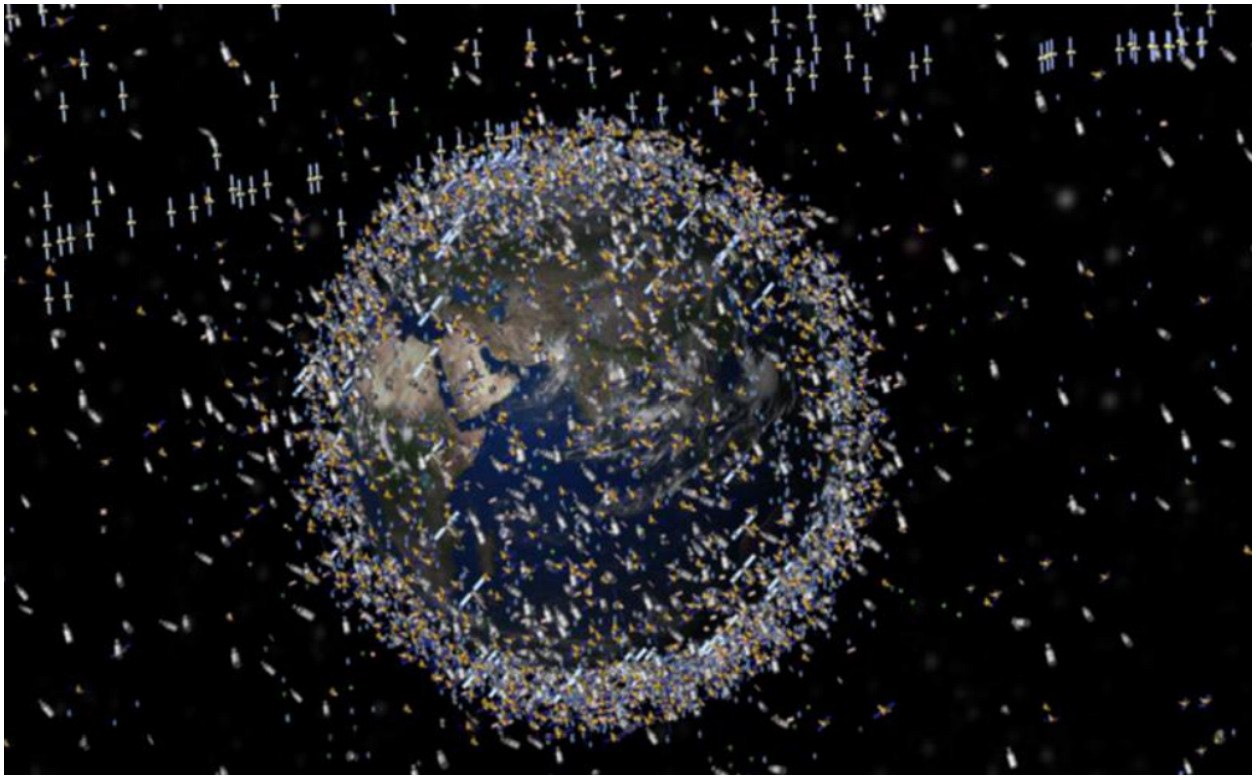


Figure 1.2. A visualization of the exponential growth of CubeSats Development

However, such opportunities come at certain costs. The development of CubeSats faces some challenges:

- Orbital debris: CubeSats have a relatively minimal debris impact because of their size. However, there are over 50,000 objects between 1 cm and 10 cm and more than 100 million objects less than 1 cm that are not being tracked in space, potentially causing collisions with other space systems and crowding the space environment. CubeSats can both be impacted by orbital debris or add to it, in part because CubeSats have a 25 year deorbit guideline, despite their short missions. Even inactive CubeSats remain in orbit and continue to pose a risk to other space systems.
- Radiofrequency spectrum management: The radiofrequency spectrum is becoming more crowded at the low frequencies at which CubeSats operate. Even after proceeding with the licensing processes that assign and manage frequencies, CubeSats often experience interference due to crowding. According to a presentation given at the Space Communication and Navigation Symposium, titled "Communication Architecture and International Policy Recommendations Enabling the Development of Global CubeSat Space Networks," authorities are aware of crowding but have not yet developed a strategy to accommodate it. Developing infrastructure and communication protocols to manage the radiofrequency spectrum becomes even more important as CubeSat constellations or swarms increase in popularity. Currently, there are no regulations to prevent space systems in constellations from individually using a unique frequency—instead of the entire constellation using one coordinated frequency—which can waste limited radio communication resources or add more interference.
- Cybersecurity: Satellites are becoming increasingly vulnerable to cybersecurity threats. With improved capabilities of small satellites and CubeSats, there are many satellites collecting information that may require encryption or other cybersecurity protocols but do not currently do so. The ambiguity and gaps in cybersecurity policy can lead to sensitive data and information not being correctly protected from cyberthreats.
- Rendezvous and proximity operations: As space becomes more crowded and CubeSat capabilities increase, there are more chances of proximity operations, when satellites or other space systems move intentionally into the vicinity of one another. There is currently no clear standard and regulatory process for commercial and non-federal satellites.

Time-Triggered Architecture

The operating system is a computer program that supports a computer's basic functions, and provides services to other programs (or applications) that run on the computer. The applications provide the functionality that the user of the computer wants or needs. The services provided by the operating system make writing the applications faster, simpler, and more maintainable. If you are reading this web page, then you are using a web browser (the application program that provides the functionality you are interested in), which will itself be running in an environment provided by an operating system. The case of an embedded system is not different, and an OS is developed to schedule and manage the operation of the whole system. Most operating systems appear to allow multiple programs to execute at the same time. This is called

multi-tasking. In reality, each processor core can only be running a single thread of execution at any given point in time. A part of the operating system called the scheduler is responsible for deciding which program to run when, and provides the illusion of simultaneous execution by rapidly switching between each program.

The type of an operating system is defined by how the scheduler decides which program to run when. For example, the scheduler used in a multi user operating system (such as Unix) will ensure each user gets a fair amount of the processing time. As another example, the scheduler in a desktop operating system (such as Windows) will try and ensure the computer remains responsive to its user. The scheduler in a Real Time Operating System (RTOS) is designed to provide a predictable (normally described as deterministic) execution pattern. This is particularly of interest to embedded systems as embedded systems often have real time requirements. A real time requirement is one that specifies that the embedded system must respond to a certain event within a strictly defined time (the deadline). A guarantee to meet real time requirements can only be made if the behavior of the operating system's scheduler can be predicted (and is therefore deterministic).

Traditional real time schedulers, such as the scheduler used in FreeRTOS, achieve determinism by allowing the user to assign a priority to each thread of execution. The scheduler then uses the priority to know which thread of execution to run next. In FreeRTOS, a thread of execution is called a task.

But why use the time-triggered approach? Time-triggered architecture (abbreviated as TTA), also known as a time-triggered system, is a computer system that executes one or more sets of tasks according to a pre-determined and set task schedule. Implementation of a TT system will typically involve use of a single interrupt that is linked to the periodic overflow of a timer. This interrupt may drive a task scheduler (a restricted form of real-time operating system). The scheduler will—in turn—release the system tasks at predetermined points in time.

Because they have highly deterministic timing behavior, TT systems have been used for many years to develop safety-critical aerospace and related systems.

An early text that sets forth the principles of time triggered architecture, communications, and sparse time approaches is Real-Time Systems: Design Principles for Distributed Embedded Applications in 1997. Use of TT systems was popularized by the publication of Patterns for Time-Triggered Embedded Systems (PTTES) in 2001 and the related introductory book Embedded C in 2002, both by M. J. Pont. The PTTES book also introduced the concepts of time-triggered hybrid schedulers (an architecture for time-triggered systems that require task preemption) and shared-clock schedulers (an architecture for distributed time-triggered systems involving multiple, synchronized, nodes). Since publication of PTTES, extensive research work on TT systems has been carried out.

There are more TT Books written by Michael J. Pont (Founder of SafeTTy Systems). SafeTTy Systems is a company that aims at developing software for reliable space-based systems, automotive systems (including autonomous vehicles), industrial control systems, medical systems, railway systems, sports equipment, and others. Various Time-Triggered Reference Designs (TTRDs) can be downloaded from their website. In our project, TTRD2-19A was used to build our TT Operating System. Swift-Act, a company based in Egypt who is the sponsor of our project, has been a partner with SafeTTy Systems since 2014.

Other CubeSats: SwissCube and UPSat

SwissCube is the first satellite entirely built in Switzerland. It is very small in size since it occupies a volume of 1 litre (10x10x10 cm) and weighs less than 1 Kg. SwissCube follows the Cubesat standard which was developed by Standford and CalPoly (USA) and allows universities and research centres to build their own satellites. Due to its size and available power (less than a few Watts are generated by the solar panels), SwissCube can of course not compete with the capabilities of much larger satellites. However, it carries most of the sub-systems (e.g. structure, on-board computer, communication, attitude control, antennas) that exist on large satellites and allows students to build a complex engineering system.

SwissCube was mainly built by students from different universities under the supervision of the Space Center EPFL. More than 180 students participated in the adventure from EPFL, from the university of Neuchatel, from the HES-SO (Sion, Yverdon, Fribourg, St-Immer, Le Locle), and from the FHNW (Brugg-Windisch). The project was managed in a similar manner as some of the programmes in space agencies such as ESA and NASA with detailed reviews after each major phase of the project. The goal being to prepare at best the students for their future professional careers.

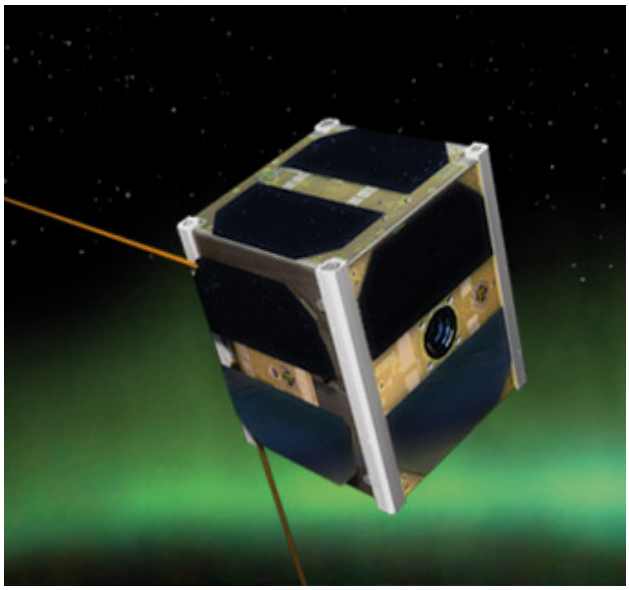


Figure 1.3. SwissCube

UPSat is a QB50 cubesat by Libre Space Foundation & University of Patras. UPSat says that it is the “first open-source satellite.” UPSat, the first open source hardware and software satellite, was released in orbit by NanoRacks deployer from the International Space Station at 08:24 UTC 2017-05-18. After 30 minutes, UPSat subsystems commenced normal operations in orbit. The SatNOGS open ground station network started receiving telemetry signals from UPSat in several ground-stations deployed globally shortly after its deployment. All subsystems are reporting nominal operations and the UPSat team is proceeding with LEOP phase in preparation for the science phase of the mission. The launch of the UPSat according to their website: “At April 18th 11:11 EDT at Cape Canaveral in Florida, an Atlas-V rocket launched a Cygnus cargo spacecraft to dock to the [International] Space Station with supplies and scientific experiments. Among its cargo UPSat, the first open source hardware and software satellite bound to be

released in orbit by the NanoRacks deployment system on-board ISS in the coming weeks.This launch marks an important milestone for open source space exploration, paving the way for more open source software and hardware space technologies to come.”

Our CubeSat was deeply inspired by UPSat, including the hardware and software components, the breaking-down of the satellite into four subsystems. The figure below shows the UPSat as shown on their website.

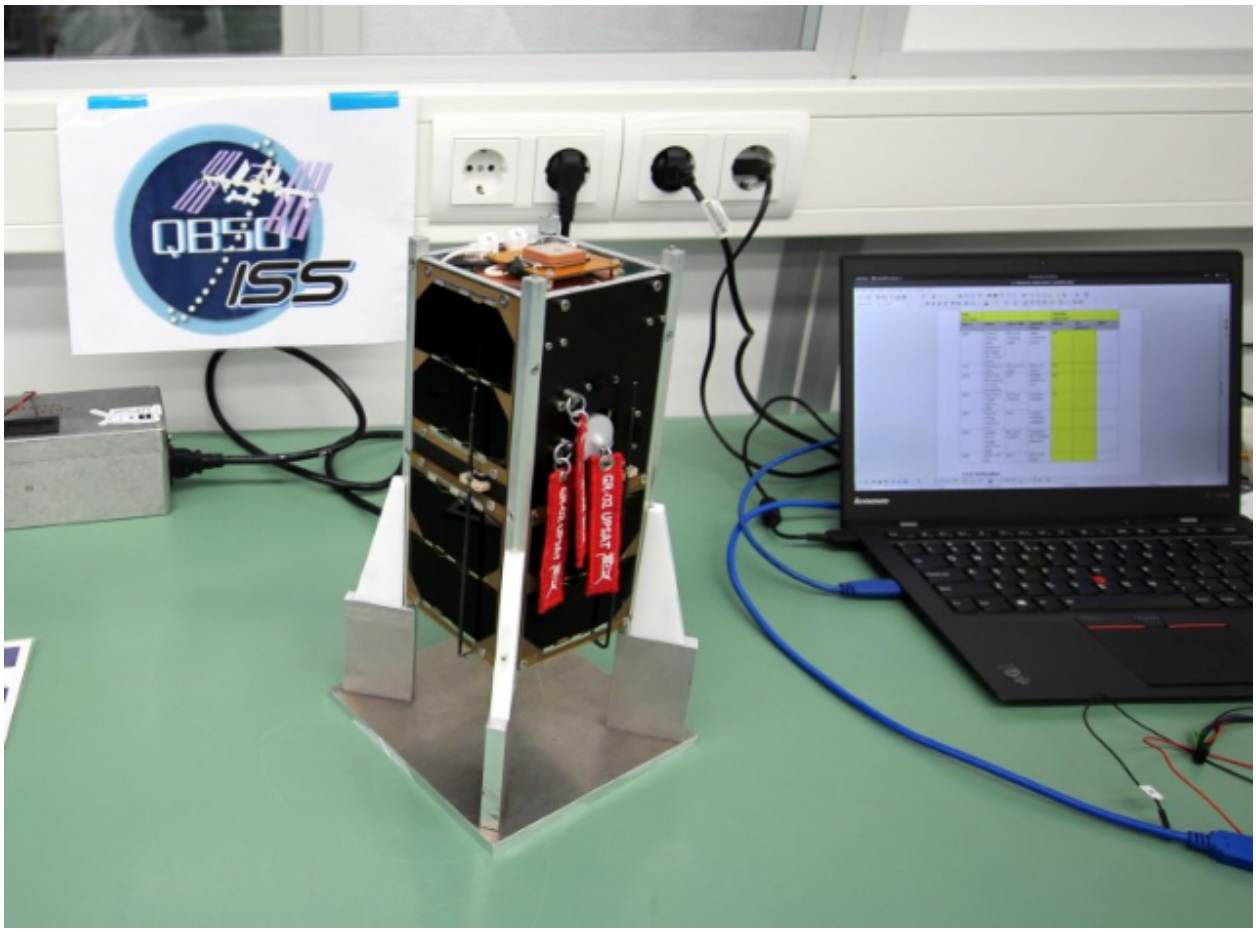


Figure 1.4. UPSat

.

Our System Structure

The CubeSat system is built-up from four different subsystems that, through their interaction with each other, deliver the required and expected functionality of a CubeSat. The four subsystems are:

- The Electrical Power Subsystem (EPS),
- The Attitude Detection and Control Subsystem (ADCS),
- The Communications Subsystem (COMMS), and
- The On-Board Computer Subsystem (OBC).

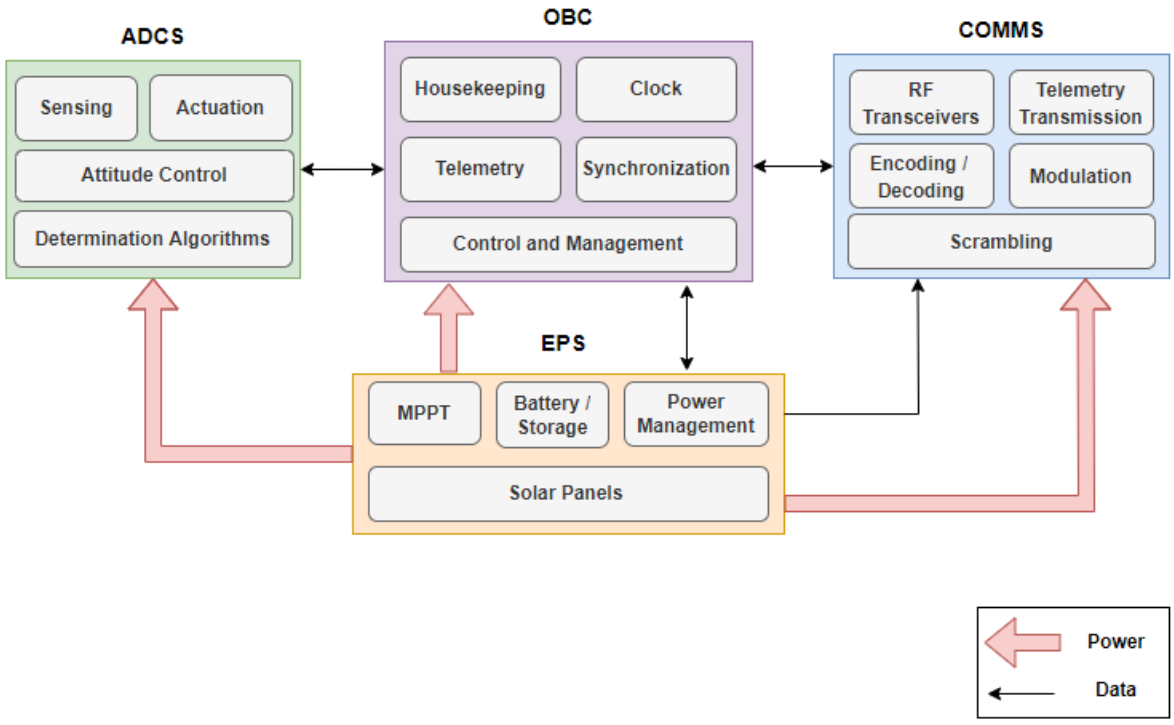


Figure 1.5. The Interaction among the CubeSat's Subsystems

The EPS is mainly responsible for the power storage, supply and management of the CubeSat. The OBC is the brain of the CubeSat, and is, thus, responsible for the control and management of the CubeSat and its subsystems. It is also responsible for the telemetry that is transmitted by the COMMS to the Ground Station (GS) to report the status of the satellite, and for maintaining clock synchronization. The ADCS mainly manages the position of the CubeSat in space through determining the position parameters, reading sensors, and controlling and actuating accordingly. The COMMS is responsible for the communication with the GS. It sends regular CubeSat status information, and receives commands from the GS regarding actions that are required to be taken on the CubeSat.

In the upcoming chapters, each subsystem is explained in detail showing the design and implementation of the various software and hardware components making up the subsystem. We start in chapter two by covering the EPS.

CHAPTER TWO

EPS SUBSYSTEM

Introduction

The primary EPS function is capturing solar energy from the sun and albedo with solar cells during the daylight and then to transmit it to the subsystems. So, a part of the captured solar energy must be kept in a battery in order to return it to the subsystems during the eclipse (during this time the solar cells are of course quasi inefficient. When sunlight reaches the Earth's surface, some of it is absorbed and some is reflected. In the study of the CubeSat EPS we see that we must know the types of solar panels and how we design these to make higher current or higher voltage and the types of batteries we need. Also, we need to know subsystems that we will feed with electricity and calculate our power budget to ensure that our EPS can cover all requirements of our subsystems: On-Board Computer (OBC), Communications Subsystem (COMMS) and Attitude Determination and Control System (ADCS).

Functional Description

The Electrical Power Subsystem (EPS) is responsible for charging the batteries from the solar panels using the MPPT technique, subsystems power management and batteries temperature control. It is also responsible for the post launch sequence that keeps the subsystems turned off for 30 minutes after the launch from the ISS and after the 30 minutes have passed, it deploys the antennas and the SU m-NLP probes by using a resistor to burn a thread that keeps the mechanism closed.

Hardware main Components:

- STM32L152(Ultra Low Power Consumption) microcontroller with an ARM cortex M3 cpu core that runs the MPTT algorithm for charging the batteries.

- 3x 2600mAh 18650 Li-ion batteries.

- MOSFET switches for controlling the subsystems power
- 4x 1.5W (6Vx250mA)

Software main functions:

EPS Subsystem software is concerned with main functions or tasks like:

1. Communication with OBC.
2. Power management and distribution including:

   MPPT battery charging system, load control (ON/OFF).

3. Measuring and Sensing:
   for voltages, currents, and temperatures for the subsystems, batteries, and solar panels.
4. Performing Safety Checks:
   Checking if the voltages, currents, and temperatures are within the safety limits.
5. Status and Error Reporting:
   Sends error status via the UART debug port.

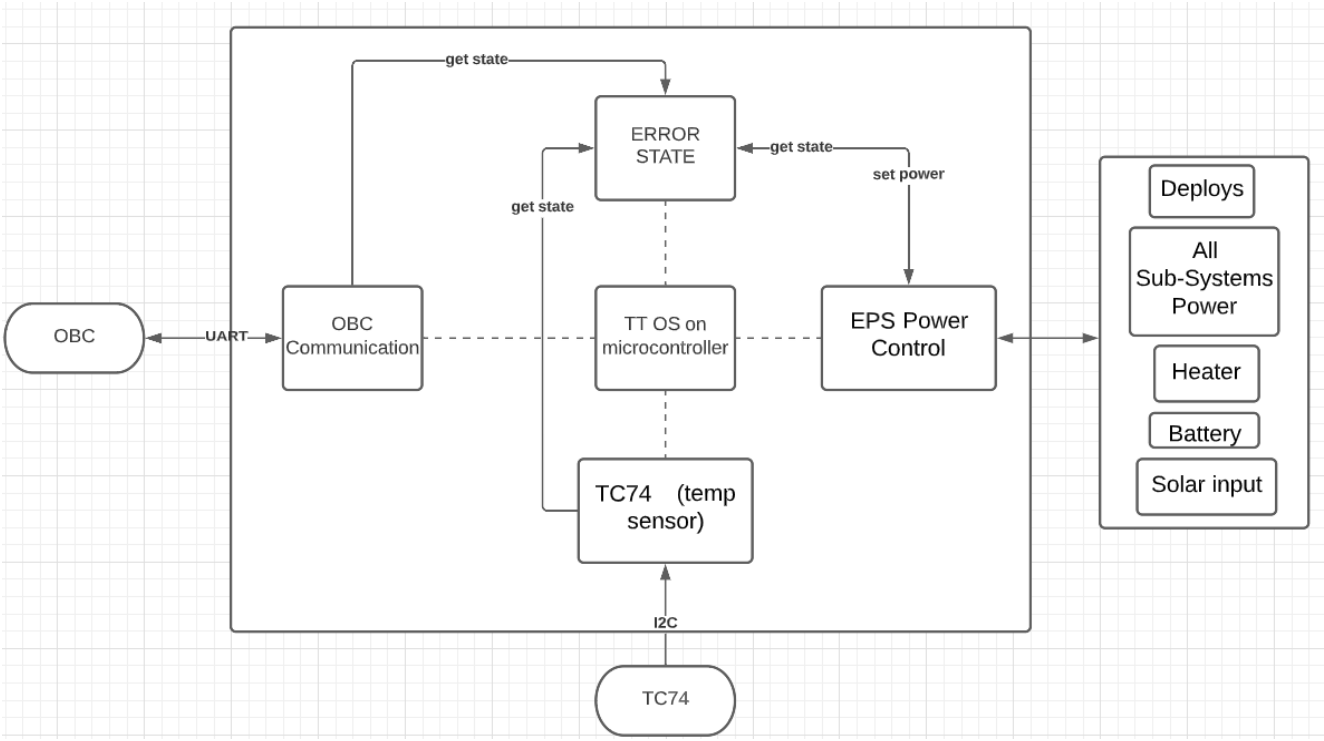


Figure 2.1. Functional description diagram

Static Software Design

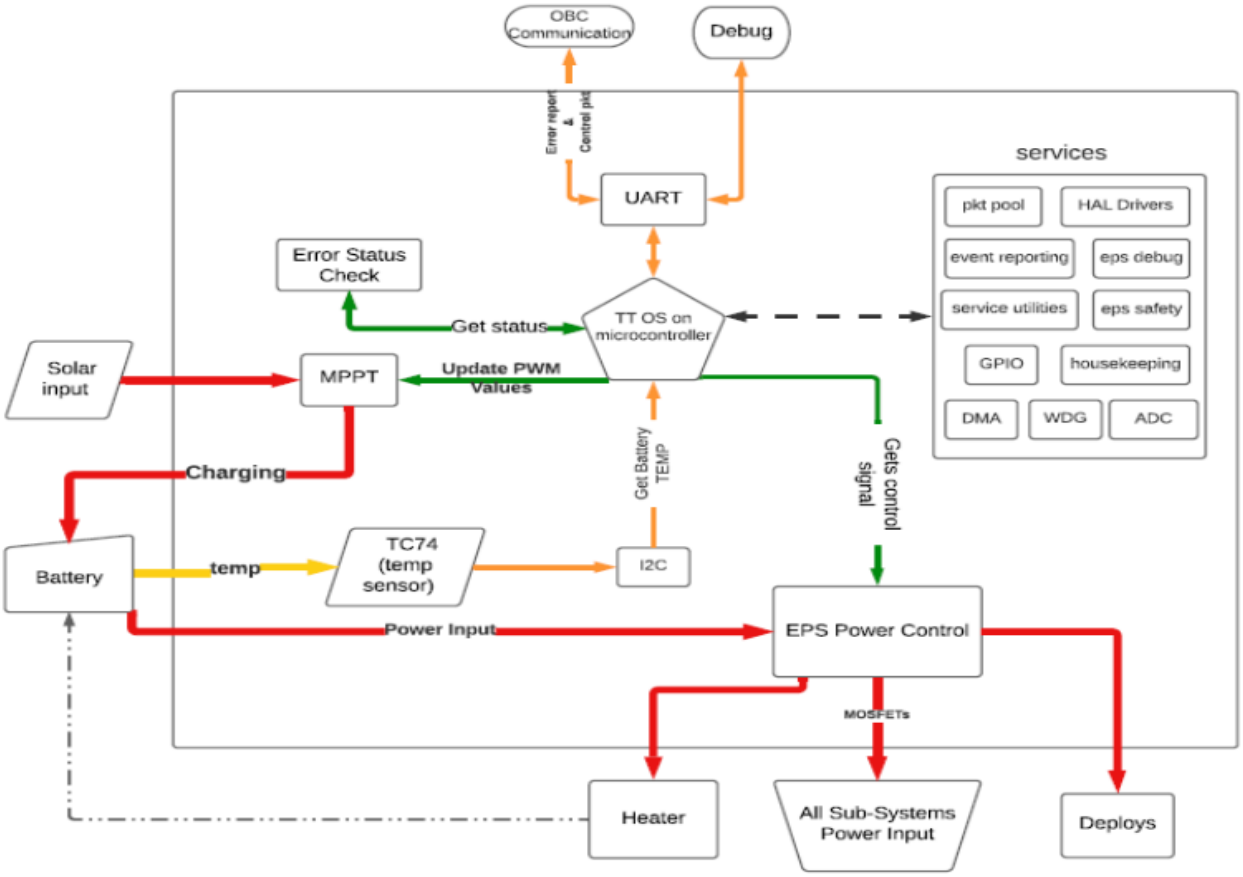


Figure 2.2. System Block Diagram

Time Triggered OS on microcontroller:

Here we mainly focus on the ttrd19a and its scheduler in which we add tasks in order with its best-case execution time (BCET) and worst-case execution time (WCET).

More details will be discussed in the Dynamic software design.

The EPS has a major tasks and functions that we have focused on while developing the software on the TTRD19A.

Tasks:

OBC-EPS Communication:

which is done by UART in both directions.
This task includes two modes:

1. EPS to OBC Communication: EPS reports the voltages, currents of all subsystems and the battery status as well as the solar panels voltage, PWM duty cycle of MPPT phase.
2. OBC to EPS Communications: OBC gives packet to EPS which contains what subsystems to be ON or OFF because; each subsystem must report a heartbeat message to OBC and if not the OBC orders the EPS to turn off the power of this subsystem and restart it again
   (Power ON RESET), Also if the subsystem consumes a higher current than the threshold will also lead to the same order.

Power Control & MPPT:

1. This Task is responsible for getting the status of the subsystems, batteries, and solar panels.
2. Switching ON/OFF the power to the subsystems according to OBC packet or the startup sequence.
3. Updating the PWM duty cycle of the MPPT battery charging system.

Get Reading:

Here the EPS gets all the readings voltages, currents as well as the battery temperature and the ambient temperature.

These readings are collected by ADC Pins except the ambient temperature which is taken by the TC74 sensor which is using I2C Protocol.

Error Check & Error Reporting:

This task contains subtasks:

1. Load safety limits from the memory
2. Perform Safety limits check
3. UART debug service

Dynamic Software Design

In run time, we have put our main tasks to the scheduler in order and with their BCETs and WCETs.

According to the given data the scheduler will construct a timeline of execution which is repeated periodically each 4620 mS.

TTRD19A has many features that made the execution time more deterministic and provides more safety checks and error reporting.

For example: if a certain task with BCET=100 ms and WCET = 300, if the execution time exceeds 300ms it will report that this task has over run its allowed period and if it finished execution before 100ms it will report an error that the task has under run its allowed period, and in both cases, it will perform fail safe shutdown with closing all the critical outputs and so on.

To measure the task BCET and WCET we used TTRD8a which enabled us to estimate the timing data and the scheduler array.

Now we will show the timeline and scheduler table.

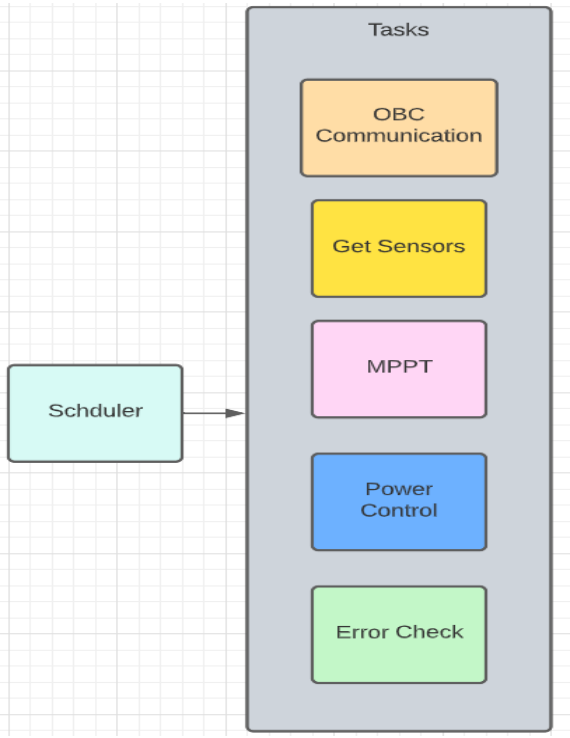


Figure 2.2. EPS Scheduler

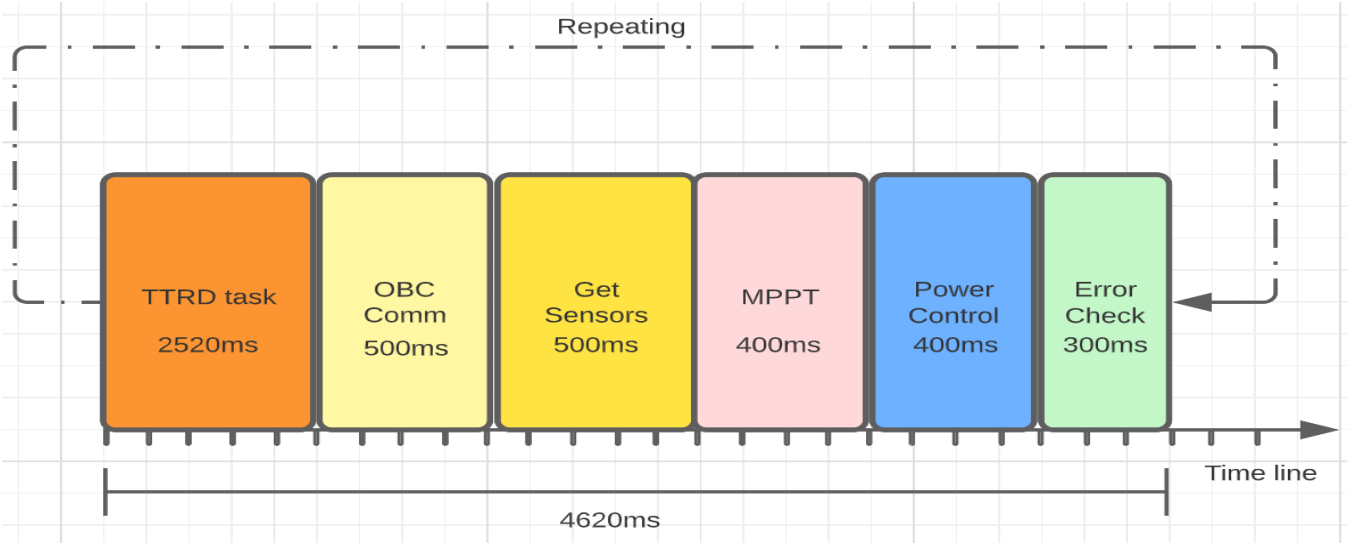


Figure 2.3 EPS Main tasks timeline

Scheduler Table:

Task Group	Task	Offset (tick)	Period (mS)	Period (tick)
TTRD-19A related tasks	Watchdog Update	0	10	1
	HEARTBEAT_SW_Update	0	1000	100
	ADC1_Update	0	500	50
	PROCESSOR_TASK_Update	0	1000	100
	UART2_BUF_O_Update	0	10	1
Service Task	OBC Comm Update	1	500	50
Get Sensor	Currents Update	2	150	15
	Voltages Update	3	150	15
	TMP Sensor Update	4	200	20
MPPT & Power Control	PWM Update	6	200	20
	Apply PWM	7	200	20
	Update Power Module status	8	200	20
	Power Switches Update	9	200	20
Error Check	Error Check Update	10	200	20
Error Report	Error Report Update	11	100	10
Total			4620	462

Hardware Implementation

Introduction & Hardware Choices

We have been through many hardware choices when it comes to solar panels, batteries, and microcontroller.

In the coming statements we will discuss a brief about each component in our subsystem to explain our vision in the choices we have made.

1.Solar Panels

As we know the first spacecraft to use solar panels was the Vanguard 1 satellite, launched by the US in 1958. This was largely because of the influence of Dr. Hans Ziegler, who can be regarded as the father of spacecraft solar power. Gallium arsenide-based solar cells are typically favored over crystalline silicon in industry because they have a higher efficiency and degrade more slowly than silicon in the radiation present in space. The most efficient solar cells currently in production are multi-junction photovoltaic cells. These use a combination of several layers of gallium arsenide, indium gallium phosphide, and germanium to capture more energy from the solar spectrum.

1.1Types of Solar Panels

- Thin film.
- Polycrystalline.
- Monocrystalline.

The variety of solar panel technologies available run on a scale of efficiency, price, durability, and flexibility, depending on what you need. Photo voltaic (PV) solar technology generates power because substances like silicon generate an electrical current when they absorb sunlight, in a process known as the photovoltaic effect. Like semiconductors, solar PV technology needs purified silicon to get the best efficiency, and the price behind PV solar manufacturing is often driven by the crystalline silicon purification process.

Thin Film

The technology with the lowest market share is thin film, but while it has several disadvantages, it is a good option for projects with lesser power requirements but needs for light weight and portability. Thin-film technologies have produced a maximum efficiency of 20.3%, with the most common material amorphous silicon at 12.5%. Thin-film panels can be constructed from a variety of materials, with the main options being amorphous silicon (a-Si), the most prevalent type, cadmium telluride (CdTe) and copper indium gallium selenide (CIS/CIGS). As a technology that's still emerging, thin-film cells have the potential to be less expensive. Thin film could be a driver in the consumer market, where price considerations could make it more competitive.

Figure 2.4. thin film.

Polycrystalline

The first solar panels based on polycrystalline silicon, which also is known as polysilicon (p-Si) and multi-crystalline silicon (mc-Si), were introduced to the market in 1981. Unlike monocrystalline-based solar panels, polycrystalline solar panels do not require the Czochralski process. Raw silicon is melted and poured into a square mold, which is cooled and cut into perfectly square wafers. The process used to make polycrystalline silicon is simpler and cost less. The amount of waste silicon is less compared to monocrystalline. The efficiency of polycrystalline-based solar panels is typically 13-16 % because of lower silicon purity.

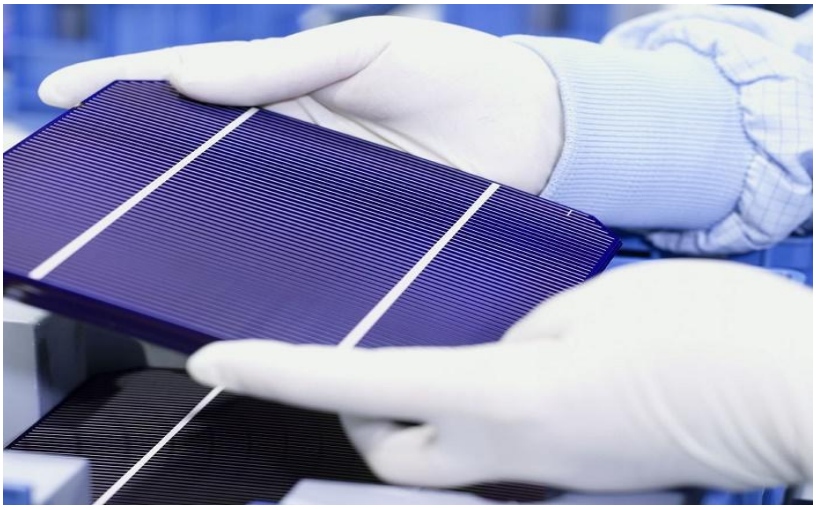


Figure 2.5. Polycrystalline.

Monocrystalline

Solar cells made of monocrystalline silicon (mono-Si), also called single-crystalline silicon (single-crystal-Si) and are quite easily recognizable by an external even coloring and uniform look, indicating high-purity silicon. The efficiency rates of monocrystalline solar panels are typically 18-24%. Monocrystalline solar panels have the highest efficiency rates since they are made from the highest-grade silicon, but Monocrystalline solar panels are the most expensive.

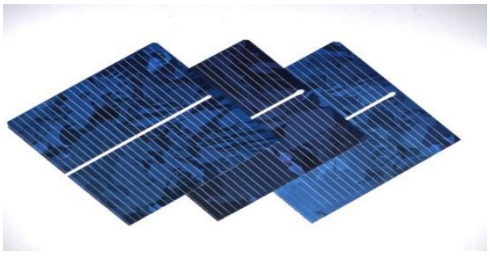


Figure 2.6. Monocrystalline.

1.2. Solar PV Efficiency

If we want to increase the efficiency of solar panels of course we must add junctions (layers) of solar cell in fabrication these junctions consist of compounds of semiconductor compound such as indium, gallium, arsenide, and phosphide. Once we add some junctions with good materials that make solar cell has more efficiency and more durability.

2. Batteries

The battery is a source of electrical energy, which is provided by one or more electrochemical cells of the battery after conversion of stored chemical energy. In today's life, batteries play an important part as many household and industrial appliances use batteries as their power source.

2.1. Types of Batteries

Batteries can be divided into two major categories:

primary batteries, and secondary batteries.

A primary battery is a disposable kind of battery. Once used, it cannot be recharged.

Secondary batteries are rechargeable batteries. Once empty, it can be recharged again.

This charging and discharging can happen many times depending on the battery type.

Alkaline batteries, Mercury batteries, Silver-Oxide batteries, and Zinc carbon batteries are examples of primary batteries whereas Lead-Acid batteries and Lithium ion and lithium polymer batteries fall into the secondary battery's category and that we will take about the rechargeable battery such as:

- Nickel Cadmium (NiCd).
- Nickel-Metal Hydride.
- Lead Acid.
- Lithium Ion (Li-ion).
- Lithium-Ion Polymer (Li-Po).

The variety of batteries depend on the application and the user need of this battery.

Also, by adding solar panels we will reduce the choices of which type we need at the small size and higher number of charge and discharge we decide to choose the lithium category because According to a U.S. Solar Energy Monitor report, lithium-ion batteries are the most common storage technology.

Also, Lead Acid category is good, but the problem is the huge size that take because our CubeSat is 10 by 10 by 30 cm and this is a small size can get size of battery like lithium batteries.

2.2 Important Parameters in Batteries

Table 2.1. Comparison between lead acid and lithium ion.

	Lead acid	**Lithium ion**
Weight	Heavy	One-third that of lead acid.
Efficiency	Low	High (almost 100%)
Discharge	Less than 80%	100%
Cycle life	400 – 500 cycle	5000 cycles
Voltage	Short-lasting	Longer-lasting
Cost	Low	High
Environmental Impact	Not clean for environment.	Clean and safe.

Weight: Lithium-ion batteries are one-third the weight of lead acid batteries.

Efficiency: Lithium-ion batteries are nearly 100% efficient in both charge and discharge, allowing for the same amp hours both in and out. Lead acid batteries' inefficiency leads to a loss of 15 amps while charging and rapid discharging drops voltage quickly and reduces the batteries' capacity.

Discharge: Lithium-ion batteries are discharged 100% versus less than 80% for lead acid. Most lead acid batteries do not recommend more than 50% depth of discharge.

Cycle Life: Rechargeable lithium-ion batteries cycle 5000 times or more compared to just 400-500 cycles in lead acid. Cycle life is greatly affected by higher levels of discharge in lead acid, versus only slightly affected in lithium-ion batteries

Voltage: Lithium-ion batteries maintain their voltage throughout the entire discharge cycle. This allows for greater and longer-lasting efficiency of electrical components. Lead acid voltage drops consistently throughout the discharge cycle.

Cost: Despite the higher upfront cost of lithium-ion batteries, the true cost of ownership is far less than lead acid when considering life span and performance.

Environmental Impact: Lithium-ion batteries are a much cleaner technology and are safer for the environment.

By Looking of lithium ion and lithium polymer we see that Li-polymer is unique in that a micro porous electrolyte replaces the traditional porous separator. Li-polymer offers slightly higher specific energy and can be made thinner than conventional Li-ion, but the manufacturing cost is higher by 10–30 percent.

Table 2.2. Comparison between lithium ion and lithium polymer.

	Lithium polymer	Lithium ion
Chemical reaction	Varies, depending on electrolyte.	Varies, depending on electrolyte.
Operating temperature	Improved performance at low and high temperatures.	4º F to 140º F (-20º C to 60º C)
Recommended for	Cellular telephones, mobile computing devices.	Cellular telephones, mobile computing devices.
Initial voltage	3.6 & 7.2	3.6 & 7.2
Discharge rate	Flat	Flat
Charging temperature	32º F to 140º F (0º C to 60º C)	32º F to 140º F (0º C to 60º C)
Storage lite	Loses less than 0.1% per month.	Loses less than 0.1% per month.
Storage temperature	-4º F to 140º F (-20º C to 60º C)	-4º F to 140º F (-20º C to 60º C)
Notes	- Typically designed to be recharged in the device rather than in an external charger. - Lighter than nickel-based secondary batteries with (Ni-Cd and NiMH). - Can be made in a variety of shapes.	- Typically designed to be recharged in the device rather than in an external charger. - The chemical construction of this battery limits it to a rectangular shape.

After reading all specifications of both battery types, you can see that there isn't much of a competition here. Although the lithium-polymer battery is sleeker and thinner, lithium-ion batteries have a higher energy density and cost less to manufacture. Therefore, we obviously know which one is chosen by companies like Samsung, Apple, Motorola, and more. Finally, with new chemicals batteries may make them the best choice for the CubeSat on the long run.

3.Power Distribution and Management

Electrical power subsystem will distribute the power for all other subsystems: OBC, ADCS and COMM. The electrical subsystem is presented in Figure x-x.

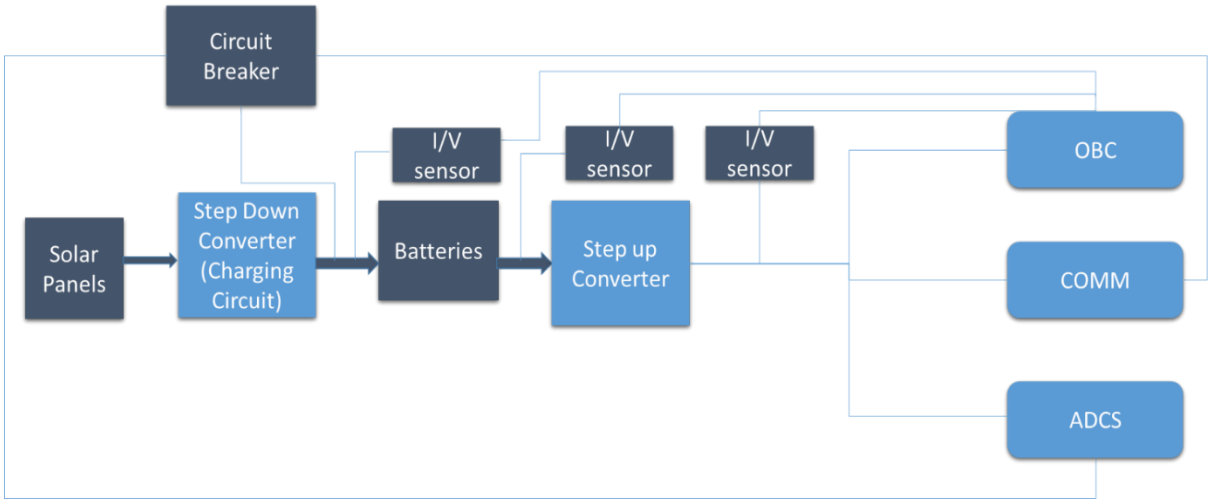


Figure 2.7. Block diagram for power distribution

4. Microcontroller

We have made the choice of the controller to be STM32L152 because it's ultra-low power consumption and it has enough ADC channels.

ST Ultra-Low Power (L Family)

ST's ultra-low-power MCU platform is based on a proprietary ultra-low-leakage technology and optimized design.

STM32 ultra-low-power microcontrollers offer designers of energy-efficient embedded systems and applications a balance between performance, power, security, and cost effectiveness. The portfolio includes the STM8L (8-bit proprietary core), the STM32L4 (Arm® Cortex®-M4), the STM32L0 (Arm® Cortex®-M0+) and the STM32L1 (Arm® Cortex®-M3). The STM32L5 MCU (Arm® Cortex®-M33) with its enhanced security features is the latest addition to this rich portfolio.

Achieving the industry's lowest current variation (25 to 125 °C), STM8L/STM32L solutions guarantee outstanding low-current consumption at high temperatures. STM32L1 MCUs also feature the industry's lowest power consumption of 170 nA in low-power mode with SRAM retention. Wake-up times are as low as 3.5 µs from stop mode.

The ultra-low-power STM32L151x6/8/B and STM32L152x6/8/B devices incorporate the connectivity power of the universal serial bus (USB) with the high-performance ARM Cortex®-M3 32-bit RISC core operating at 32 MHz frequency (33.3 DMIPS),

a memory protection unit (MPU),

high-speed embedded memories (Flash memory up to 128 Kbytes and RAM up to 16 Kbytes)

and an extensive range of enhanced I/Os and peripherals connected to two APB buses.

All the devices offer a 12-bit ADC, 2 DACs and 2 ultra-low-power comparators,

six general-purpose 16-bit timers and two basic timers, which can be used as time bases.

STM32L1 MCU Series
32-bit Arm® Cortex®-M3 – 32 MHz

- Reset POR/PDR
- 2x watchdogs
- Hardware CRC
- Internal RC
- Crystal oscillators
- PLL
- RTC calendar
- 16- and 32- bit timers
- 1x12-bit ADC
- Temperature sensor
- Multiple-channel DMA
- Single-wire debug
- Unique ID
- USB 2.0 (with internal 48 MHz PLL)

Product line	Flash (KB)	RAM (KB)	EEPROM (KB)	Memory I/F	Op-Amp	Comp.	Temp. Sensor	Capacitive Touch	Segment LCD Driver	AES 128-bit
STM32L100 Value line	32 to 256	4 to 16	2						Up to 8 x 28	
STM32L151	32 to 512	16 to 80	4 to 16	SDIO FSMC	•	•	•	•		
STM32L152									Up to 8 x 40	
STM32L162	256 to 512	32 to 80	8 to 16	SDIO FSMC	•	•	•	•	Up to 8 x 40	•

5. Hardware Design

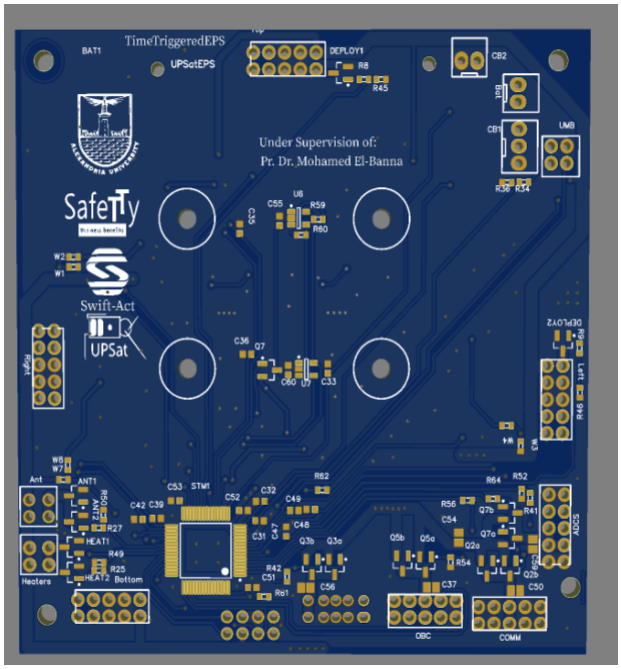
PCB Design:

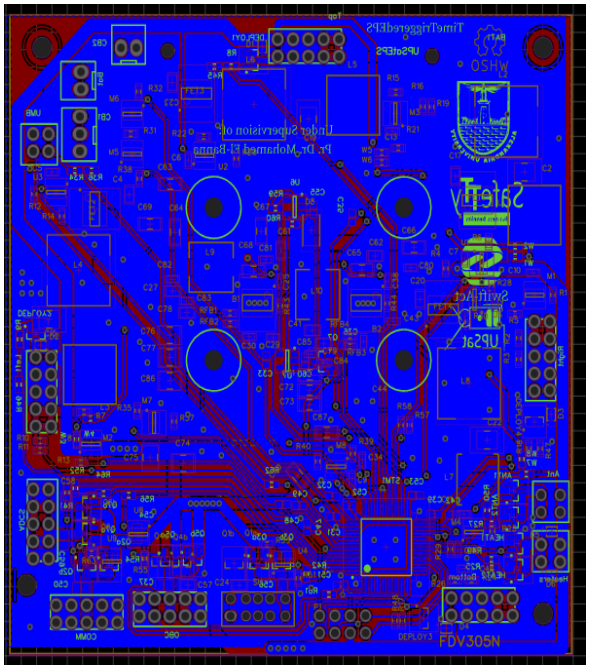
We have used Altium Designer & Easyeda Software in the PCB Design.

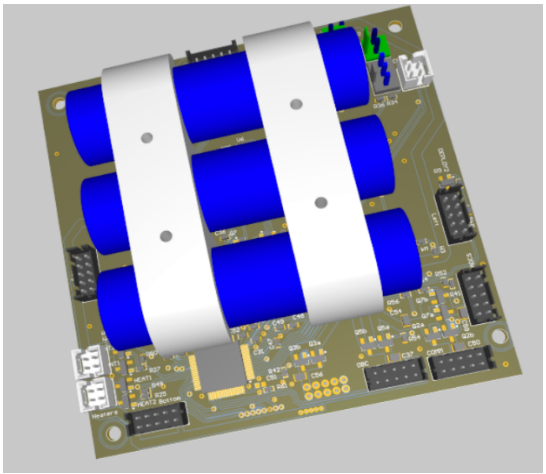
We have manufactured all the PCBs by JLCPCB.

We have ordered the components from the Chinese biggest supplier LCSC.

We assembled all the components manually.


TimeTriggeredEPS
UPSatEPS
Under Supervision of:
Pr. Dr. Mohamed El-Banna
SafeTTy
Swift-Act
UPSat





CHAPTER THREE

ADCS subsystem

ADCS Mission

As the name suggests, the ADCS (attitude detection and control system) is responsible for the detection and control of the attitude of the CubeSat. The attitude of a CubeSat (or any other body in space) is its orientation, angular velocity, and angular acceleration. So the responsibility of the ADCS subsystem is to detect those attributes using sensors and feed them to microcontroller to get the desired attitude and output those values on the actuators to achieve the desired attitude.

A famous application of CubeSats is to take images of earth, horizon, or stars. In these applications, it's obvious that pointing the CubeSat in the right direction is a must. Otherwise, they will not have control over the camera angle, and you can only pray that the camera aligns to your target at the right time. The odds are very high. Pray hard in that case.

Please note that in our CubeSat, we have no jets. Hence, we don't have control over the displacement of the CubeSat. We can by no means control the position, speed, or acceleration. The movement track is predetermined and precalculated before launch. Once launched, only God rules for the universe (aka: physics) defines the CubeSat displacement. Therefore, it's not the responsibility of the ADCS subsystem.

How it works.

In the previous section, you learnt what ADCS mission is and what it isn't. Its mission is to first detect the attitude of the CubeSat, and then control that attitude to tune it to the desired state. In this section, we talk about how the ADCS achieves his mission.

Before diving deep, we assume that the reader has decent knowledge of some topics like vectors and frames of reference. For example, we assume the reader knows what a local frame of reference is and how one frame can be translated to another one.

Back to our track. The work of ADCS is better divided into two parts: detection and control. So, for now, let's discuss the detection part. Detection is based on a key idea that is simple and straightforward. Let's assume there's two vectors. If we can measure (or model) those vectors in two different frames of reference, we can then relate one frame to the other. And this is exactly what we need. We need to relate the frame of the CubeSat to the one frame of earth. If we can do that, we will know how the CubeSat is oriented with respect to earth. In our implementation, the two vectors used are the sun vector, and the geomagnetic vector.

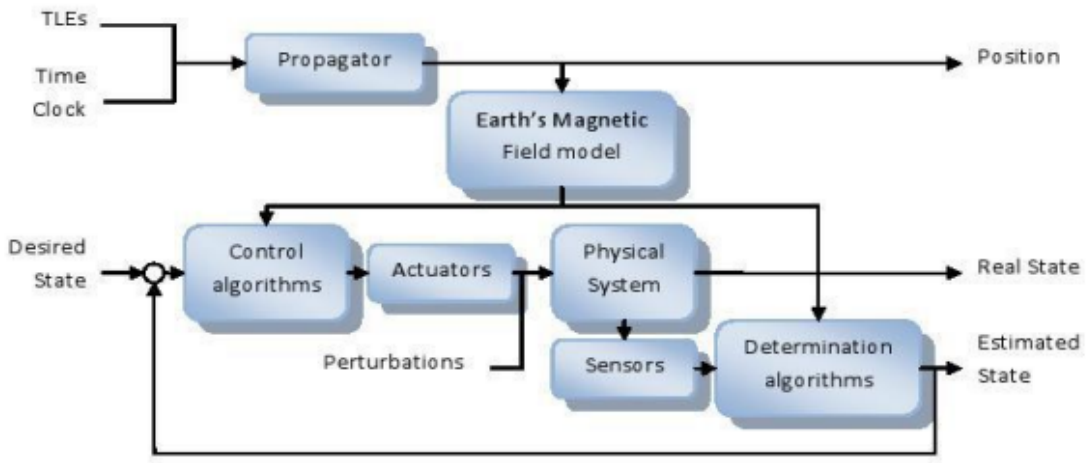


Figure 3.1

The sun vector is obviously the vector pointing to the sun. In the frame of earth, it is pointing from the center of the earth to the sun. and in the frame of the CubeSat, it's pointing from the cube center to the sun. Now we need to get that vector in the two frames. In the CubeSat frame we simply used the solar panels to roughly estimate where the sun is. In the earth frame, we used a model that once you told it the time, it tells you where the sun should be based on some mathematical equations.

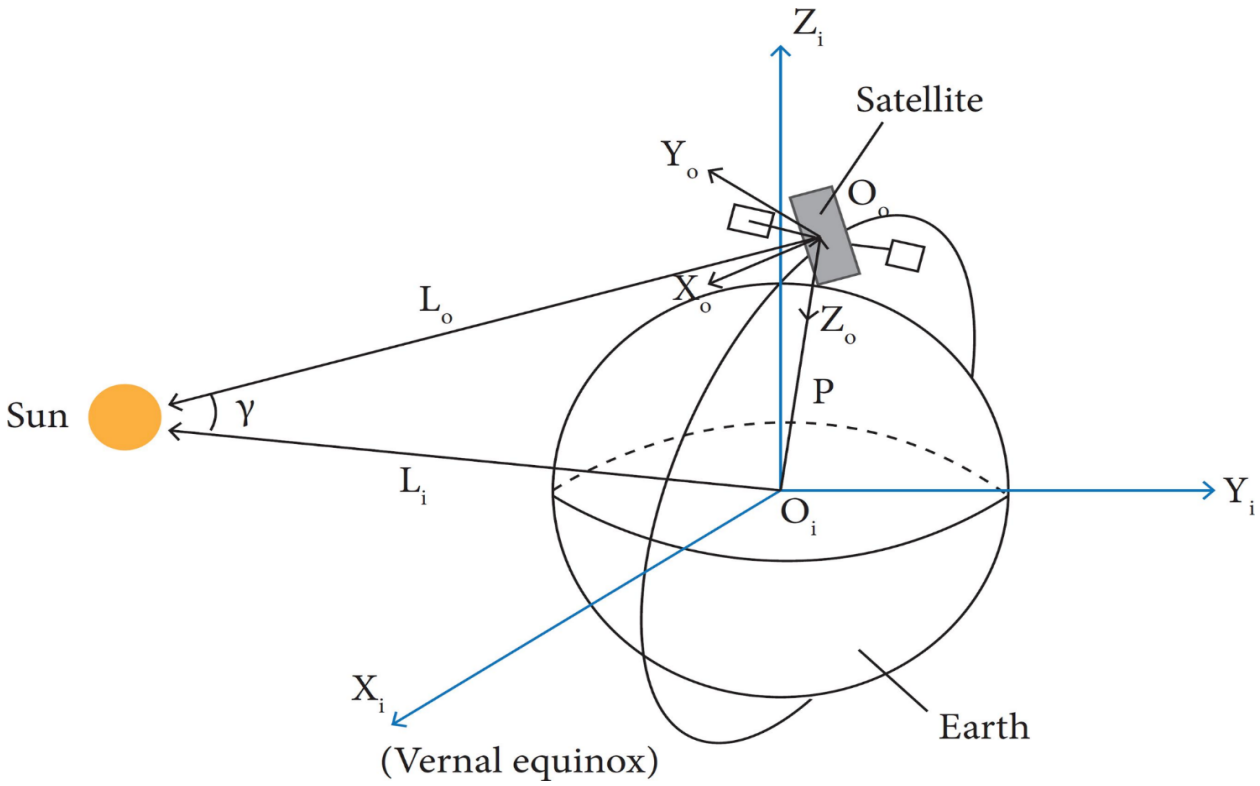


Figure 3.2

The geomagnetic vector, on the other hand, is the vector of the magnetic field of the earth. Like the sun vector, it is directly sensed by the CubeSat using magnetic compass sensors to

get the local vector. It's then calculated in the frame of earth using a model that takes position and time and gives the modeled geomagnetic vector.

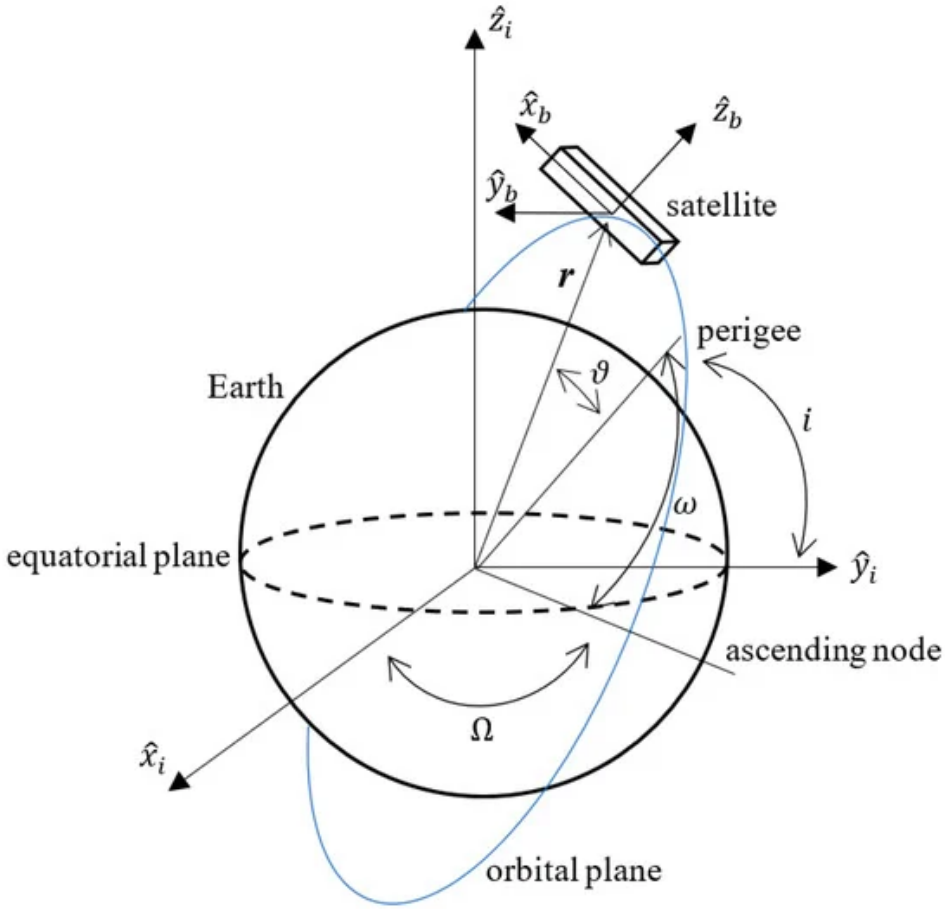


Figure 3.3

Having the two pairs of vectors; one pair in the CubeSat local frame of reference, and the other pair in the earth frame of reference, we can now try to relate those two vectors to each other. Basically, if you want to relate some set vectors to another set, you try to calculate the translation matrix. That matrix is defined so that when it's multiplied by one specific set of vectors, the output is the other set of vectors. If you want to translate in the opposite direction, use the inverse of the original matrix.

In our case, the translation matrix is called rotation matrix instead, and it's defined as:

$$V_{cubesat} = M_{rotation} \, x \, V_{earth}$$

Finding that rotation matrix is the output of the detection process.

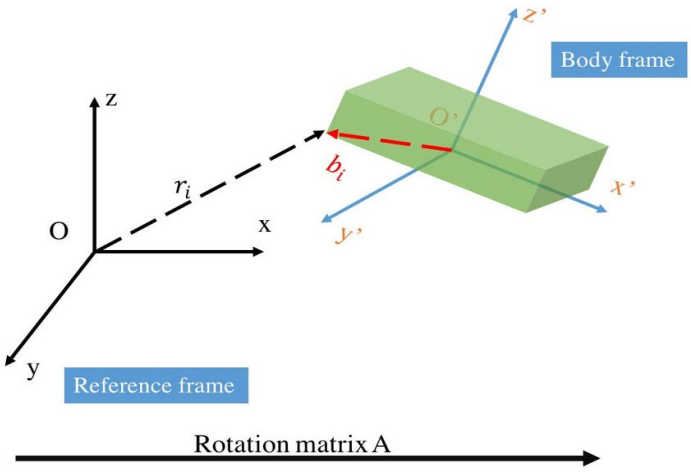


Figure 3.4

The second process is attitude control and it's much easier that detection. We have two kinds of actuators to influence the CubeSat attitude: reaction wheel and magnetic torquer. Reaction wheel is famous. It's some wheel or disk attached to a motor in such a way that allows the motors turn the wheel. Reaction wheel affects the attitude such that when it's turned in one way, the CubeSat body turns in the opposite way proportionally. Magnetic torquer simply consists of two perpendicular coils. When electric current flows in those coils, an induced magnetic field is generated. The induced magnetic field reacts to the geomagnetic field and tries to set the attitude of the CubeSat to orientation of least conflict.

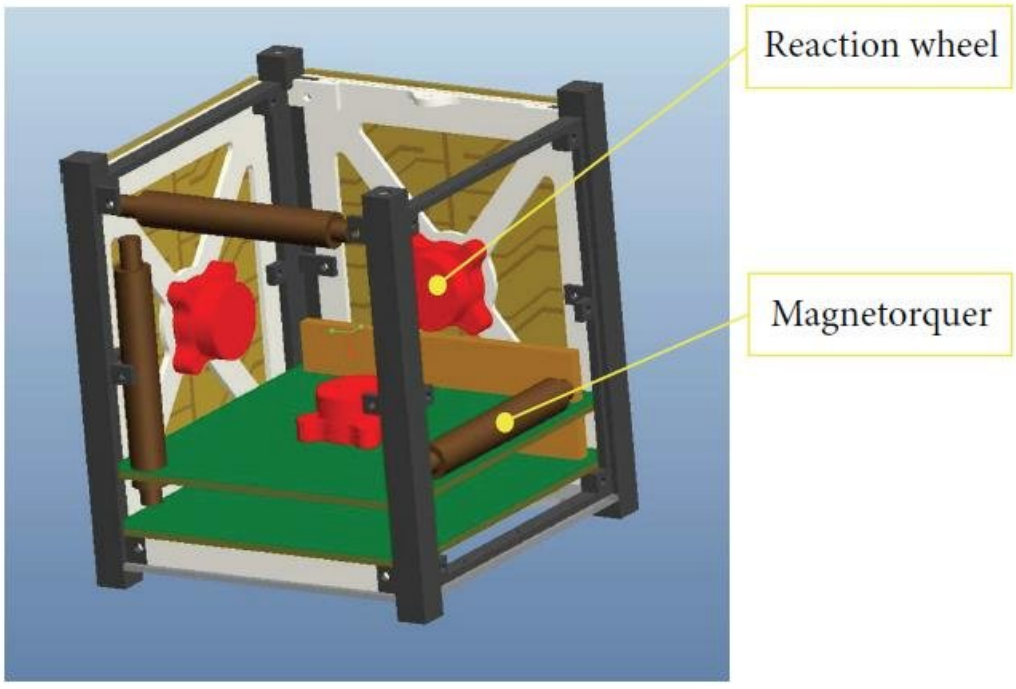


Figure 3.5

We have mentioned before that we use mathematical models to calculate the reference vectors in the frame of the earth if we know time and position. But we didn't discuss how to obtain and keep track of time and position in the first place. However, we will now.

A hardware peripheral called RTC, or Real-Time Clock is used to keep track of time. It's initially set with the help of the OBC clock and/or the GPS clock. The RTC is periodically checked against the same two previously mention sources to encounter any drift or bad initialization errors.

Position on the other hand is as easy. To accurately calculate the current position of the Cubesat, we use some piece of software called "Space Global Propagator" or SGP for short. And it's very simple in principle. If you know the position, velocity and some other parameters at some point in the past, you can propagate (pridect) those parameters at the present. This is possible because the cubesat in assumed to be in free space and therefore obeying well-defined planetary physics rules.

TLE or Two-Line Elements, is the data format fed to the propagator. It holds information about time and position as well as the track parameters. So, if we have one TLE frame, and we know what time it is, the propagator will tell us where we should be. TLEs are supposed to be created based on some ground RADARs monitoring the state of our CubeSat in space and time. The ground station is responsible for acquiring most recent TLE frames and sending them to the CubeSat through the communication link. On the CubeSat, TLE frames are received by the communication subsystem, sent to the OBC, and finally to the ADCS to be used with the propagator.

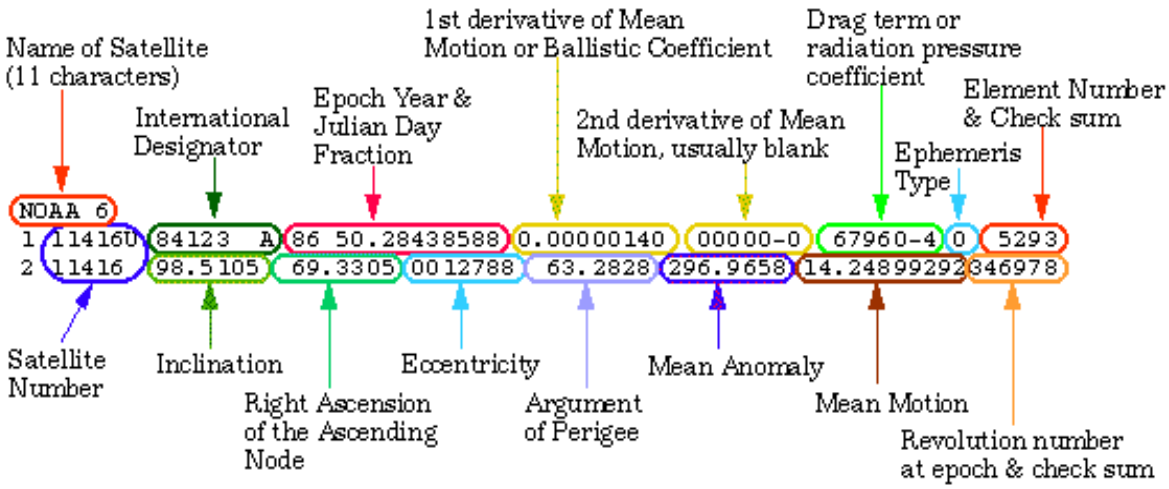


Figure 3.6

ADCS software static design

We believe that the reader is now ready with a complete description of the subsystem. So, It's now time to discuss the overall static design. A static design is meant to define the modules of the system and how they relate to each other. A static design also defines the interfaces of

each module and the other modules using those interfaces. Before discussing the static design, it may be better to have a look at the block diagram first as it's simpler and clearer.

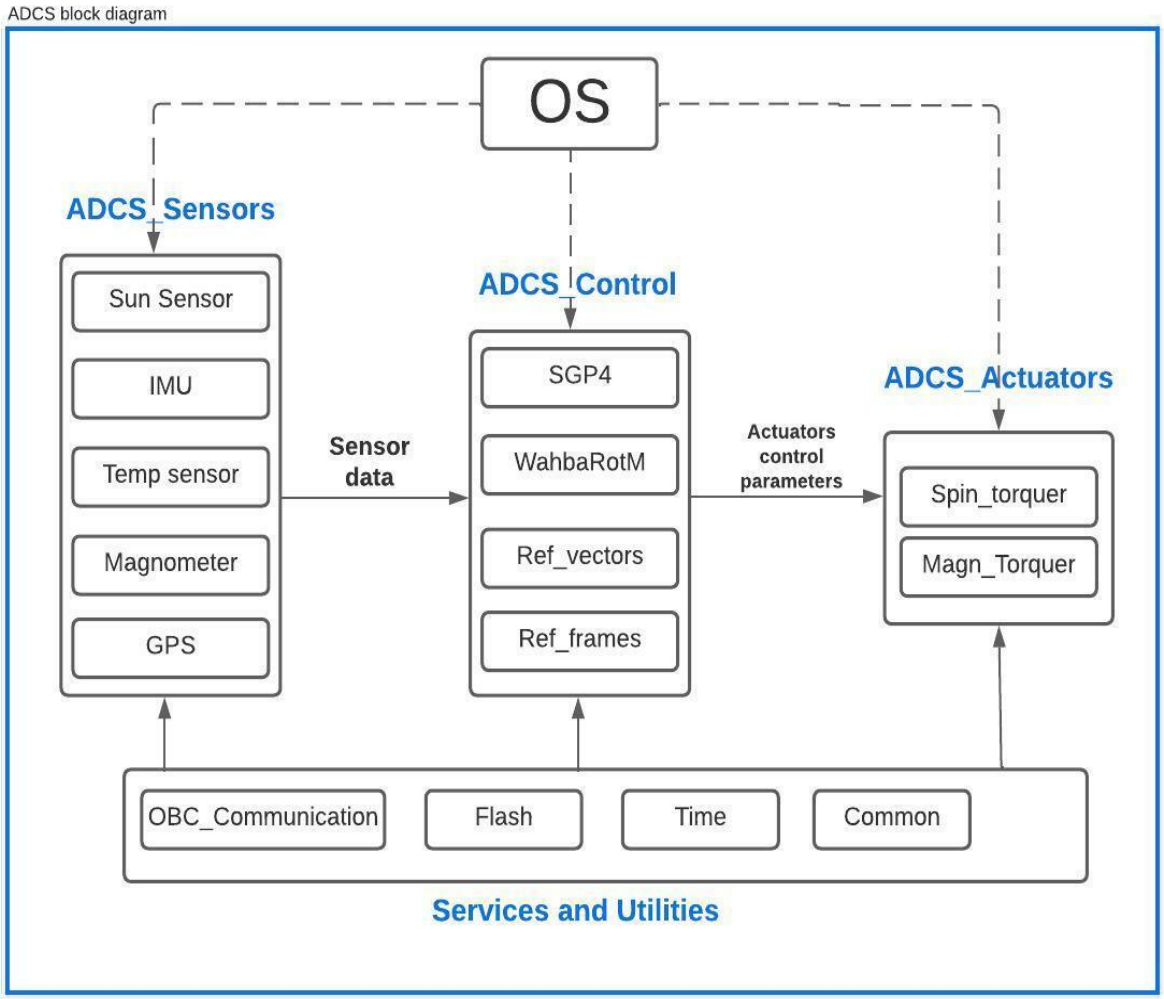


Figure 3.7.

Have a good look at the block diagram. It is pretty much self-explaining. You may notice the four main groups of the ADCS:

- Sensors
- Controller
- Actuators
- Services and utilities.

We will now discuss those four groups alongside with the OS component.

Starting with the sensors group. It contains five sensors, three of which we mentioned before; sun sensor to get the sun vector in local frame, magnetometer, or magnetic compass to get the geomagnetic vector in the local frame, and the GPS to provide time and position information when possible. Two more sensors are temperature sensor and IMU sensor. The temperature is necessary to be sensed to account for errors due to drift in temperature. The IMU sensor itself contains three sub sensors; an accelerometer to measure acceleration, a gyroscope to

sense rotation, and a magnetometer that may be used with the other magnetometer for sanity check or as backup.

The controller is responsible for calculating the reference vectors; sun and geomagnetic vectors in the earth's frame of reference, and for calculating the position using the space global propagator, as well as processing the sensor data. Once the controller finishes these three processes, it sends the right control signals to the actuators.

The actuator module does nothing but applying the parameters set by the controller to the actuators. For the services and utilities group, it is a group that serves the whole system. It provides an interface to the OBC communication port, as well as the flash memory, the RTC time peripheral and other common constants used in calculations.

The OS component is any operation technique of your choice. You may use a simple super-loop to run all modules sequentially and repeatedly. You may also use a real-time operating system. You can use anything in between of those two alternatives. For our choice of OS, we chose the TTRD-19A for reliability and the reasons discussed in previous chapters.

Let's now have a deeper more detailed look over the whole static design diagram. This is a full description to the system's modules and the interfaces they provide. The inter-module interfaces are mostly setters and getters. This is better for security and reliability.

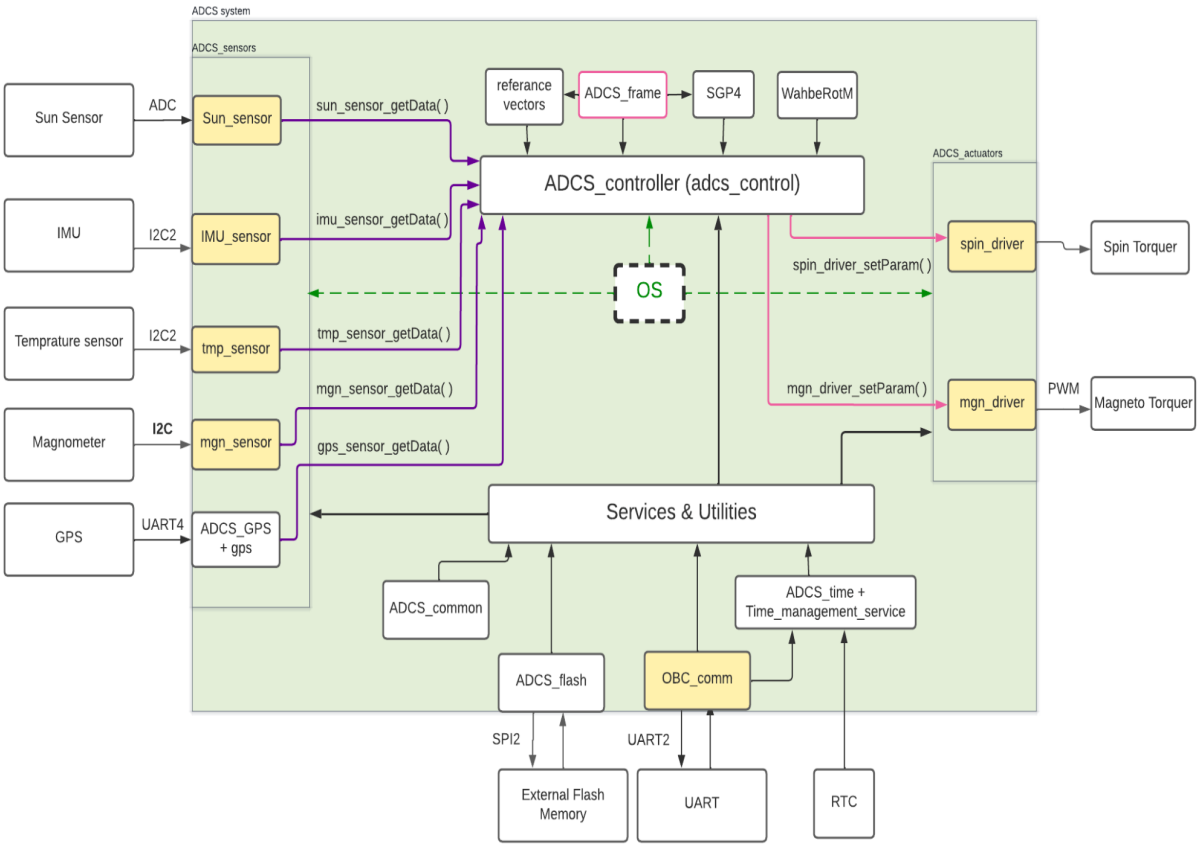
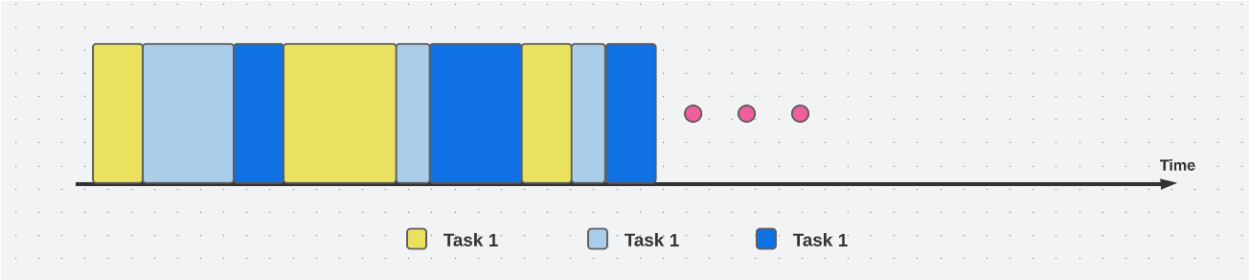


Figure 3.8.

ADCS Software Dynamic Design

A dynamic design defines how modules interact dynamically and defines the timings of each module. The final output of a dynamic design process is what I like to call “time line of the processor”, which describes what task runs at what time. The dynamic design is tightly coupled to your choice of OS. For example, if you use the super-loop architecture, there will be no time gaps between running tasks and the processor is always busy. The result will be something like this figure. Of course, This is not the choice for determinism.

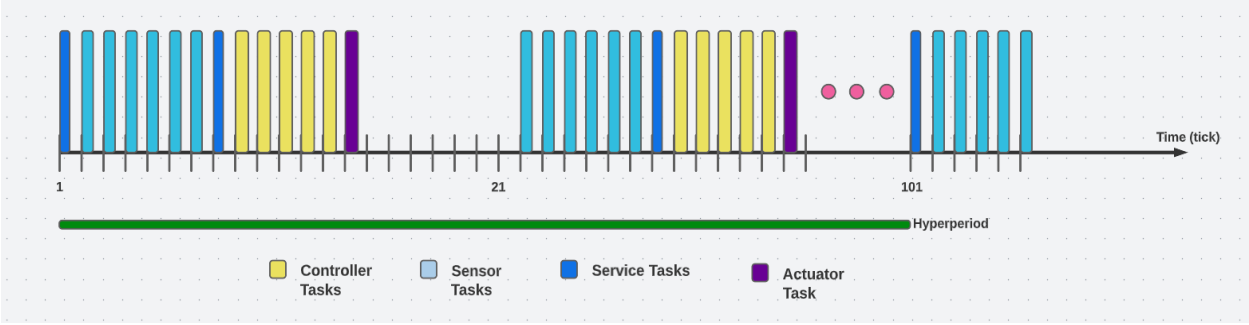


Choosing TTRD-19A as our OS, this will impact our dynamic design. The most key component in TTRD-19A taking effect on our dynamic design is the time-triggered cooperative scheduler. Time-triggered means that the schedular tick source is a timer. Cooperative means that there is no preemption and tasks are allowed to finish on their on. This is as far as the scheduler is concerned. However, there are components in the TTRD-19A OS that prevent tasks from running faster or slower than predefined rate but that’s irrelevant now.

Relating to the modules illustrated in our static design, each module has one initialization function and one update function. We consider the update function of each module to be a task. We then decided on the timing parameters of each task (i.e. task period, task offset, etc..). Based on the timing parameters of individual tasks, we deduced the timing parameters of the whole system (i.e., hyper period, tick period). The output of the dynamic design phase is the figures and tables below.

Task Group	Task	Offset (tick)	Period (mS)	Period (tick)
TTRD-19A related tasks	Watchdog Update	0	10	1
	HEARTBEAT_SW_Update	0	1000	100
	ADC1_Update	0	500	50
	PROCESSOR_TASK_Update	0	1000	100
	UART2_BUF_O_Update	0	10	1
Service Task	OBC Comm Update	1	1000	100
Sensors related tasks	IMU Sensor Update	2	200	20
	MGN Sensor Update	3	200	20
	TMP Sensor Update	4	200	20
	GPS Sensor Update	5	200	20
	Sun Sensor Update	6	200	20
	Health Check Update	7	200	20
Service Task	Time Keeping Update	8	200	20
Controller related tasks	Tle Update	9	200	20
	Sgp4 Update	10	200	20
	ref_vectors update	11	200	20
	attitude Determination	12	200	20
	attitude Control update	13	200	20
Actuators Task	actuators Update	14	200	20

TICK Interval
10 mS
Hyper period
1000 mS = 100 tick

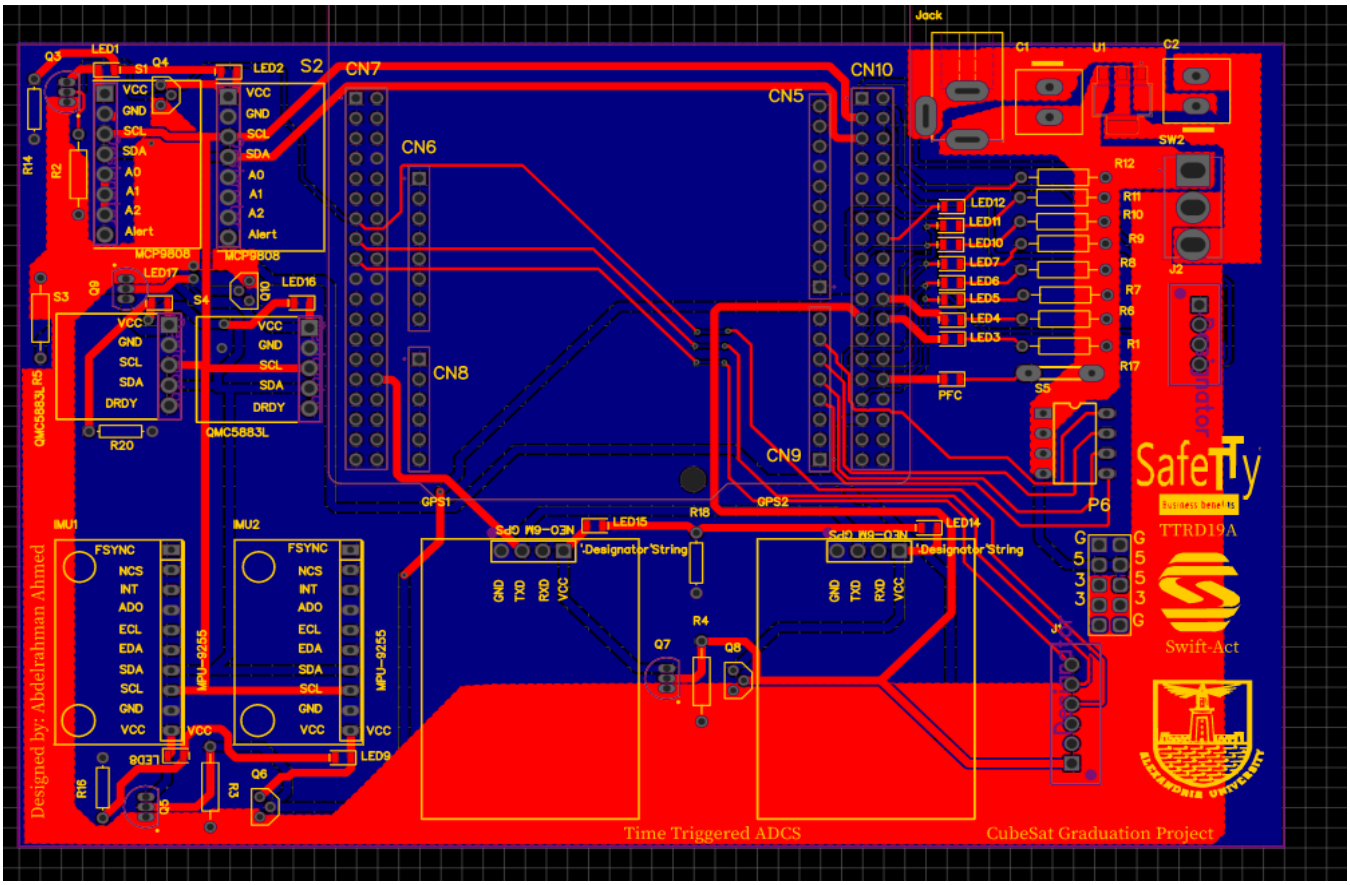


Please note that TTRD-19A related task are not present on the timeline chart as they are irrelevant.

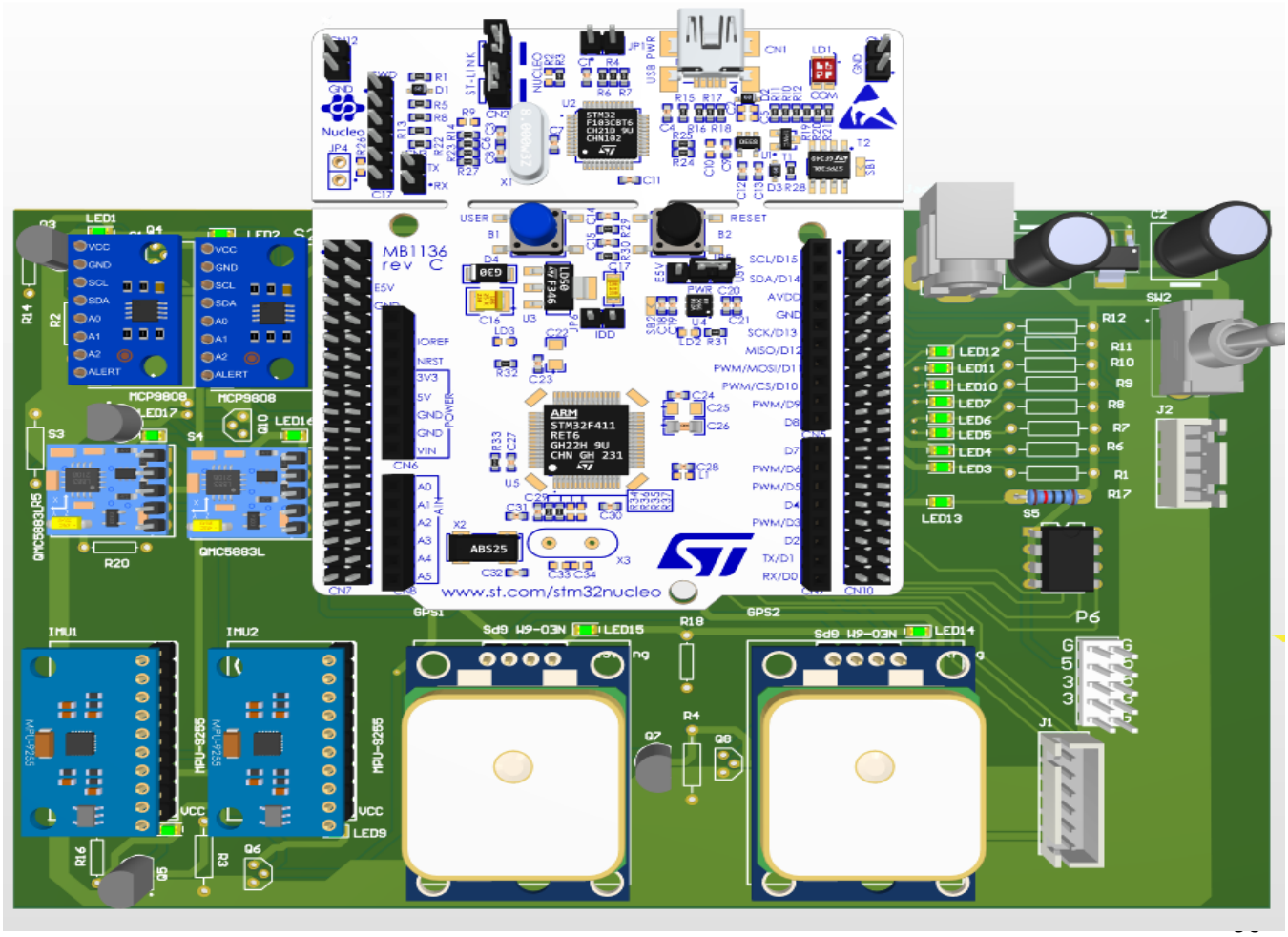
Hardware Design

We designed our hardware circuit using Altium Designer which facilitated our design as it supports 3D modeling of the circuit, which helped us to organize our components.

In our design, you can see that there are two instants of each sensor; we did that to ensure the reliability of our subsystem. If one sensor is disconnected or there is a failure in it for any reason. Our software will deactivate that sensor, activate the other one, and take the readings from it, increasing our system reliability.



Figures 3.9 (Above), 3.10 (Below)



CHAPTER FOUR

COMMS Subsystem

Mission

The communications subsystem (COMMS) is responsible for the CubeSat communication with the Earth and the ground station. The communications subsystem RF (Radio Frequency) is the most powerful and mature option to get information or send commands from/to long distance.

The objectives of the communications subsystem are:

- To get telemetry (digital packet data of housekeeping of the satellite. Temperature, voltage, current, status and altitude parameters are examples).
- To control the satellite (change operational mode, start mission camera shooting, registering the mission schedule).
- To confirm the satellite survival (e.g. using CW Morse).
- To get the distance information between the satellite to ground (ranging, especially for deep space mission),
  For the LEO case: GPS/GNSS-based positioning is popular.

The primary design process includes:

1) Identifying the requirements (Orbit, data amount, update period).
2) Selecting the frequency (Amateur, Experimental or Commercial).
3) Selecting and designing the Hardware (Antenna, Transmitter, Receiver)
    a) Identifying power, gain, sensitivity, G/T.
4) Selecting the data protocol.
5) Identifying the Link Budget (Margin).

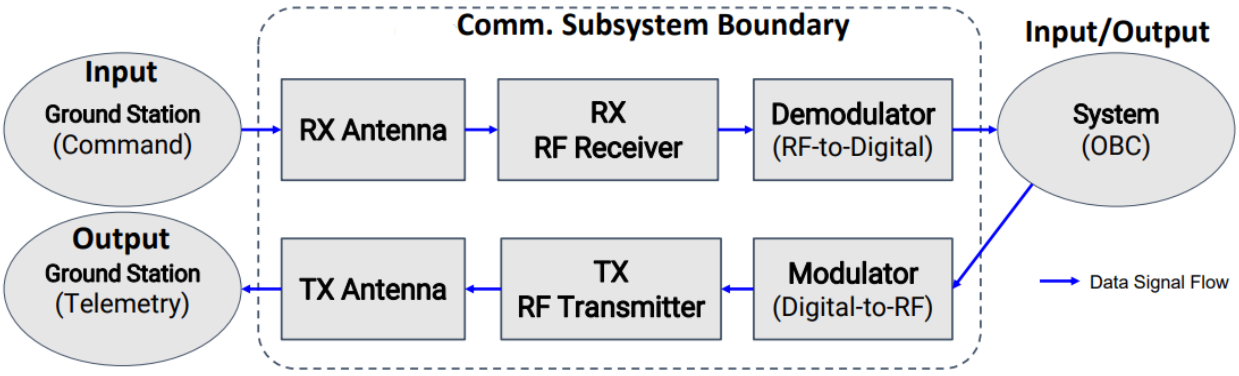


Figure 4.1. Communications Subsystem Architecture

This chapter proceeds as follows:

- The system design is shown in detail using a block-diagram representation.
- The hardware components making up the system are shown and described.

- The realization of the system is explained.

System Design

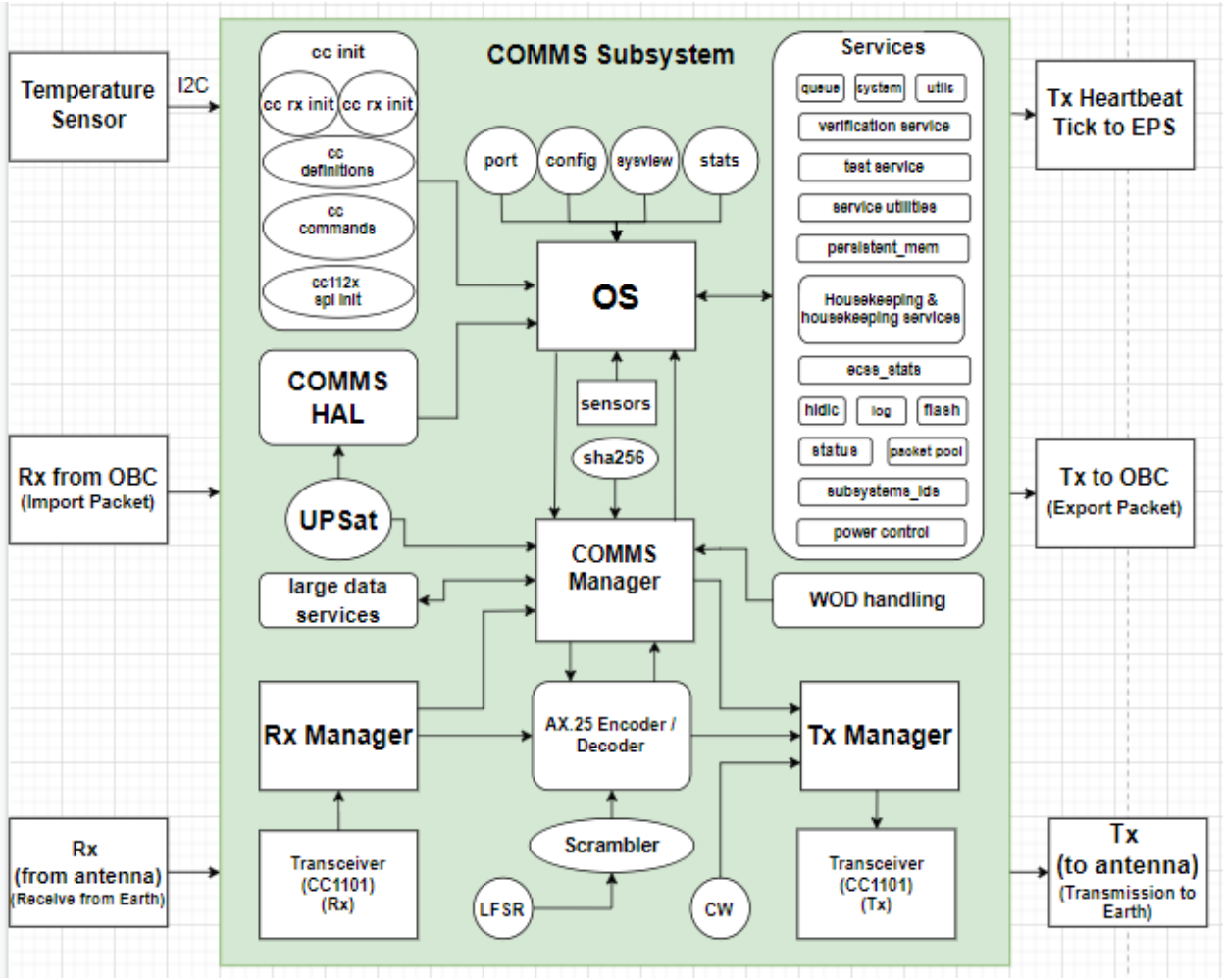


Figure 4.2. The Block-Diagram Representation of the COMMS Subsystem

COMMS Manager

This module is the brain of the COMMS subsystem. It handles the communications' main functionalities by communicating with the modules responsible for them.

Tx Manager

The module responsible for communicating with the HC-12 Transceiver module for the transmit operation. It makes sure the message is encoded and encapsulated as required before passing it to the module for transmission.

Rx Manager
The module responsible for managing the received data by the HC-12 Transceiver module that is responsible for the receive operation.

AX.25 Encoder / Decoder
X.25 stands for Amateur X.25. It is a data link layer protocol originally derived from layer 2 of the X.25 protocol suite and designed for use by amateur radio operators. It is used extensively on amateur packet radio networks.

It is the protocol used by radio amateurs in packet radio networking. AX.25 is a derivation (or adaptation) from the X.25 standard protocol used in public data networks.

AX.25 is commonly used as the data link layer for the network layer such as IPv4, with TCP used on top of that. AX.25 supports a limited form of source routing. Although it is possible to build AX.25 switches similar to the way Ethernet switches work, this has not yet been accomplished.

About X.25:
A packet-switching protocol for wide area network (WAN) connectivity that uses a public data network (PDN) that parallels the voice network of the Public Switched Telephone Network (PSTN). The current X.25 standard supports synchronous, full-duplex communication at speeds up to 2 Mbps over two pairs of wires, but most implementations are 64-Kbps connections via a standard DS0 link.

X.25 Structure:
Each X.25 packet contained up to 128 bytes of data. The X.25 network handled packet assembly at the source device, the delivery, and the reassembly at the destination. X.25 packet delivery technology included not only switching and network-layer routing but also error checking and re-transmission logic should a delivery failure occur. X.25 supported multiple simultaneous conversations by multiplexing packets and using virtual communication channels.

X.25 had three layers:

- Physical layer
- Data link layer
- Packet layer (corresponds to the network layer in the internet protocol).

X.25 pre-dates the OSI Reference Model, but the X.25 layers are analogous to the physical layer, data link layer and network layer of the standard OSI model.

With the widespread acceptance of Internet Protocol (IP) as a standard for corporate networks, X.25 applications migrated to cheaper solutions using IP as the network layer protocol and replacing the lower layers of X.25 with Ethernet or with new ATM hardware.

Link layer packet radio transmissions are sent in small blocks of data, called frames.
There are three general types of AX.25 frames:

a) Information frame (I frame);
b) Supervisory frame (S frame); and
c) Unnumbered frame (U frame).

Each frame is made up of several smaller groups, called fields. The following two figures illustrate the three basic types of frames. Note that the first bit to be transmitted is on the left side.

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

Figure 4.3. U and S frame construction.

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

Figure 4.4. Information frame construction.

Notes:

- The Info field exists only in certain frames
- FCS is the Frame Check Sequence field
- PID is the Protocol Identifier field

Each field is made up of an integral number of octets (8-bit byte of binary data) and serves a specific function.
All fields except the Frame Check Sequence (FCS) are transmitted low-order bit first. FCS is transmitted bit 15 first.

Scrambler and LFSR

The reason why this is necessary is that a transmission system has no control over the data the user is going to transmit. This causes problems because it violates the assumptions that are usually made when designing transmission systems, such as having independent data symbols. Note that usually the data are not random. A frequently occurring problem are long strings of zeros in the data, which can cause difficulties in timing recovery and adaptive equalization. These problematic sequences are removed (or, actually, made much less likely) by the scrambler.

A scrambler will result in a flat power spectrum, regardless of the data. This cannot be guaranteed with arbitrary user-supplied data.

The process of randomization of binary data is known as scrambling. The logic circuit which performs randomization is known as scrambler.

A scrambler is a generic linear feedback shift register with XOR gate.

A linear feedback shift register (LFSR) linear feedback shift register (LFSR) is a shift register whose input bit is the output of a linear function of two or more of its previous states (taps).

Since any register has a finite number of possible states, it must eventually be periodic. However, an LFSR with a well-chosen feedback function and initial-state can produce a sequence of bits which appears random (has good statistical properties) and which has a large period.

LFSRs can be applied in

- Generating pseudo-random numbers,
- Generating Pseudo-noise sequences,
- Fast digital counters
- Generating whitening sequences.
- Cryptography,

and others, and they can be implemented in both software and hardware.

There are two types of scrambler.

1. Additive or synchronous scrambler. It uses modulo-two addition.
2. Multiplicative or self-synchronizing scrambler, it performs multiplication of input signal by scrambler's transfer function in z-domain.

Scrambling helps achieve the following:

- It eliminates long strings of 0s to provide more transitions in the data. This helps the receiver for synchronization to recover the original bit pattern.
- It does not have any DC components as it creates balance between positive voltage levels and negative levels during the encoding process in line coding techniques such as R8ZS and HDB3.
- It offers error detection capability.

WOD Handling

Whole Orbit Data Packet (1856 bits)				
Time	Data set 1	Data set 2	...	Data set 32
32 bits	57 bits	57 bits	1653 bits	57 bits

Figure 4.5. WOD Packet Format

Data set X (57 bits)							
Mode	Bat. voltage	Bat. current	3V3 bus current	5V bus current	Temp. Comm	Temp. EPS	Temp. Battery
1 bit	8 bits	8 bits	8 bits	8 bits	8 bits	8 bits	8 bits

Figure 4.6. WOD Dataset

Extended WOD:

Since WOD offers only basic information, it was decided to add an independent extended WOD. That would provide more information about the state of CubeSat. It was asked of all engineers to supply a set of variables that would help them understand what happens in the subsystems. A small refactored happened in order to fit all data in a single frame. Extended WOD is encapsulated in an ECSS frame.

CW WOD:

In addition to WOD and extended WOD, a CW WOD was added. The reason was that CW has better chances of receiving it from the ground than the FSK modulated WOD and extended WOD. Moreover, in CW there isn't a need for complicated demodulation hardware, even a human with proper training can understand it. The disadvantage of CW and the reason that it hosts minimal information, is that it has far lower data rate than FSK and higher consumption due to lower data rate.

Specifications of CW WOD:

- **Modulation:** CW / Morse code.
- **Length:** approx. 10 seconds.
- **Interval (Period):** 5 minutes.
- **Speed:** 20 words per minute.

Large Data Services

The specification has 2 ways for ensuring that a transfer has been completed successfully. In the first one, every packet that is successfully received sends an acknowledgement. In order to send the next one, an acknowledgement should have been received for the previous one. The second technique uses a sliding window, where multiple parts are sent and the acknowledgement verifies the packets up to the part. Here, in our case, for uplink an acknowledgement for each packet is required and for downlink a modified sliding window is used.

The Large data transfer service has 2 different parts: the software running on the CubeSat and the one running on the ground station. Since the CubeSat has limited resources, it is more difficult to write code for it and after a point the software won't be changeable, it was decided that the operation complexity should be handled from the ground station's software, whenever that was possible. For that reason, there was a different approach for the service design if the transfer was initiated from the CubeSat (downlink) or from the ground station (uplink).

For downlink, the CubeSat sends all of the parts with 1ms delay in each transfer and waits for a specific timeout period. The first part is denoted as, all intermediates with and the final part with. During that period, the ground station should send for a packet retransmission if it didn't receive it properly or that all packets were successfully received. If that time period expires without an acknowledgement, the service aborts the transfer.

When the ground station receives large data transfer packets, it checks if there are any packets missing and asks for retransmission. If there aren't any packets missing, it sends that the transfer finished successfully. In the case that the last part hasn't been received, the ground station requests the next part, until the last one is sent.

For uplink, the ground station initiates a new transfer by sending the first part, if the CubeSat receives it, it sends an acknowledgement. If the ground station doesn't receive the acknowledgement, it retransmits the packet. This continues till the last part or if the operation timeouts.

For both of transfers uplink and downlink, the ground station is responsible of retransmitting a packet in a time frame that is less than the time of the CubeSat's timeout, if the ground station hasn't got a proper response, either a acknowledgment in the case of uplink or the part that ground station requested in a downlink.

For keeping the design simple enough, it was decided to allow only one transfer at the time for both of the uplink and downlink. For the case, it was decided in the future to allow multiple transfers, a large data unit ID was added in the data frame, that ID shows which transfer that packet belongs to. For the current design, if a packet arrives that has a different ID than current transfer, it drops the packet.

The sequence number shows where the data should be placed when the original packet is reconstructed. For uplink if a packet arrives that has a larger sequence number than the one expected, the packet is dropped, that simplifies the design and it doesn't allow broken packets. For the current design a maximum of 11 parts are allowed.

Each part should have the max size of 196 bytes from max data size of 198 bytes minus the 2 bytes of the large data transfer service header, except the last part.

SHA256

SHA-256 is a patented cryptographic hash function that outputs a value that is 256 bits long.
To clarify what hashing is, in encryption, data is transformed into a secure format that is unreadable unless the recipient has a key. In its encrypted form, the data may be of unlimited size, often just as long as when unencrypted. In hashing, by contrast, data of arbitrary size is mapped to data of fixed size. For example, a 512-bit string of data would be transformed into a 256-bit string through SHA-256 hashing.

Services

This part is not limited to the COMMS subsystem. It is shared among all of the other subsystems of the CubeSat: the OBC, EPS, ADCS and COMMS.
These services are referred to as ECSS services.
The ECSS-E-70-41A specification is a work of European Cooperation for Space Standardization and is based on previous experiences. For simplicity ECSS-E-70-41A would be refereed as ECSS in this book.
ECSS is a well-defined protocol and the specification is clear and well written. ECSS describes the frame header and a set of services. The header has a lot of optional parameters that makes it easily adapted to the application needs. The services listed are all optional and it's up to the user to implement them. Moreover, each service has a list of standard and additional features depending again on the application needs.

Each service is implemented in a module that is uncoupled with each other, The services module contains all the configuration parameters for the services. The service utilities contain functions that are helpful for the ECSS services. If a service needs information that is specific to a subsystem it uses the function that is defined in the platform folder in the respective file e.g. housekeeping for the housekeeping service. The subsystem-id file has all the application ids of the CubeSat.

OBC sends the WOD and extended WOD in 1 minute intervals. First it sends the requests to EPS and COMMS with 1 second delay between them and with the health report structure ID. When a response arrives, the data is temporarily stored in memory. If a response doesn't come the respective data are left with 0 value. A mechanism for re-requesting the data wasn't implemented. After 29 seconds OBC forms the WOD report with the responses from the

subsystems, stores it and sends it to COMMS for transmission. After the WOD comes the extended WOD. The same mechanism is used, only this time requests are sent to all 3 subsystems and the structure ID is extended health.

Since the WOD requires 31 historic values resulting in data logged 31 minutes ago, the persistent RAM region of the OBC was used. The SRAM was used instead of the SD card because it minimizes the data access time and improves reliability since the delete function of the FatFS is not invoked. For storage, a circular buffer was implemented in the SRAM containing 32 sets of values.

A single-entry point for all incoming packets is used for a better modular design and is denoted by the -app. Each function that belongs to a module starts with the service name e.g. mass-storage-app. The function must end with -api if that function could be used from another module but that rule was later discharged due to project time limitations.
Each subsystem sends a heartbeat packet (ECSS test service) to the EPS every 2 minutes. When the packet is received from the EPS, it updates a timestamp variable. If that timestamp minus the current time is longer than 20 minutes then the subsystem is reset. It doesn't have to be a heartbeat packet by any packet that will update the timestamp. The heartbeat is used because in normal circumstances, the ADCS and COMMS don't communicate with the EPS.

Since the OBC does the packet routing, if the OBC fails, the heartbeat packets from ADCS and COMMS destined to EPS won't be delivered. For that reason, the EPS checks if the OBC has an updated timestamp and then checks for the other subsystems. OBC and EPS have communication every 30 seconds for housekeeping needs, so it should be clear if the OBC works. Otherwise the OBC is reset and all subsystems timestamps are updated (ADCS, COMMS, OBC). This happens so the other subsystems are not reset in case of an error of the OBC.

External Interfaces
Since only COMMS and OBC is using the 2k data sizes and EPS has strict memory constraints, it was decided to have 2 types of packets: one normal with maximum size of 210 bytes and the extended that reaches 2k bytes of data payload. For simplification 2 types are used instead of 3.

The Hardware

- The Transceivers.
- The Microcontroller.

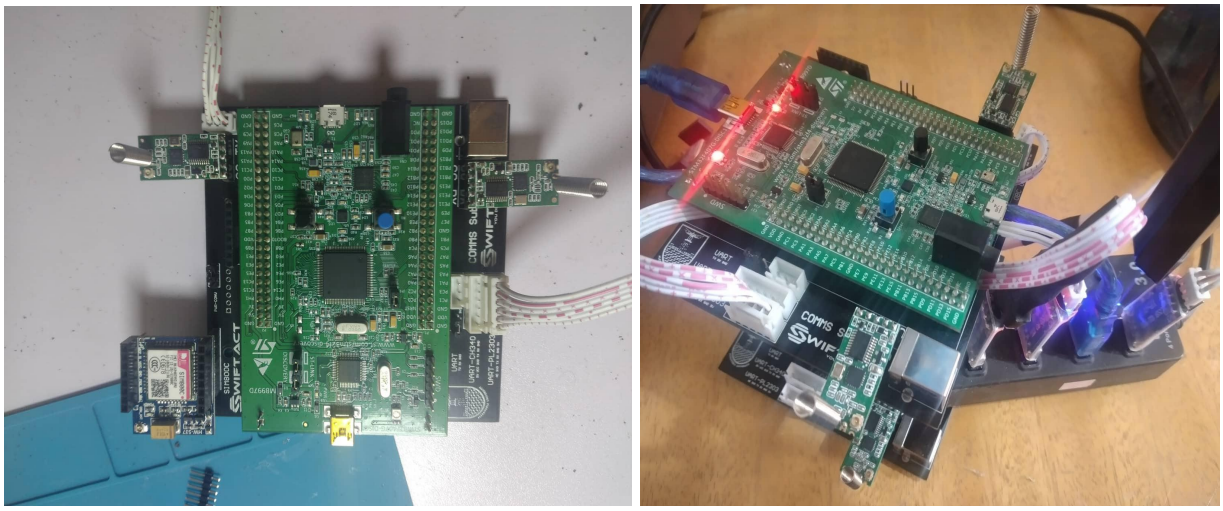


Figure 4.7. COMMS Hardware

The microcontroller is an STM32F407, ARM-based 32-bit microcontroller unit (MCU).

The transceiver modules used in the COMMS subsystem that are responsible for wireless transmission and reception are two HC-12 modules. Without going into deep technical details, the following lines describe this module.
HC-12 wireless serial port communication module is a new-generation multichannel embedded wireless data transmission module. Its wireless working frequency band is 433.4-473.0MHz, multiple channels can be set, with the stepping of 400 KHz, and there are totally 100 channels.

The maximum transmitting power of the module is 100mW (20dBm), the receiving sensitivity is -117dBm at a baud rate of 5000bps in the air.

Communication distance is 1000m (FU3 mode at 4800bps serial speed) in open space, 1800m in FU4 mode at reduced baud rate and volume of data.

The HC-12 module uses a Silicon Labs Si4463 to provide the RF communications link. This is a high performance, low current, single-chip “EZRadioPRO” family transceiver with up to 20dBm (100mW) transmitting output power. The Si4463 communicates through an SPI bus with an STMicroelectronics STM8S003F3 8-bit MCU that runs the HC-12 firmware. The STM8S provides a transparent serial data interface for interfacing to the module, allowing two HC-12 modules to act like a wired TTL level serial cable without any attached hardware devices needing to be aware of the RF link. Serial port and transceiver configurations are held in onboard non-volatile flash memory
The receiver is a multiple channel receiver using band selection filters to a common RF path. It’s a single board, communication to/from the OBC is done via two wire BUS.

Further specifications:

- Frequency range:

    - 119–1050 MHz
- Receive sensitivity

(*The threshold below which the receiver won't be able to detect the signal*)

    - –126 dBm
- Modulation:
    - FSK.
    - GFSK.
    - 4FSK.
    - 4GFSK.
    - MSK.
    - GMSK.
    - OOK
- Max output power
    - +20 dBm
- Data rate
    - Min.: 100 bps
    - Max.: 1 Mbps

System Realization

As with other communication paths, to realize the communication between the CubeSat and the Earth Station, a *Link Budget* calculation is necessary.
Following is a list for a brief overview on what a Link Budget is:

- It is a theoretical calculation of end-to-end performance for a communications path under a specific set of conditions.
- Sometimes the conditions are stated; most often at least some of them are implied or assumed.
- Every link budget implies everything not included is irrelevant, which is sometimes true.
- Link budget is a way of quantifying the link performance.
- A link budget is used to predict performance before the link is established.
    - Show in advance if it will be acceptable
    - Show if one option is better than another
    - Provide a criterion to evaluate actual performance

The criterion based on which a system is deemed realizable is the following condition:
If the received power, minus all losses, is greater than the minimum received signal level of the receiving radio, then a link is possible.

It is worth noting that the Link Budget Formula is written and handled in decibels (dBs) which simplifies the calculations into just additions and subtractions.

Multiple online tools help in designing links by providing Link Budget Calculations. We can use one of them for our communication link.

As mentioned earlier, the link budget is necessary in designing any radio (wireless communication) link, as it helps the designer determine the realizability of the link and if there is a possibility to establish communication that is reliable and that has certain criteria met (e.g. link margin, availability duration, etc.). Furthermore, the properties, specifications and details of the communication system's components can be more easily determined and tuned using the Link Budget (e.g. receiver sensitivity, transmitting power, etc.). Therefore, it not only provides information about the feasibility and the implementation of the wireless communication link, but it also helps in the determination of the properties and parameters that need tweaking and tuning in order for the system to meet a minimum level of acceptable performance based on the application being studied.

It helps with:

- Identifying the required antenna that can support such a communication link under the operation conditions studied.
- Identifying the minimum receiver sensitivity needed, which helps in determining the allowed region of motion as well as the transmitter power, transmitting antenna's gain, receiving antenna's gain, and more.
- Identifying the link margin.
- Identifying the required power from the transmitter, whether on the CubeSat or on the Ground Station.
- Identifying the antenna's gains.
- Tuning the transmitter power based on the distance, modulation scheme used, or others.
- Tuning other tunable parameters as distance, directions, data rate, modulation order, etc.
- After launching, the information provided from the link budget might help the ground station to give certain commands to the CubeSat, while in space, to adjust the various parameters involved in the transmitted signal based on the conditions.

If the link budget calculation showed that a wireless communication link can be established reliably and according to the required specifications and operation conditions, the design process can then proceed with a much reduced risk of wasting valuable resources on the design, equipment and implementation of an unreliable communication link.
However, if the link budget calculation results were evident that, given the conditions and specifications of the application at hand, a wireless communication link cannot be reliably established, a redesigning procedure shall then take place, in which the designer can tune the parameters that need changing in order to provide the required results as the calculations show, or it might be needed to replace the components with other ones that provide the required performance that supports the targeted application.

The choice of the antenna has to be carefully made in order to establish a reliable wireless communication link between the CubeSat to the ground station. Power conditions of the CubeSat have to be considered, as the sources of power are limited under these circumstances. This plays

a role in the choice of antenna. Basically, an antenna can be either directive or unidirectional, which is also known as the isotropic antenna. An isotropic radio transmitting antenna radiates its power, the transmitter power, equally in all directions. At a given distance *d* from the transmitter, the transmitted power is distributed equally on the surface of a sphere with a radius equal to this distance *d*.
The isotropic antenna has a gain that is equal to unity (or, in decibels, 0 dB). This is somewhat clear from the fact that it transmits its power equally in all directions, which means that all directions have a gain of 0 dB. The clear result from that is a need to increase the power transmitted compared to the case of using an antenna that provides a gain, i.e. a directive antenna. For the CubeSat, the case is not different at all. An isotropic antenna would require extra power in order to transmit signals, and power is scarce in the case of the CubeSat. Therefore, a directive antenna is needed in order to provide enough gain that makes the effective isotropic radiated power, or EIRP, reach the threshold required to support the communication link. In our case, however, an omnidirectional (isotropic) antenna is used on the CubeSat.
The Major characteristics of antenna mainly include:

- Antenna gain (dBi)
  (where i stands for isotropic antenna)
- Antenna Pattern (Wide (broad) or narrow half power beam width (HPBW) (measured in degrees).
- EIRP (Effective Isotropic Radiated Power). EIRP = P_T x G_T, or, in dB, $P_T + G_T$
- G/T (Antenna gain-to-noise temperature).
- VSWR (Voltage Standing Wave Ratio).
- Polarization (Linear or Circular).
- Deployable or non-deployable.

The antenna gain depends on the antenna design; while parabolic antennas feature antenna gains of 60 dB and more, a Yagi-type antenna has an antenna gain of some 20 dB (depending on the number of directing – elements in the design) and a dipole antenna has an antenna gain of 2 dB. The following figure shows some types of antennas:

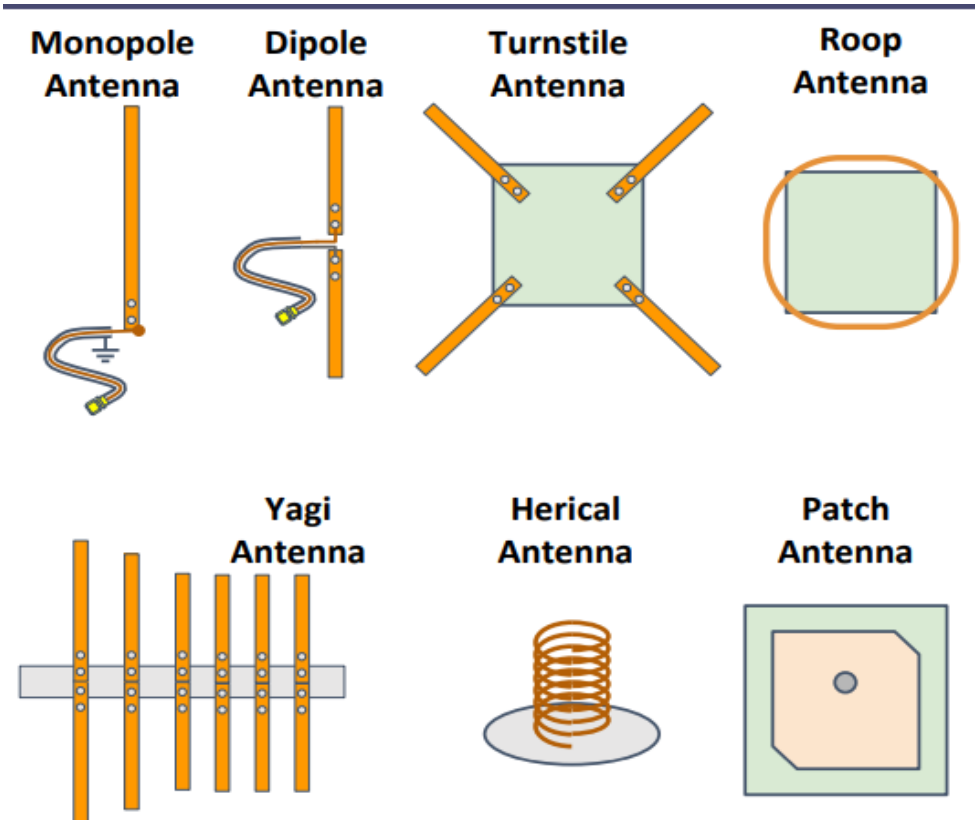


Figure 4.8. Various Antenna Types

Using equation (4.1), we perform the link budget calculations:

$$P_{R\ (dBW)} = EIRP_{\ (dBW)} + G_R - LOSSES \qquad \text{eq. 4.1}$$

LOSSES include:

- FSPL: Free Space Path Loss, which is calculated as 20log(R) + 21.98 dB - 20log(λ), where R is the Slant Range in meters, and λ is the wavelength in meters.
- L_{pol}: polarization mismatch losses
- L_{TL}: transmission line loss
- L_{atm}: losses in atmosphere.

The values of the different parameters are as follows:

- Downlink Frequency: 433 MHz.
- Slant Range: 450 Km
- CubeSat:
    - G = 0 dB (isotropic).
    - Output Power = 1 watt.
- GS:
    - G = 33.1 dB

Therefore, using eq. 4.1,

- EIRP = 0 dBW
- G_R = 33.1 dB
- LOSSES
    - FSPL = 138.2 dB
    - L_{pol} = 0.2 dB
    - L_{TL} = 0.75 dB
    - L_{atm} = 4.1 dB,

  Which totals 143.2 dB

P_R = -110.1 dBW = -80.1 dBm

From the previous calculation of P_R, after adding a Link Margin, we can find out what the GS sensitivity requirements are.

Communication in the CubeSat is a very long-term activity (satellite projects start from frequency allocation, then satellite projects finish with sending the RF transmitting termination command). Antennas apparently are the most critical item for successful communication of a CubeSat.

CHAPTER FIVE

OBC Subsystem

The On-Board Computer can be thought of as the brains of the satellite. It's also known as the Command & Data Handling subsystem, as it not only handles the execution of any commands sent by our Ground Station, but also any data that is collected and sent back for analysis. It also serves another critical function, which is to coordinate between all of the different subsystems of the satellite, maintaining proper execution of periodic and aperiodic tasks to ensure correct operation.

1- REQUIREMENTS

The main functions of this subsystem can be summed up as follows:

- **Ensure correct operation of the satellite:** the system should be able to handle any unexpected faults, and should recover from them as swiftly and with as minimal damage as possible. It should also make sure that its main tasks are executed in a timely manner.
- **Process and execute commands:** the system should be able to understand commands sent from our GS and ensure its execution on the relevant subsystems.
- **Perform housekeeping tasks:** the system should periodically log the current state of the satellite along with critical readings.
- **Operating the mass storage memory:** the system should operate the mass storage memory used for logs and configuration storage.
- **Provide data interfaces to all of the other subsystems:** the system should have adequate interfaces to connect to all of the other subsystems by using UART.
- **maintaining UTC time.**

2- ARCHITECTURE

The OBC architecture is essentially based on the connectivity between subsystems within the CubeSat. This simply means that the microcontroller's peripherals are configured according to the data flow within the CubeSat's computing scheme. This has several benefits:

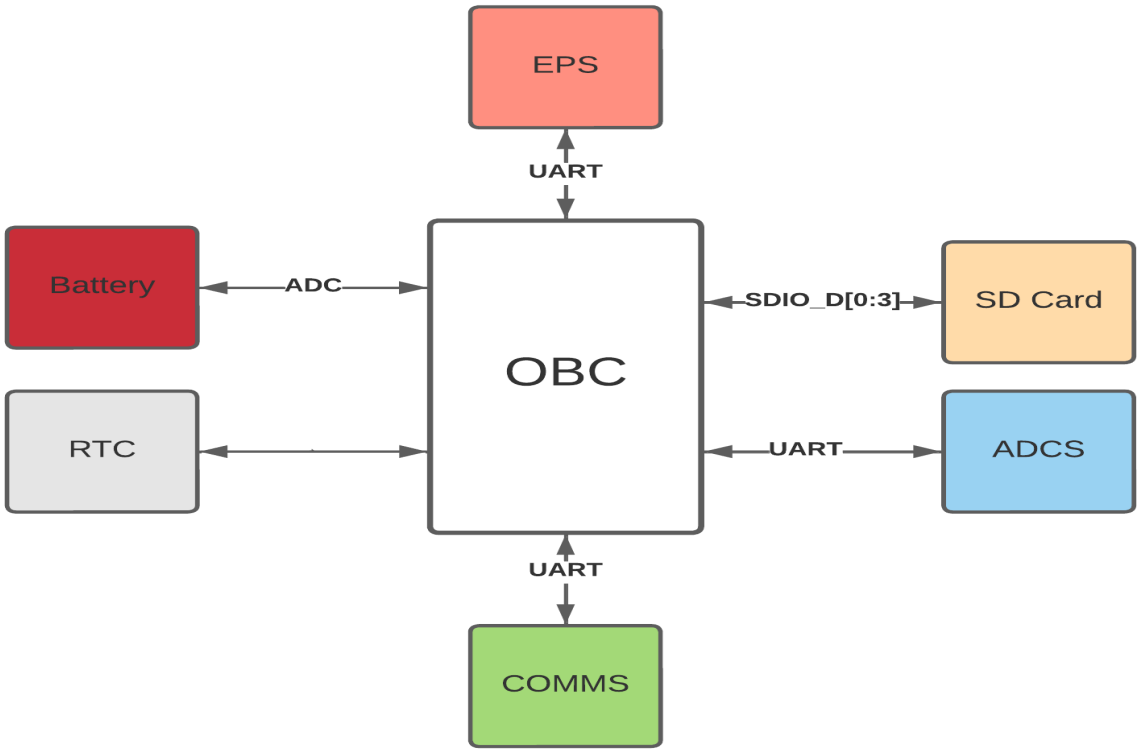


Figure 5.1: System topology.

- **No single point of failure:** as there's no common bus for all of the subsystems, it allows for more leeway to act upon failures in which a single link is down, as the other links would be unaffected.
- **Better performance:** as the subsystems do not need to share the same bus.
- **Flexibility:** as the various modules can have different interfaces than one another while still maintaining proper functionality, as long as the OBC supports it.

3- HARDWARE

- We are using a STM32F405 with an ARM cortex M4 cpu core that has 1 Mbyte of Flash and 192 Kbytes of SRAM; shown in Figure 5.2, as our On-Board Computer, which provides processing power, memory and adequate interfaces to connect to the rest of our satellite

- The microcontroller's internal Real Time Clock connected with a coin cell battery.
- An SD card connected with SDIO.
- IS25LP128 128 MBIT Flash memory connected with SPI.

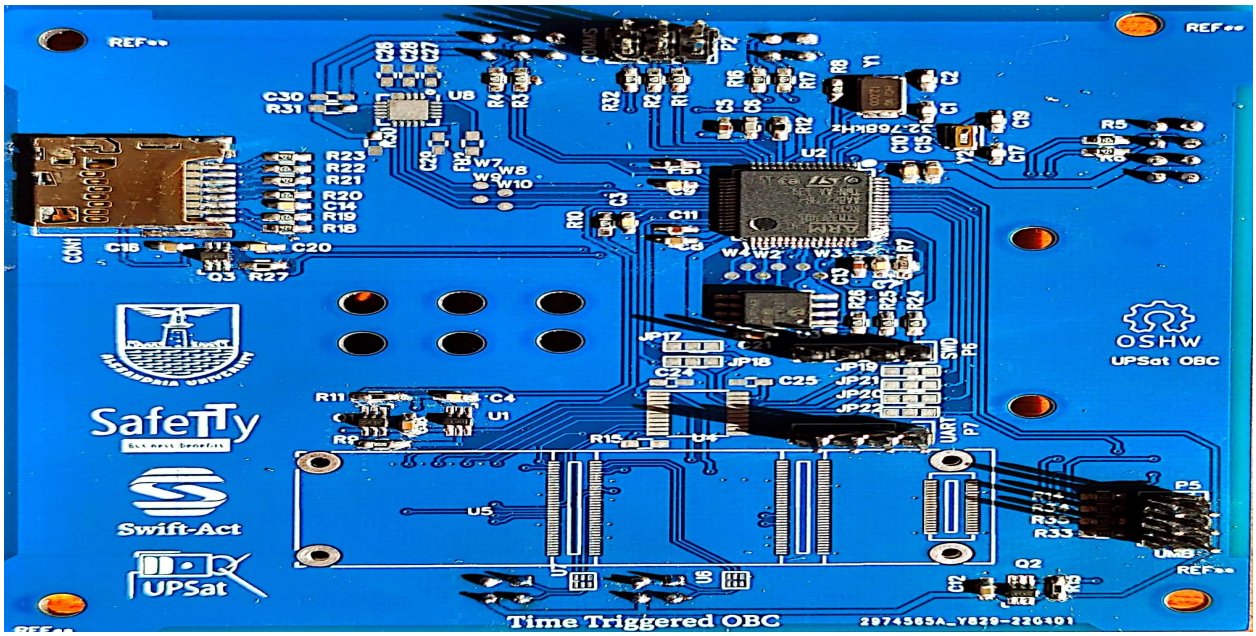
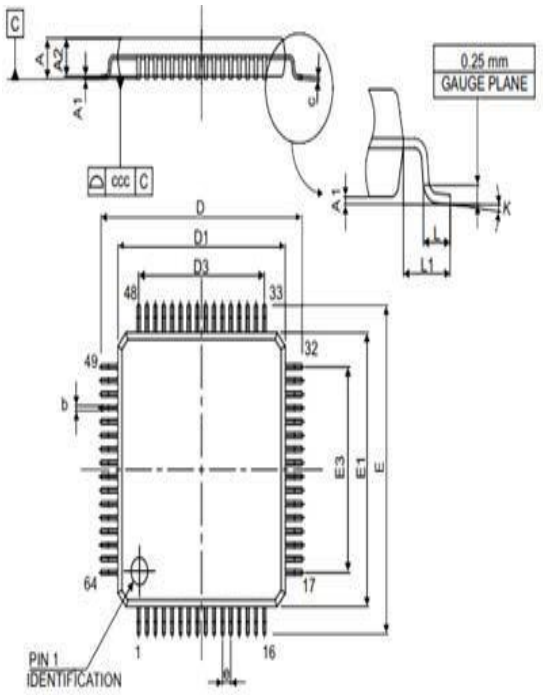


Figure 5.2: OBC Subsystem

It fits our desired specifications, including:

- **A powerful yet efficient CPU:** with an ARM cortex M4 cpu core processor, which is more than adequate for our application.
- **A large amount of RAM:** with 1 Mbyte of Flash and 192 Kbytes of SRAM, while not being the absolute fastest it still provides a very large amount that would allow running tasks with room to spare.
- **Low power consumption:** drawing less than 0.5W of power while idle.

- **Large community support:** with available documentation and test results.
- **Commercial availability:** being readily available in most markets.
- **Flexible I/O:** Up to 15 communication interfaces (including 6x USARTs running at up to 10.5 Mbit/s, 3x SPI running at up to 42 Mbit/s, 3x I²C, 2x CAN, SDIO), 12-bit ADCs, two DACs, a low-power RTC, twelve general-purpose 16-bit timers including two PWM timers for motor control.
- **Small size:** which saves valuable space inside of our mechanical structure, where it's at a premium. The mechanical diagram is shown in Figure 5.3.



Symbol	millimeters			inches[1]		
	Min	Typ	Max	Min	Typ	Max
A	-	-	1.600	-	-	0.0630
A1	0.050	-	0.150	0.0020	-	0.0059
A2	1.350	1.400	1.450	0.0531	0.0551	0.0571
b	0.170	0.220	0.270	0.0067	0.0087	0.0106
c	0.090	-	0.200	0.0035	-	0.0079
D	-	12.000	-	-	0.4724	-
D1	-	10.000	-	-	0.3937	-
D3	-	7.500	-	-	0.2953	-
E	-	12.000	-	-	0.4724	-
E1	-	10.000	-	-	0.3937	-

Figure 5.3: STM32F405 mechanical diagram.

4- CUBESAT MODES OF OPERATION

Table 5.1: Modes of operation.

Command	*Function*

Start	Start state.
Initialization	Boot up OBC, start initializing all subsystems.
Idle	Only maintains critical functionality
Nominal	Normal operating mode.

The satellite has different modes of operation to account for different states, they can be summarized as follows:

- **Start state:** when first powered on, the satellite should not go back to this state after the first time.
- **Initialization sequence:** power up the OBC, and initialize links to all connected subsystems.
- **Safe mode:** when a critical failure occurs, all non-mission critical functions should be halted.
- **Nominal mode:** all subsystems are up and running, payload can be used at this state.

5- COMMUNICATIONS PROTOCOL

The OBC is also responsible for the processing of commands and data, adding appropriate metadata to ensure accurate delivery. In this case, we're using ECSS (European Cooperation for Space Standardization) protocol.

COMMAND AND CONTROL MODULE

The CnC (command and control) module, defines the protocol for earth to satellite (and vice versa) communication and inter subsystem communication. It consists of the packet format, header and data definition. Operations are grouped into services, defined by the protocol.

REQUIREMENTS

The following requirements were set in order to evaluate ECSS:

- Low protocol overhead.
- Lightweight.

- Highly modular and customized.

Since all interactions to subsystems and earth uses the protocol, if the protocol overhead isn't efficient, it leads to power waste and added data traffic.

The CnC module will be used with microcontrollers that have limited processing power and resources, so it needs to be lightweight.

The protocol should be designed in way that allows some degree of customization, so the implementation will be tailored on the resources available and be efficient.

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 5.4: ECSS TC frame header

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)

Figure 5.5: ECSS TC data header

The ECSS-E-70-41A specification is a work of European Cooperation for Space Standardization and is based in previous experiences. For simplicity ECSS-E-70-41A would be refereed as ECSS in this document.

ECSS is a well-defined protocol and the specification is clear and well written. ECSS describes the frame header and a set of services. The header has a lot of optional parameters that makes it easily adapted to the application needs. The services listed are all optional and it's up to the user to implement them. Moreover, each service has a list of standard and additional features depending again to the application needs.

The header has 2 parts: a) the packet header, which is standard, and is 6 bytes. b) the data field header, which is variable due to different options available. The options are: packet error control (Checksum [27]), timestamp, destination id and spare bits (so the that the frame is in octet intervals).

Depending on the application and the number of distinct application ids, the user can select to have only 1 set of application ids, defining both the source and destination in one byte, If the number of application ids is large than a separate source/destination has to be used.

The header implemented was 12 bytes. The switch between TC/TM and source/destination can be confusing. One feature that the protocol lacks is that it doesn't have a mechanism for different configurations on runtime, having configuration bits denoting different configurations, such as the existence of a timestamp or a checksum in the packet, would have made the protocol a lot more versatile.

ECSS is packet oriented e.g., mass storage service use of store packets, even though it's not a huge disadvantage, it can a bit tedious and could lead to inefficiency on the implementation.

For the following reasons, it was decided to use the ECSS protocol.

- ECSS was recommend by QB50.
- ECSS is used by many other cubesats.
- ECSS is based on experience on previous protocol designs, as a result the protocol is highly refined.
- ECSS is highly flexible on the actual implementation and it allowed to customize it according to our needs.

ECSS Services

The ECSS standard is highly adaptive and provides many different choices. In this section, the design choices for CubeSat are analyzed.

Services

- The telecommand verification service provide a way to receive a response about the successful or not outcome of a telecommand's operation. This service is required since for some operations it is critical to know the outcome of the operation.
- The housekeeping & diagnostic data reporting service provide a way to transmit and receive information (housekeeping) that denote the status of the CubeSat. The housekeeping operation is standard in CubeSats.

- The function management service is used for operations that aren't part for other services operations. In CubeSat the service is used mainly for controlling the power in subsystems and devices and setting configuration parameters in different modules.

- The time management service is providing a way to synchronize time between subsystems and the ground station. This service was added later when the need to synchronize time between ADCS and OBC and the ability to change the time from the ground came up.

- The on-board operations scheduling service provide a way for to trigger events in specific times or continuously with specific intervals with the release of telecommands. This service allows to perform events without having connection with the ground station.

- The large data transfer service provides a way to exchange packets that are larger than the size that is allowed by cutting the original packet in chunks that their size is allowed. In CubeSat it is used for transferring large files.

- The on-board storage and retrieval service provide a way to store and retrieve information in mass storage devices. In CubeSat it is used to store various logs and configuration parameters in the SD card of the OBC.

- The test service provides a simple way to verify that a subsystem is working. It is very similar to the ping program used in IP networks.

- The event reporting service provides a way for a subsystem to report events. It was originally designed for subsystems that didn't have storage devices to report events that were critical to the CubeSat's operation to the OBC so that it would store them for later review from a human operator. Not implemented.

- Event-action service uses the event service to generate action when a particular event takes place. Since the event service was removed there wasn't a way to use the event-action service.

From the 16 services only 8 were decided to with the time management service added later and the event reporting services removed during the implementation phase.

The specification states that any custom services or services subtypes should have an number larger than 128. In CubeSat's design this rule wasn't followed and the custom could have any number that it's not used from the specification. This happens because there is a a large lookup table for every subsystem which defines which services are used, the way it was implemented having 128 service number and above would create a huge lookup table that wouldn't fit in the microcontroller's memory.

Application ids

Application ids are a core concept in ECSS, it is the address of a module that the packet is heading towards, it is very similar to the IP address concept. Using 11 bits for application ids a

total of 2047 address can be achieved. Application ids are not confounded only in hardware subsystems but software modules can be given an id.

In CubeSat a total of 6 application ids were used: 4 for each subsystem and 2 for the ground station. The 2 application ids used for the ground station is because there are 2 different paths available and there was a need to differentiate them. The first is the serial connection through the umbilical connector and was used only during testing. The second was through the RF communication and the COMMS subsystem. The software is design of CubeSat is simple enough so there wasn't a need for more application ids.

Packet frame

Even if the ECSS standard treats the telecommand and telemetry as packets with different frame structure, the design intention of the packet frame in CubeSat was that both of frames could be as identical as possible. The reason behind that was that the software remains as simple as possible, using the same code for manipulating telecommand and telemetry frames. The simpler design in software would lead in less developing the code and testing it.

The packet header and packet error control are identical in telecommands and telemetry packets. The data field header is designed to be identical with source ID in telecommands and destination ID in telemetry packets and without the optional fields of packet sub-counter and time in telemetry that don't exist in a telecommand packet.

For routing purposes, the source and destination application ID was added in telecommand and telemetry packets. When the packet is a telecommand the application ID in packet id denotes the subsystem destination and in the data header field the subsystem that the packet originated and vice versa in a telemetry packet. Without the source ID, it would be impossible to know to which subsystem a possible response should be send and without the destination ID where to route the telemetry packet. It was possible to only use the application ID by having application IDs would denote both the source and destination ID but that design would be more obscure leading to confusion during testing.

The maximum length of a normal frame is 210 bytes, by subtracting the headers and error correction it leaves with 198 bytes for application data. This number derives from restrictions in RF communication with the Earth. Because the COMMS subsystem doesn't use error correction algorithms, if the size is larger than 210 bytes the probability that the packet is received correctly from the ground station, quickly deteriorates.

For the case of handling large files the normal packet size is inefficient and restrictive. For that reason and for subsystem communication only, the length of a packet can be extended to a maximum of 2050 bytes. This transaction happens only for OBC-COMMS communication only. For RF communications if the size is larger than normal, the large data service is used.

The version number and data field header flag of the packet ID have the default values of 0 and 1.

The type equals to 1 if the packet is a telecommand and 0 if it is a telemetry packet.

The application ID uses only the 8 bits for efficiency but for compatibility reasons 11 bits are used in the frame.

The Sequence flags in telecommands or Grouping flag in telemetry packets are used only in standalone mode with default value equal to 3.

The sequence count is a counter that counts the packets that the subsystem has transmitted to another subsystem, there is a different counter for each application id. If a subsystem routes the packet to its indented destination, it doesn't modify the counter. The counter is uses 8 bits instead 14 bits as the standard for efficiency reasons. For compatibility reasons for the packet frame remains 14 bits. Every system that transmits packets frames needs to implement the counter. Moreover, the counter in CubeSat is not stored in mass storage memory and it is reset to zero in each subsystem reset. By observing when the counter resets to zero in a subsystem it can be deduced that a reset happened to that subsystem.

The packet length is calculated by subtracting from the actual packet size the size of the packet header which is 6 bytes and subtracting 1.

Packet length = Packet size (bytes) - 6 (packet header) - 1

The data field header varies in a telecommand and a telemetry packet. The CCSDS Secondary Header Flag has a default value of 0 in a telecommand and it's used as padding in a telemetry packet. The Ack is used in a telecommand from the verification service and as padding in a telemetry since the verification service doesn't work with telemetry packets.

In both telecommand and telemetry packets the Packet PUS Version Number has a default value of 1.

The service type and subtype denote the functionality of the packet and the service associated with the packet.

In a telecommand the source ID denotes the application ID of the subsystem that the packet originates from and in a telemetry packet the destination ID denotes the destined subsystem. Both the source and destination ID reside in the same position in a telecommand and telemetry packet.

The Packet error control is implemented as a CRC8 algorithm that occupies 8 bits but for compatibility reasons the frame has 16 bits with the first 8 bits are unused.

The packet sub-counter and time fields in a telemetry packet are not used because there wasn't any need for them in CubeSat.

Finally, no optional spare fields were added in both telecommand and telemetry packets.

Table 5.2: ECSS TC data header

Packet Header (48 Bits)						Packet Data Field (48 + MAX 1584)		
Packet ID				Packet Sequence Control		Packet Length	Data Field Header	Application data	Packet error control
Version Number (=0)	Type (=0)	Data Field Header Flag	Application Process ID	Sequence Flags	Sequence Count				
3	1	1	11	2	14				
16				16		16	32	MAX 1584	16

Services in subsystems

Verification, housekeeping and test service are basic services needed in all subsystems.

Time management is only used in ADCS and OBC since only them require precision time keeping, OBC for the ADCS for the control calculations.

On-board scheduling and On-board storage services are only used in OBC since they require a mass storage device.

Finally, large data transfer is only implemented in COMMS since it is the only capable for RF communications.

Telecommand verification service

The Telecommand verification service is the 1. In Table 5.3 are shown the minimum and additional capabilities offered by the services. The service is used when a telecommand has the values shown in Table 5.3 The service doesn't support telemetry packets.

For CubeSat it was decided that only the minimum capabilities would be used, even if the additional would be definitely helpful they would also complicate the software design. For that reason, only values of 0 and 1 are valid in the ACK field, if other values are present the packet is flagged as invalid and dropped.

Since most of the telecommands are usually finished immediately, the acceptance report means also the competition of the telecommand but the semantics of the acceptance should be considered to be a telecommand.

If the telecommand results in failure, the frame has an error code field. By checking the error code, the ground station operators could find the reason for the failure and make correcting procedures accordingly. The ECSS standard provides some error codes about packet decoding failure listed in Table 5.3.

One particular idiosyncrasy of the service is found in the Table 5.7 where are listed errors about packet decoding such as error 2 incorrect checksum, that leads reporting acceptance failure about a packet that could have corrupted information including the ACK field.

Table 5.3 Telecommand packet data ACK field settings

Value	Value meaning
0	none
1	Acknowledge acceptance
2	Acknowledge start of execution
4	Acknowledge progress of execution
8	Acknowledge completion of execution

Table 5.4 Telecommand verification service subtypes

Telecommand acceptance report
success Telecommand acceptance report
failure Additional capabilities Telecommand execution started report
success Telecommand execution started report
failure Telecommand execution progress report
success Telecommand execution progress report
failure Telecommand execution completed report
success Telecommand execution completed report

Table 5.5 Telecommand verification service acceptance report frame

Packet sequence control	16 bits 16 bits

Table 5.6 Telecommand verification service acceptance failure frame.

Packet sequence control Error	16 bits 16 bits 8 bits

Table 5.7 Telecommand verification service error codes

Value	Value meaning
0	Illegal APID
1	incomplete or invalid length packet

2	incorrect
3	illegal packet type
4	illegal packet subtype
5	illegal or inconsistent application data

On-board operations scheduling service

There are two scenarios where the capability for the on-board execution of operations that have been loaded in advance from the ground shall be implemented:

- Those missions that perform operations outside of ground contact because of limited ground station visibility (e.g., LEO spacecraft) or signal propagation delays (e.g., deep-space probes).
- Those missions whose operations concept is to minimize the dependency on the ground segment. Thus, a geostationary telecommunication or meteorological mission can perform all of its routine operations in this manner, even though the spacecraft is permanently in view of a ground station. This approach potentially increases the availability of operational services or mission products, since the continuous availability of the uplink is eliminated.

The simplest form of on-board operations scheduling -that has been implemented-consists of storing time-tagged commands that have been loaded from ground and releasing them to their destination application process(es) when their on-board time is reached.

Housekeeping

Information indicating the status of the CubeSat, are broadcasted to earth, in specific intervals. WOD is transmitted automatically, so it can be easily gathered by ground stations that don't have transmit capabilities.

CubeSat has 3 different WODs and each is used for different purposes:

- QB50 WOD.
- Extended WOD.
- CW WOD.

The QB50, extended and CW WOD, is used for understanding the state of CubeSat. It is the last line of defense in the case there isn't a communication link between ground stations and CubeSat. It is going to be the first indication if CubeSat works correctly or not. The CW WOD is crucial during the first days of operation, in order to track and verify the operation of CubeSat. Since HAM operators around the globe could listen for CW WOD, a global coverage can be obtained.

WOD

In the QB50 requirements, there is WOD. In the frame, it provides historical information. The dataset provides general information. For compatibility with the rest of the missions in QB50, WOD is not encapsulated in ECSS.

Table 5.8 WOD packet format

Whole Orbit Data Packet (1856 bits)				
Time	Data set 1	Data set 2	...	Data set 32
32 bits	57 bits	57 bits	1653 bits	57 bits

Table 5.9 WOD dataset

Data set X (57 bits)							
Mode	Bat. voltage	Bat. current	3V3 bus current	5V bus current	Temp. Comm	Temp. EPS	Temp. Battery
1 bit	8 bits	8 bits	8 bits	8 bits	8 bits	8 bits	8 bits

Extended WOD

Since WOD offers only basic information, it was decided to add an independent extended WOD. That would provide more information about the state of CubeSat. It was asked of all engineers to supply a set of variables, that would help them understand what happens in the subsystems. A small refactored happened in order to fit all data in a single frame. Extended WOD is encapsulated in a ECSS frame.

CW WOD

In addition to WOD and extended WOD, a CW WOD was added. The reason was that CW has better chances of receiving it from the ground than the FSK modulated WOD and extended WOD. Moreover, in CW there isn't a need for complicated demodulation hardware, even a human with proper training can understand it. The disadvantage of CW and the reason that it

hosts minimal information, is that it has far lower data rate than FSK and higher consumption due to lower data rate.

Housekeeping & diagnostic data reporting service

In this service, the design has deviated a lot from the specification. In the specification, each housekeeping structure send a report in specific intervals. There are also functions that implement new structures and modify the intervals that the reports are generated.

For the CubeSat, a different, simpler design was followed: OBC would send a telecommand report parameters request (3,21) and the subsystem would respond with a telemetry parameters report (3,23). OBC handles all the timing and not the service. In the request, there is the structure ID that the OBC wants. The structure ID is most of the times general and the parameters in report are in conjunction with the subsystem that reports it. For WOD variants the OBC gathers the health and extended health reports, forms the WOD variant structures and then transmits it to earth. Some subsystems have structure IDs that are specific to them. Finally. the ground station could also send a request for a parameter report and receive the response.

Even though the request/response mechanism works well, a design that uses intervals as specified in the ECSS standard would have been a better solution for 3 reasons:

- It removes the task of sending requests.
- Minimizes traffic.
- Easier to change intervals.

Table 5.10 Housekeeping service structure IDs

Structure ID name	Structure ID Meaning
HEALTH_REP	Health report
EX_HEALTH_REP	Extended health report
EVENTS_REP	Events report
WOD_REP	WOD report
EXT_WOD_REP	Extended WOD report
ADCS_TLE_REP	ADCS TLE report
EPS_FLS_REP	EPS flash memory contents report
ECSS_STATS_REP	ECSS statistics report

Table 5.11 Housekeeping service request structure id frame

Structure ID	8 bits

Table 5.12 Housekeeping service report structure id frame

Structure ID Data	8 bits 1 - 1584 bits

On-board storage and retrieval service

The first design choice was that the file names for the logs should only be numbers and not characters. That has the advantage of lower size overhead when there is ground to CubeSat communication e.g., the file name as a string could be 8 characters long meaning 8 bytes as string when the equivalent number 99999999 only uses 4 bytes. Moreover, file operation could be performed with simple mathematics e.g. The next file name could be found by adding 1 to the current file name.

For the logs storage design, it was decided to use one file per log entry. This approach adds great size overhead since the minimum size a file occupies is 512 bytes but simplifies the actions needed to operate (retrieve, delete, store) with a log entry. A maximum file number of 5000 is defined, that way it is ensured that logs can't consume all the disk size.

For the logs, on top of the file system an extra layer was added. A circular buffer was used with the head and tail pointers pointing into files. All logs must be placed in sequential manner into files. Using that mechanism, the logs operation is simplified: First when a log entry is deleted it doesn't actual delete the file, leaving the data intact in case there is a need to retrieve it and saves time by not calling the delete function of the file system. Secondly when a downlink operation happens there is no need to know the actual file name, only the log number should be specified and the file name is found by adding the head pointer file name number and the log number e.g. if the head points to a file with a name 5 the 3rd log that is stored is found by adding 5 and 3 resulting in the file name 8.

Service subtype Enable (15,1) and Disable (15,2) control the power of the SD card, there is no data used. Instead of using the function management for power control of the SD it was decided to use the specification's mechanism.

Service subtype Downlink (15,9) and the response Content (13,8) provide a way to download a file from CubeSat. The telecommand downlink provides the storage ID, the file name and if batch download is needed, the number of files. Batch download is used for logs only. For batch download, the files are sequential to the file name specified in the telecommand. The response has the storage id, the file name and the file content and if batch download is used, the next file name and file content.

Service subtype Delete (15,11) has multiple functionalities. The first field is the storage ID that the delete function should act. Second field is the mode. Depending on the mode the delete has different functionality. The mode FS reset with a logs storage ID, resets the file system, this is used in the case there is an issue with the file system. The hard delete mode is used with the logs storage ID and deletes all the files in storage ID. Also, when the hard delete is used with the SRAM storage ID, it initializes the static memory region on the OBC to zero. The hard delete mode should be only used when there is an issue with file system. The delete all mode used with logs clears the pointers used and finally the delete to mode used again with logs removes entry logs from the pointers.

Custom service subtype 15 uses the FatFS format function and formats the SD card. This was added in the case the file system gets corrupted and it's unusable from the OBC.

TEST SERVICE

The test service is very simple. It has the 17 service type and only 2 subtypes: perform test (17,1) and report test (17,2). The service doesn't use any application data.

Table 5.19 On-board storage and retrieval service uplink subtype frame

Store ID File File data	8 bits 16 bits 1 - 16384 bits

Table 5.20 On-board storage and retrieval service downlink subtype frame

Store ID File Number of files	8 bits 16 bits 16 bits

Table 5.21 On-board storage and retrieval service downlink content subtype frame

Store ID File File data	8 bits 16 bits 1- 16384

Table 5.22 On-board storage and retrieval service subtypes used on CubeSat

Name Service subtype	Explanation
ENABLE	Turns on the SD card
DISABLE	Turns off the SD card
CONTENT	File contents
DOWNLINK	Download a file request
DELETE	Delete files
REPORT	Storage status report request
CATALOGUE REPORT	Storage status report response
UPLINK	File upload
FORMAT	Formats the SD card
LIST	List file information request
CATALOGUE LIST	List file information response

Table 5.23 Store IDs

Name	ID
WOD LOG	9
EXT WOD LOG	10
EVENT LOG	11
FOTOS	12
SCHS	13
SRAM	14

6- Implementation

In this chapter, the implementation of the ECSS services and the OBC software is discussed.

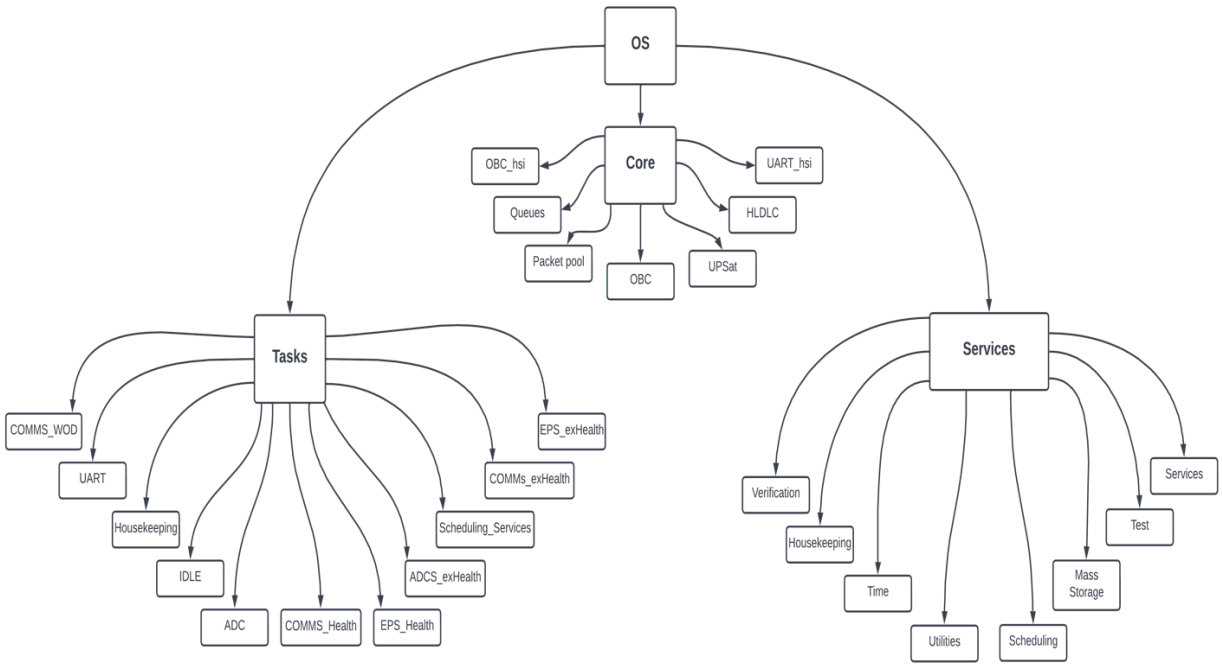


Figure 5.6: Project Organization

HLDLC

Protocols like SPI and $I\^2C$ it is possible to transfer multiple byte with one transaction. That way a transaction signifies a packet transfer. UART on the other hand transfers a byte at the time thus leaving no way to know when a packet starts or stops. For that reason, for UART subsystem communication, there are 3 options to pass the ECSS frames:

- ASCII characters.
- Binary protocol with length.
- HLDLC algorithm.

Use ASCII and have unique in the frame delimiters like ‘,’ and newline for frame endings. The use of ASCII is a good choice because it is easier to implement, it is easier for a human to debug since the data can be read. Unfortunately, ASCII requires more processing power in order to convert strings to numbers, requires more time to transfer and need more memory than a binary protocol. A good example for an ASCII protocol is the NMEA, which is used in all modern GPS receivers.

Use common delimiters but have a length field with the number of bytes in the packet, in the beginning of each packet and use that to calculate when the packet ends. This approach would require a timeout to discern different packets in the case there is an error. The advantage of that approach is that the implementation is easy and quick. The main disadvantage is the error handling is very poor in the case there are failures in the transfer.

The method that was eventually used, uses an algorithm that modifies the original data so there are unique delimiters. The disadvantage it has is that it could add up 2x times size overhead of the original data and it cannot be used when a fixed packet length is required. The main advantages are that is fairly easy to implement, it doesn't use much processing resources, O(n) time is needed and makes fault tolerant mechanisms easy to implement. A similar mechanism is used for CSP when the transfer is over UART. During testing the overhead that was observed in most packets was minimum.

The HLDLC is very easy: There is the frame boundary byte, which is 0x7E that signifies the start and stop of every packet and the control escape byte 0x7D. In the case there is the frame boundary byte or the control escape byte inside the data frame, a control escape byte is inserted and the 5th bit is inverted e.g. if there is the 0x14 0x7E 0x55 0x7D 0x14 byte frame the frame is modified into 0x7E 0x14 0x7D 0x5E 0x55 0x7D 0x5D 0x14 0x7E .

The HLDLC module is very simple, it has 2 functions: one for adding HLDLC in a packet and one to deframe the HLDLC. The functions only need the buffer that has the data and the buffer that stores the modified data, plus the size of the original buffer that returns the size of the modified data. The deframe function checks for HLDLC integrity. The return code is SATR-EOT when it was successful or SATR-ERROR when there was an error.

Listing 0-1 HLDLC module functionsh

```
SAT_returnState HLDLC_deframe(uint8_t *buf_in, uint8_t *buf_out, uint16_t *size);

SAT_returnState HLDLC_frame(uint8_t *buf_in, uint8_t *buf_out, uint16_t *size);
```

Packet Pool

Since malloc is prohibited, there was a need for a memory storing mechanism for ECSS packets.

A fixed array of the packet structure was used. An array with the same number of elements denotes the status of the corresponding packet, a status of free or active.

There are 4 operations for the packet pool:

- Initialize.
- get packet.

- free packet.
- idle.

The initialize function must be called before the packet pool can be used. It initializes all packet structures payload data pointers with the data arrays allocated. If any other operation is performed before the Initialize function is called, will result in error.

The get packet function returns a free packet from the packet pool. The function iterates the array until a free packet is found, changes the flag to active and returns the address of the structure. If there isn't a free packet it returns NULL. The software module calling the get packet function should check for a NULL pointer. It takes the size needed for a parameter.

The free packet function returns a packet to the pool. It takes the address of the packet that it frees as a parameter, iterates the packet array and when the address is found, the status changed to free. If the address is not found in the array nothing happens.

Since this code is used in all subsystems that have different memory constraints, preprocessor statements are used to modify the number of packets and their data size.

The maximum size of a data payload is around 2k bytes. The maximum packet that can be transferred through COMMS is 210 bytes, for more than that the large transfer data service is used.

There are 3 groups of data sizes that can be distinguished:

- Telecommands that are up to 80 data bytes.
- Housekeeping packets that can reach the maximum of 210 bytes.
- Mass storage service operations that can reach to 2kbytes.

Since only COMMS and OBC is using the 2k data sizes and EPS has strict memory constraints, it was decided to have 2 types of packets: one normal with maximum size of 210 bytes and the extended that reaches 2k bytes of data payload. For simplification 2 types are used instead of 3.

A more elaborate scheme for the packet pool memory model was considered, with allocated data blocks of 210 bytes and for an extended data the packet would occupy the blocks required but it has 2 main disadvantages:

- Possible fragmentation would slow packet processing.
- The algorithm for handling of the block allocation would be more complex and would require more time for testing and development.

A note about thread safe implementation: the free function doesn't require mutex protection because each packet can be freed once and the operation is atomic. The get packet was a candidate but there isn't a problem for all subsystems since the function is not used in ISRs and on the OBC the function is only used in the serial task.

Listing 0-2 packet pool module functions

```
tc_tm_pkt * get_pkt(uint16_t size);

SAT_returnState free_pkt(tc_tm_pkt *pkt);

uint8_t is_free_pkt(tc_tm_pkt *pkt);

SAT_returnState pkt_pool_INIT();

void pkt_pool_IDLE(uint32_t tmp_time);
```

QUEUES

When a packet is about to be shipped in another subsystem, the pointer of the packet structure is pushed in a queue according to the destined subsystem. The OBC has 3 queues responding to to each subsystem the OBC connects. The other subsystems have only 1 queue since they connect only to the OBC. The export packet function checks if the queue is not empty and if the UART peripheral is available and if that's the case, it pops the structure's pointer and then sends it.

The queue functions are defined in the core folder. The queue module functions are the basic used in queues: with push a packet structure is added in the queue, pop gets a packet from the queue if it has one, size returns the number of packets in the queue and the peak gets a packet with removing it from the queue and finally the idle was developed for fault tolerance but it wasn't used.

Listing 0-3 Queue module functions

```
SAT_returnState queuePush(tc_tm_pkt *pkt, TC_TM_app_id app_id);

tc_tm_pkt * queuePop(TC_TM_app_id app_id);

uint8_t queueSize(TC_TM_app_id app_id);

tc_tm_pkt * queuePeak(TC_TM_app_id app_id);

void queue_IDLE(TC_TM_app_id app_id);
```

Peripheral modes

In this section, the operation modes of the microcontroller's peripheral will be discussed while focusing in the UART. Peripherals like SPI and I^2^C have similar functionality. The UART and most of the STM32F4's peripherals have 3 modes of operation:

- blocking.
- interrupt.
- DMA.

The first mode is very simple, the function constantly checks a flag, probably a SFR bit. When the bit changes state, it signals an event like a new character is received or a character has finished transmission.

The interrupt mode uses the built-in hardware functions in order to free the CPU from constantly checking a SFR. When an event happens, the CPU stop the normal operation and jumps to an ISR function that handles the event. That way CPU cycles are only used for the processing of the event.

Finally, when the DMA mode is used, the events are processed without the use of the CPU. In particular usually a buffer address is configured along with the size of the data. During receive the buffer is filled automatically and in transmission the data are taken from the buffer. When the transmission is finished or when the data received are reached the size defined an ISR is triggered signalling the events end.

The blocking mode has simpler operation but it wastes CPU cycles for constantly checking the SFR. This is great reduced with the interrupt mode but there is some overhead. The DMA mode is the most efficient mode of operation with no overhead but while the other modes don't have a restriction on the number of characters, the DMA is only efficient when used with a predefined fixed number of characters.

The Standard Peripheral provides 3 set of functions: blocking, interrupt and DMA. The only difference in the provided functions is in the blocking mode where a timeout parameter is added. It is worth noting that in all 3 different types of microcontrollers used in CubeSat have the almost the same implementation. This simplifies and reduces the time needed for the implementation of the ECSS handling.

DMA mode was used for the reception and the transmission.

ECSS services

The services folder contains all the code related to the services implemented in the ECSS document. Each service is implemented in a module that is uncoupled with each other; The services module contains all the configuration parameters for the services. The service utilities

contain functions that are helpful for the ECSS services. If a service need information that is specific to a subsystem it uses the function that is defined in the platform folder in the respective file e.g., housekeeping for the housekeeping service. The subsystem-id file has all the application ids of CubeSat.

A single-entry point for all incoming packets is used for a better modular design and is denoted by the -app. Each function that belongs to a module start with the service name e.g., mass-storage-app.

CubeSat module

The CubeSat module has a collection of functions that are used from all subsystems together with with the ECSS services.

Listing 0-4 CubeSat module functions

```
SAT_returnState import_pkt(TC_TM_app_id app_id, struct uart_data *data);

SAT_returnState export_pkt(TC_TM_app_id app_id, struct uart_data *data);

SAT_returnState test_crt_heartbeat(tc_tm_pkt **pkt);

SAT_returnState firewall(tc_tm_pkt *pkt);
```

The import and export packet functions are used for processing incoming packets and transmitting packets through the UART to their corresponding subsystem.

The import and export packet function takes the application ID that the function operates for and the respectively data that are packed into the uart-data structure.

The import packet functions check if there is a new packet, it is deframed from HLDCL encapsulation, unpacket into a packet taken from the packet pool and if those process are finished without error the packet is routed into the respectively services handler or forwarded to another subsystem. The verification service handler is called then and finally if the packet is for that subsystem it is returned to the packet pool since all processing is finished. If the packet is for another subsystem the packet is freed when the export packet function finishes.

The export packet function checks if the UART peripheral is not transmitting another packet which in that case isn't anything more to do until the packet is finished transmitting. After that it checks if there is any packet in the queue, if there is the packet is packed from the packet structure into a temporary buffer array, HLDLC encapsulation, transmit ion from the UART and finally the packet is returned to the packet pool.

At first the export packet was called directly when a new packet was created and needed to forward it to its destination. That created 2 issues: First in the case the export packet was used inside the import packet e.g., for a telecommand response, the import had to wait for the export function to send it. If the UART was already used from another packet, it had to wait for that operation to finish as well. That could lead in packet loss if a new packet arrived while the import function was in use. Also, it created unwanted coupling between the import and export module. Secondly for the OBC which is multithreaded, it created race conditions.

The solution of this problem was the use of queues. Each time a new packet is generated the pointer to the structure is pushed in the queue of the destined subsystem. When the export packet is called, the packet pointer is retrieved from the the queue. This uncouples the 2 functions. The race conditions are solved by the way that the export function is called from only one place, so the pop function of the queue is atomic. The push function which can be used from all the threads is solved through the use of queues.

Listing 0-5 import function

```
SAT_returnState import_pkt(TC_TM_app_id app_id, struct uart_data *data)
        {

    tc_tm_pkt *pkt;
    uint16_t size = 0;

    SAT_returnState res;
    SAT_returnState res_deframe;

    res = recieve_packet(app_id,data);

    if( res == SATR_EOT ) {

        size = data->uart_size;

        res_deframe = HLDLC_deframe(data->uart_unpkt_buf, data->deframed_buf, &size);

        if(res_deframe == SATR_EOT) {

            pkt = get_pkt(size);

            if((!C_ASSERT(pkt != NULL)) == true) { return SATR_ERROR; }

            if((res = unpack_pkt(data->deframed_buf, pkt, size)) == SATR_OK) {
                stats_inbound(pkt->type, pkt->app_id, pkt->dest_id, pkt);
                route_pkt(pkt); }
            else {
                stats_dropped_upack();
```

```
            pkt->verification_state = res;
             }

        verification_app(pkt);

        TC_TM_app_id dest = 0;

        if(pkt->type == TC)         { dest = pkt->app_id; }
        else if(pkt->type == TM)    { dest = pkt->dest_id; }

        if(dest == SYSTEM_APP_ID) {
            free_pkt(pkt);
        }
    }
    else {
        stats_dropped_hldlc();
    }
  }

  return SATR_OK;

 }
```

Listing 4.12 export function

```
SAT_returnState export_pkt(TC_TM_app_id app_id, struct uart_data *data) {

    tc_tm_pkt *pkt = 0;
    uint16_t size = 0;
    SAT_returnState res = SATR_ERROR;

    /* Checks if the tx is busy */
    if((res = uart_tx_check(app_id)) == SATR_ALREADY_SERVICING) { return res; }

    /* Checks if that the pkt that was transmitted is still in the queue */
    if((pkt = queuePop(app_id)) ==  NULL) { return SATR_OK; }

    stats_outbound(pkt->type, pkt->app_id, pkt->dest_id, pkt);

    pack_pkt(data->uart_pkted_buf,  pkt, &size);

    res = HLDLC_frame(data->uart_pkted_buf, data->framed_buf, &size);
    if(res == SATR_ERROR) { return SATR_ERROR; }

    if((!C_ASSERT(size > 0)) == true) { return SATR_ERROR; }
```

```
    send_packet(app_id, data->framed_buf, size);

    free_pkt(pkt);

    return SATR_OK;
}
```

Service module

The service module provides most of the information related to the ECSS services. Here all significant definitions of the ECSS specification used by software are found: services types and subtypes names are defined, the SAT-returnState that holds all states of the software, supporting type definitions of each service, definitions, packet structure and assertion macro definition along with configuration definitions like the maximum size of a packet.

The supporting type definitions of each service could had been define in each service module and maybe that design was better in terms of software separation design but it was decided to be in the service module in one place as a way for the developer or used to quickly find information about the service inputs. Moreover, a global definition file that holds the most important definitions would save the developer from searching in multiple files for key parameters.

Error codes

The ECSS module are using one enumeration for status codes that could happen during the process of a packet. The status codes are used for debugging, logging and in the verification service in the case of a telecommand failure. The enumeration plays a key part in error handling, since all functions in the ECSS module return the enumeration. The design intend was to have 1 status code for all the failures in order to easily identify the source of error, this happened in the most part but unfortunately not in the extend that was wished for.

There are status codes used for different modules. The first 6 (0-5) are defined in the ECSS standard. The 6th OK means that everything happened as planned. ERROR provides a generic name for failure. Scheduling service uses the 16-29 codes. The status codes of the FatFS are shifted between number 30-50. The final 5 provide specific errors.

ECSS packet structure

The ECSS packet structure holds packets in that form when they are process, this structure plays the prime role since most all functionality in the ECSS module revolves to a packet. Most of the

field of the structure are named after the counterparts in the ECSS specification and hold the same amount of information in bytes. The only difference is the source or destination ID which the name is different according the type of the packet but it was decided to both share the same field and the structure member was named dest-id from destination ID since most of the times that it was used in the code was when the variable held the destination ID and it was easier to remember.

The only structure member that doesn't belong to the ECSS specification is the verification-state that is a supporting variable for the verification service and holds the verification state that the packet is in. If the state was in another variable it would only make the code more complicated.

Listing 4.13 Assertion preprocessor macro definition and calling example

---

```
#define C_ASSERT(e) ((e) ? (true) : (tst_debugging(__FILE_ID__, __LINE__, #e)))

if(!C_ASSERT(*size <= UART_BUF_SIZE) == true) { return SATR_ERROR; }
```

Listing 4.14 ECSS module packet structure definition

```
typedef struct {

/* packet id */

//uint8_t ver; /* 3 bits, should be equal to 0 */

//uint8_t data_field_hdr; /* 1 bit, data_field_hdr exists in data = 1 */

TC_TM_app_id app_id; /* TM: app id = 0 for time packets, = 0xff for idle packets. should be 11 bits only 8 are used though */

uint8_t type; /* 1 bit, tm = 0, tc = 1 */

/* packet sequence control */

uint8_t seq_flags; /* 3 bits, definition in TC_SEQ_xPACKET */

uint16_t seq_count; /* 14 bits, packet counter, should be unique for each app id */

uint16_t len; /* 16 bits, C = (Number of octets in packet data field) - 1, on struct is the size of data without the headers. on array is with the headers */

uint8_t ack; /* 4 bits, definition in TC_ACK_xxxx 0 if its a TM */

uint8_t ser_type; /* 8 bit, service type */
```

uint8_t ser_subtype; /* 8 bit, service subtype */

/*optional*/

//uint8_t pckt_sub_cnt; /* 8 bits*/

TC_TM_app_id dest_id; /*on TC is the source id, on TM its the destination id*/

uint8_t *data; /* pkt data */

/*this is not part of the header. it is used from the software and the verification service,

*when the packet wants ACK.

*the type is SAT_returnState and it either stores R_OK or has the error code (failure reason).

*it is initialized as R_ERROR and the service should be responsible to make it R_OK or put the corresponding error.

*/

SAT_returnState verification_state;

}tc_tm_pkt;

Assertions

Used directly from the JPL's 10 rules, assertions are used whenever possible. It is defined as a preprocessor macro definition that calls the test-debugging function.

Most checks are for NULL pointers and wrong ranges in parameters. The checks that are using assertions are only for parameters that are invalid and denote wrong behavior. There are also used for fault containment.

Assertions are defined as a preprocessor macro, when the e expression is false the tst-debugging is called which takes the filename, the line in the file that the assertions is placed and finally the expression. These parameters help to identify where the error happened. The assertion code is taken directly from the JPL 10 rules.

Service utilities module

The service utilities module contains a collection of functions that complementary to the use of the ECSS modules.

The cnv functions converts from an uint8-t to the corresponding type (16

- 32 bits) and vice versa. Using that functions all subsystem share the same endianness type. It was designed for packet conversion from the structure to the 8-bit array used for transmission and from the raw 8-bit array to the packet structure.

There are 2 ways for converting a variable in C language to 8 bits and vice versa:

- Using unions.
- Using shifts and bit masking operations.

The union approach has the advantage of better code clarity.

The checksum is used for calculating the error checking byte of the ECSS packet. It is used when the packet is received for error checking and for calculating it when its for transmission. The checksum is a simple XOR based algorithm.

The STM32 that is used in all subsystems have a hardware peripheral for calculating the checksum but a software implementation was preferred even though the hardware peripheral would be more efficient. This was primary for fault tolerance issues, in the case the hardware was destroyed from the space radiation, it would render all subsystem communications useless since the checksum would fail. If there is a hardware failure in the ALU of the microcontroller that would be used from the software checksum, all operations on the microcontroller would fail, disabling the microcontroller.

The sys-data-init is used in initialization and initializes the state of the ECSS packet sequence counters for all the application IDs.

The create packet functions crt-pkt initializes a packet structure.

The unpack packet function gets a packet in a raw 8-bit array and it fills a packet structure while making the necessary checks. The pack packet function gets the information in a packet structure and fills an 8-bit array in order to transmit the data through the UART.

Listing 4.15 Service utilities module functions

```
SAT_returnState checkSum(const uint8_t *data, const uint16_t size, uint8_t *res_crc);

SAT_returnState unpack_pkt(const uint8_t *buf, tc_tm_pkt *pkt, const uint16_t size);

SAT_returnState pack_pkt(uint8_t *buf, tc_tm_pkt *pkt, uint16_t *size);

SAT_returnState crt_pkt(tc_tm_pkt *pkt, TC_TM_app_id app_id, uint8_t type, uint8_t ack,
uint8_t ser_type, uint8_t ser_subtype, TC_TM_app_id dest_id);

void cnv32_8(const uint32_t from, uint8_t *to);

void cnv16_8(const uint16_t from, uint8_t *to);
```

```
void cnv8_32(uint8_t *from, uint32_t *to);

void cnv8_16(uint8_t *from, uint16_t *to);

void cnv8_16LE(uint8_t *from, uint16_t *to);

void cnvF_8(const float from, uint8_t *to);

void cnv8_F(uint8_t *from, float *to);

void cnvD_8(const double from, uint8_t *to);

void cnv8_D(uint8_t *from, double *to);
```

TEST SERVICE MODULE

The implementation of the test service module is very simple, making the perfect candidate for a more through code examination. When a telecommand is received with (17,1) it replies with a telemetry packet of (17,2) with no data.

The test-app function serves as an entry point for the packet in the route. Most of the function's code has to do with checks. The test-crt-pkt is a complimentary function

Listing 4.16 Test service module functions

```
SAT_returnState test_app(tc_tm_pkt *pkt) {

tc_tm_pkt *temp_pkt = 0;

if(!C_ASSERT(pkt != NULL && pkt->data != NULL) == true) { return SATR_ERROR; }

if(!C_ASSERT(pkt->ser_subtype == TC_CT_PERFORM_TEST) == true) { return
SATR_ERROR; }

test_crt_pkt(&temp_pkt, pkt->dest_id);

if(!C_ASSERT(temp_pkt != NULL) == true) { return SATR_ERROR; }

route_pkt(temp_pkt);

return SATR_OK;

}

SAT_returnState test_crt_pkt(tc_tm_pkt **pkt, TC_TM_app_id dest_id) {

*pkt = get_pkt(PKT_NORMAL);
```

```
if(!C_ASSERT(*pkt != NULL) == true) { return SATR_ERROR; }

crt_pkt(*pkt, SYSTEM_APP_ID, TM, TC_ACK_NO, TC_TEST_SERVICE,
TM_CT_REPORT_TEST, dest_id);

(*pkt)->len = 0;

return SATR_OK;

}
```

Telecommand verification service module

The telecommand verification service allows to see the outcome of a telecommand operation. If the operation is critical or the operator wants confirmation the acknowledgement flags are set in the packet header.

The implementation in order to be simple and efficient was straightforward. The verification state was added in the packet structure. There each service is responsible to place the state of the operation. During the packet's allocation from the packet pool the state is set to initialized, in order to differentiate from other states.

After the processing of the telecommand is finished, the verification-app is called. It checks the acknowledgment flags in the telecommand frame and if a verification flag exists, it sends the corresponding telemetry packet. If the verification state in the telecommand flag indicates a failure, the verification state is used for the failure reason.

Moreover, the use of the verification state could be used to store more information for the status of the packet. The use of preprocessor macros that automatically add an error or a general state in the process could simplify the development.

During the implementation, the first design of export packet made the inline processing of the verification and the implementation of extra stages impossible without adding overhead that could interrupt the primary process. With the later addition of queues, the design could have been simpler. One different design would have a table with all packets needing verification and the verification service module residing in a different or in the idle task, checking for a change in the state. That way more verification states could have been used.

The implementation follows more or less the same scheme as the test service module.

Event reporting service module

Table 5.24 Event service frame

Subsystem ID Event ID Event time Data	8 bits 8 bits 32 bits 32 bits

Since the only storage medium is on the OBC, there wasn't a way to store event logs in other subsystems. For that reason, it was decided that all events are to be sent to the OBC for storage. There is always the issue that some events won't be stored for various reasons like power resets or packet loss but it is better to have more information and lose some than the opposite. An implication of using the OBC for event storage is that only the most critical events are send in order not to create too much traffic for the OBC to handle.

At first the UART was used for displaying debug messages. After the subsystems integration and since only ECSS packets could be forwarded to the umbilical UART, the debug messages were encapsulated in event service packets. The ASCII messages were used only for early debugging and there were replaced later.

The next design had a fixed packet format. For the implementation, each event had it's own function and the subsystem's developer was responsible for defining the subsystem's events.

For design simplicity, the event-app had a pre-processor ifdef that checked the subsystem and if it was the OBC, it stored the event, in all subsystems it forwarded the packet to the OBC.

Listing 4.17 Event service module boot event example function

---

```
SAT_returnState event_boot(const uint8_t reset_source, const uint32_t boot_counter) {

tc_tm_pkt *temp_pkt = 0;

if(event_crt_pkt(&temp_pkt, EV_sys_boot) != SATR_OK) { return SATR_ERROR; }

temp_pkt->data[10] = reset_source;

cnv32_8(boot_counter, &(temp_pkt->data[11]));

for(uint8_t i = 15; i < EV_DATA_SIZE; i++) { temp_pkt->data[i] = 0; }

if(SYSTEM_APP_ID == OBC_APP_ID) {

event_log(temp_pkt->data, EV_DATA_SIZE);

} else {

route_pkt(temp_pkt);

}

return SATR_OK; }
```

Housekeeping & diagnostic data reporting service module

The most common task for a cubesat is housekeeping: in regular intervals data from each subsystem is gathered, stored and transmitted to earth. The data provide an insight to the state of the cubesat. For CubeSat there are 2 housekeeping functions, WOD and extended WOD.

The mechanism for WOD formation is that the OBC sends a telecommand request in each subsystem with the structure ID, the subsystems respond with the data, the OBC gathers all the data, stores it and forwards it to the COMMS subsystem, in order to transmit them to Earth. Each structure ID and application ID form a unique set of data.

There are 2 distinct functions of the housekeeping service:

- Requesting and retrieving data.
- Storing housekeeping data (OBC).

Since each subsystem has its own set of data related to different structure IDs, the housekeeping service calls the subsystem dependent functions that are in the platform folder. The functions in the module are used only for checking input parameters and calling the functions in the platform folder.

OBC Housekeeping

OBC sends the WOD and extended WOD in 1 minute interval. First it sends the requests to EPS and COMMS with 1 second delay between them and with the health report structure ID. When a response arrives, the data are temporary stored in memory. If a response doesn't come the respective data are left with 0 value. A mechanism for re-requesting the data wasn't implemented. After 29 seconds OBC forms the WOD report with the responses from the subsystems, stores it and it sends it to COMMS for transmission. After the WOD comes the extended WOD. The same mechanism is used, only this time requests are send to all 3 subsystems and the structure ID is extended health.

Since the WOD requires 31 historic values resulting in data logged 31 minutes ago, the persistent RAM region of the OBC was used. The SRAM was used instead of the SD card because it minimizes the data access time and improves reliability since the delete function of the FatFS is not invoked. For storage, a circular buffer was implemented in the SRAM containing 32 sets of values.

Mass storage service module

The mass storage service was the most difficult and complex module in CubeSat. It has the most lines of code and functions in services.

There are 2 main reasons that shaped the design of the module: the restraints of data sizes send through the RF channel and the limitations from the design and documentation of the FatFS library and the FAT file system specification.

Logs come from various sources like housekeeping and events. There are implemented as a circular buffer so in the case of a overflow the newest log overwrites the oldest one. The logs are stored for downloading when there is link with the ground station. After the logs are downloaded, they should be deleted so in the next link, the operator doesn't re download them. Configuration files are files that are unique and they store values for parameters. Large unique files are the files that are larger than the normal data size used in mass storage and unique.

In CubeSat there are 4 specific types:

- Event logs.
- WOD logs.
- Extended WOD logs.
- Scheduling service configuration files.

It was decided that each type should have its own store ID and stores should be implemented with folders. Each store has different set of properties.

the normal and extended WOD. The logs share the same properties:

- Fixed size of records.
- Implemented as circular buffers.
- Unique data in entries.
- No need for modification after the log entry.
- Able to search, download and deleted.

Each log is implemented as a separate file. This happens mainly due to the FatFS and its limitations. The Filenames are implemented as numbers only, stored in ASCII number characters. With the addition of folders, it gives unique log entries. Numbers are used because it is easier and more efficient for processing them in the microcontroller than ASCII strings. e.g. the log 4916 occupies only 2 bytes in RAM as an integer in contrast of using 5 bytes as an ASCII string. Again, with integers file comparisons are a lot faster than as strings.

There was a though of using timestamps as the filenames of logs. The timestamp would had been the time of the log creation. The unique file name would have been achieved by using the correct time resolution. The reason that it wasn't used was that with 1 millisecond resolution and 8 characters for the timestamp, the timer would rollover in ~51 days, which was way lower than the mission specifications. When the timer rolled over the result would had been to have logs that would be impossible to figure if they were created before or after the roll over. Moreover, it was difficult to guarantee that all the times the OBC would had correct time.

The circular buffer mechanism was used because of the nature of the log operations. First of all, having the circular buffer ensures that the logs won't overrun and fill all the SD card. Moreover, since communication with the ground is limited, makes the downlink of a very large log number impossible.

LIFE OF A PACKET

After the analysis of each part, it is imperative to give the reader an overview of the work flow, the best way is by describing the life of an incoming packet.

There are 4 steps for processing an incoming packet:

I. First the packet is received byte per byte from the UART DMA. If the packet is encapsulated in a valid HLDLC frame, the packet is stored from the DMA to a temporary buffer.

II. When the import function is called, the packet goes through the HLDLC deframe, a new packet is allocated from the packet pool and then the ecss-unpack gets the packet from the array, performs all necessary checks and places it in the packet structure. If the unpack receives a valid packet, then the route function is called. If there was a failure the verification service is called. Finally, if the packet was destined for that subsystem the packet's state returns to free and it is available for reuse from the packet pool.

III. Route directs the packet to the correct service or the correct queue if the packet is indented for another subsystem. It is highly probable that the service used, generates a response, then the route is called again and the packet is placed in the correct queue for transmission. If there was an error in the route the packet is freed and the function returns the error.

IV. The verification module checks if the incoming packet needed an acknowledgement and if that's the case, if the packet has all the necessary information the response is routed back.

The life of an outgoing packet is a lot simpler. The export function checks the queue for packet, if the UART is available, the packet is popped from the queue, it is packed to an 8-bit array from the packet structure, then is encapsulated in a HLDLC frame and finally is transmitted.

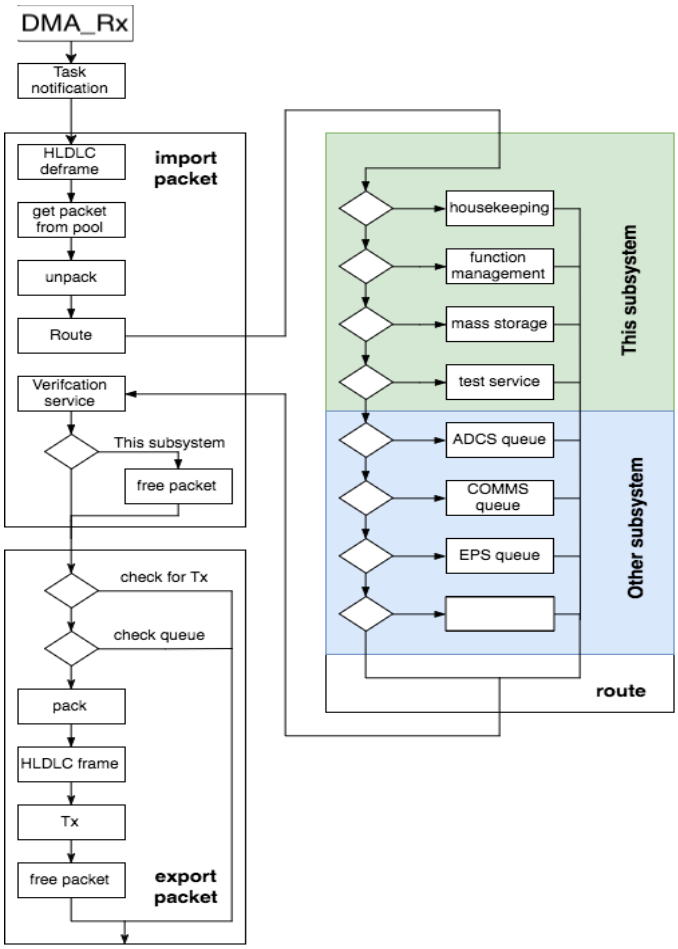


Figure 5.7: The life of a packet

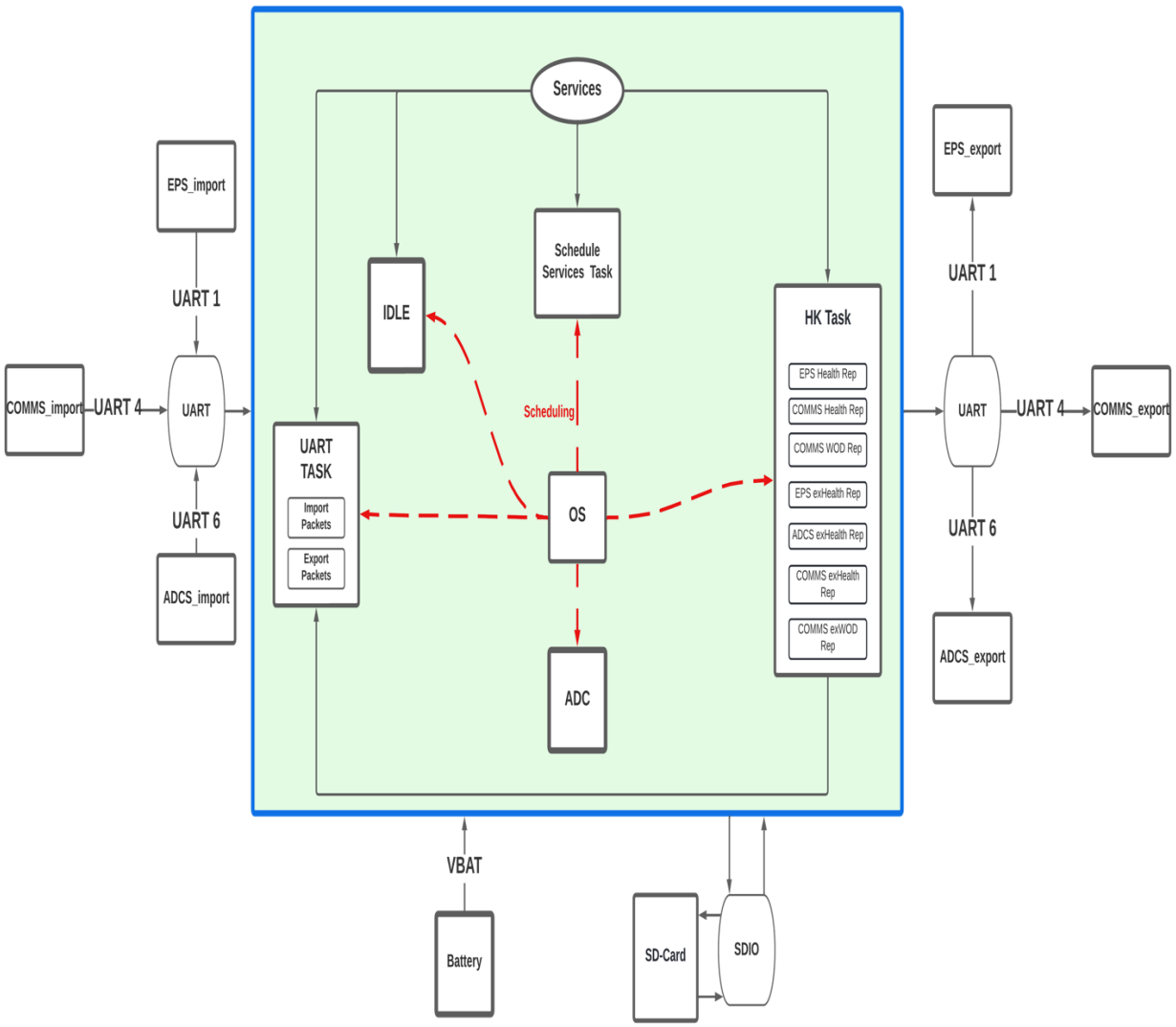
7- On-board computer Software

In this section, the software implementation for the OBC will be discussed.

Operating System

We do this using Time Triggered‘ (TT) architectures that incorporate patented run-time monitoring techniques.

STATIC DESIGN:



There are 11 tasks:

- UART task.
- IDLE task
- Scheduling task.
- EPS Health Report task.
- EPS exHealth Report task.
- COMMS Health Report task.
- COMMS exHealth Report task.
- COMMS WOD Report task.

- COMMS EXT WOD Report task.
- ADCS exHealth Report task.
- ADC task.

UART task: It handles incoming and outgoing packets. This task is the most important since the first hard real-time requirement is to handle as soon as possible incoming packets so there aren't any packet losses.

IDLE Task

- Check the sanity of the packet pool.
- Check the sanity of the queues.

ADC Task: Start and get the results of the A/D converter that is connected to the OBC battery.

EPS Health Report task, EPS exHealth Report task, COMMS Health Report task, COMMS exHealth Report task, COMMS WOD Report task, COMMS EXT WOD Report task and ADCS exHealth Report task are Housekeeping tasks handle all the housekeeping activities. First the housekeeping service module initialization is called which sets the buffer that is allocated for the reserved outgoing housekeeping packet. Then the task calls continuously the hk-sch function which sends housekeeping requests to the subsystems that form the normal and extended WOD. After each packet request is put in the queues , the UART task is notified and switched in order to send the requests.

Scheduling Task:

First, the scheduling service init function is called which loads the scheduling packets stored in the SD-card.

Then the update function takes the stored time-tagged commands(packets) that have been loaded from ground and release them to their destination application process(es) when their on-board time is reached.

Housekeeping Tasks:

First, the housekeeping service module initialization is called which sets the buffer that is allocated for the reserved outgoing housekeeping packet. Then the tasks are called periodically, sending housekeeping requests to the subsystems that form the normal and extended WOD. After each packet request is put in the queues, the UART task is notified and switched in order to send the requests.

EPS Health Request task, COMMS Health Request task: Create the health request packet and route the packet to the subsystem's queue.

COMMS WOD Report task: Take the health report parameters received from the subsystems and create a packet, then store it on the SD-card and route it to COMMS queue.

EPS exHealth Request task, COMMS exHealth Request task, ADCS exHealth Request task: Create the extended health request packet and route the packet to the subsystem's queue.

COMMS EXT WOD Report task: Take the extended health report parameters received from the subsystems and create a packet, then store it on the sdcard and route it to COMMS queue.

Real time clock

The OBC has a coin li-on battery that supplies the RTC peripheral and 4kbytes of the SRAM when the main power is off. The RTC is part of the STM43F4 microcontroller but runs independently.

The RTC peripheral is used for storing the time without the need for the microcontroller running. It has a time drift that is larger than what the mission profile requires. For that reason, the time needs to be regular updated either from the ground station or from the GPS in the ADCS subsystem. Using the RTC simplifies the design.

The battery also powers a 4K bytes portion of the SRAM. This part stores configuration parameters of the OBC. Having the battery, the data remain after a reset.

This data is not stored in the SD or the flash memory because storing the data would take more time, that could lead to normal operation interaction. Moreover, continuous writing of the data could lead to file system corruption.

The event buffer was intended to be temporary stored in this portion of the memory and write the events in the SD when there was enough data, during the idle time of the CPU.

Dynamic design

Table 5.25 OBC Dynamic Design

Task Group	Task	Offset (tick)	Period (mS)	Period (tick)
TTRD-19A related tasks	Watchdog Update	0	10	1
	HEARTBEAT_SW_Update	0	1000	100
	ADC1_Update	0	500	50
	PROCESSOR_TASK_Update	0	1000	100
Services Tasks	EPS_Health_Rep_task_Update	1	200	20
	EPS_exHealth_Rep_task_Update	4	200	20
	ADCS_exHealth_Rep_task_Update	5	200	20
	COMMS_Health_Rep_task_Update	2	200	20
	COMMS_WOD_Rep_task_Update	3	200	20
	COMMS_exHealth_Rep_task_Update	6	200	20
	COMMS_EXT_WOD_Rep_task_Update	7	200	20
	SCHEDULE_SERVICES_Update	2	1000	100
Vbat Task	ADC1_vbat_Update	5	200	20
IDLE Task	IDLE_Update	1	1000	100
OBC UART Task	UART_Update	0	100	1

SECURE DIGITAL INPUT/OUTPUT INTERFACE (SDIO)

SDIO is an interface designed as an extension for the existing SD card standard, to allow connecting different peripherals to the host with the standard SD controller.

It provides an interface between the APB2 peripheral bus and MultiMediaCards (MMCs), SD memory cards, SDIO cards and CE-ATA devices.

SDIO FEATURES

- Data transfer up to 50 MHz for the 8 bit mode
- Data and command output enable signals to control external bidirectional drivers.
- Allows card to interrupt host.
- Operational Voltage range: 2.7-3.6V

SDIO FUNCTIONAL DESCRIPTION

The SDIO consists of two parts:

- The SDIO adapter block provides all functions specific to the MMC/SD/SD I/O card such as the clock generation unit, command and data transfer.
- The APB2 interface accesses the SDIO adapter registers, and generates interrupt and DMA request signals.

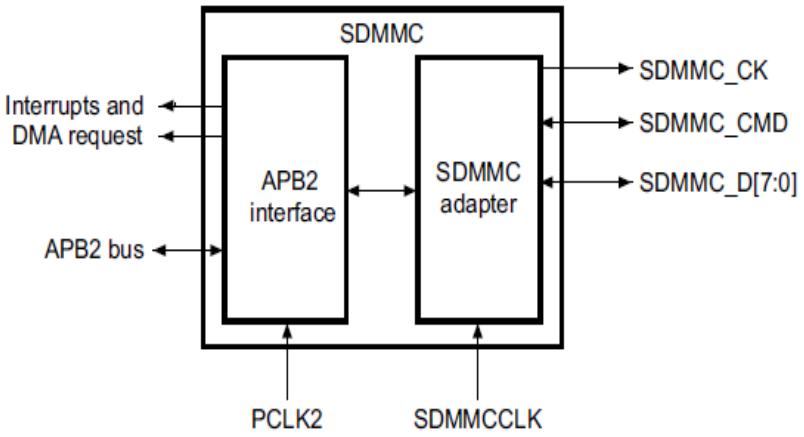


Figure 5.8: SDIO block diagram