Analysis of a Satellite Platform Based on AMSAT V.1.3.2 for the Study of Sustainable Mobility.

FINAL THESIS PROJECT

Submitted in fulfillment of the requirements for the Degree in Telecommunication Systems Engineering

by

Juan Lacosta Arpide

Under the supervision of

Daniel Valderas

Donostia-San Sebastián, March 2025

Tecnun Universidad de Navarra | ESCUELA DE INGENIERÍA INGENIARITZA ESKOLA SCHOOL OF ENGINEERING

Tecnun Universidad de Navarra | ESCUELA DE INGENIERÍA INGENIARITZA ESKOLA SCHOOL OF ENGINEERING

Final Thesis Project

TELECOMMUNICATION SYSTEMS ENGINEERING

Analysis of a Satellite Platform Based on AMSAT V.1.3.2 for the Study of Sustainable Mobility.

Juan Lacosta Arpide

Donostia-San Sebastián, April de 2025

Pº Manuel Lardizabal, 13. 20018 Donostia-San Sebastián, Gipuzkoa Tel. 943 219 877 · Fax 943 311 442 · www.tecnun.es

ABSTRACT

In recent years, CubeSats have shown great potential for urban monitoring, offering a new way to collect and analyze data in real-time. Traditional methods for studying how people and vehicles move around from the city to work (such as ground-based sensors or surveys) can be expensive and limited in how much area they can cover. This project takes a fresh approach by exploring the use of CubeSats to monitor pedestrian and vehicle flow in busy industrial and commercial areas, with the goal of improving sustainable mobility.

The project consists of three parts: Creating a CubeSat capable of reading the information from sensor, then transmitting it to a base station computer and finally processing the information for further analysis. A CubeSatSim, launched via drone, collects important telemetry data like temperature, humidity, altitude, and pressure. This information is then transmitted and analyzed via radiofrequency to better understand commuting patterns. By studying how different weather conditions affect transportation choices, the project aims to help improve public transportation schedules and encourage more sustainable commuting options.

While the project has successfully demonstrated the ability to collect and send data, there are still challenges to overcome, particularly with the CubeSat's communication range, with limitations with large distances. Moving forward, the plan is to improve the system by upgrading antennas for stronger signals, adding an amplifier for the transmitter signal, implementing a GPS module for more accurate location tracking, and incorporating image processing to analyze traffic visually. These improvements will help make CubeSats a powerful tool in smart mobility planning, offering valuable data for creating more sustainable and efficient transportation systems in cities.

INDEX OF CONTENTS

INDEX OF FIGURES

INDEX OF TABLES

1. INTRODUCTION AND OBJECTIVES

1.1. Background

CubeSats are small modular cube satellites with 10 cm sides and with a mass up to 1.33 kg. They were developed in 1999 by the Cal Poly (California Polytechnic State University) and the Stanford University with the goal of enabling universities worldwide to develop and execute space science research projects, which can be built from an office desk. It helped universities, colleges and small enterprises to experiment with aerospace systems and technology at a very low cost. Some of its functionalities are[1] earth remote sensing, space exploration, and rural connectivity, among others.

This new and cheap space technology has been revolutionary. By 2004 CubeSats could be built and launched into space at a cost of approximately $65.000-80.000 and by 2008 approximately 75 CubeSats were already orbiting the earth. This drastically reduced price, compared to traditional satellite launches, established CubeSats as an affordable and viable option for academic use. The lower cost also allowed developers to introduce new technologies into space through the CubeSat, as such risk were out of reach due to their expensive cost.

Firstly, CubeSats were designed with basic functionalities, however, the latest models carry high quality sensors, cameras and can have extremely large communication links. Furthermore, some problems such as noise (SNR), which were significant at the beginning, now with new implementations have been greatly reduced resulting in better transmission, reception and processing of data. Actually, thanks to its design the whole process of building, testing and launching a CubeSat into the space can be done in a period of just 6-12 months[2]. Since launching an object into space is very expensive, important primary missions are used to include secondary payloads, such as CubeSats, without increasing fuel consumption or complicating the journey. This significantly reduces the launch cost.

In 2018 the Mars Cube One (MarCO)[3], was the first mission along with NASA's InSight Mars Lander to create a CubeSat capable of operating beyond Earth's orbit on interplanetary missions. The mission of this CubeSat was to provide real-time data transmission while the InSight lander performs the entry, descend and landing phase. This will provide unique data outside the Earth's orbit at a faster rate, instead of waiting for several hours. Later on, more missions have been made that consisted of launching several CubeSats at the same time, each one with different functions in order to obtain more information from space and carry out different objectives. Nowadays, some enterprises such as Planet Labs, OneWeb, Spire Global, SatRevolution and even NASA have already launched a lot of CubeSats into the space for their own interests.

[1] https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9079470
[2] https://www.infoespacial.com/texto-diario/mostrar/3572586/cubesats
[3] https://es.wikipedia.org/wiki/Mars_Cube_One#:~:text=Mars%20Cube%20One%20(o%20MarCO,Mars%20Lander%20de%20la%20NASA.

1.2. AMSAT

AMSAT[4] is a group of Amateur Radio Operators with an interest in building, launching and then communicating with each other through non-commercial Amateur Radio Satellites such as CubeSats. It has had a huge impact on space-based amateur radio.

It was founded in 1969 by a group of amateur radio enthusiasts whose goal was to create satellites capable of communicating via space-based platforms. "OSCAR 1"[5] (Orbiting Satellite Carrying Amateur Radio) launched in 1961, is the first well-known satellite developed by a group of volunteers (then called AMSAT). This satellite was a rectangular box of 30 x 25 x 12 cm and weight 10 kg and was launched as a secondary payload. Its purpose was to demonstrate that satellites could be used for amateur radio communication. Then, AMSAT continued building satellites following OSCAR's characteristics. As technology and knowledge of these nano satellites advanced, AMSAT's satellites became more complex and offered better communication capabilities.

As CubeSats were developed in the late 1990s, on the 2000s, AMSAT began to investigate by adapting these smaller, cost-effective satellites for their missions, which marked a turning point for them and gained popularity. This led the enterprise to launch even more cost-effective and accessible satellites for amateur operators. Modern AMSAT satellites include technologies such as software-defined radio (SDR), which allows frequency and communication modes to be adjusted in real time. This enables amateur operators to use AMSAT satellites in a flexible and effective way.

1.3. Objectives

The main goal of this project is to study and characterize an embedded system, focusing on key aspects of Telecommunications Engineering. This includes hardware design, software programming, signal transmission, and data processing. This project also lays the foundation for future developments in aerospace telecommunications, IoT-based monitoring, and sustainable mobility solutions. It highlights how CubeSat technology can be useful beyond traditional space applications.

A key objective is to test whether a CubeSat can effectively monitor pedestrian and vehicle flow in industrial and commercial areas. Maintaining a stable communication link between the CubeSat and the ground station is essential. If successful, this system could provide real-time telemetry and visual data. Urban planners and policymakers could then use this information to optimize traffic flow, reduce congestion, and promote eco-friendly commuting options.

As a feasible application, a sustainable mobility mission is designed. It analyzes transportation patterns in industrial areas with high commuter traffic, such as Miramon. A CubeSat will collect telemetry data, including temperature, humidity,

[4] https://www.amsat.org/amsat-history/
[5] https://en.wikipedia.org/wiki/OSCAR_1

altitude, and pressure. By processing and analyzing this data, the mission aims to estimate the proportion of people using sustainable transportation methods (walking, cycling, or public transport) versus those relying on private vehicles.
The study will also assess how different weather conditions (sunny, cloudy, rainy, windy, and snowy days) influence commuting behavior. For example, if the data shows a significant increase in private vehicle use on rainy days while public transport usage remains stable, adaptive measures could be introduced to reduce emissions and traffic congestion. One possible solution would be to increase the frequency of public transport services during bad weather, encouraging commuters to choose more sustainable alternatives.

To achieve this, a CubeSat will be developed with the capability to read and transmit reliable telemetry data to a base station via radiofrequency. A key component of this project is the study of the maximum transmission distance between the CubeSat and the base station. This is a crucial aspect of evaluating the feasibility of this sustainable mobility mission. As this is an academic project designed to lay the groundwork for future advancements, several elements are planned for later development. These include deploying the CubeSat on a drone simulating a real situation for the mission, implementing a GPS module and camera for a more complete data analysis and implementing Mioty, an efficient communication protocol for uplink transmission. This would significantly improve the CubeSat's ability to send critical data over long distances and could be a perfect project for students to gain knowledge about telecommunication and electronic engineering.

Beyond its technical goals, this mission also opens the door to smart mobility applications. With access to real-time data from the CubeSat, commuters could make better-informed choices about how they move through the city, leading to more efficient and sustainable travel habits. At the same time, the system could give urban planners a clearer picture of how people are actually moving, offering solid data-driven insights, to help reduce traffic, lower fuel consumption, and cut down on emissions.

Throughout the report, the development of the CubeSat will be explained step by step across its main components. Section 2 focuses on the hardware and the process of physically building the CubeSat. In Section 3, the attention shifts to software installation and the code that runs on the system. Finally, Section 4 covers the data transmission between the CubeSat and the ground station, including an analysis of the maximum communication range, a key factor in evaluating the feasibility of the mission.

2. **HARDWARE**

The CubeSatSim consists of different pieces such as: three main boards, two raspberry pi (Zero and Pico), solar panels, antennas, batteries and other soldered components to the boards. This project is based on Alan Johnston's (Vice President - Educational Relations of AMSAT) CubeSatSim project. [6] The three main boards used were sent by Alan and the rest of the components were bought from different providers such as Digi-key and Amazon following Alan's bill of materials.[7]

In this section those components and pieces are going to be explained.

2.1.Main Boards

These three boards consist of: the STEM Payload Board, Battery board and Solar Board. The energy flow and functioning of the CubeSatSim is the following:
Firstly, the solar board generates and manages energy from sunlight. Then, this energy is regulated and used to charge the batteries of the second board, which purpose is to store and distribute it through the rest of the system. Last board, the STEM payload receives that energy from the battery board to operate sensors, read data, process it and send it to the base station. If the CubeSat neither reads data correctly nor manages power efficiently, the mission cannot be successfully completed.

So, to build the CubeSatSim, the three main boards were bought and received as shown in Figure 1.

Figure 1: Battery board, STEM payload board and solar board from left to right.

Then, the next step is to buy and weld the different components to each of the boards. Therefore, in the next subsection, the boards are going to be built.

[6] https://github.com/alanbjohnston/CubeSatSim/wiki
[7] https://cubesatsim.org/bom

2.1.1. Solar Board

The Solar Board, as explained before has a crucial role in the CubeSatSim. It's the board in charge of the energy management.

Energy Management:
The solar panels situated on the CubeSatSim sides convert sunlight into electricity, providing greater autonomy for long-duration missions. By collecting solar energy, the CubeSat can continue operating for extended periods without relying solely on limited battery power. The board features a regulation circuit which stabilizes the voltage and then charges the batteries of the battery board. Additionally, JST and QWIIC connectors make connections between boards and allow the flow of energy among all the boards.

In order to build the Solar Board, the following components are needed:

Component	Quantity	Location
IN5817 diode	6	Top
4.7k resistors	2	Top
20x2 female GPIO header non-stacking	1	Bottom
Blue INA219 High Side DC Current Sensor Breakout - 26V ±3.2A Max	6	Top/Bottom
Micro JST 2 pin connectors	14	Top
QWICC connector	1	Top
1x4 male breakaway header	1	Top
JST jumper cable	1	Top

Table 1 Solar Board components

After soldering the components to the board, this should look like this:

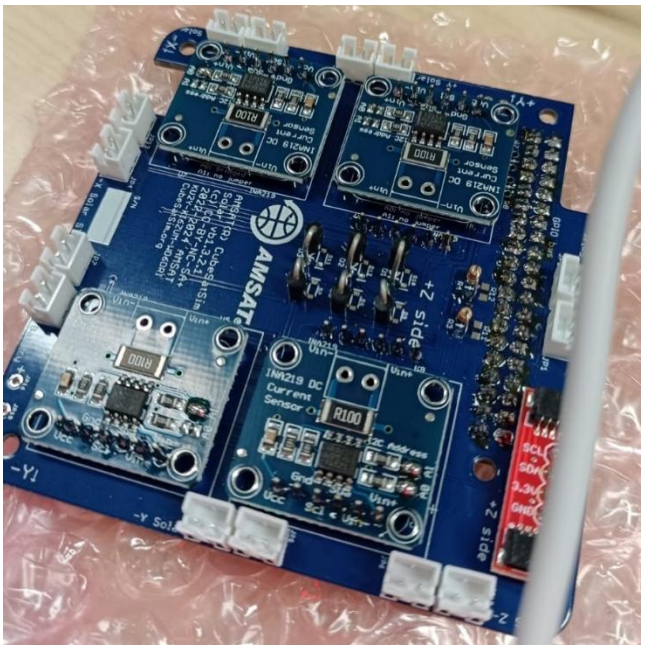


Figure 2: Solar Board completely built.

2.1.2. Battery Board

Storage energy management:
The purpose of the Battery Board is to store and manage the energy generated by the Solar Board. This board is crucial for optimizing the CubeSat's energy and therefore, its lifespan. It contains three rechargeable AA 2500mAh batteries that store this energy. When the CubeSat Sim's solar panels are unable to generate electricity (e.g., shadow or darkness), the stored energy in the batteries becomes the primary power source, ensuring continuous operation of the CubeSatSim.

During normal operation, the Battery Board continuously supplies power to the STEM Payload Board. This allows sensor data processing, communication tasks, and other mission operations. In turn, this energy is used to ensure the proper functioning of all CubeSatSim systems, including data transmission via radiofrequency. By effectively managing energy storage and distribution, the Battery Board plays a crucial role in maintaining the CubeSat Sim's functionality throughout all the time.

The components needed to solder are the following:

Component	Quantity	Location
1 cell AA battery holder 1024	1	Top
2 cell AA battery holder 1012	1	Top
Blue INA219 High Side DC Current Sensor Breakout - 26V ±3.2A Max	1	Bottom
GPIO 20x2 female stacking header extra-long pins	1	Bottom

Micro JST 2 pin connectors	1	Top
AA 2500mAh NiMH Rechargeable Battery 4 pack	1	Top
Dual battery clip	1	Top
Single battery clip	1	Top
JST jumper cable	1	Top

Table 2: Battery Board Components.

The next step is to solder the components to the board having as a result the following board:

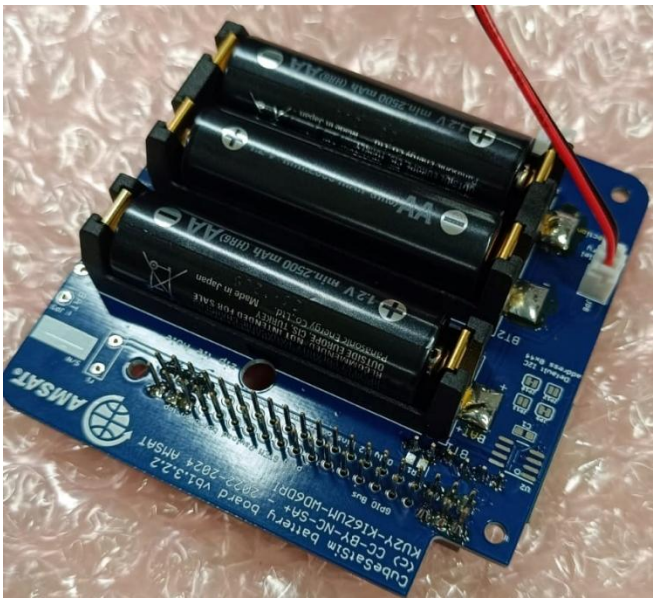


Figure 3: Battery Board completely built.

2.1.3. STEM Payload Board

The aim of this board is to oversee all the reading, processing and transmission of data to ensure the proper functioning of the CubeSatSim. Thanks to this board, the telemetry data can be read and processed to accomplish the sustainable mobility mission.

Data processing and transmission:
The STEM payload board features a Raspberry Pi Pico as its microcontroller which facilitates the management and the programming of all that data. Additionally, it employs various communication protocols, such as I2C, to connect the board with the sensors, and then uses UART to send the data to the Raspberry Pi. The Raspberry Pi, which contains the necessary software and programming, subsequently processes and transmits the data via radio frequency, as will be explained later. Furthermore, the board contains a high-frequency (UHF) radio transceiver (SR105U) that operates at 400-480MHz. It incorporates a high-performance RF transceiver chip, a microcontroller, a RF power amplifier and

433MHz SMA antennas. This module is suitable for long-distance data transfer and wireless communications. Communication with an external controller is done through UART.

Therefore, the components needed are the following:

Component	Quantity	Location
GPIO 20x2 female stacking header extra-long pins	1	Bottom
Push button switch, SPST RA-SPST	1	Top
SC1464-ND 3.50mm Headphone Phone Jack Stereo (3 Conductor, TRS) for RBF switch	1	Top
1k Ohm resistor	1	Top
Green LED	1	Top
220 Ohm resistor	1	Top
Red LED	1	Top
100 Ohm resistor	1	Top
Blue LED	1	Top
IN5817 diode	2	Top
68 Ohm resistor, 1/2 W	2	Top
180 Ohm resistor	1	Top
Raspberry Pi Pico WH	1	Top
USB-C cable and power plug	1	Side
RBF keychain	1	Side
3.5mm plug	1	Side
47uF electrolytic capacitor	1	Top
100nF capacitor	1	Top
220 Ohm resistor	1	Top
100 Ohm resistor	1	Top
20 pin female sockets for Pico	2	Top
4.7k resistor	2	Top
MPU6050 9 Axis Gyro GY-521	1	Top
Female 1x8 header	1	Top
BME280 Board Temperature/Humidity/Pressure	1	Top
Female 1x4 header	1	Top
10k resistor	1	Top

100nF capacitor	1	Top
1N5817 diode	1	Top
1N4148 diode	1	Top
JST 2.0 connector	1	Top
Yellow LED	1	Top
1k resistor	1	Top
White LED	1	Top
1k resistor	1	Top
QWICC connector	1	Top
1x4 male breakaway header	1	Top
2.5mm audio jack	1	Top
SMA vertical female connector	2	Top
SMA 6" male to female for antenna connection	2	Top
SMA 433 MHz antenna	2	Top

Table 3: STEM Payload Board components.

Sensors and data acquisition:

As observed on the table, this board is equipped with sensors which allow it to collect information from the environment. Environmental sensors such as BME280 (takes temperature, pressure and humidity) and TEMP sensor that detects just temperature. This last sensor, as will be explained later will be ignored and BME280 will be the one taken into account. Furthermore, movement sensors such as MPU6050. It is a sensor with 6 axis (accelerometer and gyroscope) that allows it to measure acceleration and rotation on the X, Y and Z axis.

Then, after welding each component and adding the antennas, the final STEM Payload Board should look like in the next *Figure 4*.

Figure 4: STEM Payload Board completely built.

After building the board, a test must be done to ensure its correct functioning. To do so, the Raspberry Pi Pico must be connected to the base computer with micro usb wire. After reading data from Arduino IDE, this should appear with real values when transmitting:

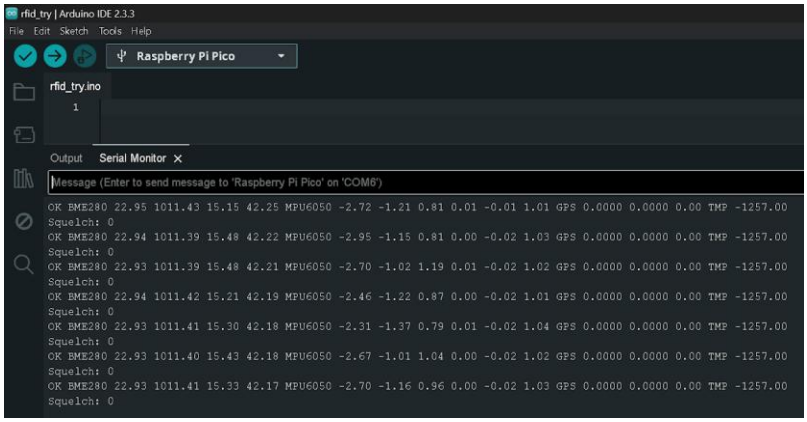


Figure 5: Sensor data read from Raspberry Pi Pico.

On the image above, the different sensors of the board are reading and showing data. The BME280 is an environmental sensor developed by Bosch Sensortec capable of measuring temperature (ºC), pressure (hPa), altitude (m, derived from pressure) and humidity (RH). The TEMP sensor is giving wrong data, as observed. However, thanks to the BME280, this data is already given, so the TEMP sensor will be ignored.

On the other hand, the MPU6050 is a 6-axis motion tracking sensor developed by InvenSense. It combines a 3-axis accelerometer and a 3-axis gyroscope in a single chip, which makes it ideal for this type of application. The accelerometer measures linear acceleration along X, Y and Z axis while the gyroscope measures the angular velocity (rotation) around the same axes. Both sensors use I2C protocol for easy communication with the raspberry.

Lastly, the GPS data appears null. This is because the STEM Payload Board does not yet include a GPS sensor. However, a GPS module is already configured in the software code, allowing for future integration.

After this, the board is correctly configured and ready to be used.

2.2.Structure

Once the three main boards are completely configured and tested, the structure of the CubeSatSim must be assembled. The following components are required:

Structure	
Component	**Quantity**
Battery Board fully assembled	1
STEM Payload Board fully assembled	1
Solar Board fully assembled	1
GPIO 20x2 female stacking header extra-long pins	2
M2.5 screws	8
M2.5 23mm + 6mm standoff	8
M2.5 11mm standoff	2
M2.5 18mm standoff	2
M2.5 6mm + 6mm standoff	2
USB Sound Card	1
OTG cable for Sound Card	1
2.5mm to 3.5mm jumper cable	1
3D Printed Frame (4 parts)	1
Nylon M3 screws for frame	10
Nylon M3 nuts for frame	10
Slotted nylon M2 screws for camera	4
Nylon M2 nuts for camera	4
Solar Cells	10

Micro JST wires	10
Pi Zero 2 with SD card	1
Pi Camera with Pi Zero ribbon cable	1

Table 4: Structure components.

Firstly, each Micro JST wire needs to be soldered to the solar panels, (see Figure 6).

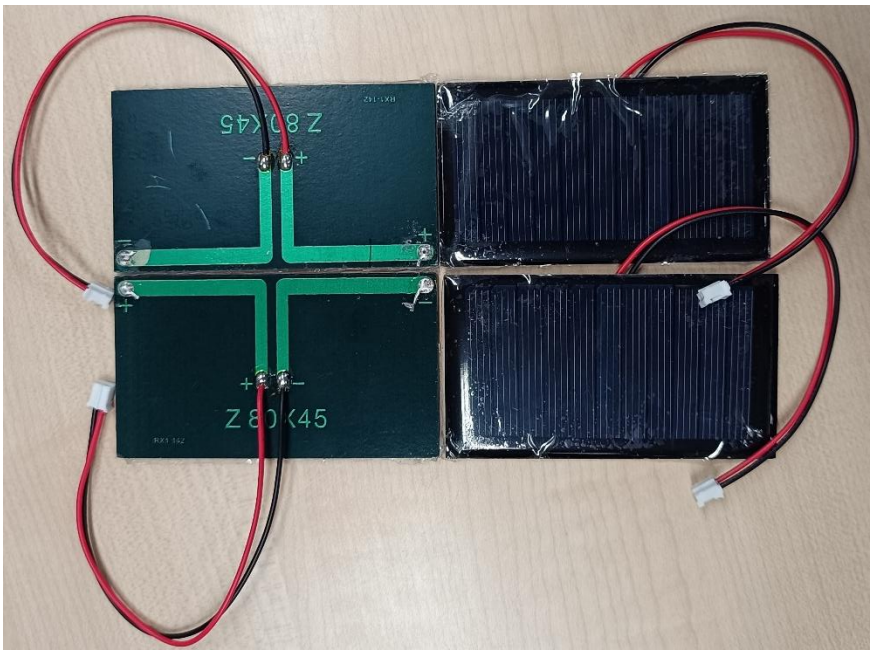


Figure 6: Solar panels with soldered Micro JST wires.

Then, the camera module should be connected to the Raspberry Pi Zero using the ribbon cable.

The three main boards must be stacked and secured with screws in the following order: Solar Board on top, STEM Payload Board in the middle, and Battery Board at the bottom. The Raspberry Pi Zero, along with the camera, should be positioned at the bottom of the Battery Board (see Figure 7).

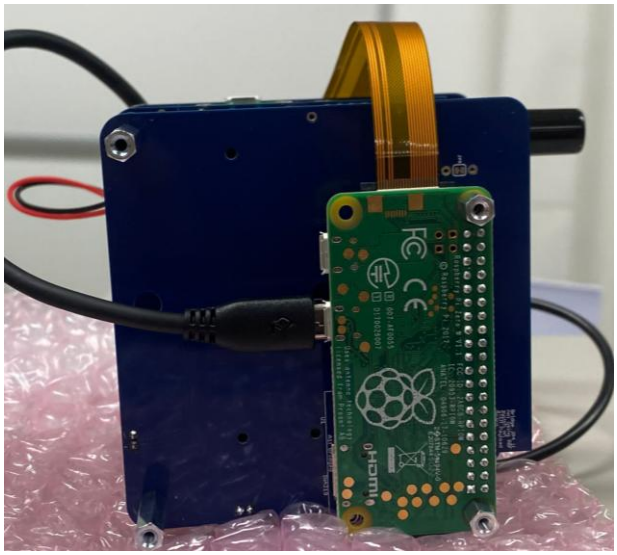


Figure 7: Camera and Raspberry Pi Zero.

JST wires are used to establish the connection between the boards. In order to integrate the USB sound card, the following two steps are needed to follow: connect the OTG cable for USB sound card to the Raspberry Pi Zero. Then, a jumper cable links the USB sound card to the STEM Payload Board. This configuration allows the Battery Board (responsible for power management) to interface with the STEM Payload Board (which processes sensor data and transmits it).

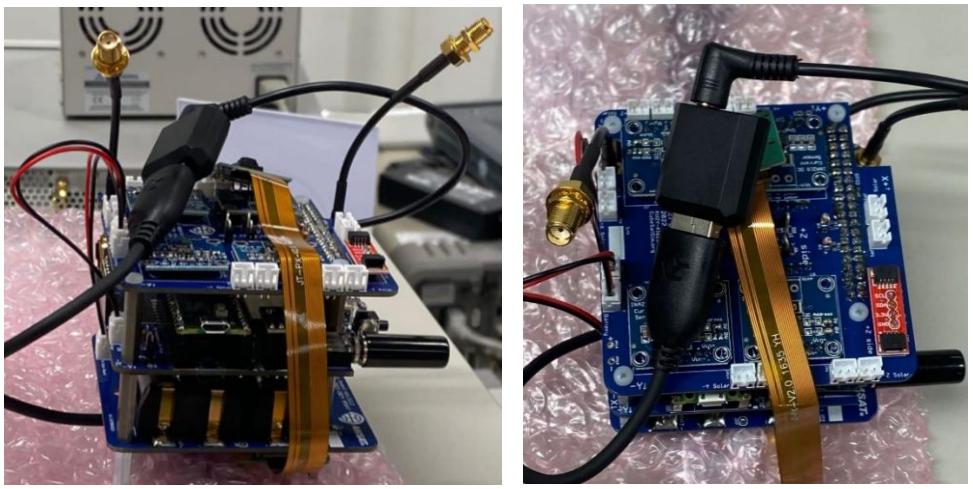


Figure 8: Three boards connected and built.

The 3D-printed frames must be prepared and checked for proper fit. Once the electronic components are secured, the entire system is enclosed within the 3D-printed CubeSatSim structure, forming a complete cube. The solar panels need to be pasted with double sided tape as shown in the following pictures.

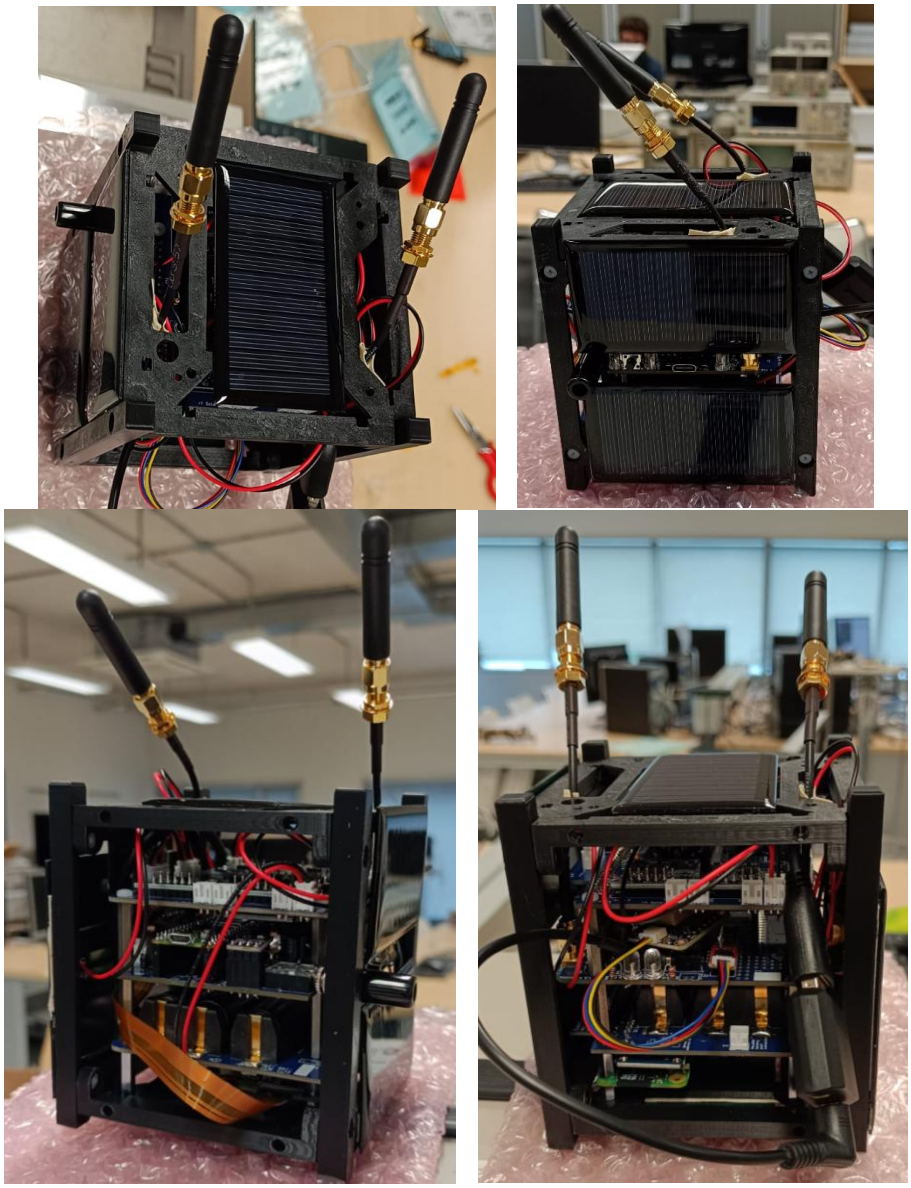


Figure 9: Full structure of the CubeSatSim.

For added stability, the antennas are glued with silicone to the interior walls and a supporting stick is attached to the antennas to hold them in position and ensure structural integrity.

At this point, the CubeSat's hardware is ready to carry out the mission.

3. SOFTWARE

In this section, the installation of the software will be explained. Furthermore, the different tests done with the boards and with the telemetry will be shown below.

In order to fully understand how it works and how has been everything tested, the first step is to install the software.

3.1. Software Installation

3.1.1. Creation and flashing the Raspberry Pi Image

Firstly, to install the necessary software for the Raspberry Pi Zero to function properly, it is necessary to clone the software files from Alan Johnson's GitHub repository into the SD card (inside the Raspberry Pi). The materials needed to do so are the following:

Software installation	
Component	**Quantity**
USB-C cable and power plug	1
Pi Zero WH (with pre-soldered headers)	1
16GB micro-SD Card	1
Pico WH (with pre-soldered headers)	1
Computer with Raspberry Pi Imager program	1

Table 5: Required materials for software installation.

The first step to flash the software into the SD card is to insert the micro-SD card into a computer and run "Raspberry Pi Imager". This program is used to write an operating system image onto the SD card, preparing it for use with the Raspberry Pi.

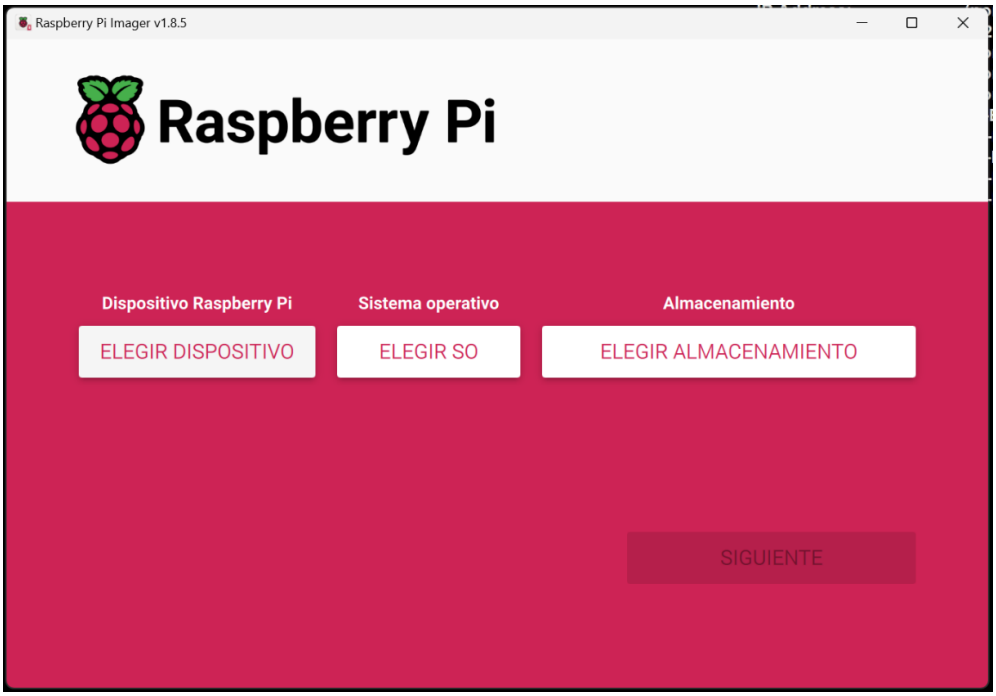


Figure 10: Raspberry Pi Imager program.

Then, the following options have to be selected: under "Dispositivo Raspberry Pi" select the Raspberry Pi Zero, for "Sistema Operativo" choose Raspberry Pi OS (Legacy, 32-bit) Lite as the operating system and under "Almacenamiento" select the micro-SD card inserted.

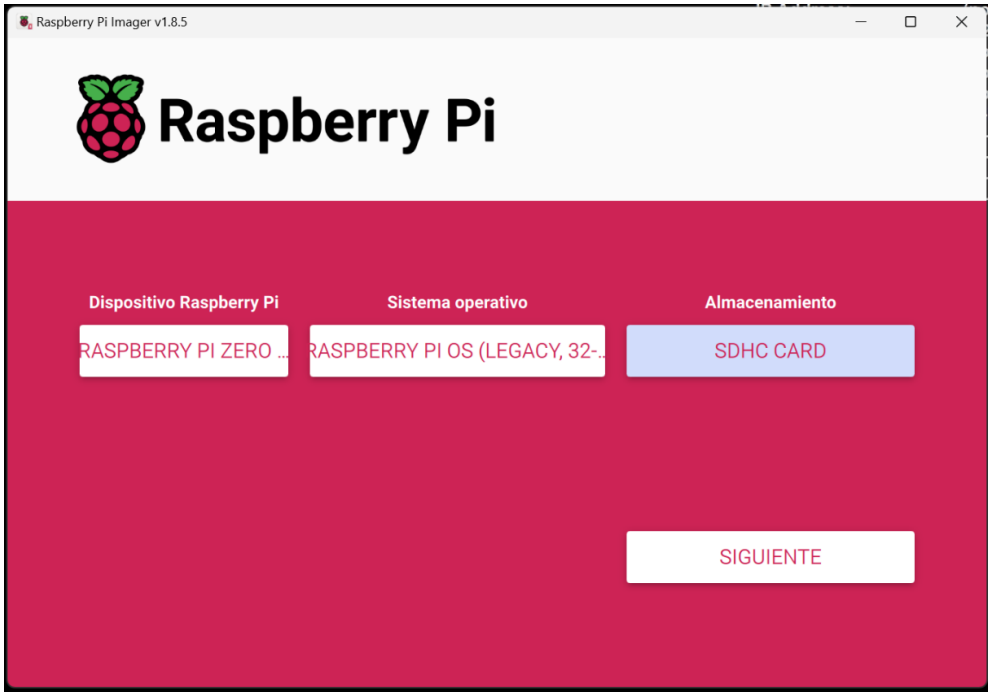


Figure 11: Options selected for the creation of the image.

When selecting "Siguiente" a new window will appear.

Figure 12: Edit setting of OS.

Select the option “Editar Ajustes” in order to apply our own configuration. These settings should look like the next image:

Figure 13: Image configuration settings for Raspberry Pi flashing.

Finally, in "Servicios" enable the option "SSH" using password authentication (which will be the one specified on Figure 13) Then, click on "SAVE" and continue flashing all data into the micro-SD card.

Figure 14: SSH activation.

Once the flash is finished, remove the card from the computer and insert it into the Raspberry Pi Zero and power it to boot it up and test its functioning.

Once the Raspberry Pi has booted up, if it has been properly enabled, a connection between a computer and the raspberry can be established. The Raspberry Pi should automatically connect to the previously configured Wi-Fi network. The computer must be connected to the same Wi-Fi network. Then, in the Terminal Window or Windows Command Prompt, you can type the next command to log in into the raspberry:

ssh pi@CubeSatSim.local (being "CubeSatSim" the hostname defined before)

The first time logging into the raspberry, a question will appear: *"Are you sure you want to continue connecting (yes/no/[fingerprint])?"* Type "yes". At the password prompt, enter the previously defined password (set in the image configuration settings). Once this process is completed, the Raspberry Pi and the computer should be connected, displaying the following interface:

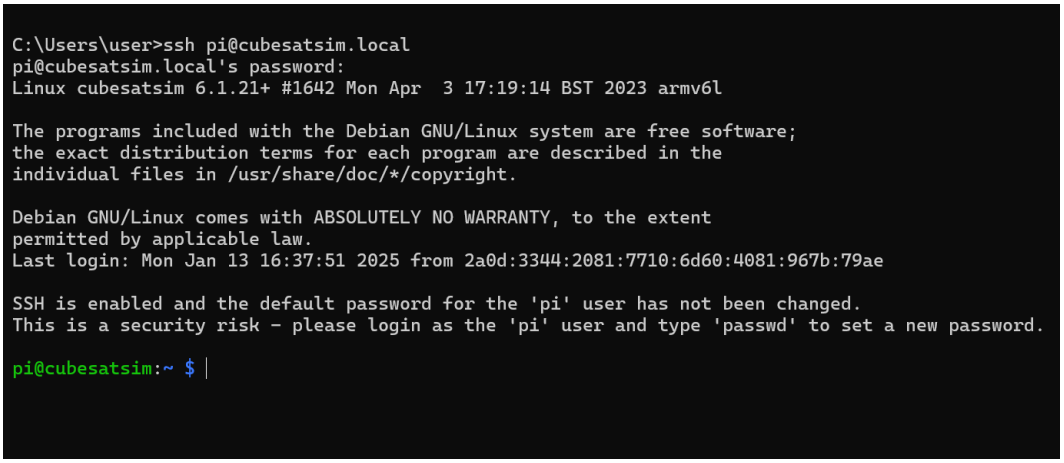


Figure 15: SSH interface once it is connected.

At this point, the base station (computer) can access files and data stored in the Raspberry Pi. Once SSH access to the Raspberry Pi Zero has been established, the next step is to install the CubeSatSim software.

The first task is to ensure that the Raspberry Pi is connected to the Internet. This can be verified by executing the following command in the terminal:

timeout 10 ping amazon.com

If the response includes messages such as "64 bytes received", this means that the device is successfully connected to the network. If not, the Wi-Fi connection should be checked. Once the Internet connection has been confirmed, the following commands must be executed in the terminal to update the system and install the necessary tools:

sudo apt-get update && sudo apt-get dist-upgrade -y
sudo apt-get install -y git

git clone http://github.com/alanbjohnston/CubeSatSim.git
cd CubeSatSim
git checkout master

After completing these steps, the CubeSatSim repository will be successfully cloned to the Raspberry Pi. To finalize the installation, the following command must be executed:

./install

Now the CubeSatSim is completely configured.

3.2. Ground Station

The ground station (receiving computer) for the CubeSatSim is used to receive data via Radiofrequency and decode it to see the telemetry. This section will explain the material and installations needed to be done in order to accomplish this function.

Ground Station	
Component	**Quantity**
PC with RTL-SDR	1
SMA Antenna for 433 MHz right angle	1

Table 6: Material needed to configure the Ground Station.

To receive the radio transmission, it is needed to use the RTL-SDR.

Figure 16: RTL-SDR receiver.

This radio receptor allows us to receive and process radio signals in a range of frequencies: normally 500 kHz to 1.75 GHz. In this case, around 434.9MHz which will be the transmitting frequency. It works as a radio scanner capable of receiving signals without a specific hardware or software. With a compatible program with the RTL-SDR, the data received can be well decoded.

3.2.1. SDR#

Firstly, a program called SDR# (SDRSharp)[8] must be downloaded and executed, to ensure that the RTL-SDR dongle works correctly and receives radio signals. This program can be used to receive, visualize, decode and analyze radio signals from radio receivers such as RTL-SDR. It provides, among other things, a spectrogram that shows active radio transmissions and their signal strength over time (see in Figure 17).

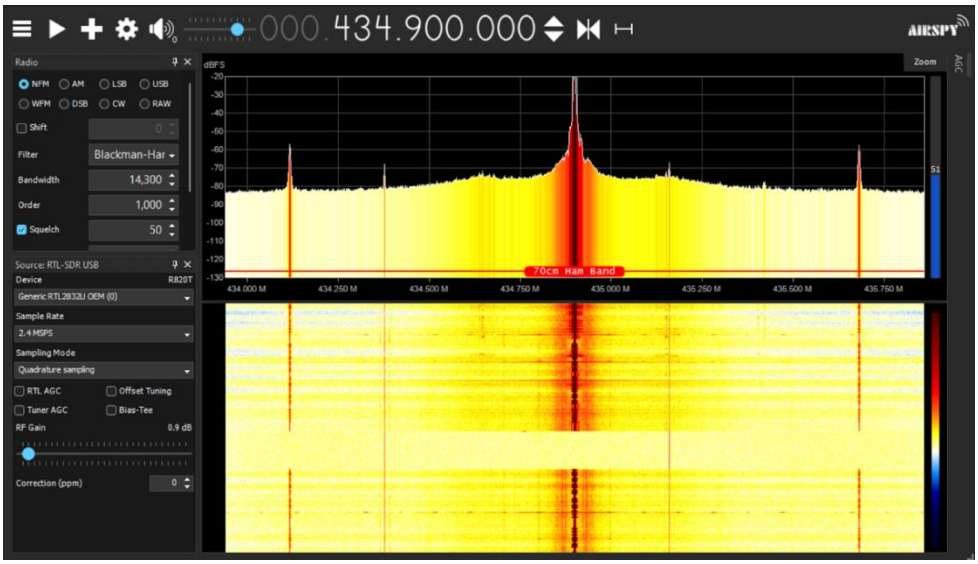


Figure 17: Interface of signal detection at 434.9MHz.

Thanks to this program, the spectrum is displayed, and it ensures that the RTL-SDR works correctly and the cubesatsim sends wave signals at a center frequency of 434.9MHz, which is the desired frequency (with a bandwidth of 14.3kHz). The modulation mode used is NFM (Narrowband Frequency Modulation), typical for amateur radio communications. Also, to reduce interference and improve the signal a Blackman-Harris filter is used.

The spectrum (signal intensity in frequency domain) is displayed in dBFS (decibels full scale for amplitude). The peak at 434.9MHz indicates the active transmission at that frequency. The spectrum below is the evolution of the signal in time (vertical axis is time). Lastly, the sampling mode used is quadrature sampling, which allows demodulation of complex signals.

Furthermore, it enables signal demodulation. This allows us to convert received radio frequency signals into intelligible audio or data, used for further processing and analysis. However, this program is just used to see the spectrogram and the power of the signal received. For reading APRS packet telemetry, this program is used.

[8] https://airspy.com/download/

3.2.2. FoxTelem

The Ground Station for the CubeSatSim uses FoxTelem, the open source AMSAT telemetry decoding software by Chris Thompson[9], to decode telemetry. The software is designed to work with radio receivers such as RTL-SDR. It can decode telemetry signals received in BPSK (Binary Phase Shift Keying) and FSK (Frequency Shift Keying). The characteristics that make this program interesting are the graphing and historical data analysis and the analysis of parameters such as power consumption, temperature fluctuations, and overall, satellite health telemetry. Once it is downloaded and the RTL-SDR is connected to the computer, the program must be executed. Two spacecrafts (from "Spacecraft" menu) ought to be added to decode telemetry correctly: "CubeSat_Simulator_DUV.MASTER" (for FSK modulation) and "CubeSat_Simulator_PSK.MASTER" (for BPSK modulation). At this point, the program will be prepared to receive data. Now the software is prepared to start receiving data from the CubeSatSim.

When there is no signal reception, the interface of the program should look like this:

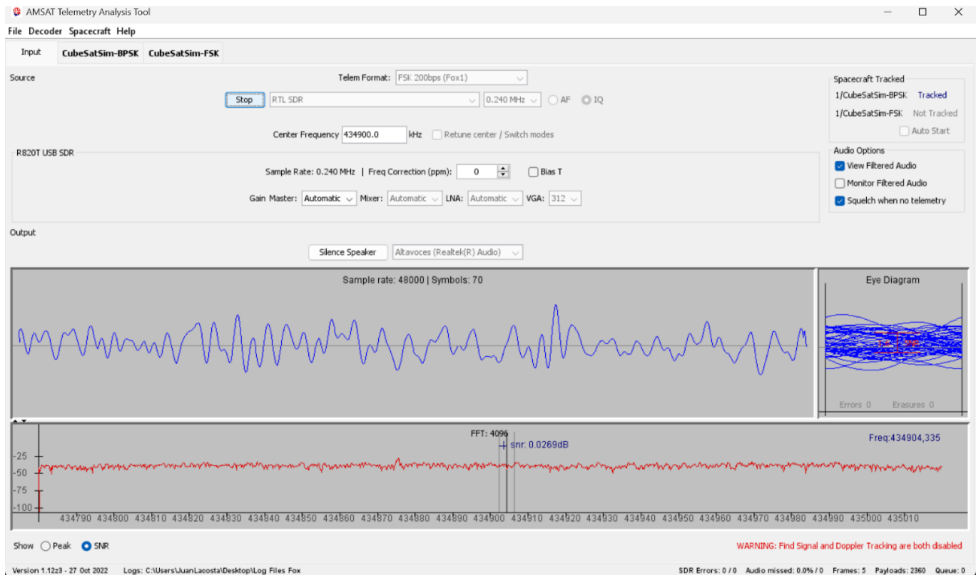


Figure 18: Interface of FoxTelem with no real FSK signal detection.

[9] https://www.amsat.org/foxtelem-software-for-windows-mac-linux/

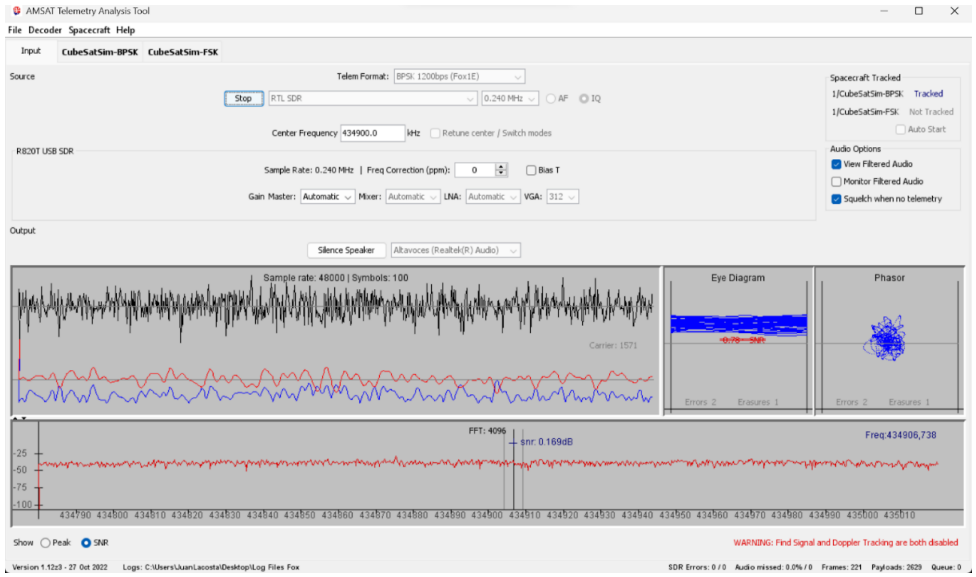


Figure 19: Interface of FoxTelem with no real BPSK signal detection.

However, when the cubesatsim starts transmitting data and the program detects it and decode it, the interface should look like in the following pictures:

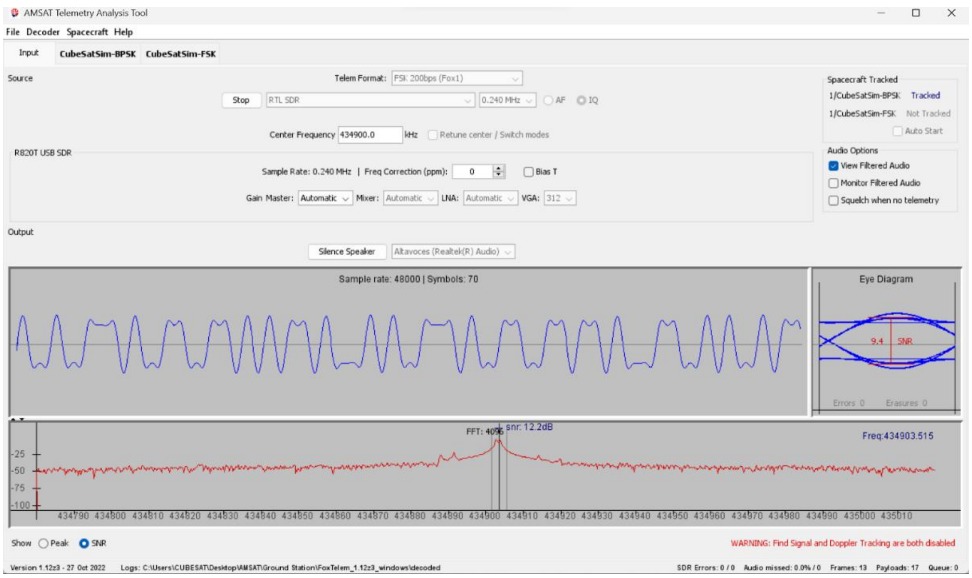


Figure 20: Interface of FoxTelem with FSK signal detection.

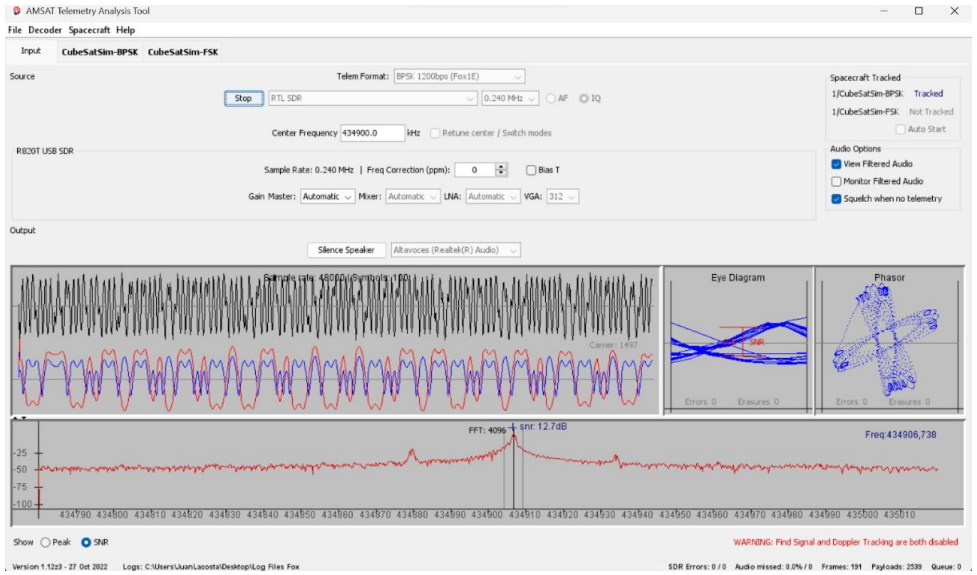


Figure 21: Interface of FoxTelem with BPSK signal detection.

In both pictures, there are three different charts: signal reception (top left), eye diagram (top right) and FFT spectrum analysis (bottom). The BPSK interface includes a Phasor graph too (top right).

On the top left graph (modulated waveform display), the representation of the waveform of the demodulated signal received via the RTL-SDR is shown. It is a time-domain representation of the signal from the Cubesatsim. It displays this modulated waveform in frequency (FSK) or phase (BPSK). As shown in both images, the signal shows a clear modulation. However, the "interruptions" or distortions seen in the signals indicate problems of noise or a low signal-to-noise-ratio (SNR). Also, these imperfections could be too because of the poor antenna quality or alignment or a low transmission power of the signal.

On the other hand, on the top left graph, the eye diagram is shown. This chart is very important in telecommunications because it shows a representation of the quality of the digital signal after demodulation. It is created by superimposing consecutive bit periods of a signal on top of each other's (overlapping signal waveforms). It can be observed that it resembles an open eye. This gives information about the signal integrity, noise levels (SNR) and overall transmission quality. The width of the eye opening represents the temporal stability of the received bits whereas the height of the eye opening indicates the clarity in distinguishing between 1s and 0s (bits) in the signal. A well opened (horizontally and vertically) signifies a clear signal with low interference and makes it easier for the receiver to distinguish between 1s and 0s. However, if this one is closed or deformed, it will mean there is noise or interference affecting transmission and reception. Furthermore, with the eye diagram, the SNR is shown in dB. A higher SNR means a better signal quality. If this exceeds 10dB it is generally good, while values below 5dB indicate severe noise issues.

On the bottom chart, the FFT (Fast Fourier Transform) is shown. It is a frequency spectrum analysis of the received signal. It allows visualizing the exact frequency at

which the cubesatsim is transmitting and shows any nearby interference in frequency. The X-axis shows frequency in Hz, the Y-axis, signal intensity in dB and the main peak shows the Cubesatsim signal's peak value in dB. The reception frequency shown in this chart should match the transmitting frequency, which in both cases it does.

Lastly, on BPSK modulation a fourth chart appears: the phasor representation. It is the representation of the signal's phase behavior in the complex plane. In signal processing, signals are represented by their magnitude (amplitude) and phase. In BPSK, data is encoded using two possible phase states. That is the reason why in an ideal case, in this chart, two distinct points (separated 180º) should appear, corresponding to binary 0 and 1. This would mean that the receiver is detecting phase shifts and consequently, low bit error rates. If instead of distinct points, they are scattered, this indicates that noise is distorting the signal. In Figure 21, two distinct (but a little bit scattered) points are shown. This means that they are a bit distorted by noise or interferences. A more compact cluster means the quality of the signal is better.

3.3. Code[10]

In this subsection of the Software, the code implemented on the Cubesatsim will be explained to provide a better understanding of the satellite's functionality.

The following flowchart illustrates the internal workings of the Cubesatsim and the execution of its software.

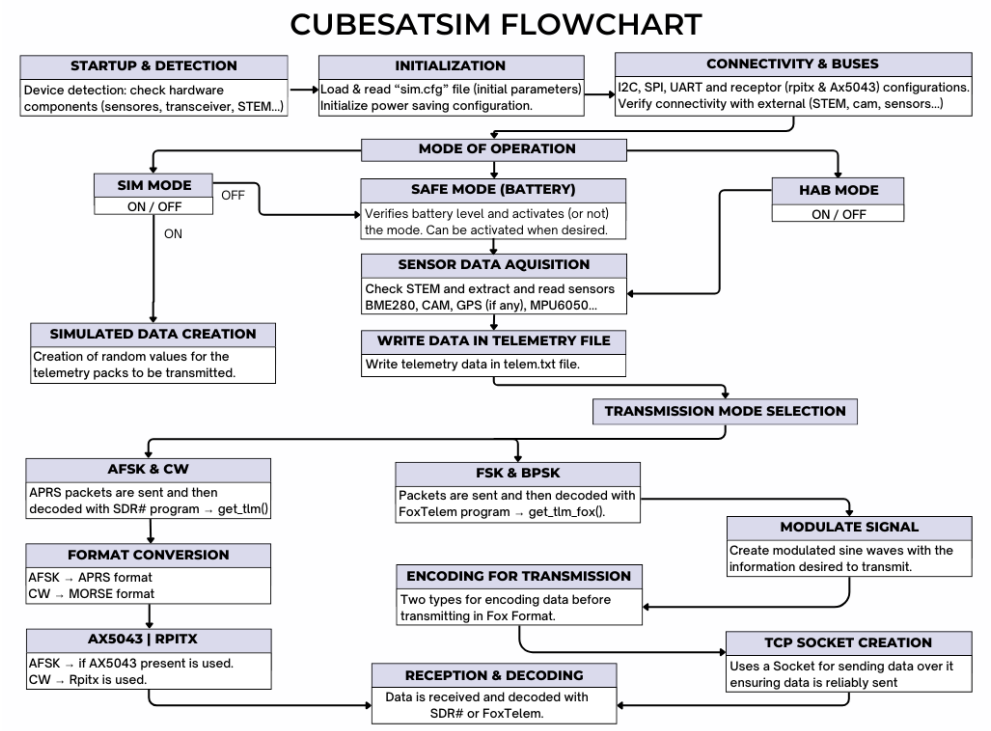


Figure 22: Flowchart of CubeSatSim's software

[10] https://github.com/alanbjohnston/CubeSatSim/blob/master/main.c

From this point onward, the flowchart will be explained in detail through the different sections of the code.

3.3.1. Startup and Initialization

Firstly, regarding the initialization, when the CubeSat turns on, the first thing it does is to detect the raspberry pi zero, in order to access the software.

```
// ============================================================================================================
// - DEVICE DETECTION
// ============================================================================================================

  char resbuffer[1000];
  const char testStr[] = "cat /proc/cpuinfo | grep 'Revision' | awk '{print $3}' | sed 's/^1000//' | grep '902120'";
  FILE *file_test = sopen(testStr);  // see if Pi Zero 2
  fgets(resbuffer, 1000, file_test);
  fprintf(stderr, "Pi test result: %s\n", resbuffer);
  fclose(file_test);

  fprintf(stderr, " %x ", resbuffer[0]);
  fprintf(stderr, " %x ", resbuffer[1]);
  if ((resbuffer[0] == '9') && (resbuffer[1] == '0'))
  {
    sleep(5);  // try sleep at start to help boot
    voltageThreshold = 3.7;
    printf("Pi Zero 2 detected");
  }

  printf("\n\nCubeSatSim v1.3.2 starting...\n\n");

// ============================================================================================================
```

Figure 23: Device detection section code.

Once the raspberry is detected, the initial configuration is loaded. It creates a file named “sim.cfg” in which loads initial configuration such as latitude, altitude, simulated telemetry status, HAB mode status and transmission and reception center frequency.

```
// ============================================================================================================
// - LOADING INITIAL CONFIGURATION
// ============================================================================================================

  // Open configuration file with callsign and reset count
  FILE * config_file = fopen("/home/pi/CubeSatSim/sim.cfg", "r");
  if (config_file == NULL) {
    printf("Creating config file.");
    config_file = fopen("/home/pi/CubeSatSim/sim.cfg", "w");
    fprintf(config_file, "%s %d", " ", 100);
    fclose(config_file);
    config_file = fopen("/home/pi/CubeSatSim/sim.cfg", "r");
  }

  fscanf(config_file, "%s %d %f %f %s %d %s %s %s",
    call, & reset_count, & lat_file, & long_file, sim_yes, & squelch, tx, rx, hab_yes);
  fclose(config_file);
  fprintf(stderr,"Config file /home/pi/CubeSatSim/sim.cfg contains %s %d %f %f %s %d %s %s %s\n",
    call, reset_count, lat_file, long_file, sim_yes, squelch, tx, rx, hab_yes);
  fprintf(stderr, "Transmit on %s Receive on %s\n", tx, rx);

  reset_count = (reset_count + 1) % 0xffff;

  if ((fabs(lat_file) > 0) && (fabs(lat_file) < 90.0) && (fabs(long_file) > 0) && (fabs(long_file) < 180.0)) {
    fprintf(stderr, "Valid latitude and longitude in config file\n");
    latitude = lat_file;
    longitude = long_file;
    fprintf(stderr, "Lat/Long %f %f\n", latitude, longitude);
    fprintf(stderr, "Lat/Long in APRS DDMM.MM format: %07.2f/%08.2f\n", toAprsFormat(latitude), toAprsFormat(longitude));
    newGpsTime = millis();

  } else { // set default
      newGpsTime = millis();
  }
// ============================================================================================================
```

Figure 24: Code for the load of initial configuration.

Finally, LED's pins and ports are being configured and initialized.

```
// ========================================================
// - CONFIGURACIÓN E INICIALIZACIÓN LEDS (PUERTOS, PINES...)
// ========================================================
  txLed = 0; // defaults for vB3 board without TFB
  txLedOn = LOW;
  txLedOff = HIGH;
  if (!ax5043) { // COMPRUEBA QUE EL TRANSCEPTOR ESTÁ PRESENTE
    pinMode(2, INPUT);
    pullUpDnControl(2, PUD_UP);
    if (digitalRead(2) != HIGH) {
      printf("vB3 with TFB Present\n");
      vB3 = TRUE; txLed = 3;
      txLedOn = LOW; txLedOff = HIGH;
      onLed = 0; onLedOn = LOW;
      onLedOff = HIGH; transmit = TRUE;
    } else {
      pinMode(3, INPUT);
      pullUpDnControl(3, PUD_UP);
      if (digitalRead(3) != HIGH) {
        printf("vB4 Present with UHF BPF\n");
        txLed = 2; txLedOn = HIGH;
        txLedOff = LOW; vB4 = TRUE;
        onLed = 0; onLedOn = HIGH;
        onLedOff = LOW; transmit = TRUE;
      } else {
        pinMode(26, INPUT);
        pullUpDnControl(26, PUD_UP);
        if (digitalRead(26) != HIGH) {
          printf("v1 Present with UHF BPF\n");
          txLed = 2;
          txLedOn = HIGH; txLedOff = LOW;
          vB5 = TRUE; onLed = 27;
          onLedOn = HIGH; onLedOff = LOW;
          transmit = TRUE;
        }
  else {
          pinMode(23, INPUT);
          pullUpDnControl(23, PUD_UP);

          if (digitalRead(23) != HIGH) {
            printf("v1 Present with VHF BPF\n");
            txLed = 2;
            txLedOn = HIGH;
            txLedOff = LOW;
            vB5 = TRUE;
            onLed = 27;
            onLedOn = HIGH;
            onLedOff = LOW;
            printf("VHF BPF not yet
                    supported so no transmit\n");
      transmit = FALSE;
          } } } }  }
  pinMode(txLed, OUTPUT);
  digitalWrite(txLed, txLedOff);
  #ifdef DEBUG_LOGGING
  printf("Tx LED Off\n");
  #endif
  pinMode(onLed, OUTPUT);
  digitalWrite(onLed, onLedOn);
  #ifdef DEBUG_LOGGING
  printf("Power LED On\n");
  #endif
// ========================================================
```

Figure 25: LED's configuration and initialization.

3.3.2. Transmission mode selection

The next section of the code explains the configuration of two modes: SIM and HAB.

```
// ========================================================
// - SIM / HAB MODE ON
// ========================================================

  if (strcmp(sim_yes, "yes") == 0) {
    sim_mode = TRUE;
    fprintf(stderr, "Sim mode is ON\n");
  }
  if (strcmp(hab_yes, "yes") == 0) {
    hab_mode = TRUE;
    fprintf(stderr, "HAB mode is ON\n");
  }
// ========================================================
```

Figure 26: SIM/HAB modes activation.

The simulation mode is the same as the simulated telemetry mode and replicates the real behavior of the CubeSat systems. It creates random values for the sensors, voltage and current of the batteries and solar panels, buses, camera, latitude, longitude, maximum values… and transmit it as if it were the real values of the CubeSat. In order to do so, the process is divided into two parts:

Initialization of random values and parameters:
When entering the simulated modes, random values are generated for several parameters, as commented before. The following code performs this initial setup:

```
// ============================================================================
// - MODO SIMULACIÓN DE TELEMETRÍA
// ============================================================================

  if ((i2c_bus3 == OFF) || (sim_mode == TRUE)) {
    sim_mode = TRUE;
    fprintf(stderr, "Simulated telemetry mode!\n");
    srand((unsigned int)time(0));
    axis[0] = rnd_float(-0.2, 0.2);
    if (axis[0] == 0)
      axis[0] = rnd_float(-0.2, 0.2);
    axis[1] = rnd_float(-0.2, 0.2);
    axis[2] = (rnd_float(-0.2, 0.2) > 0) ? 1.0 : -1.0;
    angle[0] = (float) atan(axis[1] / axis[2]);
    angle[1] = (float) atan(axis[2] / axis[0]);
    angle[2] = (float) atan(axis[1] / axis[0]);
    volts_max[0] = rnd_float(4.5, 5.5) * (float) sin(angle[1]);
    volts_max[1] = rnd_float(4.5, 5.5) * (float) cos(angle[0]);
    volts_max[2] = rnd_float(4.5, 5.5) * (float) cos(angle[1] - angle[0]);
    float amps_avg = rnd_float(150, 300);
    amps_max[0] = (amps_avg + rnd_float(-25.0, 25.0)) * (float) sin(angle[1]);
    amps_max[1] = (amps_avg + rnd_float(-25.0, 25.0)) * (float) cos(angle[0]);
    amps_max[2] = (amps_avg + rnd_float(-25.0, 25.0)) * (float) cos(angle[1] - angle[0]);
    batt = rnd_float(3.8, 4.3);
    speed = rnd_float(1.0, 2.5);
    eclipse = (rnd_float(-1, +4) > 0) ? 1.0 : 0.0;
    period = rnd_float(150, 300);
    tempS = rnd_float(20, 55);
    temp_max = rnd_float(50, 70);
    temp_min = rnd_float(10, 20);

    #ifdef DEBUG_LOGGING
    for (int i = 0; i < 3; i++)
      printf("axis: %f angle: %f v: %f i: %f \n", axis[i], angle[i], volts_max[i], amps_max[i]);
    printf("batt: %f speed: %f eclipse_time: %f eclipse: %f period: %f temp: %f max: %f min: %f\n", batt,
      speed, eclipse_time, eclipse, period, tempS, temp_max, temp_min);
    #endif
    time_start = (long int) millis();
    eclipse_time = (long int)(millis() / 1000.0);
    if (eclipse == 0.0)
      eclipse_time -= period / 2; // if starting in eclipse, shorten interval
  }
  tx_freq_hz -= tx_channel * 50000;
  if (transmit == FALSE) {
    fprintf(stderr, "\nNo CubeSatSim Band Pass Filter detected.  No transmissions after the CW ID.\n");
  }

// ============================================================================
```

Figure 27: Initialization of random values for SIM mode.

This code initializes all the necessary parameters for simulating CubeSat's real behavior such as voltages and currents for solar panels, orientation, battery values… The eclipse cycle mentioned in the code will affect energy supply throughout the simulation.

Simulation of dynamic values:

Once the parameters are randomly defined, thanks to the next simulation code, these values vary mimicking the conditions that a real CubeSat would experience.

```
// ============================================================================================================
// - GENERACIÓN DE SIMULACIÓN PARA TESTEAR CON VALORES REALES
// ============================================================================================================

  if (sim_mode) { // simulated telemetry
    double time = ((long int)millis() - time_start) / 1000.0;
    if ((time - eclipse_time) > period) {
      eclipse = (eclipse == 1) ? 0 : 1;
      eclipse_time = time;
      printf("\n\nSwitching eclipse mode! \n\n");
    }
    double Xi = eclipse * amps_max[0] * (float) sin(2.0 * 3.14 * time / (46.0 * speed)) + rnd_float(-2, 2);
    double Yi = eclipse * amps_max[1] * (float) sin((2.0 * 3.14 * time / (46.0 * speed)) + (3.14 / 2.0)) + rnd_float(-2, 2);
    double Zi = eclipse * amps_max[2] * (float) sin((2.0 * 3.14 * time / (46.0 * speed)) + 3.14 + angle[2]) + rnd_float(-2, 2);
    double Xv = eclipse * volts_max[0] * (float) sin(2.0 * 3.14 * time / (46.0 * speed)) + rnd_float(-0.2, 0.2);
    double Yv = eclipse * volts_max[1] * (float) sin((2.0 * 3.14 * time / (46.0 * speed)) + (3.14 / 2.0)) + rnd_float(-0.2, 0.2);
    double Zv = 2.0 * eclipse * volts_max[2] * (float) sin((2.0 * 3.14 * time / (46.0 * speed)) + 3.14 + angle[2]) + rnd_float(-0.2, 0.2);
    current[map[PLUS_X]] = (Xi >= 0) ? Xi : 0;
    current[map[MINUS_X]] = (Xi >= 0) ? 0 : ((-1.0f) * Xi);
    current[map[PLUS_Y]] = (Yi >= 0) ? Yi : 0;
    current[map[MINUS_Y]] = (Yi >= 0) ? 0 : ((-1.0f) * Yi);
    current[map[PLUS_Z]] = (Zi >= 0) ? Zi : 0;
    current[map[MINUS_Z]] = (Zi >= 0) ? 0 : ((-1.0f) * Zi);
    voltage[map[PLUS_X]] = (Xv >= 1) ? Xv : rnd_float(0.9, 1.1);
    voltage[map[MINUS_X]] = (Xv <= -1) ? ((-1.0f) * Xv) : rnd_float(0.9, 1.1);
    voltage[map[PLUS_Y]] = (Yv >= 1) ? Yv : rnd_float(0.9, 1.1);
    voltage[map[MINUS_Y]] = (Yv <= -1) ? ((-1.0f) * Yv) : rnd_float(0.9, 1.1);
    voltage[map[PLUS_Z]] = (Zv >= 1) ? Zv : rnd_float(0.9, 1.1);
    voltage[map[MINUS_Z]] = (Zv <= -1) ? ((-1.0f) * Zv) : rnd_float(0.9, 1.1);
    tempS += (eclipse > 0) ? ((temp_max - tempS) / 50.0f) : ((temp_min - tempS) / 50.0f);
    tempS += +rnd_float(-1.0, 1.0); other[IHU_TEMP] = tempS;
    voltage[map[BUS]] = rnd_float(5.0, 5.005);
    current[map[BUS]] = rnd_float(158, 171);
    float charging = eclipse * (fabs(amps_max[0] * 0.707) + fabs(amps_max[1] * 0.707) + rnd_float(-4.0, 4.0));
    current[map[BAT]] = ((current[map[BUS]] * voltage[map[BUS]]) / batt) - charging;
```

Figure 28: Simulation of dynamic values for SIM mode.

This block of code shows how current and voltage values change with time and the presence of the eclipse. Temperature fluctuates too as a result of these changes.

On the other hand, the HAB (High Altitude Ballon) refers to an operational mode designed to simulate a high-altitude transmission during CubeSat simulation testing. The objective of this mode is to operate the CubeSat onboard a High-Altitude Balloon (HAB), allowing it to read, process, and transmit data under conditions similar to those in space. The balloon typically ascends to altitudes between 30 and 40 km, where environmental factors such as temperature, pressure, and humidity change drastically as it moves through the troposphere and stratosphere. Most commercial GPS modules stop working above 18 km due to altitude restrictions. However, the HAB mode applies software-based filters and processing techniques to improve positioning accuracy and enable the continued transmission of coordinates beyond this limit. Additionally, HAB mode optimizes communication for long-distance transmissions by increasing transmission power, sending data at higher frequencies, and supporting multiple modulation modes (such as APRS, RTTY, and SSTV). These adjustments enhance data reliability but also require more processing power and energy consumption, reducing battery life compared to standard CubeSat mode operation.
By using HAB mode, CubeSats can simulate real mission conditions, test their communication systems in low-pressure environments, and evaluate how well they can operate before an actual space deployment. This last mode is crucial for the mobility mission. Setting the CubeSat to real conditions and enabling a long connection link for transmitting data to the base station.

```
if (strcmp(hab_yes, "yes") == 0) {
    hab_mode = TRUE;
    fprintf(stderr, "HAB mode is ON\n");
}
if (hab_mode)
    fprintf(stderr, "HAB mode enabled - in APRS balloon icon and no battery saver or low voltage shutdown\n");
//#ifdef HAB
//    if (hab_mode)
//        fprintf(stderr, "HAB mode enabled - in APRS balloon icon and no battery saver or low voltage shutdown\n");
//#endif
```

Figure 29: HAB mode functionality.

Once the modes of operation are chosen, the telemetry transmission mode is configured.

```
// ========================================
// - CONFIGURACIÓN MODO DE TELEMETRÍA
// ========================================
  mode = FSK;
  frameCnt = 1;
  if (argc > 1) {
    if ( * argv[1] == 'b') {
      mode = BPSK;
      printf("Mode BPSK\n");
    } else if ( * argv[1] == 'a') {
      mode = AFSK;
      printf("Mode AFSK\n");
    } else if ( * argv[1] == 'm') {
      mode = CW;
      printf("Mode CW\n");
    } else {
      printf("Mode FSK\n");
    }
    if (argc > 2) {
      loop = atoi(argv[2]);
      loop_count = loop;
    }
    printf("Looping %d times \n", loop);

    if (argc > 3) {
      if ( * argv[3] == 'n') {
        cw_id = OFF;
        printf("No CW id\n");
      }
    }
  } else {
      FILE * mode_file = fopen
      ("/home/pi/CubeSatSim/.mode", "r");
    if (mode_file != NULL) {
      char mode_string;
      mode_string = fgetc(mode_file);
      fclose(mode_file);
      printf("Mode file /home/pi/CubeSatSim/.mode
        contains %c\n", mode_string);

      if ( mode_string == 'b') {
        mode = BPSK;
        printf("Mode is BPSK\n");
      } else if ( mode_string == 'a') {
        mode = AFSK;
        printf("Mode is AFSK\n");
      } else if ( mode_string == 's') {
        mode = SSTV;
        printf("Mode is SSTV\n");
      } else if ( mode_string == 'm') {
        mode = CW;
        printf("Mode is CW\n");
      } else {
        printf("Mode is FSK\n");
      }
    }
  }

// ========================================
```

Figure 30: Definition of the different modes of telemetry transmission.

The CubeSat has five modes of transmitting telemetry (and encoding data) to the base station. Each of them will encode differently and, sometimes, even send different data.

<u>AFSK (Audio Frequency Shift Keying):</u>
It is a type of FSK modulation where digital data is first encoded into audio signal and then is transmitted by radio frequency. It commonly operates at a baud rate of 1200 bps and is widely used in systems like APRS (Automatic Packet Reporting System), which is one of the systems used in CubeSats for telemetry transmission.
The AX.25 protocol, used in AFSK, defines how data is packaged and transmitted in small packets, which can include telemetry, messages, and other information. This protocol allows different radio stations to communicate by sending and receiving these packets. In the case of APRS, AFSK with AX.25 is used to transmit real-time data such as location information, sensor readings, messages, and telemetry from the CubeSat to ground stations. A key component in CubeSat communication systems is the AX5043 chip, a transceiver that enables reliable data

transmission using AFSK and AX.25. In the following image, the configuration of the AFSK mode is shown. When the digital transceiver AX5043 is detected and the SPI configuration is defined, then the CubeSat is prepared to start transmitting data (transmit = TRUE)

```
// ================================================================================
// - CONFIGURARCIÓN E INICIALIZACIÓN MODE AFSK CON TRANSCEPTOR AX-5043
// ================================================================================
  fflush(stderr);
    if (mode == AFSK)
  {
  // Check for SPI and AX-5043 Digital Transceiver Board
    FILE * file = popen("sudo raspi-config nonint get_spi", "r");
    if (fgetc(file) == 48) {
      printf("SPI is enabled!\n"); // VERIFICACIÓN SOPORTE SPI
      FILE * file2 = popen("ls /dev/spidev0.* 2>&1", "r");
                printf("Result getc: %c \n", getc(file2));
      if (fgetc(file2) != 'l') {
        printf("SPI devices present!\n"); // VERIFICACIÓN DISPOSITIVOS SPI
        // CONFIGURACIÓN SPI:
        setSpiChannel(SPI_CHANNEL);
        setSpiSpeed(SPI_SPEED);
        initializeSpi();
        ax25_init( & hax25, (uint8_t * ) dest_addr, 11, (uint8_t * ) call, 11,
         AX25_PREAMBLE_LEN, AX25_POSTAMBLE_LEN);
        if (init_rf()) {
          printf("AX5043 successfully initialized!\n");
          ax5043 = TRUE;
          cw_id = OFF;
          printf("Mode AFSK with AX5043\n");
          transmit = TRUE;
        } else
          printf("AX5043 not present!\n");
          pclose(file2);
      }
    }
    pclose(file);
  }
// ================================================================================
```

Figure 31: AFSK definition and configuration.

FSK (Frequency Shift Keying):

It is a digital modulation technique which directly modulates radio frequency by shifting between two distinct frequencies to represent binary data. In this case 2-FSK (Binary Frequency Shift Keying) is used. a "1" is transmitted at one frequency, while a "0" is transmitted at another. This modulation method is simple, resistant to noise, and widely used for low-speed digital communications, typically at 1200 bps (bits per second) or much less. In 4-FSK, a more advanced modulation technique, four distinct frequencies are used to represent two bits per symbol, allowing higher speeds of at least 9600 bps. This would enable faster data transmission, but it requires more bandwidth and may be more susceptible to noise. In this case, as shown in Figure 31, the bit rate is 200bps. This is because in this type of application in which long communication distances are desired, speed is sacrificed to have a better SNR, quality of data transmitted and energy efficiency.

```
// ================================================================================
// - CONFIGURACIÓN MODO FSK
// ================================================================================

  if (mode == FSK) {
    bitRate = 200;
    rsFrames = 1;
    payloads = 1;
    rsFrameLen = 64;
    headerLen = 6;
    dataLen = 58;
    syncBits = 10;
    syncWord = 0b0011111010;
    parityLen = 32;
    amplitude = 32767 / 3;
    samples = S_RATE / bitRate;
    bufLen = (frameCnt * (syncBits + 10 * (headerLen + rsFrames *
                          (rsFrameLen + parityLen))) * samples);
    samplePeriod =  (int) (((float)((syncBits + 10 * (headerLen + rsFrames *
                    (rsFrameLen + parityLen)))) / (float) bitRate) * 1000 - 500);
    sleepTime = 0.1f;
        frameTime = ((float)((float)bufLen / (samples * frameCnt * bitRate)))
                                                  * 1000; // frame time in ms

    printf("\n FSK Mode, %d bits per frame, %d bits per second,
           %d ms per frame, %d ms sample period\n",
     bufLen / (samples * frameCnt), bitRate, frameTime, samplePeriod);
```

Figure 32: Configuration of FSK transmission mode.

BPSK (Binary Phase Shift Keying):

It is a phase modulation in which data is encoded by shifting the phase of the carrier signal between two states (0° and 180°). These two distinct phases represent binary data, with one phase (usually 0°) representing a binary "0" and the other (180°) representing a binary "1". BPSK is more spectrally efficient than other modulation schemes like FSK because it uses only two distinct phases to encode data. This results in a more efficient use of bandwidth, allowing higher data transmission rates. However, BPSK is more susceptible to noise and interferences than FSK. BPSK is commonly used for telemetry transmission. Software like FoxTelem plays a crucial role in receiving and decoding BPSK signals, facilitating the tracking and analysis of satellite data.

```
// ============================================================================
// - CONFIGURACIÓN MODO BPSK
// ============================================================================

  } else if (mode == BPSK) {
    bitRate = 1200;
    rsFrames = 3;
    payloads = 6;
    rsFrameLen = 159;
    headerLen = 8;
    dataLen = 78;
    syncBits = 31;
    syncWord = 0b10001111100110100100000101011101;
    parityLen = 32;
    amplitude = 32767;
    samples = S_RATE / bitRate;
    bufLen = (frameCnt * (syncBits + 10 * (headerLen + rsFrames *
                          (rsFrameLen + parityLen))) * samples);

       samplePeriod = ((float)((syncBits + 10 * (headerLen + rsFrames *
                (rsFrameLen + parityLen))))/(float)bitRate) * 1000 - 1800;
    sleepTime = 2.2f;
    frameTime = ((float)((float)bufLen / (samples * frameCnt * bitRate)))
                                          * 1000; // frame time in ms
    printf("\n BPSK Mode, bufLen: %d,  %d bits per frame, %d bits per second,
            %d ms per frame %d ms sample period\n",
      bufLen, bufLen / (samples * frameCnt), bitRate, frameTime, samplePeriod);

    sin_samples = S_RATE/freq_Hz;
    for (int j = 0; j < sin_samples; j++) {
      sin_map[j] = (short int)(amplitude * sin((float)(2 * M_PI * j / sin_samples)));
    }
    printf("\n");
  }
```

Figure 33: Configuration of BPSK transmission mode.

SSTV (Slow Scan Television):
Analog transmission mode that sends static images over radio frequencies in real-time from the CubeSat to ground stations. It is designed for low-bandwidth communications, sending a single image line by line, which is converted into an audio signal and can be decoded by software like MMSSTV (most recommended). Because it operates in the audio frequency range, SSTV signals can be transmitted using relatively low power and still be received over long distances.

CW (Continuous Wave):
Simplest and oldest form of communication, using on-off keying to send Morse Code, where different sequences of short and long pulses (dots and dashes) represent letters and numbers. Even though data rate is very low, it is highly efficient under weak signal conditions, and it is resistant to noise and interferences making it useful for satellite identification, beacon signals, and basic telemetry such as voltage and temperature readings. This makes CW an excellent choice for long-distance.

Each of these five modes will have a different impact on the sustainable mobility mission. Data collected will still be the same, however, their modulation process and transmission is different. Data transmission format and the consumption of power and other characteristics of the CubeSat will also depend on the mode

selection. Therefore, for some situations and environments one specific mode will be more adequate than the others. Along this project, it will be concluded that for telemetry data transmission the most effective modes are FSK and BPSK, which will be both used and compared in the mission.

3.3.3. Hardware and Communication protocols

Regarding the Hardware, the camera detection code is checked. Although the detection code is functional and the camera is properly detected, no images will be taken during this mission, as the focus is solely on telemetry data.

```
// CHECK FOR CAMERA
FILE * file4 = popen("vcgencmd get_camera", "r");
fgets(cmdbuffer, 1000, file4);
char camera_present[] = "supported=1 detected=1";
camera = (strstr( (const char *)& cmdbuffer, camera_present) != NULL) ? ON : OFF;
pclose(file4);

#ifdef DEBUG_LOGGING
printf("INFO: I2C bus status 0: %d 1: %d 3: %d camera: %d\n", i2c_bus0, i2c_bus1, i2c_bus3, camera);
#endif

FILE * file5 = popen("sudo rm /home/pi/CubeSatSim/camera_out.jpg > /dev/null 2>&1", "r");
file5 = popen("sudo rm /home/pi/CubeSatSim/camera_out.jpg.wav > /dev/null 2>&1", "r");
pclose(file5);
```

Figure 34: Camera detection code.

CubeSats, normally use more than one communication protocol, depending on how fast or complex (among other characteristics) a communication is needed to be. This Cubesatsim operates with three protocols: I2C, SPI and UART.

I2C (Inter-Integrated Circuit) is a communication protocol that allows sensors on the board to exchange data with the Raspberry Pi Zero. Each device on the I2C bus is assigned a unique address, enabling multiple sensors to communicate over the same two-wire interface (SDA and SCL). The main advantage of I2C protocol is the ability to interconnect different devices with minimal wiring, however, it is more susceptible to noise and has lower transmission speeds compared to other protocols.

Furthermore, in order to access, read and storage the information from sensors, I2C buses ought to be configured, defined and checked.

```
// ================================================================================
// - CONFIGURACIONES BUSES I2C
// ================================================================================

  config_file = fopen("sim.cfg", "w");
  fprintf(config_file, "%s %d %8.4f %8.4f %s %d %s %s %s", call,
    reset_count, lat_file, long_file, sim_yes, squelch, tx, rx, hab_yes);
  fclose(config_file);
  config_file = fopen("sim.cfg", "r");
  if (vB4) {
    map[BAT] = BUS;
    map[BUS] = BAT;
    snprintf(busStr, 10, "%d %d", i2c_bus1, test_i2c_bus(0));
  } else if (vB5) {
    map[MINUS_X] = MINUS_Y;
    map[PLUS_Z] = MINUS_X;
    map[MINUS_Y] = PLUS_Z;
    if (access("/dev/i2c-11", W_OK | R_OK) >= 0) { // Test if I2C Bus 11 is present
      printf("/dev/i2c-11 is present\n\n");
      snprintf(busStr, 10, "%d %d", test_i2c_bus(1), test_i2c_bus(11));
    } else {
      snprintf(busStr, 10, "%d %d", i2c_bus1, i2c_bus3);
    }
  } else {
    map[BUS] = MINUS_Z;
    map[BAT] = BUS;
    map[PLUS_Z] = BAT;
    map[MINUS_Z] = PLUS_Z;
    snprintf(busStr, 10, "%d %d", i2c_bus1, test_i2c_bus(0));
    voltageThreshold = 8.0;
  }
// ================================================================================
```

Figure 35: Configuration of the I2C buses.

```
// ================================================================================================
// - COMPROBACIÓN DE BUSES DISPONIBLES (1 Y 3) PARA LA POSTERIOR UTILIZACIÓN DE LOS SENSORES
// ================================================================================================

  printf("Test bus 1\n");
  fflush(stdout);
  i2c_bus1 = (test_i2c_bus(1) != -1) ? 1 : OFF;
  printf("Test bus 3\n");
  fflush(stdout);
  i2c_bus3 = (test_i2c_bus(3) != -1) ? 3 : OFF;
  printf("Finished testing\n");
  fflush(stdout);

// ================================================================================================
```

```
// ================================================================================================
// - VERIFY BUSES FUNCTIONALITY
// ================================================================================================
int test_i2c_bus(int bus)
{
  int output = bus; // return bus number if OK, otherwise return -1
  char busDev[20] = "/dev/i2c-";
  char busS[5];
  snprintf(busS, 5, "%d", bus);
  strcat (busDev, busS);
  printf("I2C Bus Tested: %s \n", busDev);

  if (access(busDev, W_OK | R_OK) >= 0)  {   // Test if I2C Bus is present
          char result[128];
          const char command_start[] = "timeout 2 i2cdetect -y ";  // was  5 10
   char command[50];
   strcpy (command, command_start); strcat (command, busS);
       FILE *i2cdetect = popen(command, "r");
       while (fgets(result, 128, i2cdetect) != NULL) {
         ; }
       int error = pclose(i2cdetect)/256;
       if (error != 0)
       {
         printf("ERROR: %s bus has a problem \n  Check I2C wiring and pullup resistors \n", busDev);
     if (bus == 3)
       printf("-> If this is a CubeSatSim Lite, then this error is normal!\n"); output = -1;
       }
  } else
  {     printf("ERROR: %s bus has a problem \n  Check software to see if I2C enabled \n", busDev);
   output = -1;
  }
  return(output); // return bus number or -1 if there is a problem with the bus
}
// ================================================================================================
```

Figure 36: Checking availability of the buses.

On the other hand, the SPI (Serial Peripheral Interface) protocol is a serial communication interface that uses multiple connection lines: MISO (Master In Slave Out), MOSI (Master Out Slave In), SCLK (Serial Clock), and CS (Chip Select). It is faster than I2C and is used when high data transfer speeds are required in the CubeSat. High speed sensors may need SPI protocol communication. SPI is used for AFSK (Audio Frequency Shift Keying) modulation via the AX-5043 transceiver.

Lastly, UART (Universal Asynchronous Receiver-Transmitter) protocol is an asynchronous communication interface that uses only two lines (TX for transmission and RX for reception). It is a reliable and simple communication method. Unlike I2C and SPI, UART is asynchronous, meaning it does not require a clock signal. While it is slower than SPI and typically supports only point-to-point

communication, it is widely used for radio frequency communication in CubeSats. For example, a 9600 baud UART connection is commonly used to send AT commands to a transceiver, enabling the CubeSat to transmit telemetry data to the ground station as in this case (see in Figure below)

```
// ============================================================================
// - STEM PAYLOAD BOARD CONNECTION
// ============================================================================
  // try connecting to STEM Payload board using UART
  if (!ax5043 && !vB3 && !(mode == CW) && !(mode == SSTV))
  // don't test for payload if AX5043 is present or CW or SSTV modes
  {
    payload = OFF;
    if ((uart_fd = serialOpen("/dev/ttyAMA0", 115200)) >= 0) {
      // was 9600 // SE COMUNICA CON LA STEM PAYLOAD
      printf("Serial opened to Pico\n");
      payload = ON;
    }
    else {
      fprintf(stderr, "Unable to open UART: %s\n -> Did you configure
        /boot/config.txt and /boot/cmdline.txt?\n", strerror(errno));
    }
  }

// ============================================================================
```

Figure 37: STEM Payload Board connection via UART.

Furthermore, UART communication is used too for transmitting data (independently of the transmission mode) from the CubeSat to the base station at 434.9MHz.

```
// ================================================================================================
// - CONFIGURACION MODULO DE RADIO DE TRANSMISION Y RECEPCION
// ================================================================================================
void program_radio() {
  printf("Programming FM module!\n");
  pinMode(28, OUTPUT);
  pinMode(29, OUTPUT);
  digitalWrite(29, HIGH);  // enable SR_FRS
  digitalWrite(28, HIGH);  // stop transmit
if ((uart_fd = serialOpen("/dev/ttyAMA0", 9600)) >= 0) {
  printf("serial opened 9600\n");
  for (int i = 0; i < 5; i++) {
     sleep(1); // delay(500);
     char uhf_string[] = "AT+DMOSETGROUP=0,435.0000,434.9000,0,3,0,0\r\n";
     char uhf_string1a[] = "AT+DMOSETGROUP=0,"; // changed frequency to verify
     char comma[] = ",";
     char uhf_string1b[] = ",0,"; // changed frequency to verify
     char uhf_string1[] = "AT+DMOSETGROUP=0,435.0000,434.9000,0,";  // changed frequency to verify
     char uhf_string2[] = ",0,0\r\n";
     char sq_string[2];
     sq_string[0] = '0' + squelch;
     sq_string[1] = 0;
  serialPrintf(uart_fd, uhf_string1a); serialPrintf(uart_fd, rx);
  serialPrintf(uart_fd, comma); serialPrintf(uart_fd, tx);
  serialPrintf(uart_fd, uhf_string1b);  serialPrintf(uart_fd, sq_string);
  serialPrintf(uart_fd, uhf_string2); sleep(1);
   char mic_string[] = "AT+DMOSETMIC=8,0\r\n";
   serialPrintf(uart_fd, mic_string);
  } }
 printf("Programming FM tx 434.9, rx on 435.0 MHz\n");
    digitalWrite(29, LOW);  // disable SR_FRS
  pinMode(28, INPUT);
  pinMode(29, INPUT);
  serialClose(uart_fd);
}
```

Figure 38: FM radio module configuration.

Firstly, after configuring the input and output PINs, the code attempts to open a serial port at 9600 baud. If successfully, then it starts programming radio module. The code sends AT commands to the radio module

3.3.4. Sensor data acquisition and Processing

In this subsection of the code, sensors will be read, extracting STEM payload board data, and then being processed. Image detection and processing will be explained too but not used.

Firstly, the code for the lecture of sensors will be shown:

```
// ================================================================================
// - LECTURA DE SENSORES E INICIALIZACIÓN DE VALORES
// ================================================================================
  if (((mode == FSK) || (mode == BPSK))) // && !sim_mode)
      get_tlm_fox();  // fill transmit buffer with reset count
      firstTime = 1;  // 0 packets that will be ignored
  {
    strcpy(pythonStr, pythonCmd); strcat(pythonStr, busStr);
    strcat(pythonConfigStr, pythonStr); strcat(pythonConfigStr, " c");
    fprintf(stderr, "pythonConfigStr: %s\n", pythonConfigStr);
    file1 = sopen(pythonConfigStr);  // python sensor polling function
    fgets(cmdbuffer, 1000, file1);
    fprintf(stderr, "pythonStr result: %s\n", cmdbuffer);
  }
  for (int i = 0; i < 9; i++) {
    voltage_min[i] = 1000.0; current_min[i] = 1000.0;
    voltage_max[i] = -1000.0; current_max[i] = -1000.0;
  }
  for (int i = 0; i < 17; i++) {
    sensor_min[i] = 1000.0; sensor_max[i] = -1000.0;
  }
  for (int i = 0; i < 3; i++) {
    other_min[i] = 1000.0; other_max[i] = -1000.0;
  }
  long int loopTime;
  loopTime = millis();
    while (loop-- != 0) {
    fflush(stdout); fflush(stderr);
    sensor_payload[0] = 0;
    memset(voltage, 0, sizeof(voltage)); memset(current, 0, sizeof(current));
    memset(sensor, 0, sizeof(sensor)); memset(other, 0, sizeof(other));
        FILE * uptime_file = fopen("/proc/uptime", "r");
    fscanf(uptime_file, "%f", & uptime_sec);

    uptime = (int) (uptime_sec + 0.5);
    printf("INFO: Reset Count: %d Uptime since Reset: %ld \n", reset_count, uptime);
    fclose(uptime_file);
    printf("++++ Loop time: %5.3f sec +++++\n", (millis() - loopTime)/1000.0);
    fflush(stdout); loopTime = millis();
  {
      int count1; char * token;
      fputc('\n', file1); fgets(cmdbuffer, 1000, file1);
      fprintf(stderr, "Python read Result: %s\n", cmdbuffer);
      const char space[2] = " ";
      token = strtok(cmdbuffer, space);
      for (count1 = 0; count1 < 8; count1++) {
        if (token != NULL) {
          voltage[count1] = (float) atof(token);
          #ifdef DEBUG_LOGGING
          #endif
          token = strtok(NULL, space);
          if (token != NULL) {
            current[count1] = (float) atof(token);
            if ((current[count1] < 0) && (current[count1] > -0.5))
              current[count1] *= (-1.0f);
            #ifdef DEBUG_LOGGING
            #endif
            token = strtok(NULL, space);
          }
        }
        if (voltage[map[BAT]] == 0.0)
    batteryVoltage = 4.5;
  else
    batteryVoltage = voltage[map[BAT]];
        batteryCurrent = current[map[BAT]];
    }

// ================================================================================
```

Figure 39: Sensor's data lecture and value's initialization.

In these lines of code, the objectives are to retrieve telemetry data if the system is in FSK or BPSK mode. Then a Python script to read sensor data is called. Also, another function this code does is initializing min/max values for voltage, current,

and other telemetry data, reading system uptime and tracking reset counts and finally, processing sensor data (voltage, current) and ensuring valid readings.

On the other hand, data from the STEM payload board is being extracted for a future processing and transmission.

```
// ============================================================================================
// - Extracción datos STEM Payload Board
// ============================================================================================
      if (payload == ON) {  // -55
        STEMBoardFailure = 0;
        printf("get_payload_status: %d \n",
          get_payload_serial(FALSE)); // RECUPERA DATOS DE TELEMETRÍA O DE ESTADO
  fflush(stdout); printf("String: %s\n", buffer2);
  fflush(stdout); strcpy(sensor_payload, buffer2);
  printf(" Response from STEM Payload board: %s\n", sensor_payload);
        if ((sensor_payload[0] == 'O') && (sensor_payload[1] == 'K'))
        // only process if valid payload response
         {int count1; char * token;
          const char space[2] = " "; token = strtok(sensor_payload, space);
         for (count1 = 0; count1 < 17; count1++) {
           if (token != NULL) {
            sensor[count1] = (float) atof(token);
              printf("sensor: %f ", sensor[count1]);  // LEE LOS DATOS DE LOS SENSORES
            token = strtok(NULL, space);
           } }
        printf("\n");
    if ((sensor[XS1] > -90.0) && (sensor[XS1] < 90.0) && (sensor[XS1] != 0.0))  {
    if (sensor[XS1] != latitude) {
      latitude = sensor[XS1];
      printf("Latitude updated to %f \n", latitude);
      newGpsTime = millis();
    }
    }
    if ((sensor[XS2] > -180.0) && (sensor[XS2] < 180.0) && (sensor[XS2] != 0.0))  {
    if (sensor[XS2] != longitude) {
      longitude = sensor[XS2];
      printf("Longitude updated to %f \n", longitude);
      newGpsTime = millis();
    }}}
```

```
else
  ; //payload = OFF;  // turn off since STEM Payload is not responding
    } if ((millis() - newGpsTime) > 60000) {
  longitude += rnd_float(-0.05, 0.05) / 100.0;  // was .05
     latitude += rnd_float(-0.05, 0.05) / 100.0;
       printf("GPS Location with Rnd: %f, %f \n", latitude, longitude);
       printf("GPS Location with Rnd: APRS %07.2f, %08.2f \n",
         toAprsFormat(latitude), toAprsFormat(longitude));
       newGpsTime = millis();
    } if ((sensor_payload[0] == 'O') && (sensor_payload[1] == 'K')) {
      for (int count1 = 0; count1 < 17; count1++) {
        if (sensor[count1] < sensor_min[count1])
          sensor_min[count1] = sensor[count1];
        if (sensor[count1] > sensor_max[count1])
          sensor_max[count1] = sensor[count1];
      }
    }
```

```
    #ifdef DEBUG_LOGGING
    fprintf(stderr, "INFO: Battery voltage: %5.2f V  Threshold %5.2f V Current: %6.1f mA Threshold: %6.1f mA\n",
      batteryVoltage, voltageThreshold, batteryCurrent, currentThreshold);
    #endif
    if ((batteryCurrent > currentThreshold) && (batteryVoltage < (voltageThreshold + 0.15)) && !sim_mode && !hab_mode)
    {
      fprintf(stderr,"Battery voltage low - switch to battery saver\n");
      if (battery_saver_mode == OFF)
        battery_saver(ON);
    } else if ((battery_saver_mode == ON) && (batteryCurrent < 0) && !sim_mode && !hab_mode)
    {
      fprintf(stderr,"Battery is being charged - switch battery saver off\n");
      if (battery_saver_mode == ON)
        battery_saver(OFF);
    }
    if ((batteryCurrent > currentThreshold) && (batteryVoltage < voltageThreshold) && !sim_mode && !hab_mode)
    // currentThreshold ensures that this won't happen when running on DC power.
    {
      fprintf(stderr, "Battery voltage too low: %f V - shutting down!\n", batteryVoltage);
      digitalWrite(txLed, txLedOff); digitalWrite(onLed, onLedOff);
      sleep(1); digitalWrite(onLed, onLedOn);
      sleep(1); digitalWrite(onLed, onLedOff);
      sleep(1); digitalWrite(onLed, onLedOn);
      sleep(1); digitalWrite(onLed, onLedOff);
      FILE * file6; // = popen("/home/pi/CubeSatSim/log > shutdown_log.txt", "r");
      file6 = popen("sudo shutdown -h now > /dev/null 2>&1", "r");
      pclose(file6); sleep(10);
    }
// ================================================================================================================
```

Figure 40: Data extraction from STEM Payload Board.

Briefly, what this code does is to retrieve telemetry data from the STEM Payload Board, processes sensor data, such as GPS coordinates, voltage, and current, update GPS coordinates, records min/max sensor values, manages battery power, enabling/disabling power-saving mode and shuts down the system safely if battery voltage is critically low.

Finally, the following code reads data from a connected camera to the battery board (connected to the raspberry pi zero) and extracts a packet (containing the photo taken). This code is based on starts and end flags. Finally, it stores the packet into a buffer, in which in other code this buffer will be accessed and transmitted to the base station.

```
// ================================================================================
// - IMAGE DETECTION AND PROCESSING
// ================================================================================
int get_payload_serial(int debug_camera)  {
  index1 = 0; flag_count = 0;
  start_flag_detected = FALSE; start_flag_complete = FALSE;
  end_flag_detected = FALSE; jpeg_start = 0;
 // #ifdef GET_IMAGE_DEBUG
 if (debug_camera)
  printf("Received from Payload:\n");
 // #endif
  finished = FALSE;
  unsigned long time_start = millis();
  while ((!finished) && ((millis() - time_start) < CAMERA_TIMEOUT)) {
     if (serialDataAvail(uart_fd)) {
       char octet = (char) serialGetchar(uart_fd);
            printf("%c", octet); fflush(stdout);
    if (start_flag_complete) {
      buffer2[index1++] = octet;
      if (octet == end_flag[flag_count]) {  // looking for end flag
          flag_count++;
#ifdef GET_IMAGE_DEBUG
//        if (debug_camera)
          printf("Found part of end flag!\n");
#endif
          if (flag_count >= strlen(end_flag)) {  // complete image
            index1 -= strlen(end_flag); buffer2[index1++] = 0;
       printf(" Payload length: %d \n",index1);
       finished = TRUE; index1 = 0;
            start_flag_complete = FALSE;
            start_flag_detected = FALSE; // get ready for next image
            end_flag_detected = FALSE; flag_count = 0;
          }
        } else {
         if (flag_count > 1)
              printf("Resetting. Not end flag.\n"); flag_count = 0;
        } //#ifdef GET_IMAGE_DEBUG
    if (debug_camera) {
          char hexValue[5];
          if (octet != 0x66) {
            sprintf(hexValue, "%02X", octet);
          } else {printf("66");}
    }
 //#endif
          if (index1 > 1000) {index1 = 0; printf("Resetting index!\n");}
    } else if (octet == start_flag[flag_count]) {  // looking for start flag
        start_flag_detected = TRUE; flag_count++;
 #ifdef GET_IMAGE_DEBUG
        printf("Found part of start flag! \n");
 #endif
        if (flag_count >= strlen(start_flag)) {
          flag_count = 0; start_flag_complete = TRUE;
 #ifdef GET_IMAGE_DEBUG
          printf("Found all of start flag!\n");
 #endif
        }
      } else {  // not the flag, keep looking
        start_flag_detected = FALSE; flag_count = 0;
 #ifdef GET_IMAGE_DEBUG
        printf("Resetting. Not start flag.\n");
 #endif
      }}}
  if (debug_camera)
     printf("\nComplete\n"); fflush(stdout); return(finished);
}
// ================================================================================
```

Figure 41: Image detection and processing.

3.3.5. Telemetry storage and Data transmission

This section is the responsible for sending all the data read and processed from the sensors. Firstly, two telemetry files are open. In the first one (*"telem_string.txt"*), battery voltage and current values are stored while in the second one (*"telem.txt"*), values such as sensors and status of the board are saved. Both of these information ought to be sent to the base station.

```
// ============================================================================================================
// - WRITING TELEMETRY ON TEXT FILE
// ============================================================================================================

    FILE * fp = fopen("/home/pi/CubeSatSim/telem_string.txt", "w");
    if (fp != NULL)  {
      printf("Writing telem_string.txt\n");
  if (batteryVoltage != 4.5)
        fprintf(fp, "BAT %4.2fV %5.1fmA\n", batteryVoltage, batteryCurrent);
  else
        fprintf(fp, "\n");  // don't show voltage and current if it isn't a sensor value

      fclose(fp);
    } else
      printf("Error writing to telem_string.txt\n");

// Open telemetry file with STEM Payload Data
telem_file = fopen("/home/pi/CubeSatSim/telem.txt", "a"); //in get_tlm() the file stores real telemetry
if (telem_file == NULL)
  printf("Error opening telem file\n");
fclose(telem_file);
printf("Opened telem file\n");
// ============================================================================================================
```

Figure 42: Writing telemetry code on text file.

Once the files are created and, as everything has been configured and ready to be transmitted, data must be sent to the base station.

```
// ============================================================================
// - TRANSMISIÓN TELEMETRÍA SEGÚN MODO DE OPERACIÓN
// ============================================================================
    if ((mode == AFSK) || (mode == CW)) {
      get_tlm(); sleep(25);
      fprintf(stderr, "INFO: Sleeping for 25 sec\n");
      int rand_sleep = (int)rnd_float(0.0, 5.0);
      sleep(rand_sleep);
      fprintf(stderr, "INFO: Sleeping for extra %d sec\n", rand_sleep);
    } else if ((mode == FSK) || (mode == BPSK)) {// FSK or BPSK
      get_tlm_fox();
    } else {          // SSTV
      fprintf(stderr, "Sleeping\n"); sleep(50);
    }
    #ifdef DEBUG_LOGGING
//    fprintf(stderr, "INFO: Getting ready to send\n");
    #endif
  }
  if (mode == BPSK) {
    #ifdef DEBUG_LOGGING
//    printf("Tx LED On 1\n");
    #endif
    printf("Sleeping to allow BPSK transmission to finish.\n");
    sleep((unsigned int)(loop_count * 5)); printf("Done sleeping\n");
    #ifdef DEBUG_LOGGING
//    printf("Tx LED Off\n");
    #endif
  } else if (mode == FSK) {
    printf("Sleeping to allow FSK transmission to finish.\n");
    sleep((unsigned int)loop_count); printf("Done sleeping\n");
  }
  return 0;
}
// ============================================================================
```

Figure 43: Sleeping sequencies for correct telemetry transmission.

As explained before and shown on the image above, this transmission will be different depending on the transmission mode selected. For AFSK and CW, the function *get_tlm()* is called and for FSK and BPSK *get_tlm_fox()* is called. Furthermore, sleeping sequences are created in order to synchronize the packets and avoid overlapping of data.

get_tlm():
This function is responsible for formatting, encoding and transmitting the CubeSat telemetry in different formats, primarily using the APRS (Automatic Packet Reporting System) protocol. This function ensures that data collected from the CubeSat's sensors is correctly structured and transmitted efficiently.

One of the main tasks of the function is converting raw sensor measurements (such as batteries and solar panels voltage and current) into a readable telemetry format.

```
void get_tlm(void) {
  FILE * txResult;
  for (int j = 0; j < frameCnt; j++) {
    fflush(stdout); fflush(stderr);
    int tlm[7][5]; memset(tlm, 0, sizeof tlm);
    tlm[1][A] = (int)(voltage[map[BUS]] / 15.0 + 0.5) % 100; // Current of 5V supply to Pi
    tlm[1][B] = (int)(99.5 - current[map[PLUS_X]] / 10.0) % 100; // +X current [4]
    tlm[1][C] = (int)(99.5 - current[map[MINUS_X]] / 10.0) % 100; // X- current [10]
    tlm[1][D] = (int)(99.5 - current[map[PLUS_Y]] / 10.0) % 100; // +Y current [7]
    tlm[2][A] = (int)(99.5 - current[map[MINUS_Y]] / 10.0) % 100; // -Y current [10]
    tlm[2][B] = (int)(99.5 - current[map[PLUS_Z]] / 10.0) % 100; // +Z current [10] // was 70/2m
    // transponder power, AO-7 didn't have a Z panel
    tlm[2][C] = (int)(99.5 - current[map[MINUS_Z]] / 10.0) % 100; // -Z current (was timestamp)
    tlm[2][D] = (int)(50.5 + current[map[BAT]] / 10.0) % 100; // NiMH Battery current
    if (voltage[map[BAT]] > 4.6)
      tlm[3][A] = (int)((voltage[map[BAT]] * 10.0) - 65.5) % 100;  // 7.0 - 10.0 V for old 9V battery
    else
      tlm[3][A] = (int)((voltage[map[BAT]] * 10.0) + 44.5) % 100;  // 0 - 4.5 V for new 3 cell battery
      tlm[3][B] = (int)(voltage[map[BUS]] * 10.0) % 100; // 5V supply to Pi
    tlm[4][A] = (int)((95.8 - other[IHU_TEMP]) / 1.48 + 0.5) % 100;
    // was [B] but didn't display in online TLM spreadsheet
    tlm[6][B] = 0;
    tlm[6][D] = 49 + rand() % 3;
```

Figure 44: Sensor's data converted to telemetry format.

Then, the following block of code is configuring the transmission header string that will be used in the communication, depending on the transmission protocol (AX5043 or RPITX) and the selected mode (AFSK or CW and the others). RPITX (Raspberry Pi as radio Transmitter) will be used for CW mode whereas for AFSK if AX5043 is present, then it will be implemented. If not, RPITX will be used for AFSK too.

```
char str[1000]; char tlm_str[1000];
char header_str[] = "\x03\xf0"; // hi hi ";
char header_str3[] = "echo '";
char header_str2[] = "-11>APCSS:";
char header_str2b[30]; // for APRS coordinates
char header_lat[10]; char header_long[10];
char header_str4[] = "hi hi ";
char footer_str1[] = "\' > t.txt";
char footer_str[] = " && echo 'AMSAT-11>APCSS:010101/hi hi ' >> t.txt && touch /home/pi/CubeSatSim/ready";
  // transmit is done by rpitx.py
char footer_str2[] = " && touch /home/pi/CubeSatSim/ready";
if (ax5043) {
  strcpy(str, header_str);
} else {strcpy(str, header_str3);}

  if (mode == AFSK) {
    strcat(str, call); strcat(str, header_str2);
  }
```

Figure 45: Transmission header string.

Depending on the mode selected, different data will be transmitted and converted into a specific format. For AFSK, APRS format will be used whereas for CW, MORSE format will be used.

```
if (mode != CW) { // AFSK (APRS)
  if (latitude > 0)
    sprintf(header_lat, "%07.2f%c", toAprsFormat(latitude), 'N'); // lat
  else
    sprintf(header_lat, "%07.2f%c", toAprsFormat(latitude) * (-1.0), 'S'); // lat
  if (longitude > 0)
    sprintf(header_long, "%08.2f%c", toAprsFormat(longitude) , 'E'); // long
  else
    sprintf(header_long, "%08.2f%c",toAprsFormat( longitude) * (-1.0), 'W'); // long

  if (ax5043)
    sprintf(header_str2b, "=%s%c%sShi hi ", header_lat, 0x5c, header_long); // add APRS lat and long
  else
if (hab_mode)
  sprintf(header_str2b, "=%s%c%sOhi hi ", header_lat, 0x2f, header_long); // add APRS lat and long with Balloon HAB icon
else
  sprintf(header_str2b, "=%s%c%c%sShi hi ", header_lat, 0x5c, 0x5c, header_long); // add APRS lat and long with Satellite icon
     printf("\n\nString is %s \n\n", header_str2b); strcat(str, header_str2b);
   } else {strcat(str, header_str4);}
printf("Str: %s \n", str);
```

```
} else {  // APRS
    sprintf(tlm_str, "BAT %4.2f %5.1f ", voltage[map[BAT]] , current[map[BAT]] );
    strcat(str, tlm_str);
  }
  strcpy(sensor_payload, buffer2);
  printf(" Response from STEM Payload board:: %s\n", sensor_payload);
  if (mode != CW) {
      strcat(str, sensor_payload); // append to telemetry string for transmission
  }
```

Figure 46: Converting data into APRS Format.

```
if (mode == CW) {
 int channel;
 for (channel = 1; channel < 7; channel++) {
   sprintf(tlm_str, "%d%d%d %d%d%d %d%d%d %d%d%d ",
     channel, upper_digit(tlm[channel][1]), lower_digit(tlm[channel][1]),
     channel, upper_digit(tlm[channel][2]), lower_digit(tlm[channel][2]),
     channel, upper_digit(tlm[channel][3]), lower_digit(tlm[channel][3]),
     channel, upper_digit(tlm[channel][4]), lower_digit(tlm[channel][4]));
     strcat(str, tlm_str);
 }
```

Figure 47: Converting data into CW format.

Now the transmission for the CW mode is done. The code creates a Morse Code string and uses RPITX to transmit it.

```
if (mode == CW) {
  char cw_str2[1000]; char cw_header2[] = "echo '";
  char cw_footer2[] = "' > id.txt && gen_packets -M 20 id.txt -o morse.wav -r 48000 > /dev/null 2>&1
   && cat morse.wav | csdr convert_i16_f | csdr gain_ff 7000 | csdr convert_f_samplerf
    20833 | sudo /home/pi/rpitx/rpitx -i- -m RF -f 434.897e3";
  char cw_footer3[] = "' > cw.txt && touch /home/pi/CubeSatSim/cwready";  // transmit is done by rpitx.py
  strcat(str, cw_footer3);
  printf("CW string to execute: %s\n", str);
  fflush(stdout);
  FILE * cw_file = popen(str, "r");
  pclose(cw_file);
  while ((cw_file = fopen("/home/pi/CubeSatSim/cwready", "r")) != NULL) {  // wait for rpitx  to be done
    fclose(cw_file); sleep(5);
  }
}
```

Figure 48: CW mode transmission.

However, if CW is not selected (AFSK) and AX5043 is present, AX.25 packets are sent via AX5043, as shown in the following image:

```
else if (ax5043) {
  digitalWrite(txLed, txLedOn);
  fprintf(stderr, "INFO: Transmitting X.25 packet using AX5043\n");
  memcpy(data, str, strnlen(str, 256));
  printf("data: %s \n", data);
  int ret = ax25_tx_frame( & hax25, & hax5043, data, strnlen(str, 256));
  if (ret) {
    fprintf(stderr,
      "ERROR: Failed to transmit AX.25 frame with error code %d\n",
      ret);
    exit(EXIT_FAILURE);
  }
  ax5043_wait_for_transmit(); digitalWrite(txLed, txLedOff);
  if (ret) {
    fprintf(stderr,
      "ERROR: Failed to transmit entire AX.25 frame with error code %d\n",
      ret);
    exit(EXIT_FAILURE);
  }
  sleep(4);  // was 2
```

Figure 49: Transmitting X.25 packet using AX5043.

However, if AX5043 is not present, RPITX is used for transmission:

```
  } else {  // APRS using rpitx
   if (payload == ON) {
    telem_file = fopen("/home/pi/CubeSatSim/telem.txt", "a");
    printf("Writing payload string\n");
    time_t timeStamp;
    time(&timeStamp);   // get timestamp
    char timeStampNoNl[31], bat_string[31];
    snprintf(timeStampNoNl, 30, "%.24s", ctime(&timeStamp));
    printf("TimeStamp: %s\n", timeStampNoNl);
    snprintf(bat_string, 30, "BAT %4.2f %5.1f", batteryVoltage, batteryCurrent);
    fprintf(telem_file, "%s %s %s\n", timeStampNoNl, bat_string, sensor_payload);  // write telemetry string to telem.txt file
    fclose(telem_file);
  }
    strcat(str, footer_str1);
    if (battery_saver_mode == ON)
      strcat(str, footer_str);  // add extra packet for rpitx transmission
    else
      strcat(str, footer_str2);
    fprintf(stderr, "String to execute: %s\n", str);
    printf("\n\nTelemetry string is %s \n\n", str);
    if (transmit) {
      FILE * file2 = popen(str, "r"); pclose(file2);
sleep(2); digitalWrite(txLed, txLedOff);
    } else {
      fprintf(stderr, "\nNo CubeSatSim Band Pass Filter detected.  No transmissions after the CW ID.\n");
    }
    sleep(3);
  }
```

Figure 50: AFSK (APRS) mode transmission using RPITX.

get_tlm_fox():

This function is similar to *get_tlm()* but, in this case, the function will execute when modes FSK or BPSK are selected. The reason for it is that when sending in those modes, a program called FoxTelem can decode the telemetry when transmitting in a specific format. This function firstly converts data into that format and then transmits it to the base station letting the program decode the telemetry.

Firstly, some variables are created in order to store data and make Boolean conditions on the code:

```
void get_tlm_fox() {
    int i;
    long int sync = syncWord;
    smaller = (int) (S_RATE / (2 * freq_Hz));
    short int b[dataLen]; short int b_max[dataLen]; short int b_min[dataLen];
      memset(b, 0, sizeof(b)); memset(b_max, 0, sizeof(b_max));
    memset(b_min, 0, sizeof(b_min)); short int h[headerLen];
    memset(h, 0, sizeof(h)); memset(buffer, 0xa5, sizeof(buffer));
    short int rs_frame[rsFrames][223];
    unsigned char parities[rsFrames][parityLen], inputByte;
    int id, frm_type = 0x01, NormalModeFailure = 0, groundCommandCount = 0;
    int PayloadFailure1 = 0, PayloadFailure2 = 0;
    int PSUVoltage = 0, PSUCurrent = 0, Resets = 0, Rssi = 2048;
    int batt_a_v = 0, batt_b_v = 0, batt_c_v = 0, battCurr = 0;
    int posXv = 0, negXv = 0, posYv = 0, negYv = 0, posZv = 0, negZv = 0;
    int posXi = 0, negXi = 0, posYi = 0, negYi = 0, posZi = 0, negZi = 0;
    int head_offset = 0; short int buffer_test[bufLen]; int buffSize;
    buffSize = (int) sizeof(buffer_test);
    if (mode == FSK)
      id = 7;
    else
      id = 0; // 99 in h[6]
```

Figure 51: Initialization of variables for the function.

Then, in order to transmit packets in regular periods of time, maintaining periodicity, a time mechanism is created. This is very useful to synchronize packets and receive them always on time.

```
for (int frames = 0; frames < frameCnt; frames++) {
  if (firstTime != ON) {
   int startSleep = millis();
   if ((millis() - sampleTime) < ((unsigned int)frameTime - 250))  // was 250 100 500 for FSK
      sleep(2.0); // 0.5);  // 25);  // initial period
   while ((millis() - sampleTime) < ((unsigned int)frameTime - 250))  // was 250 100
      sleep(0.1); // 25); // 0.5);  // 25);
    printf("Sleep period: %d\n", millis() - startSleep); fflush(stdout);
    sampleTime = (unsigned int) millis();
  } else
    printf("first time - no sleep\n");
```

Figure 52: Time mechanism for periodical transmission.

Then, the telemetry values read are saved into new buffers. A constant comparison is done to assign the maximum and minimum values of each parameter: voltage, current, accelerometer, temperature, pressure, etc. These last parameters are read from the arrays "*sensor" and "other"*.

```
for (int count1 = 0; count1 < 8; count1++) {
  if (voltage[count1] < voltage_min[count1])
    voltage_min[count1] = voltage[count1];
  if (current[count1] < current_min[count1])
    current_min[count1] = current[count1];

  if (voltage[count1] > voltage_max[count1])
    voltage_max[count1] = voltage[count1];
  if (current[count1] > current_max[count1])
    current_max[count1] = current[count1];
}
 for (int count1 = 0; count1 < 3; count1++) {
  if (other[count1] < other_min[count1])
    other_min[count1] = other[count1];
  if (other[count1] > other_max[count1])
    other_max[count1] = other[count1];
}
if (mode == FSK)
{
    if (loop % 32 == 0) {  // was 8
  printf("Sending MIN frame \n");
  frm_type = 0x03;
```

```
 for (int count1 = 0; count1 < 17; count1++) {
   if (count1 < 3)
     other[count1] = other_min[count1];
   if (count1 < 8) {
     voltage[count1] = voltage_min[count1];
     current[count1] = current_min[count1];
   }
   if (sensor_min[count1] != 1000.0) // make sure values are valid
     sensor[count1] = sensor_min[count1];
 }}
   if ((loop + 16) % 32 == 0) {  // was 8
 printf("Sending MAX frame \n");
 frm_type = 0x02;
 for (int count1 = 0; count1 < 17; count1++) {
   if (count1 < 3)
     other[count1] = other_max[count1];
   if (count1 < 8) {
     voltage[count1] = voltage_max[count1];
     current[count1] = current_max[count1];
   }
   if (sensor_max[count1] != -1000.0) // make sure values are valid
     sensor[count1] = sensor_max[count1];
 }}
else
    frm_type = 0x02;  // BPSK always send MAX MIN frame
```

```
sensor_payload[0] = 0;  // clear for next payload
  memset(rs_frame, 0, sizeof(rs_frame));
memset(parities, 0, sizeof(parities));
h[0] = (short int) ((h[0] & 0xf8) | (id & 0x07)); // 3 bits
 if (uptime != 0)   // if uptime is 0, leave reset count at 0
{
  h[0] = (short int) ((h[0] & 0x07) | ((reset_count & 0x1f) << 3));
  h[1] = (short int) ((reset_count >> 5) & 0xff);
  h[2] = (short int) ((h[2] & 0xf8) | ((reset_count >> 13) & 0x07))
}
h[2] = (short int) ((h[2] & 0x0e) | ((uptime & 0x1f) << 3));
h[3] = (short int) ((uptime >> 5) & 0xff);
h[4] = (short int) ((uptime >> 13) & 0xff);
h[5] = (short int) ((h[5] & 0xf0) | ((uptime >> 21) & 0x0f));
h[5] = (short int) ((h[5] & 0x0f) | (frm_type << 4));
if (mode == BPSK)
  h[6] = 99;
posXi = (int)(current[map[PLUS_X]] + 0.5) + 2048;
posYi = (int)(current[map[PLUS_Y]] + 0.5) + 2048;
posZi = (int)(current[map[PLUS_Z]] + 0.5) + 2048;
negXi = (int)(current[map[MINUS_X]] + 0.5) + 2048;
negYi = (int)(current[map[MINUS_Y]] + 0.5) + 2048;
negZi = (int)(current[map[MINUS_Z]] + 0.5) + 2048;

posXv = (int)(voltage[map[PLUS_X]] * 100);
posYv = (int)(voltage[map[PLUS_Y]] * 100);
posZv = (int)(voltage[map[PLUS_Z]] * 100);
negXv = (int)(voltage[map[MINUS_X]] * 100);
negYv = (int)(voltage[map[MINUS_Y]] * 100);
negZv = (int)(voltage[map[MINUS_Z]] * 100);

batt_c_v = (int)(voltage[map[BAT]] * 100);
battCurr = (int)(current[map[BAT]] + 0.5) + 2048;
PSUVoltage = (int)(voltage[map[BUS]] * 100);
PSUCurrent = (int)(current[map[BUS]] + 0.5) + 2048;
```

Figure 53: Sensor readings and assignment of maximum and minimum values.

After storing all data into different arrays, the encoding process is necessary to be done. The encodeA and encodeB functions encode 16-bit values into two 8-bit segments for efficient storage in an array. The encodeA splits the value, placing the lower 8 bits in the current index and the upper 4 bits in the next index, while encodeB reverses this order, storing the upper 12 bits first and the lower 8 bits in the next index. These functions are essential for compressing data, making them useful in memory, or bandwidth, limited such as this communication system.

```
// ============================================================================================
// - ENCODER
// ============================================================================================
int encodeA(short int  *b, int index, int val) { // Codifica un valor de 16 bits (val) en 12 bits
    b[index] = val & 0xff;
    b[index + 1] = (short int) ((b[index + 1] & 0xf0) | ((val >> 8) & 0x0f));
    return 0;
}

int encodeB(short int  *b, int index, int val) { // Sobrescribe mas ampliamente los datos en b
    b[index] =  (short int) ((b[index] & 0x0f)  |  ((val << 4) & 0xf0));
    b[index + 1] = (val >> 4 ) & 0xff;
    return 0;
}
// ============================================================================================
```

Figure 54: Encoder function.

Now the encoding process is done, as shown in the following images:

```
 if (payload == ON)
   STEMBoardFailure = 0;
 encodeA(b, 0 + head_offset, batt_a_v);
 encodeB(b, 1 + head_offset, batt_b_v);
 encodeA(b, 3 + head_offset, batt_c_v);

 encodeB(b, 4 + head_offset, (int)(sensor[ACCEL_X] * 100 + 0.5) + 2048); // Xaccel
 encodeA(b, 6 + head_offset, (int)(sensor[ACCEL_Y] * 100 + 0.5) + 2048); // Yaccel
 encodeB(b, 7 + head_offset, (int)(sensor[ACCEL_Z] * 100 + 0.5) + 2048); // Zaccel

 encodeA(b, 9 + head_offset, battCurr);
 encodeB(b, 10 + head_offset, (int)(sensor[TEMP] * 10 + 0.5)); // Temp
```

```
if (mode == FSK) {
  encodeA(b, 12 + head_offset, posXv);
  encodeB(b, 13 + head_offset, negXv);
  encodeA(b, 15 + head_offset, posYv);
  encodeB(b, 16 + head_offset, negYv);
  encodeA(b, 18 + head_offset, posZv);
  encodeB(b, 19 + head_offset, negZv);

  encodeA(b, 21 + head_offset, posXi);
  encodeB(b, 22 + head_offset, negXi);
  encodeA(b, 24 + head_offset, posYi);
  encodeB(b, 25 + head_offset, negYi);
  encodeA(b, 27 + head_offset, posZi);
  encodeB(b, 28 + head_offset, negZi);
} else // BPSK
{
  encodeA(b, 12 + head_offset, posXv);
  encodeB(b, 13 + head_offset, posYv);
  encodeA(b, 15 + head_offset, posZv);
  encodeB(b, 16 + head_offset, negXv);
  encodeA(b, 18 + head_offset, negYv);
  encodeB(b, 19 + head_offset, negZv);

  encodeA(b, 21 + head_offset, posXi);
  encodeB(b, 22 + head_offset, posYi);
  encodeA(b, 24 + head_offset, posZi);
  encodeB(b, 25 + head_offset, negXi);
  encodeA(b, 27 + head_offset, negYi);
  encodeB(b, 28 + head_offset, negZi);
```

```
//BPSK
encodeA(b_max, 12 + head_offset, (int)(voltage_max[map[PLUS_X]] * 100));
encodeB(b_max, 13 + head_offset, (int)(voltage_max[map[PLUS_Y]] * 100));
encodeA(b_max, 15 + head_offset, (int)(voltage_max[map[PLUS_Z]] * 100));
encodeB(b_max, 16 + head_offset, (int)(voltage_max[map[MINUS_X]] * 100));
encodeA(b_max, 18 + head_offset, (int)(voltage_max[map[MINUS_Y]] * 100));
encodeB(b_max, 19 + head_offset, (int)(voltage_max[map[MINUS_Z]] * 100));

encodeA(b_max, 21 + head_offset, (int)(current_max[map[PLUS_X]] + 0.5) + 2048);
encodeB(b_max, 22 + head_offset, (int)(current_max[map[PLUS_Y]] + 0.5) + 2048);
encodeA(b_max, 24 + head_offset, (int)(current_max[map[PLUS_Z]] + 0.5) + 2048);
encodeB(b_max, 25 + head_offset, (int)(current_max[map[MINUS_X]] + 0.5) + 2048);
encodeA(b_max, 27 + head_offset, (int)(current_max[map[MINUS_Y]] + 0.5) + 2048);
encodeB(b_max, 28 + head_offset, (int)(current_max[map[MINUS_Z]] + 0.5) + 2048);

encodeA(b_max, 9 + head_offset, (int)(current_max[map[BAT]] + 0.5) + 2048);
encodeA(b_max, 3 + head_offset, (int)(voltage_max[map[BAT]] * 100));
encodeA(b_max, 30 + head_offset, (int)(voltage_max[map[BUS]] * 100));
encodeB(b_max, 46 + head_offset, (int)(current_max[map[BUS]] + 0.5) + 2048);

encodeB(b_max, 37 + head_offset, (int)(other_max[RSSI] + 0.5) + 2048);
encodeA(b_max, 39 + head_offset, (int)(other_max[IHU_TEMP] * 10 + 0.5));
encodeB(b_max, 31 + head_offset, ((int)(other_max[SPIN] * 10)) + 2048);
```

```
if (sensor_min[0] != 1000.0) // make sure values are valid
{
    encodeB(b_max, 4 + head_offset, (int)(sensor_max[ACCEL_X] * 100 + 0.5) + 2048); // Xaccel
    encodeA(b_max, 6 + head_offset, (int)(sensor_max[ACCEL_Y] * 100 + 0.5) + 2048); // Yaccel
    encodeB(b_max, 7 + head_offset, (int)(sensor_max[ACCEL_Z] * 100 + 0.5) + 2048); // Zaccel

    encodeA(b_max, 33 + head_offset, (int)(sensor_max[PRES] + 0.5)); // Pressure
    encodeB(b_max, 34 + head_offset, (int)(sensor_max[ALT] * 10.0 + 0.5)); // Altitude
    encodeB(b_max, 40 + head_offset, (int)(sensor_max[GYRO_X] + 0.5) + 2048);
    encodeA(b_max, 42 + head_offset, (int)(sensor_max[GYRO_Y] + 0.5) + 2048);
    encodeB(b_max, 43 + head_offset, (int)(sensor_max[GYRO_Z] + 0.5) + 2048);

    encodeA(b_max, 48 + head_offset, (int)(sensor_max[XS1] * 10 + 0.5) + 2048);
    encodeB(b_max, 49 + head_offset, (int)(sensor_max[XS2] * 10 + 0.5) + 2048);
    encodeB(b_max, 10 + head_offset, (int)(sensor_max[TEMP] * 10 + 0.5));
    encodeA(b_max, 45 + head_offset, (int)(sensor_max[HUMI] * 10 + 0.5));
}
else
{
    encodeB(b_max, 4 + head_offset, 2048); // 0
    encodeA(b_max, 6 + head_offset, 2048); // 0
    encodeB(b_max, 7 + head_offset, 2048); // 0

    encodeB(b_max, 40 + head_offset, 2048);
    encodeA(b_max, 42 + head_offset, 2048);
    encodeB(b_max, 43 + head_offset, 2048);

    encodeA(b_max, 48 + head_offset, 2048);
    encodeB(b_max, 49 + head_offset, 2048);
}
```

```
encodeA(b_min, 12 + head_offset, (int)(voltage_min[map[PLUS_X]] * 100));
encodeB(b_min, 13 + head_offset, (int)(voltage_min[map[PLUS_Y]] * 100));
encodeA(b_min, 15 + head_offset, (int)(voltage_min[map[PLUS_Z]] * 100));
encodeB(b_min, 16 + head_offset, (int)(voltage_min[map[MINUS_X]] * 100));
encodeA(b_min, 18 + head_offset, (int)(voltage_min[map[MINUS_Y]] * 100));
encodeB(b_min, 19 + head_offset, (int)(voltage_min[map[MINUS_Z]] * 100));

encodeA(b_min, 21 + head_offset, (int)(current_min[map[PLUS_X]] + 0.5) + 2048);
encodeB(b_min, 22 + head_offset, (int)(current_min[map[PLUS_Y]] + 0.5) + 2048);
encodeA(b_min, 24 + head_offset, (int)(current_min[map[PLUS_Z]] + 0.5) + 2048);
encodeB(b_min, 25 + head_offset, (int)(current_min[map[MINUS_X]] + 0.5) + 2048);
encodeA(b_min, 27 + head_offset, (int)(current_min[map[MINUS_Y]] + 0.5) + 2048);
encodeB(b_min, 28 + head_offset, (int)(current_min[map[MINUS_Z]] + 0.5) + 2048);

encodeA(b_min, 9 + head_offset, (int)(current_min[map[BAT]] + 0.5) + 2048);
encodeA(b_min, 3 + head_offset, (int)(voltage_min[map[BAT]] * 100));
encodeA(b_min, 30 + head_offset, (int)(voltage_min[map[BUS]] * 100));
encodeB(b_min, 46 + head_offset, (int)(current_min[map[BUS]] + 0.5) + 2048);

encodeB(b_min, 31 + head_offset, ((int)(other_min[SPIN] * 10)) + 2048);
encodeB(b_min, 37 + head_offset, (int)(other_min[RSSI] + 0.5) + 2048);
encodeA(b_min, 39 + head_offset, (int)(other_min[IHU_TEMP] * 10 + 0.5));
```

```
  if (sensor_min[0] != 1000.0) // make sure values are valid -- BPSK
  {
      encodeB(b_min, 4 + head_offset, (int)(sensor_min[ACCEL_X] * 100 + 0.5) + 2048); // Xaccel
      encodeA(b_min, 6 + head_offset, (int)(sensor_min[ACCEL_Y] * 100 + 0.5) + 2048); // Yaccel
      encodeB(b_min, 7 + head_offset, (int)(sensor_min[ACCEL_Z] * 100 + 0.5) + 2048); // Zaccel

      encodeA(b_min, 33 + head_offset, (int)(sensor_min[PRES] + 0.5)); // Pressure
      encodeB(b_min, 34 + head_offset, (int)(sensor_min[ALT] * 10.0 + 0.5)); // Altitude
      encodeB(b_min, 40 + head_offset, (int)(sensor_min[GYRO_X] + 0.5) + 2048);
      encodeA(b_min, 42 + head_offset, (int)(sensor_min[GYRO_Y] + 0.5) + 2048);
      encodeB(b_min, 43 + head_offset, (int)(sensor_min[GYRO_Z] + 0.5) + 2048);

      encodeA(b_min, 48 + head_offset, (int)(sensor_min[XS1] * 10 + 0.5) + 2048);
      encodeB(b_min, 49 + head_offset, (int)(sensor_min[XS2] * 10 + 0.5) + 2048);
      encodeB(b_min, 10 + head_offset, (int)(sensor_min[TEMP] * 10 + 0.5));
      encodeA(b_min, 45 + head_offset, (int)(sensor_min[HUMI] * 10 + 0.5));
  }
  else
  {
      encodeB(b_min, 4 + head_offset, 2048); // 0
      encodeA(b_min, 6 + head_offset, 2048); // 0
      encodeB(b_min, 7 + head_offset, 2048); // 0

      encodeB(b_min, 40 + head_offset, 2048);
      encodeA(b_min, 42 + head_offset, 2048);
      encodeB(b_min, 43 + head_offset, 2048);

      encodeA(b_min, 48 + head_offset, 2048);
      encodeB(b_min, 49 + head_offset, 2048);
  }
}
```

```
encodeA(b, 30 + head_offset, PSUVoltage);

encodeB(b, 31 + head_offset, ((int)(other[SPIN] * 10)) + 2048);

encodeA(b, 33 + head_offset, (int)(sensor[PRES] + 0.5)); // Pressure
encodeB(b, 34 + head_offset, (int)(sensor[ALT] * 10.0 + 0.5)); // Altitude

encodeA(b, 36 + head_offset, Resets);
encodeB(b, 37 + head_offset, (int)(other[RSSI] + 0.5) + 2048);

encodeA(b, 39 + head_offset, (int)(other[IHU_TEMP] * 10 + 0.5));

encodeB(b, 40 + head_offset, (int)(sensor[GYRO_X] + 0.5) + 2048);
encodeA(b, 42 + head_offset, (int)(sensor[GYRO_Y] + 0.5) + 2048);
encodeB(b, 43 + head_offset, (int)(sensor[GYRO_Z] + 0.5) + 2048);

encodeA(b, 45 + head_offset, (int)(sensor[HUMI] * 10 + 0.5)); // in place of sensor1

encodeB(b, 46 + head_offset, PSUCurrent);
encodeA(b, 48 + head_offset, (int)(sensor[XS1] * 10 + 0.5) + 2048);
encodeB(b, 49 + head_offset, (int)(sensor[XS2] * 10 + 0.5) + 2048);
```

Figure 55: Encoding process of data.

As can be observed from Figure 53, depending on the mode selection, the encoding process is very different. After encoding, a text file, which stores the number of commands sent from the ground station, is opened in a reading view. If the file is available, its contents are read and converted into an integer assigning the result to a defined variable. If the file cannot be opened, an error message is displayed in the console. Then, the code verifies the deployment status of the transmission (txAntennaDeployed) and reception (rxAntennaDeployed) antennas, leaving a message when deployed.

```
FILE * command_count_file = fopen("/home/pi/CubeSatSim/command_count.txt", "r");
if (command_count_file != NULL) {
  char count_string[10];
  if ( (fgets(count_string, 10, command_count_file)) != NULL)
   groundCommandCount = atoi(count_string);
  fclose(command_count_file);
} else
    printf("Error opening command_count.txt!\n");
printf("Command count: %d\n", groundCommandCount);
int status = STEMBoardFailure + SafeMode * 2 + sim_mode * 4 + PayloadFailure1 * 8 +
  (i2c_bus0 == OFF) * 16 + (i2c_bus1 == OFF) * 32 + (i2c_bus3 == OFF) * 64 + (camera == OFF) * 128 + groundCommandCount * 256;
encodeA(b, 51 + head_offset, status);
encodeB(b, 52 + head_offset, rxAntennaDeployed + txAntennaDeployed * 2);
if (txAntennaDeployed == 0) {
  txAntennaDeployed = 1;
  printf("TX Antenna Deployed!\n");
}
if (rxAntennaDeployed == 0) {
  rxAntennaDeployed = 1;
  printf("RX Antenna Deployed!\n");
}
```

Figure 56: Command count calculation and antenna deployment management.

Next, before sending all data telemetry, two more encodings are done, but with different purpose.

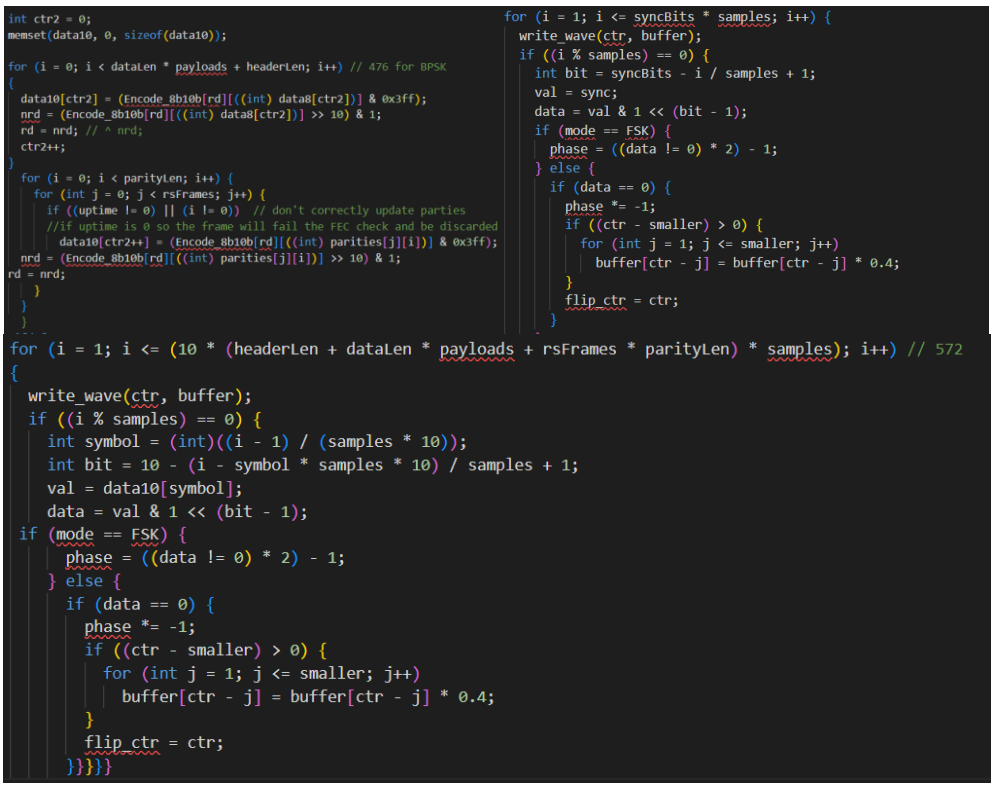
```
short int data10[headerLen + rsFrames * (rsFrameLen + parityLen)];
short int data8[headerLen + rsFrames * (rsFrameLen + parityLen)];
int ctr1 = 0; int ctr3 = 0;
for (i = 0; i < rsFrameLen; i++) {
  for (int j = 0; j < rsFrames; j++) {
    if (!((i == (rsFrameLen - 1)) && (j == 2))){ // skip last one for BPSK
      if (ctr1 < headerLen) {
        rs_frame[j][i] = h[ctr1]; update_rs(parities[j], h[ctr1]);
        data8[ctr1++] = rs_frame[j][i];
      } else {
     if (mode == FSK)
     {
        rs_frame[j][i] = b[ctr3 % dataLen]; update_rs(parities[j], b[ctr3 % dataLen]);
     }  else // BPSK
    if ((int)(ctr3/dataLen) == 3)
    {
        rs_frame[j][i] = b_max[ctr3 % dataLen]; update_rs(parities[j], b_max[ctr3 % dataLen]);
    } else if ((int)(ctr3/dataLen) == 4)
    {
        rs_frame[j][i] = b_min[ctr3 % dataLen]; update_rs(parities[j], b_min[ctr3 % dataLen]);
    } else
    {
        rs_frame[j][i] = b[ctr3 % dataLen]; update_rs(parities[j], b[ctr3 % dataLen]);
    }
        data8[ctr1++] = rs_frame[j][i]; ctr3++;
      }}}}
```

Figure 57: Reed-Solomon encoding.

In the image above, Reed-Solomon encoding has been applied, enabling error correction by adding parity to the data. Depending on the selected modulation

mode (FSK or BPSK), different data assignments are made. For FSK, data is taken from an array “b”, while for BPSK, values from the “b_max” and “b_min” arrays are selected, adjusting the data according to phase modulation. Reed-Salomon encoding allows the detection and correction of errors that may occur during transmission due to noise or distortion in the communication channel.

Once the data is organized and protected with Reed-Solomon, the next step is to apply 8b/10b encoding, which transforms 8-bit blocks into 10-bit blocks. This technique is very important for maintaining proper synchronization in transmission and ensuring that the signal has a constant rate of transitions between 0 and 1 values, which is essential for the correct reception and recovery of the signal. Additionally, 8b/10b encoding helps improve signal integrity and facilitates its transmission through communication channels.



```
int ctr2 = 0;
memset(data10, 0, sizeof(data10));

for (i = 0; i < dataLen * payloads + headerLen; i++) // 476 for BPSK
{
  data10[ctr2] = (Encode_8b10b[rd][((int) data8[ctr2])] & 0x3ff);
  nrd = (Encode_8b10b[rd][((int) data8[ctr2])] >> 10) & 1;
  rd = nrd; // ^ nrd;
  ctr2++;
}
  for (i = 0; i < parityLen; i++) {
    for (int j = 0; j < rsFrames; j++) {
      if ((uptime != 0) || (i != 0))  // don't correctly update parties
      //if uptime is 0 so the frame will fail the FEC check and be discarded
        data10[ctr2++] = (Encode_8b10b[rd][((int) parities[j][i])] & 0x3ff);
  nrd = (Encode_8b10b[rd][((int) parities[j][i])] >> 10) & 1;
rd = nrd;
    }
  }
  }
```

```
for (i = 1; i <= syncBits * samples; i++) {
  write_wave(ctr, buffer);
  if ((i % samples) == 0) {
    int bit = syncBits - i / samples + 1;
    val = sync;
    data = val & 1 << (bit - 1);
    if (mode == FSK) {
      phase = ((data != 0) * 2) - 1;
    } else {
      if (data == 0) {
        phase *= -1;
        if ((ctr - smaller) > 0) {
          for (int j = 1; j <= smaller; j++)
            buffer[ctr - j] = buffer[ctr - j] * 0.4;
        }
        flip_ctr = ctr;
      }
```

```
for (i = 1; i <= (10 * (headerLen + dataLen * payloads + rsFrames * parityLen) * samples); i++) // 572
{
  write_wave(ctr, buffer);
  if ((i % samples) == 0) {
    int symbol = (int)((i - 1) / (samples * 10));
    int bit = 10 - (i - symbol * samples * 10) / samples + 1;
    val = data10[symbol];
    data = val & 1 << (bit - 1);
  if (mode == FSK) {
      phase = ((data != 0) * 2) - 1;
    } else {
      if (data == 0) {
        phase *= -1;
        if ((ctr - smaller) > 0) {
          for (int j = 1; j <= smaller; j++)
            buffer[ctr - j] = buffer[ctr - j] * 0.4;
        }
        flip_ctr = ctr;
      }}}}}
```

Figure 58: 8b/10b encoding.

The code shown above calls a function called *“write_wave”*, which creates modulated waves that contain the information desired to be transmitted. These waves are the physical form that travels in the air from the CubeSat to the base station.

```
// ============================================================================================
// - CREATION OF SINE WAVES WITH MODULATIONS ACCORDING TO MODE AND RANGE CONDITIONS
// ============================================================================================

void write_wave(int i, short int *buffer)
{
    if (mode == FSK)
    {
      if ((ctr - flip_ctr) < smaller)
        buffer[ctr++] = (short int)(0.1 * phase * (ctr - flip_ctr) / smaller);
      else
        buffer[ctr++] = (short int)(0.25 * amplitude * phase);
    }
    else
    {
      if ((ctr - flip_ctr) < smaller)
            buffer[ctr++] = (short int)(phase * sin_map[ctr % sin_samples] / 2);
    else
            buffer[ctr++] = (short int)(phase * sin_map[ctr % sin_samples]);     }
}
// ============================================================================================
```

Figure 59: Modulated sine waves creation.

The last part of *get_tlm_fox()* is to establish a TCP socket connection and transmit data over it, ensuring data is reliably sent.

The process begins by checking if the socket is already open. If it is not and transmission is enabled, the program attempts to create a TCP socket. It then sets up a connection to the localhost (127.0.0.1) using a specified port. If the connection is successful, the socket is marked as open, allowing data transmission to proceed. Once the connection is established, the program sends a buffer of data over the socket using the *send()* function. If a transmission attempt fails, the socket connection is closed, preventing further communication. Furthermore, if transmission is disabled, a message is displayed, indicating that no data is being sent. Finally, if the socket remains open after a successful transmission, the program updates its state to indicate that the initial setup phase is complete.

```
int error = 0;
if (!socket_open && transmit) {
  printf("Opening socket!\n"); struct sockaddr_in serv_addr;
   if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    printf("\n Socket creation error \n"); error = 1;
  }
  memset( & serv_addr, '0', sizeof(serv_addr));
  serv_addr.sin_family = AF_INET; serv_addr.sin_port = htons(PORT);
  if (inet_pton(AF_INET, "127.0.0.1", & serv_addr.sin_addr) <= 0) {
    printf("\nInvalid address/ Address not supported \n"); error = 1;
  }
  if (connect(sock, (struct sockaddr * ) & serv_addr, sizeof(serv_addr)) < 0) {
    printf("\nConnection Failed \n"); printf("Error: %s \n", strerror(errno));
    error = 1; sleep(2.0);  // sleep if socket connection refused
    error = 0;
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
      printf("\n Socket creation error \n"); error = 1;
    }
    memset( & serv_addr, '0', sizeof(serv_addr));
    serv_addr.sin_family = AF_INET; serv_addr.sin_port = htons(PORT);
    // Convert IPv4 and IPv6 addresses from text to binary form
    if (inet_pton(AF_INET, "127.0.0.1", & serv_addr.sin_addr) <= 0) {
      printf("\nInvalid address/ Address not supported \n"); error = 1;
    }
    if (connect(sock, (struct sockaddr * ) & serv_addr, sizeof(serv_addr)) < 0) {
      printf("\nConnection Failed \n"); printf("Error: %s \n", strerror(errno)); error = 1;
    }}
  if (error == 1)
  ; //rpitxStatus = -1;
  else {socket_open = 1;}
}
```

```
 if (!error && transmit) {
   start = millis();
   int sock_ret = send(sock, buffer, (unsigned int)(ctr * 2 + 2), 0);
   printf("socket send 1 %d ms bytes: %d \n\n", (unsigned int)millis() - start, sock_ret);
   fflush(stdout);
   if (sock_ret < (ctr * 2 + 2)) {
     sleep(0.5); sock_ret = send(sock, &buffer[sock_ret], (unsigned int)(ctr * 2 + 2 - sock_ret), 0);
     printf("socket send 2 %d ms bytes: %d \n\n", millis() - start, sock_ret);
   } loop_count++;
   if ((firstTime == 1) || (((loop_count % 180) == 0) && (mode == FSK)) || (((loop_count % 80) == 0) && (mode == BPSK)))
   // do first time and was every 180 samples
   { int max;
     if (mode == FSK)
        if (sim_mode)
            max = 6;
           else if (firstTime == 1) {max = 4;}  // 5; was 6
           else {max = 3;}
     else
         if (firstTime == 1) {max = 5;}  // 5; was 6
           else {max = 4;}
     for (int times = 0; times < max; times++)
     {
        start = millis();  // send frame until buffer fills
        sock_ret = send(sock, buffer, (unsigned int)(ctr * 2 + 2), 0);
        printf("socket send %d in %d ms bytes: %d \n\n",times + 2, (unsigned int)millis() - start, sock_ret);
        if ((millis() - start) > 500) {printf("Buffer over filled!\n");break;}
        if (sock_ret < (ctr * 2 + 2)) {
      sleep(0.5); sock_ret = send(sock, &buffer[sock_ret], (unsigned int)(ctr * 2 + 2 - sock_ret), 0);
      printf("socket resend %d in %d ms bytes: %d \n\n",times, millis() - start, sock_ret);}
     }
     sampleTime = (unsigned int) millis(); // resetting time for sleeping
     fflush(stdout);  }
   if (sock_ret == -1) {printf("Error: %s \n", strerror(errno));socket_open = 0;}
 } if (!transmit) {fprintf(stderr, "\nNo CubeSatSim Band Pass Filter detected.  No transmissions after the CW ID.\n");}
 if (socket_open == 1) {firstTime = 0;}
 return; }
```

Figure 60: TCP Socket Communication for Data Transmission.

Finally, as observed, *get_tlm_fox()* is a more complex and specific function than *get_tlm()*. The *get_tlm_fox()* function is a specialized version designed for decoding

programs such as FoxTelem, incorporating more detailed data including solar panel voltages and currents, sensors like accelerometers and gyroscopes, as well as environmental parameters such as pressure, humidity, temperature, and altitude. Furthermore, both functions are used when different transmission modes are selected, as explained in this whole subsection. All these differences make this last function more effective when detailed telemetry and reliable data transmission are essential, as needed for this mobility mission.

3.3.6. Power management

The final component of the CubeSatSim code is power management. As shown in previous figures of the software code, the CubeSatSim includes a "safe mode." When activated, this mode reduces energy consumption by shutting down non-essential tasks, ensuring that only critical and necessary components remain active. Safe mode is triggered when the system detects that the battery level has reached a predefined low-energy threshold (see Figure 59).

```
if (sim_mode) { // simulated telemetry
  if (batt < 3.0) {
    batt = 3.0;
    SafeMode = 1;
    printf("Safe Mode!\n");
  } else
    SafeMode= 0;

  if (batt > 4.5)
    batt = 4.5;

  voltage[map[BAT]] = batt + rnd_float(-0.01, 0.01);

  // end of simulated telemetry
}
else {

  if (batteryVoltage < 3.7) {
    SafeMode = 1;
    printf("Safe Mode!\n");
  } else
    SafeMode = 0;
```

Figure 61: Safe mode activation in simulated telemetry and no simulated.

This mode relies on several functions that continuously monitor the system's energy levels to determine whether safe mode should be activated. This proactive approach helps prevent unexpected shutdowns caused by low battery levels.

```
// ================================================================================================
// - SAFE MODE CONFIGURATION
// ================================================================================================
int battery_saver_check() { // Verifica si esta on o off el ahorro de bateria
  FILE *file = fopen("/home/pi/CubeSatSim/battery_saver", "r"); // se crea cuando se activa el modo ahorro
  if (file == NULL) {fprintf(stderr,"Battery saver mode is OFF!\n");  return(OFF);}
  fclose(file);
  fprintf(stderr,"Battery saver mode is ON!\n"); return(ON);
}
void battery_saver(int setting) { // Activa o apaga el modo ahorro
if (setting == ON) {
  if ((mode == AFSK) || (mode == SSTV) || (mode == CW)) {
    if (battery_saver_check() == OFF) {
      FILE *command = popen("touch /home/pi/CubeSatSim/battery_saver", "r");
        pclose(command);
      fprintf(stderr,"Turning Battery saver mode ON\n");
      command = popen("sudo reboot now", "r");
      pclose(command); sleep(60); return;
    } else
      fprintf(stderr, "Nothing to do for battery_saver\n");
  }
  } else if (setting == OFF) {
  if ((mode == AFSK) || (mode == SSTV) || (mode == CW)) {
    if (battery_saver_check() == ON) {
      FILE *command = popen("rm /home/pi/CubeSatSim/battery_saver", "r");
        pclose(command);
      fprintf(stderr,"Turning Battery saver mode OFF\n");
      command = popen("sudo reboot now", "r");
        pclose(command); sleep(60); return;
    } else
      fprintf(stderr, "Nothing to do for battery_saver\n");
  }
  } else {fprintf(stderr,"battery_saver function error");return;}
  return;
}
// ================================================================================================
```

Figure 62: Safe mode checking and activation functions configuration.

On the picture above, the safe mode configuration is shown. Firstly, a function tries to open and read a file. If it exists the output of the function is ON (meaning that the safe mode is activated) otherwise it returns an OFF. Then, another function is created to turn on or off the safe mode depending on a parameter. When transmitting in AFSK, SSTV or CW, it checks the state of the mode and then changes it depending on what is needed.

3.4.SSH

In order to make changes or add something new to the CubeSat Sim's software, the raspberry files have to be accessed and modified. In order to do so, the best and easiest way is to connect the base station computer to the raspberry pi via SSH (Secure Shell).

It is a network protocol that allows secure remote access to a device (raspberry in this case), over an untrusted network like the Internet. It is mainly used for secure file transfers, files modification and to properly access the device. In order to connect to the raspberry via SSH, the CubeSat needs to be operative and the computer base command prompt (CMD) executed. The connection can be made in two different ways: with the IP address or with the local host name, both with password authentication.

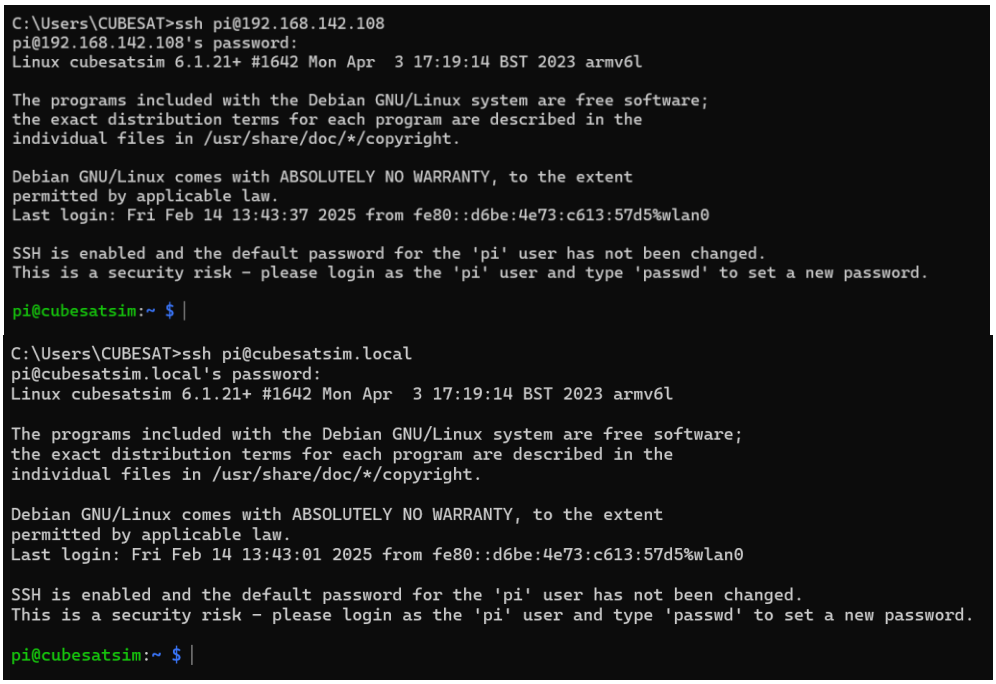


Figure 63: SSH connection established with the Raspberry Pi.

Once the command: *ssh pi@IP-Address* or *ssh pi@local-host-name.local* is executed, the raspberry pi's password must be introduced. If data is correct, the base station will have access to the raspberry pi, as shown on the image above. Once the connection is stablished a great variety of things such as: modify, add or remove files to the software, add code and modify it for different interests, change CubeSatSim real-time status, change modes of operation and transmission of the CubeSatSim, enable or disable safe mode, simulated transmission, HAB mode… and much more can be made.

In this section, the most important and used SSH command options are going to be explained so as to optimize CubeSatSim's efficiency. Firstly, the files inside the CubeSatSim are shown:

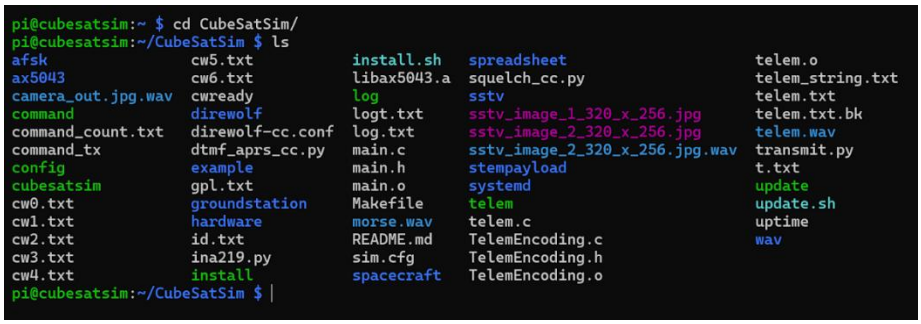


Figure 64: CubeSatSim raspberry pi's files.

In those files, there are some of them important for this project. As it has been already explained in the code, in the "telem…txt" files the telemetry read from the sensors, solar panels and batteries is stored. The "log" files have the history of the

execution, the commands executed on the CubeSatSim's software. The folder "config" stores the configuration of the CubeSat and a menu for changing the state and modes of the CubeSat (explained in detail below). "Sim.cfg" stores the initial configuration of the simulated telemetry CubeSatSim's parameters, as explained in the Code section.

3.4.1. Config Menu

When executing the *CubeSatSim/config -h* command in a generic route path, a menu created in the config folder with different options will appear. When executing those options, the CubeSatSim will experience either changes in the software or in its operating mode.

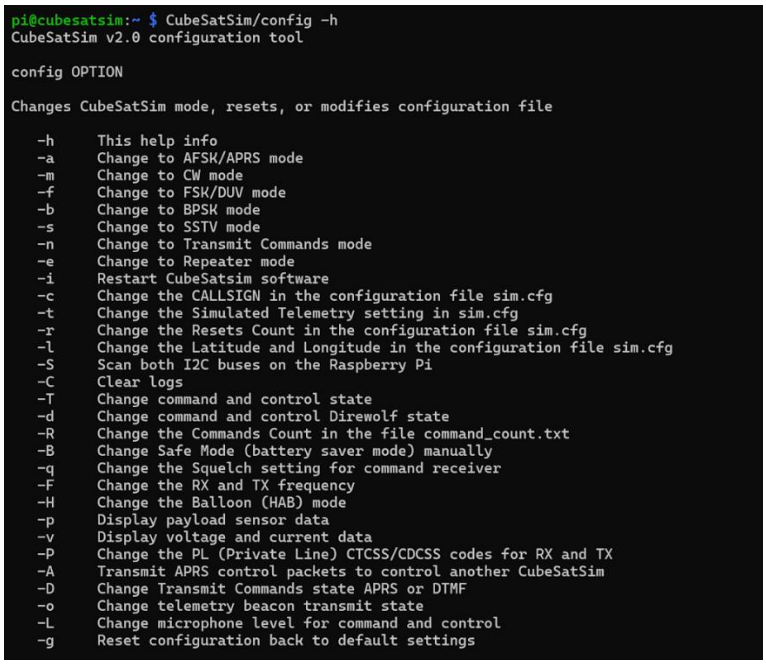


Figure 65: Configuration file options.

The first commands, from "-a" to "-s" are for changing the operation mode of the CubeSatSim. This is very useful for the mobility mission because this mode of operation can be changed wirelessly via SSH without needing to have near the physical button of the CubeSat.

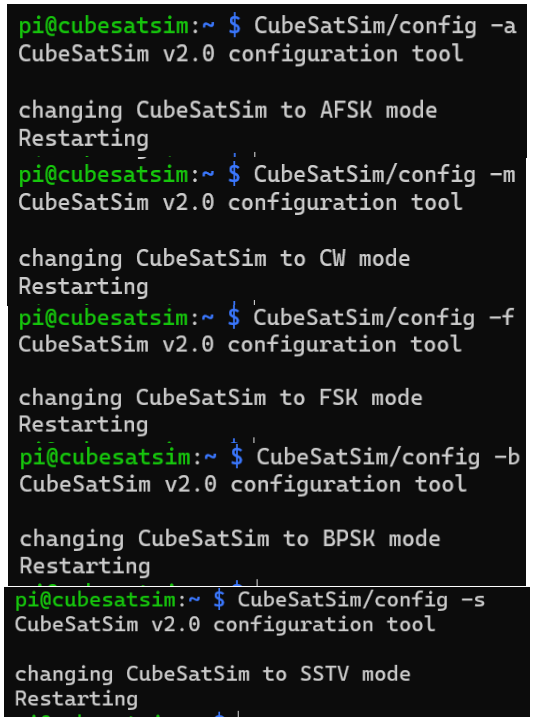
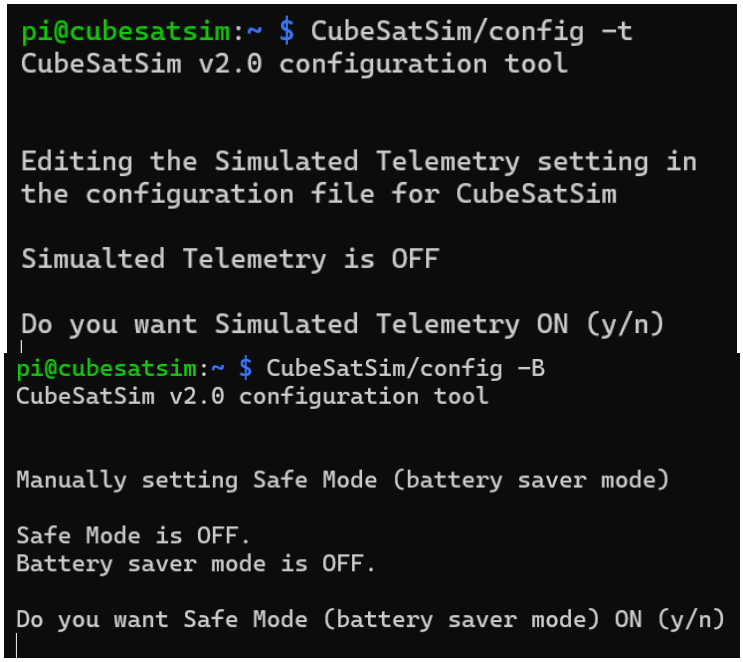


Figure 66: Changing CubeSatSim's modes of operation.

Then the "-t", "-B" and "-H" commands are used for enabling or disabling the simulated telemetry, safe mode and HAB mode respectively.



```
pi@cubesatsim:~ $ CubeSatSim/config -H
CubeSatSim v2.0 configuration tool

Editing the Balloon mode setting in
the configuration file for CubeSatSim

Balloon mode is OFF

Do you want Balloon mode ON (y/n)
y

Balloon mode is ON

CubeSatSim configuration sim.cfg file updated to:

AMSAT 56 43.2909 -1.9834 no 3 434.9000 435.0000 yes 0 0


Broadcast message from pi@cubesatsim (pts/0) (Fri Feb 14 13:13:36 2025):

Reboot due to config change!
```

Figure 67: Simulated telemetry, safe mode and HAB modes activation or deactivation.

Commands "-c", "-r" and "-l" are used for modifying data from the "sim.cfg" file of the CubeSat. When the CubeSat turns on, the first data read and uploaded is the sim configuration file. Therefore, when modifying the data inside this file, it will be applied for further ignitions. The command "-C" clear the software logs and the "-F" sets the transmission and reception frequency in which the CubeSatSim is desired to transmit information.

Finally, the commands that show information about the CubeSat are the "-S", "-p" and "-v" are for displaying real-time CubeSat data in the computer.

This first command will show a I2C bus scanner which indicates the addresses where something is connected to it.

```
pi@cubesatsim:~ $ CubeSatSim/config -S
CubeSatSim v2.0 configuration tool


Scan both I2C buses on the Raspberry Pi

I2C Bus 1

     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:                         -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: 40 41 -- -- 44 -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --

I2C Bus 3

     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:                         -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: 40 41 -- -- 44 45 -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Figure 68: I2C bus scanner of the Raspberry Pi.

In the figure above, both buses 1 and 3 are being shown up. The different addresses with connected or not connected devices can be seen.

The second command shows a line code with the real-time telemetry data sending as can be observed on the following image:

```
pi@cubesatsim:~ $ CubeSatSim/config -p
CubeSatSim v2.0 configuration tool

Real-time output from the serial port from the Pico:

_START_FLAG_OK BME280 29.94 1006.56 55.86 33.28 MPU6050 -3.15 -0.99 1.08 0.01 -0.00 1.03 GPS 0.0000 0.0000 0.00 TMP -1257.00_END_FLAG
_
_START_FLAG_OK BME280 29.95 1006.53 56.13 33.27 MPU6050 -3.53 -1.25 1.19 0.01 -0.01 1.02 GPS 0.0000 0.0000 0.00 TMP -1257.00_END_FLAG
_
_START_FLAG_OK BME280 29.96 1006.56 55.85 33.27 MPU6050 -3.39 -0.96 1.21 0.01 -0.00 1.02 GPS 0.0000 0.0000 0.00 TMP -1257.00_END_FLAG
_
```

Figure 69: Real-time output telemetry data.

Finally, the third command shows the real-time output voltage and current data from solar panels and batteries.

```
_
pi@cubesatsim:~ $ CubeSatSim/config -v
CubeSatSim v2.0 configuration tool

Real-time output from the INA219 voltage and current sensors:

CubeSatSim v2.0 INA219 Voltage and Current Telemetry
/usr/local/lib/python3.9/dist-packages/adafruit_blinka/microcontroller/generic_linux/i2c.py:30: RuntimeWarning: I2C frequency is not
settable in python, ignoring!
  warnings.warn(

+X   |  2.87 V     0 mA
+Y   |  1.39 V     0 mA
+Z   |  3.18 V     0 mA
-X   |  3.17 V     0 mA
-Y   |  1.33 V     0 mA
-Z   |  1.15 V     1 mA
Bat  |  4.09 V  -671 mA
Bat2 |  0.00 V     0 mA
```

Figure 70: Real-time output voltage and current data from solar panels and batteries.

4. DATA TRANSMISSION

In this section, data will start to be transmitted and the different modes of transmission will be explained too. Firstly, a test for the STEM Payload Board will be done. If the data received is coherent, the conclusion will be that the board works correctly. Then, after ensuring the correct functioning of all the boards individually, the final structure of the CubeSatSim will be built (as explained in subsection *2.2 Structure*) and at that point, will be ready to start communicating with the base station

4.1. STEM Payload Board Testing

The first step is to connect the STEM Payload Board to current and wait 30 seconds until it completely turns on. The board has three LEDs which represent different actions.

Figure 71: LEDs turned on.

1. Red LED: turns on when connecting PCB to power.
2. Green LED: turns on when the board is powered on (after removing the RBF jack pin) and the software is running.
3. Blue LED: turns on when transmitting data.

Furthermore, the green LED acts as a mode indicator when changing modes. When pressing and holding the button between red and green LEDs, the green LED will blink:

- 1 time for Mode AFSK
- 2 times for Mode FSK
- 3 times for Mode BPSK
- 4 times for Mode SSTV
- 5 times for Mode CW

To select a mode, release the button when the LED blinks the number of times corresponding to the desired mode.

If any of the LEDs do not turn on when specified, it will mean either they are bad soldered or there is a problem with the software. Once the LED's testing works perfectly, the board is prepared to send data to the computer.

Firstly, as explained in subsection 3.2 Ground Station, the SDR# program is used to ensure the correct functioning of the receiver RTL-SDR and the signal transmission of the board:

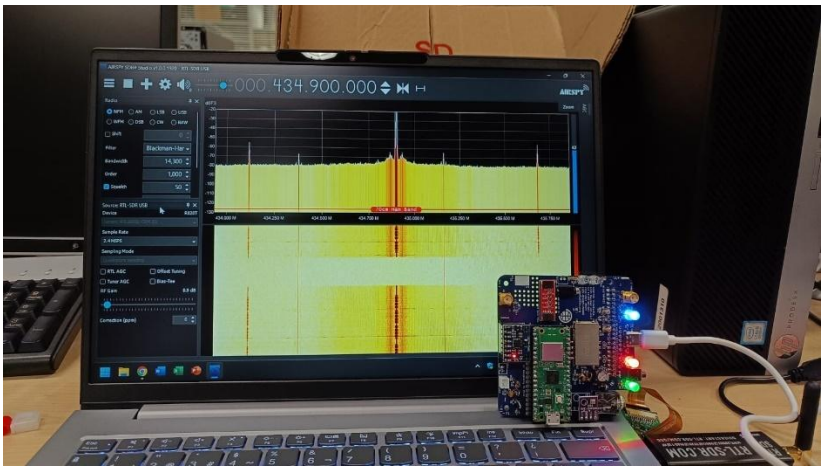


Figure 72: Signal received on SDR# with the RTL-SDR (receiver) and STEM Payload Board (transmitter).

On Figure 70, the spectrum of a signal in 434.9MHz can be observed, which means that both the transmitter and receiver work correctly. Now, once it is ensured that STEM payload board transmits correctly, the whole CubeSat, after building it, must send data correctly too.

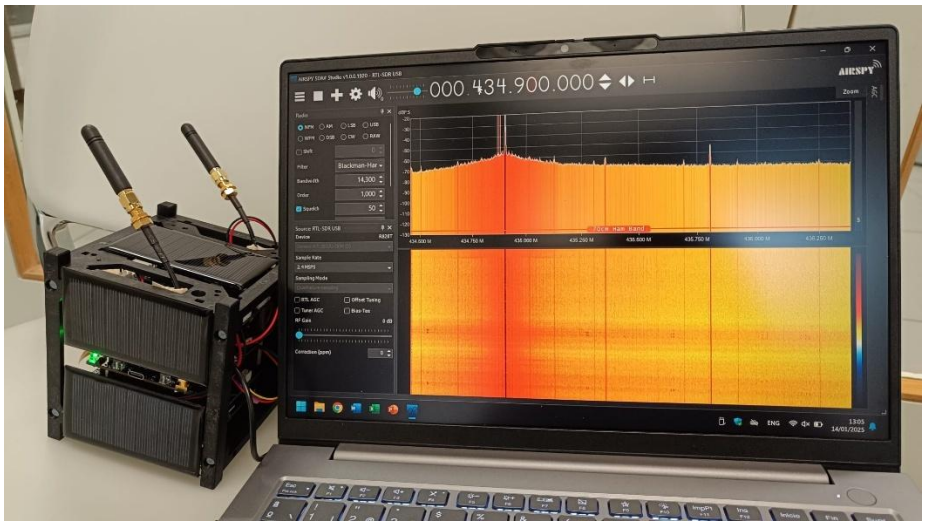


Figure 73: CubeSat data transmission.

On the image above, a signal can be observed at the desired frequency, meaning that the CubeSat sends correctly signals at 434.9MHz and the RTL-SDR receives a signal. At this point, telemetry and data sent can be now decoded and studied.
For FSK and BPSK FoxTelem will be used whereas for AFSK, SDR# will be needed to receive the information packets. However, it is not yet demonstrated that the data received from the board is real telemetry or just noise or random data.

In order to do so, the three transmission modes: FSK, BPSK and APRS are going to be explained in the next subsections.

4.1.FSK

This is the standard mode of transmission and operation of this CubeSatSim. The program FoxTelem has to be executed and the FSK mode has to be selected for the reception program (200bps Fox1) as well as for the transmitter board (with the button mentioned before).

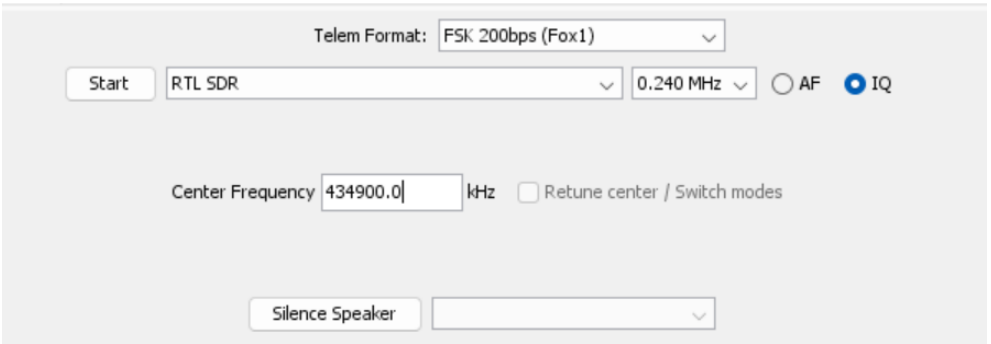


Figure 74: Telemetry format selection.

Once the blue LED is on and the program is running for reception, a similar signal to the following image would be detected.

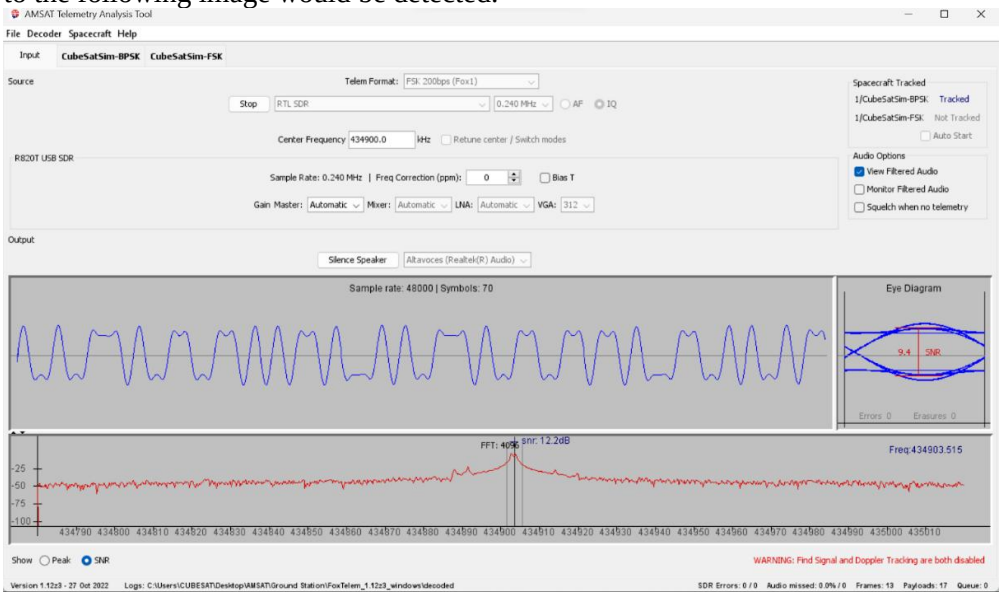


Figure 75: FoxTelem FSK detection signal interface.

As explained in other section, different parameters and representations can be obtained with this program so as to study the signal received. It can be observed that data makes sense thanks to the signal representation (similar to the typical FSK waveform) and the eye diagram, which is very accurate.

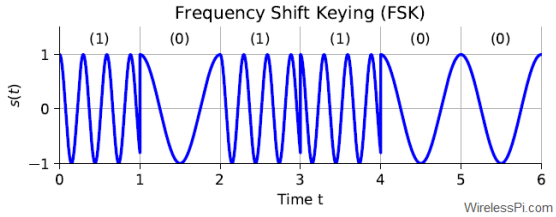


Figure 76: Theoretical FSK signal waveform.[11]

Looking at the "CubeSatSim-FSK" window, the telemetry decoded data from the payload packets will be shown.

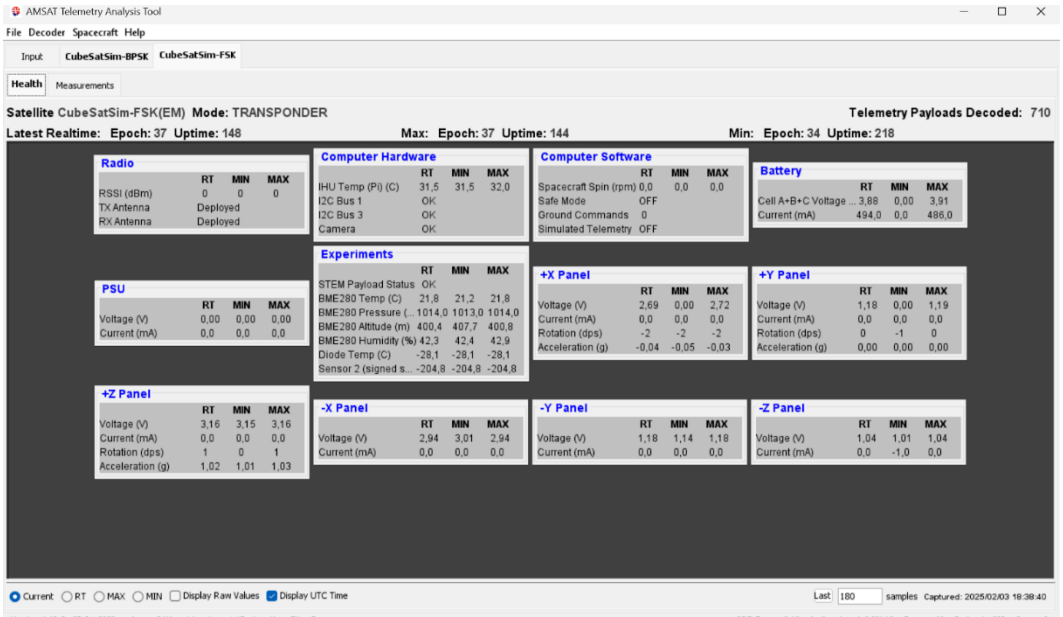


Figure 77: Correct telemetry received from STEM payload Board.

In this section of the program the telemetry received from the CubeSat is shown. When receiving more packets, the Telemetry Payloads Decoded number must increase, the time and date of the last packet received, update and the data from the different boxes replaced with the new information.

[11] https://wirelesspi.com/i-q-signals-101-neither-complex-nor-complicated/

4.2. BPSK

On the other hand, the decoding for the BPSK works exactly the same as for the FSK. When selecting the "BPSK 1200bps (Fox1E)" telemetry format on the program and starting the reception, the program interface should look like in the next picture:

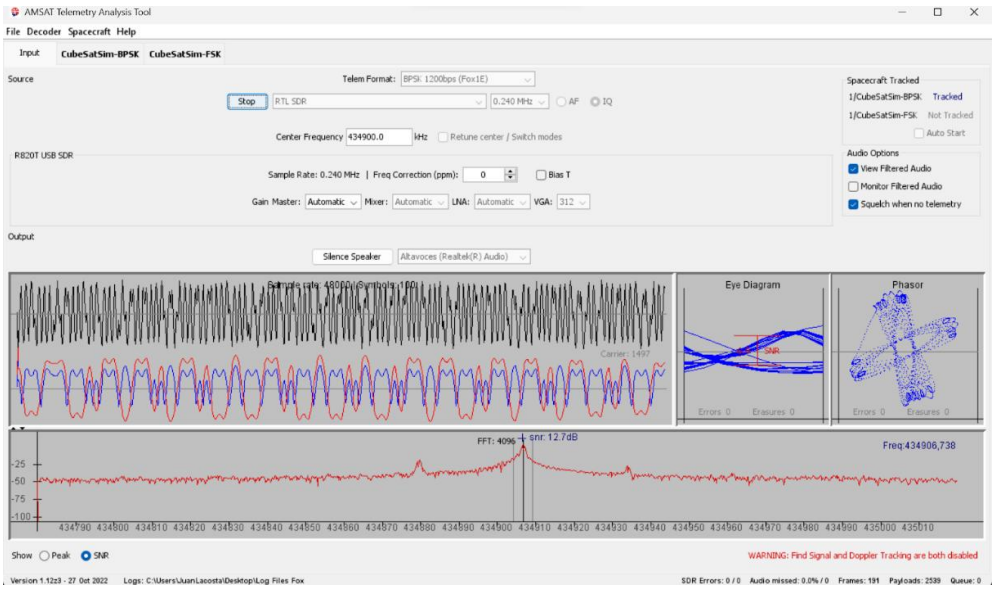


Figure 78: Signal BPSK FoxTelem reception.

It can be seen that the BPSK signal waveform received is similar to the theoretical one too, as was expected. This means that the signal received from the CubeSat is received in a well BPSK modulation and it is well decoded by the program.

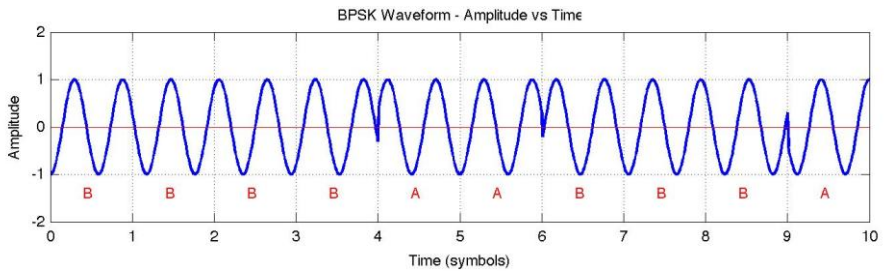


Figure 79: Theoretical BPSK waveform.[12]

Also, as in FSK modulation, there is a telemetry section in which data can be seen and updated in real-time.

[12] https://www.ece.unb.ca/tervo/ece4253/qpsk.shtml

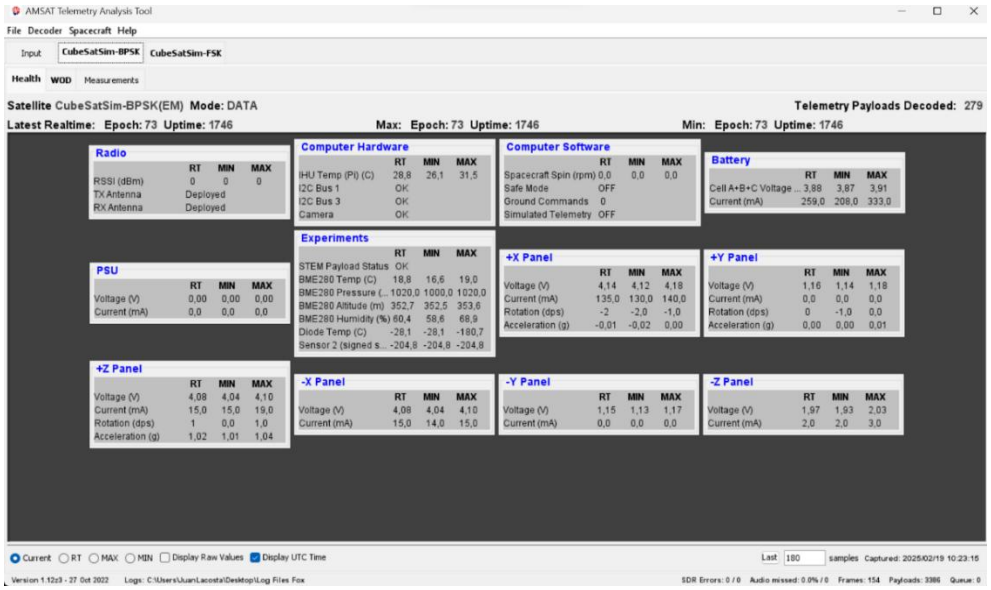


Figure 80: BPSK telemetry data reception.

Furthermore, all of these telemetry data obtained and read from the decoding programs must match the data seen before in the SSH connection (the "-p" and "-v" options from the menu). In the following images, the decoded telemetry from FoxTelem is demonstrated to be correct, as it matches the data from the CubeSatSim software. Therefore, as these data and the sensor's data coincide with the program decoded data, it can be concluded that the data transmission and reception is correct.

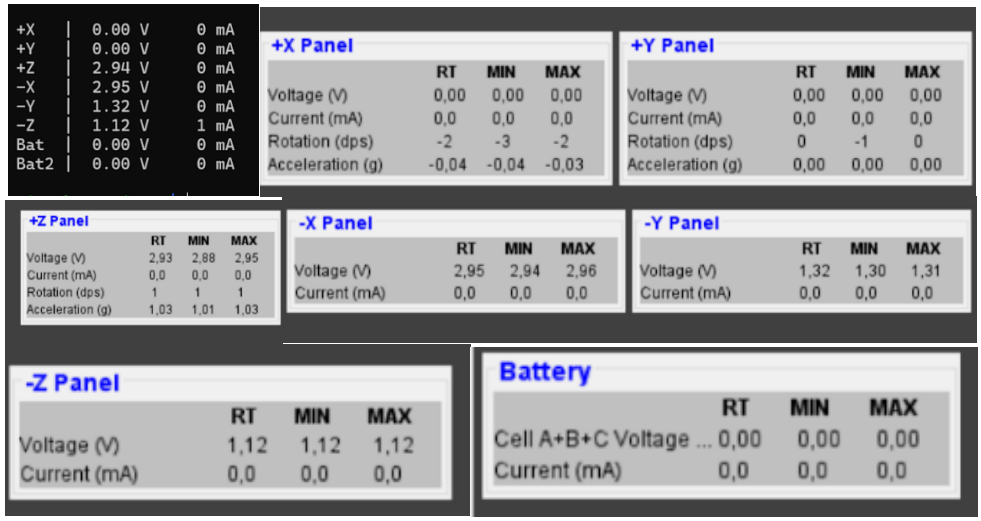


Figure 81: Batteries and solar panels telemetry check.

4.3. AFSK

Finally, the last mode in which CubeSat can send data is by sending APRS packages via radio frequency. This information can be decoded with SDR#. In order to do so the VBCable software has to be installed[13] so as to set it as input. To decode AFSK APRS telemetry in Windows, AFSK 1200 Decoder software must be downloaded[14] Then, once the program is installed, the VBCable ought to be selected as input, as shown in the following picture:

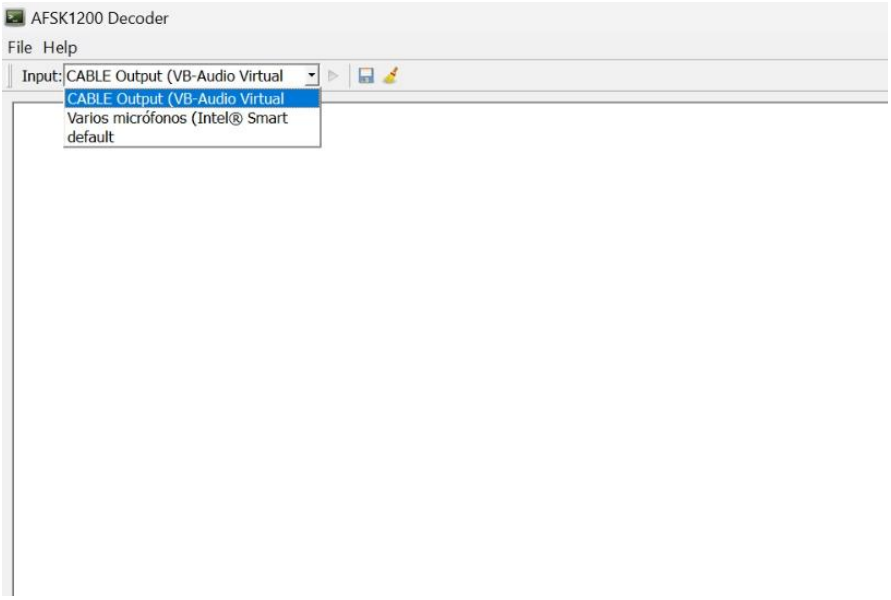


Figure 82: Setting as input the VB virtual cable.

Then, the last step is to make sure, in SDR#, you are tuned into the AFSK radio signal and that the audio is unmuted and at full volume. At this point, when running both programs, decoded packets must appear.

[13] https://download.vb-audio.com/Download_CABLE/VBCABLE_Driver_Pack43.zip
[14] https://sourceforge.net/projects/qtmm/

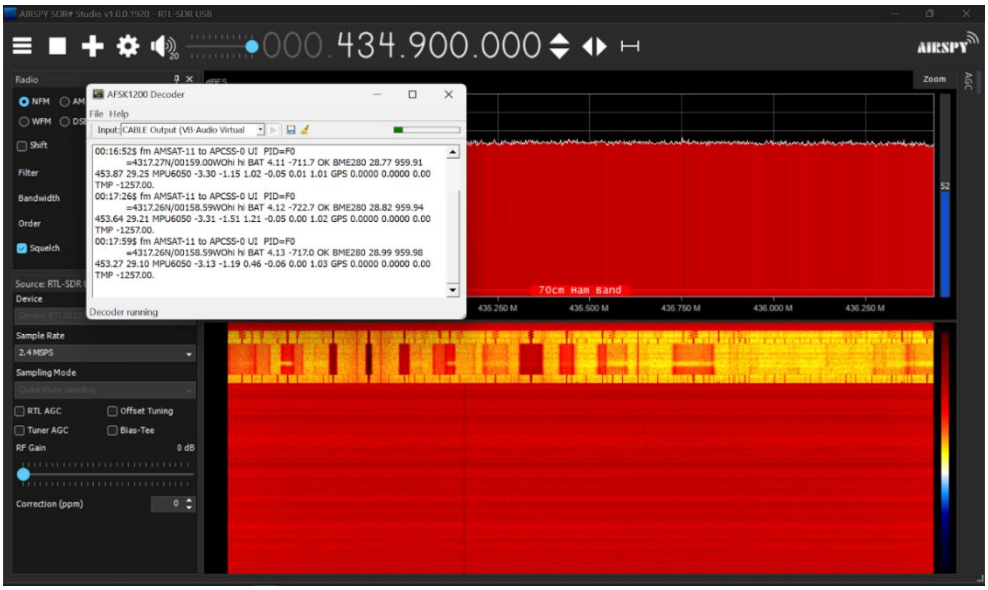


Figure 83: Decoded information APRS packets in AFSK mode.

Also, this data can be read, in the same format, from the CMD program:

```
C:\Users\CUBESAT\Desktop\PFG\Juan\Ground Station\multimon-ng>rtl_fm -f 434.9M -s 22050 -g 48 - | multimon-ng -a AFSK1200 -A -t raw -
multimon-ng  (C) 1996/1997 by Tom Sailer HB9JNX/AE4WA
             (C) 2012-2014 by Elias Oenal
available demodulators: POCSAG512 POCSAG1200 POCSAG2400 Found 1 device(s):
EAS UFSK1200 CLIPFSK FMSFSK AFSK1200 AFSK2400 AFSK2400_2 AFSK2400_3 HAPN4800 FSK9600 DTMF ZVEI1 ZVEI2 ZVEI3 DZVEI PZVEI EEA EIA CCIR
MORSE_CW DUMPCSV
Enabled demodulators: AFSK1200
  0:  Realtek, RTL2838UHIDIR, SN: 00000001

Using device 0: Generic RTL2832U OEM
Found Rafael Micro R820T tuner
Tuner gain set to 48.00 dB.
Tuned to 435153575 Hz.
Oversampling input by: 46x.
Oversampling output by: 1x.
Buffer size: 8.08ms
Exact sample rate is: 1014300.020041 Hz
Sampling at 1014300 S/s.
Output at 22050 Hz.
APRS: AMSAT-11>APCSS:=4317.34N/00158.57WOhi hi BAT 4.23 -534.8 OK BME280 28.04 960.13 451.92 29.87 MPU6050 -3.02 -1.47 1.39 -0.06 0.0
0 1.02 GPS 0.0000 0.0000 0.00 TMP -1257.00

APRS: AMSAT-11>APCSS:=4317.34N/00158.57WOhi hi BAT 4.24 -573.9 OK BME280 28.15 960.18 451.52 29.76 MPU6050 -2.93 -1.33 1.57 -0.06 0.0
0 1.02 GPS 0.0000 0.0000 0.00 TMP -1257.00

APRS: AMSAT-11>APCSS:=4317.34N/00158.55WOhi hi BAT 4.24 -570.8 OK BME280 28.70 960.12 452.03 29.05 MPU6050 -3.30 -1.33 1.40 -0.06 0.0
1 1.02 GPS 0.0000 0.0000 0.00 TMP -1257.00

APRS: AMSAT-11>APCSS:=4317.34N/00158.55WOhi hi BAT 4.24 -568.3 OK BME280 28.81 960.09 452.29 28.80 MPU6050 -2.95 -0.89 1.34 -0.06 0.0
0 1.02 GPS 0.0000 0.0000 0.00 TMP -1257.00
```

Figure 84: Decoded packets read in CMD.

It can be observed that the interface of the decoded packets is very different in comparison with the FoxTelem layout seen before. These APRS packets receive exactly what the CubeSat is transmitting from the lecture of the sensors (as explained before from the *get_tlm()* function). However, in FoxTelem, as the *get_tlm_fox()* function is defined strictly for this program, data is received in a specific format and printed in those tables and graphs. That is one of the reasons why FoxTelem is more used than SDR# for receiving information.

Up to this point, the complete CubeSat structure has been built and the software installed and configured. After managing all the energy maintaining the CubeSat operative, reading all the data and encoding it correctly, the last step is to transmit it to the base station correctly to accede to the data telemetry and continue with the analysis and objectives of the sustainable mobility mission.

4.4. Maximum transmission distance

Now that the telemetry has been properly verified and data transmission and reception are functioning correctly, it is crucial to determine the maximum range of this communication link in order to achieve the previously mentioned objective of studying sustainable mobility in a public area using the CubeSat. This maximum distance depends on the transmission and operation modes of the CubeSatSim.

To assess the maximum range of the communication link before it is lost, three distance tests were conducted using different modes of operation and transmission. Two of these tests were carried out along a 2.5 km-long beach in Zarautz, while the third test took place in an industrial park in Miramon, Tecnun, where the presence of buildings and other obstacles could potentially interfere with signal propagation.

For the beach tests, the CubeSatSim was securely mounted on a tripod (as seen in the image on the left) to ensure stability and minimize human-induced signal reflections. Once properly positioned, the CubeSatSim was powered on using its internal energy management system and began transmitting telemetry data. For this initial test, both HAB and SIM modes were disabled, and FSK modulation was used.

The first test resulted in a maximum range of approximately 210 meters. Beyond this distance, the computer base was unable to receive further decoded telemetry payloads using FoxTelem.

Figure 85: FSK maximum distance of transmission.

The second test was identical to the first, except for the transmission mode, which was switched to BPSK. The result of this test is as follows:

Figure 86: BPSK maximum distance of transmission.

These results indicate that such a limited communication range is insufficient to achieve the intended objective. The CubeSatSim will be adequate for achieving the mission as long as the maximum desired communication link distance is less than 200 meters. Therefore, in these conditions, the mobility mission will be determined for this distance.

To explore possible improvements, a third test was conducted. This time, the transmission mode remained FSK, but HAB mode was enabled to analyze its impact on performance.

Figure 87: FSK, with HAB mode on, maximum distance of transmission.

In this case, the range improved, but still did not reach an optimal level. Despite potential interferences from buildings in the industrial park, the communication link extended up to 280 meters. While 280 meters altitude for a CubeSat is not entirely inadequate, it still leaves significant room for improvement.

5. CONCLUSIONS AND FUTURE WORK

5.1. Conclusions

As explained in the objectives, the aim of this project was to create from zero a CubeSatSim capable of taking some telemetry data, encoding and transmitting it, and then decoding and analyzing it at a base station located at a considerable distance, in order to support the sustainable mobility mission.

The mission focused on studying how different weather conditions affect commuting habits and the proportion of people using sustainable transportation (walking, cycling, or public transport) versus private vehicles. By analyzing this data, adaptive measures—such as optimizing public transport frequency—could be proposed to reduce private vehicle usage and lower CO_2 emissions.

As seen throughout the project, a CubeSatSim has been successfully created and built. It was capable of taking telemetry, encoding and decoding it and finally transmitting it to a base station computer for further analysis. However, the communication range was limited (with a maximum of 280 meters) which, in some environments, may not be sufficient for the mission to be carried out effectively. These limitations, though, provide valuable insights into the system's current capabilities and serve as a solid starting point for future improvements. Therefore, future work is needed to do to the CubeSat in order to be an efficient satellite and worth launching.

5.2. Future Work

There is a lot of future work to be done on this project. With ambitious objectives, the ultimate goal of this CubeSat would be to improve it to such an extent that it could be launched into space and left orbiting, thus providing various data to the base computer for a long time. However, that remains a long-term goal. For the current mission, several improvements can be made to enhance performance and ensure the project's success.

Firstly, the low-cost antennas used in this prototype could be replaced with higher-performance antennas capable of transmitting signals more efficiently, thereby increasing the maximum communication range. Furthermore, adding an amplifier for sending a more powerful signal will also be necessary. Without addressing this limitation, the CubeSat would be unable to fulfill its intended mission.

Secondly, integrating a GPS module would significantly enhance the system by allowing precise location tracking of the CubeSat. This would ensure that all collected telemetry data is accurately linked to a specific geographical area, improving data analysis and interpretation.

Finally, adding a camera module that captures images at regular intervals would provide visual data to complement the telemetry readings. By analyzing these images, it would be possible to estimate the proportion of buses and private vehicles in the area, further supporting the study of mobility patterns. This

additional information would improve decision-making and allow for a more comprehensive understanding of commuting behaviors. Furthermore, after implementing this camera module, a software for processing the images will be needed too.

Implementing these enhancements would greatly improve the CubeSat's capabilities, making the project more robust and increasing its potential for real-world applications.

6. **BUDGET**

The budget corresponding to the development of the project is detailed in the present Section and is divided into the following parts:

- Fixed costs: this includes all components bought to third parties that have been used in the project. These are presented in Table 7

- Labor costs: this includes the labor hours needed to carry out the project. This cost is presented in Table 8.

The overall budget of the project is shown in Table 9.

Fixed costs:

Description	Quantity	Unity price	Total Price
2.5MM TO 3.5MM JUMPER CABLE	1	6,70 €	6,70 €
4.5MM HEX X 23MM X M2.5 THD	16	3,12 €	49,84 €
AUDIO PLUG, 3.5 MM, STRAIGHT, 2	4	1,20 €	4,80 €
BATTERY HOLDER AA 2 CELL SMD	2	5,03 €	10,06 €
BATTERY HOLDER AA SMD	6	3,87 €	23,22 €
BLUE INA219 HIGH SIDE DC CURRENT SENSOR BREAKOUT - 26V ±3.2A MAX	1	21,18 €	21,18 €
BLUE INA219 HIGH SIDE DC CURRENT SENSOR BREAKOUT - 26V ±3.2A MAX	1	9,79 €	9,79 €
BME280 BOARD TEMPERATURE/HUMIDITY/PRESSURE	1	20,99 €	20,99 €
CAP ALUM 47UF 20% 63V RADIAL TH	3	0,37 €	1,11 €
CAP CER 0.1UF 50V Z5U RADIAL	4	0,20 €	0,80 €
CAP CER 18PF 50V C0G/NP0 0603	4	0,17 €	0,68 €
CLEAR MOUNTING TAPE	2	7,41 €	14,82 €
COAX CBL SMA TO SMA 2.9"	4	6,67 €	26,68 €
CONN HDR 20POS 0.1 TIN PCB	4	1,30 €	5,20 €
CONN HDR 40POS 0.1 TIN PCB	2	2,41 €	4,82 €
CONN HDR 4POS 0.1 GOLD PCB	2	0,51 €	1,02 €
CONN HDR 8POS 0.1 TIN PCB	2	0,59 €	1,18 €
CONN HEADER VERT 20POS 2.54MM	4	1,22 €	4,88 €
CONN HEADER VERT 2POS 2MM	6	0,17 €	1,02 €
CONN HEADER VERT 40POS	2	0,93 €	1,86 €

2.54MM			
CONN HEADER VERT 4POS 2.54MM	2	0,56 €	1,12 €
CONN JACK STEREO 2.5MM R/A	2	0,66 €	1,32 €
CONN JACK STEREO 3.5MM R/A	2	3,99 €	7,98 €
CONN RCP USB2.0 TYP C 24P SMD RA	4	0,72 €	2,88 €
DIODE SCHOTTKY 20V 1A AXIAL	4	0,41 €	1,64 €
DIODE SCHOTTKY 20V 1A AXIAL	12	0,33 €	3,92 €
DIODE STANDARD 100V 200MA DO35	2	0,09 €	0,18 €
FIXED IND 13NH 600MA 0.13OHM SMD	4	0,12 €	0,48 €
GPIO 20X2 FEMALE STACKING HEADER EXTRA LONG PINS	6	7,89 €	47,34 €
HEX NUT 3.81MM NYLON M2	8	0,08 €	0,62 €
HEX NUT 4.83MM NYLON M2.5	21	0,09 €	1,89 €
HEX SPACER M2.5X0.45 STEEL 11MM	4	0,39 €	1,56 €
HEX SPACER M2.5X0.45 STEEL 18MM	4	0,48 €	1,92 €
HEX STANDOFF M2.5X0.45 STEEL 6MM	4	0,47 €	1,88 €
JST 2 PIN WIRES AND CONNECTORS	1	8,99 €	8,99 €
JST-PH 2-PIN JUMPER CABLE - 100M	4	0,85 €	3,40 €
LED BLUE CLEAR 5MM ROUND T/H	4	0,22 €	0,88 €
LED GREEN CLEAR 5MM ROUND T/H	4	0,26 €	1,04 €
LED RED CLEAR 5MM ROUND T/H	2	0,16 €	0,32 €
MACH SCREW FLAT SLOTTED M2X0.4	8	0,10 €	0,78 €
MACH SCREW PAN PHILLIP M2.5X0.45	16	0,23 €	3,63 €
MACH SCREW PAN PHILLIP M2.5X0.45	16	0,09 €	1,44 €
MACH SCREW PAN SLOTTED M2.5X0.45	4	0,43 €	1,72 €
MPU6050 3 AXIS GYRO 3 AXIS ACCEL GY-521	1	11,99 €	11,99 €
OTG CABLE FOR SOUND CARD	1	5,99 €	5,99 €
PA 4035 CF FIL, 1.75 MM, 1KG	1	98,94 €	98,94 €
PI CAMERA WITH PI ZERO	2	11,77 €	23,54 €

RIBBON CABLE			
PIN HEADER, THR, SINGLE ROW, .10	2	0,09 €	0,18 €
QWIIC ADAPTER	4	1,43 €	5,72 €
REMOVE BEFORE FLIGHT TAG	1	8,99 €	8,99 €
RES 100 OHM 5% 1/8W AXIAL	10	0,04 €	0,38 €
RES 180 OHM 5% 1/8W AXIAL	2	0,09 €	0,18 €
RES 1K OHM 5% 1/6W AXIAL	8	0,09 €	0,72 €
RES 220 OHM 5% 1/8W AXIAL	10	0,04 €	0,38 €
RES 4.7K OHM 5% 1/6W AXIAL	8	0,09 €	0,72 €
RES 5.1K OHM 5% 1/10W 0603	10	0,01 €	0,13 €
RES 68 OHM 5% 1/2W AXIAL	4	0,09 €	0,36 €
RF ANT 433MHZ WHIP RA SMA 42MM	2	5,49 €	10,98 €
RF ANT 433MHZ WHIP STR SMA 48MM	4	5,80 €	23,20 €
RTL-SDR	2	56,42 €	112,84 €
SMA JACK VERTICAL PCB	4	3,50 €	14,00 €
SOLAR CELL 6V 60MA (SET OF 10) 72MM X 45MM	1	24,96 €	24,96 €
SWITCH TACTILE SPST-NO 0.05A 12V	5	0,39 €	1,95 €
TENERGY AA 2500MAH NIMH RECHARGEABLE BATTERY 4 PACK	2	21,37 €	42,74 €
USB SOUND CARD	2	10,99 €	21,98 €
USB-C CABLE AND POWER PLUG	2	9,99 €	19,98 €
PAYLOAD BOARDS	3	31,36€	94,07€
		Total	**832,50 €**

Table 7: Fixed costs budget

Labor costs:

A typical starting salary for an engineer in Spain is around 1600€/month. Then, supposing a month has 30 days and 4 weekends (22 working days) with 8 hour long working days, the final price per hour will be 9.0909€/hour.

Task	Duration (h)	Price per hour	Total price
Junior Engineer research	300	9.0909 €	2727.27 €

Table 8: Labor budget

Overall Budget:

Item	Partial price	Accumulated price
Fixed Costs	832,50 €	832,5 €
Indirect Costs (10%):	83,25 €	915,75 €
Labor Costs	2727.27 €	3643.02 €
Total Budget: 2961,2045 €		

Table 9: Total budget of the project

7. REFERENCES

[1] A Brief History of AMSAT. (n.d.). https://www.amsat.org/amsat-history/

[2] Airspy.com. (2025, January 31). SDR# and Airspy Downloads - AIRSPY. https://airspy.com/download/

[3] Alanbjohnston. (n.d.-b). GitHub - alanbjohnston/CubeSatSim: CubeSatSim, the AMSAT CubeSat Simulator. GitHub. https://github.com/alanbjohnston/CubeSatSim/

[4] AMSAT – the Radio Amateur Satellite Corporation. (n.d.). https://www.amsat.org/

[5] Bill of Materials for AMSAT CubeSatSim v2.0. (s. f.). Google Docs. https://cubesatsim.org/bom

[6] Chaudhari, Q. (2025, March 23). I/Q Signals 101: Neither Complex nor Complicated. Wireless Pi. https://wirelesspi.com/i-q-signals-101-neither-complex-nor-complicated/

[7] CubeSat. (n.d.). CubeSat. https://www.cubesat.org/

[8] CubeSat Communications: Recent Advances and Future Challenges. (2020, January 1). IEEE Journals & Magazine | IEEE Xplore. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9079470

[9] Download VBCable Driver Pack. https://download.vb-audio.com/Download_CABLE/VBCABLE_Driver_Pack43.zip

[10] FoxTelem Software for Windows, Mac, & Linux. (n.d.). https://www.amsat.org/foxtelem-software-for-windows-mac-linux/

[11] How much does a newly graduated engineer earn in Spain? | News. (n.d.). Comprehensive Internship and Employment Service. https://sipem.upct.es/noticia/cuanto-cobra-un-ingeniero-recien-graduado-en-espana#:~:text=Por%20un%20lado%2C%20est%C3%A1%20el,una%20vigencia%20de%20tres%20a%C3%B1os.

[12] Ibáñez, I. (2015, November 2). The Era of CubeSats. Infoespacial. https://www.infoespacial.com/texto-diario/mostrar/3572586/cubesats

[13] Mars Cube One. (2024, November 26). Wikipedia, the Free Encyclopedia. https://es.wikipedia.org/wiki/Mars_Cube_One#:~:text=Mars%20Cube%20One%20(o%20MarCO,Mars%20Lander%20de%20la%20NASA.

[14] OpenAI. (2025). ChatGPT (version GPT-4) [Artificial intelligence language model used occasionally for English translations and specific queries]. https://openai.com

[15] OSCAR 1. (2025, March 1). Wikipedia, the Free Encyclopedia. https://en.wikipedia.org/wiki/OSCAR_1

[16] Phase Shift Keying. https://www.ece.unb.ca/tervo/ece4253/qpsk.shtml

[17] Qtmm AFSK1200 Decoder Downloader. https://sourceforge.net/projects/qtmm/

[18] WebCite Query Result. (n.d.). https://webcitation.org/6ABSpR8qR?url=http://www.cubesat.org/images/developers/cds_rev12.pdf

[19] Wikipedia Contributors. (2025, February 24). CubeSat. Wikipedia, the Free Encyclopedia. https://en.wikipedia.org/wiki/CubeSat