www.editada.org

International Journal of Combinatorial Optimization Problems and Informatics, 15(5), Dec 2024, 96-107. ISSN: 2007-1558. https://doi.org/10.61467/2007.1558.2024.v15i5.562

Development of a Graphical Interface for Communication with CubeSat Space Protocol-Based Cards: Telemetry Reception Results

Miguel Limón González, Enrique Rafael García Sánchez, Selene Edith Maya Rueda, Nicolas Quiroz Hernández

Benemérita Universidad Autónoma de Puebla, Facultad de Ciencias de la Electrónica.
miguel.limongonzalez@gmail.com, rafael.garciasan@correo.buap.mx, selene.maya@correo.buap.mx, nicolas.quirozh@correo.buap.mx.

Abstract. A graphical user interface is proposed to facilitate telemetry collection and configuration set-up for CubeSat Space Protocol-compatible nanosatellites during their testing and assembly stage. The graphical user interface connects to the various subsystems of the nanosatellite through a module that functions as a gateway. The graphical user interface is written in Python and Qt. Python is an interpreted high-level language compatible with Linux systems, and Qt is a multi-platform graphical user interface development framework. Through the graphical user interface, telemetry from the various satellite modules can be collected.

Keywords: Nanosatellite, CubeSat, Python, Qt, GUI, Telemetry

Article Info
Received May 10, 2024.
Accepted Nov 20, 2024.

1 Introduction

Satellites can be classified by their mass and size. Within the category of small satellites, we have picosatellites (less than 1 kg), nanosatellites (1 to 10 kg), microsatellites (10 to 100 kg), and minisatellites (100 to 1000 kg) [1]. The establishment of standards for nanosatellite construction has allowed for a decrease in manufacturing and launch costs. An example of this is the CubeSat standard, which defines the basic form factor as 1U, with dimensions of 10 cm x 10 cm x 10 cm, and a mass of less than 2 kg [2]. In addition to the emergence of standards, the reduction of costs and risks in nanosatellite manufacturing projects has driven using commercial off-the-shelf components (COTS) [3]. In the last decade, the launch of nanosatellites has seen significant growth. According to the NanoSats database in 2011 the number of launched nanosatellites was twelve launched while in 2023, the number of launched nanosatellites increased to 396 [4].

Internally, a satellite is composed of the following subsystems: Electrical Power System (EPS), On Board Computer (OBC), Attitude Determination and Control System (ADCS), radio, and payload. EPS refers to the satellite's power module. OBC refers to the flight computer that manages the satellite's systems. ADCS is responsible for controlling the satellite's orientation. The radio is responsible for receiving commands and transmitting telemetry and mission information to the ground segment or to other satellites [5], [6]. Finally, the module containing the necessary elements to fulfill the satellite's mission is known as payload. For example, if the mission is about Earth observation, the payload is likely to be a device for capturing images.

CubeSat Space Protocol (CSP) is a protocol stack designed for CubeSats. CSP shares the layered design of TCP/IP. One of its main highlights is that it allows embedded systems to define different services through the

same communication channel, similar to computer networks using TCP/IP where a server can offer different services using the same IP address on different TCP/UDP ports [7].

The CSP protocol was developed in 2008, and it was published under the MIT license as libCSP. Since then, it has received periodic updates. The CSP protocol stack is written in C, and it is considered cross-platform. Currently, it has support for different operating systems such as FreeRTOS, Windows, MacOS, and Linux as well as bindings for the Python language. The libCSP library includes code that allows packet routing, ICMP-like requests such as ping and buffer status [8]. CSP is available under LGPL license.

Python is an interpreted high-level programming language widely used in various fields such as web development, data analysis, artificial intelligence, and more [9]. On the other hand, Qt is a multi-platform application development framework that allows creating graphical user interfaces (GUIs). Qt uses a dual licensing model, which means it is available under both a commercial license and an open-source license (GNU Lesser General Public License, LGPL) [10]. This provides flexibility for developers to choose the option that best suits their needs.

Regarding the integration of Qt with Python, PyQt is a set of bindings that allows utilizing Qt capabilities in Python-written applications. Qt Designer is a tool included in Qt that enables designing user interfaces visually by dragging and dropping components to achieve the desired appearance [11].

As mentioned earlier, nanosatellites are typically composed of multiple modules. Therefore, to test such systems, flat-sat test benches are commonly used, allowing the interconnection of satellite modules in the same manner as they would be once the system is assembled [12]. To debug the firmware of these systems, it is necessary to have multiple connections that allow monitoring the debug console of each module. It is worth noting that the more components the system has, the more complicated it becomes to evaluate such a system.

Testing during nanosatellite development is a crucial step to ensure reliability in space. To achieve this, it is necessary to have tools that facilitate the work, particularly for long and repetitive tasks such as gathering data or connecting to the subsystem's shell. In these cases, we found that a GUI compatible with the CubeSat Space Protocol, and equipped with logging capabilities, can be very useful.

1.1 Related work

In [13], Grillo conducted a presentation where he displayed a graphical interface for visualizing vibration test data and highlighted the advantages of using Python and Qt as an alternative to proprietary software. Among the advantages mentioned by Grillo of using Python and Qt, we can find: the ability to edit the code, making it highly adaptable to the needs of each project since the algorithm used by the graphical interfaces can be understood, and programs developed with these technologies are multi-platform and scalable.

In [14], Esposito developed a GUI based on Python and the Tkinter graphics library, resulting in a cross-platform graphical inte2ace that allowed the operator to visualize telemetry received via radio on any computer capable of running Python.

In [15], G. Orbiols et al. explain the difficulties they faced during the verification tests of the E-St@r-II nanosatellite due to the lack of interfaces other than radio communication to monitor and control the subsystems' status during the tests.

In [16], Rodríguez proposes the development of a device focused on testing EPS subsystems for CubeSats. This module is responsible for recording values of various parameters of the power system, and this information can be visualized from the computer through a LabView-based GUI.

In [17], Tapsawat uses HiL testing using custom-build interfaces with multiple pre-designed MCU boards and embedded computers to test ADCS subsystem.

After an exhaustive search, we found similar work focused on power module testing as and ADCS module testing. On those works they depict very specific testing hardware and complex connection systems. However,

the approach that we seek is towards to a slightly more general solution focused on collecting and managing data through a single interface and avoid situations as those mentioned above.

This work proposes the development of a graphical interface written in Python, utilizing Qt as the graphics library, and leveraging the capabilities of CubeSat Space Protocol (CSP) to communicate the PC with different satellite subsystems through a single interface. It is expected that the use of a graphical interface and CSP will facilitate telemetry data collection and visualization during the testing period of nanosatellite missions by allowing the visualization of the nanosatellite's overall telemetry through a single physical interface.

2 CubeSat Test Environment

Several tools were used for the development of this project to evaluate the functionality of the graphical interface, the equipment used, as well as some relevant characteristics and the configuration used are described in this section:

2.1 Flat-Sat

A NanoUtil TestDock flat-sat model from GOMspace was used. A flat-sat is a test bench for nanosatellites that allows interconnecting cards in the same way they would be in an assembled nanosatellite. For this purpose, the system has three PC-104 type connectors interconnected. The flat-sat used in this project has ports to connect an external power supply used to perform battery charging and discharging tests for the EPS module. In addition to the EPS, it also has a switch that simulates the behavior of the kill switch required by the launchers of these type of satellites.

2.2 Modules or Cards Used in the Tests

Four modules were placed in the flat-sat for the tests:

1) NanoMind A3200 OBC: An onboard computer based on a 32-bit AVR microcontroller with a clock frequency of up to 64 MHz, external NOR flash memory of 128 MB, external RAM of 32 MB, and an RTC [18].
2) NanoPower P31u EPS: This EPS has two 3.7 V batteries with 2600 mAh each. For the battery charging system, it has up to 10 W DC-DC converters with maximum power point tracking (MPPT) to extract the maximum possible power from the solar cells and has outputs of 3.3 V at 5 A and 5 V at 4 A [19].
3) NanoCom AX100U Radio: A half-duplex radio configurable by software that operates in a UHF frequency range of 430 to 440 MHz. It supports modulations such as FSK/MSK/GFSK/GMSK and supports various frame encapsulation types such as: 32-bits ASM+Golay, AX.25, HDLC+Viterbi encoding, and HDLC+AX.25 [20].
4) Payload 1: A card based on an ARM Cortex-M4 processor microcontroller that incorporates a camera which is connected to the main bus through a gateway using SPI.
5) Payload 2: A card that provides intersatellite communication capabilities using a STINGR simplex modem from GlobalStar. It is connected to the main bus through a gateway using an UART interface.
6) Gateway: A card based on an ARM Cortex-M4 processor microcontroller that incorporates the CubeSat Space Protocol and has the capability to route packets between payloads and the main communication bus. The Gateway is connected to the main bus through CAN and I2C interfaces and supports a variety of digital protocols.

The complete setup can be observed in Figure 1.

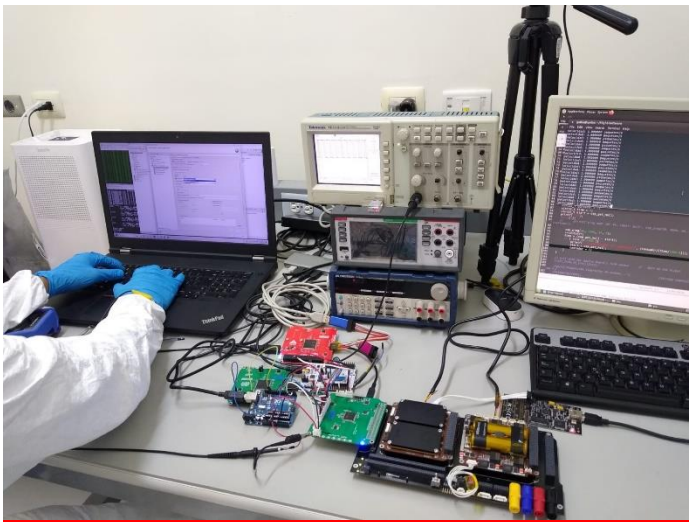


Figure 1. Photograph of flat-sat used in tests.

2.3 CubeSat Space Protocol

The CubeSat Space Protocol (CSP) version 1.6 was used, which is the latest stable version of version 1 of the protocol. CSP is written in C++, so before using it in Python, it needs to be compiled. The compilation tool waf is used for this purpose, considering the library's features. As a result of building the library, two binary files are obtained that must be installed in the system, one file named libcsppy.so containing the encapsulated methods and constants for use in Python, and another named libcsp.so containing the core of the libcsp library. Both files must be installed in the user libraries folders; in the case of Linux systems, it is usually in the path "/usr/lib/".

3 Graphical User Interface development

The graphical interface design was done using the QT Designer tool. Once the design was completed and the control names defined, it was exported to Python code using the pyqt uic module to be able to use it in our code. The graphical interface version can be seen in Figure 2.

The program initialization consists of loading the graphical interface and linking the events or "signals" of the button-type controls to their corresponding methods. The software developed in Python uses the libcsp library to use the CubeSat Space Protocol, however, it is not initialized until the "connect" button located in section 1 of Figure 2 is pressed.

The user interface is divided into 6 sections grouped according to their functionality and the information they display, as depicted in Figure 2:

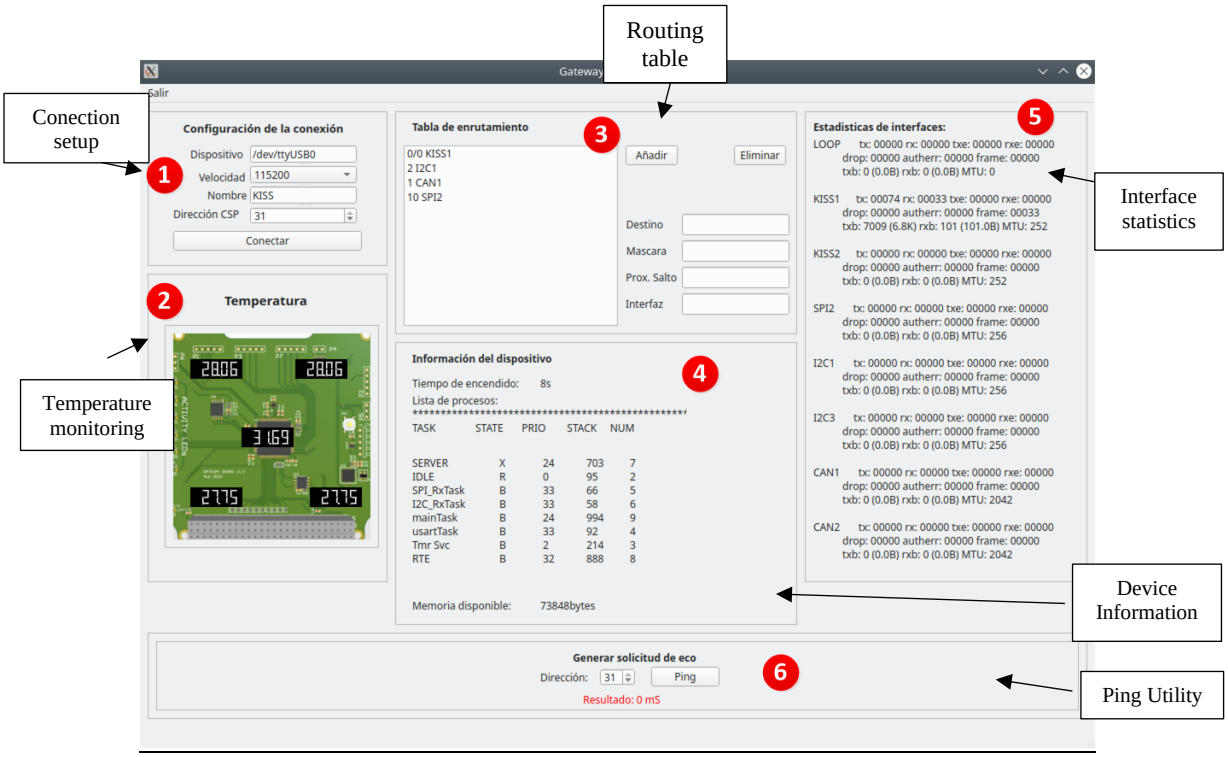


Figure 2. Graphical User Interface for nanosatellite testing.

3.1 Connection Set-Up

To set up the communication between the computer and the card, the following settings must be set: the path of the serial port, the speed of the serial communication, the name of the interface in CSP, and the address of the PC on the network. In this case the values of those settings are shown in Table 1.

Table 1. Settings used for connection between gateway card and Graphical User Interface.

Setting	Value
Device	/dev/ttyUSB0
Speed	115200
Name	KISS
CSP Address	31

When the connect button is pressed, the graphical interface initializes the CSP library and then configures interfaces. In this case the serial interface is configured as the default interface as shown in the block diagram in Figure 3. After the interface initialization, there are three tasks that that needs to be started: the CSP router task, which is responsible for processing the packets; the server task, which allows the processing of incoming connections and their data; and the logger task that allows to record all the telemetry received from the network. The initialization process is depicted in Figure 3 and a extract of code is shown in Listing 1.

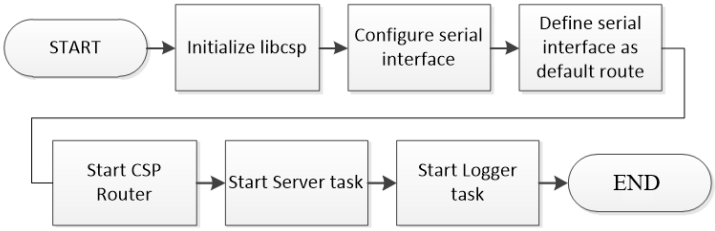


Figure 3. Initialization diagram flow.

```
289     def iniciar_csp(self, addr, device, velocidad, iface_name):
290         # init csp
291         libcsp.init(addr, "pc", "python_gui", "1.2.3", 100, 300)
292         libcsp.kiss_init(device, velocidad, 252, iface_name)
293         libcsp.rtable_set(0, 0, iface_name)
294         libcsp.route_start_task()
295 
296         print("Hostname: %s" % libcsp.get_hostname())
297         print("Model:    %s" % libcsp.get_model())
298         print("Revision: %s" % libcsp.get_revision())
299 
300         print("Routes:")
301         libcsp.print_routes()
302 
303         # start CSP server
304         if ( self._server_thread is None):
305             self._server_thread = threading.Thread(target=self.csp_server, daemon=True)
306             self._server_thread.start()
307         threading.Thread(target=telemetry_client).start()
```

Listing 1. Initialization code used to set up the csp interfaces and logger tasks.

The Server task is responsible for setting up a port that remains listening and allows receiving data sent to the node. The diagram shown in Figure 4 on the left side illustrates the process followed by the task to receive packets sent by other nodes. When the server receives data, it processes them and send them to the corresponding destination port.

The logger task, whose diagram is shown in Figure 4 on the right side, depicts the process that is responsible for processing incoming telemetry. Incoming Telemetry is checked for errors using CRC32 that is a feature of CubeSat Space Protocol, then the length of the packet is verified, and the data is decoded. Finally, the data decoded is used to refresh the GUI and stored to disk.

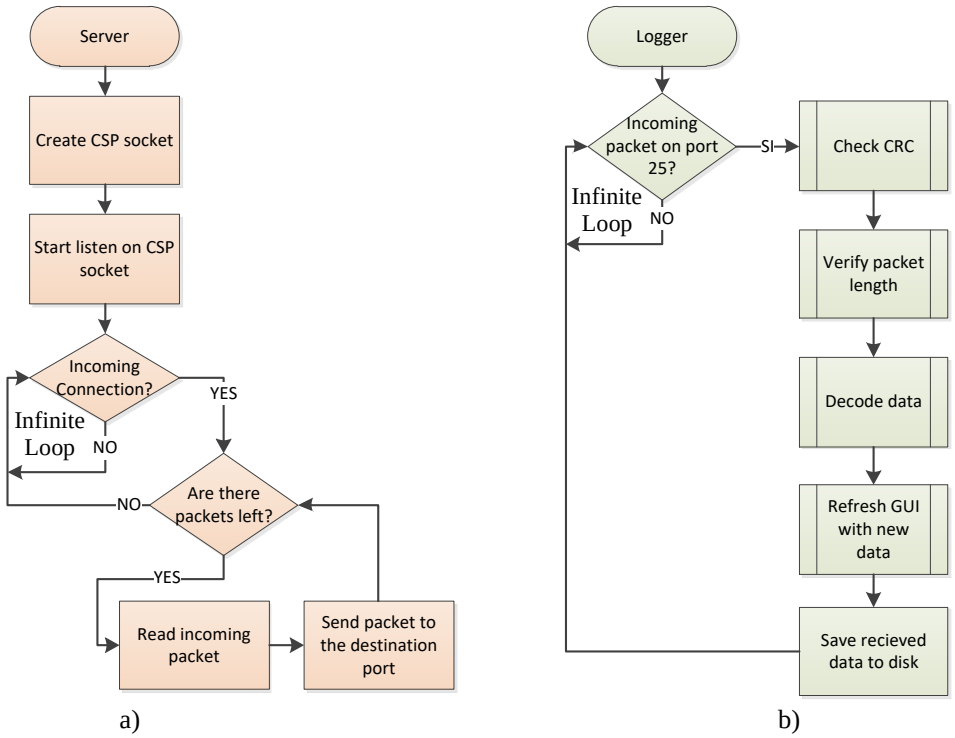


Figure 4. Flow diagrams corresponding to server task (a) and logger task (b).

3.2 Parameters transmitted by OBC categorized by subsystem

The flat sat used in this setup, was configured to emit a housekeeping telemetry every 5 minutes. The housekeeping telemetry includes information from all the subsystems, and it is intended to give the user an idea of the satellite's health. The information is collected by the OBC through many digital protocols, and then assembled in a single packet of 130 bytes that is transmitted to the PC. The complete list of parameters gathered by the OBC are presented in Table 2.

Table 2. Housekeeping data recollected and transmitted by the OBC as telemetry data.

Subsystem	Description of Parameters	Unit or Expected Value
Radio	Power amplifier temperature	°C
	Last Received Signal Strength Indicator	16-bit integer
	Background RSSI	8-bit integer
	Last transmitter power	mW
	Packets transmitted	16-bit integer
	Packets received	16-bit integer
EPS	Power converters voltage - Array	mV
	Battery voltage	mV
	Current in - Array	mA
	Boost converters current	mA
	Battery's output current	mA
	Output current - Array	mA
	Output channels status - Array	0 or 1
	Temperature sensor	°C
OBC	Temperature Sensor	°C
	Pulse Width Modulation (PWM) current	mA
	Uptime in seconds	32-bit number
	Magnetometer - X axis	32-bit float number
	Magnetometer - Y axis	32-bit float number
	Magnetometer - Z axis	32-bit float number
	Gyro - X axis	32-bit float number
	Gyro - Y axis	32-bit float number
	Gyro - Z axis	32-bit float number
Payload 1	Camera status	8-bit number
	Number of images captured	32-bit number
	Number of seconds since last capture	32-bit number
	Space available	32-bit float number (MB)
Payload 2	Number of Channel	8-bit number
	Number of Bursts	8-bit number
	Minimum Burst Interval: Units of 5 s.	01h to 3Ch (5 to 300 s)
	Maximum Burst Interval: Units of 5 s.	02h to 78h (10 to 600 s)
	Status code	8-bit number
	Number of seconds since the device unit last attempted to send a satellite transmission.	16-bit number

	Number of seconds until the device unit attempts to send a satellite transmission.	16-bit number
	Packet size of last or current message.	8-bit number
	Currently waiting on or sending burst number	8-bit number
	Number of seconds until burst transmission number 2	16-bit number
	Number of seconds until burst transmission number 3	16-bit number
	Total messages transmitted in current mode.	16-bit number
	Total Packet transmission count since hard power on.	16-bit number
	STINGR Antenna temperature	°C
	Payload Board temperature	°C
Gateway	Board temperature	°C
	Memory available	32-bit float number in kb
	Interface Status	32-bit number
ADCS	Coarse Sun Sensors Value - Array [+Y, +X, −X, −Y, −Z]	16-bit number
	Solar panel's temperature on +Y axis	32-bit float number
	Solar panel's temperature on +X axis	32-bit float number
	Solar panel's temperature on −X axis	32-bit float number
	Solar panel's temperature on −Y axis	32-bit float number
	Solar panel's temperature on −Z axis	32-bit float number
	Bdot status	−2 to 2
	Bdot value from low pass filter slow	32-bit float number
	Bdot value from low pass filter slow2	32-bit float number
	Value of detumbled state	0 or 1

3.3 Gathering housekeeping information

The process to gathering housekeeping information is executed by the OBC. This module makes requests to all modules of the satellite. The modules respond to the previous OBC's request and the OBC proceeds to process the information. This information helps the OBC to change the satellite behavior according to the status of some variables like the battery voltage, radio status, space available, etc. Once the information is processed, the housekeeping frame is conformed and sent over the CSP network. In this case the OBC has been configurated to send housekeeping every 5 minutes to the address 31 that corresponds to the PC. The connection used in the testing process can be observed in Figure 5.

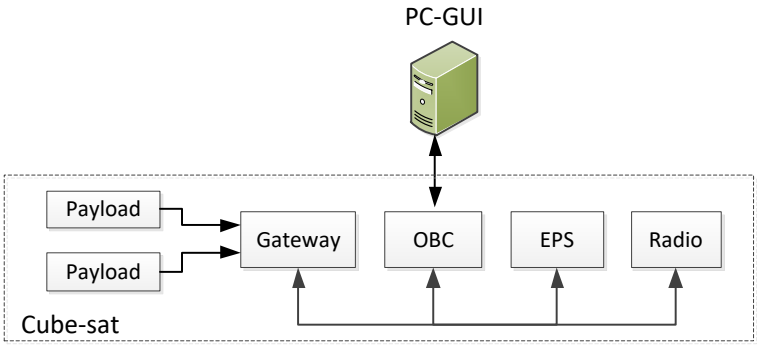


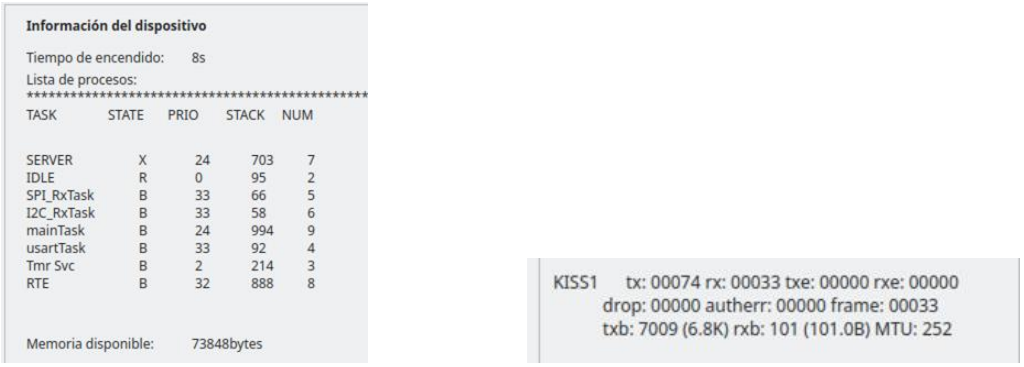
Figure 5. Connection diagram of nanosatellite modules.

3.4 System Status Visualization

In the main window the user can visualize the system status parameters, a space was added in the graphical interface for its display as observed in Figure 2, section 4. The parameters shown include uptime, process list, and available memory. The uptime since the OBC's startup is indicated in seconds. The process list is displayed in a 4-column table as detailed in Figure 6a, where the first column corresponds to the task name, the second column to the task state, the third column to the task priority, which can be a number between 0 and the maximum number of priority levels supported by the hardware, the fourth column to the amount of memory in bytes used by the task, and the fifth to the task identification number. The task state is denoted by the letters: X for Running, R for Ready, B for Blocked, and S for Suspended. Lastly, the amount of available memory in bytes on the device for creating more tasks is shown.

3.5 Communication Interface Statistics Visualization

It was considered useful to visualize communication statistics such as the number of packets sent and received. For this purpose, space was allocated in the graphical interface as observed in Figure 1, section 5. The statistics displayed for each interface include: the interface name, the number of packets transmitted and received by that interface, the number of errors in transmitted packets, the number of errors in received packets, the number of discarded packets, the amount of bytes sent, the number of bytes received, and the maximum transmission unit of the interface. In Figure 6b, an example of the statistics for a serial communication interface of the gateway card can be observed in detail.



a) b)

Figure 6. Details of statistics visualization on GUI: device memory status (a), communication interface status (b).

3.6 Echo Request

To conduct connection tests with other subsystems of the satellite, controls were placed in the graphical interface to send echo requests, as seen in Figure 2, section 6. In the QSpinBox control, the address to which the echo request is to be sent can be selected and pressing the "Ping" button is required to send it. The result of the echo request can be observed in the QLabel control located to the right of the "Ping" button.

3.7 Storage of telemetry data

When a packet is received the interfaces take advantage of CSP features such as CRC32 functions for error detection. If data received is free of errors, the data is decoded and saved to disk later. Relevant information about the communication with satellite modules and information from certain sensors is shown to the user through the GUI. After, checking information for errors and refreshing GUI, the information is stored to disk. Format of stored data can be CSV or JSON. Also is possible to storage information in a SQLite3 database.

4 Results

Several tests were conducted to ensure the correct transmission of data between the OBC card and the graphical interface. These tests are listed below:

4.1 Connection Test

The graphical User Interface is connected to the OBC using a USB-TTL converter. In the OBC configuration the Interface KISS is enabled over this port, at 115200 bps. To test communication, 4 ping requests were sent from PC to the OBC, the responses for the request were visualized in the GUI, the results can be observed in Table 3.

Table 3. Ping test and its response time in milliseconds.

Origin	Destination	Response Time 1	Response Time 2	Response Time 3
PC	OBC	15 ms	14 ms	15 ms
PC	OBC	15 ms	16 ms	15 ms
PC	OBC	15 ms	15 ms	14 ms
OBC	PC	14 ms	14 ms	14 ms
OBC	PC	15 ms	15 ms	14 ms
OBC	PC	16 ms	15 ms	15 ms

4.2 Packet Count Test

To verify the accurate counting of packets sent and received, tests were conducted involving the transmission of bursts of packets with a known number of packets per burst. Each packet sent has a 150 bytes length. The results obtained are presented in Table 4.

Table 4. Packet Count Test.

Origin (Address)	Destination (Address)	Packets Sent	Packets Count	Result of test
PC (31)	OBC (1)	25	25	OK
PC (31)	OBC (1)	50	50	OK
PC (31)	OBC (1)	150	150	OK
OBC (1)	PC (31)	25	25	OK
OBC (1)	PC (31)	50	50	OK
OBC (1)	PC (31)	150	150	OK

4.3 Data decoding and storage test

To ensure the usefulness of the GUI during nanosatellite testing periods, it is necessary to ensure that the information received via CSP can be successfully decoded and stored. For this purpose, tests were conducted where the system was left running for 10 periods of 12 hours each. During this time, the graphical interface received, decoded, and stored housekeeping packets. The total number of packets received was 1440. Out of the total packets, only 0.005% presented errors, which were located in the fields corresponding to information from payload 1, possibly due to programming errors in payload 1. The data was stored on disk in CSV format for easy manipulation with spreadsheet applications.

The data received using the GUI can be processed to generate graphs once they have been decoded. The GUI utilizes libraries such as pandas and matplotlib to generate graphs of the relevant information. For example, in Figure 7, a graph generated from the voltage information of the batteries and the input and output current of the power subsystem can be observed, allowing the test operator to understand the system's consumption and make an estimation of the battery's duration.

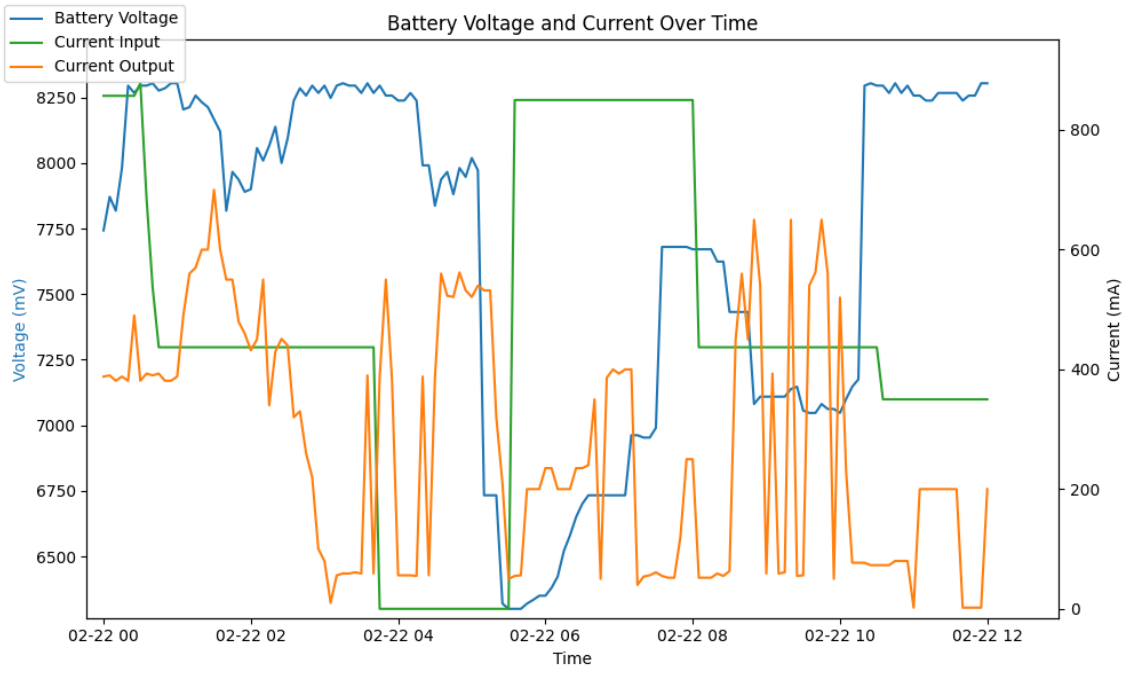


Figure 7. Data of a test period of 12 hours, where the battery voltage is compared with the system's current input and output.

5 Conclusions

The use of a graphical interface allows for data visualization in a more user-friendly manner, and when combined with CubeSat Space Protocol in applications for nanosatellites facilitates connection testing between the various subsystems within a satellite. Furthermore, the use of Python as a programming language allows for rapid development cycles [21], enabling the graphical interface to be adapted to user needs in a short amount of time. In addition, access to the vast array of data processing libraries available in Python such as pandas and matplotlib enables users to visualize test results easily and effectively, with libraries such as pandas and matplotlib being particularly useful for this purpose. Regarding data storage, popular formats such as CSV or JSON can be used, or more robust solutions like MySQL databases can be opted for [22]. The implemented graphical interface worked as expected, and we hope it will be useful for future card tests to be conducted in the laboratory. The graphical user interface worked as expected and it helped us to reduce the time required for repetitive tasks such as connecting and preparing loggers for each of the subsystem modules that were to be tested therefore minimizing human error. Also, we hope that access to clear and effective data visualization supports informed decision-making during testing and development phases.

6 Acknowledgments.

The authors of this article would like to thank the Universidad Popular Autónoma del Estado de Puebla for all the support in the realization of this article.

References

1. Jahku, R. S., & Pelton, J. N. (2014). Why small satellites and why this book? In *Small Satellites and Their Regulation* (pp. 1–12). New York: Springer. SpringerBriefs in Space Development.
2. Johnstone, A. (2022). The CubeSat Program: Cubesat design specification. *California Polytechnic State University, USA, Technical note, Rev.* 14.1. Retrieved from https://static1.squarespace.com/static/5418c831e4b0fa4ecac1bacd/t/62193b7fc9e72e0053f00910/1645820809779/CDS+REV14_1+2022-02-09.pdf [Accessed August 10, 2023].
3. Camps, A. (2020). Nanosatellites and applications to commercial and scientific missions. *Satellite Missions and Technology in Geosciences*, 145–169.
4. Kulu, E. (2023). Nanosats Database. Retrieved from https://www.nanosats.eu/ [Accessed January 27, 2024].
5. Hansen, L. J., Hosken, R. W., & Pollock, C. H. (1999). Spacecraft computer systems. In *Space Mission Analysis and Design* (pp. 645–684). California: Microcosmos Press.
6. Wertz, J. R., & Larson, W. J. (1999). Spacecraft subsystems. In *Space Mission Analysis and Design* (pp. 353–518). California: Microcosmos Press.
7. Alminde, L., Christiansen, J., Kaas Laursen, K., Midtgaard, A., Bisgard, M., Jensen, M., ... & Le Moullec, Y. (2012). Gomx-1: A nano-satellite mission to demonstrate improved situational awareness for air traffic control.
8. Prasai, S. (2012). *Access control of NUTS uplink* (Master's thesis, Institutt for telematikk).
9. Challenger Pérez, I., Díaz Ricardo, Y., & Becerra García, R. A. (2014). El lenguaje de programación Python. *Ciencias Holguín, 20*(2), 1–13.
10. Summerfield, M. (2007). *Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming*. Pearson Education.
11. Qt Company. (2023). Qt for Python. Retrieved from https://doc.qt.io/qtforpython-5/index.html [Accessed July 3, 2023].
12. Surivet, A. (2021). Integration and validation of a nanosatellite flight software (ESA OPS-SAT project). Stockholm.
13. Grillo, V. (2019). Python & Qt, powerful tools for technical computing. In *90th S&V Symposium*. Atlanta.
14. Esposito, A. (2019). CubeSatControl Centre for the management of telemetry, telecommand and operations based on the CCSDS standards. *Politecnico di Torino*, Torino.
15. Obiols-Rabasa, G., Corpino, S., Mozzillo, R., & Stesina, F. (2015). Lessons learned of a systematic approach for the E-ST@R-II CubeSat. In *66th International Astronautical Congress*. Jerusalem.
16. Rodríguez Delgado, C. A. (2023). *Gestión de requerimientos para el desarrollo de un sistema integrado de pruebas para CubeSats*. Tecnológico de Costa Rica, Cartago.
17. Tapsawat, W., et al. (2018). *IOP Conference Series: Materials Science and Engineering, 8th TSME-International Conference on Mechanical Engineering*, Bangkok, Thailand.
18. GOMspace. (n.d.). NanoMind A3200 Datasheet. Retrieved from https://gomspace.com/UserFiles/Subsystems/datasheet/gs-ds-nanomind-a3200_1006901-117.pdf [Accessed July 3, 2023].
19. GOMspace. (n.d.). NanoPower P31u. Retrieved from https://gomspace.com/UserFiles/Subsystems/datasheet/gs-ds-nanopower-p31u-30.pdf [Accessed June 2023].
20. GOMspace. (n.d.). NanoCom AX100. Retrieved from https://gomspace.com/UserFiles/Subsystems/datasheet/gs-ds-nanocom-ax100-33.pdf [Accessed June 2023].
21. Saabith, A. S., Vinothraj, T., & Fareez, M. (2020). Popular Python libraries and their application domains. *International Journal of Advance Engineering and Research Development, 7*(11).
22. Deedar, M. H., & Hernández, S. M. (2019). Extending a flexible searching tool for multiple database formats. In *Emerging Trends in Electrical, Communications, and Information Technologies: Proceedings of ICECIT-2018* (pp. 25–35). Singapore: Springer Singapore.