SatNOGS-COMMS: An Open-Source Communication Subsystem for CubeSats

Manolis Surligas*, Agis Zisimatos*, Aris Nikas*, Dimitris Moustroufis*,
Pierros Papadeas*, Manthos Papamathaiou*, Daniel Bita*,
Dimitris Zournatzis*, Victoria Malyshkina*
*Libre Space Foundation, Athens, Greece

***Abstract*—SatNOGS-COMMS is an open-source, open-hardware communications subsystem for CubeSats, developed by the Libre Space Foundation in collaboration with the European Space Agency (ESA). This innovative system combines advanced hardware and software to meet the challenges of CubeSat missions while promoting accessibility and flexibility through an open ecosystem. The COMMS transceiver provides separate half-duplex RF frontends for UHF and S-Band operation. It adheres to CCSDS recommendations for Category A spacecraft, supporting user-selectable framing schemes (CCSDS, IEEE 802.15.4, AX.25). Baseband modulations include BPSK, QPSK, MSK, and FSK, with data rates up to 50 kbps (UHF) and 1 Mbps (S-Band), depending on the chosen modulation and coding. The UHF frontend operates between 395 and 450 MHz, while the S-Band operates at 2025–2110 MHz (uplink) and 2200–2290 MHz (downlink). Both frontends offer up to 32 dBm transmit power, complying with SFCG 21–2R4 emissions standard.**

I. Introduction

In our previous work presented in [1], we provided details about the hardware architecture of the transceiver, its capabilities, the performance characteristics of the transmitter, as well as the system's power consumption under various workload scenarios.

In this paper, we present the software architecture that controls the entire transceiver, the design considerations that enable seamless integration and extensibility in diverse mission contexts, as well as the performance characteristics of the UHF and S-Band receivers. In addition, this paper provides details about the automated testing procedure, and the integration of the YAMCS [2] mission control center.

II. Flight Software Architecture

A. Firmware Architecture and Platform Abstraction

The main control firmware of the SatNOGS-COMMS system runs on the STM32H743 [3] Micro-Controller Unit (MCU). It is responsible for interfacing with all available board subsystems, collecting and analyzing telemetry data, and executing autonomous actions to ensure the optimal operation and protection of the transceiver.

The reference firmware that ships with the board is based on Zephyr-RTOS [4] version 4.1.0. However, to maximize flexibility and accommodate varying mission requirements, the firmware components have been designed to remain agnostic to the underlying RTOS. This design decision enables the firmware to be adapted for use with alternative real-time operating systems, depending on user preference or mission constraints.

At the heart of this design lies the *libsatnogs-comms* [5] library, a platform-agnostic C++ library that provides full control over all board components. It exposes a comprehensive API and encapsulates the entire board configuration within a singleton instance, ensuring that all subsystems are correctly initialized and easily accessible from any part of the firmware, including across concurrent tasks (Figure 1).

To support RTOS independence, the library uses C++ polymorphism. Platform-specific operations, such as DMA handling, message queues, or peripheral access, are defined as pure virtual methods, requiring the user to implement them based on their chosen platform. This abstraction layer allows developers to adopt any RTOS or bare-metal

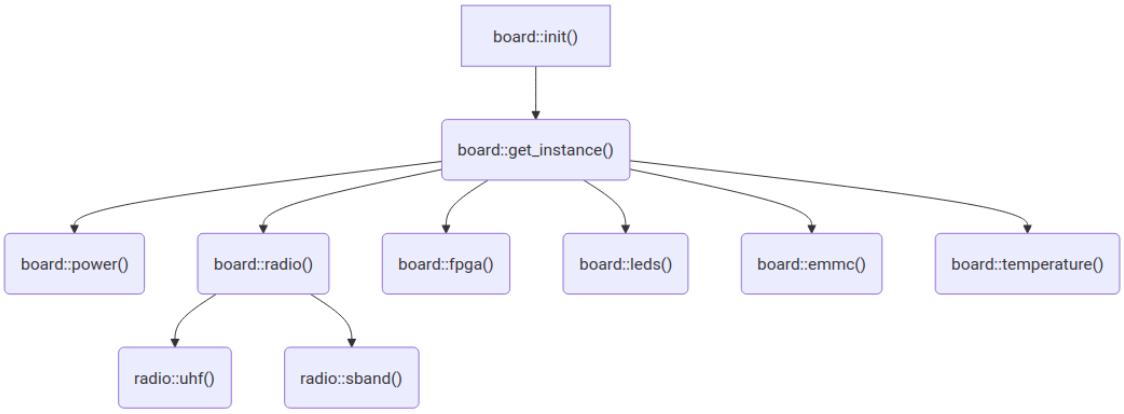

Fig. 1: Software components architecture

system, not just Zephyr-RTOS, with minimal effort without requiring deep knowledge of the underlying hardware design.

One of the typical challenges in embedded C++ development is the use of Standard Template Library (STL) containers, which often rely on dynamic memory allocation. This can lead to memory fragmentation and non-deterministic behavior, making them unsuitable for real-time or safety-critical applications. To address this, libsatnogs-comms integrates the etlcpp [6], a header-only library that replicates much of the STL's functionality using a static memory model. This approach ensures predictable memory usage and deterministic execution, which are essential for embedded mission-critical systems.

B. Configuration and Customization

The SatNOGS-COMMS software offers a powerful and flexible interface for configuration and customization, addressing a wide range of mission-specific requirements. This is achieved by taking advantage of the Kconfig system [7] and devicetree overlays [8] provided by the Zephyr-RTOS.

Kconfig enables clear and structured compile-time initialization of numerous firmware parameters. Although most of these parameters can be adjusted during runtime via telecommands, their initial defaults and reset values are determined by the Kconfig configuration. This approach ensures both flexibility during operations (e.g. qualification vs flight configuration, development, etc.) and consistency in baseline behavior. Figure 3 depicts some of the configuration options available from the corresponding menu interface.

In contrast, I/O port configuration is handled through devicetree overlays. SatNOGS-COMMS includes a comprehensive set of predefined overlays, covering many common scenarios and tailored to the hardware capabilities. These overlays define, for example, the pin assignments for UART and I2C interfaces on the PC104, antenna deployment I/O options, and additional logging UART ports (e.g. Listing 1).

The overlay system is designed to be highly flexible and easily extensible. If the built-in overlays do not fully meet the requirements of a mission, users can easily provide custom overlay files to the build system. This capability allows for seamless adaptation of the firmware to new hardware setups or evolving mission needs.

C. Adding Mission-Specific Features

While the SatNOGS-COMMS reference firmware is capable of operating out-of-the-box without any modifications, this setup typically serves as a starting point. In practice, every mission presents unique requirements, necessitating tailored behavior and additional functionality. To accommodate this, the firmware architecture

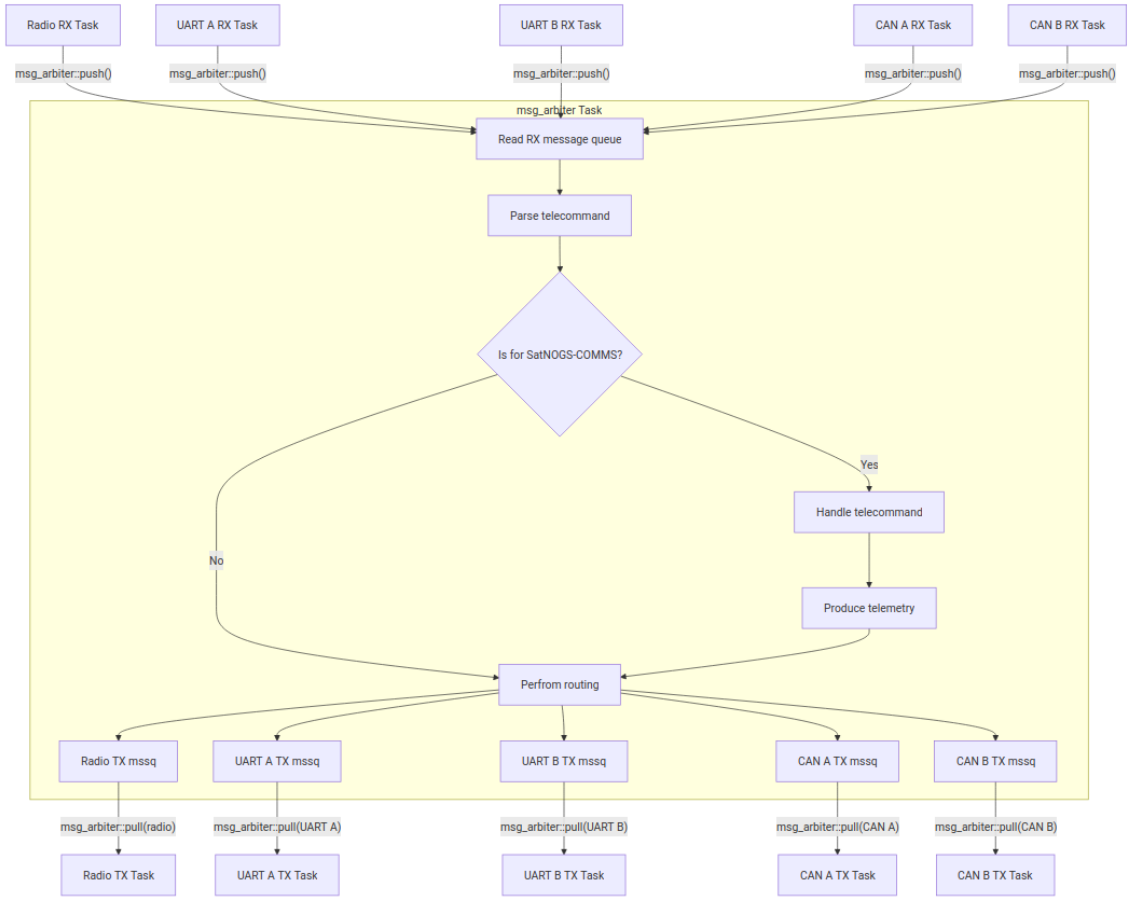

Fig. 2: IO messages handling and routing

```
 /*
  * Sets the UHF antenna in GPIO mode using the
  * pins ANT_DEP_A and ANT_DET_A
  */
&uhf_antenna {
    status = "okay";
    deploy-gpios = <&gpiod 13 GPIO_ACTIVE_HIGH>;
    sense-gpios = <&gpioc 5 GPIO_ACTIVE_LOW>;
};
```

Listing 1: Overlay example for the UHF antenna GPIO pins selection

has been intentionally designed to be easily extensible and hackable, all without requiring direct modifications to the original codebase. This modular design not only simplifies code maintenance but also encourages community contributions, allowing all users to benefit from enhancements through the collaborative nature of open-source software.

The recommended approach for integrating SatNOGS-COMMS into a mission-specific project is by including it as a Git submodule. This setup ensures strict version control, facilitates tracking of upstream changes, and provides a clean workflow for contributing improvements or bug fixes, while simultaneously working on the mission's custom code. The exact process is documented at the relevant section of the firmware documentation [9].

Furthermore, the firmware offers several hooks

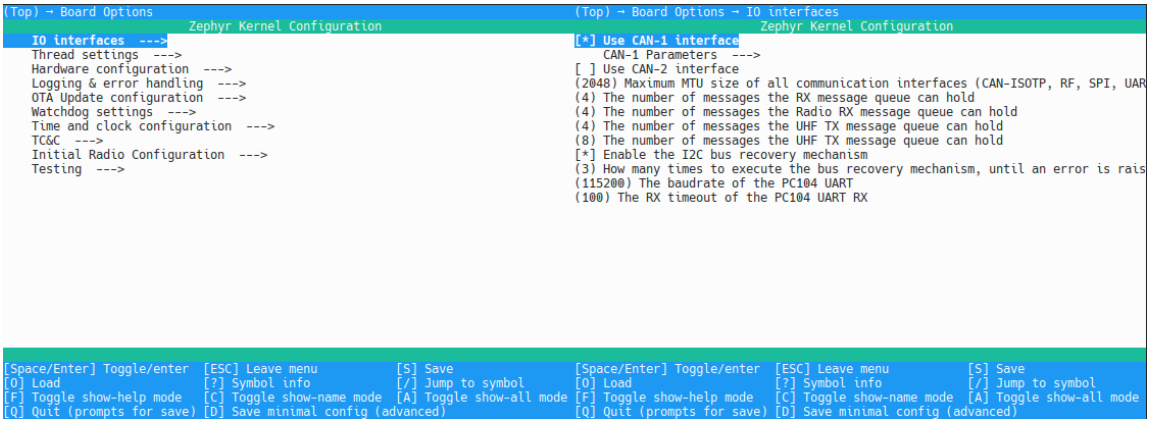

Fig. 3: Kconfig configuration menu interface

and callbacks at strategically selected points throughout the codebase, utilizing the available methods of the *mission* class. These extension points allow developers to seamlessly inject mission-specific functionality with minimal effort, and without altering the core firmware. This strategy ensures high maintainability, reusability, and consistency across missions.

Behind-the-scenes, this is accomplished using the *etl::delegate* class of the etlcpp library. This class provides a type-safe and reliable mechanism for user to define their own callbacks and register them at the corresponding hooks. The procedure is quite easy as the Listing 2 example shows. Users have just to provide the implementation of the desired callback and register it at the desired hook utilizing the available methods of the *mission* class.

D. Message Handling

SatNOGS-COMMS implements a unified and extensible framework for handling I/O messages across all available communication interfaces. At the core of this system is the *msg_arbiter* class, which serves as the central component for managing message reception, processing, and distribution within the transceiver. The *msg_arbiter* is responsible for several key tasks:

- **Receiving Messages:** It listens to and collects incoming messages from all configured input interfaces.

```cpp
#include <mission.hpp>
#include <startup.hpp>
#include <satnogs-comms-lib/board.hpp>
#include <zephyr/kernel.h>

void
post_init() {
  // Do stuff
}

int
main(void)
{

  // This should be the first call!
  auto &startup = startup::get_instance();
  auto &mission = mission::get_instance();
  auto &err     = error_handler::get_instance();

  // Custom code here
  // ....
  // ...
  // Register mission specific functionality
  auto pinit
     = etl::delegate<void(void)>::create<post_init>();
  mission.register_post_init(pinit);

  startup.prepare();

  // Custom code here
  // ....
  // ...
  startup.start();

  // Custom add threads, calls, etc
  // ....
  // ...
}
```

Listing 2: Example of mission specific functionality injection

- **Message Queuing:** It maintains a thread-safe, centralized message queue for incoming frames, ensuring safe access in concurrent scenarios. Additionally, each output interface is assigned its own dedicated queue to optimize message delivery and minimizing blocking periods waiting for the interface availability
- **Message Processing:**
  - **Local Telecommand Handling:** It interprets and processes telecommands intended for the SatNOGS-COMMS subsystem itself.
  - **Telecommand Forwarding:** It forwards commands to the appropriate output interface for routing to the correct subsystem elsewhere in the satellite.

This hybrid queuing architecture—where all input interfaces feed into a single queue, while output interfaces each maintain individual queues—strikes an effective balance between memory efficiency and throughput. It ensures that messages are processed in a consistent, timely manner, while minimizing overhead and maintaining scalability across diverse mission configurations.

E. Error Handling and Logging

Error detection, reporting, and recovery are among the most critical aspects of any satellite software system—especially for a core subsystem like communications. Faults must not only be handled autonomously when possible, but also logged in sufficient detail to enable effective remote diagnostics and troubleshooting by ground operators. To meet these needs, SatNOGS-COMMS provides a dedicated `logger` class that ensures a consistent and flexible logging strategy across the system. The logger supports multiple log targets, including:

- **SWO (Serial Wire Output)** mostly used during debugging with a debug probe
- **User-selectable UART** for runtime diagnostics or propagating logs to another subsystem (e.g. OBC)
- **RAM-based ring buffer** (non-persistent across reboots) with remote telemetry support
- **eMMC storage** for persistent log files with TC&C support
- **Backup SRAM**, retaining logs across reboots and power cycles (if an RTC battery is available) also available by the remote telemetry while in-flight

```
/**
* @brief Severity levels of exceptions
*
* @see FDIR analysis at
* https://cloud.libre.space/s/xzskpy8m3Nb54YL
*/
enum class severity : uint8_t
{
  CATASTROPHIC = 0,
  CRITICAL = 1,
  MAJOR = 2,
  MINOR = 3,
  NONE  = 4
};
```

Listing 3: Exception severity levels

For error management, SatNOGS-COMMS adopts C++ exceptions rather than traditional error codes. Exceptions offer a more structured and powerful mechanism for managing faults, enhancing code clarity, safety, and modularity. While exception support increases firmware size, this is well accommodated by the STM32H743, which provides enough flash memory. Each exception thrown by the `libsatnogs-comms` control library includes detailed context and metadata, enabling fine-grained responses to system faults. Key features of the exception handling system include:

- **Severity Levels**: Each exception is classified based on its criticality (Listing 3), allowing the firmware to initiate appropriate recovery or escalation procedures.
- **Source Context**: Exceptions record the filename and line number from which they were thrown, aiding rapid debugging and traceability.
- **Logging**: Each exception defines a verbose message providing detailed context for development and debugging and terse, compact message optimized for in-flight logging, conserving bandwidth and storage.

Based on the severity level of each exception caught, the error reporting and recovery mechanism after performing the proper log operations, decides

the evasive actions. These actions may just ignore the error and retry, retry again for a specific amount of times, or perform a reboot if the integrity of the system cannot be guaranteed. This approach ensures that SatNOGS-COMMS is both resilient and observable, with mechanisms in place to intelligently and transparently handle unexpected scenarios, both during development and live operations. A simplified example of the afforementioned mechanism is presented at Listing 4.

```
void
error_handler::handle(const scl::exception &e)
{
  log(e);
  switch (e.get_severity()) {
  case scl::exception::severity::CATASTROPHIC:
  case scl::exception::severity::CRITICAL:
    system_reboot();
    break;
  case scl::exception::severity::MAJOR:
    if (m_last_errno == e.get_errno()) {
      m_errno_cnt++;
    } else {
      m_last_errno = e.get_errno();
    }
    if (m_errno_cnt > CONFIG_MAX_MAJOR_ERRORS) {
      system_reboot();
    }
    break;
  default:
    break;
  }
}

void task_x() {
while (1) {
try {
  rx_conf.freq = s.get<param::SBAND_RX_FREQ>();
  radio.rx_async(radio::interface::SBAND, rx_conf);
} catch (const scl::pll_ls_exception &e) {
  /* Maybe we can recover! */
  auto &err = error_handler::get_instance();
  err.handle(e);
  radio.enable(false);
  radio.enable(true);
} catch (const scl::radio_exception &e) {
  auto &err = error_handler::get_instance();
  err.handle(e);
} catch (const std::exception &e) {
  auto &err = error_handler::get_instance();
  err.handle(e);
}
}
```

Listing 4: Error recovery simplified example

F. Bootloader

The bootloader used in SatNOGS-COMMS is MCUBoot [10], a widely adopted, secure, and reliable bootloader designed for embedded systems. Among its key advantages are support for multiple firmware versions, robust integrity verification via cryptographic hashing, rollback capabilities, and the ability to verify asymmetrically signed firmware images, enhancing overall system security. MCUBoot supports several methods for performing firmware upgrades. For SatNOGS-COMMS, the selected approach is Execute-In-Place (XIP). In this scheme, flash memory is partitioned into multiple slots of fixed, known size, with each slot capable of storing an independent firmware image. At startup, MCUBoot performs a validity check on each firmware slot using the SHA-256 fingerprint of the firmware by comparing the slot contents and the precomputed fingerprint stored in the header section of the firmware. If the verification passes, the corresponding firmware is executed directly from its storage address in memory, eliminating the need for copying or swapping images. If a slot fails validation, MCUBoot automatically proceeds to the next available valid firmware image. This method increases system reliability, even in cases where specific flash regions may be physically damaged, as the bootloader can continue booting from one of the remaining valid slots. The primary trade-off of XIP is that each firmware binary must be specifically compiled for the memory address associated with its target slot. To mitigate this complexity, the SatNOGS-COMMS codebase includes tooling that simplifies slot-specific firmware builds. Developers can specify the desired slot number in the build system and the resulting image can then be directly sent to the transceiver without additional configuration.

III. Mission Control

SatNOGS-COMMS uses the YANCS framework [2] for mission control operations, utilizing the XTCE (XML Telemetric and Command Exchange) schema as specified in CCSDS 660.0-B-2 [11]. While this standards-based architecture does not necessarily simplify the mission database, it greatly expands its capabilities, ensures interoperability across diverse ground infrastructures, and facilitates broader collaboration on future mission requirements.

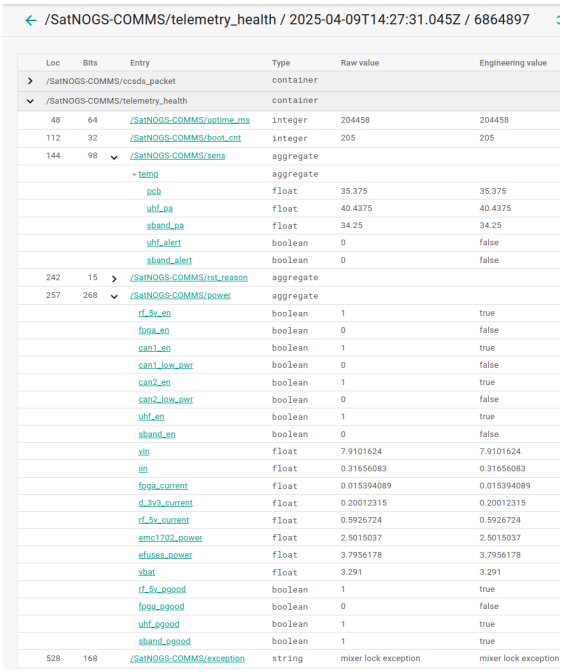

Fig. 4: Decoded health telemetry packet visualization in YAMCS web interface

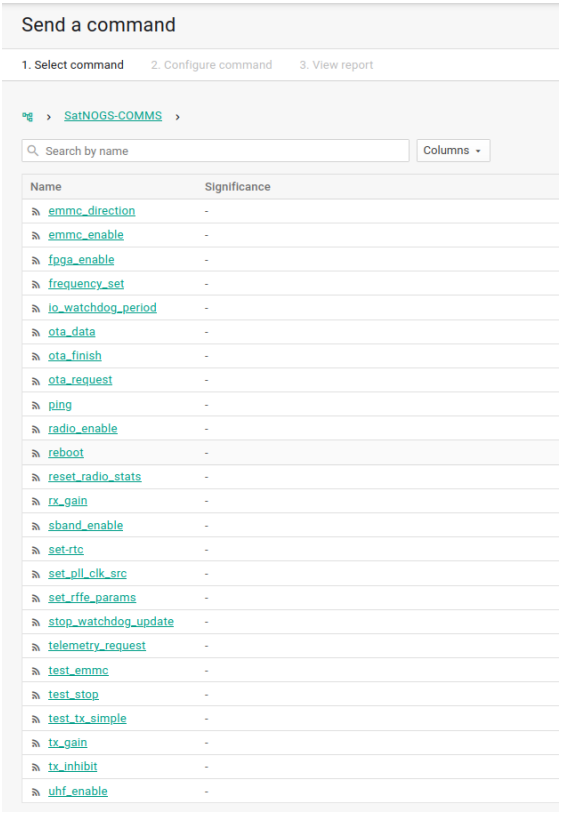

Fig. 5: Set of telecommands in YAMCS web interface

Telemetry and telecommand (TM/TC) data are encapsulated in CCSDS-compliant packets with standardized header fields such as version, Application Process Identifier (APID), and sequence count. These fields enable the SatNOGS-COMMS firmware to efficiently parse, filter, and route incoming commands, including discriminating by SPACEID in multi-spacecraft scenarios and by APID ranges for mission-specific functionality.

YAMCS fully supports XTCE-compliant definitions, automatically parsing parameters, containers, calibrators, and limit configurations. Notably, it accommodates multiple calibration methods (linear, polynomial, exponential, switch-based), enabling data transformations to be performed on the ground rather than on the satellite. This approach preserves raw telemetry for potential debugging and minimizes on-board complexity. Furthermore, XTCE alarm configurations allow automated fault detection using five standardized severity levels (watch, warning, distress, critical, severe), thereby improving operator situational awareness and accelerating decision-making during nominal and off-nominal operations.

The system supports both RF and CAN FD transport layers, which are connected to YAMCS through a custom gateway, as YAMCS natively supports only TCP and UDP protocols. Furthermore, YAMCS features a plugin-based architecture that simplifies the incorporation of additional data links and custom interfaces.

Internally, YAMCS stores all mission data in a RocksDB-backed archive, serializing packet entries via Protocol Buffers and indexing them by timestamp and parameter ID. This architecture serves as a foundation for the system's testing framework and enables advanced features such as real-time visual-

Fig. 6: Set of telemetry parameters in YAMCS web interface

ization, historical data acquisition and efficient data export through YAMCS web interface or API.

Meanwhile, on the firmware side, each telecommand is represented by a dedicated class that includes a deserialization method, which uses *etl::bit_stream* [12] for bit-level data manipulation. By operating on statically allocated arrays and avoiding dynamic memory, these bit-streams deliver predictable, deterministic performance, making them ideal for real-time, resource-constrained embedded space applications.

IV. Testing

SatNOGS-COMMS also ships with powerful and extensive testing suite. This suite utilizes the Python-based YAMCS Client [13] in conjunction with Robot Framework [14], providing a fully automated HIL (Hardware-in-the-Loop) testing suite out of the box. The project uses this testing suite throughout the development by integrating it with the Gitlab CI/CD process.

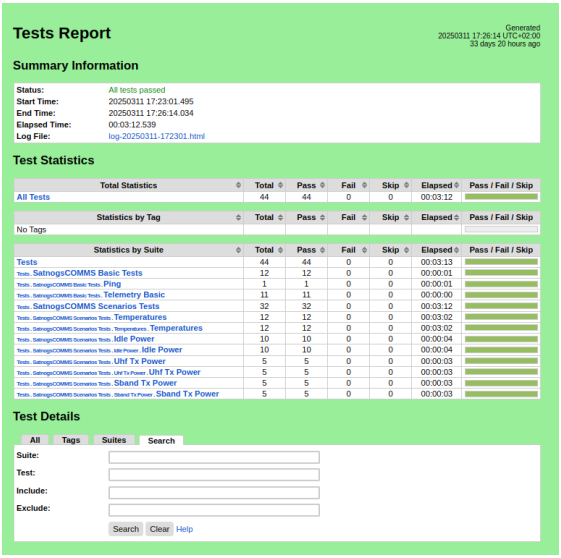

Fig. 7: Robot framework web-based report

Robot Framework offers a powerful and expressive syntax that simplifies the creation of multiple test cases, while also enabling advanced customization through native integration with the Python programming language. Listing 5 illustrates a simplified test case designed for validating the power subsystem of the transceiver. This test checks whether the power-related telemetry values reported by the transceiver fall within predefined acceptable ranges. To retrieve these values, the Robot test case issues the appropriate telecommand using the YAMCS Python interface, as previously discussed.

Robot framework produces also extensive reports and log outputs in multiple formats (pdf, html, text) containing a plethora of information including the exact execution time, the duration, the result as well as the failing criteria in case of failures. An example of such a report can be found in Figure 7.

The development of the testing suite is, and will continue to be, closely aligned with both the ongoing software development and future hardware

```
*** Settings ***
Resource        ${CURDIR}/../common.resource
Variables       ${INPUTS}/basic/02__telemetry_basic.yaml

Test Tags       telemetry    basic


*** Variables ***
@{power_keys}
...                 vin
...                 iin
...                 fpga_current
...                 d_3v3_current
...                 rf_5v_current
...                 emc1702_power
...                 efuses_power
...                 vbat


*** Test Cases ***
Issue Basic Telemetry and Fetch Power
    ${ct}    Issue Command and Get Time    ${cmd}[tlm_req]    ${payload}
    ${res}    Fetch Parameter    ${params}[power]    ${ct}
    Set Suite Variable    ${res}

Validate Boolean Power States
    Dictionaries Should Be Equal
    ...    ${res}
    ...    ${expected_bool}
    ...    ignore_keys=${power_keys}

Validate Power Limits
    FOR    ${key}    IN    @{power_keys}
        Validate Param In Limits    ${key}    ${res}
    END


*** Keywords ***
Validate Param In Limits
    [Arguments]    ${key}    ${status}
    ${val}    Set Variable    ${status}[${key}]
    ${lim}    Set Variable    ${${key}_limits}
    ${lower}    Set Variable    ${lim}[lower]
    ${upper}    Set Variable    ${lim}[upper]
    Log    \n[${key}] = ${val}, expected range: ${lower} - ${upper}
    Should Be True    ${lower} <= ${val} and ${val} <= ${upper}
```

Listing 5: *02__telemetry_basic.robot*

modifications or customizations. By integrating extensive testing into the platform, we are able not only to identify and resolve bugs during the software lifecycle, but also to support and streamline the production and manufacturing process. This is achieved by providing hardware engineers with an automated testing framework that offers valuable insights into potential faults. Furthermore, users can leverage this infrastructure to validate their own setup and effortlessly implement custom test cases, enabling seamless integration of SatNOGS-COMMS into their satellite with minimal overhead.

V. Performance

A. Receiver Performance Characterization

In our previous work [1], mainly the transmission performance metrics were presented. In this paper, we extend the system characterization with the receiver performance too. To evaluate the performance of the receiver, a series of controlled experiments were conducted. The test parameters were carefully selected to closely replicate real-world conditions likely to be encountered during a LEO mission. These parameters are presented in Table I. Note that for the UHF Uplink the worst case of ground station transmission power is taken into consideration. This should be at the 401-403 MHz band, where the 5.264A rule of the ITU regulations is in effect. The 5.264A dictates that the maximum EIRP for any earth station should not exceed 7 dBW in any 4 kHz band.

The first set of experiments focuses on characterizing the sensitivity of the receiver by measuring the Frame Delivery Ratio (FDR) under varying signal levels at the input of the receiver. A reference ground station was employed to transmit a total of 1000 frames containing randomized data.

For the UHF interface, both FSK (modulation index 1) and MSK modulation schemes were tested at a bitrate of 50 kbps. For the S-Band interface, MSK modulation was used with bitrates of 100, 200, and 400 kbps, respectively.

Across all experiments, the following communication parameters were kept fixed:

- **Preamble**: 32-bits
- **Synchronization Word**: CCSDS Attached Sync Marker (ASM)
- **Error Correction**: CCSDS Reed-Solomon RS(255,223)
- **Randomization**: CCSDS randomizer
- **Frame Size**: 512 bytes for UHF, 1024 bytes for S-Band

B. Doppler Tolerance Evaluation

The second set of experiments aims to evaluate the receiver's ability to handle Doppler frequency offsets typical of a spacecraft in LEO, assuming no Doppler compensation is applied by the ground station. To ensure representative results, the input signal levels for these tests were selected based on the sensitivity thresholds identified in the previous experiments. Specifically, the lowest input level at which the receiver exhibited no significant degradation in FDR. These levels are summarized in Table II.

TABLE I: Link Budget Parameters for UHF and S-Band Uplink

Transmitter - Parameters	**UHF Uplink**		**S-Band Uplink**	
Transmitter Power (dBm)	23.50	23.50	36.00	36.00
Transmitter Power (dBW)	-6.50	-6.50	6.00	6.00
Antenna circuit loss (RFDN) (dB)	-1.00	-1.00	-1.00	-1.00
Antenna gain (dBi)	14.00	14.00	35.80	35.80
Δ3dB antenna (deg)	30.00	30.00	5.00	5.00
EIRP (dBW)	6.50	6.50	40.80	40.80
EIRP (dBm)	36.50	36.50	70.80	70.80
Path - Parameters				
Elevation angle (deg)	5.23	25.03	5.23	25.15
Altitude (km)	600.00	600.00	600.00	600.00
Slant Range (km)	2192.92	1125.25	2192.92	1121.96
Free Space Loss (dB)	-151.35	-145.56	-165.75	-159.92
Atmospheric/Ionospheric Loss (dB)	-0.13	-0.03	-0.45	-0.10
Rainfall Loss (dB)	-0.003	-0.001	-0.002	-0.002
Total Path Loss (dB)	-151.48	-145.59	-166.20	-160.02
Receiver - Parameters				
Polarization loss (dB)	-3.00	-3.00	-3.00	-3.00
Pointing loss (dB)	0.00	0.00	-1.50	-1.50
Δ3dB antenna (deg)	close to omni	close to omni	60.00	60.00
Pointing accuracy (deg)	$\sim$20	$\sim$20	$\sim$20	$\sim$20
Antenna circuit loss (RFDN) (dB)	-1.00	-1.00	-1.00	-1.00
Antenna gain (dBi)	1.90	1.90	6.50	6.50
Total gain antenna(dB)	-2.10	-2.10	1.00	1.00
Received Signal (dBm)	-117.08	-111.19	-94.40	-88.22

TABLE II: Input Power Levels for Doppler Testing

Interface	**Modulation and Bitrate**	**Input Power Level (dBm)**
UHF	FSK @ 50 kbps	-117
UHF	MSK @ 50 kbps	-112
S-Band	MSK @ 100 kbps	-108
S-Band	MSK @ 200 kbps	-105
S-Band	MSK @ 400 kbps	-96

As illustrated in the corresponding figures, the receiver demonstrates robust performance across the majority of Doppler-induced frequency offsets encountered during a typical satellite pass.

However, for the UHF interface and the S-Band MSK at 100 kbps, a noticeable degradation in performance was observed under extreme offset conditions. This behavior has been traced to a specific digital filter configuration within the transceiver.

Adjusting this filter to a more relaxed setting is expected to improve Doppler tolerance and allow the receiver to handle the full offset range. However, this improvement comes at the cost of approximately 3 dB reduction in sensitivity.

VI. Conclusion

In this paper, we presented the ongoing development of the SatNOGS-COMMS transceiver, including its software architecture and recent advancements in testing infrastructure. Additionally, we provided a preliminary characterization of the receiver's performance for selected modulation and coding schemes. Although this evaluation is not yet exhaustive, it offers valuable insights into the system's capabilities and potential. As an open-source and open-hardware initiative, SatNOGS-COMMS continues to evolve through active community contributions and ongoing development. The project's

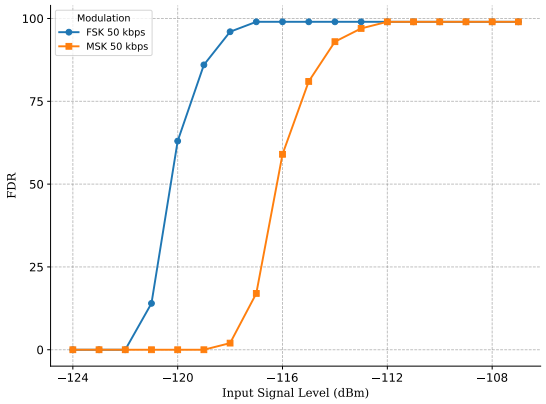

Fig. 8: UHF RX performance

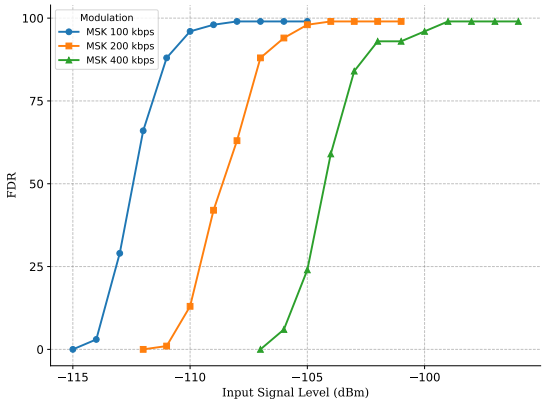

Fig. 9: S-Band RX performance

development can be tracked through its GitLab repositories [15].

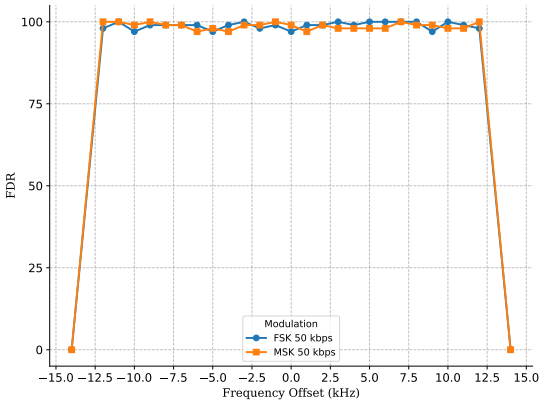

Fig. 10: UHF RX performance versus Frequency Offset

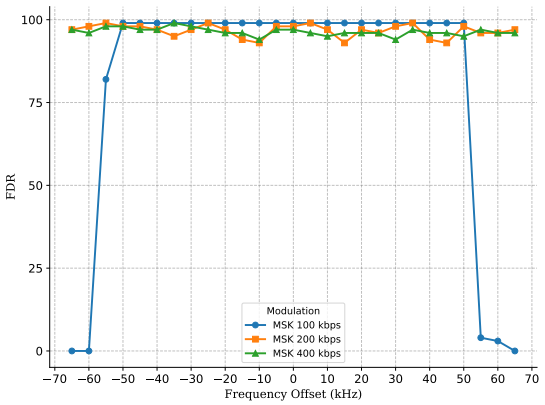

Fig. 11: S-Band RX performance versus Frequency Offset

References

[1] M. Surligas, A. Zisimatos, I. Daradimos, A. Nikas, D. M. P. Papadeas, M. Papamathaiou, A.-P. Damkalis, V. Tsiligiannis, and V. Malyshkina, "Satnogs-comms: Turnkey nanosatellite communications," *Small Satellite Conference*, 2024.

[2] "Yamcs Mission Control software." [Online]. Available: https://yamcs.org

[3] STMicroelectronics, "RM0433 Reference Manual for STM32H742, STM32H743/753 and STM32H750."

[4] "Zephyr-RTOS project." [Online]. Available: https://www.zephyrproject.org

[5] Libre Space Foundation, "SatNOGS-COMMS platform independent Control Library." [Online]. Available: https://gitlab.com/librespacefoundation/satnogs-comms/libsatnogs-comms

[6] "Embedded Template Library: A C++ template library for embedded applications." [Online]. Available: https://www.etlcpp.com

[7] "Zephyr-RTOS Configuration System." [Online]. Available: https://docs.zephyrproject.org/latest/build/kconfig/index.html

[8] "Zephyr-RTOS Devicetree." [Online]. Available: https://docs.zephyrproject.org/latest/build/dts/index.html

[9] "SatNOGS-COMMS: Adding Mission specific features." [Online]. Available: https://librespacefoundation.gitlab.io/satnogs-comms/satnogs-comms-software-mcu/group__

mission-specific.html
[10] “MCUboot Bootloader.” [Online]. Available: https://docs.mcuboot.com
[11] Consultative Committee for Space Data Systems, “XML Telemetric and Command Exchange–Version 1.2,” CCSDS 660.0-B-2, Blue Book, 2020, available at: https://public.ccsds.org/Pubs/660x0b2.pdf.
[12] “Embedded Template Library (ETL) Bit Streams.” [Online]. Available: https://www.etlcpp.com/bit_stream.html
[13] “Python YAMCS Client.” [Online]. Available: https://docs.yamcs.org/python-yamcs-client/
[14] “Robot Framework: automation and testing framework.” [Online]. Available: https://robotframework.org/
[15] Libre Space Foundation, “SatNOGS-COMMS repositories.” [Online]. Available: https://gitlab.com/librespacefoundation/satnogs-comms