Software Design

1. Overview

As introduced in the previous sections, the On-board Computer (OBC) is responsible for governing the entire spacecraft. Since only one microcontroller is used to control the various subsystems and GPIOs, it is crucial to ensure efficient and robust performance of the PocketQube.

The primary purpose of the OBC is to manage the housekeeping of the overall satellite, which includes timely data processing and storage while maintaining low power consumption.

In this context, the Flight Software (FSW) running on the OBC must be carefully designed to meet the timing requirements associated with the diverse tasks the satellite is expected to perform. A common technique used in embedded systems to address these timing constraints is the implementation of Real-Time Operating Systems (RTOS). For the PocketQube, FreeRTOS has been selected as the open-source RTOS to manage the on-board FSW.

For a better understanding of this section, it is recommended to read and comprehend the "A Hands-On Tutorial Guide" provided by FreeRTOS beforehand, especially the sections on Task Management, Memory Management, Task Notifications, Event Groups, Queues, and Software Timers.

2. OBSW Architecture

The OBSW Architercure is designed with the purpose of performing the housekeeping of the overall satellite. In order to meet the time requirements linked to the varied tasks that the satellite is expected to perform, a Real time operating system (RTOS) is implemented. This RTOS allows task scheduling, crucial in our system with only one core, as well as inter-task communication and other functionalities.

Considering the time and human limitations, as the Lektron team is comprised by students, the OBSW makes use of the Hardware Abstraction Layer provided by STM for their microcontrollers. This Hardware Abstraction Layer (HAL) simplifies the control of the Microcontroller Unit (MCU) interfaces greatly, reducing the amount of lower-level coding requiered by the programming user. Do note that to make full use of such libraries parts of it have been modified slightly for them to be FreeRTOS compatible.

On the other hand drivers provided by Semtech are used to control the SX1262 Transciver. With the use of the aforementioned libraries, user functions and FreeRTOS tasks are designed and implemented in order to perform the specific mission requirements and tasks such as power management, telecommand processing and command, on-board data handling, payload data acquisition, attitude and orbit determination and control, between others.

All user code is embedded into FreeRTOS tasks. Such code will make use of HAL and other libraries as well as the basic functions and methods of the C language in order. A simple diagram of this architecture is provided next:

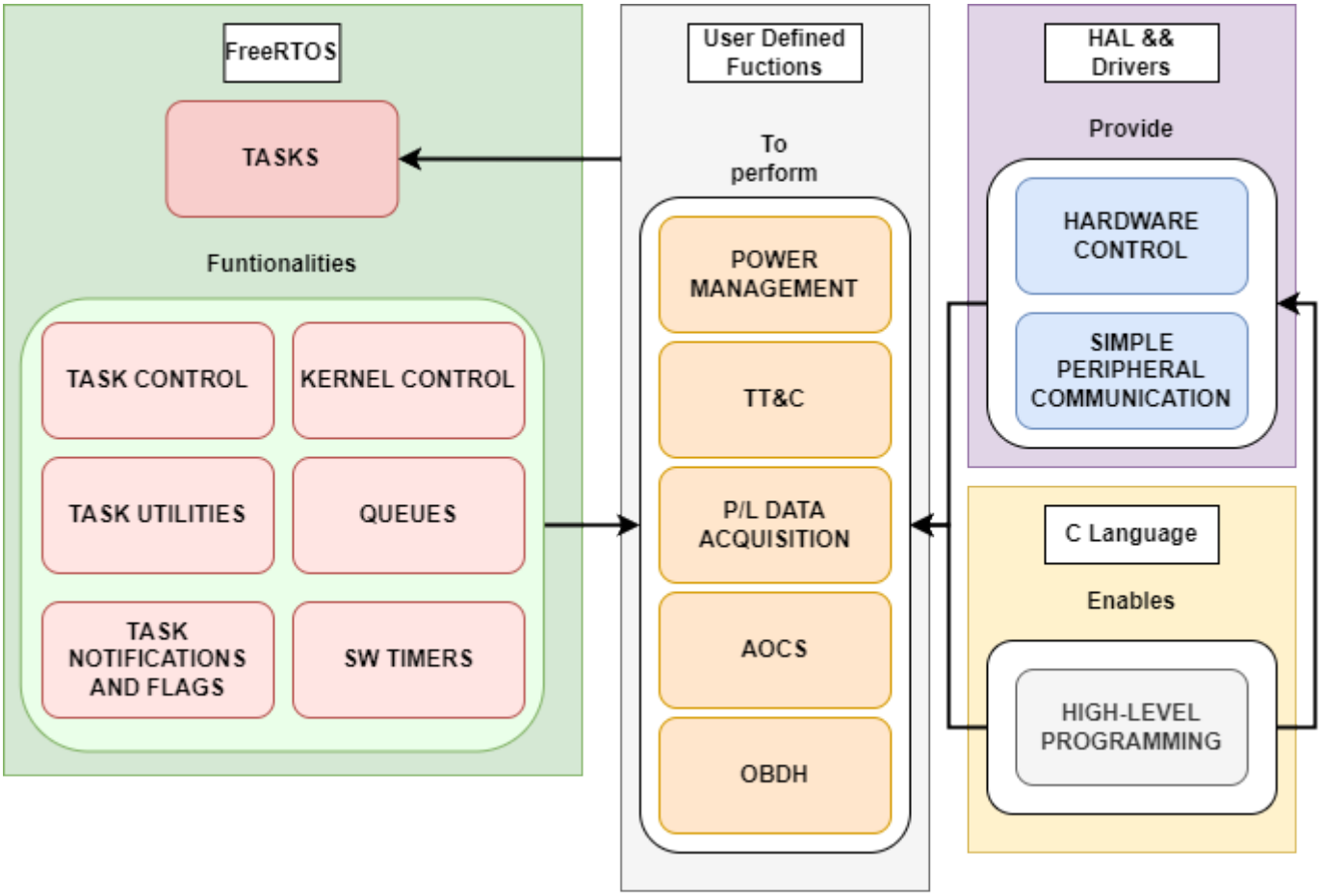


Figure 1: Software Architecture Block Diagram

3. Multitask environment

The FSW operates in a naturally multi-tasking environment, where each task serves a specific purpose and has a designated level of priority. The definition of the various tasks is based on the different sub-modules of the satellite, along with supplementary functions that need to be provided.

In this way, the FSW is structured around several key tasks, including OBC, OBDH, COMMS, AOCS, EPS, PAYLOAD, and FSS. Additionally, there will be a task dedicated to time management, as well

as an existing daemon task.

The **OBC Task** serves as the central scheduler, coordinating the operation of all other tasks. It is responsible for managing transitions between different operational modes, task scheduling, power control, and essential satellite checkups.

The **OBDH Task** oversees the management of internal data within the spacecraft. Its primary focus includes housekeeping data, scientific data, and configurations, as well as managing access to flash memory.

The **COMMS Task** is responsible for managing the transmission and reception of data packets according to the LoRa protocol.

The **AOCS Task** handles satellite orientation and stabilization, along with polling data from the magnetometer, gyroscope, and photodiode sensors. It operates in four primary modes:

1. **IDLE Mode**: This serves as the default state, where the AOCS remains inactive and does not perform any operations.
2. **Safe Mode**: Employed to reset the sensors and algorithms in the event of an issue, ensuring system integrity.
3. **Nadir Pointing**: Essential for conducting payload measurements, allowing the satellite to accurately gather data.
4. **Detumbling Mode**: Activated to stabilize the satellite when it experiences undesired angular velocities.

The **EPS Task** is responsible for two primary functions:

1. Monitoring the voltage, current, capacity, and temperature of the power board.
2. Activating the battery heater if the temperature falls below a specified threshold, ensuring optimal operating conditions for the battery.

The **PAYLOAD Task** manages payload data acquisition.

The **FSS Task** is responsible for implementing the Federated Satellite System protocols.

The **sTIM Task** works alongside the daemon task to implement software timers for error handling purposes. To enhance the robustness of the flight software, timers will be employed to track task execution. Specifically, there will be one software timer for each FreeRTOS task. The software timers feature both active and suspended periods. If a task operation takes longer than its active period, that task will be suspended by the software timer during the suspended period. This ensures that no single task can consume processing time indefinitely, thereby preventing task starvation.

4. Satellite Operational Modes

The satellite can operate in five distinct modes: Init, Nominal, Contingency, Sunsafe, and Survival. These modes reflect the satellite's operational conditions in relation to its battery power levels. Each mode establishes a set of rules that control which actions the satellite can perform, ensuring that power consumption is managed effectively.

Init: This mode is directly associated with the LEOP. The LEOP phase is the most critical, as it begins when the satellite is deployed and concludes when the COMMS antenna is deployed and contact with the ground segment is established. For the PoCat mission, it includes the following steps in this order:

1. **Standby:** To comply with the requirements, after injection in orbit, the spacecraft must wait a minimum of 30 minutes before beginning operations or deploying appendages, specifically the COMMS and payload antenna. At the same time, no radio emissions are permitted after the spacecraft has been integrated into the PocketQube deployer, and this restriction remains in effect for 45 minutes following deployment. Therefore, no beacon transmissions are allowed. This precaution is necessary to prevent interference with other satellites being deployed or with other systems on the rocket. To meet these requirements, the satellite will wait 45 minutes before commencing operations and communications.
2. **Deployment of the LoRa Antenna:** As mentioned, it is necessary to deploy the COMMS antenna to begin transmitting beacons, which will facilitate the first contact with the GS.
3. **First Contact with the Ground Station:** The satellite will remain in this state until it receives confirmation from the GS that everything is functioning correctly, at which point nominal operations can begin.

Nominal: If all the performance criteria and actions of the init phase are successfully completed, the satellite enters nominal mode. This mode serves as the default operating state in the best-case scenario, and the satellite remains in this mode as long as the batteries are above a certain threshold value. While in nominal mode, payload-related operations can be conducted, including the mission experiments and the FSS.

Contingency: The satellite enters this mode when the batteries fall below a certain value. In this mode, some functionalities of the flight software are disabled, including the experiments conducted by the payload and the FSS. Additionally, the AOCS subsystems stop performing nadir pointing to reduce power consumption.

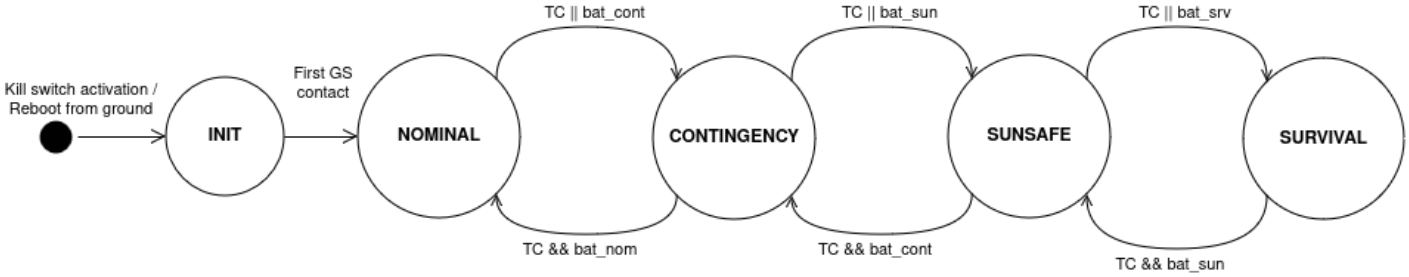
Sunsafe: This mode is associated with a critical condition of the satellite and is activated when the batteries drop below an even lower threshold. More restrictions are applied in this mode to further conserve energy. The EPS and AOCS subsystems cease all activities, such as heating the batteries in the case of the EPS or dumping and detumbling in the case of the AOCS. These subsystems enter a passive state, during which they only collect HK information for telemetry messages.

Survival: This final state is the most critical, occurring when the satellite lacks sufficient energy to perform any vital actions. It may also arise if a significant error occurs in the code, making it safer to remain in this state until operators determine the best course of action. In this state, all subsystems remain active but are only polling information from the sensors. At the same time,

transmissions are ceased, and no beacons are transmitted to prolong battery life; the COMMS subsystem is only in reception mode.

The figure below illustrates the transitions between the states. From the Init state, which is the first state the satellite enters after being released by the deployer or after a reboot, the satellite will transition to the Nominal state following the first contact with the ground segment. Once the satellite exits this state, it can only re-enter if a reboot event occurs.

All other states can transition up or down to the adjacent state. Transitioning to a state with more restrictions can occur automatically if the satellite does not have enough energy to remain in the current state, or it can be triggered by operators through the reception of a telecommand. Conversely, to transition to a less restrictive state, two conditions must be met: the battery level must be appropriate, and the satellite must receive a telecommand requesting the change of state.



To improve performance and reduce power consumption based on the current state of the satellite, high-power tasks are restricted. Additionally, other mechanisms are employed to optimize power usage, such as adjusting the CPU frequency of the microcontroller.

The table shown below, presents the various frequencies at which the OBC operates in each of the different operational modes. It is important to note that higher frequencies provide greater computational power but also result in increased energy consumption. Conversely, lower frequencies reduce power consumption but limit computational speed.

Operational Mode	Frequency
Init	2 MHz
Nominal	80 MHz
Contingency	26 MHz
Sunsafe	8 MHz
Survival	2 MHz

Table 1: Operational Modes and their frequencies

In addition, the actions permitted in the different modes are:

Action	Init	Nominal	Contingency	Sunsafe	Survival

COMMS Tx beacon	Yes	Yes	Yes	Yes	No
EPS heating	Yes	Yes	Yes	Yes	No
AOCS tumbling	Yes	Yes	Yes	No	No
AOCS Nadir Pointing	No	Yes	No	No	No
Payload experiment	No	Yes	No	No	No
FSS services	No	Yes	No	No	No

Table 2: Operational modes permitted actions

5. Scheduling Policy and Task Priorities

The scheduling policy is designed to ensure that tasks are executed in a manner that meets the timing and responsiveness requirements of the OBSW. In FreeRTOS, the kernel employs a preemptive scheduling algorithm, which allows higher-priority tasks to interrupt lower-priority tasks. This approach ensures that critical tasks receive immediate attention, thereby enhancing the system's overall responsiveness.

Task priorities in FreeRTOS are assigned a value ranging from 0 (lowest priority) to a maximum value determined by the configuration settings. Each task is assigned a specific priority level, and the FreeRTOS scheduler uses these priorities to determine the order in which tasks are executed. When a higher-priority task becomes ready to run, it preempts the currently running lower-priority task, allowing the system to respond quickly to time-sensitive operations.

The priority of each task is:

Task	Priority
sTIM	9
Daemon	8
OBDH	7
PAYLOAD	6
FSS	5
AOCS	4

Task	Priority
EPS	3
OBC	2
COMMS	1

Table 3: Task priorities

Timekeeping mechanism

FreeRTOS employs a robust timekeeping mechanism that is essential for managing task scheduling and timing operations within the RTOS. At its core, FreeRTOS uses a tick timer, which generates periodic interrupts at a defined frequency.

The tick timer is responsible for incrementing a global tick count, which serves as the basis for time management in the OBSW. Each tick represents a fixed time interval, allowing the system to keep track of elapsed time and manage task delays, timeouts, and scheduling. Tasks can be delayed for a specified number of ticks, and the system can also implement timeouts for various operations, ensuring that tasks do not block indefinitely.

In addition to the basic tick count, FreeRTOS allows to query the current tick count, convert ticks to milliseconds, and manage task timing effectively. To ensure satellite synchronization with the ground segment, the tick timer is utilized as a clock. However, this timer may experience delays or inaccuracies, particularly in the event of a reboot. To address this, the system can be synchronized again through the reception of the telecommand UPDATE_TIME, which allows for the correction of the tick count and ensures accurate timekeeping.

6. Data extraction and hardware driving

Data extraction from hardware mechanisms differ between the different subsystems. All I2C hardware data polling is blocking, meaning no further execution of the code will be done until the operation is finished or certain time has passed. The same philosophy is applied to DAC/ADC intefaces.

Payload data extraction can make use of Direct Memory Access (DMA), in circular mode, as the amount of information to receive is of the order of kilobytes, yet blocking polling is also possible. Futher software development and testing will determine the final approach. The COMMS subsystem, interfaced through SPI, also uses blocking polls of data.

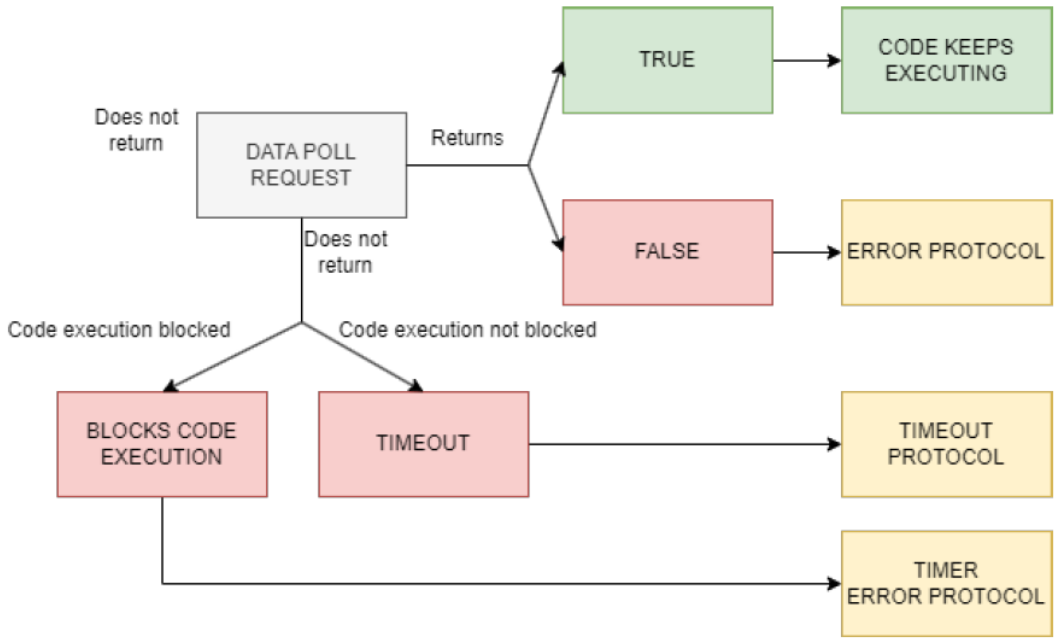


Figure 2: Blocking data polling diagram

Hardware driving is facilitated by the provided STM Hardware Abstraction Layer as well as Semtech provided drivers in case of the COMMS subsystem. Each interface requieres data to be formatted accordingly to the components specifications.

6.1. Data overwriting

The flash memory is divided into 2 kB pages. Once information is written on a page it cannot be overwritten unless the page is previously completely erased. This leads to the use of cyclical overwriting of data as pages are full. For example, if certain information has two pages dedicated to its storage it will full both pages before deleting the first one and start writing there again.

In cases where the information stored is not to fill a full page of itself, the page containing such and other information is to be copied, erased, and then rewritten with the new values.

7. Software Verification Strategy

At every stage of software development different strategies are employed in order to validate the progress done.The first steps in development and validation are characterized by the use of a Nucleo STM L4746RG Development Board. As most subsystems only rely on the MCU and the power line, and this second can be easily emulated, the software corresponding to each of them will first be tried on a system composed by the Nucleo and the hardware to be tested.

In the first iterations of the process a bare bone approach is employed, not implemeting the software into the FreeRTOS environment until the hardware is validated. Once the software is adapted and implemented into the RTOS, still using the development board, the unit to be verified will consist on the prevoius hardware and the software as a single FreeRTOS task. Do note that some subsystems do requiere of values read from sensors in other to be verified. To solve this issue numerical approaches to their values are taken so as to keep testing straighforward.

Once the subsystem to be validated is sure to not be a root of problem in of by itself it will be integrated with the actual OBC-COMMS board for further testing. This process may be done with the help of the EGSE (Electrical Ground Support Equipment) (FlatSAT) or manually connecting both parts, depending on the system. During the next phases of the validation the unit to test expands until it covers the full system, or as much as must be reasonably tested. This includes the integration of the rest of the subsystems both in hardware and in software to the process. The modular approach of the system facilitates this more complete verification. FlatSAT testing is of vital importance during this phase as it provides a suitable platform for system level software tests.

The last steps are to verify the software in the environmental conditions that the satellite will find itself in as much as possible. This process involves the whole system and simulation of said conditions using different facilities and tools.

As a final note, to support communication with the external environment for control and debugging purposes, the satellite is equipped with two communication mechanisms. The first is an umbilical cable, which facilitates the transmission of output logs generated by the satellite and allows for command input via a wired connection. The second communication method utilizes RF (Radio Frequency) through the communications antenna, enabling command transmission in a manner consistent with real operational scenarios.

Revision #4
Created 15 November 2024 18:09:10 by artur.cot
Updated 26 January 2026 10:44:56 by jaume.cortes