POLITECNICO DI TORINO

Department of Mechanical and Aerospace Engineering

Master's Degree Thesis

DEVELOPMENT OF GROUND SUPPORT EQUIPMENT FOR TIME AND COST EFFECTIVE CUBESATS VERIFICATION

Tutor:

Prof. Corpino Sabrina

Candidate:

Deva Luca

Supervisor:

Ing. Stesina Fabrizo

July 2020

TABLE OF CONTENTS:

TABLE OF FIGURES:

TABLE OF TABLES:

ABSTRACT:

This thesis project born from activities carried out during seven months internship at Tyvak International in Turin.
Aim of this project thesis was the development of hardware and software solutions able to support verification and testing of CubeSats platforms during differ stages of their verification process.
Over the last two decades a miniature revolution in space science brought to the development CubeSat concept. Born as educational tools CubeSat's capabilities and advantages related to its "low cost" and "fast delivery" features led to adopt these nano and microsatellite standard also to commercial and scientific applications.
Verification processes play an important role ensuring a balance between these two key-concepts and CubeSat mission reliability, in addition the continuous research of cost and time-effective solutions able to ensure an higher reliability may increase competitiveness of space companies involved in CubeSat development.
In the first chapter of this thesis is presented a general overview on CubeSats highlining advantages, applications, and differences compared with larger satellites.
In the second chapter are introduced approaches and strategies regarding CubeSats verification referring to standard and guidelines proposed by space agencies, in addition are introduced and analysed common tests currently performed in a CubeSat project.
In later chapters are presented the two experimental activities carried out during this thesis project, both these activities are presented following this three steps logic: introduction of the context and associated advantages, description of the design and implementation process and finally analysis of a case study in order to verify the expected operation and identify any future upgrades.
Third chapter describes the development of a tool aimed to support test operator during FlatSat testing allowing a FlatBoard remote control in order to implement a process automatization able to reduce costs and time associated to this process
Finally in fourth chapter is described the development of Sensor Emulator Board (SEB) a dedicated GSE which replacing inoperative flight hardware during FlatSat testing, allows to emulate its behaviour in terms of data transmission and power consumption.

ABBREVIATIONS:

CDS: CubeSat Design Specification

ICD: Interface Control Document

ESA: European Space Agency

NASA: National Aeronautics and Space Administration

CSLI: CubeSat Launch Initiative

COTS: Commercial Off The Shelf

ISS: International Space Station

IOD: In-Orbit Demonstration

TRL: Technology Readiness Level

MarCO: Mars Cube One

DSN: Deep Space Network

CDH: Command and Data Handling

GSE: Ground Support Equipment

SEB: Sensor Emulator Board

HFT: Hardware Functional Test

HPC: Hardware Performance Characterization

DITL: Day In The Life

1 CUBESAT PROJECT:

1.1 CubeSats: Advantages and applications

The term “CubeSat” is used to indicate a standard small satellite developed according to the CubeSat Design Specification (CDS) which includes general design requirements and criteria such as form factor and weight threshold.
Each CubeSat is characterized by a defined number of units, most common configurations adopted are 1U, 3U, 6U and 12U, reported in **Table 1** with their specific size and mass thresholds.
The possibility to have multiple configurations depending on mission objectives, combined with miniaturization of electrical components allows to significantly increase these nano and microsatellite capabilities.

Units	Size (cm)	Mass (kg)
1	10x10x11	1.33
3	10x10x34	4
6	10x22x36	12
12	24x24x36	24

Table 1: CubeSat classification and thresholds

The CubeSat standard does not refer only to detailed form factor and mass, but is related in particular to low-cost and fast-delivery concepts, two key concepts which contribute to giving CubeSats an important role in the framework of the New Space Economy.
The CubeSat concept born in 1999 as result of a collaboration between Jordi Puig-Suari, professor at California Polytechnic State University (Cal Poly) and Bob Twiggs, professor at Stanford University. The purpose of this collaboration was ensuring affordable access to space for universities through the definition of a satellite standard able to reduce development time and satellite project costs. The conformance to a specific standard in facts permits a costs reduction both related to development process and related to launch.
Regarding development cost, the adoption of standardized outer sizes and shape offers the opportunity of integrating in CubeSats platforms COTS “Commercial Off The Shelf” components and subsystems compliant with this standard and developed external suppliers. This approach contributes to enforce CubeSats supply chain and allows CubeSat developers to cut off from budget, costs related to in-house development of these components.
Regarding launch cost reduction, the limited dimensions and mass of CubeSats platforms and their conformance to a specific ICD which defines the interface between the satellite and the launch vehicle, open to new launch opportunities including piggyback payload capabilities and low-cost deployments exploiting the International Space Station (ISS) as logistic link.
In additions, the effort of both private companies and space agencies on developing of small satellites launchers characterized by lower capacity and able to deploy in orbit several micro and nanosatellites, further contribute on launch costs minimization.

Advantages related to CubeSats concept led to a growing interest in space panorama as proved by statistics about CubeSats launches.
In fact according to Nanosats Database **[2]**, as reported in **Figure 1**, since the first CubeSat launch in 2003 the total number of launched CubeSats until the beginning of 2020 has overcame 1200 platforms with 64 countries involved. A value expected to still grow in next years according to the forecasts.

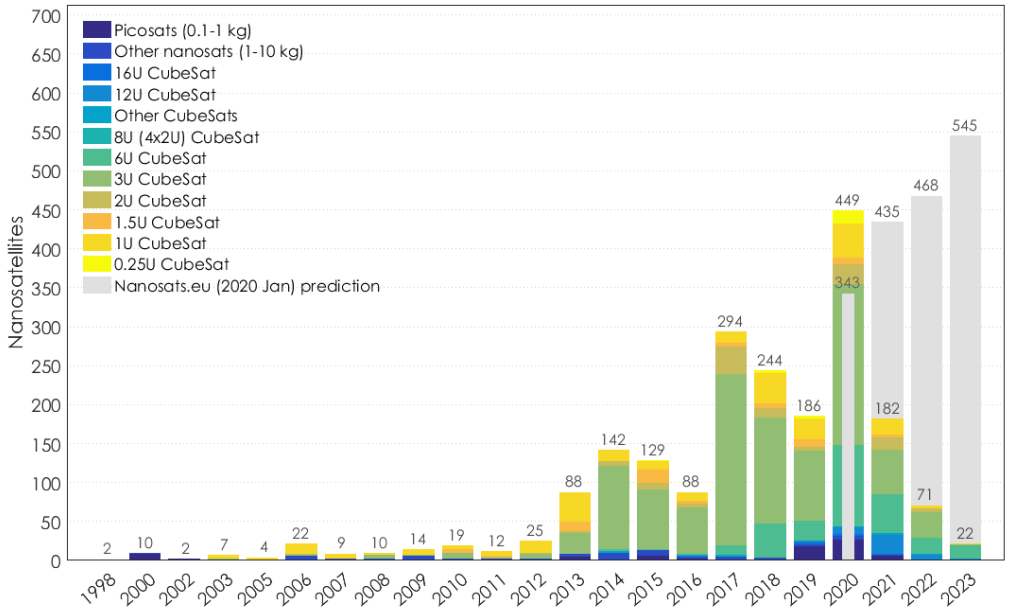


Figure 1: CubeSats and nanosats launches by types

Born as educational tool, nowadays the universities participation to CubeSat program is also promoted by the born of CubeSats launch initiatives developed by space agencies, such as ESA's "Fly your Satellite!" program and NASA's CSLI which offer the opportunity to further reduce costs related to a CubeSat project.
In these programs the agency takes in charge costs associated to launch and environmental tests and in addition offers technical support to teams of students during the development of their CubeSats. It allows to create a partnership which permits to optimize agencies investments, promoting open innovation, and facilitating technology infusion.
However CubeSats do not represent only a valuable resource for educational purpose. CubeSat capability to maximize the functionality per unit mass has resulted a significant reduction of space mission cost, causing a domino effect, which has led to the adoption of this standard also by commercial, civil and scientific entities.
The potential high value in terms of science return and commercial revenue, led to a significant spread of interest between space companies on CubeSat standard, and in addition led several countries to start developing their own space program despite lower budgets and skills level.
CubeSats saw fit for a wide range of applications, starting from In-Orbit Demonstration (IOD) which support the development of flight-qualified technologies through the raise of their Technology Readiness Level (TRL), and allows testing of new mission concepts.
CubeSat missions also play an important role in scientific applications, in particular through remote sensing missions (currently the main sector in which CubeSats are used) which collect

reliable data that can contribute to validation of models or guide decisions related to safety and sustainable economy growth.
Regarding commercial applications, the interest on satellite constellations (e.g. Planet constellation) is constantly growing thanks their capability to provide services, improving coverage and reducing revisit time keeping in account low cost paradigm.
Finally during the last decade, the complexity and potentiality of CubeSat missions have significantly increased opening to new applications not only restricted to low Earth orbit, as demonstrated by 2018's MarCO mission which successfully supported the Insight lander during EDL (entry, descent and landing) operations on Mars surface relaying data back to DSN ground stations.

1.2 CubeSat Reliability: differences

In this section, analysing the balance between mission requirements, risk level and costs of a mission **[3]**, differences between the CubeSats and larger spacecrafts are highlighted, in order to introduce how these differences entail the need to adopt different methodologies in the design phase and in particular during the verification phase.
The balance reported in **Figure 2** is established by programmatic and technological constraints, influenced by design choices and trade off performed can be carried out for each type of mission.

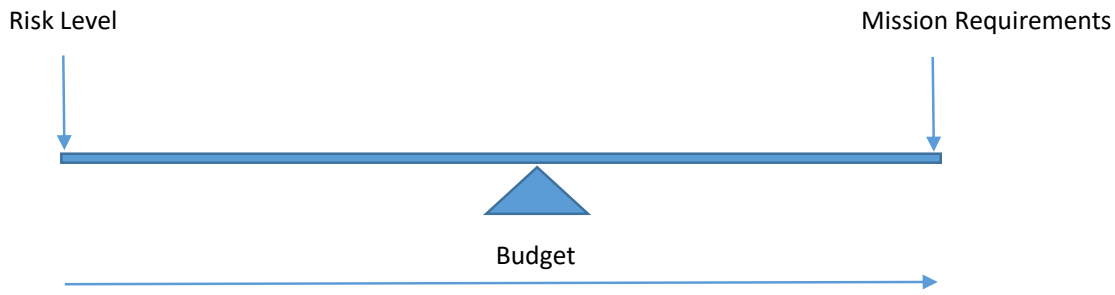


Figure 2: Balance between risks, requirements and cost of a mission

Mission requirements derive from mission objectives and are related to the complexity of a mission, an increasing weight to this parameter can be consequence of an high number of payloads or stringent requirements (high pointing accuracy, high on board system autonomy, ecc…).
Risk level reflects the reliability of the system, intended as the probability of a system to perform a required function under specific conditions for a given time interval (a lower risk level can be translated in an higher probability to have a successful mission).
Each mission will be characterized, depending on its type, by a maximum acceptable risk level. CubeSat low-cost paradigm, which represents the main advantage and most attractive aspect associated with this technology, means also accept increased risks level compared to larger mission.

Large spacecraft in fact, are often unique platforms characterized by very high costs and long development time, a failure in these platforms may imply a large loss of money and the impossibility to replace it with a new platform due to budget limitation.
These features led to a adopt conservative design solutions, redundancies and extensive qualification and performance testing in order to minimize risk of failure.
On the other hand CubeSats, in order to achieve the aimed project cost reduction and fast development capabilities, adopt a lighter and "less conservative" design and testing philosophy (based on COTS products adoption and reduced number of test).
Lower cost and the opportunity to rapidly replace the failed platform makes CubeSats ideal for testing new technologies, innovative hardware, and mission concepts, reducing the impact of possible failure.
Considering the "balance" previously introduced and assuming a fixed budget, it can be noticed how the trend to increase the mission objectives and consequently mission requirements leads implies an increase in the level of risk therefore an higher probability for the system be meet a failure that can compromise the mission.
This trend may lead to exceed mission's maximum risk level, in order to limit it to an acceptable level the typical solution adopted consists on increasing mission budget (for example adding redundancies or extending both development and verification phase).
However this solution seems to be not suitable with CubeSats where low-cost and fast delivery concept have to be limited in order to preserve advantages related to this small satellite category.
CubeSat's balance between Risks, Mission requirements and Budget is illustrated in **Fig.3** and shows how low-cost paradigm can be guaranteed by an higher risk acceptance profile and adopting very focused mission objectives respect to larger satellite.
Focused mission objectives do not represent a limitation in CubeSat philosophy, rather contribute on make these nano and microsatellite ideal on supporting larger spacecraft mission.

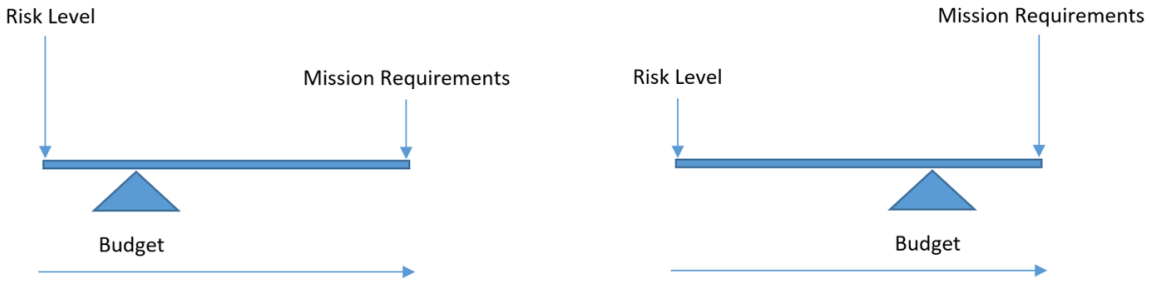


Figure 3: balance in CubeSat missions (left) and balance in large satellite mission (right)

As mentioned the acceptance of an higher level of risk represents an mandatory aspect to preserve CubeSat advantages.
This higher risk level is a consequence of multiple factors: "not conservative" design choices, limited redundancies (often related to physical constrains, which increase the exposure to

single point failure) and low cost and short schedule which bring to adopt lighter verification philosophy and reduced number of test (both analysed in detail in the next chapter).
On the other hand the attributes which characterized a CubeSat project allowing to accept this increased risk level are a shorter operational lifetime foreseen, compared to larger satellite missions, the distributed risk of mission failure in case of mission architecture foresees the adoption of multiple satellite (which implies a reduced cost per-unit) and a limited number of requirements to satisfy related to reduced complexity of the mission compared with other space projects).
Starting from this consideration emerges the importance on the adoption of a tailored verification approach. The role of the verification approach ensures an increased reliability of the system considering a defined risk level.
The verification approach effectiveness is reflected on failure rate of a satellite, in particular analysing the evolution of this parameter in the last decades.
Considering failure rate statistics related to CubeSats mission, according to several statistical report such as "Small-Satellite Mission Failure Rates" developed by NASA Ames Research Center **[3]** and several nanosatellite database, during the last decade CubeSat missions have been characterized by an improved reliability which has led to increase their success rate up to about 80% as shown by **Figure 4**.

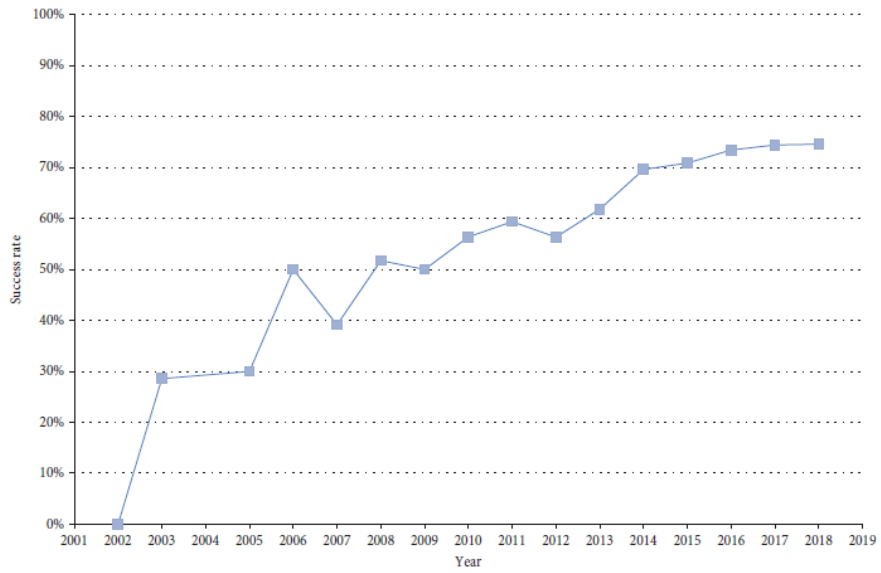


Figure 4: CubeSat success rate evolution according to "Towards the Thousandth CubeSat: A Statistical Overview"

This positive trend can be explained through the gained experience by universities during both design and verification process and in particular is related to the growing interest in the last decade of space companies and space agency on development of this type of nano and microsatellites as proved by Figure 5 showing how CubeSats application have changed since 2013 according to dataset collected by CGEE database [4]

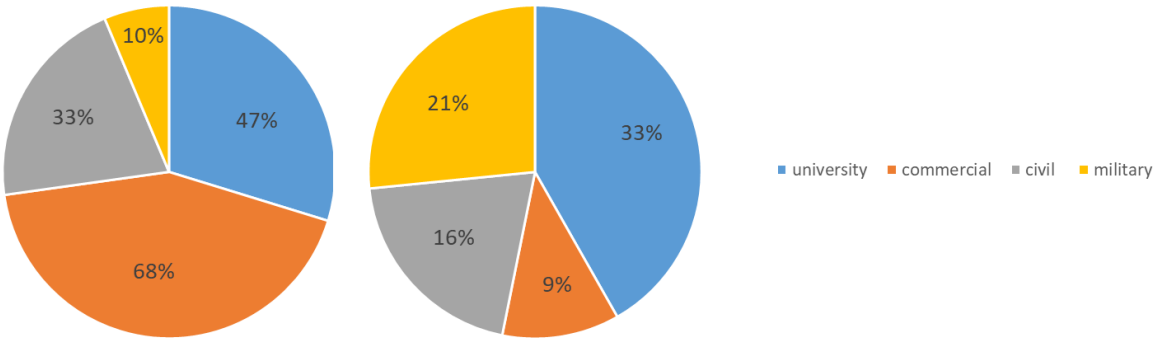


Figure 5: Evolution of CubeSats application from 2013 (right) to 2019 (left)

In addition this growth evidences the importance on searching tailored solutions and verification methodologies for these types of satellites able to increase their reliability level up to larger satellite's level.

2 CUBESATS VERIFICATION PROCESS:

The growing demand of CubeSats expected in next decade according the forecasts, led CubeSat developers such as universities, space companies and space agencies to research smart solutions to verify and test these platforms able to increase reliability without compromising low-cost and fast delivery paradigms related to CubeSat technology.
These solutions contribute to satellite verification process which, depending on type of mission and customer will be regulated by specific standard and regulations.
In the first section of this chapter verification process is analysed referring main applicable standards such as ECSS highlighting how the verification approach changes in the case of CubeSats, while in the second section are described main tests and verification steps adopted during CubeSat verification process in order to offer a clear context before the analysis of solution designed during this thesis project reported in the next chapters.

2.1 CubeSat tailored Verification Standards

The ECSS (European Cooperation for Space Standardization) is an initiative born from the cooperation of the European Space Agency, national space agencies and European industry associations with the purpose of developing standards applicable in all European space activities **[4]**.
ECSS standard are organized in four branches of applicability (Space project management, Space Product Assurance, Space Engineering and Space sustainability) and for each of them divided in disciplines as shown in **Figure 5**.
Reference standards regarding respectively Verification and Testing are ECSS-E-ST-10-02C **[5]** and ECCSS-E-ST-10-03C **[6]** and are collected in Space engineering branch, under System Engineering discipline.

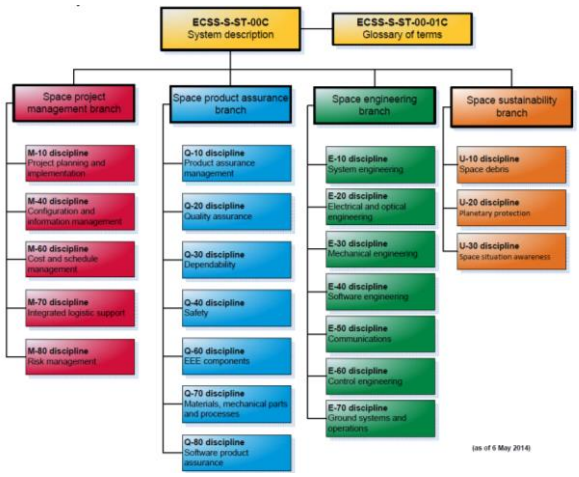


Figure 6: ECSS Overview

ECSS-E-ST-10-02C standard provide the requirements for the verification of a space system product defining fundamental concepts of the verification process, and the guidelines for the implementation of verification programme.

This standard does not concern testing and analysis performed during the product development process not being them formal requirement verification activities in the sense of the customer-supplier relationship.

The verification process consists of three main activities:

- Verification Planning
- Verification Execution end reporting
- Verification Control and closeout

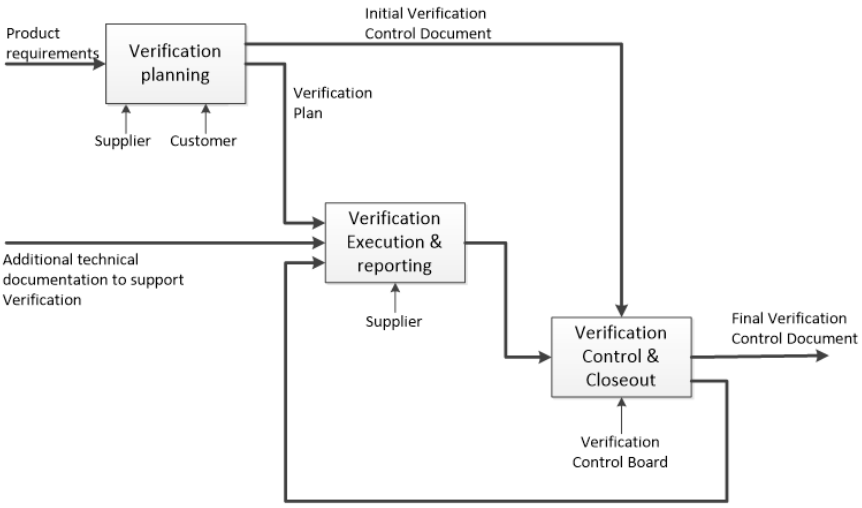


Figure 7: Verification process and activities

These activities lead to delivering of the final VCD (Verification Control Document) where is described the verification control approach and are listed in form of the Verification Matrix the requirements to be verified specifying verification methods adopted in a specific applicable stages at the defined levels (defining verification approach).
To highline verification process differences “Tailored ECSS Engineering Standards for In-Orbit Demonstration CubeSat Project” **[8]** has been considered. This document specifies the applicability of the ECSS Space Engineering Standards to In-Orbit Demonstration CubeSat projects developed through “CubeSat Technology Pre-development” programme specifying for each of the tailored applicable standards applicability of the requirements within these documents.
The differences between a CubeSat and a larger satellite show a lower complexity of the system architecture compared with the latter, according to this asset the verification process required to be adapted to the features of this category of small satellites.

CubeSats platforms as mentioned in previous chapter are characterized by an higher risk acceptance profile, by a low cost, short schedule, these characteristics lead to a adopt a testing approach focused on system level (functionality and environmental qualification/acceptance) and the adoption of different model philosophy.

2.1.1 Verification method

The verification of a specific requirements can be executed by one or more of the following verification methods: test, analysis, review of design and inspection (these methods are listed in order of confidence in the results provided):

- Test method is based on measurement of product performance and functions under representative simulated environments. The verification by test is carried-out on different physical models in agreement with the selected model philosophy.
- Analysis method consists on performing a theoretical or empirical evaluation using a pre-defined technique such as modelling and computer simulation
- Review of Design method adopts approved records or evidence that unambiguously show that the requirement is met
- Inspection method is based on visual determination of physical characteristics

2.1.2 Verification level

Indicates level of decomposition at which the verification is performed. The usual verification levels for a space product are equipment, subsystem, element, segment and overall system

2.1.3 Verification stage

Indicates verification stages along the project life cycle where the verification process is implemented. The verification stages are:

- Qualification
- Acceptance
- Pre-launch
- In-orbit (including commissioning)

2.1.4 Model philosophy

Indicates the definition of the characteristics of physical, virtual, and hybrid models required to achieve confidence in the product verification with a suitable weighting of costs and risks.

Model philosophy is defined by means of an iterative process which combines programmatic constraints, verification strategies and the integration and test programme, taking into account the development status of the candidate design solution.

Model	Description
Engineering Model (EM)	EM is developed to verify the design, it can have same difference compared with FM (even physical and geometrical). Being a development model allows to perform modification
Qualification Model (QM)	A QM is a prototype developed to perform qualification test such as functional, mechanical (both static and dynamic), acoustic, thermal (though TVAC) QM needs to be equal to FM in order to ensure that requirement satisfied on QM is consequently satisfied even on FM, that is why QM is over stressed even beyond expected operative conditions. A QM could serve as a spare part replacement and moreover could be used to troubleshoot if a complex problem occurs. This is especially useful if the problem occurs while the FM CubeSat is not accessible - such as at the launch site, or in orbit. Hardware costs are usually low compared to the overall cost.
Flight Model	Final model of the satellite intended for launch and operational life in orbit.
ProtoFlight Model	The ProtoFlight represents an intermediate model between qualification and fight. Tests performed on ProtoFlight model are lighter than qualification tests.
Virtual Model	Such as CAD or TM (Thermal Model)

Table 2: Model philosophy

Common model philosophy adopted in CubeSat project consists on adoption of an Engineering Model, a FlatSat, and a Flight Model.
Considering this model philosophy qualification testing are performed on the EM, troubleshooting on the FlatSat, and final environmental testing can be performed on the FM.

Frequently with CubeSat development, the additional expense of building a flight-like engineering model is avoided, when this solution is preferred protoflight model is adopted where qualification and acceptance tests shall be simultaneously performed.

2.2 CubeSat verification and test philosophy

Testing represent an expensive and time-consuming step in the lifetime of a space product. Applied at different levels of the satellite and at different stages of the product life cycle, testing allows to verify if the test object performs as expected under foreseen operational conditions and is able to satisfy specific requirements ensuring system survivability to launch vehicle integration, launch, and on-orbit operations.

2.2.1 FlatSat

Realize a FlatSat exactly as the term suggests, means to arrange on a workbench each satellite's component and subsystem interfacing them in order to emulate their operation as a single unit, reproducing an electrical model of the satellite.
CubeSat's sub-assemblies in flight configuration are integrated and stacked into primary structure in order to maximize packing density, this configuration is not ideal during test and verification phase due to reduced accessibility to different satellite components.
FlatSat configuration facilitates testing and troubleshooting of the satellite through exposing sub-units test points and managing power and communication lines through dedicated switches and jumpers.
Connecting the CubeSat subsystems in FlatSat configuration allows: interfaces testing, switch on/off subassemblies simulating operative mode transition, identify errors in data transmission, test the real-time clock circuit operation, test telemetry input-output capability, verify a correct power transmission, test satellite actuators, troubleshooting software and perform several other test dedicated to satellite verification at system level.
During FlatSat testing several GSE (Ground Support Equipment) support test operation, in particular a key role is played by FlatBoard which represents the FlatSat main board designed to support and facilitate verification operations. This board interfacing with the CubeSat processing units and managing power received by the satellite allows the handling and control of the satellite from a single dedicated hardware.
In general GSE indicates hardware and software not intended to fly needed to support testing and verification phase. In addition to Flat-board, other common GSE adopted are power suppliers which replaced CubeSat's solar array power generation (supplying output voltage of MPPT during orbit) and SDR (software defined radio) which emulates ground station allowing FlatSat communication through RF link, antennas and spectrum analyser to test CubeSat RF capabilities.
Flatsat testing offers the opportunity of verifying operation and interactions between CubeSat's subsystems even when they are still not available. If a sub-unit is not available (or result to be inoperative during FlatSat testing) it can be replaced by a dedicated GSE able to

simulate its outputs, this concept brought to the development of SEB (Sensors Emulator Board) analysed in detail in chapter 4.
FlatSat plays a key role not only during on ground verification phase before the launch but also during commissioning phase of a mission allowing to support spacecraft patch development and testing.
FlatSat can be considered as the software and the electrical model of the vehicle, characterized by the same software image of the flight model, it allows to test on ground software image modifications with the aim to evaluate system response on ground model before to update the new software image on the vehicle through a patch implementation.

The need to implement a patch may arises during the commissioning phase when bugs or needed corrections able to imply a mission impairment are be noticed.

In order to create a new software image two possible options exist:

- Modify source files and repeat the building process
- Update a starting image with a patch

In the first option the software image is generated through a "build" process, where starting from multiple configuration files a copy of the entire state of a system is generated stored in a single file. This file will be consequently associated to a specific hardware through the flashing process, however, this procedure could be applicable only when the system where the image has to be flashed is available on ground.
The second option consists on applying a patch on starting image, once uplinked the file containing the patch when a reboot occurs the vehicle will reload the flight image flashed before the launch and then progressively reloads all the patches updated.
This represents the only option the satellite has to update its software image in orbit, nevertheless this sequence is repeated also on ground during FlatSat testing in order to replicate the sequence that will be repeated in orbit in case of successful test.

To better understand role of a FlatSat in patch development and testing, the process which brings to update a software image in the satellite during commissioning phase is described:

1. Same software image is updated both on FlatSat and on vehicle in orbit
2. Telemetry and data received by satellite during commissioning phase led to identify bugs or correction needed which require to evaluate a patch development
3. Identify necessary changes to solve a certain problem through the analysis of telemetry data
4. A new software image is implemented modifying source file and repeating the building process
5. Current software image flashed on FlatSat and vehicle is compared with the new one, through generating the patch
6. Patch is updated on FlatSat
7. Testing of the new image on the FlatSat in order to evaluate resolution of bugs identified at the beginning of the process
8. Uplink of the patch on vehicle

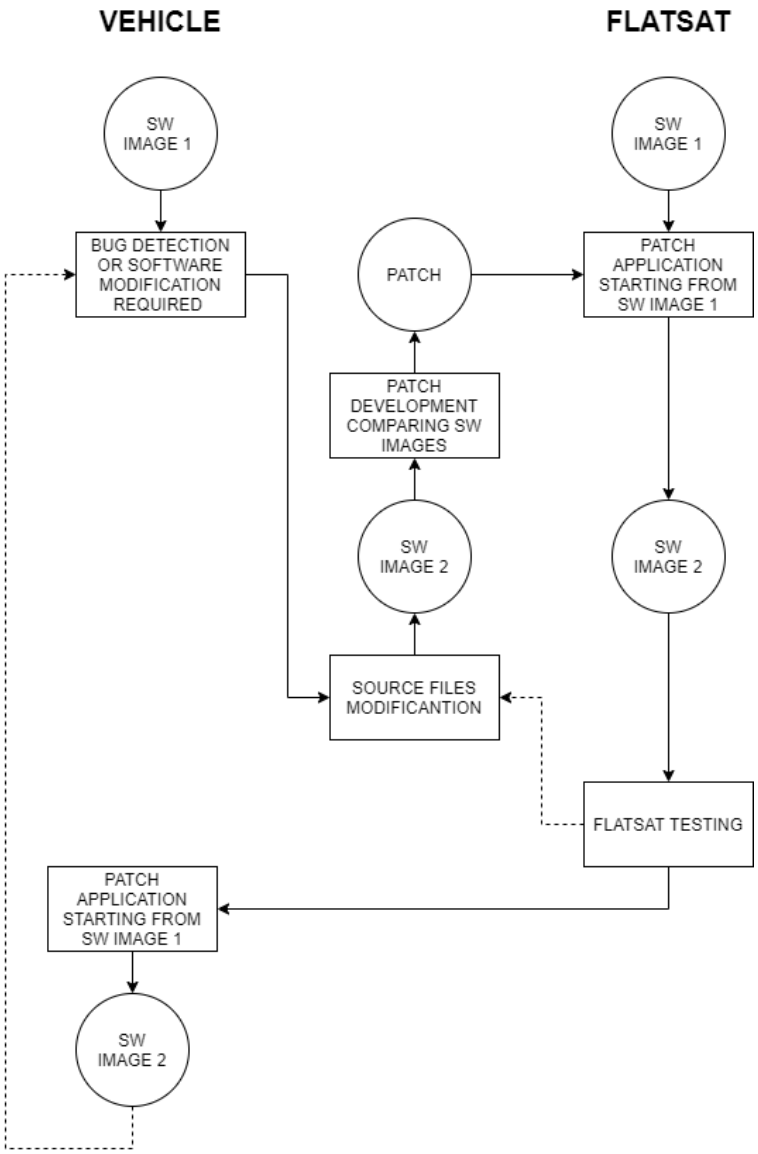


Figure 8: Patch release process

Once this process is completed and verified if the onboard software changes introduced by the patch uplink have corrected the bugs identified by the analysis of the telemetry data, the operation phase continues nominally evaluating the need for possible new patches.
FlatSat testing performed during operations consists on the verification of the new software image generated by the patch upload.
To evaluate the successful development of the patch before its release and its application to the flight vehicle in orbit, FlatSat testing which consists on the verification of the new software image generated by the patch upload and is performed by analysing the telemetry generated by the FlatSat. In particular, it is analysed concordance of the response of the satellite equipped with a new software image with the expected changes or the resolution of the identified bugs.

A possible situation which requires the development and application of a patch, is related to modification of dedicated configuration file where reference value a and threshold are read and compared by spacecraft processing unit.
In this case during patch testing with FlatSat, the opportunity to reproduce same conditions detected by the satellite through the emulation of sensors reading would be useful to verify expected responses of the new software image. The solution developed and detailed in chapter 4 aim to support FlatSat testing in these situations.
During commissioning FlatSat is not used only for patch testing and development but also for telecommands test and validation. This activity consists in testing a particular "off-nominal" command by sending it to the FlatSat, verifying the conformance of the FlatSat response with the expected results, and if confirmed uplink the command to the vehicle in orbit.
According to what described in this section, the FlatSat play a key role both in the ground verification phase and during the commissioning phase, facilitating its verification activities.
In order to support commissioning phase returning reliable data, FlatSat has to ensure the accomplishment of a key requirement, the software images of the FlatSat and satellite must be identical before. To ensure equality between the software images "Versioning" is performed reviewing and managing software changes, in addition each time the FlatSat is used for a test, "test preparation" is carried out, during this step the flight software image is re-flashed and subsequently all patches are progressively re-applied to the vehicle.

2.2.2 HFT (Hardware Functional Test)

HFT (Hardware Functional Test) purpose is to test spacecraft electronics functionality identifying possible issues related to software development, electronics workmanship or modules integration.
HFTs are performed during multiple steps of the verification process starting from acceptance of components received by external supplier, up to environmental and pre-launch tests.
Being these tests performed in a distributed way during the entire integration and verification process, they require the need to segment this test into subsections which will be progressively executed according to the level of integration of the satellite under test.
HFTs can be performed when the satellite is multiple build configurations: FlatSat configuration, when it is assembled in a flight configuration or even if only some specific modules of the satellite are available (for example CubeSat CDH).
Carrying out a large number of HFTs since the beginning of the satellite assembly phase allows tracking the status of the satellite by identifying the various bugs emerged permitting to identify (adopting a pass / fail logic) when and how an error is resolved or when an error emerges at a certain stage of the integration and testing process.

2.2.3 HPC (Hardware Performance Characterization)

HPC (Hardware Performance Characterization) purpose is to test performance of spacecraft module and subsystems completing functional verification started with HFTs.

HPC should be performed prior to vehicle build and before performing any permanent operation (such as stacking, mounting solar cells, permanently attach mechanical) or after critical steps of satellite AIT process, these steps are for example dry build, after thermal tests, wet build and after ENVT (environmental tests).
The difference between these types of tests and hardware functional tests related to capabilities of HPC to evaluate the performance associated to each module or critical component response. This approach allows you to verify if the system is capable of meeting the requirements foreseen.
The process adopted carrying out HPC is longer than HFT'process, and is carried out on the satellite when it is in the configuration after the dry build. Furthermore, the need to carry out the HPCs separately on the various modules of the satellite prevents the automation of the verification process as done with functional tests.

2.2.4 Thermal Design Verification test

Thermal Design Verification test purposes are:

- verify that spacecraft is free from possible issues introduced during assembly and integration phase which may compromise mission
- verify spacecraft capability to perform key steps to complete the mission in the temperature ranges foreseen such as the deployment of the antennas and solar array
- functionally verify processing units, sensors and actuators in foreseen temperature ranges

This step is performed once spacecraft has completed dry and wet build and all systems are integrated inside the spacecraft in their flight configuration.

2.2.5 Day In the Life Test

A key role between system level verification test is covered by Day in the Life (DITL) test.
Performed after environmental test campaign, DITL represent the last verification step before to proceed with satellite transportation on launch site.
DILT involves simulating a day of activities for the satellite and verifying subsystems operations and conformances with expected behaviour. Different scenarios such as nominal operation or critical emergency will all be tested during DILT.
This series of tests in addition to space segment testing:

- offer MOPs the ability to gain understanding of the satellite processes and critical aspects helping with failures detection, and recovery during the commissioning phase
- allow to test mission control software, and operator ground interface

During DITL testing CubeSat flight model is interfaced only through a power supply software-controlled which depending on condition simulated by orbit propagator provide a power supply to the system by-passing solar arrays and MPPT. No cabled data transmission is needed during this test but communication between ground and space segment is completed performed via RF.
During DITL on-board software performs the same steps performed during LEOP phase, once released deployment switches starts executing mode transition and executes command scheduling and appendices deployment.

3 FLATSAT REMOTE COTROL

3.1 Context and introduction

First activity developed during this thesis project, regards the development of a tool which allows to remote control FlatBoard during FlatSat testing.
As previously analysed in section 2.2 FlatSat testing represents an indispensable step in CubeSat's verification process, capabilities and advantages related on performing these tests in this configuration are mainly related to FlatBoard properties. This board interfacing with satellite's sub-assemblies allows to communicate with satellite through common serial interfaces, offers a breakout points to support electrical signals troubleshooting and allows operators to enable/disable satellite channels through the management of dedicated switches.
The purpose of this activity was to implement a solution to manage and control these switches remotely avoiding a manual control by the operator during the FlatSat testing.
The opportunity to remotely control the satellite operations by enabling or disabling these switches offers advantages related in particular to HFT executions.
This activity in particular represents the first step needed to implement process automatization of hardware functional testing activities performed adopting FlatSat configuration.
The current HFT procedure foresees the execution of a dedicated script, which running on a processing unit interfaced to the spacecraft, allows the functional verification of satellite components and sub-assemblies. During the execution of this script, the correct transmission of data and the identification of software issues are checked by interrogating sensors (and comparing the reading with a range of pre-set values) or by sending commands to spacecraft actuators and check their expected response.
This checking process is limited by the need of an operator in charge of manage FlatBoard's switches in order to enable or disable spacecraft communication channels, allowing a complete testing of the system under verification.
The opportunity to remotely control these switches via software adding dedicated instructions to the current adopted script would allow an automatization of the HFT procedure avoiding the need of a manual control by the operator.

HFT process automatization and FlatSat remote control leads to a resource optimization able to ensure:

- increased efficiency of the test process
- cost reduction

The increased efficiency of HFT can be translated in terms of time reduction adopted to perform a single HFT. HFT process automatization allows to speed up the execution of the test and to increase the number of HFT that can be performed, ensuring an higher level of reliability of the spacecraft under test.

A high number of HFT allows to drastically reduce the risk of infant mortality.
Analysing statistical estimated failure rate of a product in relation to its expected lifetime, can be referred to the graphic representation called the "Bathtube curve" shown in **Figure 7**.

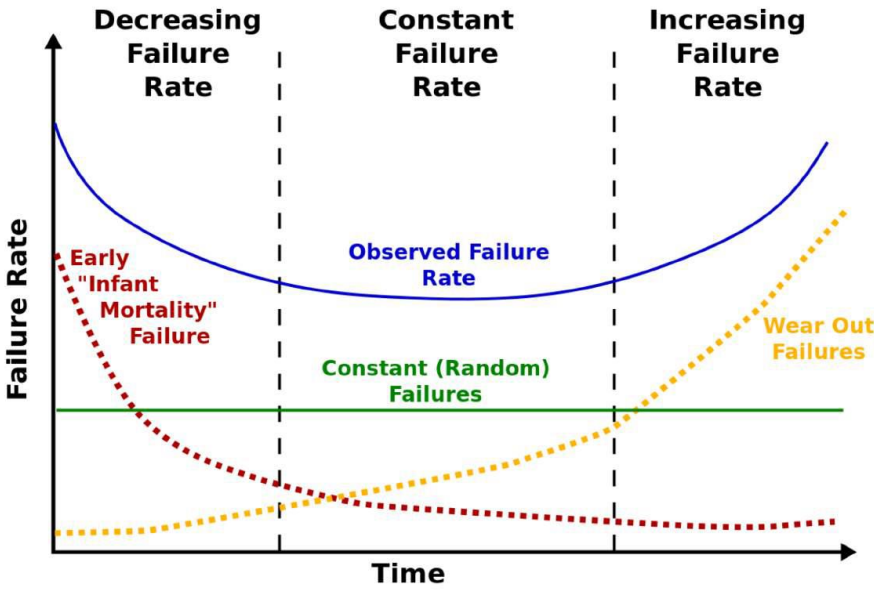


Figure 9: Bathtube curve

According to this graph three types of failure may occur during lifetime of a product:

- Early "infant mortality" failure → mainly related to manufacturing errors not detected during first stage controls
- Random failure → related for example to unexpected operating conditions
- Wear out failure → related to wear or the exceeding of the certified life of the component considered during design phase

Combining rates associate to these types of failures, "observed failure rate" curve is obtained. In this curve three distinct regions can be identified: a first period mainly influenced by infant mortality failures and characterized by a decreasing failure rate, a second period also named "useful life" with a low, relatively constant failure rate and concluding with a period that exhibits an increasing failure rate mainly due to wear out failures.
Failures during infant mortality are unacceptable under a customer satisfaction viewpoint, in particular between CubeSats where often low-cost design philosophy leads to avoid conservative choices such as components redundancies, and meet an failure in the early stages of operative life can compromise mission accomplishment.
This type of failure is caused by design blunders, loss of components unable to survive to operative condition or defects arisen during assembly and integration process.

Test philosophy in space sector is focused on ensuring through dedicated tests (in particular the list of tests described in section 2.2) that any of this condition do not occur during spacecraft expected lifetime.
However even a correct design can fail to cover all possible interactions of components in operation. Failures can be a consequence of a component's manufacturing phase, they can emerge during the assembly and integration with other spacecraft sub-assemblies phase, or they can be induced during pre-launch tests (such as environmental tests) performed to stress the system under test in order to evaluate design weaknesses.
This underscores the importance of performing frequent tests to functionally evaluate the system. HFTs cover exactly this role. Performed since the first stages in which the test object is available (during the acceptance process), performed before and after the integration between the various sub-assemblies and finally during thermal and environmental tests when the spacecraft is integrated with the satellite in its configuration final, allow to evaluate and track the issues emerged in order to resolve them before being released into orbit.
Considering again bathtube graph previously illustrated, automatization and incrementation of HFT allows, by emulating the instruction routine performed during the operational phase, to reduce the unexpected failure rate once the cubesat is released in orbit, so as to increase its reliability.
Second relevant advantages related to this solution is related to time saving and cost reduction aspects. These advantages are consequence of resources optimization induced by process automation and in particular by the possibility of controlling the satellite remotely.
Time-saving and cost reduction aspects are closely linked, economic advantages are related in particular management costs, associated to perform multiple tests in parallel (common situation in particular during acceptance test of modules and components manufactured by external supplier). In addition the opportunity to remotely control the satellite by enabling or disabling these switches is particularly advantageous during environmental campaigns allowing to reduce rental costs of external facilities.

3.2 Solution

The procedure followed to enable remote control switches involved 3 steps:

1. Hardware Procurement
2. Hardware Configuration
3. Python script implementation

Starting from this point in this chapter the FlatBoard will be referred to as EDUB.

3.2.1 Hardware Procurement

Hardware procurement consisted on the purchase of Raspberry Computer Module 3 **[X]** and D-Link wifi USB adapter DWA-121 **[X]**.

While Raspberry CM3 has been chosen taking into account existing interface with Tyvak developed FlatBoard, wifi USB adapter has been chosen after a trade-off considering performance, size and compatibility between COTS available devices.

Figure 10: Raspberry Computer Module 3

The purchase of these components is aimed at enabling remote communication between the PC used by the operator to manage the FlatSat, also the addition of a processing unit allows to command EDUB operation through the execution of dedicated scripts and the interface with EDUB sensors and devices.

3.2.2 Hardware Configuration

Enabling "switches remote override" through a dedicated switch on the FlatBoard will be possible modify the state of the other available switches commanding via I2C a dedicated gpio expander. Each output gpio of the gpio expander is associate to single switch as reported in **Figure 10**.

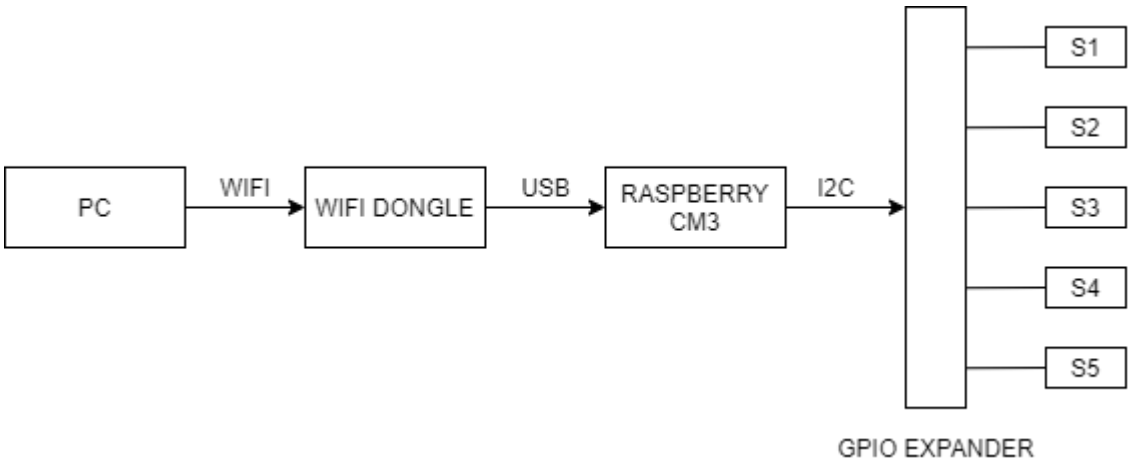


Figure 11: Remote switches override architecture

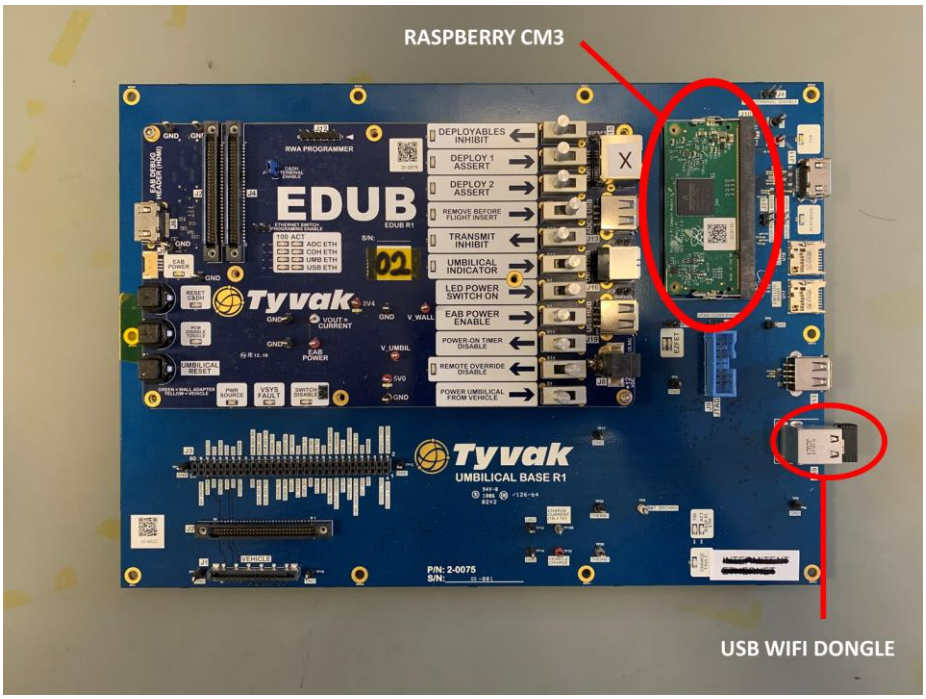


Figure 12: EDUB FlatBoard layout

In order to communicate with gpio expander via I2C a software re-configuration of Raspberry CM3 is needed to enable an i2c bus disable by default on bus 0.
To accomplish this re-configuration an analysis of CM3 setup process has been carried out. Compute Module 3 (CM3) contain the Raspberry Pi BCM2837 chipset which is provided has three banks of General-Purpose Input/Output (GPIO) pins (28 pins on Bank 0, 18 pins on Bank 1, and 8 pins on Bank 2), these pins can be used as true GPIO pins which can be set to 'alternate functions' such as I2C, SPI, I2S, UART.

The BCM283x devices as used on Raspberry Pi and Compute Module boards have a three-stage boot process:

1. GPU (graphics processing unit) core, comes out of reset and executes a code from a small internal boot ROM) which loads a second stage boot-loader called bootcode.bin from eMMC
2. bootcode.bin is responsible for executing the main GPU firmware called start.elf
3. start.elf first reads a binary file called dt-blob.bin to determine initial GPIO pin states, then parses config.txt and finally loads the ARM device tree file bcm2708-rpi-cm.dtb and any device tree overlays specified in config.txt before starting the ARM subsystem and passing the device tree data to the booting Linux kernel.

Changing pin configuration can performed modifying files read and loaded by firmware in this last step of the boot process.
Device Tree is a dedicated way of encoding all the information about the hardware attached to a system (and consequently required drivers) through binary files (.dtb extension) which are compiled from human readable text (.dts extension files) by the Device Tree compiler.

On a standard Raspbian image two different types of device tree files can be founded:

- dt-blob.bin → compiled file read by GPU firmware to set up functions, reported as attachment in Annex 1.
- bcm2710-rpi-cm3.dtb → ARM Linux device tree file, which GPIOs are used, what functions those GPIOs have, and what physical devices are connected

During boot the user can specify the ARM device tree to use via the device_tree parameter in config.txt, In addition to loading an ARM dtb, start.elf supports loading additional Device Tree 'overlays' via the dtoverlay parameter in config.txt.
Overlays are used to add data to the base dtb that (nominally) describes non board-specific hardware, it includes GPIO pins used and their function, as well as the devices attached, so that the correct drivers can be loaded.
Overlays are merged with the base dtb file before the data is passed to the Linux kernel when it starts, an represent represent the easiest way to assign i2c function to gpio pins (i.e. GPIO0=SDA0, GPIO1=SCL0).
Following are reported dedicated instruction added to CM3 config.txt supported by GPIO alternative function. Only gpio of interest whose configuration represents an indispensable step for enabling the remote control of the FlatSat switches are shown in Figure X, addition alternate GPIO function are reported in Raspberry Pi Compute Module 3 (CM3) datasheet [8].

GPIO	Default Pull	ALT0	ALT1	ALT2
0	High	SDA0	SA5	PCLK
1	High	SCL0	SA4	DE
2	High	SDA1	SA3	LCD_VSYNC
3	High	SCL1	SA2	LCD_HSYNC

Figure 13: CM3 GPIO alternate functions for I2C enabling

3.2.3 Python script implementation

In order to enable FlatSat remote control has been developed a simple Python module which running on Raspberry CM3 allows to easily manage FlatBoard switches.
In **Figure X** is reported the architecture of Python module implemented:

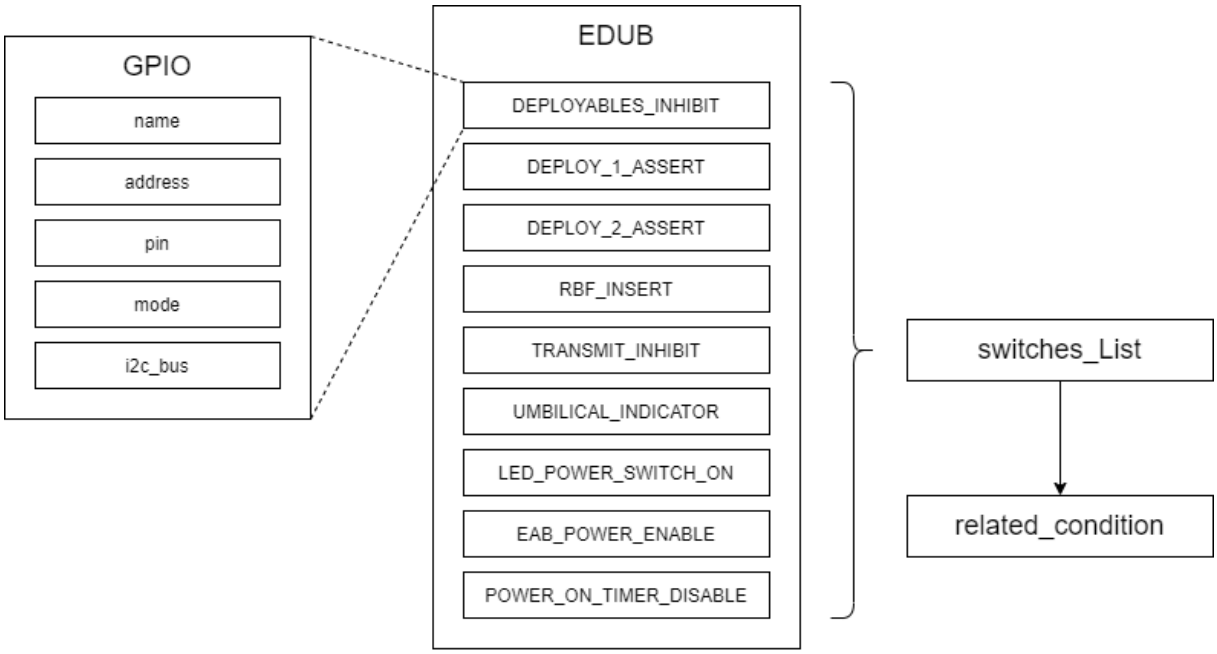


Figure 14: EDUB_switches_manager.py structure

Analysing the architecture of the code a main class named EDUB is defined, the object belonging to this class will be characterized by nine instances, one for each controllable switch on the FlatBoard.
Each of these instances is represented by an object associated with the GPIO class, this class is defined in a lower level module implemented considering the structure of the registers characteristic of the TCA9539 gpio expander controlled via i2c by the raspberry CM3.
According to gpio expander datasheet [X] each switch object associable with a GPIO element is characterized by following attributes:

- Name → related to switches label printed on FlatBoard
- Address → 0x74, related to gpio expander slave address on the i2c bus
- Pin → related to output pin of gpio expander
- Mode → whose state will be input or output depending on setting
- Bus → related to CM3 i2c bus where gpio expander is located

These attributes are managed to lower level, operator on controlling these switches shall only set a condition (True of False), as consequence internal registers of the GPIO expander will be modified enabling or disabling the switch.

Depending on the switch high logic level can be associate to True or False condition, in order to assign this correspondence a dedicate list named related_condition as been implemented. Python module not only permits to remote enable or disable spacecraft channels usually managed manually by test operator, but it also allows the operator to check switches status when controlled hardware is not visible. Enabled "remote override" switch the mechanical correspondence with the switch is lost by-passing the component and associating the state of the switch with the value of the relative gpio.
Consequently Python script implemented running on Raspberry CM3 covers two main functions:

- Read switches status
- Command single switches to a new status

The first function is implemented by reading the current value of each switch, comparing it with the relative condition contained in the homonymous vector (where they are conditions associated to high logic level are reported) and printing the condition for each switch.
In Figure X and Figure is reported in code snippets the section of the EDUB_switches_manager implementing this check status function.

```
# CHECK SWITCHES STATUS

def switches_status(self):
    self.switches_conditions = []
    x = 0
    if self.switchesList[x] is not None:
        while x<len(self.switchesList):
            state = self.switchesList[x].value()
            # value is a method of GPIO class
            # translation of GPIO status in condition saved in switches_conditions[x]
            if state == "1":
                cond = self.related_conditions[x]
            elif state == "0":
                cond = not self.related_conditions[x]
            else:
                logging.warning("GPIO READING PROBLEM")
                break

            self.switches_conditions.append(cond)
            print("{} = {}" .format(self.switchesList[x].name,self.switches_conditions[x]))
            x += 1

        logging.info("SWITCHES STATUS HAVE BEEN PRINTED")
```

Figure 15: switches status check module

An example of the expected output related to the execution of this "switches_status" method is shown in **Figure 16**.

```
PS C:\Users\Luca\EDUB_switches_control\devices> & C:/Users/Luca/AppData/Local/Programs/Python/Python37/python.exe c:/Users/Luca/EDUB_switches_control/devices/EDUB
_switches_manager.py
DEPLOYABLES_INHIBIT = True
DEPLOY_1_ASSERT = True
DEPLOY_2_ASSERT = True
REMOVE_BEFORE_FLIGHT_INSERT = True
TRANSMIT_INHIBIT = True
UMBILICAL_INDICATOR = False
LED_POWER_SWITCH_ON = False
EAB_POWER_ENABLE = False
POWER_ON_TIMER_DISABLE = True
INFO:root:SWITCHES STATUS HAVE BEEN PRINTED
```

Figure 16: switches_status output

The second function of EDUB_switches_manager.py script consist on set a boolean condition to change status of a specific switch. In Figure X and Figure X is reported code snippets of the section of the EDUB_switches_manager.py script implementing this control switch function.

```
def deployables_inhibit(self, condition):
    if condition is True:
        if self.switches_conditions[1] is not True:
            # change mode in OUTPUT:
            self.DEPLOYABLES_INHIBIT.set_mode(GPIO.OUTPUT)
            # change GPIO value
            self.DEPLOYABLES_INHIBIT.set_value(1)
            # change condition
            self.switches_conditions[1] = condition
            # re-change mode in INPUT:
            self.DEPLOYABLES_INHIBIT.set_mode(GPIO.INPUT)
            logging.info("DEPLOYABLES HAVE BEEN INHIBITED")
        else:
            logging.warning("CONDITION ALREADY SETTED")

    elif condition is False:
        if self.switches_conditions[1] is not False:
            # change mode in OUTPUT:
            self.DEPLOYABLES_INHIBIT.set_mode(GPIO.OUTPUT)
            # change GPIO value
            self.DEPLOYABLES_INHIBIT.set_value(0)
            # change condition
            self.switches_conditions[1] = condition
            # re-change mode in INPUT:
            self.DEPLOYABLES_INHIBIT.set_mode(GPIO.INPUT)
            logging.info("DEPLOYABLES HAVE BEEN ENABLED")
        else:
            logging.warning("CONDITION ALREADY SETTED")
    else:
        logging.warning("INVALID CONDITION INPUT")
```

Figure 17: switch control module

During coding phase of this activity Git has been used, it represents a version control system for tracking changes in several type of files like Python scripts. Being a distributed version control system, Git allows to multiple developers to work on a single project or code storing different changes made and permitting to revert back to any specific version of a file as long it was committed to the git repository. This repository is saved in local but in any time of the project development it can be updated and “pushed” to a remote repository.

3.3 Test Case

In order to verify the capabilities of the solution presented in the previous sections, a test case has been adopted, according to company priorities test case considered is CubeSat's CDH cold boot test.
This test is part of the components acceptance process which are in-house designed but manufactured by external suppliers.
For each satellite CDH (Command and Data Handling) is a key subsystem which require to withstand to critical operative conditions during the entire mission and in each CubeSat operative mode. Before integrating this critical component into the satellite during its assembly process a series of dedicated tests needs to be performed in order to ensure CDH functionality and capability to satisfy design requirements, cold boot test is one of them.
Purpose of this test is determination of minimum temperature above -20 degC where CDH boot without showing any failures. Depending number of successful boots and minimum cold boot temperature determined, tested CDH will be assigned to flight purposes or to ground applications like integration in FlatSat.
To test boot capabilities of the CDH it is brought to an ambient temperature of -20 °C and through power cycling the power line is alternately enabled and disabled. During each power cycle the CDH software image tries to boot, in case of successful attempt, a dedicated counter is incremented and the temperature from the sensors in the CDH is read through a suitable process, finally the CDH is halted and after a in following a hard reboot the sequence is repeated.
FlatBoard cover a key role in this test, interfacing CDH, allows operator to power on or off CDH through "EAB POWER ENABLE" switch. This control traditionally has been performed only manually asserting or de-asserting the switch, however the need to control this switch periodically after each boot results to be repetitive and time consuming activity which can be automated if CDH power enable switch is controllable via software.
According to the needed resource optimization this test perfectly suits as test case to verify FlatBoard re-configuration and a python script capabilities.
Resuming, the objectives of this test case can be dived in cold boot test and remote switch control objectives:

Cold boot test	Remote control switches test
• CDH thermal requirements verification • Assign component to flight/ground applications • Test processor capabilities to read sensors during critical operative conditions	• Test remote access to satellite's CDH • Test "EDUB_switches_manager.py" script controlling "EAB POWER ENABLE" switch • Verify test time reduction compared with traditional test procedure

Table 3: Test case objectives

Analysing test case setup:
Test environmental conditions are reproduced through a thermal chamber controlled by a Raspberry 3b+ which allows to remotely set a target temperature and the relative slope defining test thermal profile. Raspberry pi 3b+ is also adopted to provide a feedback of the temperature interfacing with a control thermistor through a dedicated ADC.
CDH's temperature during the test is instead tracked through a thermocouple placed on test object's critical component a interfaced PicoTech DAC TC-0B in order to software record temperature variations associated to CDH power cycling.
Finally, FlatBoard equipped with wifi USB adapter and Computer Module 3 running switch manager script is interfaced to CDH via power line and a debug serial connection and via wifi to test operator pc. While CDH under test is placed inside the thermal chamber during the test, Ground Support Equipment and processing units introduced are located outside the chamber as reported in configuration illustrated in **Figure X** and **Figure Y**.

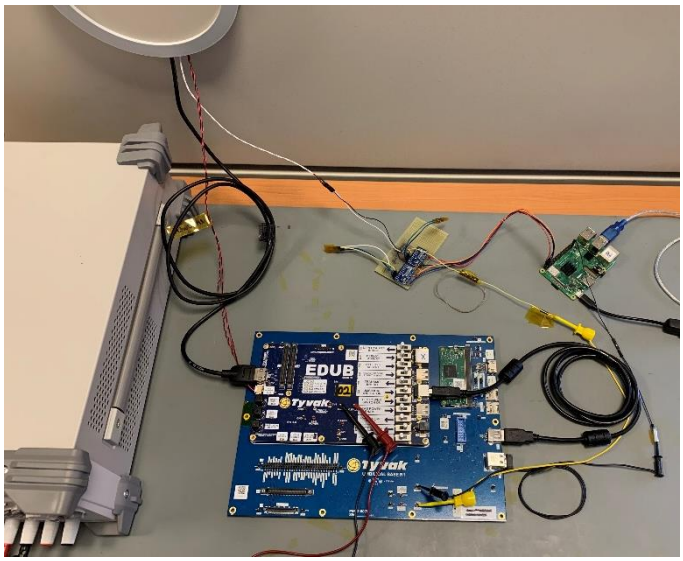


Figure 18: set-up outside the thermal chamber

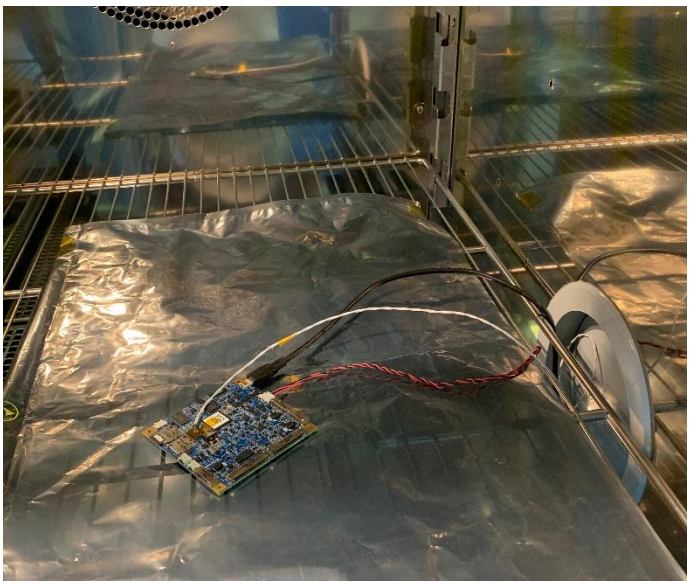


Figure 19: set-up inside thermal chamber

Analysing test procedure following steps have been executed:

- Three different SSH sessions are established between operator's pc and Raspberry CM3
- In the first Terminal serial communication is established in order to access to CubeSat CDH located inside thermal chamber
- CDH's reboot counter is initialized
- Thermocouple temperature recording is started
- In the second Terminal thermal chamber temperature is set to -20°C executing dedicated python script
- Once reached target temperature, using third Terminal "cdh_cold_boot_test.py" is executed. this script running on raspberry CM3 will execute in loop following steps:
    1. Sets "true" condition to "EAB_POWER_ENABLE" switch supplying power to CDH
    2. Access to CDH processor
    3. Reads and prints temperature from CDH internal sensors
    4. Halt CDH processor
    5. Sets "False" condition to "EAB_POWER_ENABLE" switch turning off CDH
- If CDH doesn't successfully boots, test operator, receiving feedback from CDH messages printed in the first terminal, will increase temperature by +2°C and verify again if CDH boot success
- If CDH successfully boots, test will proceed in a fully automated way up to stop command by the operator
- Once completed the test after at least ten consecutive boots, "cdh_cold_boot_test.py" execution will be arrested and both temperature recording and reboot counter are analysed

The results obtained from the execution of the cold boot test are then analysed, in a first case by adopting the traditional procedure in which the operator forces the power cycling of the CDH under test, and in a second case in which through the execution of a dedicated script the process is automated. This comparison is performed in order to evaluate the effective time reduction of the test.
Considering traditional procedure, in **Figure 19** is reported thermocouples acquisition track. Two curves are relative to two different CDH tested during same test process adopting two separate set-up as the one previously described.

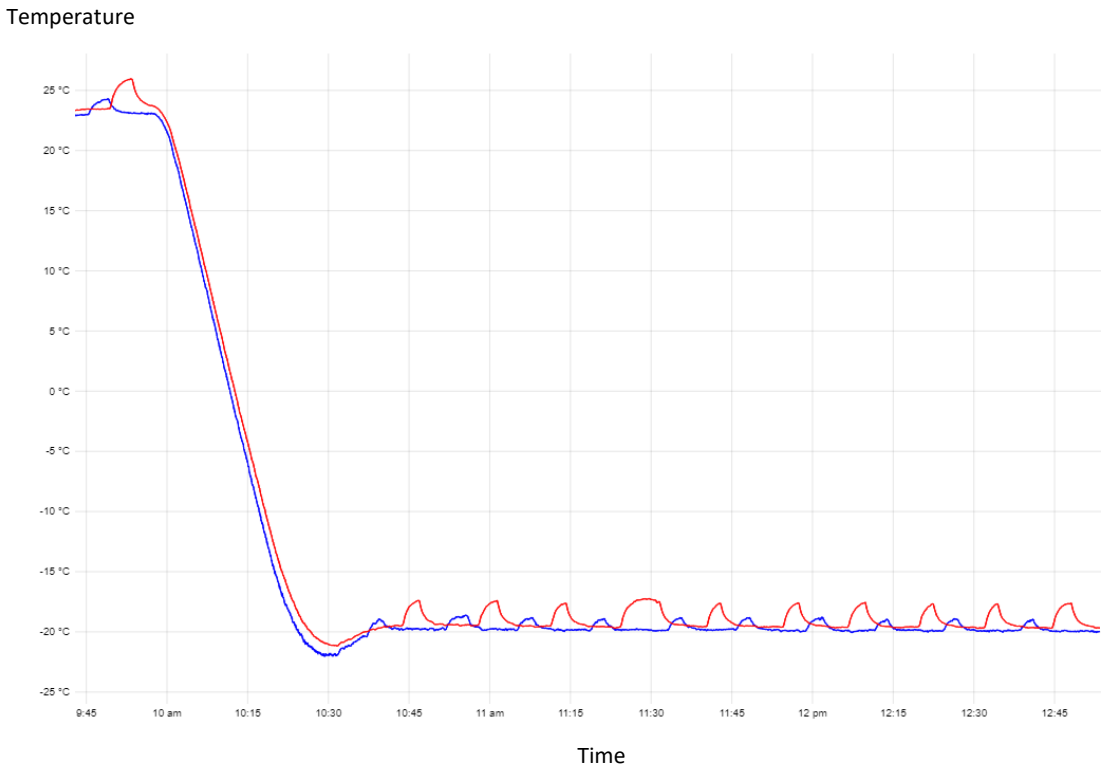


Figure 20: Thermocouple CDH temperature during test adopting traditional procedure

Each temperature oscillation is related to a successful boot of the CDH, when the boot occurs you can see a transient in which the temperature rises due to chip power consumption, during this transient, the internal temperature sensors are read in order to verify the effective operation of the CDH. A second transient is shown when the component is turned off allowing the return to a temperature near to -20 °C before the next power cycle performed by remotely controlling the "EAB POWER ENABLE" switch.

Considering **Figure 20** where are reported the results obtained adopting automated test procedure can be notice how these temperature oscillations tend to disappear. This trend is due to the reduced time interval between each power cycle which imply a stabilization at an equilibrium temperature. In order to verify effective success of at least 10 consecutive boots (requirement that has to be satisfied in order to determine the end of the cold boot test), reboot counter can be checked during test execution.

Comparing the curves obtained as output during the two tests, it emerges that test time to perform ten power cycles is reduced by over 60% in the second case, confirming advantages related to test process optimization.

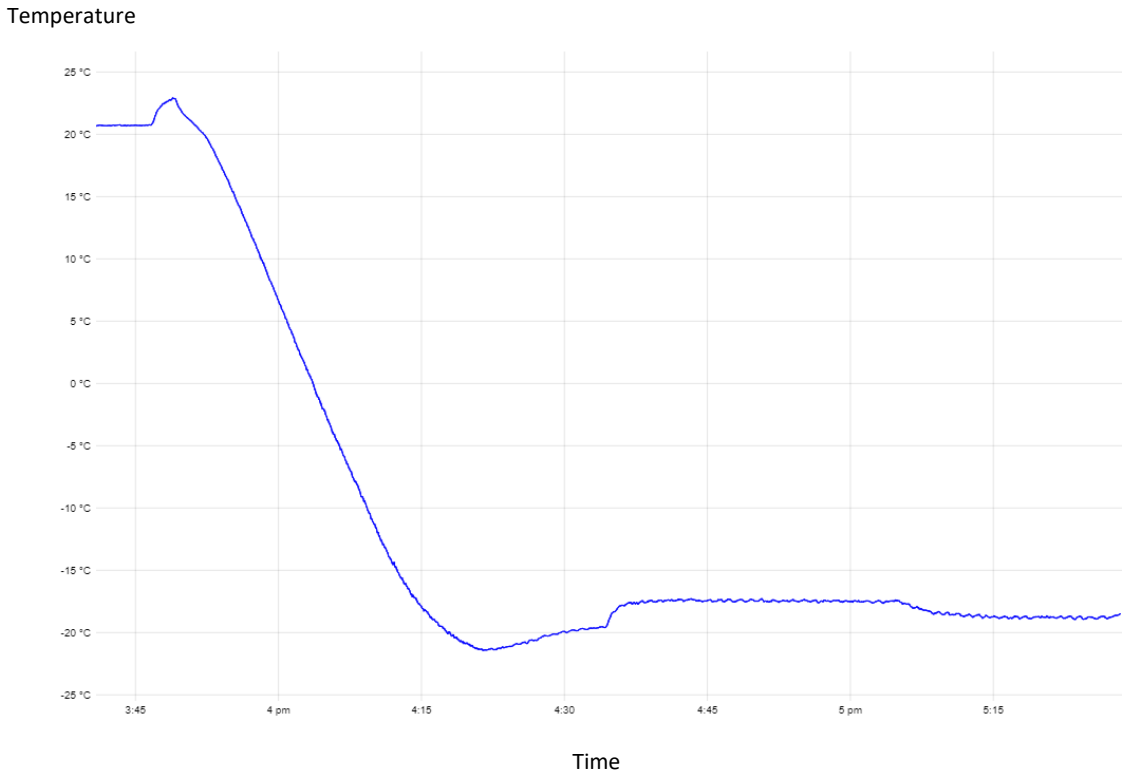


Figure 21: Thermocouple CDH temperature during test adopting automated procedure

Test performed can be considered fully successful, FlatBoard configuration, and edub_switches_manager.py script have proved the capability of the system to remotely control FlatSat through the reading and setting of switches status.
Although the positive test result, during the implementation of this solution some criticalities related to FlatBoard hardware configuration have been identified. As a consequence of this result, in order to accomplish the fully operability of this solution translatable in capability to control each switch of the board, a re-design of the board is required.

4 Sensor Emulator Board (SEB)

4.1 Context and introduction

FlatSat testing, previously introduced in **section 2.2**, represents a key step in verification process allowing the interface with satellite's sub-units and facilitating testing and troubleshooting of the system, however some limitations exist during this test.
In this configuration in facts some components of the spacecraft, like ADCS sensors (Attitude Determination and Control System) result to be inoperative or unable to provide data in accordance with sensors readings during the operational phase in orbit.
These limitations are due to the missing interaction on ground with external space environment (specially in terms of solar radiation, magnetic field and zero-friction condition) and imply the impossibility to feed the hardware under test with sensor inputs expected in orbit preventing an end to end testing of the system.
Several solutions have been investigated and designed to fill this gap allowing the emulation of specific orbit conditions in order to permit an higher level verification prospective.
A possible approach is based on adoption of facilities and GSE (depending on spacecraft's category) which reproducing physical conditions characteristics of space environment allows test of the satellite interacting with its real sensors and actuators.
Examples of these GSE are Sun Simulators adopted to reproduce solar radiation, Air-bearing platforms to emulate frictionless motion and Helmholtz Cages to simulate Earth magnetic field sensed by satellite during its orbit [].
Another approach consists on the replacement of flight hardware, which interacts with space environment in orbit, with dedicated processing hardware capable of simulating the behaviour of these components.
Considering this second approach and researching solutions capable to ensure a more efficient verification of the satellite both in terms of cost and time saving led to design and implement a GSE named SEB (Sensor Emulator Board).
Analysed in this chapter, SEB consists on a dedicated board which replacing inoperative hardware during FlatSat tests allows the emulation of their behaviour in terms of data transmission and power consumption.
The adoption of SEB offers three main advantages:

- cost reduction of the test → having an easily reprogrammable tool capable of emulating multiple hardware, it will be possible to avoid the purchase of components currently purchased exclusively for verification activities
- time savings during testing process → releasing the continuation of tests from procurement time of the hardware that can be emulated
- emulation of in-orbit operative conditions → replacing the inoperative hardware during FlatSat test, SEB feeds the spacecraft hardware under test with simulated inputs allowing the evaluation of the system response in specific operating conditions

Emulate in-orbit operative conditions, a test that actually can be performed only via software.

Being able to transfer simulated sensor data to the satellite and evaluate the outputs of their subsystems and the control algorithms response, allow the verification of the whole system.
The possibility to feed hardware under test with simulated sensor data in order to verify on-board software's response implies advantages in multiple applications both during development facilitating detection of software issues and during test phases.
Possible SEB application identified are:

- Development and test of new boards
- During module acceptance
- Test FDIR algorithm through failure injection
- During commissioning and nominal operation phase during patch testing

In each of these applications SEB during needs to act as a slave in order to emulate replaced hardware behaviour in terms of data transmission.
The "slave" concept is related to I2C or SPI buses, these two serial communication protocols (analysed in depth in section 4.4) have been adopted as available external interfaces in this SEB first revision.

4.2 SEB design

SEB design process can be resumed in following steps: definition of system design drivers, functional analysis, requirements.
Solution implemented during this thesis project represents a prototype to evaluate capabilities and applicability of this Ground Support Equipment (GSE) in prospective of future revision of the board, however the approach adopted during design phase was based on identification of all the different function in a full operative version taking in consideration design choices and predispositions need to accomplish additional functions in next versions.

4.2.1 Design drivers

In the first stage of the project have been identified SEB design drivers with the aim to lead system design and derive system high level requirements.
Design drivers have been derived from Tyvak needs emerged from company experience on CubeSats test and verification process.
SEB design drivers identified are:

- **Accessibility**: SEB shall be easy and fast to configure
- **Reusability:** indicates capability of SEB to be adoptable both multiple times during development and verification of a single satellite and to be adoptable to different mission
- **Cost:** SEB solution total cost shall result advantageous compared with actually adopted testing solutions

4.2.2 Functional analysis

Sensor Emulator Board functional analysis coincided with the definition of:

- Functional tree
- Product tree
- N2 matrix

During functional analysis four decomposition levels have been considered, starting from system level associated to Sensor Emulator Board, also subsystem, unit (o segment) and component level have been taken in consideration.
Functional Tree reported in **Figure 21** has been developed as part of an iterative process in order to identify functions that SEB have to perform at different levels.
Once figured out the complete set of functions which allows to accomplish system main function, the associated hardware and products have been identified and reported in SEB Product Tree illustrated in **Figure 22**.
The association between function and the relative component which allows its execution in all different SEB levels is specified in Function-Component matrices reported in **Annex 2**.

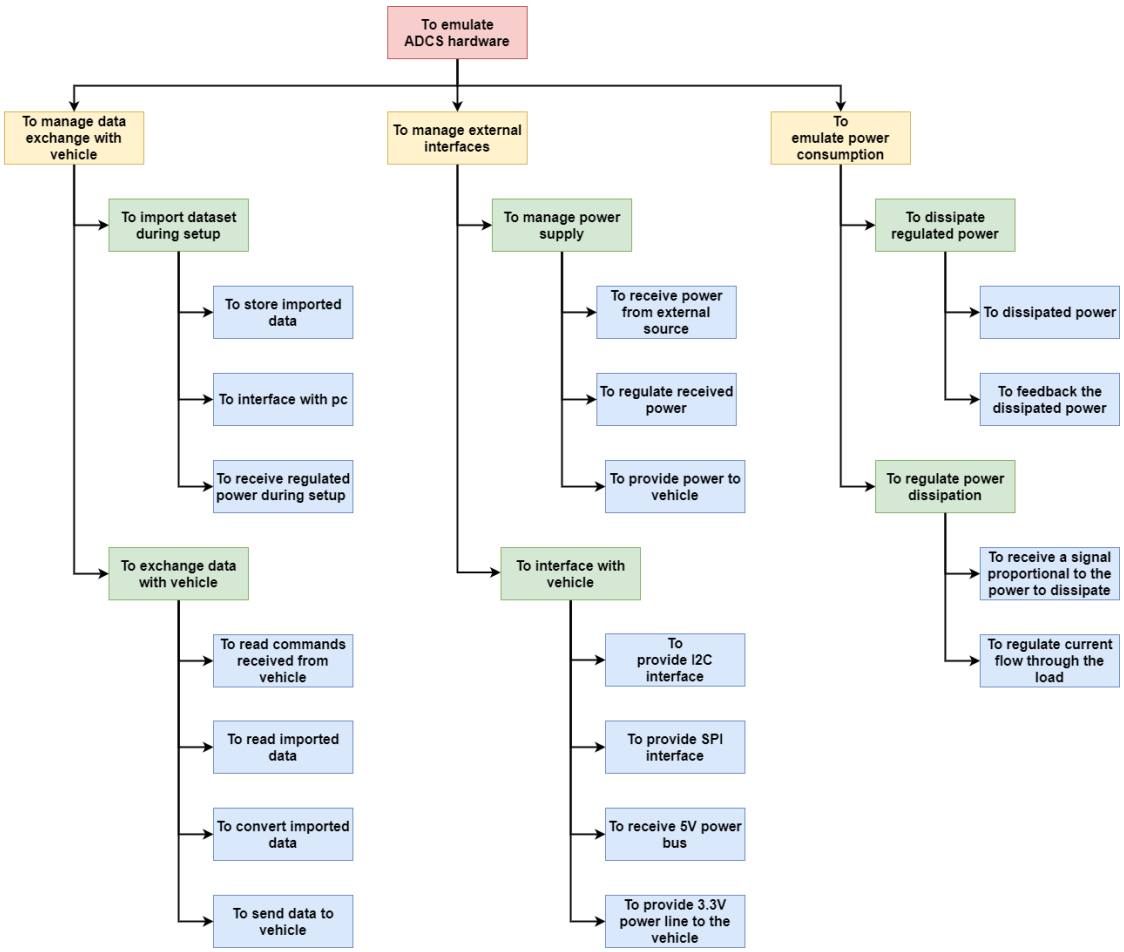


Figure 22: SEB Functional Tree

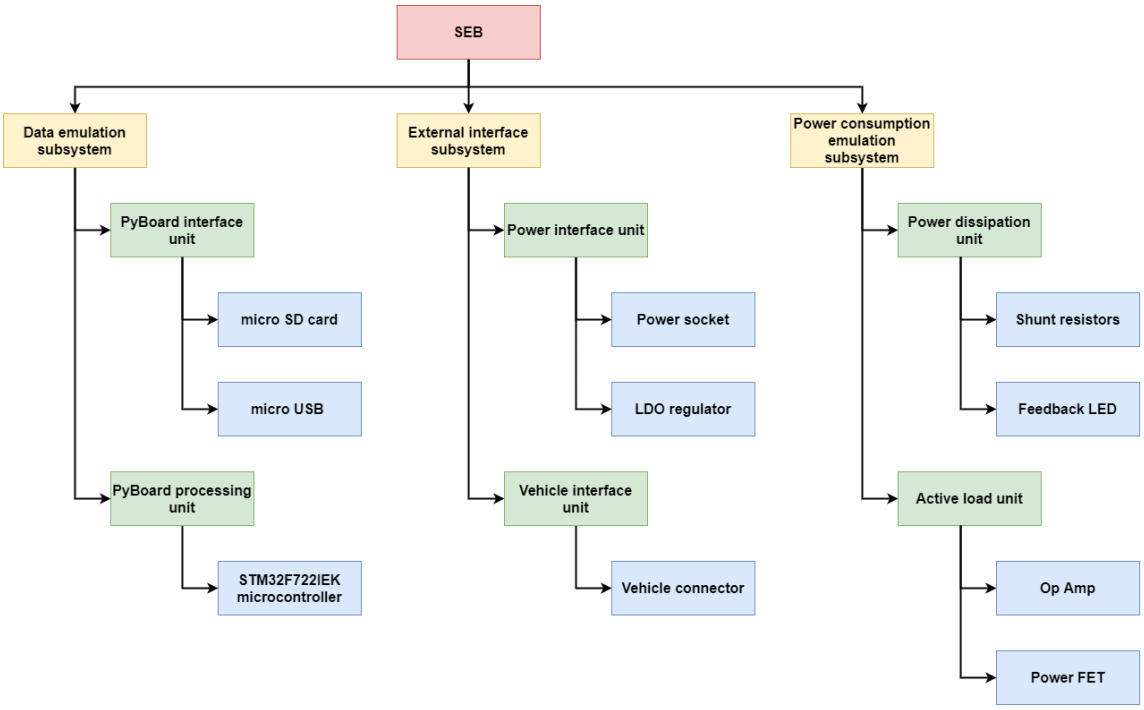


Figure 23: SEB Product Tree

At the system level, the sensor emulator board can be associated with the macro function "Emulate ADCS hardware" highlighting the objective of this GSE to emulate not only sensors but also actuators associated with the Attitude Determination Control System. The choice to emulate components associated with this subsystem is due to the complexity of performing system-level testing involving this hardware reproducing operative conditions.
To accomplish this macro function three functions have been identified at subsystem level:

- Data exchange management with the spacecraft
- External interfaces management
- Emulation of power consumption

Management of data exchange with the CubeSat sub-assemblies under test is a performed by SEB's data emulation subsystem represented in this first revision of the SEB by PyBoard microcontroller, which will be analysed in detail in the next section.
Pyboard plays a fundamental role in the emulation of the data of a specific hardware, representing the processing unit of the Sensor Emulator Board it will host and execute the code responsible for the communication with the processing units on board the satellite.
Pyboard is responsible for interpreting the commands or requests received by spacecraft on-board computer and for processing a response, which has to respect the constraints set by the communication protocol and which has to be consistent with the type of response expected by the replaced sensor.
In order to elaborate this response, during emulation process, a previous setup step is needed. During this step (following analysed in detail in section 5.1) the Pyboard is interfaced with the PC adopted by the operator to coordinate the test. This connection can be established using the Pyboard's micro USB port, in this way will be also possible supply the microcontroller with power needed during this setup phase.

During setup it will be possible to import the dataset from which the data will be extracted, and converted during the emulation phase. The imported dataset represented by a csv file will be stored on an external SD card and read by the Pyboard through a dedicated port.

Regarding the management of SEB's external interfaces in addition to the interface through pc used during the setup phase, the SEB provides:

- interface dedicated to the power supply of the SEB
- interface dedicated to communication with the satellite

SEB power interface unit collects the components that allow to supply power to the other subunits, and active components.
During the emulation phase, power is supplied by a dedicated GSE as an external power supply interfaced to the SEB via a power socket. Once received, power is regulated through an LDO (Low Drop Out regulator) to Pyboard and Active load unit. According to power requirements of these active components resumed in **Table 4** a 5V input is expected from SEB power socket.

Active component	Power supply requirements
Pyboard	3.4V @ 1A
Active load unit – Low power Op Amp	3.3V @ 0.1mA

Table 4: SEB active components power requirements

Considering the interface between SEB and the sub-units of the satellite under test, it will be necessary to distinguish between two different channels:

- the data communication channel represented by the I2C and SPI buses
- the power bus through which the satellite electric power system powers the emulated sensors in flight configuration

The choice of adopting I2C and SPI (analysed in detail in section 4.5) as data interfaces available to the satellite, is linked to the common use of these protocols for communication with sensors and other distributed devices, for which point-to-point communication is not it would be functional.
Considering the application selected in this thesis project power channel is represented by 5V line in input and a 3.3V line provided by the sensor module to the spacecraft.
Finally the power consumption emulation function is covered by a dedicated subsystem interfaced both with Pyboard and with vehicle power bus, details related to this subsystem are reported in section 4.3.2.

4.3 SEB architecture and critical components

Defined the functional architecture of the SEB and identified the interfaces at the component level through the functional analysis, the physical architecture of the system reported in **Figure 23** was defined.

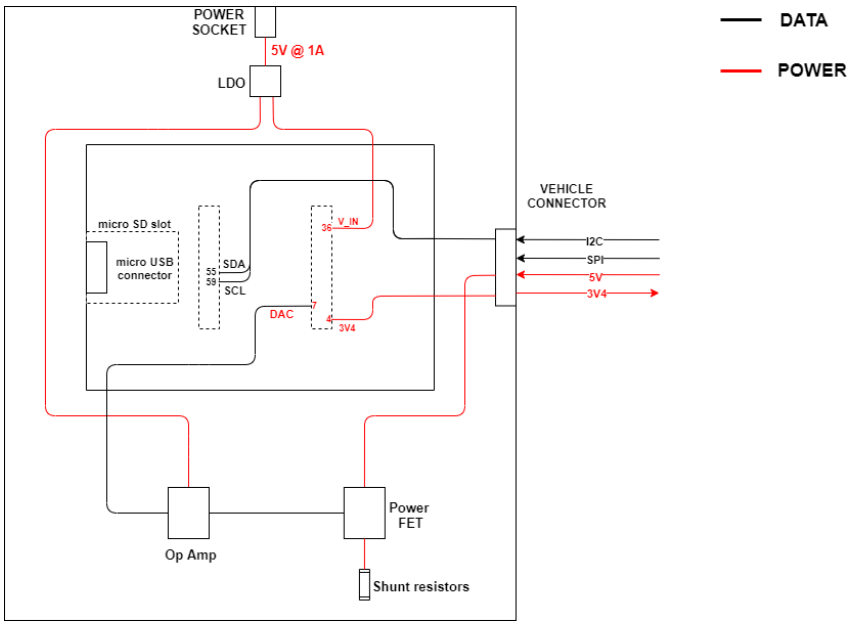


Figure 24: SEB architecture

The physical architecture shown in the figure shows how two physical blocks of the SEB can be identified.
The first block is represented by the Pyboard which performs the functions of processing unit and interface unit during the set-up phase, while the second block is represented by a customized base board hosting the power consumption emulation subsystem and the external interfaces related to the SEB power supply and interface with the vehicle.
These two boards are connected through WBUS header, a 42+42 pin mezzanine bus connector giving access to all power and IO ports. Details of WBUS connector are extracted from Pyboard D-series schematic [] and reported in **Figure 25**.

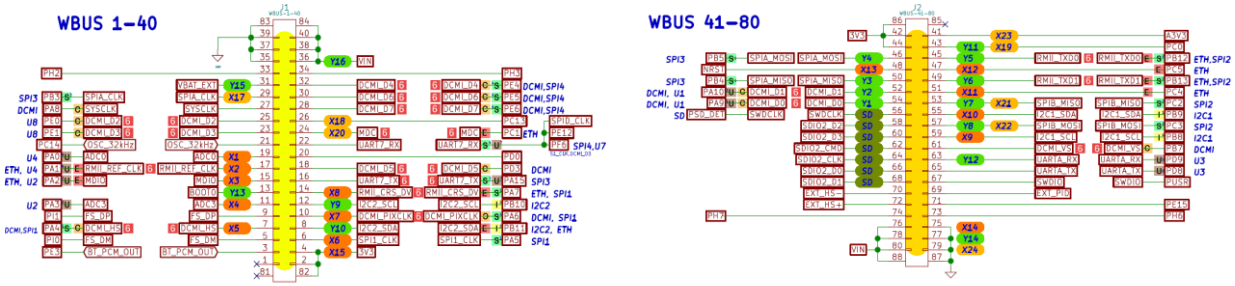


Figure 25: Pyboard WBUS connector pins

In **Figure 25** is reported FlatSat configuration highlighting GSE setup during SEB emulation phase. Each SEB will be characterized by an independent power line through

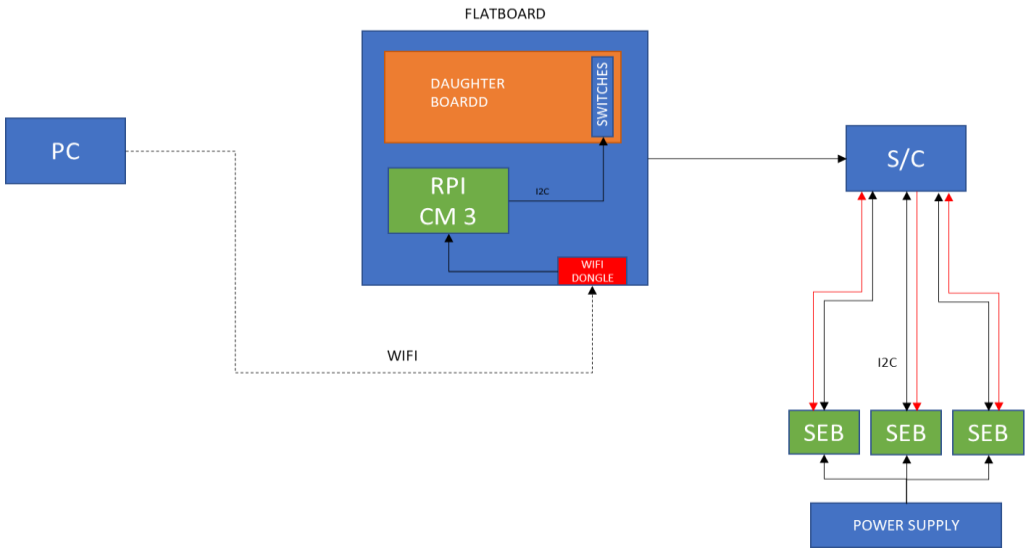


Figure 26: SEB configuration diagram during test

4.3.1 Micropyhton and Pyboard

According to design driver, and purpose of this activity, the developed GSE needs to be a device easy and fast to program, able to act as an i2c slave, and able to communicate with the spacecraft adopting the same protocols adopted in flight configuration by the emulated hardware. These needs lead to the selection of programming language and SEB's critical components. Regarding programming language Micropython has been selected over C and Python. C is a complied programming language this mean that every code changes in order to execute it is needed to compile and re-upload the code on microcontroller. This process may result time consuming during test procedure.

MicroPython is a lean and efficient implementation of the Python 3 (version 3.4) programming language that includes a small subset of the Python standard library and is optimised to run on microcontrollers [13].

"Easy and fast to program" is a necessary feature for SEB, which like other GSE has to support test, offering the capability to be reprogrammed also during test avoiding delay or relevant interruption of the test procedure, this aspect brought to prefer languages such as Python above C. However Python requires to run on pc or board equipped with a processor supported by an operative system, such as a Raspberry Pi device with NOOBS or Raspbian.

Difficulties related to the configuration of Raspberry Pi as slave device in I2C and SPI communications, lead to choose Micropyhton as programming language as a compromise between Python and C capabilities.

Factors which have led to use Micropython on a microcontroller instead of a complied language like C are related to advantages during prototyping.

If for some reason working for example with an Arduino the program crashes, it is complicate figure out the reason of the failure. Adopting Micropython is possible to detect the error and rapidly fix it reducing the time of prototyping, with Micropython you can run your code locally first and test it before uploading to the board. MicroPython provides an interactive prompt (the REPL) to execute commands immediately, along with the ability to run and import scripts from the built-in filesystem. In addition MicroPython aims to be a solution for low power applications, being very light in terms of lines of code and not requiring a lot of space to run.
Another advantage are the numerous syntactic analogies between Micropython and Pyhton. Given the high popularity of the latter, the operator responsible for completing the tests will not be required to train on a new programming language before actively using the Sensor Emulator Board.
MicroPython is used mainly by Makers, Educators and Software Developers, but it represents a good solution also for industrial applications, several companies and start apps are already using MicroPython in order to exploit its benefits. Growing interest on MicroPython is also proved by ESA involvement on funding further MicroPython development to determine suitability of the language for space-based applications and to make MicroPython more robust for critical embedded systems.
In late 2013, Damien P. George launched a campaign on Kickstarter to finance the MicroPython project. This project aims to implement the Python 3 interpreter on a SoC (in particular STM32F405RG SoC) in order to allow its programming via a subset of Python instructions.
The results of mentioned crowdfunding campaign also financed the implementation of an official Micro Python board named PyBoard.
The Pyboard D-series model shown in Figure 27 has been selected as SEB's processing unit.
The factors that led to choose this microcontroller, compared to the other boards capable of supporting Micropyhton, are: better performance in terms of processing and memory capacity (216MHz CPU, 256k RAM), the possibility of interfacing via wifi and finally the ability to support I2C slave mode.

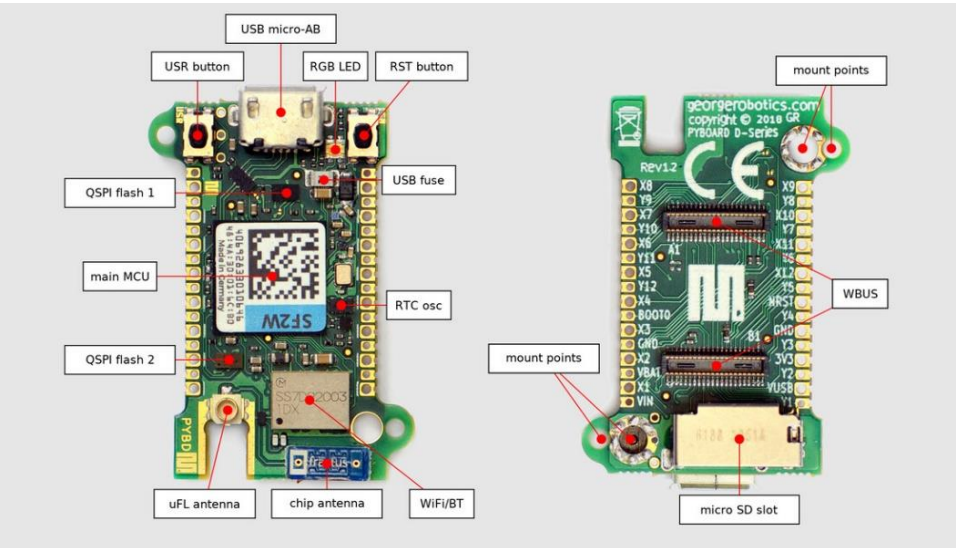


Figure 27: PyBoard D-series

4.3.2 SEB power consumption emulator

To support a complete system verification of the CubeSat under test, in addition to the possibility of simulating data transfer, SEB design allows to reproduce the power consumption of the emulated hardware.
Some components such as the sensors emulated in this first phase will be characterized by a low power consumption which may be negligible in relation to the satellite's power budget. However, considering future emulation of components such as IMU, magnetic rods and reaction wheels, during the SEB design phase, a preliminary analysis of the power consumption emulator subsystem was carried out in order to implement the "regulated dissipation" of the power received by the power satellite bus.
The implementation of this function may allow the verification of CubeSat's power safe condition, extending verification beyond software level.
The emulation of power consumption of different hardware requires the capability to have a variable resistive load allowing a regulated dissipation depending on the hardware emulated and depending on its operating mode. This requirement led to design an active load, which controlled by Pyboard guarantees flexibility regarding the level of power dissipated.

The proposed solution to implement this function foresees the adoption of three main components:

- power mosfet
- operational amplifier
- shunt resistors

In order to manage power dissipation through a "controllable" resistance, a dedicated power FET can be adopted.
Control the gate-source voltage can be influenced the drain-source resistance where power is received from vehicle through vehicle connector.
In order to change the power dissipation, a simple controller is needed, the solution proposed consists on the adoption of an Op-Amp to drive power mosfet gate.
The load current flows from mosfet to a "sense resistor": mosfet's drain is connected to the positive rail, and its source is connected to one end of the resistor while the other end of the resistor goes to the ground rail. This resistor will be in charge of dissipate power received by mosfet from CubeSat power bus.
Resulting voltage on this resistor is fed back as inverting input of the opamp creating a feedback loop.
In this feedback loop the opamp will adjust the voltage on the FET's gate to regulate the current flowing through the mosfet and consequently the voltage across the sense resistor (proportional to power dissipated) ensuring it is equal to the voltage on its positive terminal provided by pyboard analog pin.

4.4 SEB external interfaces

SEB external data interfaces can be divided in interfaces towards the pc and interfaces towards the FlatSat. Firsts are represented mainly by Pyboard micro-USB connector which allows to power the board and exchange data during setup phase before starting of simulation.
Alternative configuration foresees to plug the board via micro-USB to a power charger and communicating to pyboard processor via wifi.
The SEB interfaces towards CubeSat subsystems under test have been selected considering communication protocols adopted in flight configuration with the components replaced by SEB.
As previously mentioned emulated hardware in flight configuration act as a slave in communication with satellite processing unit, in order to emulate I2C and SPI slave behaviour bit banging technique has been implemented.
The bit banging is a technique for data communications which employs software as a substitute of dedicated hardware module to generate transmitted signals or to process received signals. To implement this technique means develop the code to control the state of the MCU pins, keeping in account all parameters of the signals: timing, levels and synchronization.
In order to explain the implementation of I2C (and SPI) slave behaviour using bit banging an overview of the protocols is needed.

4.4.1 I2C protocol

The I2C interface is a two wires serial bus which allows the communication between multiple masters and multiple slaves, it is a very common bus used in particular for interface with sensors or peripheral devices that need to be addressed only occasionally.
Even if characterized by a lower bit rate compared to other protocols (100 kbit/s in Standard mode or 400 kbit/s in Fast mode), I2C protocol allows to up to 127 slave devices (identified by a 7-bit address) to exchange data on the bus through only 2 wires (in addition to power supply Vcc and GND): SDA (serial data line) and SCL (serial clock line).
Both SDA and SCL are open-drain, feature that allows to use each line for bidirectional data flow. Open drain setup means that signals on each line are generated driving low (generally to ground) the bus voltage or releasing it and let it be pulled high by pull up resistors which connect SDA and SCL to Vcc.
A simplified scheme of an I2C device control on the SDA/SCL lines is reported in Fig.X, [?] it consists on a buffer to read input and a FET to transmit data.
As shown in Fig.X+1 when the output foreseen is a low logic the FET is enabled providing a short to ground which pulling the line low, while when the device need to transmit an high logic the FET will be disabled generating an high impedance to ground and consequently releasing the line and allowing the pull up resistors to raise the voltage.

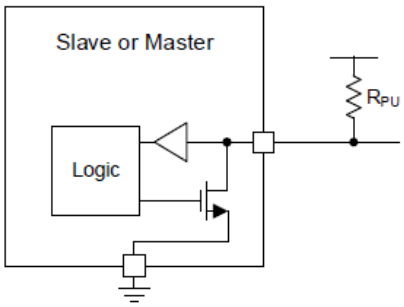


Figure 28: I2C device control on SDA/SCL line

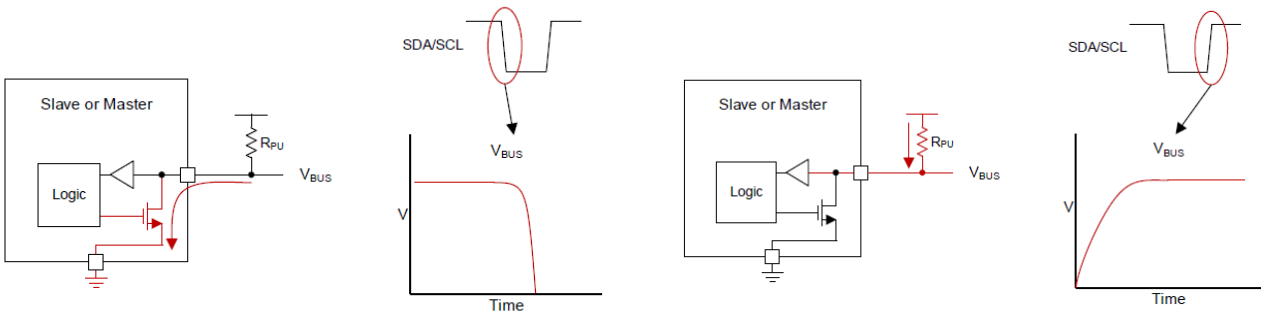


Figure 29: Pulling low (left) and releasing (right) of SDA/SCL

In I2C communication the master controls data transfer providing clock through SCL and start bus transition.
Following is described the general procedure for master communication with a slave device **[]**.
Bus transition starts when START signals is sent by master in the first bit, this condition occurs when there is an high to low transition on SDA with SCL held high.
After the START, the master drives SCL low and sends the first byte which stores the I2C 7 bit address of the slave followed by R/W bit (1 = read, 0 = write).
During bit transmission SDA is stable when SCL is high, and transitioning when SCL is low.
If a slave device matches the address, it responds with an ACK bit pulling the SDA low.
ACK bit follows each byte transmitted on the bus, it is generated by the receiver (be it a master or a slave) to confirm the successfully reception of the byte.
If SDA line remains high during ACK/NACK related clock period, this is interpreted by the transmitter as a NACK which aborts the communication.
Ones received the ACK from the slave the master need to select the slave's register, a location in slave's memory where it will write or read data. To select it, the master has to send a byte containing the register address of the register it wishes to write or read to (depending on R/W bit).
If the master is writing to a slave, sent the register address and received the relative ACK, it will start sending bytes of data to the slave, and once finished it will terminate the transmission with a STOP condition. On the other hands if the master need to read data from slave the process required an extra step, first the master sends slave address with R/W

bit equal to 0 (write) followed by the needed register address in order to instruct the slave on which register it wishes to read from, then received the ACK it sends a START again followed by the slave address with the R/W bit set to 1 (read), now ones slave acknowledged the read request, the master releases the SDA bus continuing to supply the clock.
In this way the slave can start sending data stored in selected register while the master became the receiver will answer with an ACK at the end of every bytes of data successfully received.
Once the master has received the number of bytes it is expecting, it will send a NACK, to the slave (in order to halt communications and release the bus) followed by a STOP condition (defined by a low-to-high transition on the SDA with the SCL held high) setting the bus in idle.

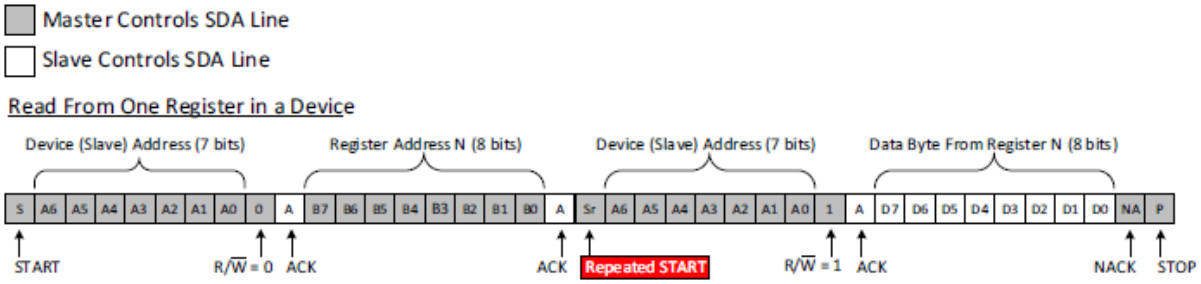
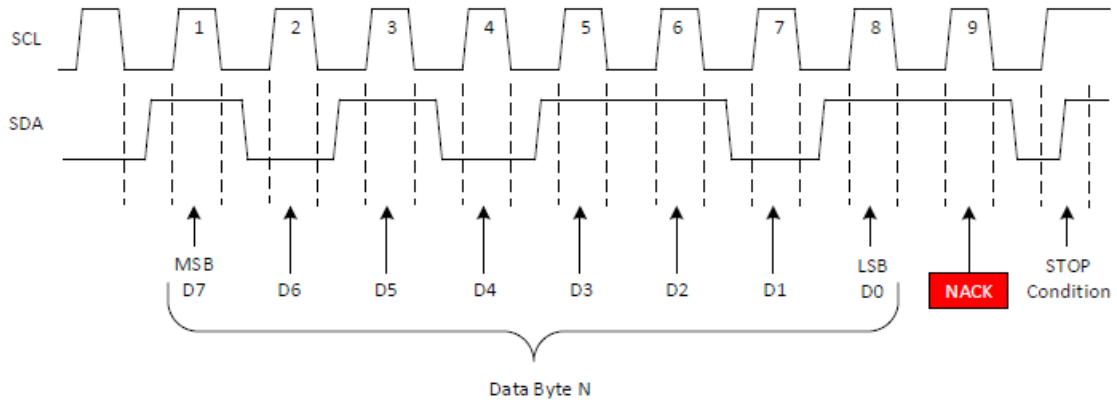


Figure 30: Example of I2C reading sequence



4.4.2 SPI protocol

Like for I2C, the SPI (Serial Peripheral Interface) is a standard synchronous communication bus when communication occurs between a master a one or more slaves.
The SPI master is responsible of controlling the bus sending the clock and initiating each bus transaction. The communication in SPI protocol occurs through four lines: MOSI (Master Out Slave In), MISO (Master In Slave Out), CLK (Clock) and CS or SS (Chip/Slave Select).

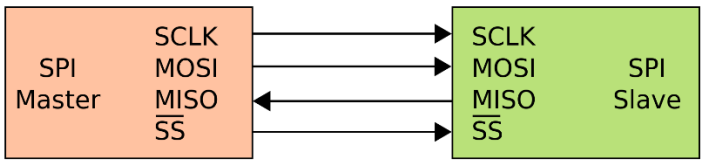


Figure 31: SPI protocol

MOSI and MISO are respectively the lines where data are sent from the master to the slave and from the slave to the master, CLK is the serial clock controlled by the master and marks the timing of the emission and reading of each bits on the data lines and finally CS, issued by the master to choose which slave device he wants to communicate with (a CS line exists for each slave), it must be put at a low logical level to communicate with the slave device (following an active low logic).
This protocol is full duplex and faster compared with other common protocol such as I2C or UART.

4.5 Sensors emulation process

In order to emulate a specific sensor with SEB two steps are required:

- generate csv containing data which will be converted, packed and sent to the spacecraft once requested
- analysis of emulated sensor datasheet in order to identify its internal register structure, data conversion and register writing logic (this step is described in next subsections for each sensor considered)
- implement SEB Micropython internal code running on Pyboard

While first step can be performed adopting external software and data depending on the emulated sensor and application, second and third steps are related to Pyboard logic adopted to implement via software hardware slave capabilities, **Figure 33** block diagram resumed this slave logic in I2C bus communication.

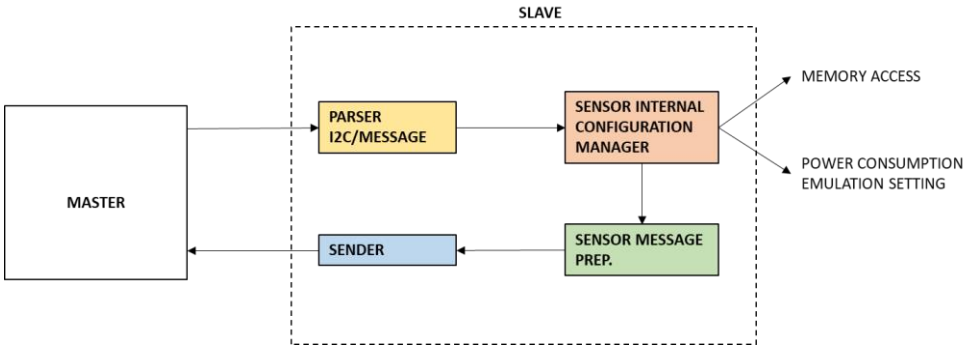


Figure 32: SEB algorithm functional block diagram

Each block identified in previous scheme can be decomposed in code subsections, translatable in specific in lower level instructions implemented adopting Micropython available API.

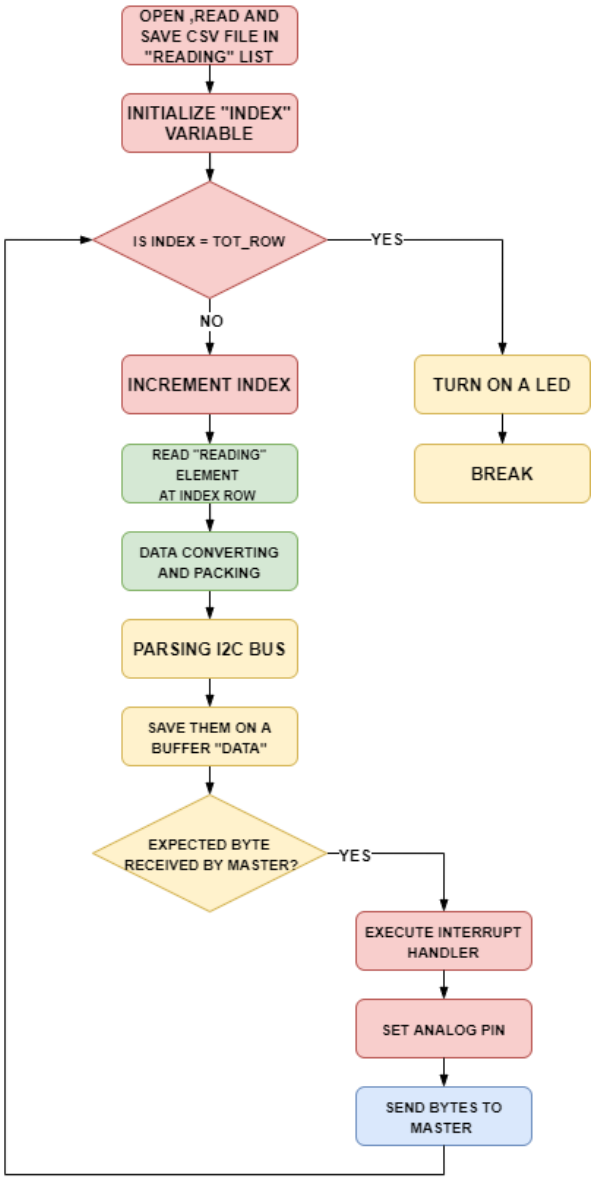


Figure 33: SEB emulation process flowchart

Once described the framework of the code responsible for managing the emulation process, the list of sensors and actuators that are interested in emulating through the SEB is reported in **Table X** identifying for each of these components the necessary input data (collected in the

csv and imported before the simulation), output, and the characteristics related to power consumption.

Emulated hardware	Input from csv	Output read by s/c	Communication protocol	Power consumption
Sun sensor (Sensor Module)	- Azimuth (deg) - Elevation (deg) - Range (deg) - Face reference	- Received intensity (V) - Theta angle (deg) - Phi angle (deg)	I2C / SPI	Neglectable
Magnetometer (Sensor Module)	- X magnetic field measurement (mG) - Y magnetic field measurement (mG) - Z magnetic field measurement (mG)	- X magnetic field measurement (mG) - Y magnetic field measurement (mG) - Z magnetic field measurement (mG)	I2C	Neglectable
Temperature sensor (Sensor Module)	Temperature (degC)	Temperature (degC)	I2C	Neglectable
IMU	Not in this thesis	Not in this thesis	Not in this thesis	Not in this thesis
Torque rods	Not in this thesis	Not in this thesis	Not in this thesis	Not in this thesis
RW	Not in this thesis	Not in this thesis	Not in this thesis	Not in this thesis

Table 5: List of SEB emulable components

This thesis has focused on the emulation of sun sensors, magnetometer and temperature sensor taking into account in the design phase the SEB's predisposition to emulate also the other hardware listed in the table, such as IMU, Torque rods and reaction wheels.
The choice which leads to focus on these three sensors is based on the analysis of Tyvak developed spacecraft bus which foresees the implementation of these three sensors installed on ad dedicated sensor module positioned on five of the six panels of the company developed CubeSat as shown in **Figure 31**.

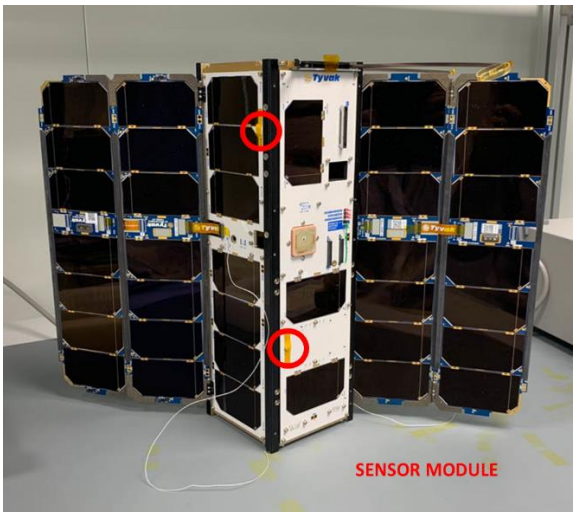


Figure 34: Example of sensor modules positioning

The aim is to emulate in a cost e time effective way satellite in orbit conditions in terms of data acquisition and system power consumption involving real hardware in system level simulation.
SEB represents a first prototype to reach this level of simulation where through the emulation of attitude sensors inputs and the reproduction of actuator effect on system (in terms of power consumption) will be possible test satellite behaviour and responses before and after the launch.
The idea of emulating the behaviour of sensors and actuators of a satellite through dedicated GSEs has also been applied to large satellites (as shown in Figure 32), however the reduced level of complexity and the small dimensions characterizing the CubeSat platforms allow to adopt this solution keeping costs limited.

Figure 35: Rosetta EQM hardware emulation

Register reading and writing activities performed by master have priority over internal activities, such as index increment and message preparation. The purpose of this priority is to avoid the master waiting and the I2C bus engaged for longer than necessary.
In following subsections for each of the three previously mentioned sensors we are interested to emulate, a detailed analysis is carried out. Purposes of this detailed analysis are:

- Identify sensor's register structure in order to understand reading and writing process
- Identify how the digital reading is converted in from binary to decimal readable form, in order to implement in MicroPython the inverted operation which allows to convert the value read from imported dataset in spacecraft expected form
- Identify how the converted value is structured at bit level in the registers read by the master during i2c communication

In order to highlight the time-effective advantages associated with the SEB solution, it is important to highlight that the first and second steps do not represent additional steps to the traditional test process. The analysis of the reading and writing process and the conversion of the data read into an operator-readable telemetry value is carried out during the implementation of the device drivers used during the sensor acceptance and performance tests. This information was extracted from the datasheets for each of the three components, reported in the following documents **[13] [14] [15]**.

4.4.1 Temperature sensor

The emulated temperature sensor is AD7415 which characterized by a 10-bit ADC provide a temperature reading in range -40°C to 125°C with a resolution of 0.25°C.
This sensor has three internal register, two data registers (Configuration register and Temperature value register) and an address pointer register.

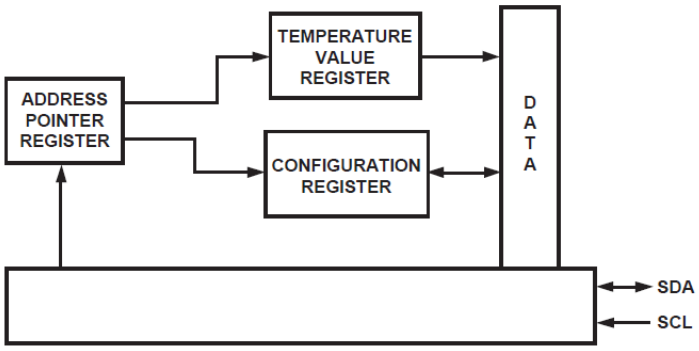


Figure 36 Temperature sensor Register Structure

Address pointer register stores a 1-byte address that points to Configuration Register (0x01) or Temperature value Register (0x00).
Before being read, some sensors require a dedicated configuration, in AD7415 the operating mode can be set writing in 8-bit read/write Configuration Register, in particular only three of

the bits are used (D7, D6, and D2) to set the operating modes while D0, D1, D3, D4 and D5 are used for factory settings and must have zeros written to them during normal operation.

D7	D6	D5	D4	D3	D2	D1	D0
PD	FLTR	TEST MODE			ONE SHOT	TEST MODE	
0[1]	1[1]	0s[1]			0s[1]	0s[1]	

D7	Full power-down if = 1.
D6	Bypass SDA and SCL filtering if = 0.
D2	Initiate a one-shot temperature conversion if set to 1. The bit status is not stored; thus this bit is 0 if read.

Table 6: Configuration register setting

The temperature value register is a 10-bit, read-only register that stores the temperature reading from the ADC in format defined in **Figure X**, consequently two reads will be necessary to read data from this register.
One shot temperature reading can be set via configuration register, in this case if temperature sensor is in power down mode, the sensor will be power up, will perform a single conversion and will power down again.

D15	D14	D13	D12	D11	D10	D9	D8
MSB	B8	B7	B6	B5	B4	B3	B2

D7	D6	D5	D4	D3	D2	D1	D0
B1	LSB	N/A	N/A	N/A	N/A	N/A	N/A

Table 7: Temperature value register

Temperature (°C)	Digital Output (DB9,…,DB0)
-55	1100100100
-50	1100111000
-25	1110011100
-0.25	1111111111
0	0000000000
0.25	0000000001
10	0000101000
25	0001100100
50	0011001000
75	0100101100

Figure 37: Temperature conversion

According to measurement range, temperature will be converted following this format:

$$positive\ temperature = \frac{digital\ output}{4}$$

$$negative\ temperature = \frac{digital\ output - 512}{4}$$

During sensor message preparation these conversions are performed starting from positive or negative value read by csv file in order to obtain the related digital output which will be packed and sent to master in case of request.

4.4.2 Magnetometer

The emulated magnetometer is Honeywell HMC5883L, a 3-Axis Magnetoresistive Sensors which supported by a 12-bit ADC Coupled with Low Noise AMR Sensors allows to achieve a resolution of 2 mG in ±8 Gauss Fields translatable in 2° determination accuracy.
Once applied power supply, the sensor converts any incident magnetic field in the sensitive axis directions to a differential voltage output.
The magnetoresistive sensors are made of a nickel-iron (Permalloy) thin-film and patterned as a resistive strip element. In the presence of a magnetic field, a change in the bridge resistive elements, aligned on a common axe causes a corresponding change in voltage across the bridge outputs.
This magnetometer provides an I2C serial bus interface and requires a supply voltage up to 3.3V @ 0.1 mA during measurement which can be set in a single or continuous mode writing on Mode Register. Moreover an Idle mode is available to use other devices on the I2C bus, in this mode the device is accessible through the I2C bus, but all registers maintain their values, and major sources of power consumption are disabled.
In orbit Single measurement mode is adopted for this sensor, describing reading and writing process which will be emulated, the sensor's registers available (reported in **Table X**) are described:

Address Location	Name	Access
00	Configuration Register A	Read/Write
01	Configuration Register B	Read/Write
02	Mode Register	Read/Write
03	Data Output X MSB Register	Read
04	Data Output X LSB Register	Read
05	Data Output Z MSB Register	Read
06	Data Output Z LSB Register	Read
07	Data Output Y MSB Register	Read
08	Data Output Y LSB Register	Read
09	Status Register	Read
10	Identification Register A	Read
11	Identification Register B	Read
12	Identification Register C	Read

Table 8: Magnetometer registers list

During single-measurement mode, once the measurement is complete output data registers are updated, and the device is placed in idle mode changing the Mode Register (by setting MD[n] bits) where all registers will maintain same values while in single-measurement mode and I2C bus is enabled for use by other devices on the network.
Settings in the configuration register affect the measurement configuration (bits MS[n]) in particular the configuration register A is used to configure the device for setting the data output rate and measurement configuration, while configuration register B for setting the device gain.
The devices adopt an address pointer to indicate which register location is to be read from or written to.
As shown in **Table 5**, six different 8-bit data output registers are available to store the measurement result from channel. Considering for example Data output X registers, two register exist for each channel (i.e. Data output X register A and Data output X register B), as shown in **Table 6**, the value measured is stored in these two registers is a 16-bit value in two complement form , whose range is 0xF800 to 0x07FF.

$$2's\ complement\ value\ of\ n_{10} = (2^{16} - n)_2$$

DXRA7 and DXRB7 denote the first bit of the data stream, and number in parenthesis indicates the default value of that bit.

DXRA7	DXRA6	DXRA5	DXRA4	DXRA3	DXRA2	DXRA1	DXRA0
(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)
DXRB7	**DXRB6**	**DXRB5**	**DXRB4**	**DXRB3**	**DXRB2**	**DXRB1**	**DXRB0**
(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

Table 9: Data output X registers A and B

Considering aspects related to communication timing HMC5883L magnetometer has a fairly quick stabilization time from no voltage to stable and ready for data retrieval, adopting default single measurement mode after 56 milli-seconds six bytes of magnetic data registers (DXRA, DXRB, DZRA, DZRB, DYRA, and DYRB) are filled with a valid first measurement.

4.4.3 Sun sensor

The emulated sun sensor is composed by:

- E910.86 ELMOS integrated solar angle sensor
- MCP3421 18-bit ADC

The emulated sun sensor is E910.86 ELMOS integrated solar angle sensor which provide angle of light incidence in XZ an YZ planes with their relative intensity with a resolution of 2.7° in a measurement range between 15° and 165°.

While angles data can be read through SPI interface, data related to solar intensity is provided as analog outputs to the dedicated 18-bit ADC, which ones sampled the intensity signal received will provide the voltage measured to the master through I2C interface.

5 SEB TEST CASE

Described in the previous chapter highlighting its characteristics, critical components and operating logic, the Sensor Emulator Board in this chapter is analysed with the aim to verify its expected operations in a real application through a dedicated test case.
Before introducing the test case described in the next sections, it is highlighted in Figure 38 as the main strength of the SEB system (represented in the figure by its main element, the Pyboard) is to exploit the data generated by different sources and software in order to allow a wider spectrum of verification able to involve the hardware, keeping limited development costs.
The adoption of the SEB to emulate the behaviour of one or more sensors requires the availability of a dataset associated with these sensors. This dataset can be derived from software used in the design phase such as Matlab Simulink or STK or alternatively, if emulation process is necessary in a post-launch phase, from telemetry data derived during the commissioning.
Using common formatting, such as CSV format (comma-separated values), these data, once collected, converted and packed according to the emulated hardware features, can be read by a FlatSat or by specific satellite sub-unit. This process will allow to evaluate the answer of the system under test to these inputs hardly reproducible on ground during testing.

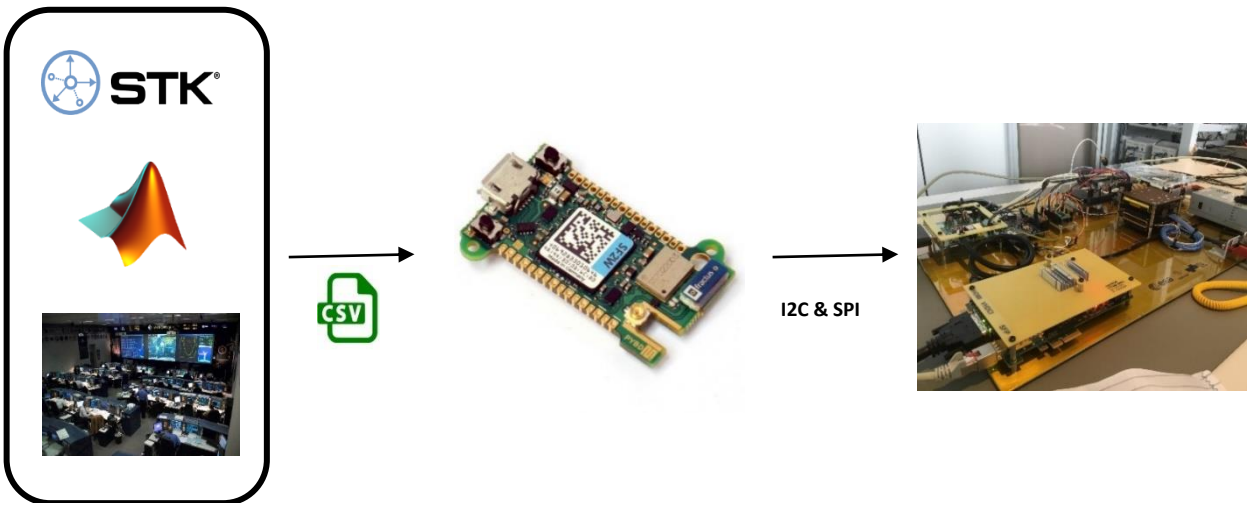


Figure 38: SEB flow operation

The next sections describe the test procedure and the temperature sensor emulation test case. In relation to the development phase of the system, in this first test case can be verified and validated the correct operation of Sensor Emulator Board in terms of data emulation.
As a consequence, during the test SEB will be replaced only by its data processing unit represented by the Pyboard.

5.1 Test procedure

SEB emulation process can be divide in two main phases:

- initial setup
- sensor emulation phase

During initial setup SEB is interfaced with a laptop through Pyboard microUSB interface, in order to power on the SEB processing unit and transfer on Pyboard SD card the csv file storing dataset associated to sensors expected to be emulated.
PyBoard is characterized by a built-in internal filesystem stored in microcontroller's flash memory (identified by */flash* path), if and external SD card is used during the boots up, then /sd filesystem will be used as the boot filesystem.
Interfacing Pyboard (if no SD card are mounted like in this test) with the pc following window will pop-up:

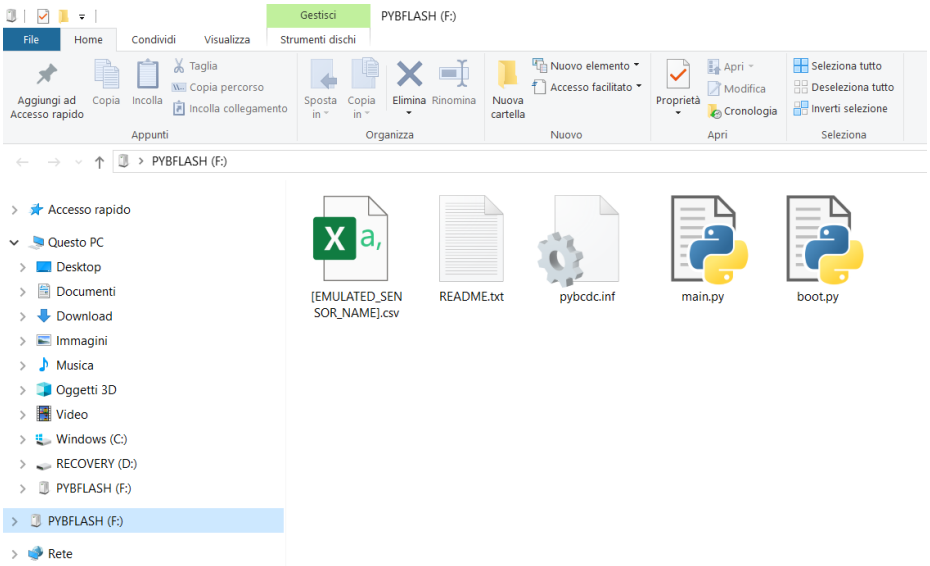


Figure 39: Pyboard filesystem layout during set-up phase

SEB MicroPython code described in section 4.5 managing the sensor emulation process running on the Pyboard is saved in main.py script, in case of standard boot process, after the execution boot.py, it will be automatically executed.
As previously mentiond .csv file storing dataset associated with emulated sensors is required to proceed with the emulation process. This file has to be stored in same folder of main.py script and will have a format similar to the one reported in Figure 40.

```
Time (UTCG),Azimuth (deg),Elevation (deg),Range (km),To Nadir Angle (deg),Temperature (degC)
20 Jan 2020 11:00:00.000,43.134,14.058,147200141.7,75.942,23
20 Jan 2020 11:01:00.000,42.566,11.309,147199878.8,78.691,23.25
20 Jan 2020 11:02:00.000,42.129,8.537,147199598.7,81.463,23.5
20 Jan 2020 11:03:00.000,41.82,5.748,147199302.5,84.252,23.75
20 Jan 2020 11:04:00.000,41.635,2.949,147198991.6,87.051,24
20 Jan 2020 11:05:00.000,41.572,0.143,147198667.3,89.857,24.25
20 Jan 2020 11:06:00.000,41.632,-2.662,147198331,92.662,24.5
20 Jan 2020 11:07:00.000,41.814,-5.463,147197984.3,95.463,24.75
20 Jan 2020 11:08:00.000,42.121,-8.253,147197628.5,98.253,25
20 Jan 2020 11:09:00.000,42.554,-11.026,147197265.4,101.026,25.25
20 Jan 2020 11:10:00.000,43.12,-13.778,147196896.4,103.778,25.5
20 Jan 2020 11:11:00.000,43.822,-16.502,147196523.2,106.502,25.75
20 Jan 2020 11:12:00.000,44.669,-19.191,147196147.5,109.191,26
20 Jan 2020 11:13:00.000,45.669,-21.838,147195770.8,111.838,26.25
20 Jan 2020 11:14:00.000,46.831,-24.434,147195394.9,114.434,26.5
20 Jan 2020 11:15:00.000,48.167,-26.97,147195021.3,116.97,26.75
20 Jan 2020 11:16:00.000,49.687,-29.437,147194651.7,119.437,27
20 Jan 2020 11:17:00.000,51.407,-31.823,147194287.8,121.823,27.25
20 Jan 2020 11:18:00.000,53.338,-34.116,147193931.1,124.116,27.5
20 Jan 2020 11:19:00.000,55.495,-36.301,147193583.3,126.301,27.75
20 Jan 2020 11:20:00.000,57.891,-38.363,147193245.7,128.363,28
```

Figure 40: csv dataset for temperature sensor test case

In this example adopted for temperature sensor test case first five columns collect data, derived from STK simulation, related to sun sensor while the sixth column collects a set of hypothesized data relating to the emulated temperature sensor.
During the first phase of the emulation process, this file is read and saved in a matrix (or in a list of lists in python language) read in the following steps.
Collecting datasets associated with different sensors that can be emulated within the same csv file is necessary in the event that during the emulation process the master reads data form more than one sensor alternately. In this case to each sensor will be associated the indexes of the columns containing the related data.

5.2 Test case emulation process

According to ECSS S-ST-10C [8] all ground support equipment (GSE) shall be verified under expected operational constraints, in addition the compatibility of the GSE interfaces with flight shall be test, in order to prevent any damage on the flight product due to ground support equipment (GSE) failure.
In order to verify SEB expected behaviour during process sensors emulation several tests have been performed on SEB processing unit represented by PyBoard. The choice to progressively tests different sections of the algorithm expected to run on PyBoard to during SEB emulation steps has been taken according to MicroPython capabilities, described in 4.3.1 section.
The capability to instantly type and execute developed code, just like you would when running Python on your pc strongly reduce time associated to code implementation and testing.
Code section test during pyboard coding phase regards:

1. testing of csv management and sensor message preparation
2. testing of communication between master and Pyboard
3. comparison of master request's response of flight sensor module and Pyboard

In this section the third step is described in detail considering as a test case the emulation of sensor module's temperature sensor analysed in section 4.4.1.
Despite the lower complexity compared to other emulated sensors, the emulation of the data read by the temperature sensor represents an asset for the verification of the satellite at the system level.
The advantages associated with the emulation of a temperature sensor are related to the difficulty in reproducing the environmental conditions encountered by the sensor in orbit during on ground FlatSat testing, and the possibility of testing FDIR algorithms, which during the operational phase use the data read by the sensors temperature to implement control laws which allow the isolation and recovery of any failures identified by reading these sensors.
Comparing the responses of the real flight sensor and the software-implemented sensor following a request from the master, it will be possible verify that no differences exist in terms of format and timing.
In particular this test offers the opportunity to verify following requirements:

FUN-02	SEB shall perform a set-up phase before each emulation process
FUN-03	SEB shall import dataset in csv format during set-up phase
FUN-04	SEB shall store imported dataset in dedicated SD card
FUN-05	SEB Pyboard shall receive 5V regulated power supply during set-up phase through USB connection with pc
FUN-06	SEB Pyboard shall allow the update of the emulation algorithm during set-up phase through USB connection with pc
FUN-08	SEB Pyboard when emulation phase starts, shall convert imported csv in list variable
FUN-09	SEB Pyboard shall increment the csv reading index each timestep defined by emulated sensor and available dataset
FUN-10	SEB shall covert data read at the updated index according to emulated sensor logic
FUN-11	SEB Pyboard shall packed the converted data according to register structure of the emulated sensor
FUN-12	SEB Pyboard shall send packed byte of data to master respecting I2c protocol contrains when a reading request is received

Figure 41: SEB's requirements verified during test case

To perform this test case two steps are considered:

- first step is represented by traditional sensor module acceptance test where in addition oscilloscope has been adopted to read data exchange on i2c bus
- second step where adopting the same set up but sensor module is replaced by SEB Pyboard

Analysing test case setup of these two steps:

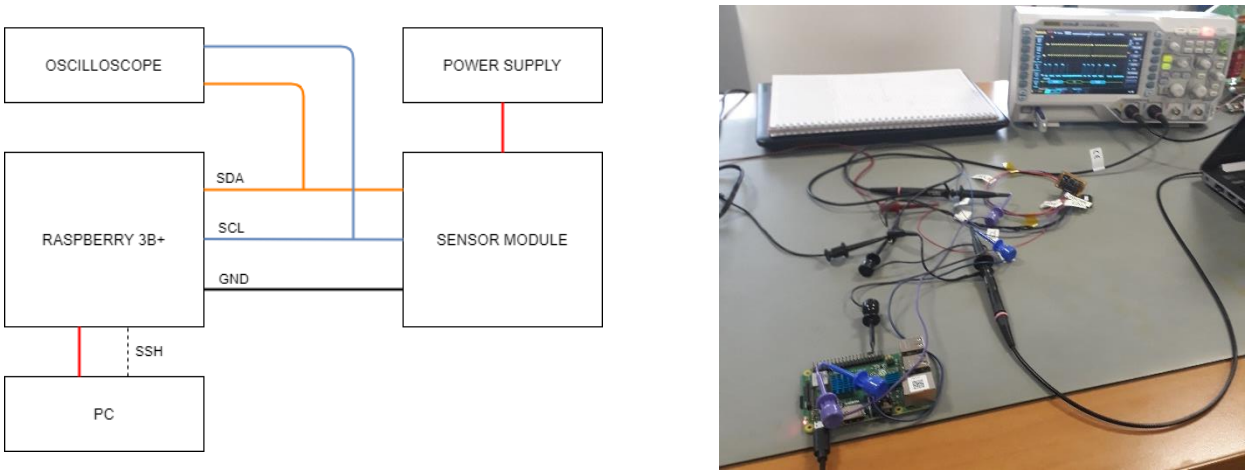


Figure 42: Sensor module test set-up

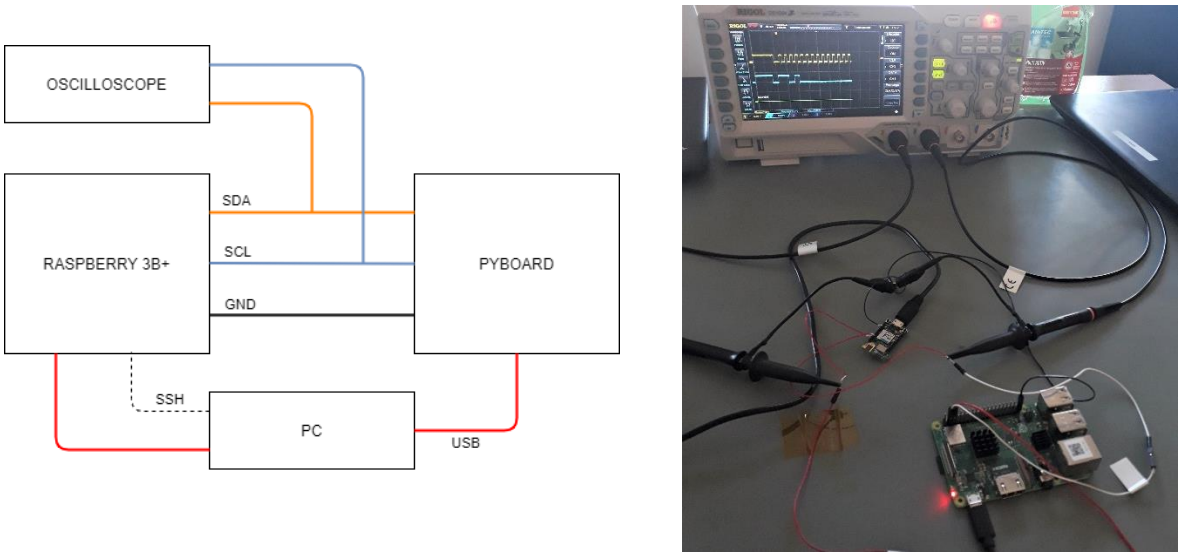


Figure 43: Pyboard temperature sensor test case set-up

In this test case Raspberry Pi 3b+ is adopted in substitution of the satellite hardware which, interfaced with sensor modules distributed in different panels of the CubeSat, is responsible of reading from these sensors and process this data for attitude determination purposes.
This configuration between Raspberry Pi 3b + and sensor module is also adopted (excepted for the oscilloscope) to test the operation of the various sensors soldered on the module before its integration in flight model. Test which will be repeated in the next steps of the verification process during HFTs.
The purpose of this test is the verification of concordance between the response of sensor module's temperature sensor and Pyboard after master reading requested, this concordance will be proved comparing oscilloscope reading carried out during these two tests.

The temperature value read during the test will be displayed bit by bit with the oscilloscope and in decimal format on the PC, which will be interfaced with the Raspberry via serial or through an ssh session, in order to perform the sensor reading request.
In **Figure 42** and 43 are shown the acquisitions made through the oscilloscope on the I2C bus between the Raspberry 3b+ acting as a master and the Pyboard acting as slave temperature sensor. As can be notice the data exchange confirms the Pyboard capability to emulate temperature acquisition and communication via I2C bus.

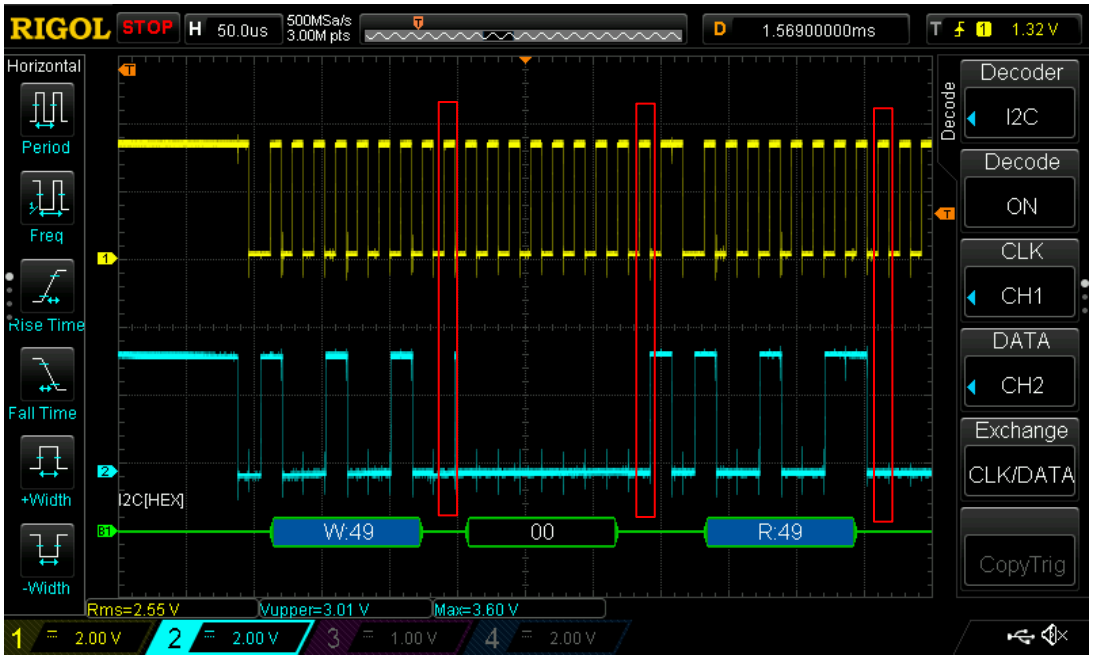


Figure 44: I2C data exchange between master and Pyboard acting as temperture sensor (1/2)

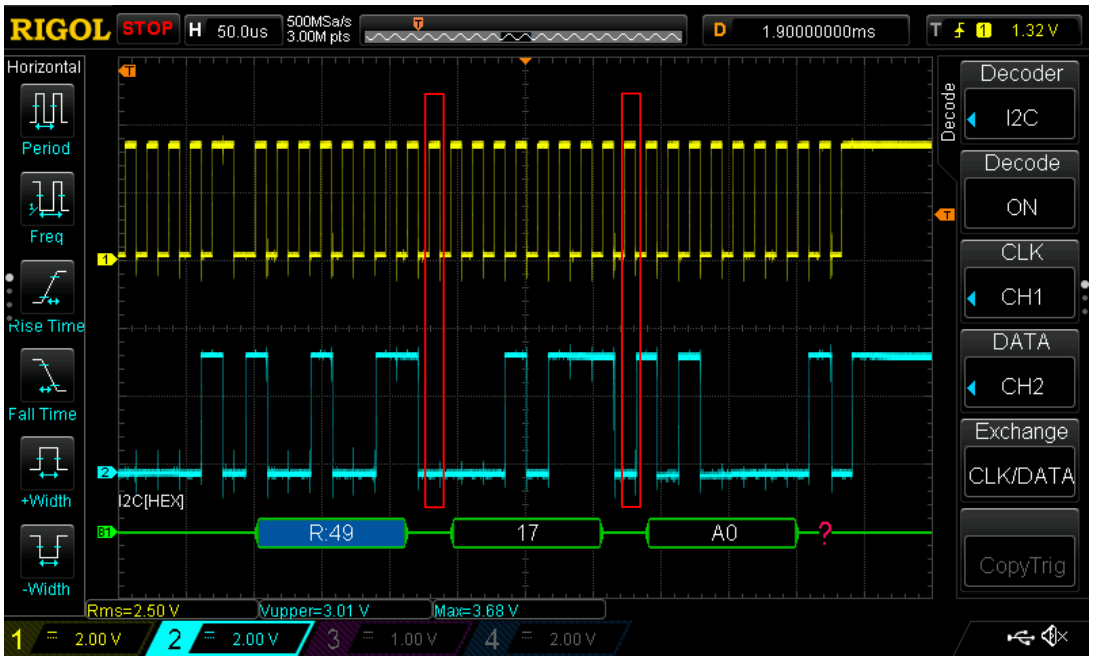


Figure 45: I2C data exchange between master and Pyboard acting as temperture sensor (2/2)

Yellow signal in channel 1 represents I2C SCL line managed by the master, while cyan signal on channel 2 is related to SDA signal.
Analysing oscilloscope acquisitions, communication between master and slave identified by exadecimal 0x49 address start by idle condition with SDA and SCL line set to 3V4 level by pull up resistor. Initialised with “start condition” the communication proceed according to I2C Bus Specifications with an 8-bit data/address format where each bit is divided by a dedicated acknowledge bit where the master release SDA line allowing Pyboard to pull the SDA line low to acknowledge (ACK) the successful transfer or leave the SDA high in case of not acknowledge (NACK).
First byte sent by Raspberry host 0x49 address in the first 7-bit followed by R/W bit set to 0 (indicating write purpose), the successful transfer is confirmed by Pyboard ACK (highlighted in red). In second byte sent by Raspberry Pi is specified the 8-bit register where master is pointing, in this case 0x00 register which indicates the read-only Temperature value Register (as described in 4.4.1), after this byte a new ACK is sent by Pyboard.
Selected the register a new start condition is sent followed by a read request to 0x49 address, ones acknowledge this third byte SDA line pass to Pyboard control which will send two bytes (divided by master ACK to confirm successfully reception):
According to Table 6 data received is decomposed in single bits extracting 10-bit temperature measurement which is re-converted in decimal format (same value read by csv dataset) and is printed on terminal.

0x17 → 00010111

0xA0 → 10100000

⟶ 0001011110 ⟶ 94 ⟶ 23.5 °C

Once sent two data bytes containing temperature info, NACK is sent by master, according to one shot reading mode, in order to interrupt the data sending from slave.

This test case has demonstrated the ability of the Pyboard to manage I2C communication via software on the slave side. Missing interruptions during the master's reading requests and the concordance between answers provided by the real sensor and subsequently by the Pyboard, confirm capability of this latter to replace the flight sensor during the testing phase allowing the reading of data released from the real test environmental conditions.
However in conclusion, considering the applications for the Sensor Emulator Board, emerges that some limitations related to the introduced solution are still unsolved.
These limitations are related to the capability of using a single device (Pyboard for example) to simulate multiple sensors simultaneously. The accomplishment of this requirement is associated to the capability on attributing multiple I2C addresses to a single device.
The possible solution identified consists in using an additional microcontroller (ATtiny 202 possible candidate) as an intermediary between the master and the Pyboard communication, exploiting capability of this new component to masking one or more address bits using the TWI Address Mask Register. Eventual advantages related to this solution will be considered during the design of second version of Sensor Emulator Board.

CONCLUSIONS

The role of the CubeSats in the international space panorama is continuously growing.
Low cost development fast and delivery capabilities characterizing this small satellite standard have allowed the transition from educational purposes to the adoption for commercial and scientific applications. The verification processes play an important role ensuring a balance between low cost, fast delivery concepts and CubeSat's reliability.
Aim of this project was the development of hardware and software solutions able to support verification and testing of CubeSats platforms ensuring greater reliability of these platforms without compromising low cost and fast delivery advantages.
The first solution developed consists of an expansion of the current resources used to control the various units of the satellite in the FlatSat on ground configuration, by enabling remote control of the Flatboard switches, which allow you to control the power lines of the satellite permitting to enable the passage between the various operating modes of the satellite encountered in the early stages after launch.
The enabling of the remote control of these switches has the aim of allowing an automation of the test process both related to single modules of the satellite and for the satellite in flight configuration.
Advantages associated to the implementation of this solution have been verified during the test case, where the automation of the cold boot test process of the satellite CDH has lead to a reduction of the time required to perform the test over 60%.
This test case implementation although the successful result has allowed to identified some criticalities related to FlatBoard hardware configuration which require a board re-design in order to accomplish the fully operability of this switch remote control.
The second solution consists on a dedicated board which replacing inoperative hardware during FlatSat tests allows the emulation of their behaviour in terms of data transmission and power consumption, A first preliminary design has been performed in order to identify critical components and define a physical architecture of the system.
The implemented test case has demonstrated the SEB capability through the Pyboard to manage I2C communication emulating via software the slave behaviour, however, this design has lead to identify limitation related to the capability of emulate multiple sensors simultaneously.

REFERENCE:

[1] CubeSat Design Specification Rev. 13 The CubeSat Program, Cal Poly SLO

[2] Nanosats Database: https://www.nanosats.eu/

[3] CubeSats Lecture, NASA JPL: https://www.youtube.com/watch?v=ZHxIgGYJojw&t=592s

[4] “Small-Satellite Mission Failure Rates”, Stephen A. Jacklin NASA Ames Research Center, Moffett Field, CA

[5] CGEE database: https://www.cgee.org.br/web/observatorio-espacial/bancos-de-dados

[6] ECSS S-ST-00C - Description, implementation and general requirements

[7] ECCSS-E-ST-10-02C - Verification

[8] ECCSS-E-ST-10-03C – Testing

[9] Tailored ECSS Engineering Standards for In-Orbit Demonstration CubeSat Project

[10] Raspberry Pi Compute Module 3 (CM3) datasheet

[11] DWA-121 Wireless N 150 Pico USB Adapter datasheet

[12] TCA9539 Texas Instruments GPIO expander datasheet

[13] Micropython homepage: http://micropython.org/

[14] AD7415 temperature sensor datasheet

[15] 3-Axis Digital magnetometer HMC5883L datasheet

[16] Integrated Solar Angle Sensor E910.86 datasheet

[17] “MicroPython Documentation - Release 1.9.4”, Damien P. George, Paul Sokolovsky

[18] Pyboard D SFxW schematic

[19] “Understanding the I2C Bus” – Texas Instruments

[20] “Firmware emulation of an I2C Slave device”, Steve Kolokowsy, Cypress Semiconductor

ANNEX

ANNEX 1 – dtblob.dts

```
dts-v1/;

/ {
 videocore {

   pins_cm {

    pin_config {

      pin@default {
       polarity = "active_high";
       termination = "pull_down";
       startup_state = "inactive";
       function = "input";
      }; // pin

      // BANK 0 - USER GPIO //
      pin@p0  { function = "input";   termination = "pull_up";    }; // DEFAULT STATE
      pin@p1  { function = "input";   termination = "pull_up";    }; // DEFAULT STATE
      pin@p2  { function = "input";   termination = "pull_up";    }; // DEFAULT STATE
      pin@p3  { function = "input";   termination = "pull_up";    }; // DEFAULT STATE
      pin@p4  { function = "input";   termination = "pull_up";    }; // DEFAULT STATE
      pin@p5  { function = "input";   termination = "pull_up";    }; // DEFAULT STATE
      pin@p6  { function = "input";   termination = "pull_up";    }; // DEFAULT STATE
      pin@p7  { function = "input";   termination = "pull_up";    }; // DEFAULT STATE
      pin@p8  { function = "input";   termination = "pull_up";    }; // DEFAULT STATE
      pin@p9  { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p10 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p11 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p12 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p13 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
//////////////////////////////////////////
// TO ENABLE UART0 UNCOMMENT THESE 2 LINES:
//      pin@p14 { function = "uart0";   termination = "no_pulling"; }; // UART0 TX
//      pin@p15 { function = "uart0";   termination = "pull_up";    }; // UART0 RX
// AND COMMENT OUT/REMOVE THESE 2 LINES:
      pin@p14 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p15 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
//////////////////////////////////////////
      pin@p16 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p17 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p18 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p19 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p20 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p21 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p22 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p23 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p24 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p25 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
```

```
      pin@p26 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p27 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE

      // BANK 1 - USER GPIO//
      pin@p28 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE WAS
INPUT NO PULL
      pin@p29 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE WAS
INPUT NO PULL
      pin@p30 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p31 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p32 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p33 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p34 { function = "input";   termination = "pull_up";    }; // DEFAULT STATE
      pin@p35 { function = "input";   termination = "pull_up";    }; // DEFAULT STATE
      pin@p36 { function = "input";   termination = "pull_up";    }; // DEFAULT STATE
      pin@p37 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p38 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p39 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p40 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p41 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p42 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p43 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p44 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE WAS
INPUT NO PULL
      pin@p45 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE WAS
INPUT NO PULL

      // BANK 2 - DON'T TOUCH UNLESS YOU KNOW WHAT YOU'RE DOING //
      pin@p46 { function = "input";   termination = "no_pulling"; drive_strength_mA = <8>;
polarity = "active_high"; }; // HPD_N
      pin@p47 { function = "output";  termination = "no_pulling"; drive_strength_mA = <8>;
polarity = "active_low"; startup_state = "active"; }; // STATUS LED / EMMC_DISABLE_N
CONTROL
      pin@p48 { function = "sdcard";  termination = "pull_up";    drive_strength_mA = <8>; }; //
SD CLK
      pin@p49 { function = "sdcard";  termination = "pull_up";    drive_strength_mA = <8>; }; //
SD CMD
      pin@p50 { function = "sdcard";  termination = "pull_up";    drive_strength_mA = <8>; }; //
SD D0
      pin@p51 { function = "sdcard";  termination = "pull_up";    drive_strength_mA = <8>; }; //
SD D1
      pin@p52 { function = "sdcard";  termination = "pull_up";    drive_strength_mA = <8>; }; //
SD D2
      pin@p53 { function = "sdcard";  termination = "pull_up";    drive_strength_mA = <8>; }; //
SD D3

    }; // pin_config

    pin_defines {
      pin_define@HDMI_CONTROL_ATTACHED { type = "internal"; number = <46>; }; // HPD_N
on GPIO46
    }; // pin_defines

  }; // pins_cm
```

```
   pins_cm3 {

    pin_config {

      pin@default {
       polarity = "active_high";
       termination = "pull_down";
       startup_state = "inactive";
       function = "input";
      }; // pin

      // BANK 0 - USER GPIO //
      pin@p0  { function = "input";  termination = "pull_up";   }; // DEFAULT STATE
      pin@p1  { function = "input";  termination = "pull_up";   }; // DEFAULT STATE
      pin@p2  { function = "input";  termination = "pull_up";   }; // DEFAULT STATE
      pin@p3  { function = "input";  termination = "pull_up";   }; // DEFAULT STATE
      pin@p4  { function = "input";  termination = "pull_up";   }; // DEFAULT STATE
      pin@p5  { function = "input";  termination = "pull_up";   }; // DEFAULT STATE
      pin@p6  { function = "input";  termination = "pull_up";   }; // DEFAULT STATE
      pin@p7  { function = "input";  termination = "pull_up";   }; // DEFAULT STATE
      pin@p8  { function = "input";  termination = "pull_up";   }; // DEFAULT STATE
      pin@p9  { function = "input";  termination = "pull_down"; }; // DEFAULT STATE
      pin@p10 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE
      pin@p11 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE
      pin@p12 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE
      pin@p13 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE
////////////////////////////////////////
// TO ENABLE UART0 UNCOMMENT THESE 2 LINES:
//      pin@p14 { function = "uart0";  termination = "no_pulling"; }; // UART0 TX
//      pin@p15 { function = "uart0";  termination = "pull_up";   }; // UART0 RX
// AND COMMENT OUT/REMOVE THESE 2 LINES:
      pin@p14 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE
      pin@p15 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE
////////////////////////////////////////
      pin@p16 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE
      pin@p17 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE
      pin@p18 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE
      pin@p19 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE
      pin@p20 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE
      pin@p21 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE
      pin@p22 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE
      pin@p23 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE
      pin@p24 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE
      pin@p25 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE
      pin@p26 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE
      pin@p27 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE

      // BANK 1 - USER GPIO//
      pin@p28 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE WAS
INPUT NO PULL
      pin@p29 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE WAS
INPUT NO PULL
      pin@p30 { function = "input";  termination = "pull_down"; }; // DEFAULT STATE
```

```
      pin@p31 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p32 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p33 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p34 { function = "input";   termination = "pull_up";    }; // DEFAULT STATE
      pin@p35 { function = "input";   termination = "pull_up";    }; // DEFAULT STATE
      pin@p36 { function = "input";   termination = "pull_up";    }; // DEFAULT STATE
      pin@p37 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p38 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p39 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p40 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p41 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p42 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p43 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE
      pin@p44 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE WAS
INPUT NO PULL
      pin@p45 { function = "input";   termination = "pull_down";  }; // DEFAULT STATE WAS
INPUT NO PULL

      // BANK 2 - DON'T TOUCH UNLESS YOU KNOW WHAT YOU'RE DOING //
      pin@p46 { function = "input";   termination = "pull_up";    }; // SMPS_SCL
      pin@p47 { function = "input";   termination = "pull_up";    }; // SMPS_SDA
      pin@p48 { function = "sdcard";  termination = "pull_up";    drive_strength_mA = <8>; }; //
SD CLK
      pin@p49 { function = "sdcard";  termination = "pull_up";    drive_strength_mA = <8>; }; //
SD CMD
      pin@p50 { function = "sdcard";  termination = "pull_up";    drive_strength_mA = <8>; }; //
SD D0
      pin@p51 { function = "sdcard";  termination = "pull_up";    drive_strength_mA = <8>; }; //
SD D1
      pin@p52 { function = "sdcard";  termination = "pull_up";    drive_strength_mA = <8>; }; //
SD D2
      pin@p53 { function = "sdcard";  termination = "pull_up";    drive_strength_mA = <8>; }; //
SD D3

      pin@p128 { function = "input";  termination = "no_pulling"; polarity = "active_low"; }; //
Hotplug
      pin@p129 { function = "output"; termination = "no_pulling"; polarity = "active_low"; }; //
EMMC_ENABLE_N
    }; // pin_config

    pin_defines {
      pin_define@HDMI_CONTROL_ATTACHED { type = "external"; number = <0>; };
      pin_define@EMMC_ENABLE { type = "external"; number = <1>; };
      pin_define@SMPS_SDA { type = "internal"; number = <46>; };
      pin_define@SMPS_SCL { type = "internal"; number = <47>; };
    }; // pin_defines

  }; // pins_cm3

 }; // videocore

};
```

ANNEX 2 – Function-component matrices

Subsystem level:

	Data emulation subsystem	External interface subsystem	Power consumption emulation subsystem
To manage data exchange with vehicle	X		
To manage external interfaces		X	
To emulate power consumption			X

Segment level:

	Pyboard interface unit	PyBoard processing unit	Power interface unit	Vehicle interface unit	Active Load unit
To import dataset during setup	X				
To exchange data with vehicle		X			
To manage power supply			X		
To provide data interface to vehicle				X	
To dissipate power					X
To regulate power dissipation					X

Component level:

	micro SD card	PyBoard's micro USB connector	STM32F722IEK microcontroller	Power socket	LDO regulator	Vehicle connector
To store imported data	X					
To interface with pc		X				
To receive regulated power during setup		X				
To read commands received from vehicle			X			
To read imported data			X			
To convert imported data			X			
To send data to vehicle			X			
To receive power from external source				X		
To regulate received power					X	
To receive power from vehicle						X
To provide power to vehicle (3.3V)						X
To provide I2C interface						X
To provide SPI interface						X