
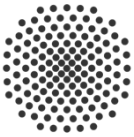

Scuola d'ingegneria industriale e dell'informazione

Tesi di Laurea Magistrale in Space Engineering

Active thermal control for a balloon-borne telescope

Author: Marco Modé

Politecnico di Milano - Advisor: Michèle Lavagna

Universität Stuttgart - Advisor: Mahsa Taheran

21 December 2021

Student ID: 944437

Acknowledgements

It has been a hard journey. I did a lot of sacrifices to reach this achievement, which is the crowning of all the work I did in my life until here. I want to dedicate this first page to the people which contributed to this goal and helped me to make it happen.
First and foremost, I would like to thank my family for giving me this opportunity and for the huge support along these five years. You helped me a lot to keep up with my goals and you backed me up during the hard moments. I would not be here without all the efforts you put on me.
I would like to thank my friends of Cherif which I consider as my family too. We shared so many things together and I'm enthusiastic of sharing with you also this moment. Whenever I needed you, you were always by my side.
I would like to thank the person which shared with me this journey, from the beginning to the end. We went through all the happiest and hardest moments together. You made me become a better person, I hope the best for you.
As last I would like to thank my supervisors for guiding me during this work. In particular I would like to thank Mahsa, which has been a great teacher and friend to me.

È stato un viaggio duro. Ho fatto tanti sacrifici per raggiungere questo traguardo, che è il coronamento di tutto il lavoro che ho fatto nella mia vita fino a qui. Voglio dedicare questa prima pagina alle persone che hanno contribuito a questo obiettivo e mi hanno aiutato a realizzarlo.
Innanzitutto vorrei ringraziare la mia famiglia, per avermi dato questa opportunità e per l'enorme supporto durante questi cinque anni. Mi avete aiutato molto a tenere il passo con i miei obiettivi e mi avete sostenuto nei momenti difficili. Non sarei qui senza tutti gli sforzi che avete speso per me.
Vorrei ringraziare i miei amici di Cherif che considero come una seconda famiglia. Abbiamo condiviso tante cose insieme e sono felicissimo di condividere con voi anche questo momento. Ogni volta che ho avuto bisogno di voi, eravate sempre al mio fianco.
Vorrei ringraziare la persona che ha condiviso con me questo viaggio, dall'inizio alla fine. Abbiamo vissuto insieme tutti i momenti più felici e quelli più difficili. Mi hai fatto diventare una persona migliore, spero il meglio per te.
Infine vorrei ringraziare i miei supervisori per avermi guidato durante questo lavoro. In particolare vorrei ringraziare Mahsa, che è stata per me una grande insegnante e amica.

Abstract

The aim of this dissertation is to design, implement, integrate and test the thermal controller software, for the STUDIO payload, in the context of the stratospheric astronomy mission ESBO-DS (European Stratospheric Balloon Observatory - Design Study) [1]. It is a project, funded under the European Union's Horizon 2020 programme, that paves the way for an astronomical observatory infrastructure based on stratospheric balloons. A central part of the project is the development of the flightworthy prototype STUDIO, which is scheduled to fly in 2022. STUDIO payload would be working at around 40 km altitude and therefore it is necessary to have a precise and reliable thermal control system.
The software is required to be developed on Linux Operating System in an object-oriented manner, integrated with the Flight Software Framework. It is a component-based framework, developed by IRS (Institute of Space Systems) [2], which contains useful and re-usable core components and functionalities for the software components to build.
The thermal analysis has been completed in 2019 but the thermal simulation is still to be finalized. Therefore, the equipment currently selected in terms of sensors and heaters will probably change in number and placement. For this reason, the software is designed to be flexible. It allows to delete or add components easily and in any moment.
This work is split in two parts: the implementation of the device handler component, for the communication with physical devices, and the implementation of the thermal controller component. These two main classes are developed and tested independently. Then they are integrated and tested as a whole system. The software on the Arduino sensor board has been fixed and improved within this project It was not meant to be an objective of this work but it revealed to be a bottleneck for the proceeding of the other components tests.
The final tests for the validation of the system are not done yet. The following operations shall be concluded before these tests: the finalization of the thermal simulation, the development of the firmware for the power switcher communication interface, the set-up of the ground system and STUDIO telescope in IRS clean room.

Abstract (Italian version)

Lo scopo di questa tesi è progettare, implementare, integrare e testare il software di controllo termico, per il payload di STUDIO, nell'ambito della missione spaziale ESBO-DS (European Stratospheric Balloon Observatory - Design Study) [1]. Si tratta di un progetto, finanziato nell'ambito del programma Horizon 2020 dell'Unione Europea, che apre la strada a un'infrastruttura di osservazione astronomica basata su palloni stratosferici. Una parte centrale del progetto è lo sviluppo del prototipo qualificato per il volo STUDIO, che dovrebbe volare nel 2022. Il carico utile di STUDIO lavorerebbe a circa 40 km di altitudine e quindi è necessario disporre di un sistema di controllo termico preciso e affidabile.
Il software deve essere sviluppato su sistema operativo Linux attraverso la programmazione orientata agli oggetti, all'interno del FSFW (Flight Software Framework). È un framework basato su componenti, sviluppato da IRS (Istituto di Sistemi Spaziali) [2], che contiene modelli utili e riutilizzabili per i componenti software da costruire.
L'analisi termica è stata completata nel 2019 ma la simulazione termica deve ancora essere finalizzata. Pertanto, le apparecchiature attualmente selezionate in termini di sensori e riscaldatori, probabilmente cambieranno in numero e posizionamento. Per questo motivo, il software è progettato per essere flessibile. Consente di eliminare o aggiungere componenti facilmente e in qualsiasi momento.
Questo lavoro è diviso in due parti: l'implementazione del componente per la comunicazione con i dispositivi fisici (device handler) e l'implementazione del componente di controllo (thermal controller). Queste due classi principali sono sviluppate e testate in modo indipendente. In seguito sono integrati e testati come un intero sistema. Il software responsabile della scheda Arduino Micro per la gestione dei sensori è stato corretto e perfezionato all'interno di questo progetto. Non sarebbe dovuto essere un obiettivo di questo lavoro ma si è rivelato un collo di bottiglia per il proseguimento dei test degli altri componenti.
I test finali per la validazione del sistema non sono ancora stati effettuati. Prima di questi test devono essere concluse le seguenti operazioni: la finalizzazione della simulazione termica, lo sviluppo del firmware per l'interfaccia di comunicazione del commutatore di potenza (power switcher), la configurazione del sistema di terra e del telescopio STUDIO nella camera bianca dell'instituto IRS.

Contents

Nomenclature

Acronyms

ESBO-DS European Stratospheric Balloon Observatory - Design Study

FDIR Fault Detection, Isolation, and Recovery

FSFW Flight Software Framework

FW Filter Wheel

ID Identity Document

IDE Integrated Development Environment

IF InterFace

IRS Institut fur Raumfahrt-Systeme (Institute of Space Systems)

OO Object-Oriented

OOP Object-Oriented Programming

OS Operating System

PIFH Polyimide Insulated Flexible Heater

RTOS Real Time Operating System

SOA Service Oriented Architecture

SPC STUDIO Payload Computer

STUDIO Stratospheric Ultraviolet Demonstrator of an Imaging Observatory

TBD To Be Decided

TIP Threat Intelligence Platform

TMB Temperature Measurement Board

UV Ultra-Violet

1 Introduction

1.1 ESBO-DS

ESBO-DS stands for European Stratospheric Balloon Observatory - Design Study. It is a Research Infrastructure project, funded under the European Union's Horizon 2020 programme, that paves the way for an astronomical observatory infrastructure based on stratospheric balloons[1].
The astronomical observations are today divided into three areas:

- ground-based astronomy with the help of stationary telescopes;
- air-supported astronomy with airplanes or balloons;
- orbital astronomy in the form of space telescopes.

Ground-based astronomy enables larger and more light-sensitive optics, but it is affected by limits due to Earth's atmosphere. On the other side, spacecrafts provide access to optimal observing conditions but they are less flexible and more expensive compared to air-supported astronomy. The airborne option is a trade-off between the previous: the anomalies caused by atmosphere are not completely eliminated but it is convenient for some astronomical applications, in particular in the far infrared wavelength region and in some parts of the ultraviolet.
Formerly, in 2016, ORISON started a feasibility study for a balloon-based research platform [3]. Consequently, this work has been pursued by ESBO-DS project which has the objective of creating a European research infrastructure to fly ballooing-based telescopes to altitudes of 30 to 40 km with regular flights, exchangeable instruments, and open access to observation time.
ESBO-DS is carried out by a consortium of five European partners from the scientific community and industry:

- The Institute of Space Systems (IRS) at the University of Stuttgart, Germany [2];
- The Swedish Space Corporation [4];
- The Institute for Astronomy and Astrophysics at the University of Tübingen, Germany [5];
- The Max Planck Institute for extraterrestrial Physics, Germany [6];
- The Instituto de Astrofísica de Andalucía, Spain [7].

In particular, the work performed in this thesis has been carried out at IRS in Stuttgart.

1.1.1 STUDIO

In 2018, the construction of the flightworthy prototype STUDIO (STratospheric Ultraviolet Demonstrator of an Imaging Observatory) begun. It is a central part of the ESBO-DS project.
The main components of STUDIO, represented in Fig. 1, are the helium balloon, the gondola, the scientific payload (a telescope with 50 cm aperture and instruments for the ultraviolet and visible spectral ranges) and the peripherals required for operations. The task of STUDIO is the search for compact stars, in other words the remains of former stars in which nuclear fusion no longer takes place, as well as possible radiation bursts from these. The prototype is scheduled to fly in 2022. It will allow testing of critical technologies and the deliver of the firsts scientific observations.

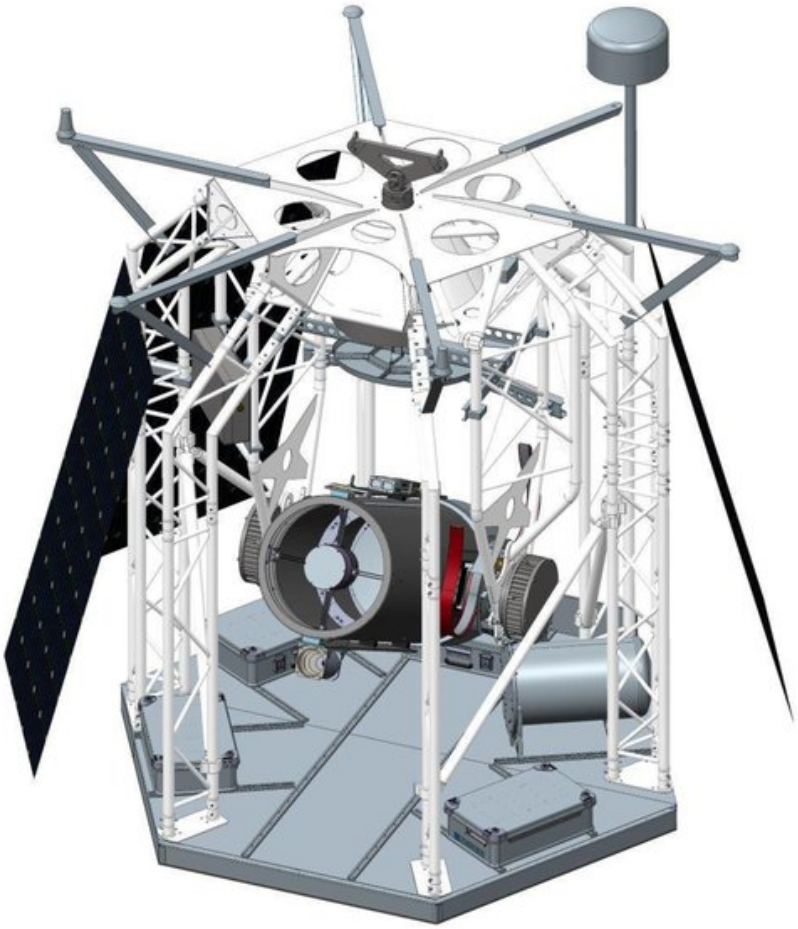

Figure 1: STUDIO gondola. One solar panel is not shown for better visibility of the payload. Most electronics and service systems are located on the gondola floor. [8]

STUDIO payload would be working at around 40 km altitude. It will be exposed to extreme environmental conditions during the ascent and at its operating altitude. On the one hand, the ambient temperature fluctuates during the ascent and reaches lows of -60 °C, and on the other hand, the solar radiation heats up to +85 °C. Combined

with an almost complete vacuum, this puts a strain on the structure and the scientific instruments. In order to minimize these and other problems, the temperature of the various instruments must be continuously monitored and, if necessary, controlled.

1.2 Software development

The software development in this project is carried out in C++ language in the context of the Flight Software Framework [9] built by IRS. It will be exploited for the development of the whole on-board software of STUDIO. The programming has been done in Linux Ubuntu operating system.
In Sec. 1.2.1, a brief summary of software engineering methods and techniques applied, is given. Than the Flight Software Framework is presented in Sec. 1.2.2.

1.2.1 Software Engineering tools and methods

Two main software engineering subtopics are here discussed:

- development methods, dealing with the organization of the software development process;
- design and implementation techniques, dealing with the application of programming languages and paradigms to efficiently solve a certain problem.

The focus is mainly on the second point. The objective is to identify techniques to allow a better development of software in terms of time, quality and efficiency. For this reason, principles like decomposition and abstraction, as well as methods to allow code reuse, are introduced.
The evolution of implementation techniques is twofold. Firstly, there are programming languages and language features which are intended to improve efficiency and code reuse. Secondly, OOP techniques allow to construct large software from existing standardized parts. This technique of developing software applications by combining pre-existing and new components is called component-oriented programming . In parallel, it was noticed that re-usability does not happen automatically when using OOP techniques. Instead, classes and objects must be designed for reuse, which leads to software frameworks development. Any software component by itself is worth nothing without a dedicated framework supporting this component.

1.2.1.1 Object-Oriented Programming This concept is the basis of the programming paradigm applied in this project. The idea of object-orientation is to merge data structures, called attributes, and functions, called methods, in a single logical unit which is called object. Some relevant features are described in the terminology of the C++ programming language.
It is important to create objects with a clear physical meaning in order to simplify the understanding. For example, an object-oriented embedded controller may consists of the following objects: sensor objects, controller objects (they implement the controller algorithms) and actuator objects. In this way, the data and functionalities of these

physical objects are well distinguished.
A class is the definition of a certain type of object. An object is the result of the concrete instantiation of a given class. Multiple instantiations may be done exploiting a parametrized constructor.
A central concept of object orientation is encapsulation of data and functionality. It enables the separation of private and public elements of a class. In this way, some sources of complexity are hidden and some typical programming errors, as coupling of classes or unwanted access to inner workings, are avoided.
Another main feature of OOP is inheritance. A child class inherits from the parent class all its attributes and methods. These could be possibly extended and refined by overriding. This is a main driver for code reuse.
Dynamic dispatch ensures that an external caller actually executes the child class code even if it has a reference of the parent's class type only. It is a technique that allows an object to select the code of a method at run-time.
In C++ the interfaces shall be implemented, as distinguished objects, using abstract classes with pure virtual functions. Interfaces are a more powerful technique for code decoupling than simple encapsulation. Firstly, an object implementing different interfaces provides specific views to different callers, only exposing relevant information for each. In addition, a caller can access different objects with the same interface uniformly.

1.2.1.2 Components As mentioned above, instead of implementing certain features over and over again, a company with domain expertise sells a ready-to-use piece of software. Developers assemble a number of such pieces to form applications, which are supposed to be built faster and more reliable, as components are already checked and tested. It is important to stick to the objective of maintaining the maximum generality possible but without increasing the complexity. Components require a certain environment for the execution, be it an entire operating system or a plug-in-capable application. This environment may be viewed as a framework for component execution.

1.2.1.3 Frameworks Firstly the common interfaces, called protocols, and algorithms of a given program are individuated. Then, good abstractions of them are created in order to form a framework to develop similar applications. This has the important implication that frameworks are always intended for a certain domain, they are not general-purpose. Frameworks typically evolve in the following steps:

- white-box framework, it can be used to develop similar applications, but it requires a deep understanding of the inner workings of the framework;
- black-box framework, it hides the inner workings of the framework from the application developer. One variant to do so is by plugging software elements or plug-ins, into existing framework containers. Thus, the programmer only needs to know the required plug-in interfaces to create a customized software.

The use of the framework is based on the concept of inheritance. Unfortunately, this requires a rather deep knowledge of the base class and the circumstances under which the subclass is called. It needs to describe the techniques for components to communicate. There are two fundamental ways of communication: synchronous and asynchronous. All communication between components in a real-time system must be asynchronous. There are two main techniques to do so [9]:

- message queues, the caller puts its request in a message, which is read and handled by the receiving component when appropriate;
- shared memory, a component writes information to a shared memory region, where it is read out by the receiver at another point in time.

1.2.1.4 Abstraction and Generalisation All of the aforementioned techniques promote some form of abstraction or generalisation. Abstraction in computer science means to hide currently unneeded details of the abstract element to the user. A named function or subroutine is an abstraction. Generalization means to group similar concepts and form a single entity capable of handling these cases, eventually by providing parameters. Both concepts are essential to manage complex programs and are often used in conjunction. Abstraction helps programmers to focus on the essentials, generalisation avoids duplicates. Both have a certain cost, therefore they must be employed carefully.

1.2.2 The Flight Software Framework

The work done at IRS on the FSFW and the choice to implement it in the STUDIO on-board computer, are motivated by the capability of the framework to reduce dramatically the costs, even with the constant increase of complexity.
The FSFW aims for a component-based software architecture. A well-defined set of components interact by making use of a lightweight component framework, the FSFW-Core, which ensures real-time capable information exchange between components. The FSFW introduces abstraction layers in order to allow portability of the software to different environments. Moreover, to reduce coupling between components, a set of common interfaces both for inter-component and ground communication is defined. The architecture of the FSFW is represented in Fig. 2.

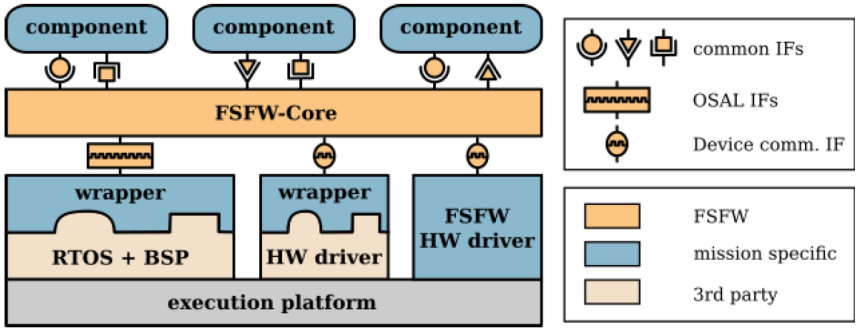

Figure 2: FSFW architecture. [9]

1.2.2.1 FSFW Components The FSFW employs four different classes of components [9]:

- Device handler components, they handle the communication between computer and physical components;
- Controller components, they perform the control algorithms;
- Subsystem components, including assemblies, they represent the engineering concept of a spacecraft subsystem and a redundant set of equipment;
- Ground service components, they provide specific functionalities for ground interaction.

The FSFW supplies component templates for these four classes containing the functionalities selected after a work of domain analysis, generalisation and abstraction. The component templates are implemented as abstract base classes.

1.2.2.2 FSFW Interfaces The FSFW defines interfaces for each functionality and ensures access is possible using the message-based software bus of the FSFW-Core. Every component is an object, the FSFW provides interface definitions in the form of standard OO interfaces. The recurring features of these interfaces are [9]:

- Actions, they are sporadic, finite and externally triggered activities for device handler components and for commanding from ground;
- Modes, they define the permanent behaviour of subsystems and equipment. A common interface to read and set the component mode is provided;
- Health, the interface allows to read and modify the health state;
- Parameters, the interface simplifies the reading and adjusting of on-bard parameters;
- Memory, the interface unifies the way in which device handler components make local and global memory accessible.

1.2.2.3 FSFW Core The Core of the framework support the execution and allow the interaction between components. The elements of the core layer are represented in Fig. 3.

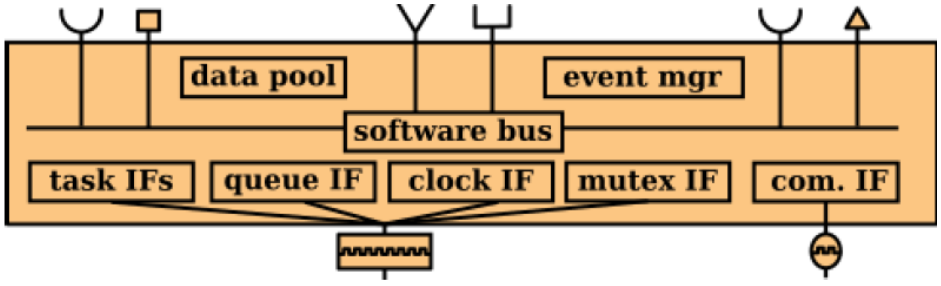

Figure 3: FSFW core. [9]

It delivers the following functionalities [9]:

- Communication: it is a major task of the core. Three methods of asynchronous information exchange are provided in order to ensure real-time execution of components: a message-based interface through a software bus, the possibility to distribute events, a data pool for the exchange of periodic data.
- Execution: the FSFW Core provides software elements and interfaces to schedule components, either on a periodic basis or in fixed time slots.
- Clocks and Timer: a common clock source in different formats along with facilities to manage and set the on-board time are delivered. Timers to measure time intervals are provided too.
- Data containers: the Core provides containers to store and modify run-time adjustable data of components.

The FSFW-Core makes use of the underlying real-time operating system (RTOS) for most of these functionalities. As mentioned before, in order to avoid hardware dependency, functionality is accessed via interfaces only.

1.3 Thesis objectives and motivation

In Sec. 1.1.1, it is stated the necessity of a thermal controller for the STUDIO payload. The objective of this thesis is to design, implement, integrate and test the thermal controller software components. This includes a central controller and software components to collect temperature sensor values and command heaters correctly, including fault detection of heaters. The software is required to be developed in the Flight Software Framework which is introduced and described above in Sec. 1.2.
Since 2019, detailed thermal analysis has been done, and based on that, the overall thermal design is decided upon. While many components are protected with passive thermal controller, there are also several components that require active heating using heaters. The number and type of heaters are determined. The list and placement of the temperature sensors are also determined and given. It is highlighted that sensor and heaters number and placement is not finalized yet. Consequently, it is a specific requirement for the thermal controller software to be easily configurable in terms of parameters and components.
The sensors are handled by an Arduino board which has already been programmed in Arduino IDE. It is necessary to define the communication between this sensor board and the FSFW. Similarly, the communication between the framework and the heater board should set-up.

1.4 Outline

The work in this project is structured as following:

- Chapter 2 shows the results of the thermal analysis. The equipment chosen, regarding sensors and heaters, are introduced and described. Following, the model of the thermal controller software is presented along with the introduction of the main components.
- Chapter 3 presents the work done for the development of the device handler components and its implementation in the FSFW. In this part, it is set-up the communication between the sensor and heaters board with the framework.
- In Chapter 4, the controller component is finalized. Firstly, the existing building blocks provided by the framework are analyzed. Then, the whole process of implementation of the controller code is described. A critical aspect is the integration of the power switcher component.
- Chapter 5 reports the result of integration of all the software parts and testing with the real hardware. The tests planned and executed are illustrated and the results, along with the faced criticalities, are shown.
- Chapter 6 concludes the thesis with final considerations about the work done. The main criticalities are resumed and, lastly, personal opinions on the next steps to face are given.

2 Thermal Controller design

As mentioned in Sec. 1.3, the thermal analysis of the STUDIO prototype has already been completed in 2019 [10]. The results are shown hereafter. The thermal simulation, instead, is still to be finalized. Below, the physical components chosen for the thermal control system and the control logic selected are described.
In the second part of the chapter, the software architecture of the thermal controller is introduced. The development of the main software components is then addressed to the next chapters.

2.1 Thermal analysis results

The thermal analysis has been carried out through the software ESATAN-TMS [10][11]. Firstly, the temperature ranges required by the different equipment of the prototype are displayed in Table 1. Furthermore, the environment thermal loads are reported. It is highlighted that the limit values for the loads are identical on different parts of the prototype.

Table 1: Components temperature requirements[1][10]

		Temperature range allowed [$°C$]			
		Operational		**Non-operational**	
Components		**min**	**max**	**min**	**max**
Telescope					
	Optical Tube Assembly	−23	30	−50	40
	M2 Focusing Mechanism	−23	30	−50	40
TIP Platform					
	M3 Tip/Tilt Platform	−20	80	TBD	TBD
Platform Baseplate					
	UV Filter Wheel	−10	50	–	–
	UV Detector Assembly	TBD	55	−40	55
	UV Proximity Electronics	−44	75	−44	75
	Visible Filter Wheel	−10	50	–	–
PCO edge		10	40	−10	60
VIS Electronics		−60	TBD	TBD	TBD
Environment loads		**min**	**max**	**min**	**max**
	DST Pointing System	−40	85	−40	85
	SSC Battery box (x2)	−40	85	−40	85
	SSC Battery box + iridium	−40	85	−40	85
	Avionics	−40	85	−40	85

Four different cases have been simulated: two hot cases in which all power settings are in nominal operative condition and two cold cases, in which all power consumers are off. For each of the two categories, an analysis case, both for launch at sunset and sunrise, has been simulated. The problem with ESATAN is that it is not capable of turning on or off any power for a specific time step. So, the heating cases are a conservative upper estimate and the two non-heating cases are a lower estimate. So, in the two hot cases, all the nominal thermal loads and heaters are set to "on". On the other side, all the heaters and thermal loads are set to "off" in the two cold cases.
The results of the thermal analysis are shown in Table 2. After the simulation three critical components are identified: the telescope tube, the M1 (primary) mirror, which has same requirements on temperature range of the telescope tube, and the M2 mirror. The table represents the minimum and maximum temperatures that they reach, in the cases in which it is critical in terms of temperature limits. It is highlighted that the temperatures obtained in the simulation must be compared to the operational ranges in Table 1.

Table 2: Thermal analysis results [10]

Case	Telescope tube	M1 mirror	M2 mirror
Sunrise no heating			
Min [$°C$]	-41	-	-
Max [$°C$]	-13	-	-
Sunset no heating			
Min [$°C$]	-42	-	-
Max [$°C$]	-12	-	-
Sunrise heating			
Min [$°C$]	-23	-20	-20
Max [$°C$]	13	-2	-6
Sunset heating			
Min [$°C$]	-30	-	-
Max [$°C$]	14	-	-

The telescope tube exceeds the operational ranges in all analysis cases apart of the one with launch at sunrise and all the nominal thermal loads and power consumers switched on. In this case, for all three components, the temperature limits are respected. Nevertheless, the margin for the minimum temperature is too low.
In conclusion, in order to avoid any problem, these three components should be actively thermal controlled. Furthermore, it is reasonable to choose the launch at sunrise such that the components have the chance to heat up at the beginning.

[1]Some values for the temperature ranges are not displayed because they weren't considered during the thermal analysis. These values are not critical for the thermal simulation.

2.2 Components choice and philosophy

As a result of the thermal analysis, the hardware for sensors and heaters has been previously selected. As already mentioned in Sec. 1.3, the number and placement of sensors and heaters is still to be finalized after the thermal simulation of the prototype. Anyway this is not matter of this work. The software is developed in order to be easily configurable, in any moment of the project, in terms of temperature ranges and components number and placement.

2.2.1 Sensors

After the thermal analysis (Sec. 2.1), the measuring system for temperature, air pressure, humidity and structural data for STUDIO payload has been developed [12]. The focus for this work is on the temperature sensors, as this is essential for thermal monitoring. Anyway, the device handler component has been programmed in order to read and store also environmental and orientation data from the sensor board. This additional work has been done for future steps of the project.
The result of this work is shown hereafter in Fig. 4. The main components are:

- Temperature Measurement Board (TMB1, TMB2, TMB3, TMB4) based on the LTC2983 [13] and use of PT1000 [14] measuring resistors;
- the BME280 [15] which provide environmental measurement;
- the BNO055 [16] which provide orientation measurement;
- Arduino Micro [17] as micro-controller.

The system now has a total of:

- 36 measuring channels for temperature from the 4 TMB;
- 3 measuring channels each for air humidity, pressure and again temperature from the BME280;
- 3 measuring channels from the BNO055, for the three axis of orientation, each for Acceleration, Gyroscope, Magnetometer, Linear Acceleration, Euler Angles.

The system can be expanded. If possible, tried and tested components were used. The data is transmitted to the STUDIO payload computer every 1.6 seconds. This is sufficient for the STUDIO thermal controller to work. The components are placed in housings and attached to the STUDIO gondola. All cables used are low in outgassing. The use of plugs makes the system flexible.

It is a prototype that is still pending validation. On the one hand, the hardware must be validated by means of tests in the thermal vacuum chamber; on the other hand, the software has not been adequately tested for long-term stability and susceptibility to errors. Changes to the software may also be necessary due to the still missing

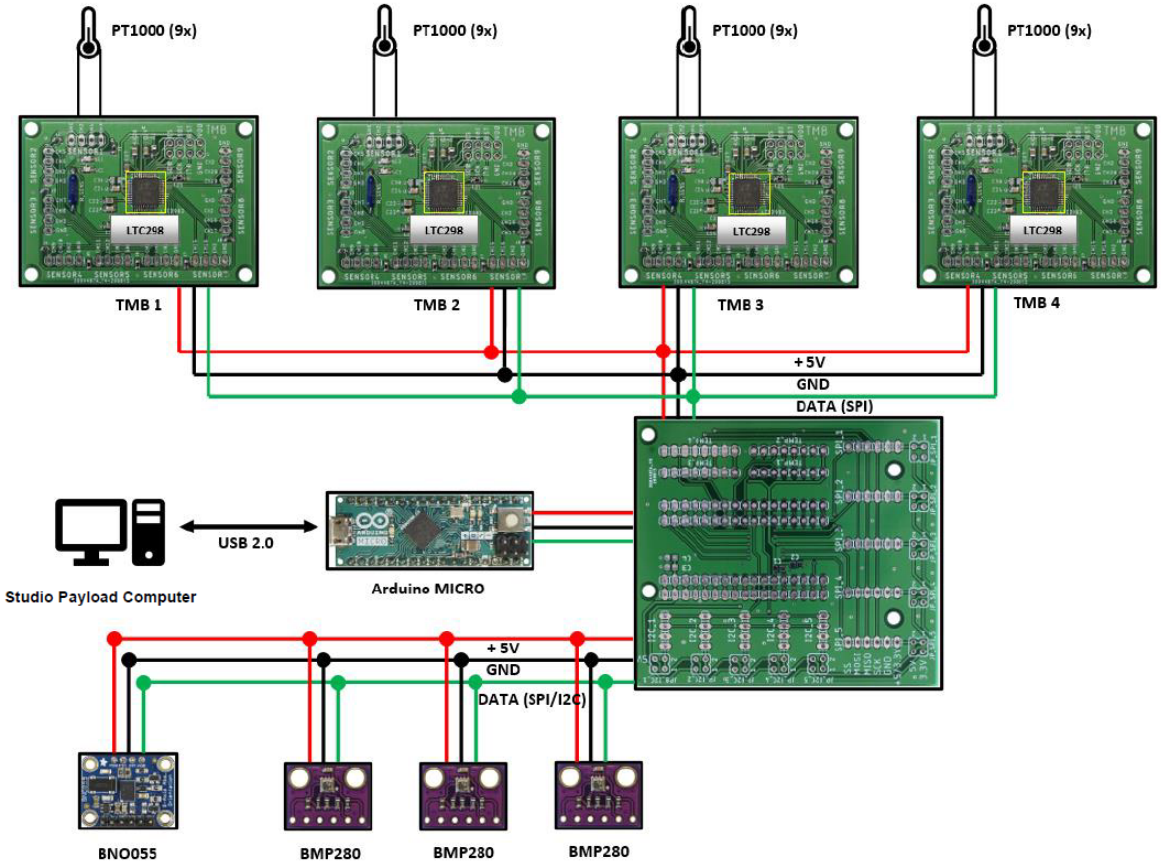

Figure 4: Sensor board. [12]

integration into the STUDIO payload computer. The use of only one micro-controller for the entire sensor system must also be viewed critically. The temperature measurement is particularly important for STUDIO, other measured values are not required in real time. For this reason, it would make sense to use a separate micro-controller for temperature measurement in order to simplify the software and increase its stability. In this context, an expansion to a redundant system would also make sense in order to increase the reliability. However, this can be implemented with little effort.

The Arduino is programmed using the platform's own programming environment Arduino IDE 1.8.13 in a C-like environment programming language. The code developed for the management and reading of the sensor is briefly discussed in Sec. 3.3.1. A part of this work is to implement a component of the thermal controller software, the device handler, responsible of the communication between the Arduino board and the FSFW.

2.2.2 Heaters

Similarly to the sensors, the number and placement of heaters is not finalized. This is not a problem for the software, which is designed, as required, with an high degree

of flexibility. From the result of the thermal analysis in Sec. 2.1, it is clear that there are not strong requirements on the heaters properties. It has been chosen a Polyimide Insulated Flexible Heater (PIFH) [18].
These heaters are thin, flexible, quick and highly precise in providing heat where needed to reduce operating costs. They add minimal weight to the entire process. Their polyimide construction provides fast and efficient thermal transfer and uniform thermal performance. They have optimal vacuum stability certified by the NASA outgassing ASTM-E595 [19].
The current placement of the heaters is described in Table 3. The heaters model with he associated power and voltage is reported. The positions selected are related to the components introduced in Table 1 and discussed durig the tehrmal analysis. This configuration will be tested and validated in clean room. After the thermal simulations, the final configuration will be chosen in terms of heaters number and placement. In this current configuration, some heaters on the TIP are added to ensure safe operations of the payload.

Table 3: Heaters placement.

Position	Power	Voltage	Model	Comments
On telescope				
M2 actuator 1	10 W	28 V	KHLVA-101/10	Integrated by OS
M2 actuator 2	10 W	28 V	KHLVA-101/10	Integrated by OS
M2 actuator 3	10 W	28 V	KHLVA-101/10	Integrated by OS
On TIP				
VIS	10 W	28 V	KHLVA-102/5-P	Combined for one power line
VIS	10 W	28 V	KHLVA-102/5-P	Combined for one power line
UV FW	7.5 W	28 V	KHLVA-103/2-P	Combined for one power line
UV FW	7.5 W	28 V	KHLVA-103/2-P	Combined for one power line
VIS FW	7.5 W	28 V	KHLVA-103/2-P	Combined for one power line
VIS FW	7.5 W	28 V	KHLVA-102/5-P	Combined for one power line
VIS	7.5 W	28 V	KHLVA-102/5-P	Contingency heater
VIS	7.5 W	28 V	KHLVA-102/5-P	Contingency heater

2.2.3 Thermal Control System model

The logic selected for the thermal control system is a basic closed-loop control system. It is represented in Fig. 5. The control required is simple. Therefore, there is no reason to complicate the thermal controller. The working principle is based on the acquisition of the temperature data measured by the sensor board. If the temperature is confined within the range given, no control action is required. Contrarily, the heaters are switched on or off in case of the temperature is too low or too high respectively. It is highlighted that the final temperature ranges for the control system will be given after

the thermal simulation, when the equipment choice and placement will be concluded. As for the equipment, the software is required to be customizable easily and directly.

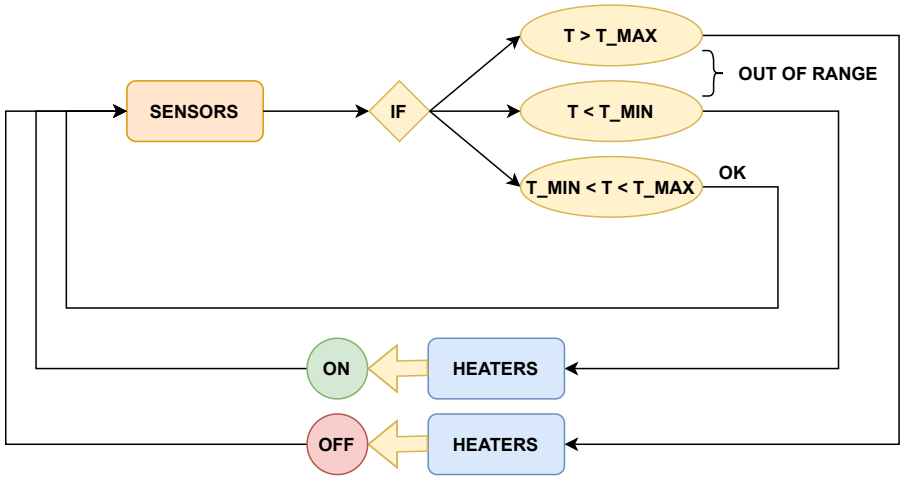

Figure 5: Thermal Control System model.

2.3 Architecture of the software

In this section, the objectives and requirements for the software, already discussed above, are briefly resumed. It is then introduced the software architecture and its main components. Then, it is presented a flowchart of the whole code.

2.3.1 Requirements

The main requirements identified for the thermal control software are:

- Auto-stabilization; the thermal controller shall stabilize automatically the thermal state on the preset temperature ranges. It requires n support by ground. Nevertheless, a communication channel with ground could be easily implemented, in order to allow a customization of the software.
- Distributed control; the defined spots to control in Sec. 2.1 shall be monitored and controlled independently. Therefore, each spot is equipped with its own sensor, heater and the selected redundancy. No thermal modules components are considered: these components are designed to monitor and control different part of the satellite as one whole unit.
- FSFW employment; as mentioned above, it is required to implement the code in the framework exploiting the tools described in Sec. 1.2.1.
- Code flexibility; it is required to keep the physical component used and the related data, as the temperature ranges, abstract. In this way, when they will be finalized after the thermal simulation of the prototype, it would be possible to add in the software as many components as needed.

These are the drivers for the whole development of the software.

2.3.2 Software components description

The main components which are developed in this project, with the aim to build the thermal control software, are:

- the device handler;
- the thermal controller.

They are implemented exploiting the FSFW templates discussed in Sec. 1.2.2. Regarding the device handler, the component template is well defined and it is expanded in Sec. 3. On the other side, the controller is implemented through the combination of different building blocks. That's because every controller has a specific purpose which is very different from that of another controller. For this reason, a lightweight controller template is provided with few common interfaces. It is then completed with the involvement of additional blocks. This is exhaustively discussed in Sec. 4.

2.3.3 Flowchart

A summarizing flowchart of the thermal control software is represented in Fig. 6. It is a simple extension of the control logic modelled in Fig. 5. The device handler, as mentioned above, is the component of the code that communicates directly with the physical equipment, as sensors and heaters in this case. The controller component does several tasks beyond the implementation of the control algorithm. In particular, it checks the state of the physical equipment and, depending on it, it does different actions, like switching operative modes or failure management (FDIR) if necessary.
If any data sent from the sensor board are corrupted, they are identified in the device handler and they are excluded. If the related sensor keeps to send corrupted data, a failure may have happened. This situation is simply handled by relying on the redundancy of the sensors. Similarly, when a heater doesn't work properly, it is switched off and set to non-operative mode by the controller. Its tasks are transferred to the redundant heater in the same position, which is set to operative mode and switched on when necessary.
All these operations of health check and mode set-up are handled by the controller. Every time a failure happens, the FDIR component is informed with a message. It automatically executes the common routine, informing the components affected and the ground station. Consequently, depending on the case, different solutions are adopted in order to recover the failure or, at least, to secure the affected components.

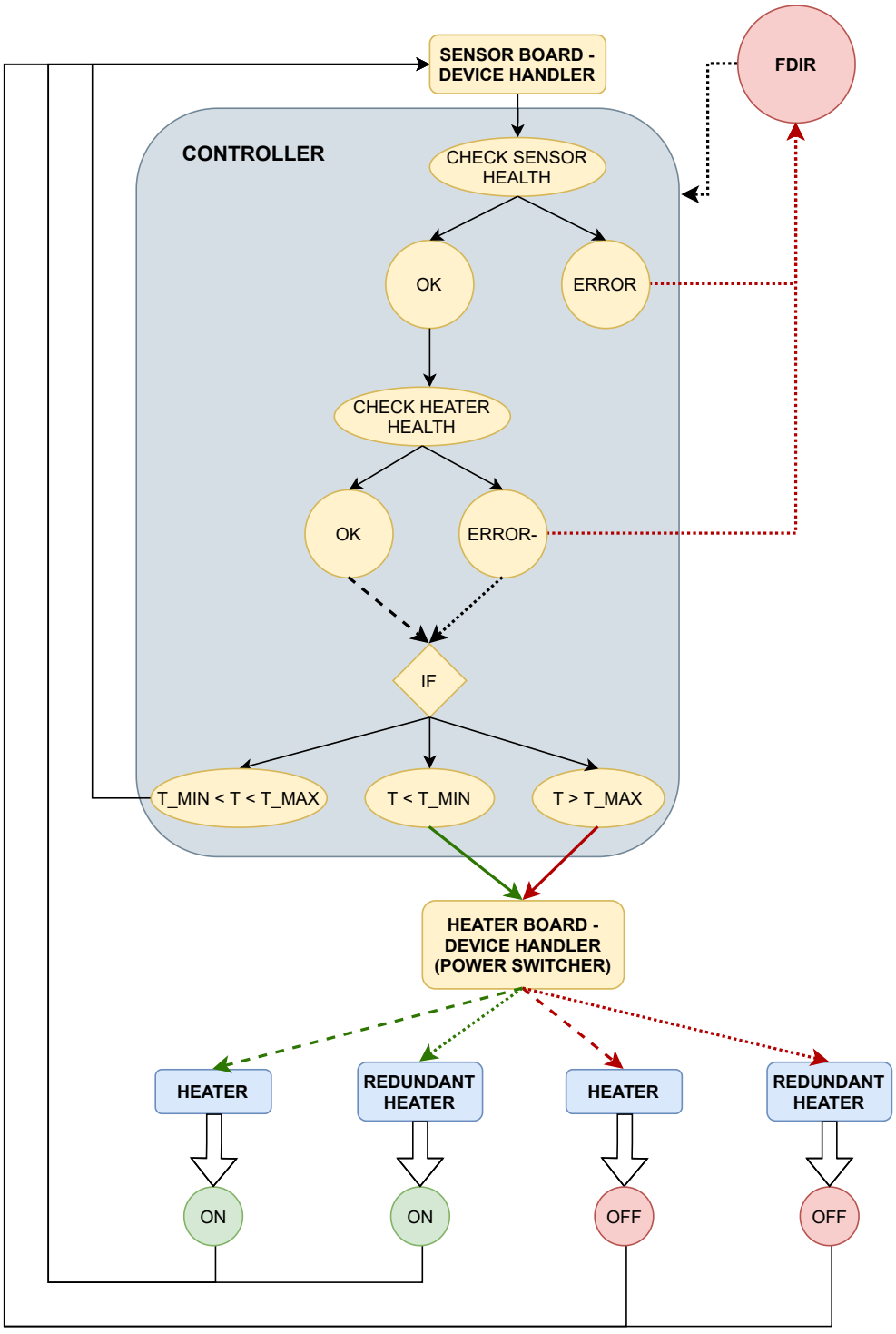

Figure 6: Flowchart[2].

[2]After the check of the heater health, if the heaters are working properly, redundancy is not employed (dashed lines). Viceversa, if the heaters fail, the redundancy is employed (dotted lines).

3 Device handler implementation

As already mentioned, the FSFW provides the core functionality required to handle the equipment. Each equipment is managed by a dedicated software component, the device handler, whose purpose is to control and monitor the equipment's status and communication [9]. It represents the equipment internally and for the ground segment. This choice brings the following advantages:

- device handlers explicitly control and monitor interaction with the device;
- device handlers encapsulate knowledge of specific properties such as initialization commands, the communication protocol and measurement representations;
- device handlers are the single source of state knowledge of the equipment, they determine the health state;
- state information along with measurements and command data are exchanged in a disciplined manner. This makes fault management more explicit and simple.

It is important to highlight that device handlers are on the same hierarchical level of controller components. The device handler is a single software component. In this project, it is developed differently for sensors and actuators.
It is remarked that, in this chapter, just the device handler related to the sensor board is described in details. The heater one will be very similar but it is still not completed since, as explained better in Sec. 4.5, the firmware for the power switcher communication interface shall be concluded before.
In addition to the device handler component, other two software components are included:

- the Communication Interface;
- the Cookie.

They are fully explained here below. Their objective is to assist the device handler tasks. Before, some concepts related to the FSFW core are introduced. They are needed to understand the development of the software components.

3.1 FSFW core concepts

Data Pool Data pool is a typical element of embedded control software in general and flight software in particular [9]. The basic idea is to provide a possibility for exchange of periodic data with blackboard logic. If only the most recent value is of interest, it is reasonable to provide a single location for this value and cyclically overwrite the current value with a newer one. As sensor and actuator data have a very short time of usage, the data pool is the main mechanism for exchanging control data between components.
It is implemented in the FSFW Core which, as represented in Fig. 2, communicates with every component. It is based on a shared memory mechanism with the following features:

- each variable is identified with a unique ID;
- the data pool allows storage of vectors and single variables;
- access is type-safe and thread-safe;
- a thread trying to overwrite parts of a set of variables is blocked if the set is currently read out, in order to avoid inconsistency;
- a flag for the validity of the variable is assigned to each entry;
- the data pool provides commit-and-rollback semantics for write access.

In the mission configuration file, this unique ID for each data pool variable is defined, and the variable or vector is mapped with its type and dimension, to allocate the required memory.

Events Events are aperiodic incidents in a system [9]. The FSFW provides an event manager, which is responsible of events distribution. It receives event messages generated by components and forwards them to those components that registered for a matching set of events. Components are capable of throwing events at any time. An event message contains the following information:

- a unique event ID;
- a severity field, it defines the priority of the event;
- the reporter ID, it is the ID of the component associated at that event;
- two parameter values, they are employed to send additional information about the event.

Objects As already introduced in Sec. 1.2, the software is composed by assembling different objects, also referred to as components. Every object included in the code is defined in the configuration with a unique ID. These objects could have a physical meaning, as the sensors, heaters, redundancies, switches, as well as they could be non-physical, as the device handler, the communication interface, the cookie, the power switcher or the thermal controller, which in this case is a simple implementation of the logic represented in Fig. 5.
All these independent objects, such as the cookie, the communication interface, the device handler and the thermal controller are instantiated using a factory. It is a part of the FSFW which manages the objects of the whole software exploiting the *object manager*. The instantiation takes place through parametrized constructors which are defined in the class of each object.
The flexibility required for the code is here satisfied: whatever are the number and the features of the objects to include in the thermal control system, in terms of sensors and heaters, they can be easily defined here at any moment.

Return Values Some methods belonging to these objects are designed to return values which are then exploited by other methods, objects or even ground operator. In order to distinguish these variables unequivocally, an ID associated with the class for this specific purpose, is added to the return values.For example, the device handler and the thermal controller are both assigned with an ID in the mission configuration, which differentiates a *RETURN_FAILED* message sent from each of them for the operator on the ground.

Polling Sequence The FSFW is capable of individual scheduling of components [9]. The components are executed periodically within a fixed time period. This a common scheduling scheme in the RTOS, which is also here exploited in a linux environment.
The components become executable by implementing the *ExecutableIF*, which mandates the implementation of the *performOperation* method. This method is called by task objects, which manage cyclic execution. The periodicity of the execution is not defined by the component itself, but it is defined by the task periodicity defined in the mission configuration. The components itself maintains all attributes and parameters unmodified until the end of the time period. For reason of efficiency, multiple components can be grouped in one task, which then execute in the same period and with the order defined. Any form of periodic execution is allowed by the FSFW. Parallel or quasi-parallel execution could be implemented as the communication is thread-safe.
Regarding the device handler, the execution schedule is more constrained by the equipment requirements. For this reason, it is necessary to optimize the scheduling. This situation is handled with the help of the *polling sequence table* [9]. The FSFW implements a dedicated tasking interface, called *FixedTimeslotTaskIF*, which allows adding components in time slots with a fixed execution time. It enables the precise scheduling of the four communication interface methods.
In the mission configuration file, the *InitMission* function defines the period tasks to be executed with the related time period and execution order. Regarding this project,

two period tasks are defined: one for the device handler and one for the controller.
As mentioned before, the device handler task is defined through the *FixedTimeslotTaskIF* with a slot of 3.2 seconds. This time represents twice the time needed by the Arduino sensor board to measure and send periodically the data obtained (Sec. 3.3.1). The precise scheduling of the internal tasks of the device handler is stated in the *pollingSequence* function. This design step has been a critical step. The solution adopted is described below in Sec. 3.4. The controller task is instead defined with the *PeriodicTaskIF* with the same time period. The time slot value is not a critical choice, therefore it is basically selected in order to have same time period of the device handler.

3.2 Cookie

The cookie component should be inherited by the FSFW class *CookieIF*. It can be used to store all kinds of information about the communication, like slave addresses, communication status or communication parameters. It is used to identify different connection over a single communication interface. In addition, the cookie state is used in the device handler base class as an indicator of the communication step a device is in.
In this case only one device is connected to the controller, for this reason the address is unique and there is no need to use the cookie for that. The constructor is therefore empty. The only task handled by this component is the initialization of the Linux serial port for the communication interface. The cookie transfers this information between the different methods within the device communication.

3.3 Communication interface

The communication interface component is created to decouple the device communication from the device handler. Thus, the device handler is not dependent on any specific interface and it can be reused.
This component should be developed through inheritance as represented in Fig. 7.

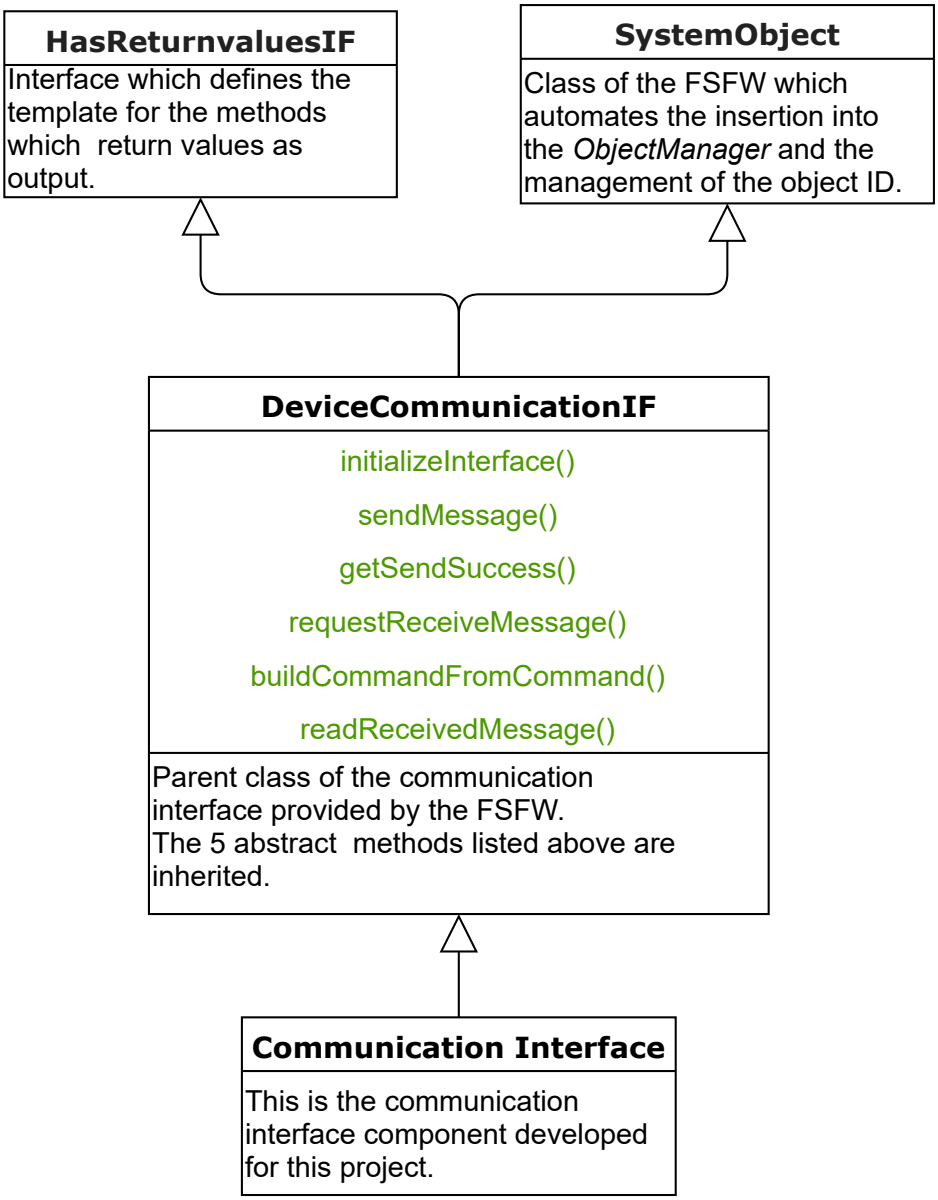

Figure 7: Communication Interface parent classes.

The main parent class is the *DeviceCommunicationIF* class of the FSFW. It works with the assumption that received data are polled by a component [9]. This class provides calls to initialize the interface (open/close) and it defines fours steps for the device communication. Its flowchart is represented in Fig. 8. The five methods here

discussed and shown in figures are tackled individually in the sections below.
The *DeviceCommunicationIF* of the FSFW is itself child of another class: the interface *HasReturnvaluesIF*. It simply defines the template for the methods which return values as output (Sec. 3.1). Furthermore, it defines the two main variables to return:

- *RETURN_OK*, returned when the execution of the method is successful;
- *RETURN_FAILED*, returned when any type of error happens during the execution of the method.

The communication interface component of the thermal controller, developed in the context of this project, is not only child of the class *DeviceCommunicationIF* as already stated, but it is also child of the class *SystemObject*. This class of the FSFW automates the insertion into the *ObjectManager* and the management of the object ID.
Before describing the class methods, it is necessary to provide a quick explanation of the Arduino sensor board software. This is done in the next section (Sec. 3.3.1).

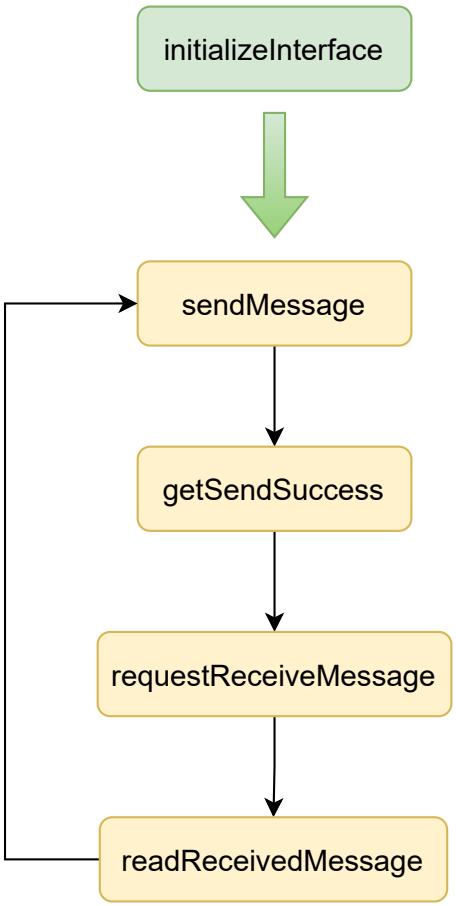

Figure 8: Communication interface flowchart.

3.3.1 Arduino sensor board code

As mentioned before, the development of the software for the sensor board is not part of this work. This software has been analyzed because it embeds the info necessary to initialize the interface for the communication with the device handler.
The process of measurement is not explained here. It is described the process of data sending from the sensor board to the the on-board computer: the STUDIO Payload Computer (SPC). The physical interface between the two hardware is a USB 2.0 connector (Fig. 4). These data are sent as serial output with a cycle time of 1600 milliseconds. The baud rate selected for the communication is 115200 bps. The SPC requires to receive this output as raw binary data. Consequently, the communication interface shall implement a method to decode these data in order to check them and use them in the thermal controller.
The structure of the data sent is here described. As already clarified in Sec. 2.2.1, the sensor board sends three different structures of data related to the three different type of measurement: temperature, environmental data, orientation. Each measurement channel sends its own measurement. In case the channel is not connected to any sensor, therefore is not operative, a default array of data is sent. In this way the controller can easily recognize which channel are not operative.
The four temperature boards, each one with nine measurement channels, send a total of 36 arrays of measurement. Each array is structured as in Table 4. Two char variables are included to define the beginning and the end of the array, in order to distinguish clearly each array with respect to the others. The variable *Type* represents the type of measurement which is always 1 for temperature measurement. The variable *SPCCnNumber* defines the unique ID of the measurement channel. It is:

- 1 to 9 for the first TMB;
- 11 to 19 for the second TMB;
- 21 to 29 for the third TMB;
- 31 to 39 for the fourth TMB.

Each temperature board number is defined and associated with a specific pin on the Arduino board. The TMBs are connected to the pins from 4 to 7. Knowing the measurement channel, it is possible to identify the sensor associated and therefore the location and control action to apply if needed. The *Value_ Cnt* defines the number of measurement registered in the array. Regarding the TMB, only the last measurement is sent and it is overwritten at each measurement cycle. Consequently this variable is always 1 for temperature measurements. The float variable *Temperature* contains the temperature value measured at each loop in Celsius degrees. Lastly, the *Timestamp* is a variable which defines the exact time instant, in milliseconds, in which the measurement is done with respect to the start of the program. It exploits the function *millis* of Arduino IDE. Each variable is associated with its corresponding size in terms of bytes. It is needed to allocate the necessary space in a buffer array which stores these data sent by the Arduino board.

Table 4: Temperature data structure.

[*ChStart*]	Type	SPCChNumber	Value_Cnt	Temperature	Timestamp	[*ChEnd*]
char	uint8_t	uint8_t	uint8_t	float	unsigned long	char
8 bytes	1 byte	1 byte	1 byte	4 bytes	4 bytes	8 bytes

As already mentioned, even if it is not interest of the thermal controller, also environmental and orientation data are read and stored in the context of this project. This is done for further usage by other flight software components.
The array of environmental data has a similar structure to the temperature one. It is shown in Table 5. The three BME280 boards are connected to the Arduino board with the pins from 8 to 10. Each board provides a measurement of pressure, humidity and temperature. Therefore, the *Type* is in this case:

- 2 for pressure [hPa];
- 3 for humidity [%];
- 1 for temperature [°C].

The *SPCCnNumber* is:

- 41 to 43 for the first BME280;
- 51 to 53 for the second BME280;
- 61 to 63 for the third BME280.

In this case the *Value_Cnt* is still 1 for each measurement channel because, as for temperature measurement, just the last measuremnt is sent and then overwritten at the next cycle. The variable *Value* contains the measurement value of environment. It has the same format for the three different channels.

Table 5: Environmental data structure.

[*ChStart*]	Type	SPCChNumber	Value_Cnt	Value	Timestamp	[*ChEnd*]
char	uint8_t	uint8_t	uint8_t	float	unsigned long	char
8 bytes	1 byte	1 byte	1 byte	4 bytes	4 bytes	8 bytes

The array of orientation data is shown in Table 6. The BNO05 board, as mentioned in Sec. 2.2.1, features five different types of measurement: Acceleration, Gyroscope, Magnetometer, Linear Acceleration, Euler Angles. Therefore the *Type* is here:

- 4 for acceleration measurement $[m/s^2]$;
- 5 for gyroscope measurement $[°/s]$;

- 6 for magnetometer measurement [μT];
- 7 for linear acceleration measurement [m/s^2];
- 8 for Euler angles measurement [°].

Each type of measurement is provided on the three axis of the reference frame. The *SPCChNumber*, then, is:

- 81 to 83 for acceleration measurement;
- 91 to 93 for gyroscope measurement;
- 101 to 103 for magnetometer measurement;
- 111 to 113 for linear acceleration measurement;
- 121 to 123 for Euler angles measurement.

The *Value_ Cnt* for orientation data is 9. This is because the variable *Value* is an array of nine measurements in which the first character represents the newest measurement. Then the last eight measurements, in order of time, are recorded in the array and at each measurement cycle they are translated one step down in the array. In the *Timestamp* variable, the nine time instants related to the measurements are collected.

Table 6: Orientation data structure.

[*ChStart*]	Type	SPCChNumber	Value_Cnt	Value{9}	Timestamp{9}	[*ChEnd*]
char	uint8_t	uint8_t	uint8_t	float	unsigned long	char
8 bytes	1 byte	1 byte	1 byte	36 bytes	36 bytes	8 bytes

The whole packet of data is composed by a total of 2580 bytes.

3.3.2 Interface initialization

The function *initializeInterface* is the first method of the Communication Interface class (Fig. 8). It is a virtual function of the base class *DeviceCommunicationIF*, therefore it must be overriden in the derived class of the thermal controller. This is the same for the four methods for device communication which are virtual functions too. The prototypes in the component header are indeed all overrides of the base class.
This method provides the set-up of the interface for the communication. The serial communication interface through which data are transferred is a Linux serial port. In typical UNIX style, serial ports are represented by files within the operating system. These files usually pop-up in */dev/*, and begin with the name *tty** [20]. This file for the serial port is configured with particular attention for some parameter.
Firstly, the serial port is opened. The device path is identified on the Linux terminal. Then, a new termios structure, which is called 'tty' for convention, is created and

configured as in [20]. The required libraries are included. The main parameters to set are shown in Table 7.

Table 7: Serial port parameters.

Parameter	Command	Explanation
Baud rate - input Baud rate - output	*cfsetispeed(&tty, B115200)* *cfsetospeed(&tty, B115200)*	The serial port baud rate is set by calling the functions cfsetispeed, for the imput, and cfsetospeed, for the output. The baud rate for the communication is defined in the Arduino sensor board code. It is 115200 bps. Here it is defined accordingly.
VMIN	$tty.c\_cc[VMIN] = 255$	This parameter represents the number of bytes which the Linux serial port shall wait for in the *read()* call. The upper limit of bytes to read in one call is 255 bytes, which is less then the total of 2580 bytes sent. Consequently, the parameter is fixed as 255 and the *read()* call shall be repeated enough times to receive all the data.
VTIME	$tty.c\_cc[VTIME] = 0$	This parameter specifies a time-out from the start of the *read()* call. Setting it to zero, there is no time block. Therefore it makes the *read()* call always wait for the bytes defined above. It could block indefinitely.

The cookie component, which ID is provided as input of this method, is necessary to store the integer number representing the serial port here opened. This number is exploited by the methods related to the device communication.

3.3.3 Device communication

The four methods implemented for the device communication are briefly described in Table 8.

Table 8: Device communication methods.

Method	Description
sendMessage	It is used to send data to the physical device by implementing and calling related drivers or functions.
getSendSuccess	It sends confirmation that the data in *sendMessage()* was sent successfully.
requestReceiveMessage	It requests reading data from a device. It is assumed that it is always possible to request a reply from a device.
readReceivedMessage	This function is used to read the received data from the physical device by implementing and calling related drivers or functions.

In this project, the aim of the device handler is to read the data received by the Arduino sensor board. There is no further communication. For this reason, the first three methods are just returning *RETURN_OK* without doing anything in their routine. The fourth methods, instead, is programmed properly in order to read the packet of data sent by the Arduino board.
Firstly, the buffer arrays for data reading are initialized. Since these buffers have a size of 255 bytes, 11 buffers are needed to read the whole packet of data. Nevertheless, as a result of the tests which are explained in Sec. 5, it is chosen to block the reading until three whole packets of data are sent. It takes more time but it ensures the FSFW to read correctly at least one whole packet among all these bytes. It means that the total bytes to read are 7740. This choice is due to the fact that the component execution in the FSFW, as already mentioned in Sec. 3.1, is periodic. This is a reason of synchronization error during the reading. Therefore, 31 buffers of 255 bytes needs to be initialized to allocate the required space for reading.
These buffers are set to zero with *memset()* function. This is done to overwrite whatever data has been read the cycle before. It prevents to keep any corrupted data and it allows to identify which channels are not read for any reason.
Then, the *read()* call is repeated for all these buffers to store the three whole packets of data as explained above. The resulting arrays are concatenated then in a single array of 7740 bytes with the function *copy()*. The objective, then, is to identify a full and not corrupted packet of measurement in this buffer. A control loop is performed on the buffer array for this reason. The logic is very simple and it is represented in Fig. 9. The check is performed in order to ensure that the whole packet starts exactly from the first channel measurement and it ends with the last channel measurement.

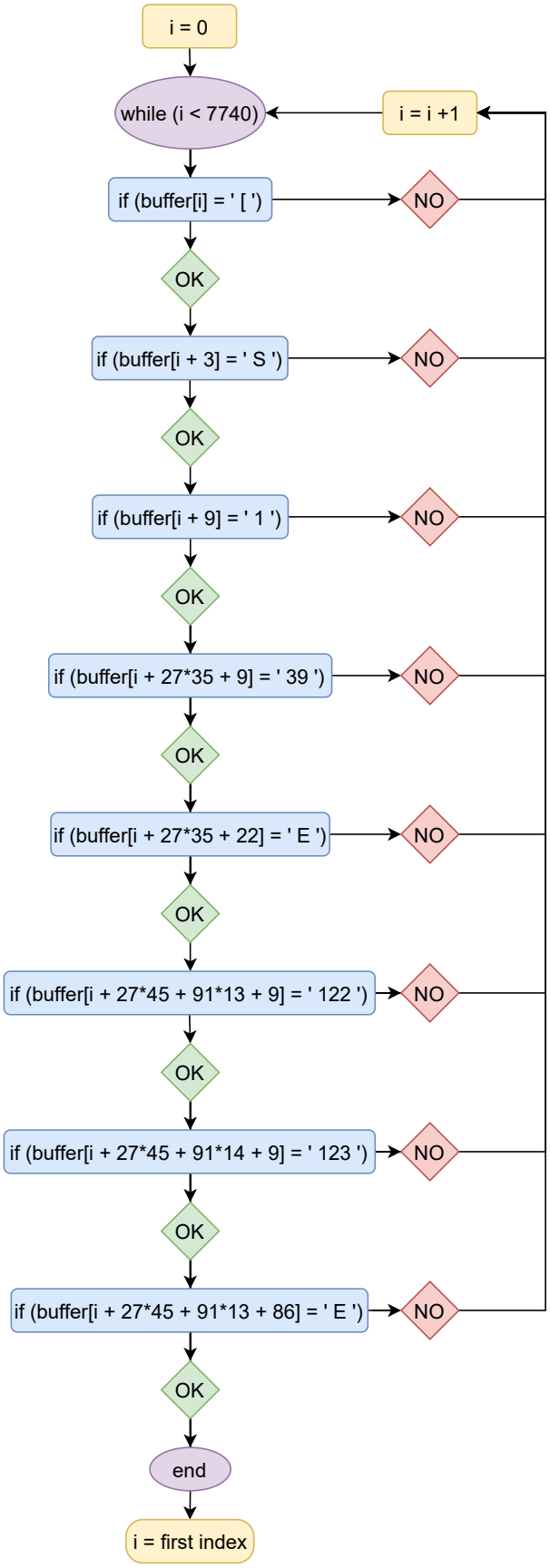

Figure 9: Buffer control loop.

As result, the presence of the whole packet is checked and the address of the first index is identified and returned as output. The address of the first index of the whole buffer array is returned too. These two parameters are then employed in the device handler component to recover the packet of data read which is then checked, interpreted and saved in the data pool for the thermal controller. The logic employed in the control loop above could be refined or detailed more. However, the tests demonstrate that it is enough.

3.4 Device handler

The Device Handler component is inherited from the FSFW *DeviceHandlerBase* class. This base class is itself child of other classes and interfaces. All the parent classes are represented in Fig. 10. Most of them are interfaces which define common functionalities for software components. A brief description of each class is given. For the *DeviceHandlerBase*, the main methods inherited are listed. In particular, the *scanForReply()* and *interpretDeviceReply()* are highlighted because they are of primary interest for this project. A description of the methods is reported in Table 9. Besides the ones already introduced in Fig. 10, two other methods are inherited and implemented in the device handler.

As already mentioned above, the two methods *scanForReply* and *interpretDeviceReply* require the biggest part of the work. They are crucial in order to achieve the device handler requisites.
In this project, the *scanForReply* checks the packet validity just in terms of packet length. This is due to the fact that in the Communication Interface, as described in Sec. 3.3.3, a control loop on the packet received is already implemented. Therefore, if the packet received has a length of 7740 bytes, corresponding to the length of three measurement packets, is valid. It means that the communication interface has read the message correctly. In this case the function returns *APERIODIC_REPLY*. This is the output to return if a valid packet, which has not been requested by any command, is received and it should be handled anyway. Differently, the function returns *LENGTH_MISSMATCH*.
The following step is the interpretation of the message received. After the check of the structure of the message, its content is analyzed and then managed accordingly. Firstly, three structures for the different measurement data (Table 4, Table 5, Table 6) are initialized in the header. Then, one whole packet of measurement data is copied in these structures byte per byte. The address of the first byte is provided by the control loop in the Communication Interface. All the 2580 bytes are in this way decoded. This procedure is performed within a for-loop exploiting the function *memcpy()*.
In succession, the variables with the temperature value and the timestamp are saved in two arrays in the data pool. In this way, they can be accessed by the thermal controller component. The other data, as well as environmental and orientation data, are decoded for further use. They are not employed by the thermal controller.

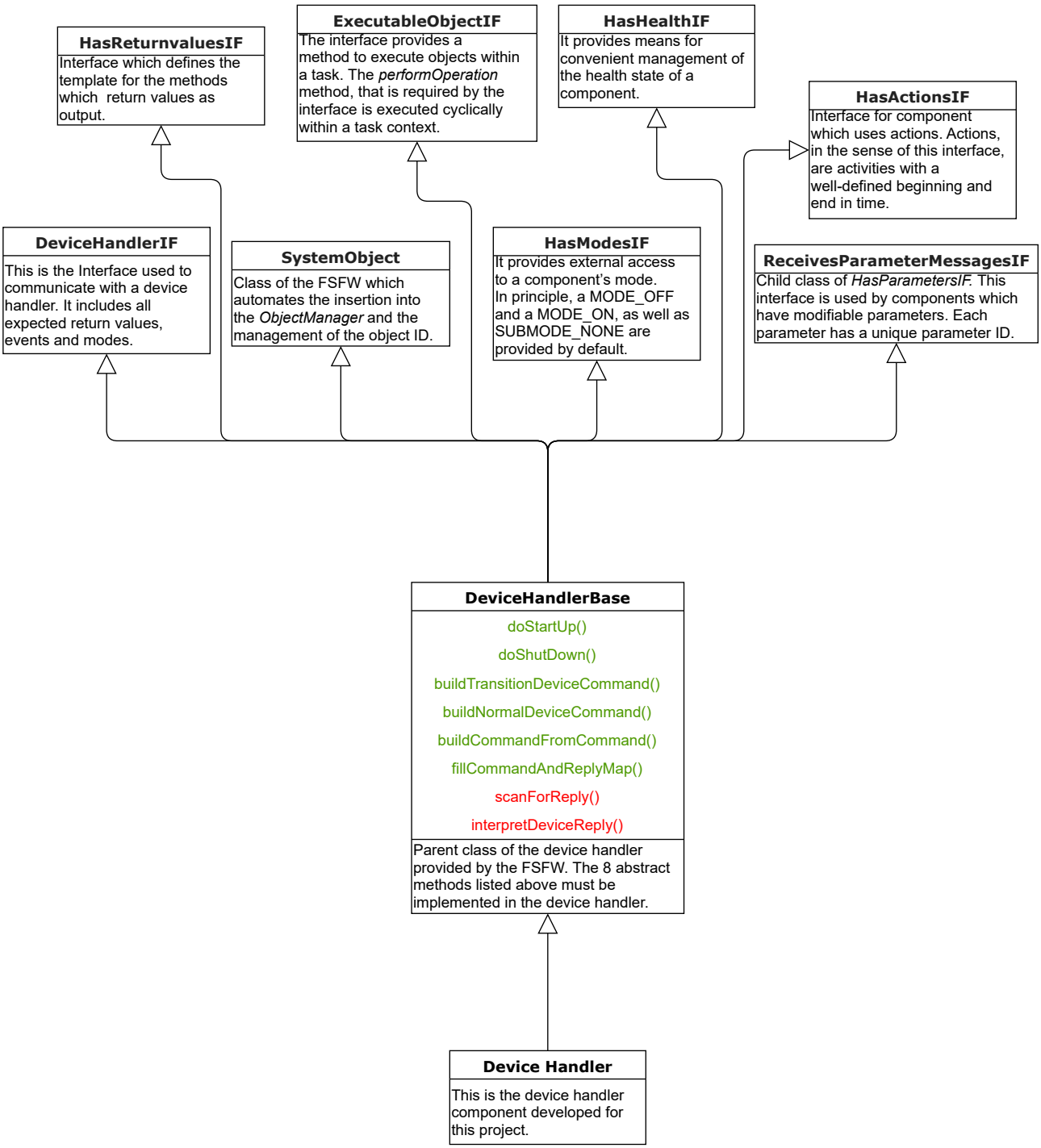

Figure 10: Device Handler parent classes.

Table 9: Device Handler methods.

Method	Description
doStartUp	This is used to let the child class handle the transition from mode *_MODE_START_UP* to *MODE_ON*. Device handler commands are read and can be handled by the child class. It should also send a reply accordingly.
doShutDown	This is used to let the child class handle the transition from mode *_MODE_SHUT_DOWN* to *_MODE_POWER_DOWN*.
buildNormal-DeviceCommand	It builds the device command to send for normal mode. Different commands can built, depending on the submode.
buildTransition-DeviceCommand	It builds the device command to send for a transitional mode. It is used by *doStartUp()* and *doShutDown()* as well as *doTransition()*.
doTransition	It performs the transition to the main modes (*MODE_ON*, *MODE_NORMAL* and *MODE_RAW*).
buildCommand-FromCommand	It builds a device command packet from data supplied by a direct command.
fillCommandAnd-ReplyMap	This is used to let the base class know which replies are expected. There are different scenarios regarding this: normal commands, periodic unrequested replies, aperiodic unrequested replies.
scanForReply	It scans a buffer for a valid reply. This is used by the device handler to check if the data received are valid packets. It only checks if a valid packet starts with the correct character and if the structure is coherent. No information check is done. Errors should be reported directly, the base class does not report any error based on the return value of this function.
interpretDeviceReply	It has the objective of interpreting a reply from the device. This is called after *scanForReply()* finds a valid packet, it can be assumed that the length and structure is valid. This routine extracts the data from the packet into a Data Set and then either sends a TM packet or stores the data in the data pool depending on external commands.
setNormalDatapool-EntriesInvalid	It sets all data pool variables, that are updated periodically in normal mode, invalid

3.5 Software development and test plan

The development and testing of the device handler software component can be resumed in the following steps:

1. Writing and testing of a C++ file to initialize the interface for the communication with the Arduino board. This step includes the study of Linux serial port and Arduino environment.

2. Inclusion in the C++ file of the code piece necessary to read and decode the measurement packet. At this stage the data are printed to the monitor after the decoding to check the results. The corrupted data are identified visually and the code is corrected.

3. Rearrangement of the file in order to obtain a continuous and periodic loop of data reading and decoding.

4. Transfer the code obtained in the environment of the FSFW. The components are built with the tools of the framework. Additional tasks and functionalities are added exploiting the FSFW.

5. Test the whole device handler software component within the FSFW. The reading from the sensor board is here tested.

6. Integration with the controller component and test of the whole thermal control software within the FSFW employing the STUDIO ground system set-up.

7. Placement of the actual temperature sensors and heaters on STUDIO telescope, located at IRS clean room, and integration of the system toward a fully-operational system.

The execution and results of these test are discussed in Sec. 5.

4 Thermal Controller implementation

The controller is the central element of this work. It monitors sensor values, applies control laws and calculates output values for actuators. As already mentioned, the FSFW provides a lightweight template with some common interfaces. In addition, some domain-specific building blocks are delivered in order to support the assembly of the control system.
All the components employed in this work are presented in Fig. 11. Each one is explained separately in this chapter. The logic selected for the thermal control is based on the concept of auto-stabilization: the control is performed automatically by the system without the need to communicate with ground. Therefore the development of the components is quicker. Nevertheless, a custom stabilization with continuous interaction with ground could be easily included in this software. Each spot to control is associated to a thermal component. Each thermal component is linked with its own sensor, heater and redundancies. This is due to the fact that each spot to control is handled independently, as required.

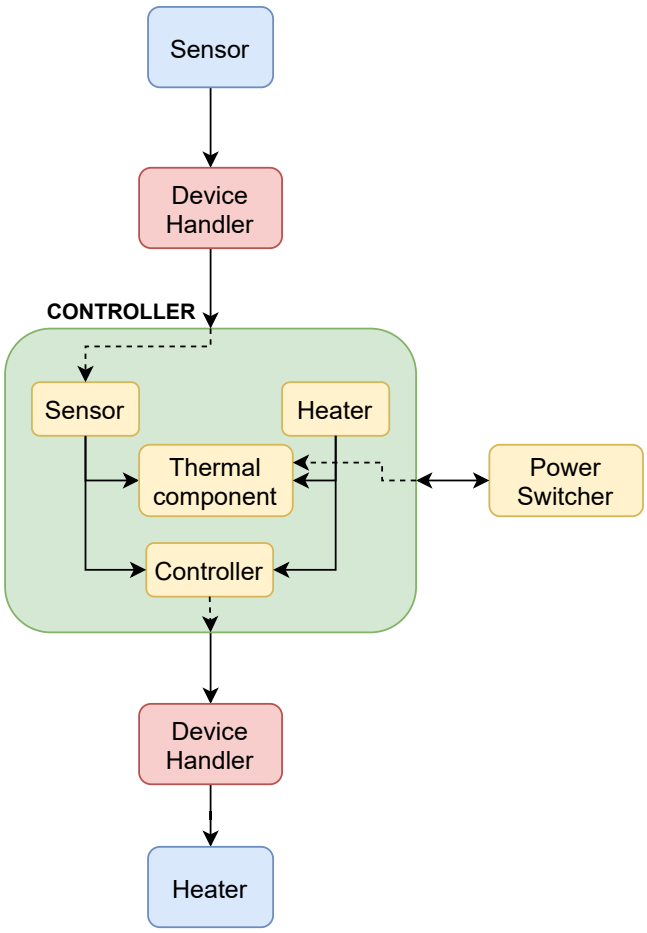

Figure 11: Thermal Controller components.

4.1 Sensor

The temperature sensor component represents the physical sensor in the controller component. Its main tasks are:

- reading of the temperature data saved by the device handler;
- check of the validity of the temperatures.

It is called within the routines of the thermal component and the controller.
It is child of the FSFW building block called *AbstractTemperatureSensor*. The process of inheritance is represented in Fig. 12.

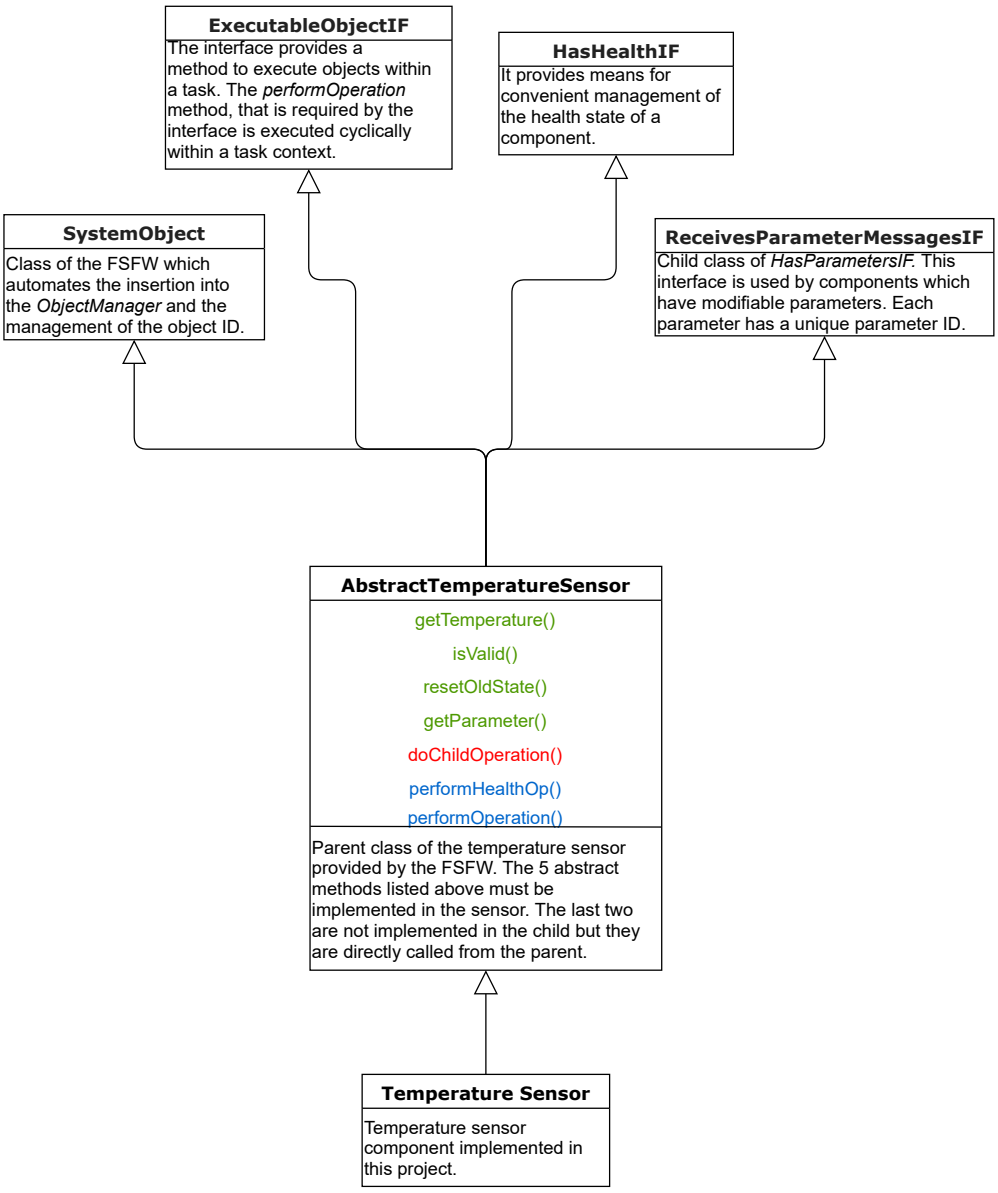

Figure 12: Temperature sensor parent classes.

As represented, the *AbstractTemperatureSensor* parent class is itself child of some common interfaces provided by the framework.The functions inherited by the child are the first five as written in figure. The first four are virtual functions which must be implemented, while the *doChildOperation()* method is the main function to implement in the child component. Furthermore, the *setInvalid()* method is created to assist *doChildOperation()*. Basically, the *doChildOperation()* method checks if the temperature values measured are within the limits defined. These limits are defined for each thermal component as $\{-1000°C, +1000°C\}$. They can be modified independently from the others in any moment. If they are within the limits, the measurement is set as valid with a flag in the data pool. Contrarily, it is set as invalid.
The last two methods written in figure are not implemented by the child class. This is because they do not need any modify. Therefore, the controller child class calls them directly from the sensor parent class. They both call the method *handleCommandQueue()* which is not implemented too in the child class. This is because no communication with ground is needed in this system. In case it is considered necessary, the method shall be overridden and modified according with the requirements. The *performOperation()* calls then the child *doChildOperation()* method to check the temperature values validity.
The tasks of this component within the controller are clarified in the following sections.

4.2 Heater

Similarly to the temperature sensor, the heater component represents the physical actuator within the controller. Its main tasks are:

- health monitoring of the heaters and the redundancies;
- monitoring of the heaters internal state;
- switch of heater internal state;
- switch from faulty heaters to redundancies.

Firstly, a breakdown of the component is given. The methods inherited by the heater are represented in Fig. 13. In addition to the heater parent class, some functionalities are inherited by one its friend class: the *RedundantHeater* class. These functionalities are exploited when the heater created is designed as a redundancy. As in the other cases, other common interfaces of the FSFW are inherited through the heater parent class.
These methods are designed to do different operations with respect to the internal state of the heater, or, in case of failure, of the redundancy. The internal state of the heater could be:

- ON;
- OFF;

- PASSIVE;
- STATE_WAIT_FOR_SWITCHES_ON;
- STATE_WAIT_FOR_FDIR;
- FAULTY;
- WAIT;
- STATE_EXTERNAL_CONTROL.

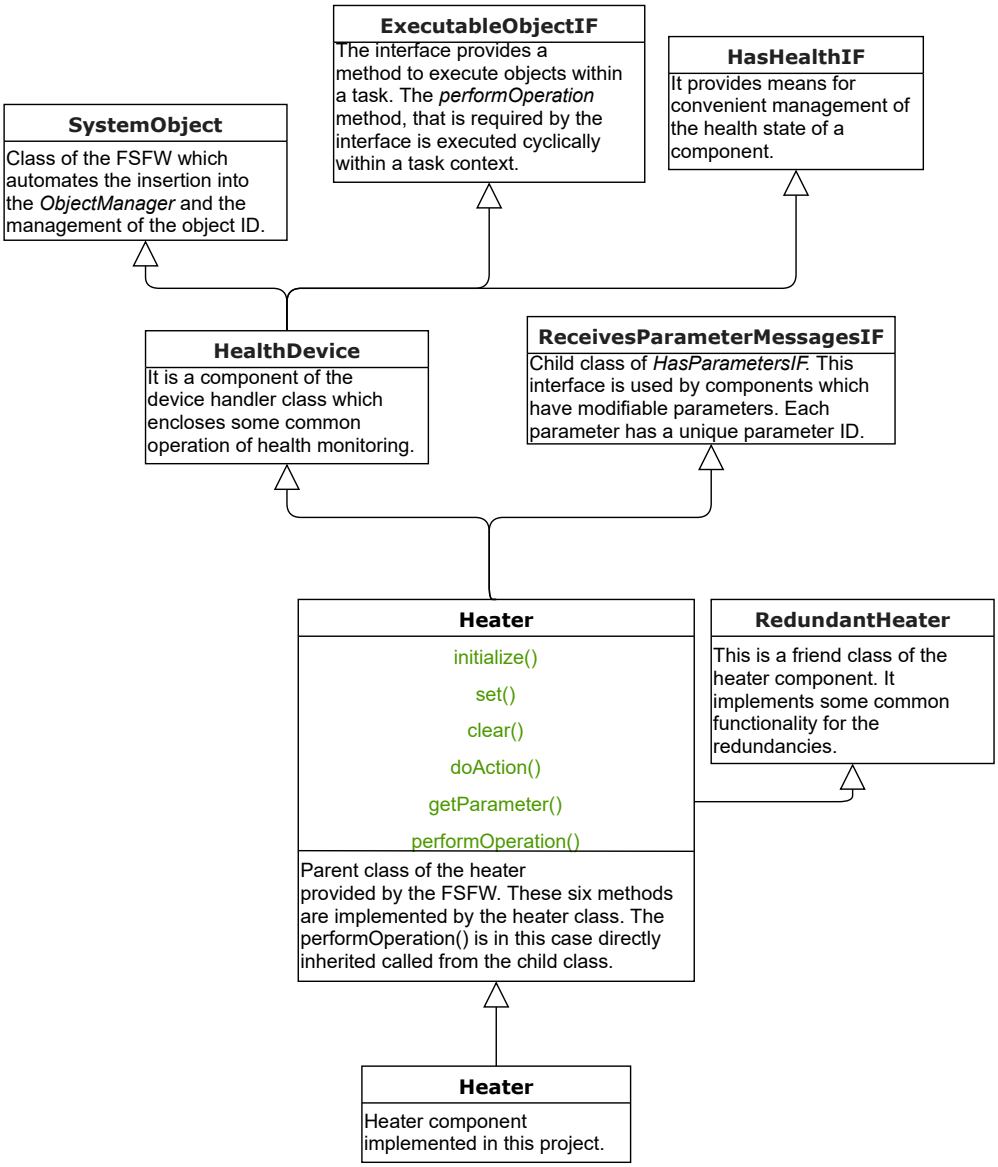

Figure 13: Heater parent classes.

As for the temperature sensor, these methods are called by the thermal component and controller. The methods *set()* and *clear()* have the objective of assisting the switch action of the physical heater. They both call the *doAction()* method to carry out the action selected. The method *set()* imposes the action:

- SET, if the heater is healty; the *doAction()* method consequently switches on the heater through the power switcher and changes the internal state;
- CLEAR, if the heater is FAULTY; the *doAction()* method consequently switches off the heater through the power switcher and changes the internal state.

On the contrary, the method *clear()* imposes the action CLEAR whatever is the condition of the heater. This method is implemented, indeed, with the aim of forcing the heater to switch off in particular cases. For example, it is employed to force the switch off at the beginning, in order to ensure the correct start of the software with internal state OFF. The *performOperation()* method, instead, checks the internal state of the heater and the state of the switch. It is highlighted that each heater or redundancy is associated with a unique switch with the parameterized constructor. As result of this check, different actions of the power switcher are triggered and the internal state is modified. The actions triggered here are several, even because there are different checks and actions for each internal state of the heater. The objective of these actions is to trigger the power switcher in order to have a switch state coherent with the internal state and the needs of the system.
As for the temperature sensor component, the employment of these methods within the whole controller logic clarifies the component tasks.

4.3 Thermal component

The thermal component is the main building block of the controller. It represents a specific spot on the satellite to control thermally. Each thermal component is associated with one temperature sensor, one heater and the redundancies. It is designed to include until two additional sensors and heaters as redundancy. The structure of the redundancies is the same of the main components, both for sensors and heaters. This is done not only for simplicity but also to ensure same functionalities also for the redundancies. The initialization of each thermal component is performed with a parameterized constructor within the thermal controller component. This initialization is done equally for both the temperature sensors and heaters. It is highlighted that these sensors and heaters components represent the equipment in the controller component. Nevertheless, the communication with the physical equipment occurs through different components, the device handlers.
The main tasks of the thermal component are:

- check of the temperatures through the sensor component methods;
- execution of the control algorithms;
- command the control action to the actuators;

- manage the redundancy switch and the mode change;
- monitor the feedback of the control actions.

The structure of the component is represented in Fig. 14. A brief description of the parent classes and the inherited methods is given.

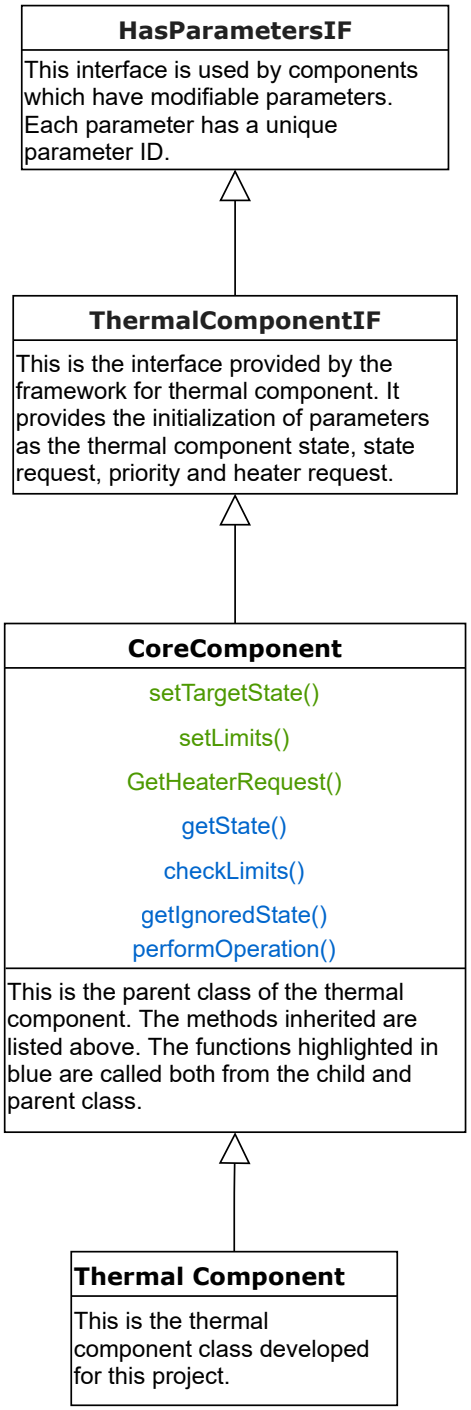

Figure 14: Thermal component parent classes.

The last four methods are highlighted because they are not overridden in the child component. The thermal component, indeed, is slightly different from its parent. It considers non-operational state of the system and non-operational low and high temperatures as parameters. For this reason, the methods inherited by the child need just a little extension of the parent functionalities. This is realized calling the parent classes directly within the child routines for these four methods, in order to avoid to rewrite the same thing. Then, the needed addition of code is included.
As always, other methods are inherited besides the ones listed in figure. These are the inherited methods which are implemented in the child component.
Apart of the usual common interfaces, the parent *CoreComponent* has a dedicated interface: the *ThermalComponentIF*. It provides the definition of many inherited methods. Furthermore, it contains the initialization of some paramount parameters for the control tasks:

- the thermal component state; it is a combination between: operational and non, out of range, low, high, ignored and unknown;
- the state request; it can be: heating request, ignore request, operational or non operational request;
- the priority; it is distinguished between: safe, idle, payload and max priority;
- the heater request; the options are: request ON/OFF, emergency request ON/OFF, heaters don't care.

The tasks carried out by these methods are here explained. Most of the methods inherited are enclosed and executed within the *performOperation()* routine. It is briefly resumed in the flowchart in Fig. 15.
The first step of this method is the call of the heater component function *performOperation()*, which is described above in Sec. 4.2. It is called also for the redundancies, if they exist for the specific component. Therefore in this step, the operations already described, related to the heaters switch and the internal state change, are executed.
Then, the second step is the call of the *performOperation()* method of the *CoreComponent* parent class. This encloses almost all the inherited functions belonging to thermal component. Firstly, it recalls the *getTemperature()* method of the sensor, component described in Sec. 4.1, in order to recover the temperature measurements. Then, if the temperature values are declared as valid, the control algorithm is executed. The validity is checked before by the sensor *doChildOperation()* method. This is shown in the next section. Here, the *getState()* method is called. It compares the temperature measurements with the operational and non operational limits. These parameters, as mentioned at the beginning of this section, are initialized in the constructor and are customizable for each component in any moment independently with respect to the others. As result, the thermal component state is here defined. The function *checkLimits()* is then called to do a check exploiting the monitoring functions provided by the FSFW. Consequently, the *GetHeaterRequest()* method is executed. Similarly to the previous function, it compares the temperature measurements with the operational

and non operational ranges in order to compute the heater request. The limits defined in the design rule thus the decision of switching on or off the heaters and redundancies. The heater request is then returned to the thermal component routine.

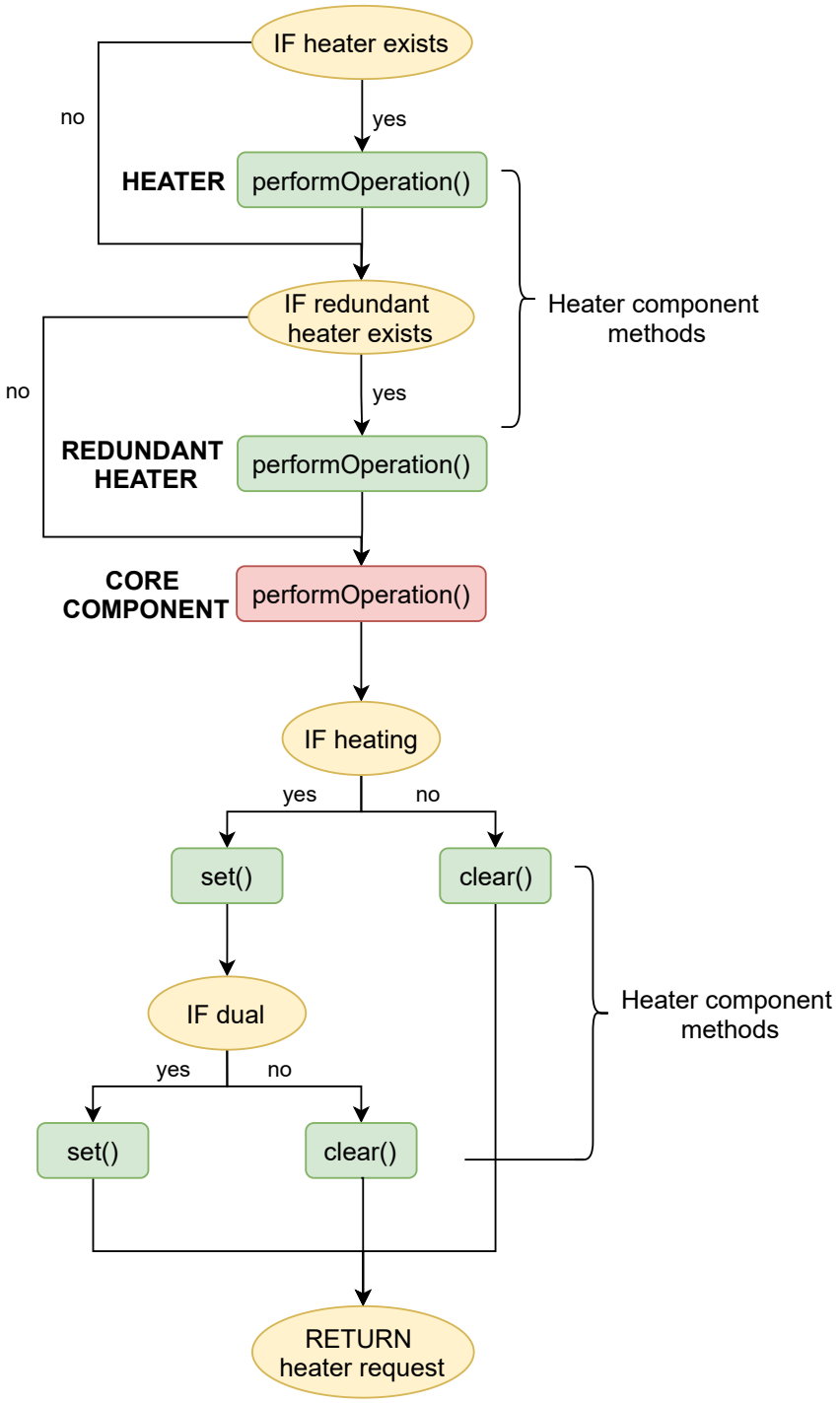

Figure 15: Thermal component - *performOperation()* method flowchart.

The third step consists of defining of the heater actions which are then executed at the beginning of the next loop, exploiting the power switcher. Whether or not it is requested to heat, different actions are ordered to the heater. These actions are executed through the methods *set()* and *clear()* of the heater component. An additional possibility is included: it can be set, with the *dual* mode, to provide the heating with both heater and redundancies switched on. This can be exploited in case a faster heating is required.
In the end, the method returns the heater request which has been computed within the loop. It could be exploited by the controller component. The methods which have been not discussed yet are executed in other parts of the software. They support and monitor the control loop.

4.4 Thermal controller

The Thermal controller is the main class, inherited from the controller base class, that exploits all the tasks and building blocks described above in order to obtain the required objectives. It inherits from the classes represented in Fig. 16. As already mentioned, the *ControllerBase* class provided by the FSFW is a lightweight template which implement some basic functionality. Other functionalities are included through inheritance of some common interfaces already seen in the previous components.
The *initialize()* method is employed to support the initialization of the objects and to define the starting mode and sub-mode of the controller. They are defined respectively as *MODE_NORMAL* and *HEATER_REDUNDANCY*. It means that the controller is in a normal operative mode and the redundancy is expected for the equipment. In order to support the mode management, the *checkModeCommand()* and the *startTransition()* methods are developed.
The *performOperation()* method of this component is the main method of the whole section. It encloses the execution of all the routines described above. Before describing the method, it is remarked that all the classes designed within the thermal controller are instantiated in this component. It means that all the sensors, heaters, redundancies and thermal components selected in the design are here defined and initialized with their own set of parameters. As mentioned, the thermal component is not a "physical" equipment. It represents a spot to control thermally which is associated with dedicated real sensors, heaters and redundancies. All these components are stored in lists of objects exploiting the *std::list* routine of C++. It helps to organize the software objects and to shorten the code. This structure respects the requirement of flexibility related to the software development. Indeed, any number and type of component could be added easily and in any moment without affecting the controller behaviour. All the parameters, related to the components instantiated, are initialized through the parametrized constructor of this controller component. Two data sets are defined for this component: one for the sensor data and one other for the thermal component data. These spaces are used to manage the periodic data used by the code and to communicate with the data pool. The temperature data are transferred from the data pool to the sensor data set.

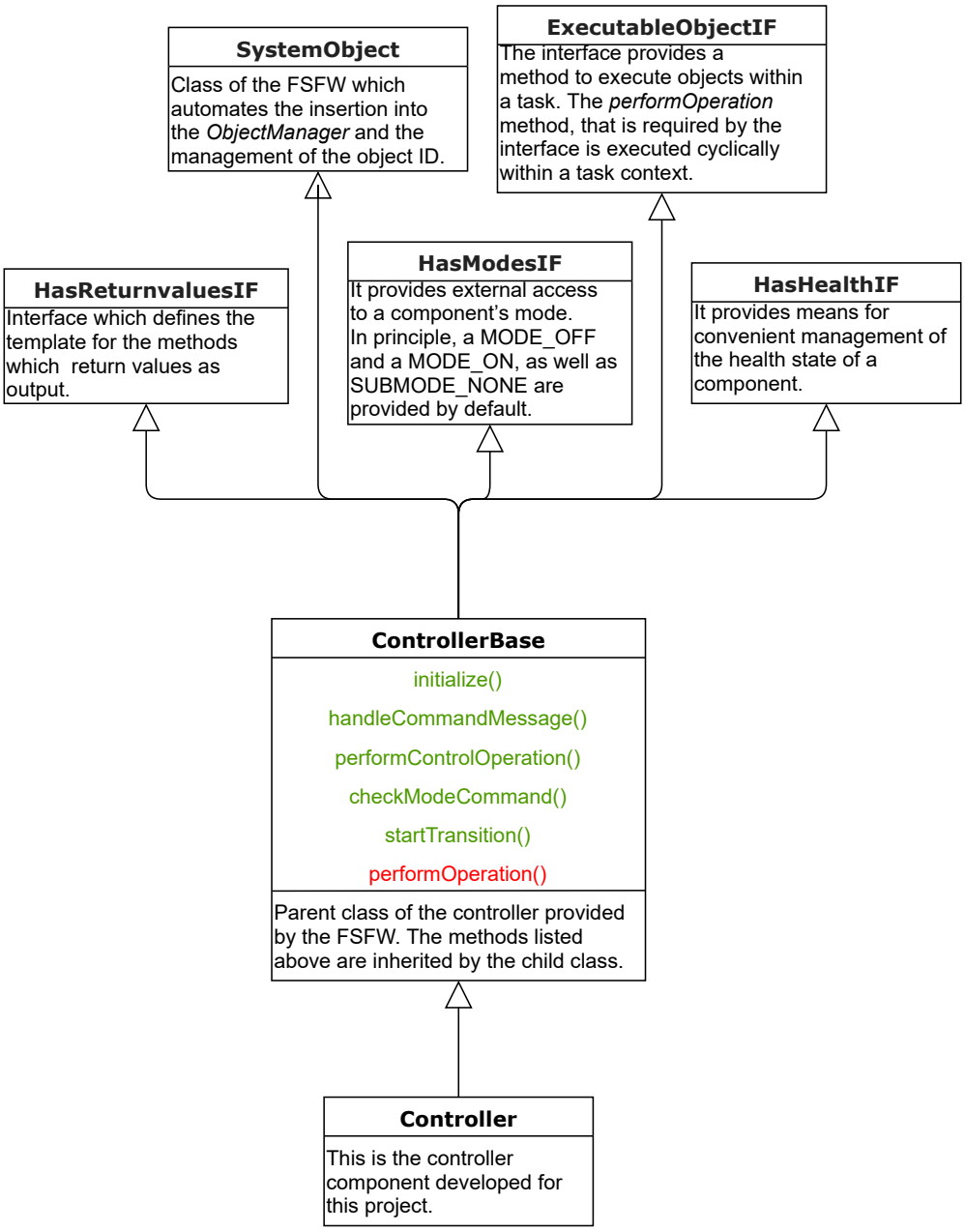

Figure 16: Thermal controller parent classes.

The *performOperation()* method has the structure defined in Fig. 17. Firstly, for each sensor instantiated, the *performHealthOp()* method is called. It is inherited from the FSFW temperature sensor component. It does some basic task of command queue handling. In the second part of the function, the own s*performOperation()* method of the FSFW *ControllerBase* class is called. It does some basic task of command handling as well. As already mentioned, the controller is required to perform its work autonomously with respect to the ground station. Therefore, it should not handle any command in normal conditions. The next step is the call of *performOperation()* method for each sensor component. The function is already explained in Sec. 4.1.

It recovers the temperature measurement and it checks the validity of these values through the *doChildOperation()* method. The data are handled by the sensor data set. As last step, the *performOperation()* method of the thermal component is called. Its working and objectives are already clarified in Sec. 4.3. It implements the control logic independently on each part of the satellite. The data collected in this routine are stored in the component data set. As mentioned in Sec. 3.1, this loop is executed periodically with a time interval of 3.2 seconds. This value is selected in order to synchronize the execution of the controller component with the device handler component. This is done to avoid problems during the reading of the measurements and the transfer of these data.

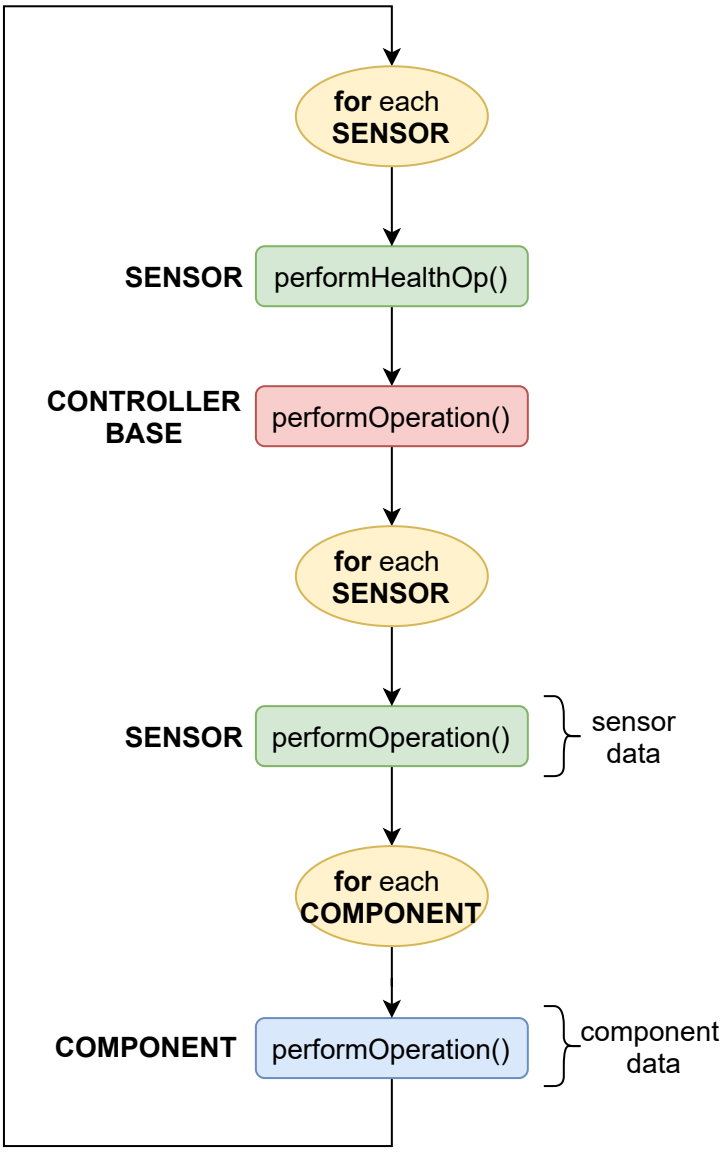

Figure 17: Thermal controller - *performOperation()* method flowchart.

4.5 Power Switcher

This is a particular component which is developed separately to assist the controller tasks. As already mentioned, when the heaters are commanded to switch on/off, the power switcher steps in and implement this action, communicating with the heater board through the device handler.
The heater board, as stated before, is an Arduino Micro. It has 12 switches, all switched off at the beginning. Each switch is described by one number between 1 and 12. Writing one number between 1 to 12, the correspondent switch is turned on. If I want to turn it off, I need to rewrite the same number. If it is written something which is not a number between 1 and 12, an error message is returned. In this case, the numbers are communicated through the interface with a simple ASCII format.
The initialization of the interface of the device handler has the same format of the sensor device handler. The only difference is that the communication is flowing on the opposite way. Therefore, instead of the routine *read()*, the function *write()* is exploited to send the switch commands. The power switcher is designed in order to know the switch states and update them after each action performed. They are stored in the data pool such that they can always be accessible.
The structure of this component is represented in Fig. 18.

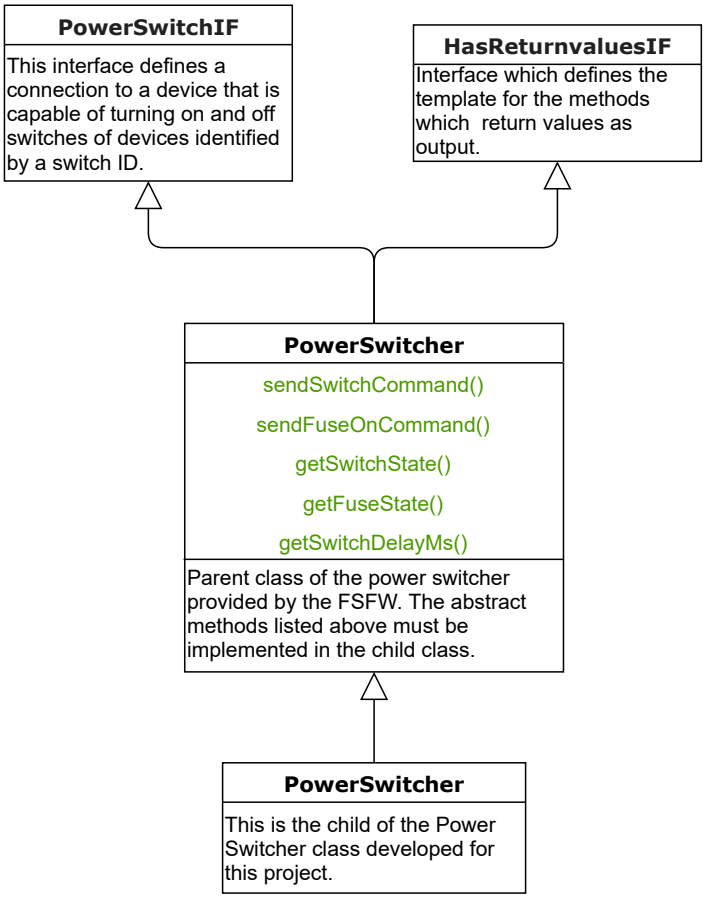

Figure 18: Power Switcher parent classes.

4.6 Software development and test plan

The development and testing of the thermal controller software component has a plan similar to the device handler (Sec. 3.5). It is resumed in the following steps:

1. Analysis of the framework components and the functionalities provided. Writing of the code directly within the FSFW, component by component;
2. Test the controller reading of measurement data which are saved in the data pool by the device handler;
3. Test the execution of all the functionalities of the controller components;
4. Development of the power switcher component and integration within the code;
5. Test of the whole integrated software component within the FSFW
6. Integration with the device handler component and test of the whole thermal control software within the FSFW employing the STUDIO ground system set-up;
7. Placement of the actual temperature sensors and heaters on STUDIO telescope, located at IRS clean room, and integration of the system toward a fully-operational system.

5 Tests and results

In this section, the tests planned and performed are described. The main criticalities encountered are here discussed and the solutions adopted are presented. The tests for the device handler and the controller component are described in two separate sections. It is remarked again that the device handler developed is related to the sensor board. The heater device handler is not completed yet, because the power switcher shall be concluded before and it's not responsibility of this work.

5.1 Device handler

The strategy for the development of this component with the related tests is presented in Sec. 3.5. The main criticalities are encountered in this part. They could be resumed in the following points, which are then extended here below:

- initialization of the interface;
- reading of the sensor data from the Arduino board with the test code out of the framework;
- reading of the sensor data from the Arduino board integrated with the FSFW.

The first problem discussed is related to the complexity of initialization of Linux serial ports. This includes the complexity of dealing with Linux environment and Arduino hardware. A key point is the definition of the parameters *VMIN* and *VTIME* described in Table 7. They are designed in order to control blocking of the *read()* call. In order to test the communication, the computer is connected to the Sensor board. The set-up is presented in Fig. 19. In Fig. 20, the Sensor board with the Arduino Micro is shown in detail. In order to simplify these initial tests, only the first temperature board with two temperature sensors is connected. This allows to reduce the complexity. The demonstration of a correct reading for a couple of sensors is enough to validate initially the software. Adding other sensors does not change the behaviour of the communication. For this reason, this same set-up is exploited also for the next tests until the end in which all the real components are integrated.

After the successful initialization of the communication interface, the device handler test code is written in Eclipse IDE environment. This leads to the second criticality described, which revealed to be the most difficult to solve. After a long period of tests, the code was reading a corrupted structure of data. It turned out that the most probable source of these errors was due to the software developed for the Arduino sensor board (Sec. 3.3.1). This problem revealed to be the biggest because it was a bottleneck for the progress of the whole project. Indeed, without having the possibility of reading correctly the data, it was impossible to complete the device handler and, even worse, to test the controller. Due to the delays in the whole mission planning, this situation has been tackled within this project even if it should have been not part of this work. As

pointed out by the Arduino software developer in [12], this version of the code has not completed yet the validation. Furthermore, the missing integration with the STUDIO hardware could have revealed the need to modify the software. After a deep analysis of the Arduino software, in particular with respect to the structure of data sent as serial output, many errors are found related to the storage of the measurement data in this structure and to the initialization of the default array of data for the empty ports. The sensor code is then fixed and improved in this project. It has been tested as well the long-term stability of the serial output. These tests on the device handler code has led, therefore, to the demonstration of the Arduino sensor board code. It still needs the validation in a simulated environment after the integration with the real prototype.

The third main criticality occurred during the tests of the device handler within the FSFW environment. Basically, the reading resulted perfect for the test code but it was always corrupted for the device handler component of the framework. This problem is related to the particular tasks execution within the framework. It is discussed in the polling sequence paragraph of Sec. 3.1. The reading with the test code in the Eclipse environment happens automatically in a synchronous way: the reading is done periodically, in a *while()* loop, coupled with the time period of the serial output. Contrarily, the reading within the FSFW is executed with a fixed period selected by the developer. It is uncoupled with respect to the Arduino serial output. The first solution tried is a tuning of the time periods of task execution. This is done experimentally by tests and it resulted in an improvement of the reading, which presented less corrupted values, but it was still wrong. The final choice for the execution period is described in Sec. 3.1. A logical solution is to increase the time of *read()* call to multiple of the serial output time period, to allow a sort of synchronization between sensors and the device handler. The second solution tried, which successfully led to fix this problem, is a change of the structure of the buffer array for data reading. As already explained in Sec. 3.3.3, it is chosen to extend the reading to an amount of bytes correspondent to three packets of data. It was tested before the reading with a total of bytes equal to two packets of data, nevertheless errors were still present. The objective of this modify is to allow the device handler to read at least one full packet of measurement data, from the beginning to the end. Indeed, the criticality was that the device handler wasn't succeeding to start the reading from the first byte of the serial output. Since the device handler is interested just in one whole packet of measurement data, sent periodically, a control loop is performed at the end of the communication interface component, in order to satisfy the device handler. The control loop is represented above in Fig. 9.

At this point, the first five steps of the plan described in Sec. 3.5 are completed. However, the set-up with STUDIO ground system and telescope, is not ready yet. It is planned to be done soon, when the thermal simulation is finalized and the equipment choice is completed. For matter of time, the results of these tests are not included in this document.

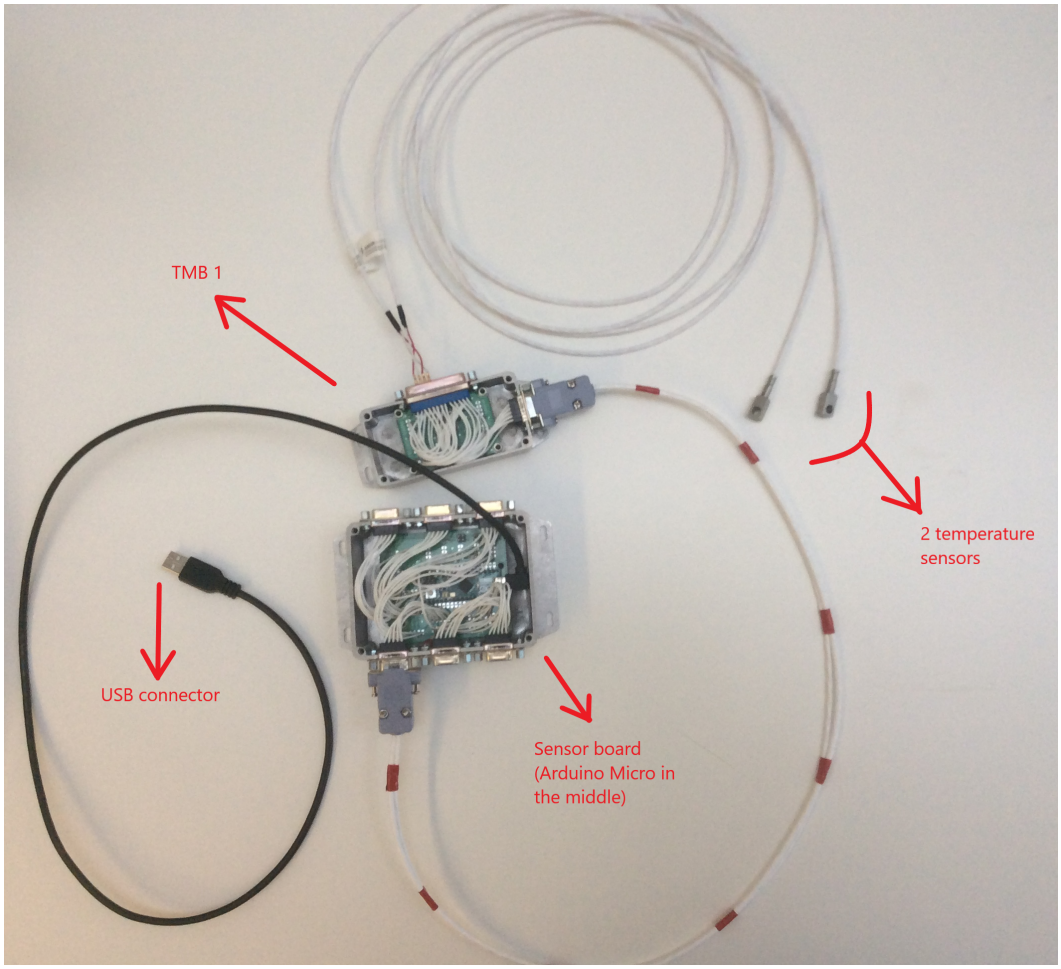

Figure 19: Sensor board set-up.

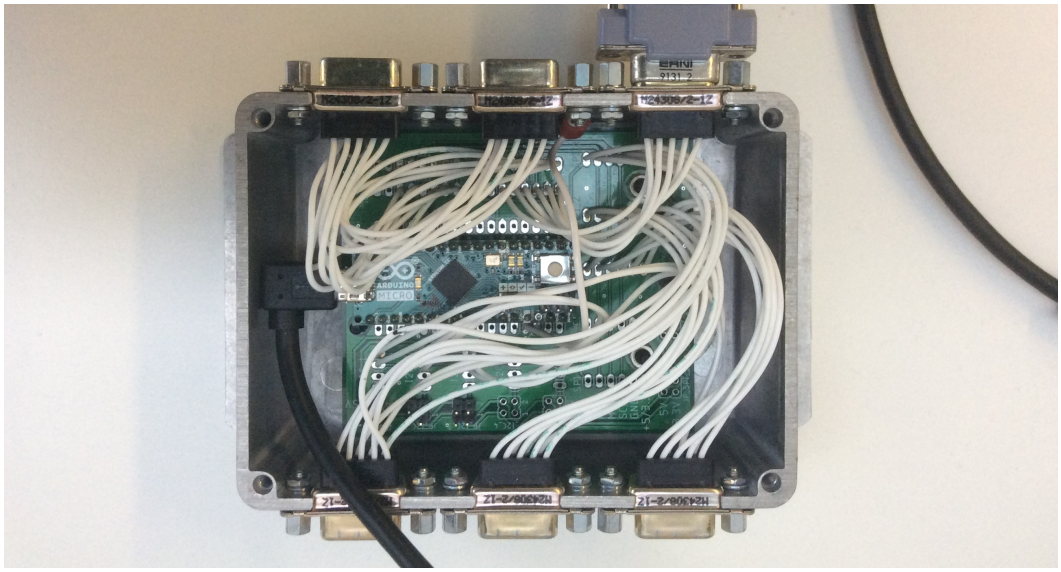

Figure 20: Sensor board.

5.2 Thermal controller

The steps for the thermal controller development with the associated tests are described in Sec. 4.6. This part of the work presented less problems during the tests even if the development of the code has been far more complex. The main problem presented here is that the tests on the controller has been delayed due to the errors on the device handler. After solving this criticality, as described in the section above, all the components exploited to build the thermal controller, along with the device handler component, are integrated within the FSFW and the whole execution is tested. Apart of minor errors, this test succeeded.
The power switcher though is built independently on this work. It is remarked that the firmware of its interface board is under development, and that is why the basic functionality of switching on/off is only tested within this project. These tests are performed through a digital voltmeter[21] as represented in Fig. 18. A detailed image of the heater board and of the switch board is given respectively in Fig. 22 and Fig. 23.
Further tests should be done regarding the power switcher integrated with the thermal controller component. The robustness of the thermal control with respect to different conditions must be validated. Tests with the real equipment, heating the sensors to different temperatures, shall be performed.
As already mentioned in the section above, the final validation with STUDIO ground system and telescope is planned to be done soon but the results will be not included in this document for matter of time.

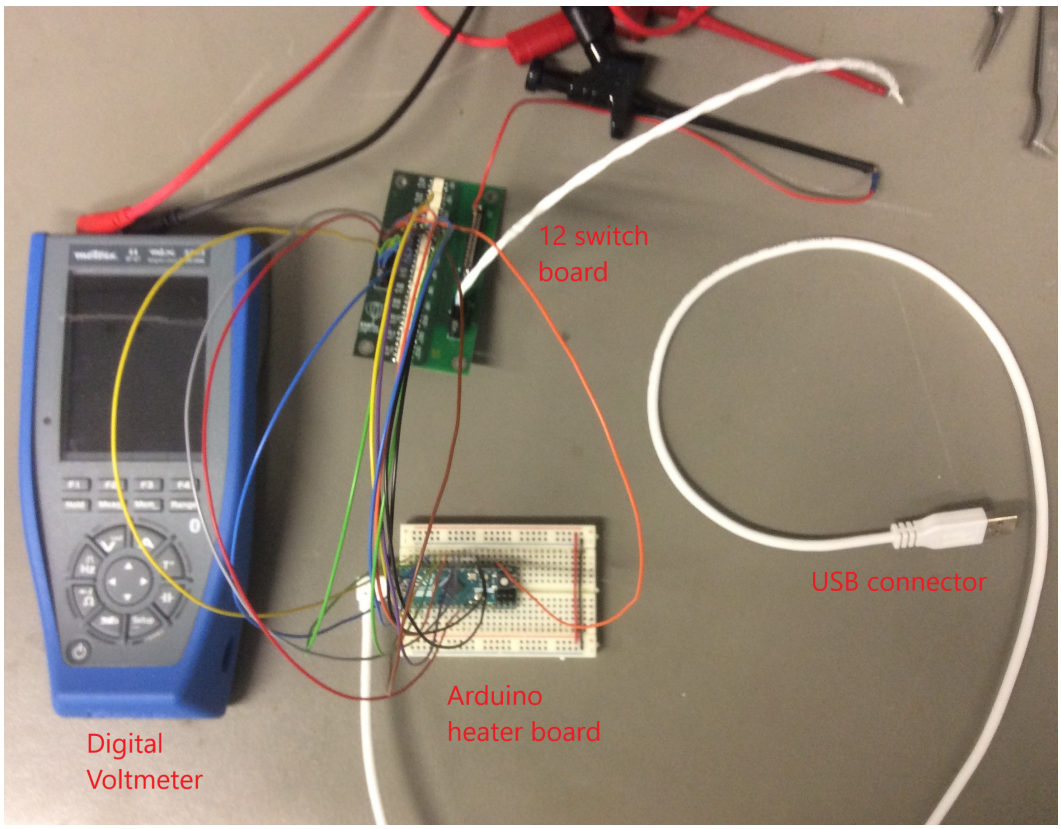

Figure 21: Power switcher set-up.

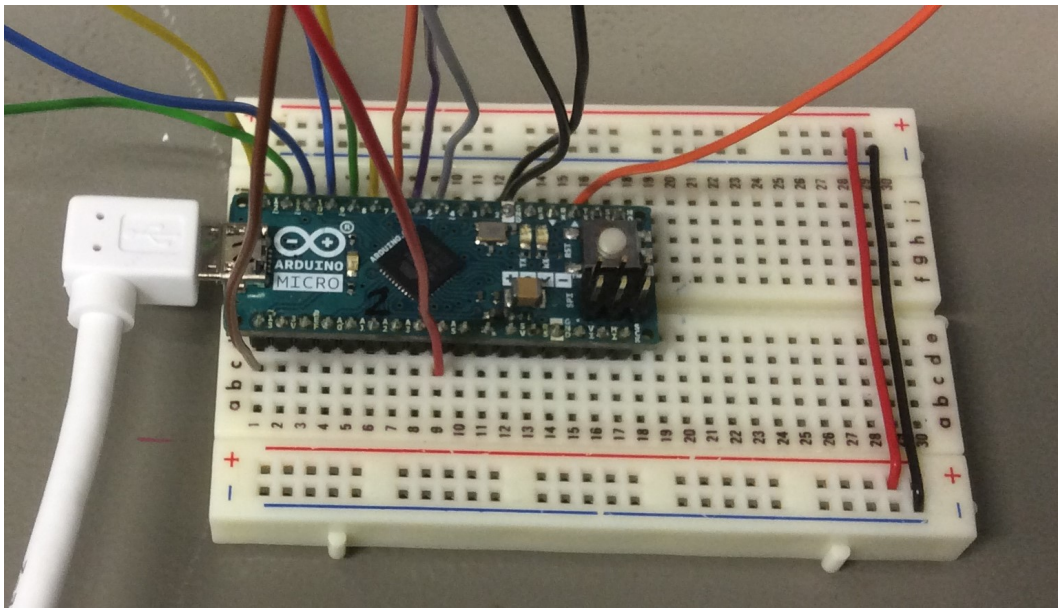

Figure 22: Heater board.

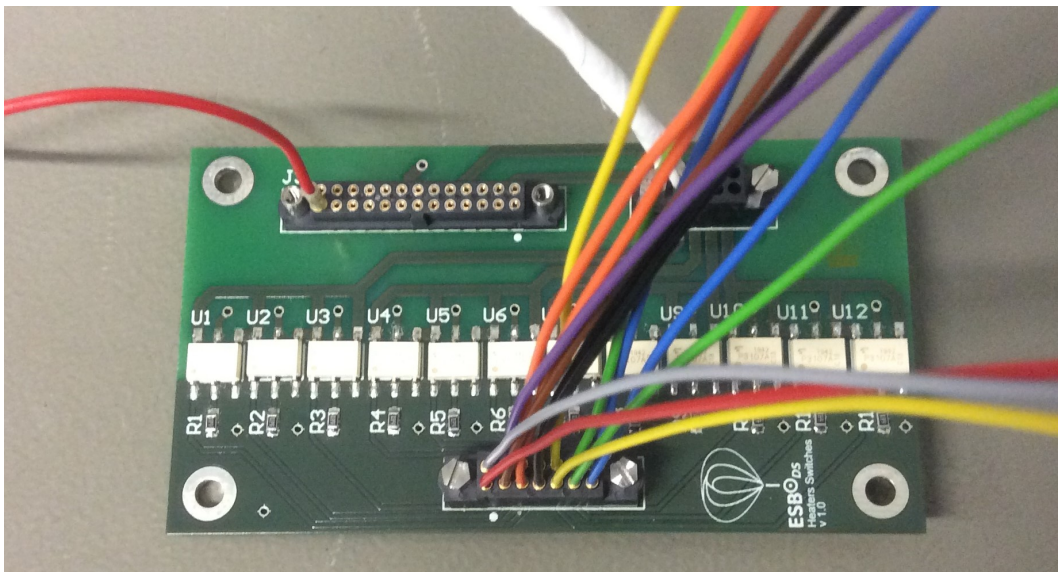

Figure 23: Switch board.

6 Conclusion

The main contribution of this work is the development of the thermal control software for the balloon-borne telescope on STUDIO prototype [1]. The whole work has been based on an existing component-based software framework: the Flight Software Framework. It is designed and built in order to satisfy the requirements listed in Sec. 2.3.1. The result can be resumed in the development of two main software components:

- the Device Handler;
- the Thermal Controller.

Additionally, the software of the Arduino sensor board has been corrected and improved in this work. It was a bottleneck for the proceeding of the tests, therefore it required to be fixed within this project.
The OOP within the FSFW is quite complex and difficult to comprehend. The framework is huge and it takes a lot of time to obtain a sufficient understanding of it, before starting to work. Furthermore, an high level of competences related to hardware and to software development were needed for this work.
The set-up of the communication interface, in particular, revealed to be the most critical part of this work. This is due to the complexity of creating a stable communication link between two different devices. This is usual for the set-up of new hardware communication interfaces. Anyway, this work allows to address the hardware dependency only on this specific communication interface component. In this way, the next missions could re-use the same software for the thermal control system, just adapting it to new requirements and hardware. When the communication interface for a specific hardware is set-up, within the FSFW, it becomes very easy to re-use it for other devices, as for the heater board in this project.
The programming skills achieved during this work are transferable on software development of generic aerospace systems and more. Therefore, the knowledge of space systems engineering could be combined with these software engineering skills to model completely a space system program, as for the thermal control system in this case.

6.1 Future work

In this paragraph, a few directions for the development and final implementation of this work are given. They are resumed in the following list:

- the thermal simulation of the prototype shall be finalized; consequently, the number and placement of sensors and heaters shall be selected and implemented in the thermal controller component;
- the hardware must be validated by means of tests in the thermal vacuum chamber;
- the complete power switcher component, after the development of the firmware of its interface board, shall be integrated within the whole software and tested;

- all the software components shall be integrated with the real hardware and validated with STUDIO ground system and telescope located at IRS clean room. These tests are already planned.

The operations needed to fully validate the software are therefore not much. Thanks to the software design choices, the implementation of final equipment within the thermal controller component is quick and easy. The final tests on the hardware and integrated system will identify the last problems or criticalities to work on, in order to improve and validate the software.

References

[1] I. o. S. S. University of Stuttgart. *European Stratospheric Balloon Observatory - Design Study*. Online; accessed 13 October 2021.

[2] U. Stuttgart. *Institut fur Raumfahrtsysteme (IRS) - Universitat Stuttgart*. Online; accessed 25 October 2021.

[3] J. L. Ortiz Moreno, T. Mueller, R. Duffard, et al. "ORISON, a stratospheric project". In: *41st COSPAR Scientific Assembly*. Vol. 41. July 2016, PSB.1-31–16.

[4] SSC. *Swedish Space Corporation: Home - SSC*. Online; accessed 25 October 2021.

[5] U. Tubingen. *Institut fur Astronomie und Astrophysik*. Online; accessed 25 October 2021.

[6] M. P. Institute. *Max Planck Institute for extraterrestrial Physics: Home*. Online; accessed 25 October 2021.

[7] I. de Astrofisica de Andalucia. *Instituto de Astrofisica de Andalucia - CSIC*. Online; accessed 25 October 2021.

[8] A. Pahler, M. Ångermann, J. Barnstedt, et al. "Status of the STUDIO UV balloon mission and platform". In: *Ground-based and Airborne Telescopes VIII*. Vol. 11445. International Society for Optics and Photonics. 2020, 114451Y.

[9] B. Bätz. "Design and implementation of a framework for spacecraft flight software". In: (2020).

[10] ESBO-DS. *Thermal Control Simulations*. Last version 20 June 2021. 2021.

[11] I. Aero. *ESATAN-TMS thermal modelling suite*. Online; accessed 14 October 2021.

[12] L. Willwand. "Design and setup of an in-situ measurement system for environmental and structural conditions during a scientific balloon mission". In: (2020).

[13] L. TECHNOLOGY. *LTC2983*. Online; accessed 27 October 2021.

[14] Baumer. *PT1000*. Online; accessed 27 October 2021.

[15] BOSCH. *BME280*. Online; accessed 27 October 2021.

[16] BOSCH. *BNO055*. Online; accessed 27 October 2021.

[17] Arduino. *Arduino Micro*. Online; accessed 27 October 2021.

[18] OMEGA. *Polyimide Insulated Flexible Heaters*. Online; accessed 27 October 2021.

[19] W. A. Campbell Jr, R. S. Marriott, and J. J. Park. "Outgassing data for selecting spacecraft materials". In: (1984).

[20] mbedded.ninja. *Linux Serial Ports Using C/C++*. Online; accessed 9 November 2021.

[21] Wikipedia. *Voltmeter*. Online; accessed 19 December 2021.