POLITECNICO DI TORINO

Master degree course in Electronic Engineering

Master Degree Thesis

UHF band channel module design for micro and nano modular satellites

Supervisor

prof. Leonardo Maria Reyneri

Second supervisor:

prof. Claudio Sansoé

Candidate

Enrico Sanino

July 2015

Acronyms & Abbreviations

ADCS	Attitude Determination and Control System/Subsystem
AraMiS	Architettura Modulare per Satelliti
AUS	AUtentication Server
BER	Bit Error Rate
COTS	Commercial Off The Shelf
CRC	Cyclic Redundancy Check
DET	Department of Electronics and Telecommunication
ESA	European Space Agency
FCS	Frame Check Sequence
FSK	Frequency Shift Keying
GENSO	Global Educational Network for Satellite Operations
GSS	Ground Station Server
ISIS	Innovative Solutions In Space
ISS	International Space Station
LEO	Low Earth Orbit
MCC	Mission Control Client
MCU	Micro Controller Unit
OBC	On-Board Computer
OBRF	On-Board Radio Frequency
P-POD	Poly-Picosatellite Orbital Deployer
PA	Power Amplifier
PDB	Power Distribution Bus
PiCPoT	Piccolo cubo del Politecnico di Torino
PIFA	Planar Inverted-F antenna
RSSI	Received Signal Strength Indicator
SEU	Single Event Upset
SMEs	Small Medium Enterprises

SNR	Signal to Noise Ratio
TCP	Telemetry Command Processor
UML	Unified Modelling Language

Acknowledgement

This work would never be completed without the help of my tutors Reyneri Leonardo and Sansoé Claudio, as long as the PhD students and colleagues, which I would like to thanks for their deep patience and availability.

I would like to thanks all my friends, which helped me in keeping the right mood and motivation during the thesis period, concluded with a big personal improvement.

Special thanks are going to my mother and my brothers, which have always supported me in the difficult moments.

A final special thanks for everything else goes to my father, which is farther than where any man made satellite will never be.

Summary

In Department of Electronics and Telecommunications (DET) of Polytechnic of Turin there are projects involved in small satellites, CubeSat compatible but not limited to it. A project started in 2007 called AraMiS, which stand for Modular Architecture for Satellites in italian, is born after the first CubeSat compatible spacecraft developed in Turin, called PiCPoT. The main philosophy in AraMiS is the modularity of the system in all its aspects.

This thesis work is focused on the engineering phase of the AraMiS telecommunication system module, which will be applicable on every AraMiS spacecraft, including the CubeSat version, named as AraMiS-C1. To accomplish this, it has been developed the whole set of system's use cases, the basic firmware design along with a new revision of the telecommunication hardware, taking into account the already defined constraints and requirements of the AraMiS spacecraft telecommunication system. The real challenge of this work is to bring into practice all the considerations made for reliability purposes, and in part adapting them to obtain a final, modular and reliable design.

In chapter 1 will be presented an introduction to the small satellite concept, letting the reader understand more in depth the actual development of these small spacecrafts, and provides the basic description of how a modular architecture like AraMiS can be the key to keep high dependability on low costs, even on more complex systems which are not limited to CubeSat environment only. Moreover, will be described briefly the ground telecommunication network which will going to be used for low cost and university small satellites.

In chapter 2 is provided an UML introduction in order to understand better the notions adopted in this work.

In chapter 3 is described the starting point of the system. Therefore are provided the specifications to comply when developing the entire system. It is introduced the concept of the OSI Stack, and as a consequence are provided the telecommunication protocols adopted to encapsulate the frame, like the AX.25, and the protocols of the content of the frame, called AraMiS protocol.

In chapter 4 are shown the constraints which are needed to be taken into account when developing a telecommunication module. Finally is implemented the whole set of the use cases of the system,

essential to implement the specifications to the module which will be designed.

In chapter 5 will be shown the final implementation of the system. It is devised an affordable power handling and a new sensor unit sub-system. The OBC now have more control on the OBRF hardware, to handle better the latch-up protection. More sensors are used in order to control the different organization of the power supply sub-system. Therefore the sensors sub-system is completely redesigned. All the hardware library is then reorganized updating the components and creating reusable locks, to comply with the AraMiS philosophy.

In chapter 6 is going to be described the firmware designed starting from the previously described use cases and the adopted hardware. The algorithms are devised starting from sequence diagrams and finite state machines, mainly for being compliant with the AX.25 radio amateur protocol in an affordable way. These algorithms are then implemented to correctly handle the RF streaming and also described, when necessary, with sequence diagrams. All the on-board and the OBC communications are also integrated with the AraMiS software modules already present in the AraMiS library. Are also integrated the housekeeping functions and the transceiver drivers, and are devised all the procedures to handle correctly the digital interface of the RF circuitry. The software is written in C++.

In chapter 7 are analysed the possible physical constraints in order to achieve a reasonable placement criteria of components on the PCB. Therefore, after a thermal rough worst case analysis of the critical components, is shown the final PCB implementation.

Contents

Chapter 1

Introduction

1.1 Small satellite concept

The progress of technology in electronic and software fields, allows a huge reduction in terms of costs in majority of designs. This allows a cost reduction on launching vectors in space environment too. For that reason the interest from universities and SMEs on building their own spacecraft is grown. Moreover, the investments can be reduced a lot when using the ready-to-use COTS components, with a less cost and powerful elaboration capabilities allowing to execute more complex and redundant algorithms. Furthermore, still due to the COTS elements, the hardware can become redundant as well. Affordable launches can be achieved by grouping more satellites and making them small, namely small-satellites, from more universities and SMEs, in a single launching vector, spreading the costs. They can also be launched "piggyback", using excess capacity on larger launch vehicles.

Improvements and documentation can grow easily under common standards. The first idea of standardizing these small spacecrafts is born between 1999 and 2001 with the nano-satellite CubeSat, developed by California Polytechnic State University in collaboration with Stanford University. The CubeSat having a starting dimension of (100x100x100)mm^3 has evolved as CubeSat Standard [1] referred to a starting point of 1 unit size (known as 1U), which can be increased along one axis. It can be stated that the state of the art of small, low cost satellites is represented by this standard. As a result, nowadays is possible to make students able to work on complex systems and become familiar with interdisciplinary problem-solving, as a result of deep cooperation among different engineering departments.

CubeSat specifications are defined to solve some high-level issues, for example the simplification of satellite infrastructure to produce a workable low cost spacecraft, by standardising the design of

pico-satellites. Then defines the encapsulation of the obital deployer interface, in order to remove the re-designing costs that would be needed if a different interface would be required every time. The minimum size, *1U*, of 100mm side, can grow on one axis dimension by creating satellites of 1U, 1.5U, 3U or 3U+ sizes. These pico-satellites [5] are called from here in their general classification miniaturized satellites or small-satellites, for sake of simplicity. Heavier spacecrafts which can differs from CubeSats, as mentioned later, are also called small-satellites since are still under the threshold of 500Kg.

A standardized CubeSat Poly-Picosatellite Orbital Deployer (P-POD) (figure 1.1) has been designed to deploy these small satellites with a face of 100x100mm^2. This P-POD is still developed at California Polytechnic State University. P-PODs are mounted to the launch vector and they carry CubeSats into orbit until the deployment command is received from the launcher. Such launchers can be used also on the International Space Station (ISS), where in picture 1.2 are deployed a pair of 3U+ CubeSats. Such a structure must avoid any possible collision among the small spacecrafts during the deployment procedure, so, once the aperture has been opened, a piston-spring pushes outside all CubeSats which are kept separated each other by means of additional intermediate springs, if are more than one inside the launcher.

Figure 1.1. P-POD launcher

A lot of nano-satellite CubeSats compatible designs are developed in Europe. Few of them are the AAU spacecraft developed at Aalborg University in Denmark, the NCube designed by four Norwegian universities, PicPoT and AraMiS-C1 developed at Polytechnic of Turin. Not to mention the University of Wurzburg in Germany, the University of Rome La Sapienza and the University of Trieste. Here will be presented a more detailed description to the PicPoT, and then the new concept of the AraMiS project, born after the PicPoT project.

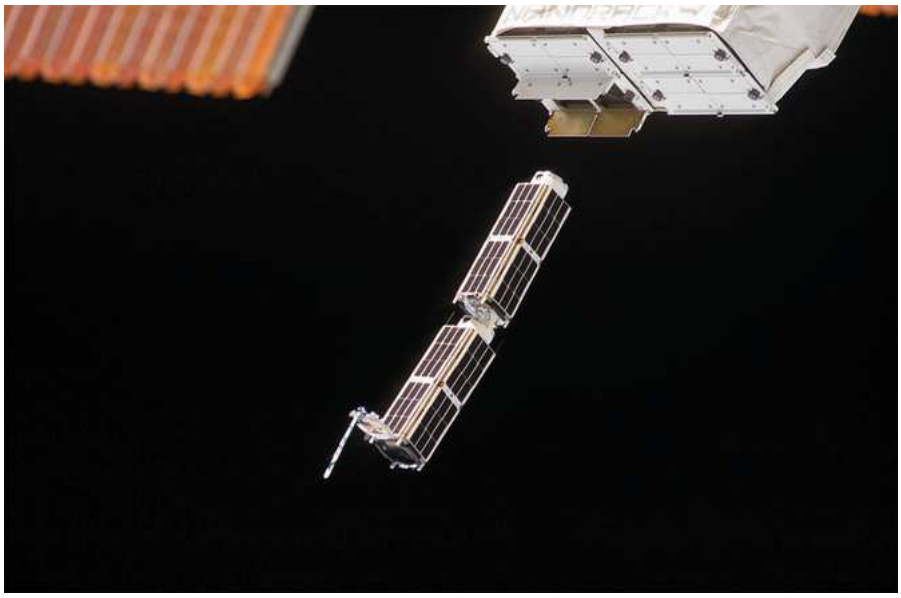

Figure 1.2. Small Satellites deployed from the ISS [4]

1.2 PiCPoT CubeSat

The Department of Electronic and Telecommunication (DET) here at Polytechnic of Turin, has developed its first nano-satellite called PiCPoT [2], which was intended to be launched together with other university and military satellites by a DNEPR Launch Vehicle rocket in July 2006, which unfortunately couldn't deploy due to a launcher failure.

The set of specifications was:

- Cubic shape with 13 cm side
- Mass equal to approximately 2.5 Kg
- Power in TX-mode lower than 1.5 W
- At least 90 days of life
- LEO target
- COTS electronic components
- P-POD launcher compatible

Initial mission requirements were to verify the reliability of COTS components in space applications, to take pictures of Earth from space, to exchange data with the ground station and to study the behaviour of GPS for LEO purposes.

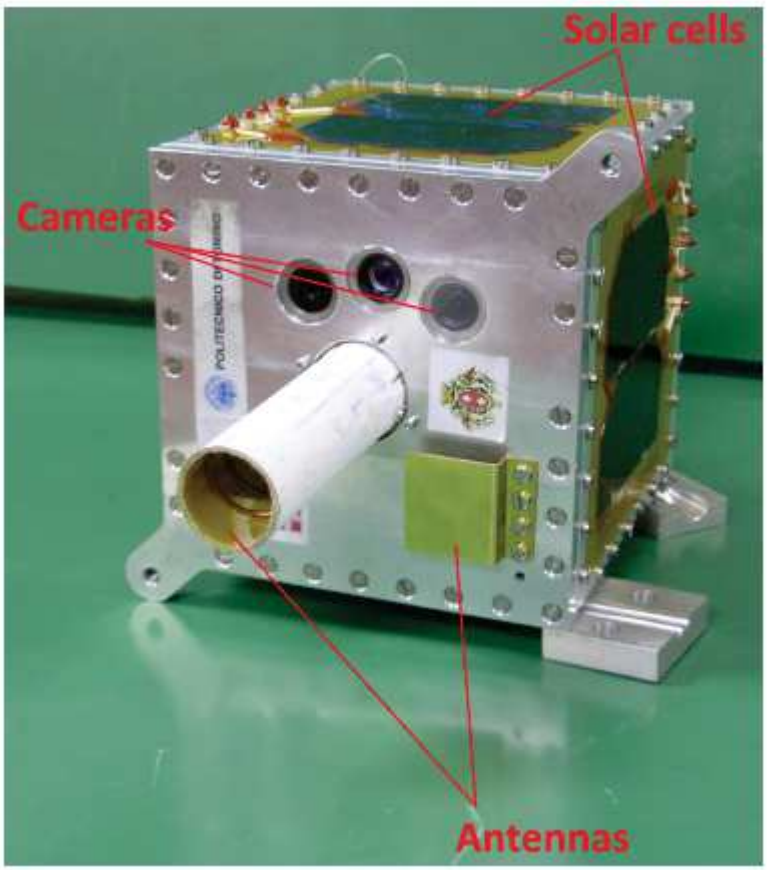

Figure 1.3. PiCPoT

The satellite incorporated two hot-redundant systems. Each one with power management, latch-up controller, housekeeping, telemetry and telecommand facilities, one optical payload with three multispectral camera-systems with JPEG compression and PAL encoding. It was composed of 5 interacting processors, two independent half duplex RF links (one at 437 MHz with a dipole antenna and the other at 2.44 GHz with a Planar Inverted-F antenna or PIFA), 5 solar panels which are covering 4 outer faces and finally, 6 battery packs. Was tested to be fully functional before launch. Moreover, a set of kill-switches was adopted to ensure the electric isolation of the satellite during launch, increasing the dependability, as required by the CubeSat standard. The project was including a ground station placed on the roof of Polytechnic of Turin.

Redundancy was the key-idea in PiCPoT project, although the failure of a single component could happen, a graceful degradation is possible. For instance, the telecommunication subsystem was based on two different physical channels (different frequencies and antennas) and also the related processing units were differentiated: one was based on Chipcon CC1010 transceiver and it handled a 9.6 Kbps data link with output power equal to 35.7 dBm, while the other was based on

MSP430 microcontroller and it handled a 10 Kbps data link with an output power of 30.8 dBm.

1.3 AraMiS

AraMiS, italian acronym for Modular Architecture for Satellites, is a project born at DET in Politecnico di Torino in 2007 and still going on, which goes beyond the CubeSat concept and aims to achieve a true modular architecture [3]. The goal of this project is to be modular at a software, hardware and mechanical levels. This modularity allows to reuse the design on more missions, i.e. more times and/or with different dimensions and power requirements, all with the same designs which are then already qualified and tested. The final user then should be aware only on placing the designs on the required shape and design the payload, which will be the only mission dependent design. Combining the designs in different shapes can lead to a different satellites which are already tested, lowering the total time to launch and designing costs.

Under this philosophy, can be accomplished different missions involving different spacecrafts, from CubeSat sized to larger ones, with the minimum effort. In picture 1.4 are presented various combinations of a one-time designed module, which is combined in different ways, giving an idea of the AraMiS mechanical modularity. A single square module is also called *tile*. This modular concept is adopted by the electronic point of view also. Most of the internal subsystems are developed in such a manner they can be composed together. For example, the power management subsystem in conventional missions is designed to get maximum solar power, by placing solar cells on all the available surfaces. But since their number can be different in various missions, a redesign will be required each time. This new modular approach instead makes use of a standard module, as can be seen in figure 1.4 which can be replicated many time to fit mission requirements [6].

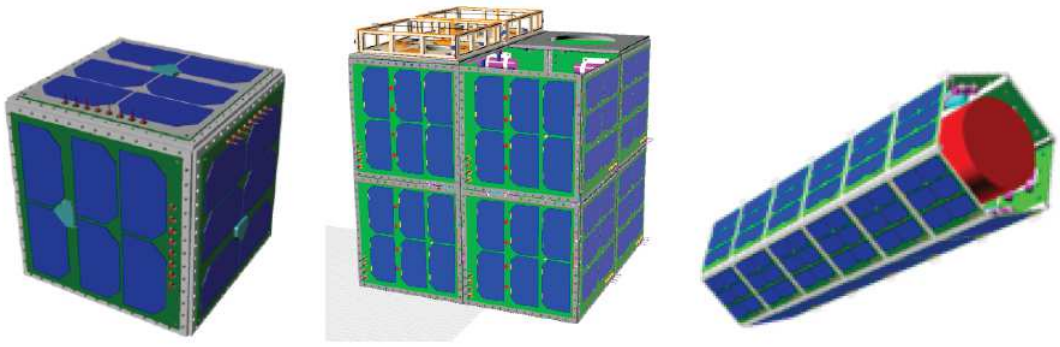

Figure 1.4. Different AraMiS architectures

In the CubeSat environment, is under development the AraMiS-C1, which is composed by 6 tiles of reduced size to follow the CubeSat standard, fixed a cubic aluminium skeleton. Tiles have both electrical and mechanical functions, where in the inner part are placed the various processors and

the outer part contains solar panels or antennas, depending on the type of tile. In this way there is a lot of room for the payload, even including the batteries and various actuators. Here are now presented the various main subsystems of the AraMiS, which combined together can bring to a complete satellite system.

1.3.1 Mechanical subsystem

Is a backbone of the spacecraft, whether it is CubeSat or not, the functions are the same. It is used to keeping combined together the various tiles, giving to them also additional mechanical strength and radiation protection. Made using aluminium, these chassis are composed by square rods on which are fixed with screws on it thin panels, which are carrying the telecommunication or power management subsystems.

The number of these tiles mainly depends on satellite size and power requirement. This provides a degree freedom to mission designers since size and generated power can be increased by simply adding more modules. Since tiles are used on the spacecraft sides, there is space inside for payload, batteries and any other additional required object. In figure 1.5 are depicted some AraMiS chassis.

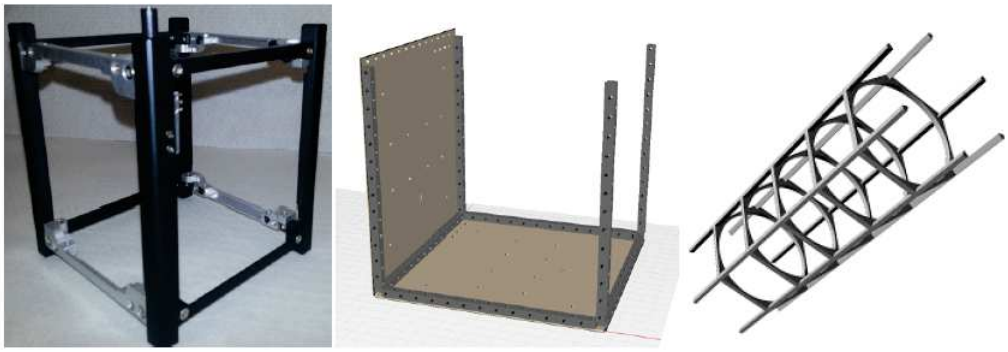

Figure 1.5. Three types of AraMiS chassis

1.3.2 Power management subsystem

It is is responsible for generating, storing and delivering power to all the other satellite subsystems and for itself. It provides various voltages according to a well defined protocol, and a maximum limited power per subsystem, therefore the more power management subsystems, the higher the power available, along as a replicated and then fault tolerant solution.

Conventionally, power management is mission dependent which requires ad-hoc development for the specific needs. This tends to increase overall system cost and testing time. For this reason the AraMiS project uses modular power management system that can be adapted for various missions.

Figure 1.6 shows different solar panels of AraMiS satellites, where specifically for the AraMiS-C1, on the other side of the PCB, are present the power management controls and the on-board computer (or at least one instance of the redundant architecture). In the C1 version, the project related to this subsystem is called 1B8_CubePMT, as will be shown later.

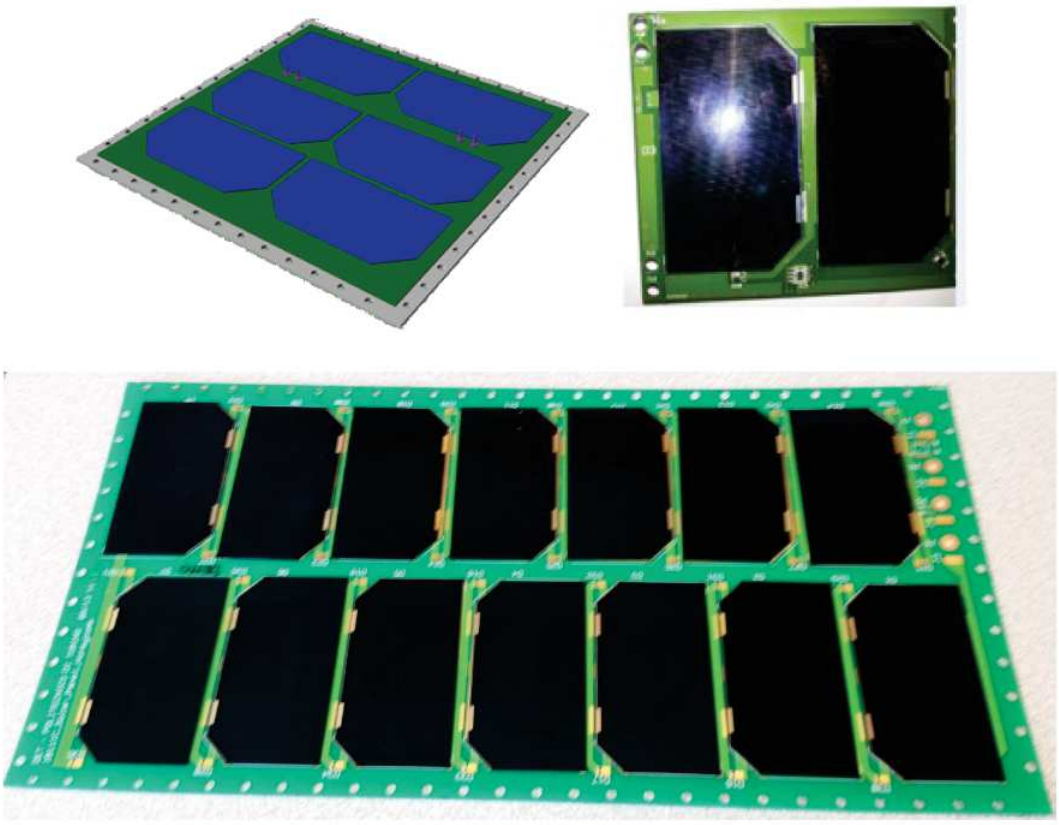

Figure 1.6. AraMiS power management tiles

1.3.3 Telecommunication subsystem

It follows the same modularity concept. There is a basic telecommunication tile that is provided in a standard AraMiS satellite. In case of special applications, dedicated tiles, like in Figure 1.7, can be added to meet mission criteria.

This module is used to receive command and control packets from ground and send back commands response, telemetry, status and beacon data. The bandwidth needed to exchange this kind of information is low, so the RF link is designed for low speed and low power. The module has been designed using COTS components. There are two different frequency bands used for satellite and

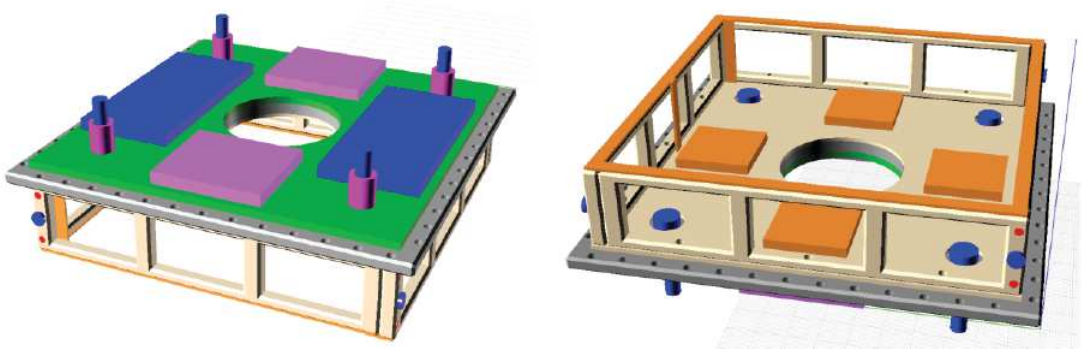

Figure 1.7. AraMiS telecommunication tiles

ground communication, i.e. the UHF 437MHz and the SHF 2.4 GHz band. To reduce occupied bandwidth, both channels are implemented using half-duplex protocol, sharing the same frequency per channel, for downlink and uplink. Any module can be shaped and reused to fit in a tile which can be different, with a very little effort since the design is already verified.

The processing capability of this tile is related in interpreting some commands from the OBC and generating the header data for the packets to be transmitted, and reading the payload from the received ones, as it will be described later. In the C1 version, the project related to this subsystem is called 1B9_CubeTCT, as will be shown later.

1.3.4 OBC Tile computer subsystem

Also called On-Board Computer (OBC), is composed by redundant MSP430 microcontrollers and, except for AraMiS-C1, also FPGAs. The firmware modularity and hardware abstraction layer allow to easily implement the OBC capabilities also in tiles with a microcontroller that is not heavily used, therefore having computation capability in excess. Under this point of view, the AraMiS-C1 uses the power management tile also as OBC.

Some of the key responsibilities performed by OBC includes:

- Creating and transmitting (by Transceiver board) Beacon packets,
- Decoding and executing commands,
- Executing attitude control algorithm,
- Storing housekeeping data,
- Controlling Payload sub-systems.

1.3.5 Attitude determination and control subsystem

This subsystem is mainly responsible for sensing and modifying satellite orientation for keeping the tile subsystems pointing at their targets, for example keeping the antenna toward Earth. In araMiS-C1 is integrated in the PMT tile.

Attitude control can be performed in passive or active way: passive attitude control is usually achieved by mounting a permanent magnet in the satellite which acts as a compass in the Earth magnetic field. This system is extremely simple and consume no power. The main drawback is lack of spin control due to the variable Earth magnetic field. Active control is performed using controlled actuators that modify satellite attitude on OBC commands. In AraMiS, attitude control is automatically performed by the satellite using magnetorquer and reaction wheels, as shown in figure 1.8.

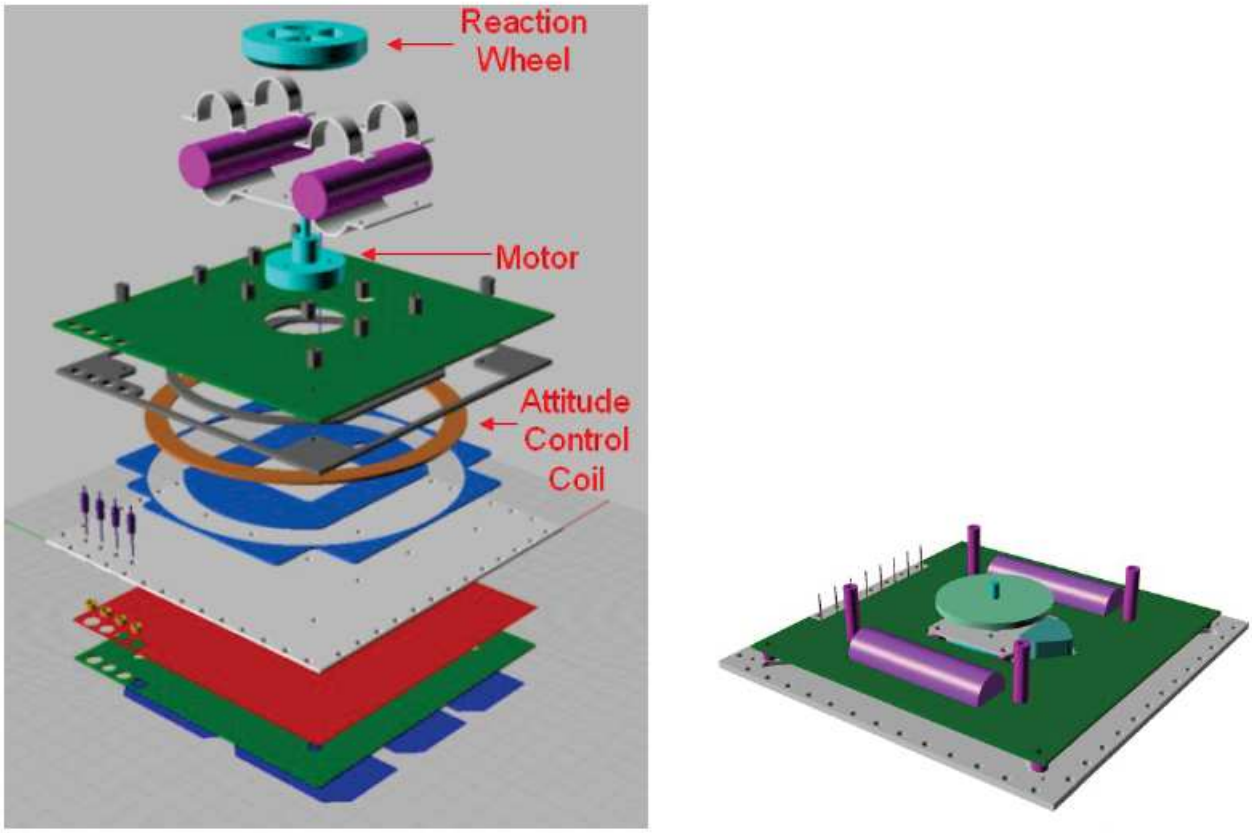

Figure 1.8. AraMiS ADCS

For attitude determination, three types of sensors are used: magnetic, spin and Sun sensors. These sensors consist of COTS components which were selected on the basis of small dimension,

light weight and low power consumption while achieving better performances.

1.3.6 Payload

The payload is heavily mission dependent and the AraMiS architecture is developed to allow high flexibility on it. The only requirements for payload are the compatibility with the power distribution bus (PDB) and data bus. This implementation can differ from AraMiS-C1 and other bigger AraMiS satellites. An exmaple of payload implementation is shown in figure 1.9.

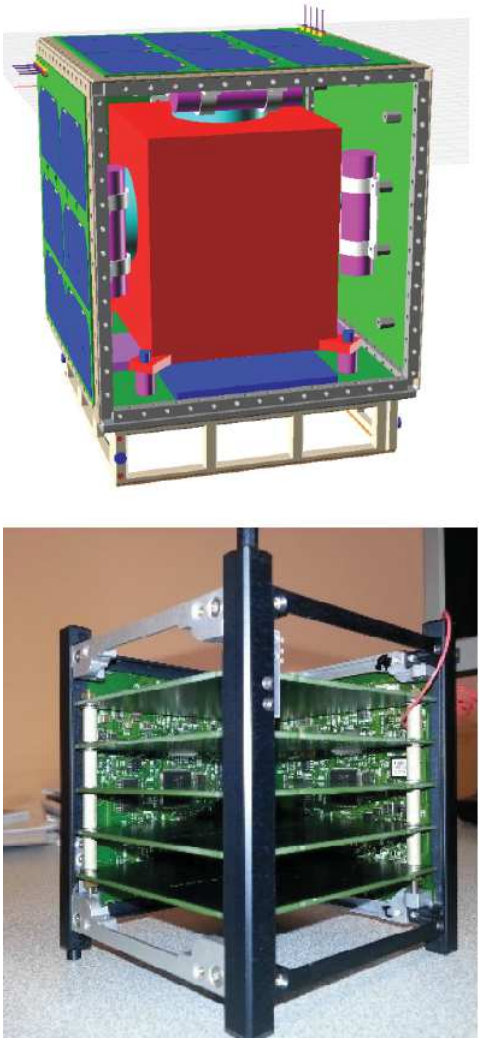

Figure 1.9. From top to bottom. Rendering of generic AraMiS payload; a picture of possible payload implementation of AraMiS-C1 payload.

1.3.7 AraMiS-C1 CubeSat

As briefly mentioned before, the C1 version is a 1U size CubeSat implementation of the AraMiS architecture. This cube is composed by two main tile modules, 1B9_CubeTCT and 1B8_CubePMT.

The 1B8_CubePMT covers four sides by four identical instances of that tile. Each PMT tile mount solar panels on the exterior PCB, while in the internal side share with each others a combined power management, attitude control and computing subsystem. The remaining two sides are the 1B9_CubeTCT, where on the external side are mounted, one for each side, a deployable UHF antenna and a patch SHF antenna. Each TCT tile mount these two band frequencies, and takes care of decoding and encoding commands from or to Earth, and communicate with the OBC. Each tile contains in its turn a modular design: for example, if the TCT tile contains two different channel modules (UHF and SHF) to allow redundancy, then only one tile can be used. Inside the satellite there is room for batteries and payload. Once deployed from the P-POD, the cube will expand four antenna baffles, which are part of ISIS deployable antenna system.

To keep a simple design, maintenance, manufacturing, testing and integration, the modular architecture apply. Here major bus functionalities are split over a number of identical modules, which are then simply placed in a proper order on the tile (in the same PCB). Various modules are dynamically connected with each other, exchange data and power in a distributed and self-configuring architecture. Its flexibility is due to the standardised interfaces between the various components. If a substitution is needed, a single module is changed without affecting the rest of the design, by simply testing a part that new module.

AraMiS-C1 is made by assembling a number of tiles developed at Politecnico di Torino, as detailed further, plus a few commercial off-the-shelf subsystems from AraMiS-C1 ISIS's CubeSat shop. Photograph of 1U AraMiS-C1 with four 1B8_CubePMT and two communication tiles is shown in figure 1.10.

The AraMiS-C1 is designed to be functional over a period of two to three years on an orbit in the 500 km range, but even lower orbits with higher atmospheric drag that will guarantee a few months in space are acceptable for our purposes. Obviously longer orbital life (at least one year) will be more appropriate for the scientific objectives of the mission.

AraMiS-C1, where C1 is related to CubeSat 1U, is structured at high level as described in diagram 1.11. Here the modules are listed under 1B classification, from 1B1 to 1B7. Every sub-project of the main one (project 1, or ARAMIS in the image) have a proper letter, here from 1A to 1C, then the subdivision become in numbers, i.e. 1B1, 1B2 and so on recursively. The previously mentioned tiles, numbered 1B8 and 1B9, are containing a combinations of projects from 1B1 to 1B7, building up a complete tile. This low cost university satellite is designed to communicate with Earth using a particular educational network, called GENSO.

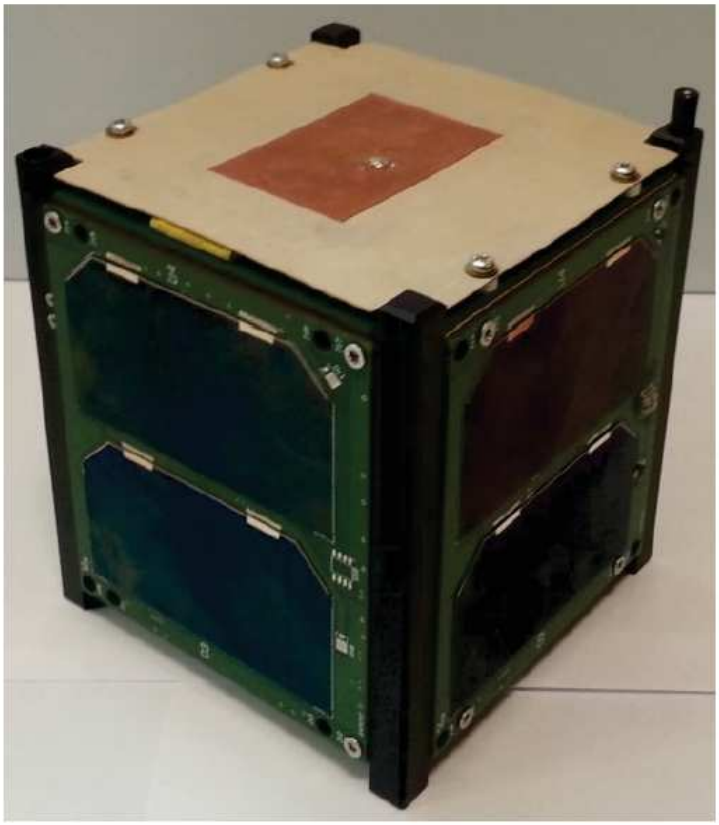

Figure 1.10. AraMiS-C1 CubeSat

1.3.8 Antenna

All of the telecommunication tiles and the 1B9_CubeTCT tile in the AraMiS-C1 are using antennas that can be designed with the tile or can be a complete external system. In AraMiS satellites which are not CubeSat, a complete antenna design has been performed. The antenna can be integral part of the extenal side of the tile as seen before in figure 1.7, or an external piece, which is an antenna tile only and it is connected through a coaxial cable to the telecommunication module. The AraMiS-C1 uses a patch antenna for the SHF channel, integrated with the tile, already shown before in figure 1.10 in the top of the satellite. Another version of an internally designed AraMiS antenna is shown in figure 1.12.

The UHF band subsystem of the telecommunication tile of the CubeSat uses an external antenna connected through a coaxial cable, implemented as a tile to be attached on a side of the cube. This antenna is shown in figure 1.13.

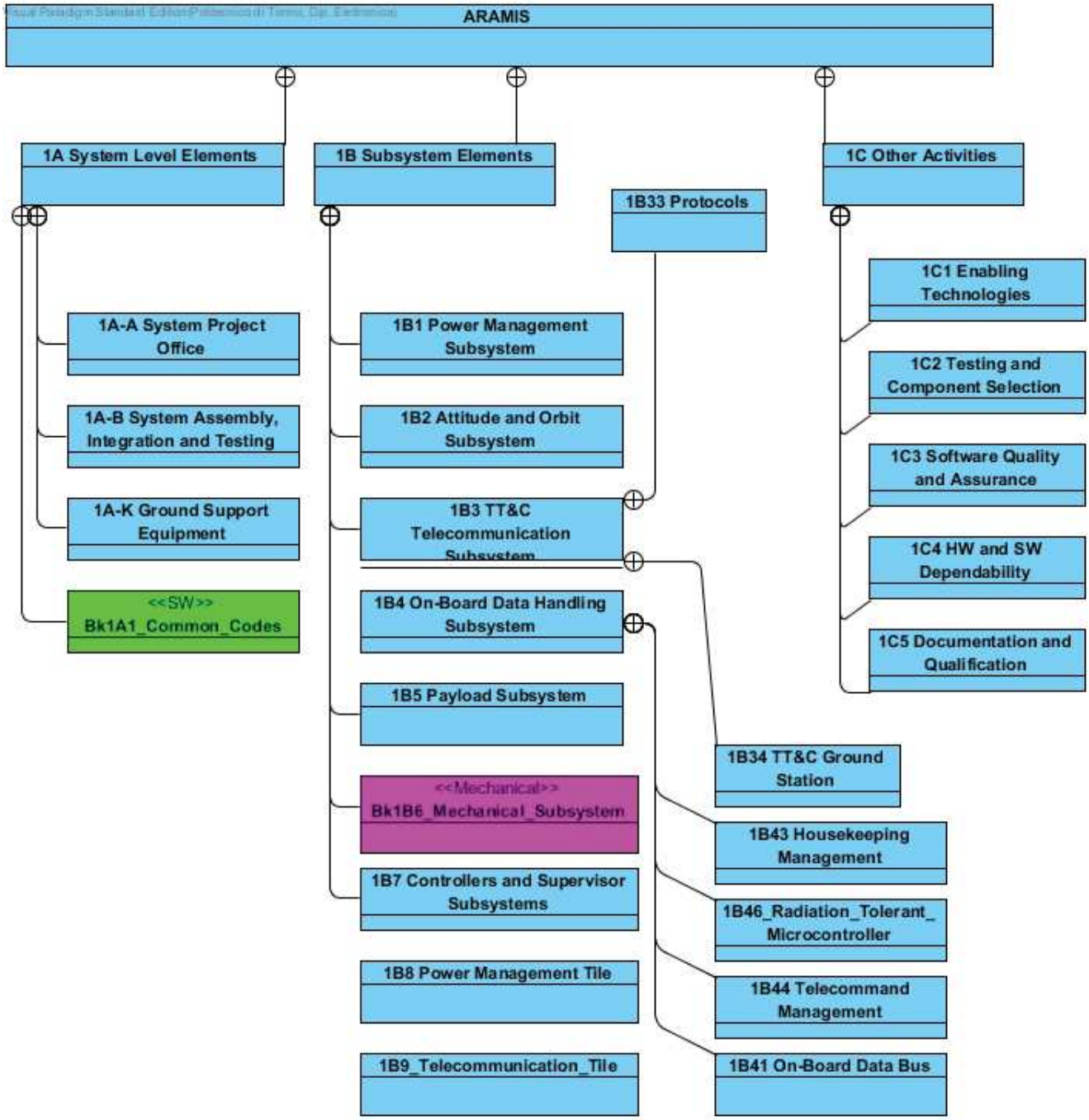

Figure 1.11. AraMiS project organization

Figure 1.12. AraMiS antenna

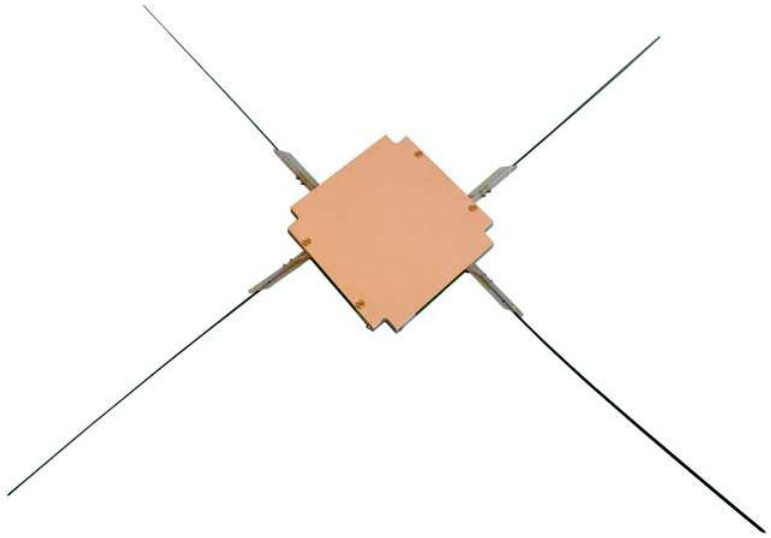

Figure 1.13. AraMiS-C1 UHF antenna

1.4 Earth network GENSO

GENSO (Global Educational Network for Satellite Operations) is a project approved by ESA in 2006. Its purpose consist in a workaround to the problem of the limited satellite visibility to the owner university, since in LEO these windows are around 20 minutes per day. The workaround is to tunnelling over the Internet the data exchanged under another university visibility window, which is part of GENSO. In this way, all the participants to GENSO project are using and providing resources, extending the visibility to potentially 24h per day, with the possibility on relying on radio amateur stations. This project offers the capability to plan and schedule the use of ground station resources, to predict the trajectories of spacecraft over the ground station and to automate

tracking the satellite during a pass. The AraMiS project will use GENSO.

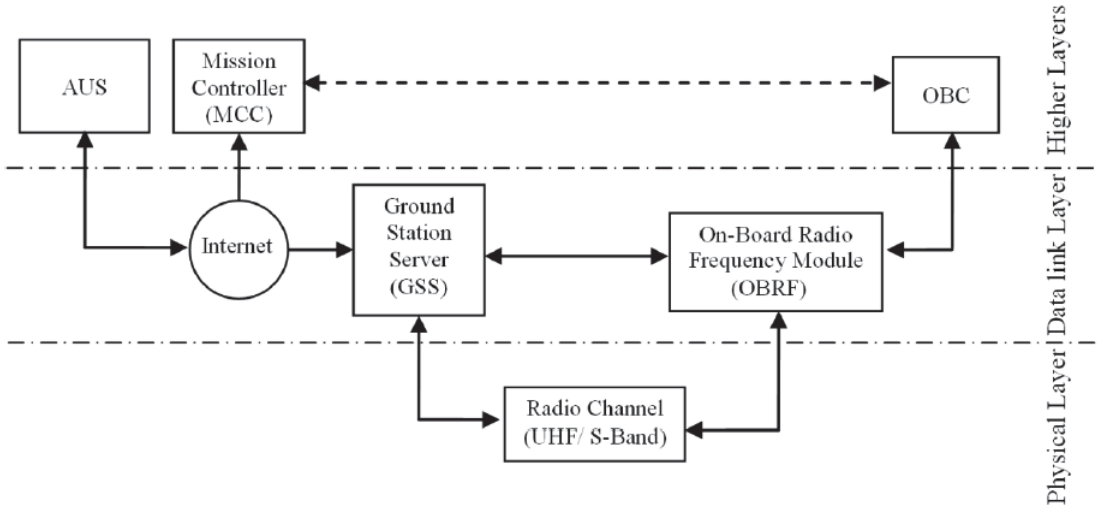

Figure 1.14. Roles between each layer in GENSO

Using this protocol consist of using some abstraction layers in order to achieve the connection from any location. A Mission Control Client (MCC), such as the university that has built the spacecraft, that needs to establish a connection with the satellite; to obtain this, will use internet for a connection to the Ground Station Server (GSS) located all over the world. This secure access is controlled by the GENSO Authentication Server (AUS), which ensures at all times that the entities participating in the network are allowed to do so. Under a visibility window of a given ground station (which is then under the footprint of the antenna's satellite), receives the spacecraft data and it is stored locally by the GSS, which can be an university that do not own the satellite. Then the GSS notifies the AUS, which in turn notifies the MCC owning the satellite. Finally, the MCC can establish downlink/uplink sessions directly from the GSS. Since ground stations are physically placed all over the world, the satellite is automatically tracked and MCC can be connected with the spacecraft independently from its real position. This behavior is layered as said before, and is depicted in figure 1.14.

To give access on amateurs, the data link layer follows the AX.25 protocol. In this way the GSS can be implemented by a TNC in kiss mode, in the same way as of the satellite OBRF. This module, in uplink, will check the correctness of the packet and its destination address, while other data contained is extracted and passed to the OBC, which in turn check and interpret the command at an higher OSI level with respect to the OBRF (except for particular functions described later, like backdoor). In downlink the situation is reversed, where the OBC send to OBRF a command to be sent to the GSS, and the OBRF itself will build it up to a complete AX.25 packet and then

send it, while the content of the packet is transparent to the OBRF.

1.5 Space environment

The AraMiS project is designed to work in LEO orbits, so between 400Km and 600Km of altitude from Earth. Here the electronic components can be affected by the environment conditions, because here the satellites are near the end of the terrestrial atmosphere. Here are starting the Van Hallen belts, in which are beginning non-negligible radiations.

1.5.1 Temperature

During the orbit, the satellite is exposed, at the same time, with faces to enlightened side and on the opposite direction to a darkened side. This provides a huge thermal gradient (creating thermal cycles) between the satellite faces. Moreover, due to the absence of the atmosphere, the solar radiation is higher than on the Earth.

In the space the temperature depends on the power balance of the satellite, therefore on the power absorbed from the Sun, the power converted to other types of energy and the power generated from components heating. Therefore, a smart way to cool the satellite is to absorb its energy using the solar panels, converting it in electric energy. Theoretically, keeping into account the power balance, temperatures are inside -30°C and +75°C .

1.5.2 Pressure

In LEO orbits the atmosphere is almost non-existent. Therefore it is a vacuum condition and the thermal dissipation through convection (where hot body transfers its energy to a surrounding flux, like air) is not possible. The exchange happens only through thermal conduction and thermal radiation.

Moreover, attention must be paid to liquids inside all the mechanical and electronic components, where they can overheat or explode. For this reason all of the capacitors used are need to be not polarized, for example electrolytic and tantalum capacitors could contain electrolyte or bring to dependability issues.

1.5.3 Radiations

The LEO altitude is in proximity of the lower Van Hallen belt, a a toroidal shaped area with charged particles, therefore full of ionizing radiations.

These charged particles when hit the semiconductor are generating the direct ionization and a pair of electron-hole is generated, leading to a various possible misbehaviours. The most known are the *Single Event Effect* (SEE) which are:

- *Single Event Latch-Up*: the parasitic BJTs of a CMOS cell start conducting, leading to a positive reaction which is creating a low impedance path between supply and ground rails.
- *Single Event Up-Set*: where one or more bits of a CMOS cells change their logical value. Writing again that value will restore the correct bit.

1.5.4 Total dose

The previous phenomenon are all instantaneous, but are existing also effects which are depending on the quantity of absorbed radiations, so are depending on the time in orbit. The *total dose* is the quantity of radiation which can be absorbed by a device before misbehaviours start happening. Threshold voltage of a MOS is a typical example, which increase its value proportionally with the absorbed radiations. This can increase the propagation time of signals leading to possible errors.

1.6 Thesis purpose

This thesis is a natural prosecution of a previous work, which has defined the main specifications and constraints of an AraMiS telecommunication module, then an initial version of the hardware were also defined. But the system's use cases were not completely devised and the firmware was missing.

Therefore, the main objective of this thesis work is focused on checking the constraints and specifications provided, completing the set of use cases and developing the firmware. Then a revision on the hardware will be performed in order to achieve better hardware performances and a PCB shape capable to be fit in a tile. And everything must be developed to be as much modular and fault tolerant as possible in order to comply with the AraMiS project specifications and dependability.

Chapter 2

UML approach

The Unified Modeling Language (UML) is a high level specification, description and documentation language. The purpose of this approach is to obtain a complete development flow for mixed-systems able to produce, on one side, documentation always close to real project implementation and, on the other, a fast and reliable method for reducing time-to-market in developing these objects. The project AraMiS is fully based on this approach. Initially developed in 1995 for designing software, the UML was optimally adapted to the description of systems made of both hardware and software.

The UML provide description of the system by means of diagrams, easing the understanding of a system's behaviour. The design flow of an AraMiS module consist in defining mainly, but not limited to, three diagrams: *Use case diagram*, *Class diagram* and *Sequence diagrams*. Reading this chapter will help in understand better the UML descriptions adopted in this thesis, because the project AraMiS has been thought to be very large and covering a lot of modules; therefore to avoid any kind of disorder and allow an easy coherent documentation process, has been heavily used the UML approach.

2.1 Use Case diagram

Use case diagrams show main function of the system (use cases) and the entities that are outside the system (actors). Use case diagrams show how the class and objects of the class relate and hierarchical associations and object interaction between classes and objects. These diagrams allow us to specify the requirements of the system and show interaction between system and external actors. These diagrams are the starting point in the system modelling and consist of actors and use cases.

Actor

Actors are generic entities, human users, other systems or the external environment, which interact with the system under design and implements one or more use cases. They are usually shown as sketched person (figure 2.1) with a short name which identifies the role in the system. They are associated with a detailed documentation. The list of actors is fundamental to understand all entities which might interact with the system. The actors are very fundamental entities and missing an actor will miss all the interfaces and functions associated with it. Are therefore external entities to the project which interacts with the use cases.

Use case

A use case is the system scenario saw by the actor which is interfacing to it. Are usually described by an oval with a name which shortly describes it. Building up the list of use cases means starting to specify the functions of the satellite or its subsystem and therefore thinking to the mission. There exist several kinds of relations between use cases and actors including generalization, inclusion, extensions, associations.

In figure 2.1 the continuous lines without direction are associations, while arrows with continuous lines are specializations of the generic use case to which they are pointing. Dashed lines mean the inclusion or the extension (depending on what the connections states) of the pointed use case, in order to guarantee the correct behaviour of the use case from which the arrow starts.

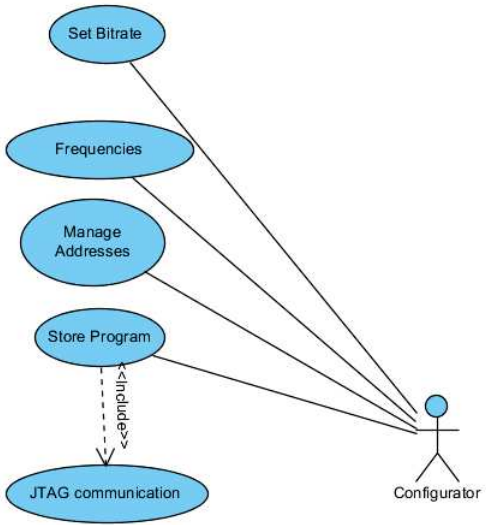

Figure 2.1. Example of use case diagram used in this thesis

2.2 Class diagram

Exploiting the analogy of the object oriented languages, classes and their characteristics are inherited also for mixed-system which are not only software. Since an attribute can refer to an object of some class type, in UML can be referred to a *physical* object too, which is contained in a particular class describing physical objects. This potential of melting hardware and software in the same language can be exploited to design a complete system, putting in evidence the relation of a physical object with a piece of software using a graphical representation.

All objects of the same type are represented by a class. A class and its attribute and methods could contain some stereotypes. A stereotype is needed to distinguish a class from another in terms of functionality and types of objects that will be instantiated by that class.

A class in UML is structured as in figure 2.2: that figure provide a software stereotyped class, square shaped divided in 3 horizontal sections inside, and from the top to bottom these sub-sections are indicating its characteristics:

- the stereotype (where the software is «SW», but in UML the classes could be also hardware, electronic modules and so on, as seen in chapter 5)
- Class name, i.e. Bk1B4221W_Tile_Processor_4M)
- Attributes (which could contains stereotypes too, like «pin», «constants» and so on)
- Operations, which are the methods if it is a software class, or connections if hardware

On the top of the class could be present a dashed white square, which is representing the templates adopted. A template is a set of parameters used to instantiate a class with different template-defined values w.r.t. another identical class, which are used in the commons methods.

When the class is not labelled (stereotyped) as software («SW»), the operations are representing the physical pins or buses. Otherwise, as used to be in UML and in object oriented programming, are referred to be methods or software operations (like stated by the UML tool).

2.3 Sequence diagrams

These diagrams are used to make clear and intuitive the relations with the various actors and classes over the time. The time-line increase vertically downwards. In this work these diagrams are heavily used to describe the protocols with their proper timing, easing the understanding of the system complexity. In figure 2.3 is provided an example, describing physical connections with classes and the actor associations.

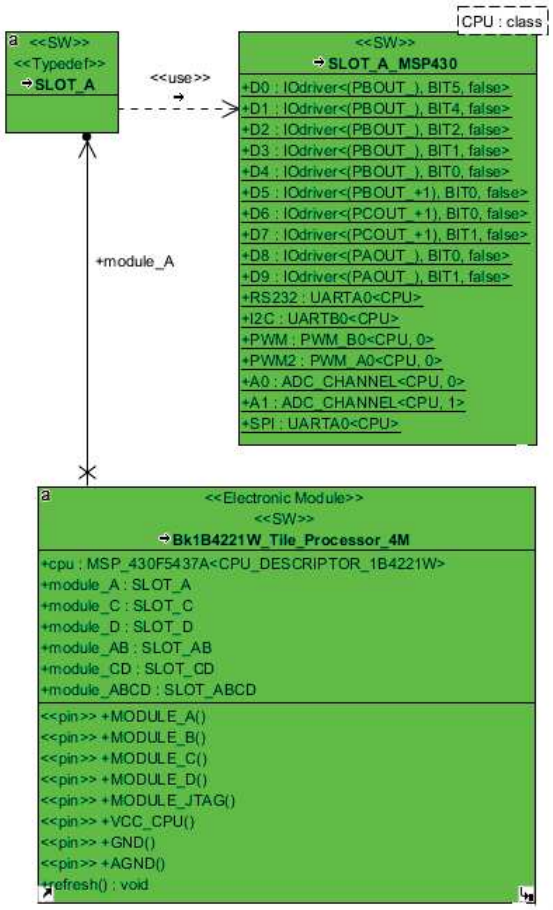

Figure 2.2. Example of class diagram used in this thesis

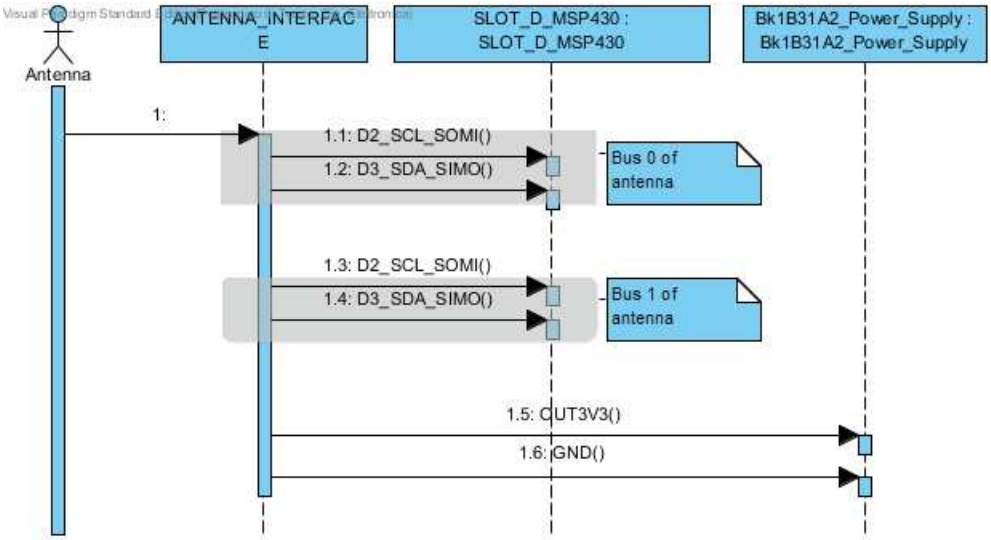

Figure 2.3. Example of sequence diagram used in this thesis

Chapter 3

System specifications and protocols

In this chapter are going to be described the satellite organization inside the AraMiS structure. Then is highlighted the developed sub-module of the spacecraft and are going to be listed its specifications and the various protocols adopted.

3.1 Satellite Organization

In this thesis is developed the sub-module of the 1B9_CubeTCT tile, applied on the AraMiS-C1 spacecraft, described in section 1.3.7. The organization of the 1B9_CubeTCT tile inside the whole system is described in figure 3.1. The communication with the ground segment happens through 2 antennas, one per sub-module. Each of these modules are communicating with the second tile on the system, the 1B8_CubePMT which contains the OBC. In each tile can be present other subsystems, and two of them are the 1B31_Telecommunication_System and the 1B42 OBC. The 1B31 sub-system contains the redundant module radios, namely 1B31A and 1B31B On-Board RF Module; redundant because each module work on different band, therefore the satellite can uses two radios. The sub-module of this thesis is the 1B31A, which operates in UHF band.

In AraMiS project classification of the 1B31A On-Board Radio Frequency Module, the **A** is related to the UHF band and it is the second revision, therefore it is referred to it also as **A2**. So the second one, called 1B31B which works in the SHF. When combined together, these modules are constituting a single redundant 1B9_CubeTCT Telecommunication Tile on the AraMiS-C1 CubeSat satellite. But the design is not limited to a CubeSat shaped spacecraft.

This module allow an AraMiS satellite to exchange data with the Ground Station, reaching the

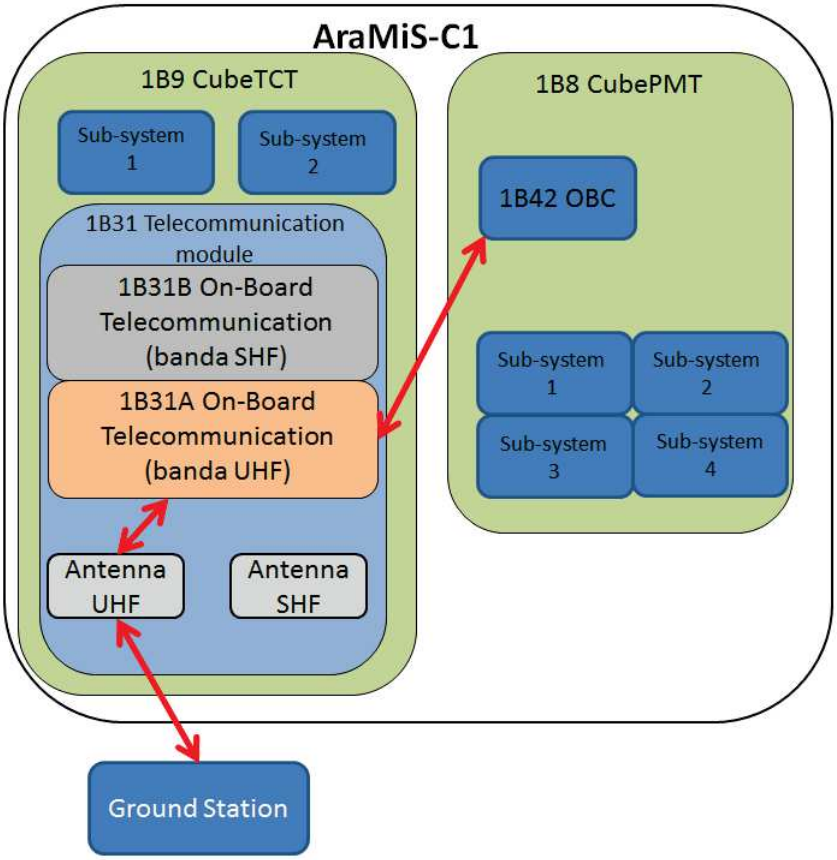

Figure 3.1. AraMiS-C1 system organization

Mission Control centre. Moreover, the data in downlink from the satellite should be accessible to the radio amateur community. The AraMiS initial specifications of the 1B31A module are:

- Operating frequency: 437 MHz
- Data rate: 9600 bit per second
- Maximum satellite transmission power: 33 dBm
- Maximum ground transmission power: 47 dBm
- Module control interface: I2C
- Available power supply:
  - Power Distribution Bus: 15W, from 12V up to 18V
  - 3.3V, 1W

  - 5V, 1W

- AX.25 protocol compatible

In these communication systems it is important to understand the protocols at high level, discovering the system dependability and timings. Before describing all the protocols devised for the system, will be reported the layering concept of provided by the *OSI stack*, to understand the role of each protocol adopted.

3.2 OSI stack

A briefing on the OSI stack is needed to understand the complexity of the adopted protocol. The Open Systems Interconnection model (OSI) is a conceptual model that characterizes and standardizes the internal functions of a communication system by partitioning it into abstraction layers. The model is a product of the Open Systems Interconnection project at the International Organization for Standardization (ISO), maintained by the identification ISO/IEC 7498-1. The model groups communication functions into seven logical layers. A layer serves the layer above it and is served by the layer below it. For example, a layer that provides error-free communications across a network provides the path needed by applications above it, while it calls the next lower layer to send and receive packets that make up the contents of that path. Two instances at one layer are connected by a horizontal connection on that layer. The 1B31A OBRF is handling the Layer 2 (except for Backdoor and RF Beacon which needs the support for Layer 3), while the OBC uses the Layer 3 at least. In figure 3.2 are depicted the various parameters. [9]

3.3 AX.25 protocol

In this section are provided the characteristics to comply with AX.25 v2.2 protocol used in AraMiS. Packet radio networks use AX.25 as a data link layer protocol, that is derived from the more general X.25 suite and adapted for radio amateur use. AX.25 is a pre-OSI model protocol, so at the origin the layering was not clearly delineated. However, since both the transceiver and the UHF channel have been already identified, the goal is to use AX.25 just for the data link layer specifications. [10]

The AX.25 protocol uses three types of packets, the Information Frame, Supervisory Frame and Unnumbered Frame. On this 1B31A On-Board Radio Frequency Module at 437MHz will be adopted a connectionless link type which uses only the Information Frames. The Information frame is structured as picture 3.3, where is clear the insulation role of the first and last Flag, with a constant value, used to separate one frame from another in the medium.

OSI Model				
Layer		Data unit	Function	Examples
Host layers	7. Application	Data	High-level APIs, including resource sharing, remote file access, directory services and virtual terminals	HTTP, FTP, SMTP
	6. Presentation		Translation of data between a networking service and an application; including character encoding, data compression and encryption/decryption	ASCII, EBCDIC, JPEG
	5. Session		Managing communication sessions, i.e. continuous exchange of information in the form of multiple back-and-forth transmissions between two nodes	RPC, PAP
	4. Transport	Segments	Reliable transmission of data segments between points on a network, including segmentation, acknowledgement and multiplexing	TCP, UDP
Media layers	3. Network	Packet/Datagram	Structuring and managing a multi-node network, including addressing, routing and traffic control	IPv4, IPv6, IPsec, AppleTalk
	2. Data link	Bit/Frame	Reliable transmission of data frames between two nodes connected by a physical layer	PPP, IEEE 802.2, L2TP
	1. Physical	Bit	Transmission and reception of raw bit streams over a physical medium	DSL, USB

Figure 3.2. Description of OSI layers

Flag	Address	Control	PID	Info	FCS	Flag
01111110	112/224 Bits	8/16 Bits	8 Bits	N*8 Bits	16 Bits	01111110

Figure 3.3. Information Frame

Address Handling

Despite the AX.25 constitutes an OSI Layer 2, in this application it is not used any repeater in the AX.25 Layer 2, so the destination address (the address field in the image 3.3) is the callsign and SSID of the amateur radio station to which the frame is addressed, and it is not followed by any repeater. The source address contains the amateur callsign and SSID of the station that sent the frame. These callsigns are parts of the two ends of a Layer 2 AX.25 link only (figure 3.4).

First
Octet
Sent

Address Field of Frame	
Destination Address Subfield	Source Address Subfield
A1 A2 A3 A4 A5 A6 A7	A8 A9 A10 A11 A12 A13 A14

Figure 3.4. Non-repeater address field encoding, byte structure

A1 through A14 in figure 3.4, are the fourteen octets (bytes) that make up the two address subfields of the address field. The destination subfield is seven octets long (A1 through A7), and

is sent first. This address sequence provides the receivers of frames time to check the destination address subfield to see if the frame is addressed to them while the rest of the frame is being received. The source address subfield is then sent in octets A8 through A14. Both of these subfields are encoded in the same manner, except that the last octet of the address field has the HDLC address extension bit set. The HDLC address field is extended beyond one octet by assigning the least-significant bit of each octet to be an "extension bit". The extension bit of each octet is set to "0" to indicate the next octet contains more address information, or to "1", to indicate that this is the last octet of the HDLC address field. To make room for this extension bit, the amateur radio call-sign information is shifted one bit left. Here the extension bit is never used, except to make room for the Source Address Subfield and here will be always set to one in the A14 byte (octet).

In fact, the spacecraft should work with address fields that are not providing repeaters from one station to another. This is due to the GENSO structure, where it is not needed any radiolink repeating mechanism. But the local callsign can have more stations therefore the SSID byte is supported without any increase in the complexity, but only in the address handling algorithm. The detailed example structure of addresses in satellite compatible AX.25 frame is shown in figure 3.5, where can be spotted the extension bit mentioned before. Bits positions of figure 3.5 are defined

Octet	ASCII	Bin Data	Hex Data
Flag		01111110	7E
A1	N	10011100	98
A2	J	10010100	94
A3	7	01101110	6E
A4	P	10100000	A0
A5	space	01000000	40
A6	space	01000000	40
A7	SSID	11100000	E0
A8	N	10011100	98
A9	7	01101110	6E
A10	L	10011000	98
A11	E	10001010	8A
A12	M	10011010	9A
A13	space	01000000	40
A14	SSID	01100001	61

Figure 3.5. Non-repeater AX.25 address bit structures

from 7 (left, MSB) to 0 (right, LSB). The SSID bits in positions from 1 to 4 are the four bits used to identify an SSID. The other bits a not used and kept as shown in this image above (i.e. octet = 011SSIDx, where x is the extension bit, active low).

NOTE: The correctness of SSID field is responsibility of the OBC as long as the correctness of the whole destination address, excluding the left shift of bytes of the destination address which are performed by the 1B31A OBRF. The correctness of the source address (i.e. the spacecraft) should be responsibility of the Configurator at compile-time, along as the correctness of destination address when dealing with autogenerated packets (see section 4.2.11).

Control Handling

The control field is shown in figure 3.6, to be part of an I-frame. The receive sequence number N(R) and send sequence number N(S) are handled by the OBC, and for what concerns transmission from the satellite, are sent in a single byte from OBC to the 1B31A On-Board Radio Frequency Module 437MHz which takes care of their correct positioning inside the AX.25 frame and finally send them over the radio-link. In reception, those are find from the frame itself and unpacked in a byte which is sent back to the OBC. The P bit is not used, it should always be 0.

Control Field Type	Control-Field Bits							
	7	6	5	4	3	2	1	0
Information	N(R)			P	N(S)			0

Figure 3.6. I-frame control field byte

Protocol Identifiel Handling

For the spacecraft communication system it is not used any OSI Layer 3. Therefore the PID value should always set to a code which is defined to denote no layer 3 adopted, PID = 0xF0 (refer to figure 3.3).

Info Handling

This field contains the payload which is sent or received. The info field defaults to a length of 256 bytes and contains an integral number of bytes. These constraints apply prior to the insertion of zero bits (bit stuffing, described in software chapter). Any information in the info field is passed along the link transparently, except for the zero-bit insertion necessary to prevent flags from accidentally appearing in the Info field. It is not written explicitly how many bits are present in the AX.25 packet, but can be devised since before the final ending flag there are 2 bytes of FCS.

Frame Check Sequence Handling

The Frame-Check Sequence (FCS) is a sixteen-bit number calculated by both the sender and the receiver of a frame. It ensures that the frame was not corrupted by the transmission medium. The Frame-Check Sequence is calculated in accordance with recommendations in the HDLC reference document, ISO 3309. The algorithm used is the standard CRC-16-CCITT ans stored reversed w.r.t. other bytes, conventional to CRC bit orders. The bit order is defined better here below.

Data and FCS bit orders

All fields except the Frame Check Sequence (FCS) are transmitted low-order bit first. FCS is transmitted the bit 15 first. Therefore the reversed (i.e. original) calculation must be adopted. In all figures shown here, the right position is the LSB, except for the FCS.

Starting/ending flag

The whole packet in included in the AX.25 flag, which denoted by the value 0b01111110.

3.3.1 CRC check and algorithm

Provides the capability of the 1B31A OBRF module to handle, check and generating the Cyclic Redundancy Check (CRC). As it will be shown in next chapters, will be used a microcontroller which can handle this elaboration directly in hardware, by using a Linear Feedback Shift Register (LFSR). The CRC module produces a signature for a given sequence of data values. The signature is generated through a feedback path by means of an LFSR, implemented in hardware and shown in figure 3.7. This feedback takes back data bits 0, 4, 11, and 15. The CRC signature is based on the polynomial given in the CRC-CCITT-BR standard polynomial:

$$f(x) = x^{16} + x^{12} + x^5 + 1$$

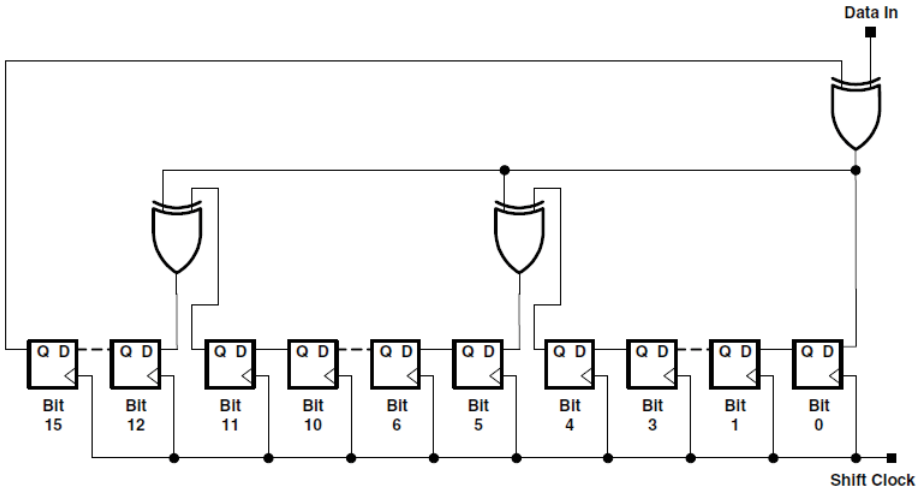

Figure 3.7. LFSR Implementation of CRC-CCITT Standard, Bit 0 is the MSB of the Result

With a given data of a given length, a unique signature of 16 bits is obtained, with one bit per flip-flop in the figure 3.7. In this way identical input data sequences result in identical signatures

when the CRC is initialized with a fixed seed value, whereas different sequences of input data, even in one single bit, result in different signatures, if the aliasing of the signature does not take place.

Initial seed must be 0xFFFF for all Basic Communication Protocols, included the AX.25 protocol. Once all data have been processed through the CRC check, the value stored inside the register is added at the end of data for error protection.

3.4 AraMiS Telecommunication protocol

This protocol is aimed to achieve a reliable data exchange on an unreliable radio-link with the GSS (at OSI layer 3), using a standardized data link (OSI layer 2) protocol, the AX.25, which was not born for the space environment. The dependability is provided by an higher layer 3 protocol, developed in the AraMiS project called *1B3_TT&C_Telecommunication_Subsystem*, capable to interpret always the content of the frame and guaranteeing that a complete packet will be always received.

In order to be adopted correctly, the protocol is described in sequence diagrams, which are representing function calls (in horizontal axis) versus time (vertical axis, downward). Since these diagrams in this chapter are dealing with a system that runs some software, the best way to achieve a description is to represent the interactions between various events as a function calls. These calls are adopted to make a clear description of what the software should be ready to accept or to prepare, what to expect, its data dimensions and alignment. In this way the description starts to be lower w.r.t previous sections, in this top-down approach.

The functions are contained in few high-level class diagrams (*1B31* class in figure 3.8) and are then used in these sequence diagrams. With considerations just made, these functions can be interpreted to be the OBRF logical interface (comprehending layers 2 and 3 of the OSI) with the OBC and Ground Segment or any other external actor involved (see previous use cases sections). So after a description of these functions, sequence diagrams have been adapted to became compatible with the OBRF behavior which, in turn, it has been thought to be as much compatible as possible with the previous protocol, obtaining a final good compromise of design.

3.4.1 OBRF interfacing functions

Here are described what are the data specifications of the OBRF, represented at logical level by some functions, in order to define what kind of data is supported in the OBRF-OBC interaction. In figure 3.8 there is the class diagram of a redundant module named 1B31 OBRF containing these interfacing functions, which includes the SHF band (1B31B) and the UHF (1B31A). This last one is the one designed in this thesis work.

An introduction on the top-level system organization is needed to understand the interaction of the interfacing functions with the system and understand their level of abstraction. The class **Bk1B31A2M_OBRF_437MHz** consist of the module implementation, marked as **M**. This codename indicates that a project which have this class, in its final implementation will have only the UHF subsystem (because the *1B31A2*), and will be complete of connectors and PCB layout (because of the *M*, module). For example a module, which when realized on AraMiS is called "tile", when complete of two redundant systems in two different bands, should have instead the module codename **Bk1B31M_OBRF_437MHz**, which includes all of the OBRF sub-systems. With these considerations in mind, the class **Bk1B31A2W_OBRF_437MHz** instead represents the wired module (marked as **W**), which contains everything but the external connectors and PCB layout, so it is not a final tile in a module of type M.

Here is now evident the modularity of AraMiS, where any complete final system can be put together with already tested sub-modules and only decide the external connectors and the PCB shape. The diagram of the complete module is shown and explained in chapter 5 at section 5.1.

In figure 3.8, since the M and W classes are coinciding, the physical interfaces are the same.

Here are now described the behaviours of the OBRF logical interfacing functions, needed by the external actors which they must know how to interface to the tile.

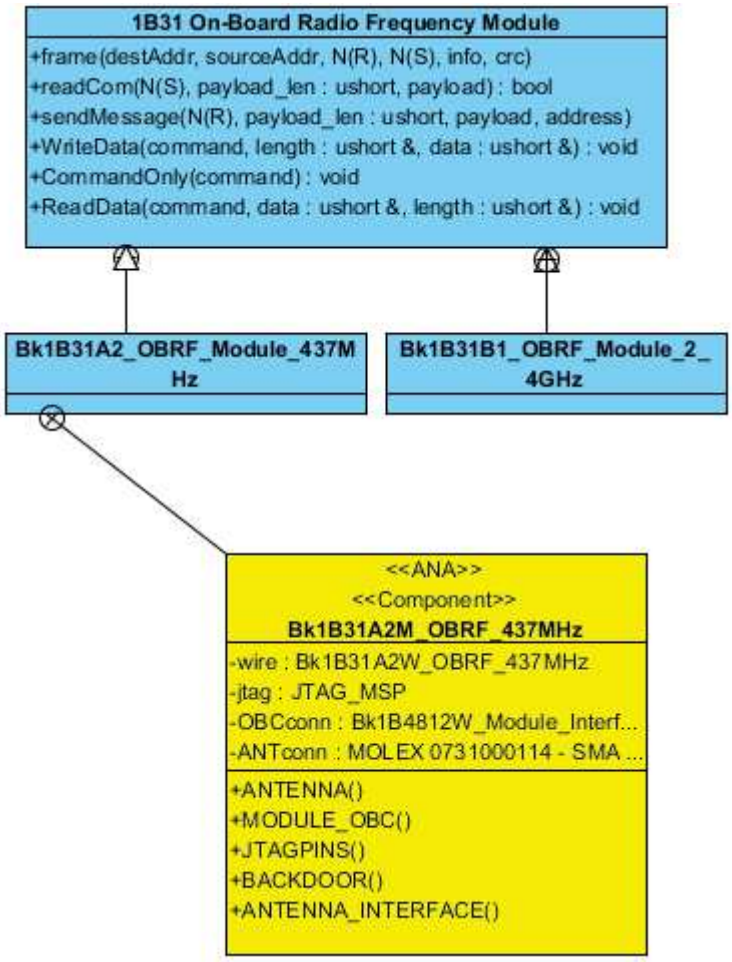

Figure 3.8. Top level class diagram of the On-Board Radio Frequency Module, with both 437MHz and 2.4GHz sub-modules

Interfacing functions

As can be seen in figure 3.8, these logical functions are shared by any RF module which should be compliant with the defined use cases. In the figure, the module at 2.4GHz will adopt the same behaviour, using the same interfacing functions.

frame(destAddr, sourceAddr, N(R), N(S), info, crc)
This function implements the use case AX.25 protocol in section 3.3, and contains all the packet parameters, therefore handle the data at OSI Layer 2 from the OBRF side, and the parameters are exactly what is transmitted/received in the RF link:

- *destAddr* corresponding to the address **AX_SAT_ADDR : char const** of 1B31A On-Board Radio Frequency Module 437MHz in reception, or Ground station address if transmission.
- *sourceAddr* corresponding to the address **AX_SAT_ADDR : char const** of 1B31A On-Board Radio Frequency Module 437MHz in transmission, or Ground station address if reception.
- *N(R)* is the sequence number used in case of 1B31A OBRF transmission.
- N(S) the sequence number used in case of 1B31A OBRF reception.
- *info* is the payload of *readCom(N(S), payload_len : ushort, payload) : bool* or the payload of *sendMessage(N(R), payload_len : ushort, payload, address)* (both described later).
- *crc* is part of AX.25 protocol and is generated by the OBRF LFSR.

readCom(N(S), payload_len : ushort, payload)
This function implements use case Get Received Packet in section 4.2.4, by calling the method *1B31A On-Board Radio Frequency Module 437MHz::ReadData(command, data : ushort &, length : ushort &) : void* with **command = GET_PACKAGE**. This description is important, because the OBC should know what command search inside the payload and the OBRF should know where to put the payload itself; the same consideration apply for the *sendMessage()* function described after this one. Therefore the OBC must support the OSI Layer 3 in order to handle the payload which contains a command. The parameters are the relevant AX.25 informations, provided by the OBRF from the Layer 2 point of view:

- *N(S)* (corresponding to data[0] from *ReadData(command, data : ushort &, length : ushort &) : void*), where the system which reads this data can assume N(R) to be 0

- *payload_len* of payload (number of bytes; corresponding to *length-1* from *ReadData(command, data : ushort &, length : ushort &)*)
- *payload* (data[1-255] taken from *ReadData(command, data : ushort &, length : ushort &)*)

In other words, this function calls the *ReadData(command, data : ushort &, length : ushort &) : void* with **command = GET_PACKAGE**, returns the **payload_len = length-1**, to take into account that N(S) is taken apart, and therefore corresponds to the AX.25 protocol payload length only, in bytes. It then splits data; returns its first byte as N(S) and the other bytes(as many as payload_len) which are the payload.

sendMessage(N(R), payload_len : ushort, payload, address)
This function implements the use case Transmit in section 4.2.5 by calling method *1B31A On-Board Radio Frequency Module 437MHz::WriteData(command, length : ushort &, data : ushort &) : void* with **command = CMD_TRANSMIT** to send from OBC to Antenna the relevant AX.25 information. Therefore handle the data at OSI Layer 2 from the OBRF side, and requires a Layer 3 from the OBC side:

- *N(R)* corresponding to data[0] from *WriteData(command, length : ushort &, data : ushort &) : void*. In case of auto-generated messages from the OBRF, this is always 0. The system which reads this data (i.e. Ground Station) can assume N(s) to be 0.
- *payload_len* of payload (number of bytes; corresponding to length-1 from *WriteData(command, length : ushort &, data : ushort &) : void*). This value is not sent but only used by the sender system. In case of auto-generated messages from the OBRF (the sender system), this value must be calculated since there is no length parameter (from OnBoard Protocol in section 4.4).
- *payload* (data[1-255] taken from *WriteData(command, length : ushort &, data : ushort &) : void*). This part of the vector contains the required fragments, in case of fragmented messages. The organization of the payload is then OBC dependent except for auto-generated messages.

In other words, this function calls *WriteData(command, length : ushort &, data : ushort &) : void* with **command = CMD_TRANSMIT**, generates the **payload_len = length-1**, which is the length in bytes of the info (from *frame(destAddr, sourceAddr, N(R), N(S), info, crc)*), to allow a correct AX.25 encapsulation and take apart the N(R) from payload. It then merge all correctly in the packet according to the AX.25 protocol (section 3.3), before serialize it to the Antenna. In case of auto-generated messages from the OBRF (e.g. RF Beacon), the OBRF module sends the content without the need of any triggering command, therefore any function which depends on any

OBC interaction (like those functions used by the namesake modes in section 4.4, i.e ReadData and WriteData, previously mentioned and described here below) is excluded from the sending process. This require, in this case, a Layer 3 support from the OBRF.

WriteData(command, length : ushort &, data : ushort &) : void
This function implements use case Write Data mode of 1B45 protocol described in section 4.4. Briefly, the OBRF will receive a command from OBC and then a subsequent length parameter, followed by the data buffer containing data related to the command.

ReadData(command, data : ushort &, length : ushort &) : void
This function implements use case Read Data mode of 1B45 protocol described in section 4.4. Briefly, the OBRF will receive a command from OBC and the OnBoard Radio Frequency will returns back a response with a given length and then the data buffer containing what was requested in command by the OBC.

CommandOnly(command) : void
This function implements use case Command Only mode of 1B45 protocol in section 4.4. It is implemented to support the 1B31A On-Board Radio Frequency Module 437MHz to receive a command from OBC and will execute it without any other data exchange.

A graphical representation of information parsing is shown in figure 3.9. That figure represents the satellite shown in figure 3.1 but at logical level. Moreover emember that the 1B45 functions are used to generate the *readCom/sendMessage* functions, therefore the 1B45 should be considered at a lower level. These interactions are described in sequence diagrams, starting from image 3.10.

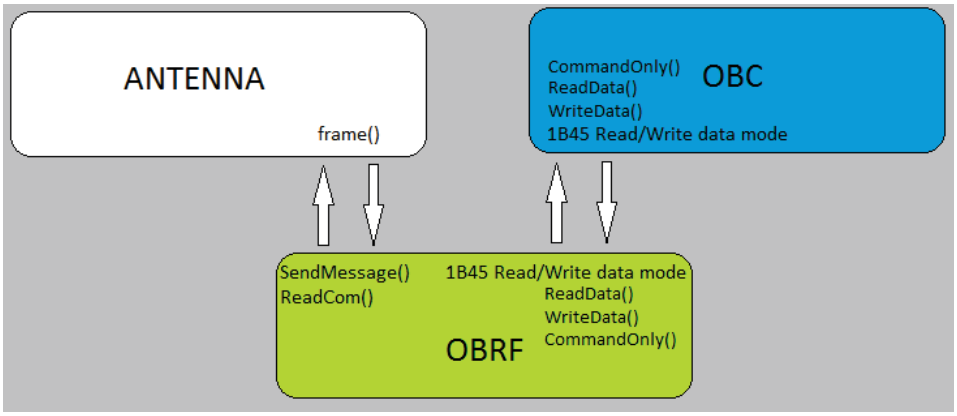

Figure 3.9. Interactions and interfacing functions involved between On-Board Radio Frequency Module, On-Board Computer and the Antenna

3.4.2 Behaviour of the protocol

Once all the functions used by the OBRF, the ones required by the OBC and various bus protocol are defined, are now introduced the sequence diagrams of the telecommunication protocol, describing the Layer 3 handling guidelines for the OBC and also the sequence of the 1B45 protocol operations described in the previous section. These diagrams are modified w.r.t. to previous implementation, but kept the same from a conceptual point of view of the 1B3 protocol, in order to be able to communicate with the OBRF while keeping at the same time original algorithm structure of the protocol itself.

The AraMiS telecommunication protocol is subdivided in multiple type of OSI Layer 3 commands (i.e. contained in payload). These can be:

- **Short command**: it is a command which is just transmitted or received without any further response.
- **Long command**: it requires an additional response after some time, for example data which can be available in next instants (for example, earth's photo of a given time).
- **Fragmented command**: with a long command is required a very long data which need to be fragmented on more packets. With a fragmented command type are required fragments of a long command response.

All of these macro-types are briefly analysed. This protocol has been adjusted to be coherent with the use cases of the module in section 4.2.

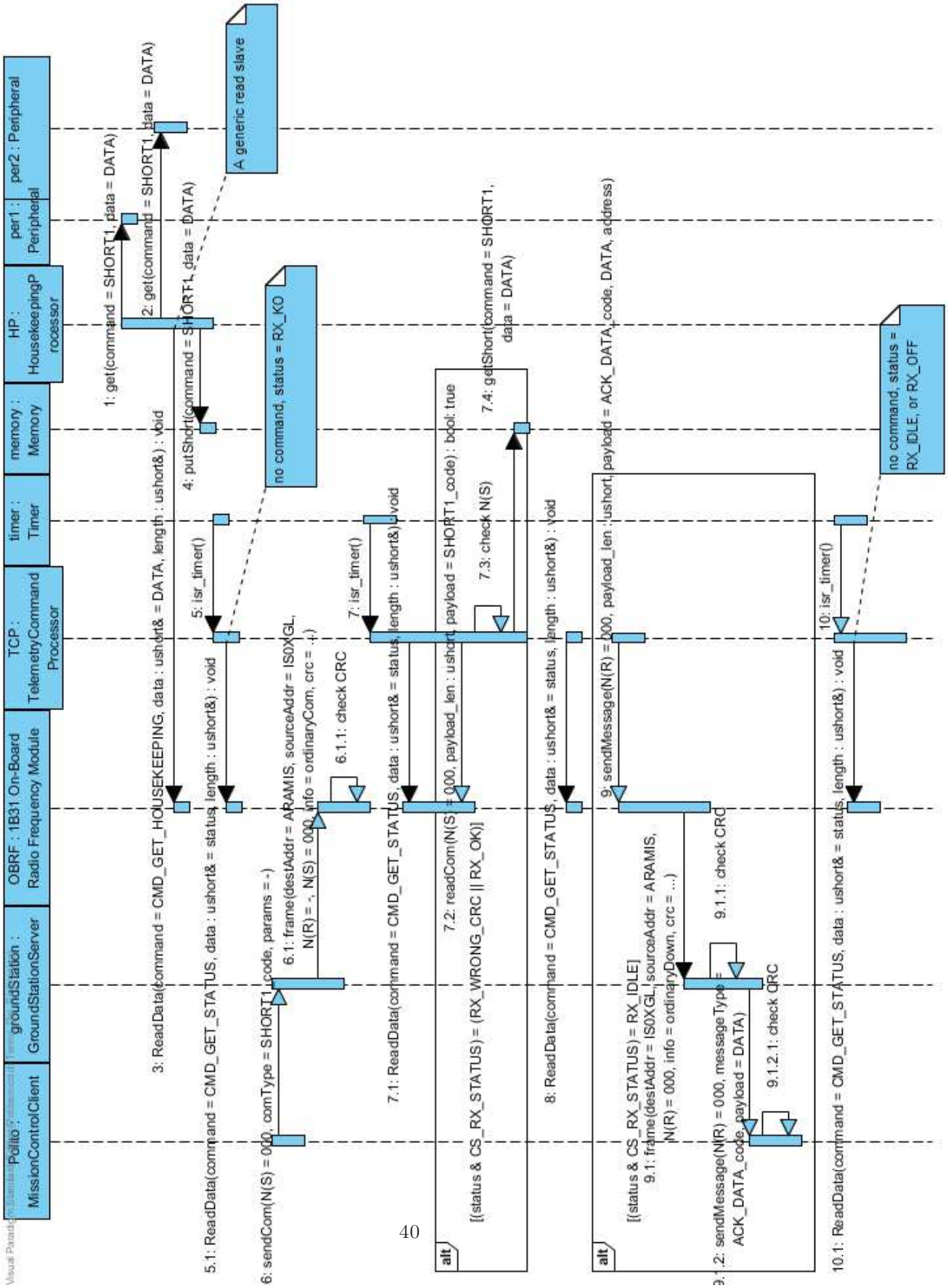

Figure 3.10. Basic Telecommunication AraMiS protocol, short command case)

Basic Telecommunication AraMiS protocol, normal case

This communication condition is referred to image 3.10. This sequence shows the events related to a short command without errors. Step 1 and 2, are associated to a housekeeping information transfer between a general peripheral on the spacecraft and the OBC, with a *ReadData(command, data : ushort &, length : ushort &) : void* and *get(command, data &)*, which is a periodic interrogation issued to other peripherals using the 1B45::ReadData. As soon as the data is retrieved, in step 4 the *putShort(command, data)*, which is another interfacing function of the TCP, therefore not used by the OBRF, puts the retrieved data in the OBC's Memory. Then, with an isr_timer() ISR which runs on the OBC in step 5, the Telemetry Command Processor ask if there is data available from Ground station by checking the status of the 1B31 OBRF by using *ReadData(command, data : ushort &, length : ushort &) : void*. This will return some data (as seen in section 4.2.7) which is associated to a no data available in this case. The OBC returns then to its other tasks.

Now suppose that in a given time the Mission Control Client requests the housekeeping data telemetry. Through the step 6 *sendCom(N(S), comType, params)* by means of the Internet sends to the Ground station the command. When Ground station receive it, will compose it according to Frame_AX.25 with this fields (see section 3.3):

- The AX_SAT_ADDR : char const in the destination address field
- Its own callsign in the source address
- The sequence numbers N(R), N(S) in the control field and the PID
- The command in Info of type byte[INFO_LEN], of a given length INFO_LEN.
- The FCS : byte[2] (CRC)

The Ground station send the frame on the radio channel towards the satellite in step 6.1 according to *frame(destAddr, sourceAddr, N(R), N(S), info, crc)* structure defined in the OSI level 2. When the frame is received is checked the FCS : byte[2] and (transparently to the OBC) the destination address AX_SAT_ADDR : char const is checked by the 1B31 On-Board Radio Frequency Module (step 6.1.1). The behaviour in the 1B31 On-Board Radio Frequency Module is described at high level in another diagram in figure 3.13 and more in depth in software chapters. If the address is correct, whether it is CRC correct or not, the content of the frame is decapsulated and ready to be sent to the OBC, signalling an eventual FCS error with updating the status with a RX_WRONG_CRC flag. But the TCP ask the availability first (after isr_timer() has triggered the check), by using again the *ReadData(command, data : ushort &, length : ushort &) : void* with the proper status command in steps 7.x.

If the OBRF module responds with a proper status of data ready (see section 4.2.7) (step 7.1), the transfer begin only after a second command which uses the *readCom(N(S), payload_len : ushort, payload) : bool* (step 8). Note that the connection between the 1B31 On-Board Radio Frequency Module and the Telemetry Command Processor is made through polling, since the former is an hardware object and the latter is a software object of the OBC. This is accomplished, as said earlier, by using an isr_timer(). The TCP checks the N(S) (step 7.2) and interpret the command accordingly. In this case it is a simple short command, which means that is retrieved only one kind of data from the Memory (step 9), regardless the sequence number. In step 10 is checked again if the transmitter is available with a proper command put in *ReadData(command, data : ushort &, length : ushort &) : void* and, if so, the message is sent back to Ground station (step 11) with *sendMessage(N(R), payload_len : ushort, payload, address)*, which has the double function to acknowledge the Ground station and carry the content of the required data. The transmission in downlink is the symmetric to the previous described for the uplink.

Basic Telecommunication AraMiS protocol, long command case

This communication condition is referred to images 3.11 and 3.12. Here are described the events in case of a LONG command without errors.

In a given time, it is supposed that the MCC decides to request an image acquisition to be taken at one hour from now. In step 2 the command is then sent with *sendCom(N(S), comType, params)* to the Ground Station Server: here a value to *params* is assigned to identify uniquely a command to a further data request generated by its execution. The command is transported to the Telemetry Command Processor as described in section 3.4.2. Then the N(S) is checked (step 3.3) to understand if it is the first command received and then interpret the command according to a TCP's table.

Supposing that is the first command, the TCP communicate with the interested peripheral and issue the command's execution *exec(command, arguments) : bool* at step 3.4, while with *put(applNum, data) : bool* at step 3.5 it allocates a location in Memory to store the future data generated by the peripheral. Since the Mission Control Client need to know if data is received even though it is not yet executed, the TCP generates an acknowledge signal with no subsequent data (pure acknowledge), steps from 4 to 5.1.2.

From this moment on, the peripheral used with that command is added to an OBC's list of peripherals that will be polled periodically, to evaluate if its execution has been finished and eventually acquire the generated data. A polling while the execution is not yet finished is made in step 7 where a NULL data is returned into Memory. After a reasonable time, the Mission Control Client in step 8 decides to check if the previously required data is available. But here is not yet ready and the TCP returns a "data not ready" at step 11, according to what has been retrieved

from Memory with a *getLong(applNum, data &)* (step 9.4).

In step 13 the TCP polls again the peripheral, and now the data is retrieved and memorized (step 14). In a subsequent moment the MCC asks again the data request, step 15. Now the *getLong(applNum, data &)* at step 16.4 will return the peripheral's data from Memory. An ACK_DATA response can be generated by the TCP with the content of the peripheral's response (steps from 16.6).

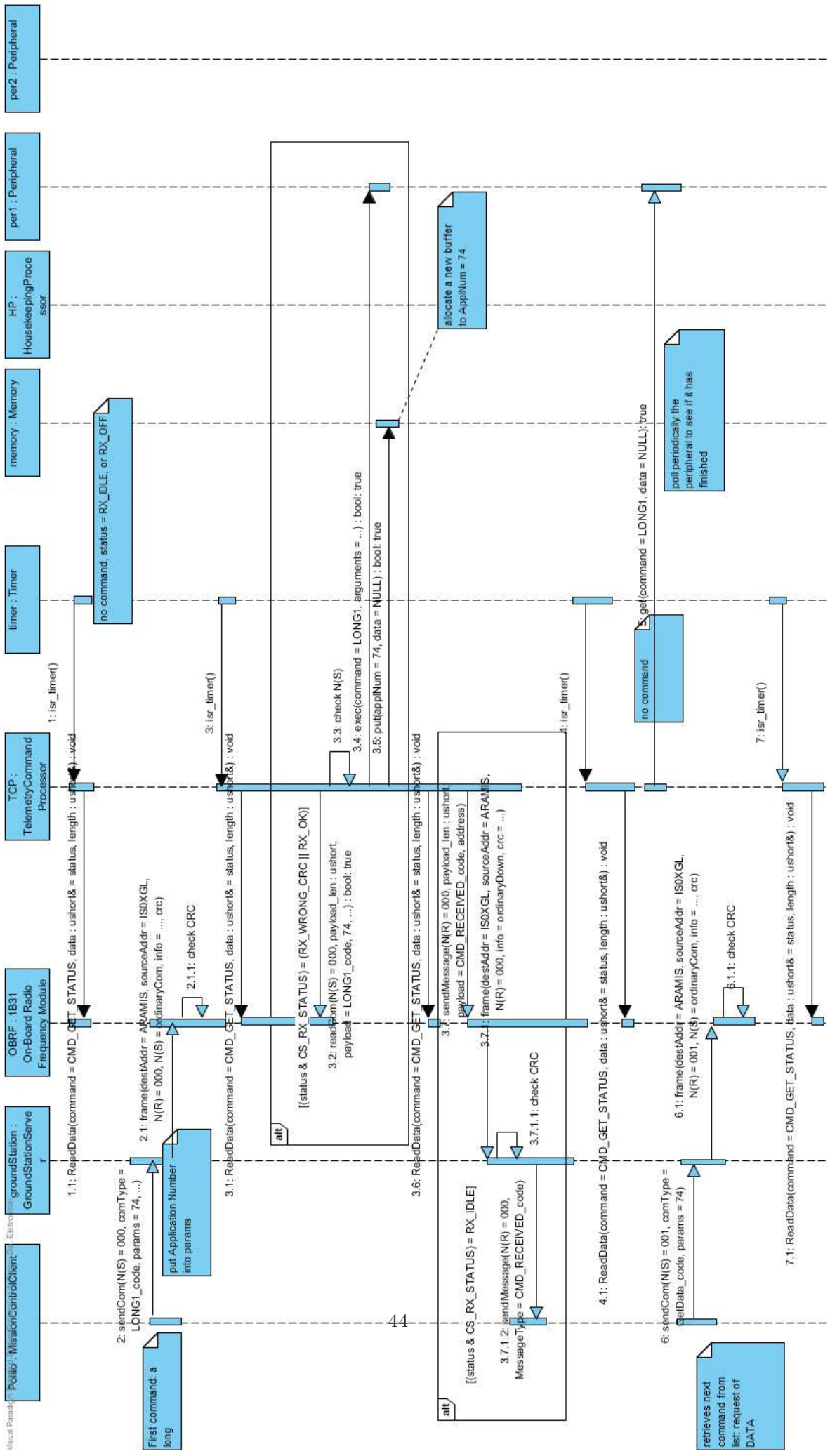

Figure 3.11. Basic Telecommunication AraMiS protocol, long command case (part 1/2)

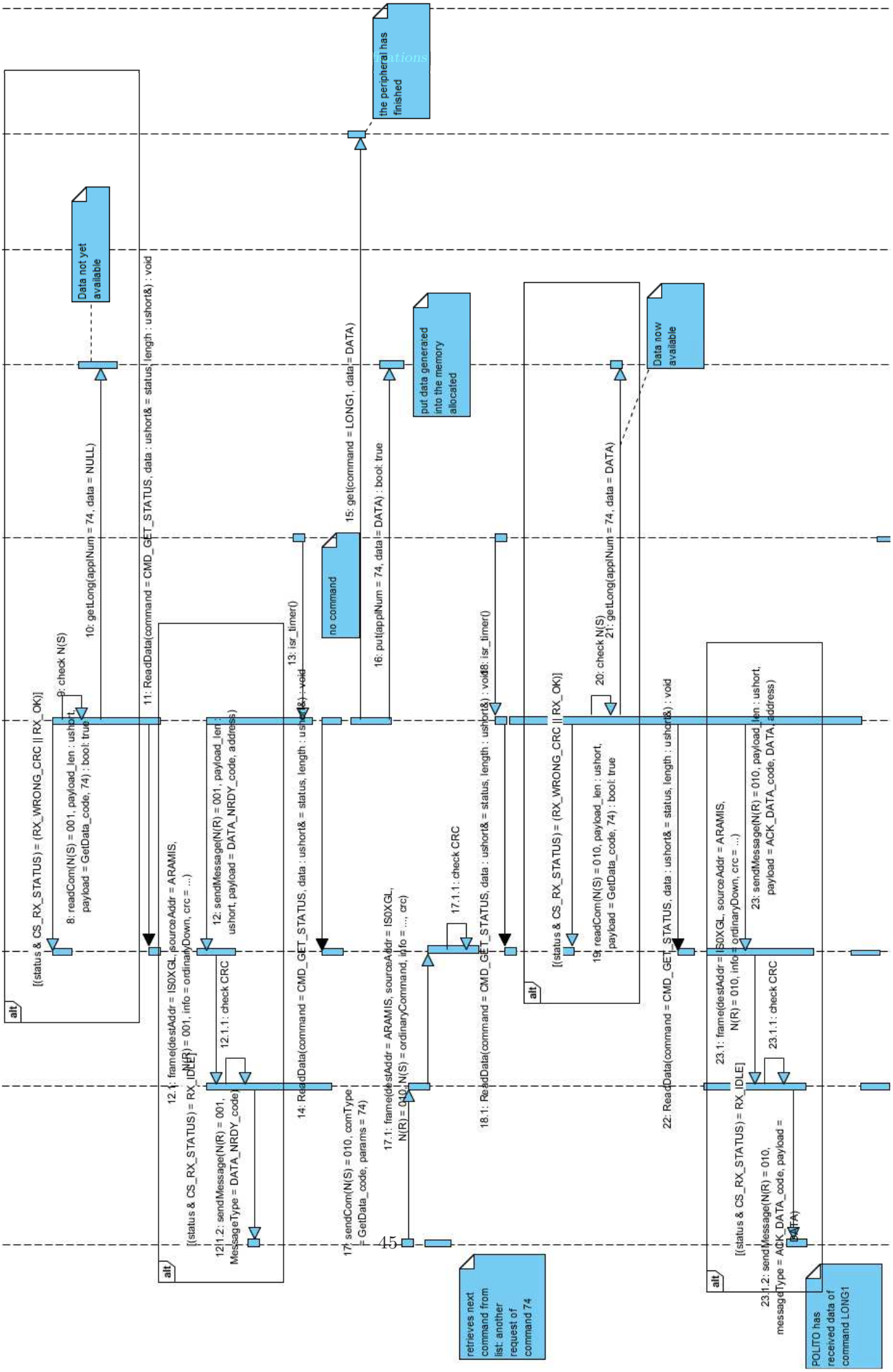

Figure 3.12. Basic Telecommunication AraMiS protocol, long command case (part 2/2)

Basic Telecommunication AraMiS protocol, OBRF behaviour

This communication condition is referred to image 3.13. In steps 1.x is presented the situation in which a packet received is not designated to be received from this satellite. In this case, the 1B31 On-Board Radio Frequency Module keeps the **RxStatus : t_RX_STATUS in RX_IDLE**. When the TCP asks to the 1B31 On-Board Radio Frequency Module its status with *ReadData(command, data : ushort &, length : ushort &) : void* with the CMD_GET_STATUS command, in the answer of the OBRF will be present a **statusRegister : CS_ REDUNDANCY [LENGTH _STATUS]** with the value of **RxStatus : t_RX_STATUS** and no data transfer takes place. The exact codification of the status should can be find in the 1B31 OBRF's module documentation. Note that it is not a mandatory to fully receive a wrong addressed packet: in this way some optimizations can be achieved. In fact, the implemented software will stops the receiving.

In steps 3.x the status is read by the TCP exactly as previous steps. But here the **RxStatus : t_RX_STATUS is RX_WRONG_CRC**, since the address is right but the packet has some errors. Here the TCP will not issue any data reception request, but if it is needed for any reason, it can be still received wrong even thought it is not advised. The OBC will signal to the 1B31 On-Board Radio Frequency Module that the packet is not needed so that the status flag can be reset to RX_IDLEand the content of the packet trashed. In order to do this, the OBRF buffer should be flushed by requesting the packet. In other words, the OBC should always receive the packet, but should keep trace if this is needed or not. If this is not made, the OBRF's buffer will overflow and the new packets will overwriting the old ones.

Then from step 4.2 on, the TCP will generate a command related to the wrong packet received, a pure NACK, to be sent to the GroundStationServer. Before a transmission, the OBRF's status is checked by reading the **RxStatus : t_RX_STATUS** of the *statusRegister : CS_REDUNDANCY [LENGTH_STATUS]*. If it is available, then a *sendMessage(N(R), payload_len : ushort, payload, address)* is performed, the data (a pureNACK generated by Telemetry Command Processor, parsed to OBC which sent it to OBRF) is packet in the chosen protocol (like AX.25) then sent to the Ground Station Server. If the transmitter is not yet available, the OBC should wait or can continue postponingg that transmission: this should be mission dependent and depending on how much the satellite on board traffic is high. Note that a missing response, anyway, is handled by this basic protocol.

In steps 5.x it is received a packet which is fully correct. Again, as in steps 1.x, is read the 1B31 On-Board Radio Frequency Module status to understand the correctness of the last received packet. The a *readCom(N(S), payload_len : ushort, payload) : bool* is issued and the whole packet is sent to the Telemetry Command Processor.

These mechanisms are applied to the whole 1B3 basic protocols (the other diagrams), each time the OBRF is involved.

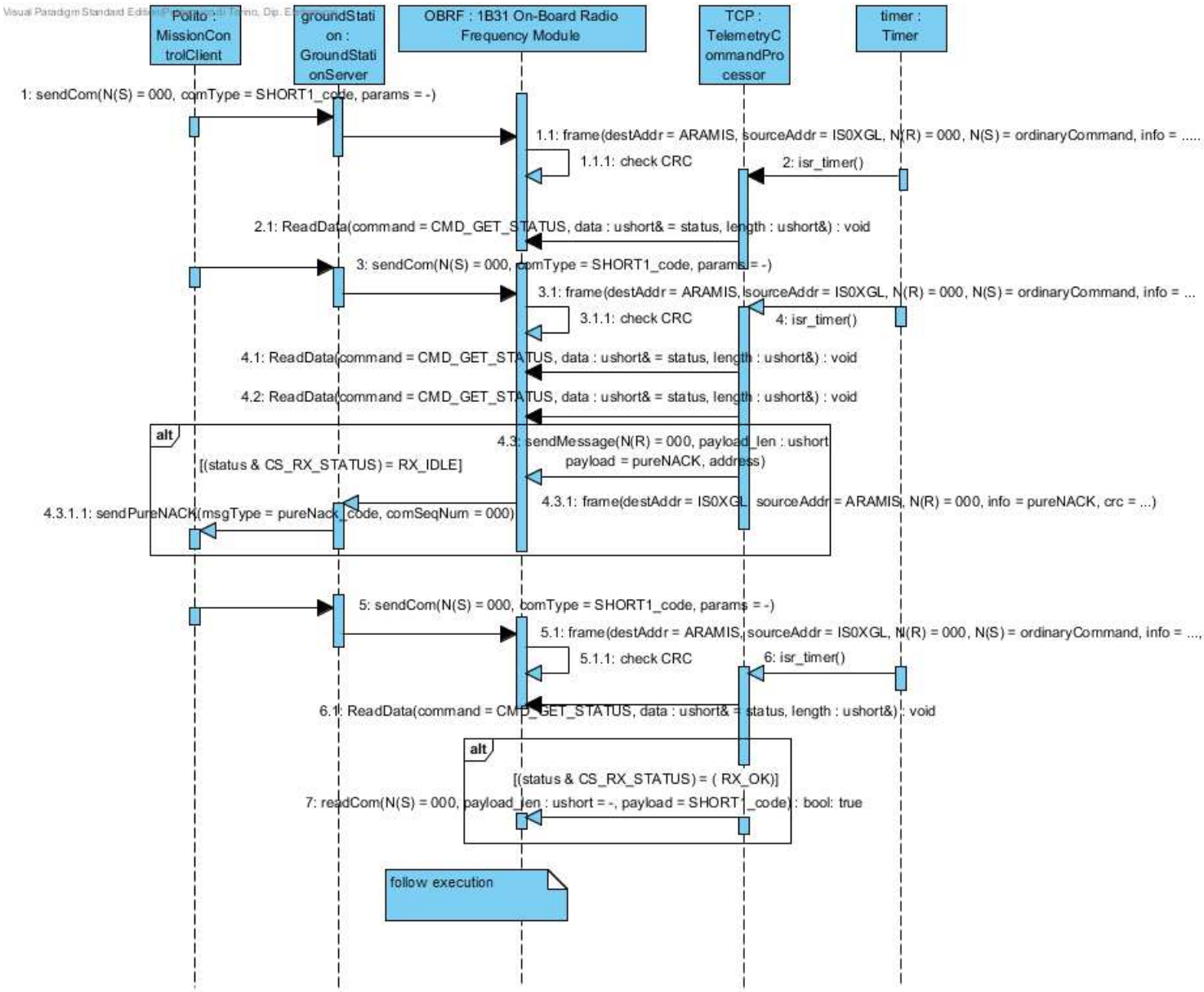

Figure 3.13. Basic Telecommunication AraMiS protocol, behaviour at the OBRF

Basic Telecommunication AraMiS protocol, fragmented packet

Previously was described a **long command** with *getLong(applNum, data &)* data where its dimension doesn't need more than one frame. Another case is presented here and shown in figures from 3.14 to 3.17, where the data need to be fragmented on more frames of type *frame(destAddr, sourceAddr, N(R), info, crc)*. This is recognized by using a GetFrag command from the Mission Control Client and sending the requested fragments with *sendCom(N(S), comType, params)* and with the fragment numbers placed in parameters *params*, related to the desired frames that will be received consecutively on the next spacecraft's transmission.

In step 1 are requested the first 4 fragment generated from the execution of an example command number 74. In step 2.3 the Telemetry Command Processor perform a Memory access to read the first fragment and generates an ACK_FRAM command with the frame's content, to be sent in downlink from step 2.5. These steps are repeated for the other 3 fragments. The fragment number 2 sent in step 2.11 is supposed to be lost. In this case, while the remaining fragments are sent, at the next request from the Mission Control Client, the fragment number 2 will be requested again, as in step 3. The GetFrag command requests are repeatable until the complete fragmented data is received. Usually the number of fragment of the requested data is known at ground, so the communication can end up with the request of the last fragment needed to complete the download.

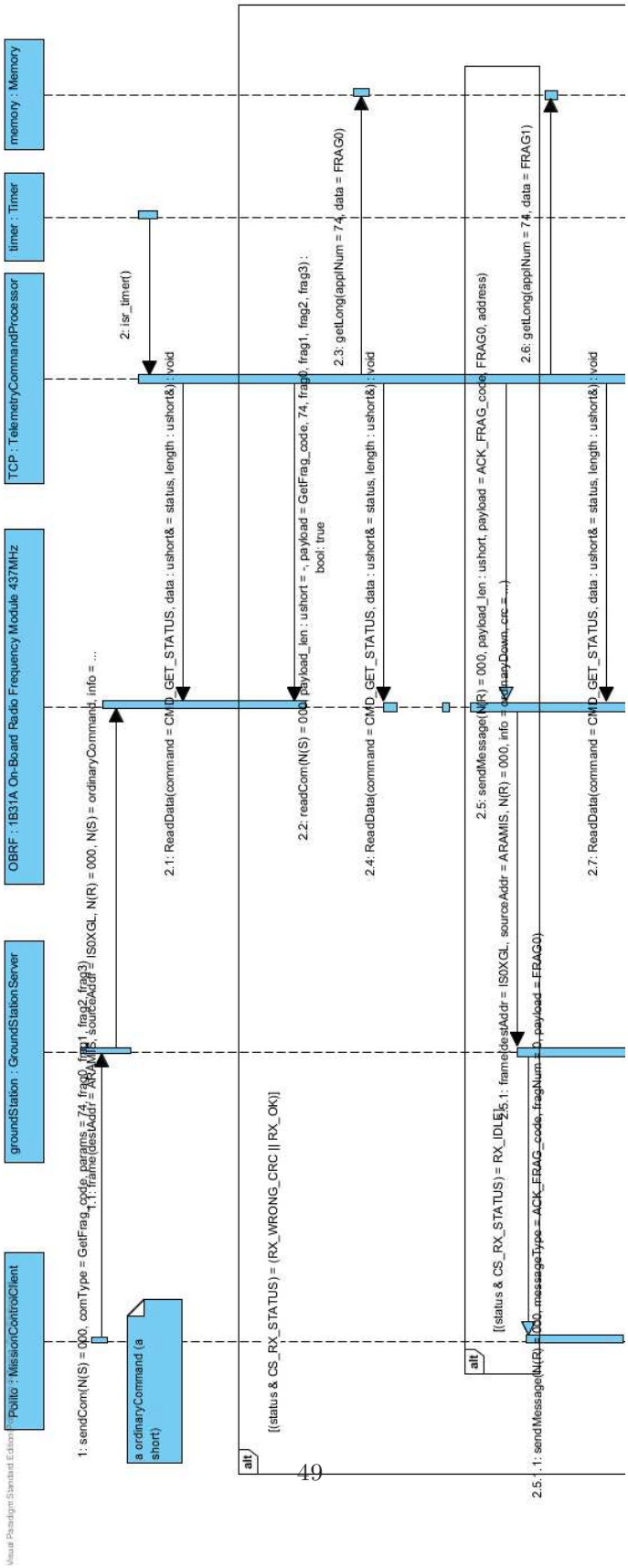

Figure 3.14. Basic Telecommunication AraMiS protocol, fragmented data (part 1/4)

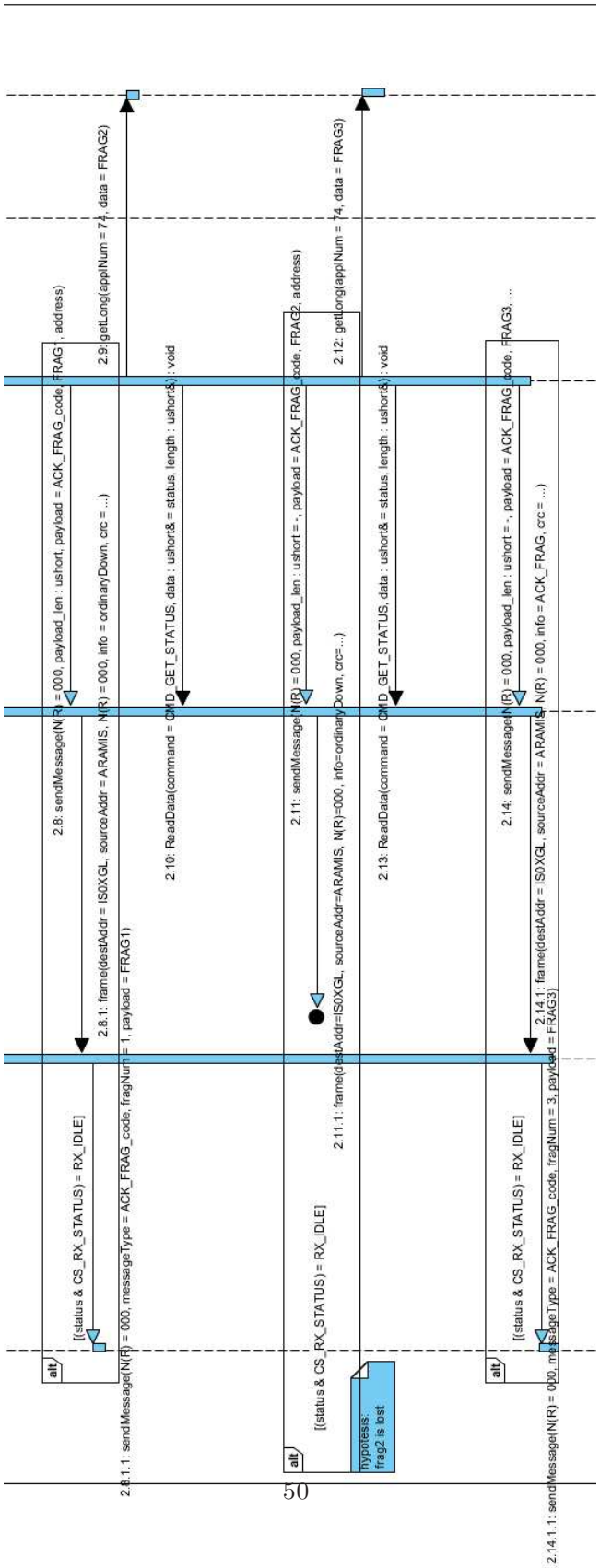

Figure 3.15. Basic Telecommunication AraMiS protocol, fragmented data (part 2/4)

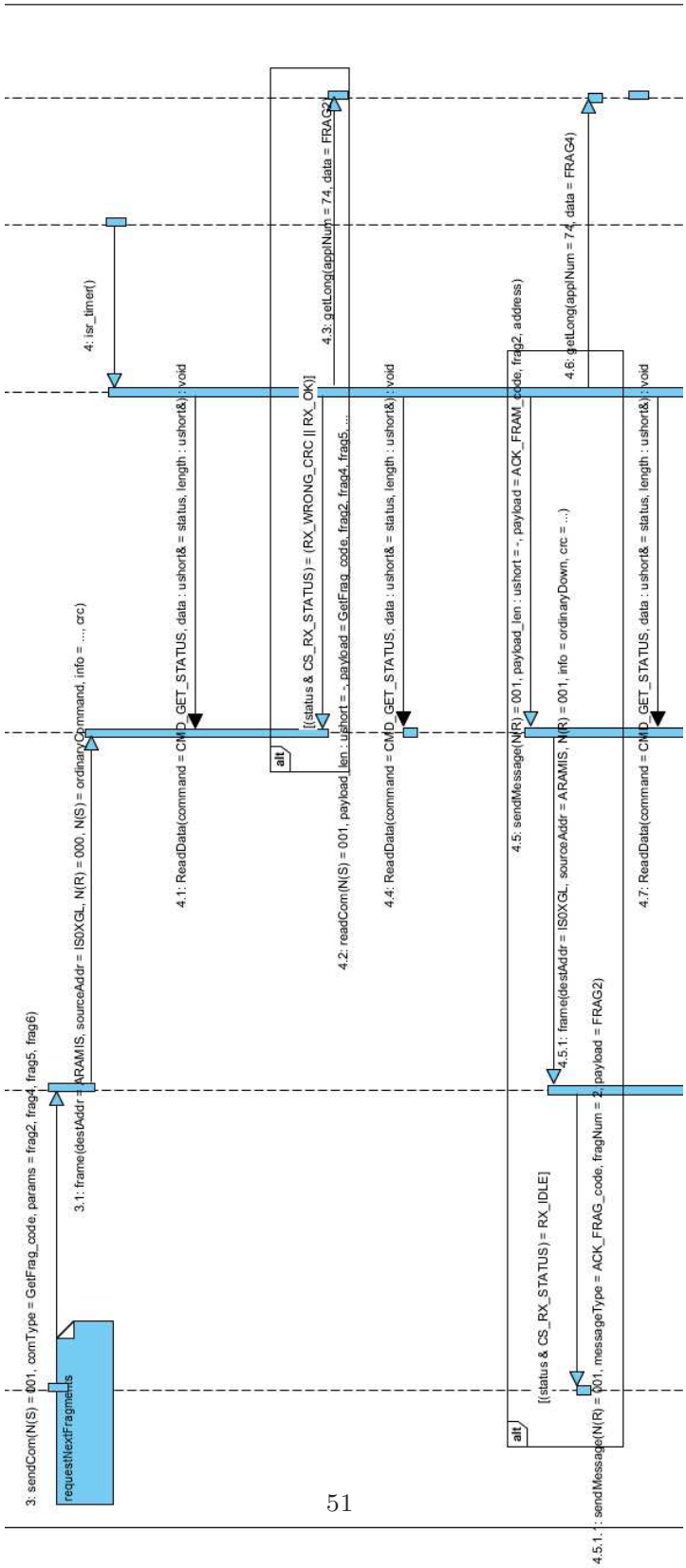

Figure 3.16. Basic Telecommunication AraMiS protocol, fragmented data (part 3/4)

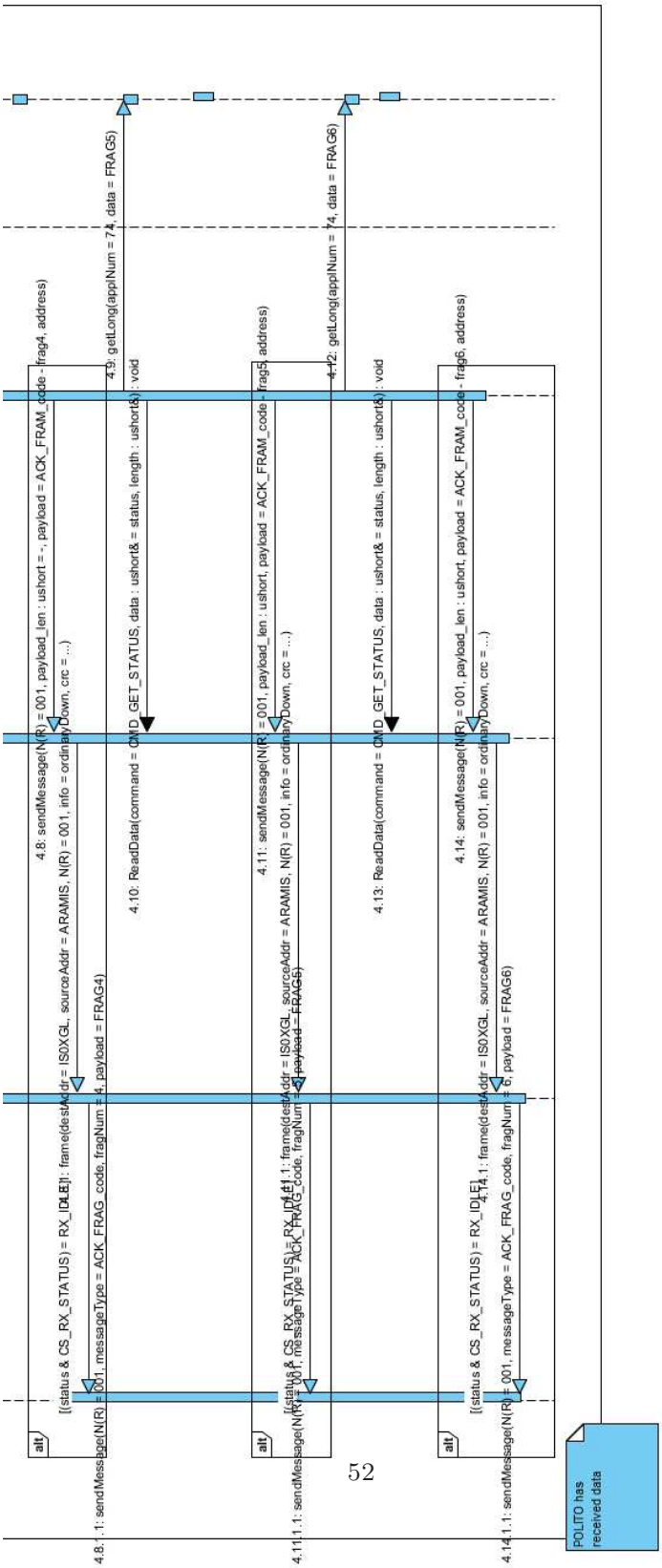

Figure 3.17. Basic Telecommunication AraMiS protocol, fragmented data (part 4/4)

Chapter 4

System constraints and use cases

The main constraints were already devised, but are here summarized and few of them are also revised. Here will be shown the frequency selection reasons, how the doppler effect could affect the channel and finally a briefing on the link budget. On the basis of these constraints, will therefore be devised a complete set of the system's use cases.

4.1 Constraints

Here are revised the constraints that will be adopted to choose the components transceiver unit in chapter 5 and configure it in section 6.3 of chapter 6.

Carrier frequency

The complete 1B31 On-board telecommunication module will use two different frequencies, allowing a real redundancy. One is the SHF band at 2.4GHz, while the other one, treated in this document, is at UHF 437MHz. This wide difference allows a low electromagnetic interference between the two channels, along as an high bandwidth per channel.

The UHF belongs to the radio-amateur bands, thus allowing a reception of the satellite data to anyone which is interested; this band is regulated by the International Amateur Radio Union allocation. The SHF band is another space which is freely available, regulated by the Industrial, Scientifc and Medical (ISM) radio bands. As a result, no additional cost is required for this logistic organization.

Doppler Effect in space environments

In LEO orbits, space-crafts are orbiting at very high speeds, appearing from one point and disappearing at the opposite horizon. This bring to a significant doppler effect which has been already devised and reported here for completeness.

In figure 4.1 is shown a simplified diagram of the various velocities. The velocity saw by the receiving point at earth is $V_a = Vcos\alpha$ and it is time varying. The frequency variation due to doppler is described by:

$$\Delta f = f_0 \frac{V_a}{c}$$

The corner case is when the satellite appear and disappear at the horizon (angle $\alpha = 0°$) obtaining almost $V_a = V$ [7][8]. The doppler effect at these speeds need to taken into account at receiving part and should be supported by the OBRF's PLL when receiving from GSS. The maximum doppler found (usually is less) is in table 4.1.

Table 4.1. Doppler effect at a given frequency and velocity

$V = 7.5\frac{Km}{s}, h = 600Km$	f_0	Δf_{max}
	437MHz	10.925kHz
	2.4GHz	61kHz

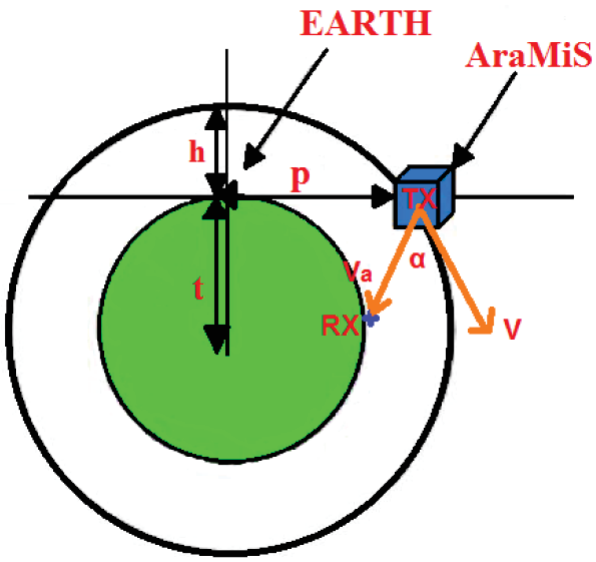

Figure 4.1. Simplified scheme of AraMiS orbit and speeds

Link budget estimation

The hardware involved on the satellite need a minimum SNR in order to achieve a reasonable BER. The link budget can be described by the Friis equation 4.1 (referred to figure 4.1):

$$P_r = P_t G_t G_r \frac{1}{\alpha_0 \alpha_p \alpha_m} \tag{4.1}$$

Where:

- P_r and P_t are respectively the received and transmitted powers
- G_r and G_t are respectively the receiving and transmitting gains of the antennas
- $\alpha_0 = (\frac{4\pi p}{\lambda})^2$ is the free space attenuation of propagation
- α_p is the polarization loss
- α_m are the medium losses (including atmospheric absorption, fading, diffraction by obstacles and ground reflection)

The equation 4.1 translated in logarithmic form became:

$$P_r|_{dBm} = P_t|_{dB} + G_t|_{dB} + G_r|_{dB} - \alpha_0|_{dB} - \alpha_p|_{dB} - \alpha_m|_{dB} \tag{4.2}$$

The distance p at worst case (satellite at horizon) is calculated using Pitagora when the elevation angle is 0°:

$$p = \sqrt{(t+h)^2 - t^2} = \sqrt{2th + h^2} \simeq 2831Km \tag{4.3}$$

Now, a quantification of the real implementation can be made with realistic values, since the system have a predetermined output power and the antenna gains are available. Losses are approximated by excess and mismatches of 1dB are applied to the satellite antenna nominal gain.

Uplink It is used from the GSS a Yagi-Uda antenna, with +47dBm with gain of 12dBi. On the satellite the deployable antenna has 0dBi, minus 1dBi to include mismatches, with a total of -1dBi. The $\alpha_0 = 155$dB, $\alpha_p = 4$ dB and $\alpha_m = 2$ dB losses are derived from above. The receiver on the satellite should then have a sensitivity which is greater than the sum of these values, so:

$$S \geq -103dBm$$

.

Downlink It is used in the satellite a commercial deployable dipole antenna, with 0dBi of gain minus 1dBi to include mismatches, with a total of -1dBi. The maximum output power from the satellite is 33dBm (3dB). The receiving Yagi-Uda antenna have a 9dBi of gain including losses. The $\alpha_0 = 155$dB, $\alpha_p = 4$ dB and $\alpha_m = 2$ dB losses are derived from above. The received signal strength from the spacecraft is the sum of these values, therefore:

$$P_r \simeq -120dBm.$$

These result are a bit more stringent w.r.t. previous defined before the OBRF engineering, but this is useful to estimate (in next pages) how weak (or not weak) can be the radiolink, on the basis of chosen COTS components.

4.2 Use case definitions of the communication channel

Here are devised the use cases of the communication channel. Are conceived starting from the requirements and the constraints. In figure 4.2 there is the diagram with actors and relations with the use cases of the 1B31A module for what concerns the channel and data handling. All the command codes used here are described better in next chapters, where here are considered in a conceptual way, in order to analyse the feasibility and flexibility of the system. In these use cases, to help start thinking on how the system can work, are defined part of the structures of logical vectors handled on the module, along with flags and bit definitions, everything at high level, most of them already defined by other use cases of other already developed projects, since every AraMiS project rely on many sub-projects. Every reference to software chapters (whether it is an FSM, class, variable or function description), refers to chapter 6.

The diagram in figure 4.2 provides the use cases of the front-end of 1B31_On-Board_Radio_ Frequency_Module. Here is documented the behaviour of the system when the OBC issue some commands on the bus (using the 1B45_Subsystem_Serial_Data_Bus sub-project), related to the RF transmission and reception. The use case documentation and recommendations are devised from the protocol in 1B3_TT&C_Telecommunication (see 3.4.2). It is described also the auto-generation of beacon.

All data is handled at OSI Layer 2 by the system and therefore the content is transparent to it, except from where specified which is at Layer 3 directly suppported by the OBRF module, because of handling the actual info of the transmitted or received frame.

4.2.1 OBC actor

The On Board Computer of the ARAMIS satellite. For small systems, the OBC can be part of a Tile Processor, that is, the OBC SW can run on one of the Tile Processors present in the system.

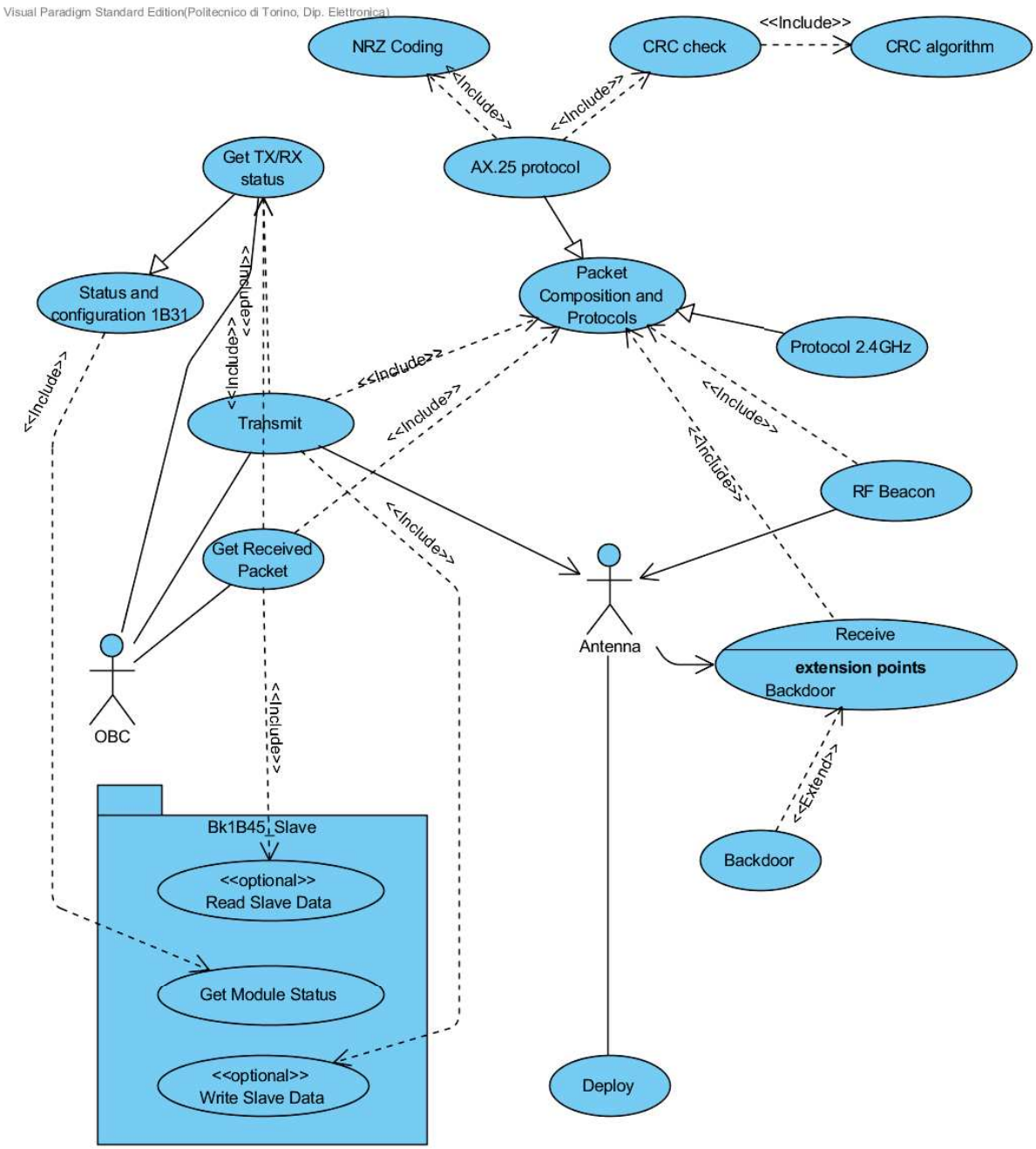

Figure 4.2. Use cases of the On-Board Radio Frequency Module 437MHz, radiolink data-handling

See also section 1.3.4.

4.2.2 Antenna actor

It is the antenna which can be connected to the external connector. See also section 1.3.8.

4.2.3 Receive

Receive and store the data from the Antenna. As soon as a transmission from Ground station begins, bits are being received from the Antenna to the system if it is not transmitting. Prior the reception of useful data, the hardware may need some synchronization bits, this triggers the RX_PREAMBLE state of the internal FSM, which has the purpose to synchronize the channel.

Every byte which is part of the payload and need to be analyzed, is stored in a buffer if the address matches the **AX_SAT_ADDR : char const***. The receiver can check at run-time if it is a Backdoor data, therefore the system supports also the reading of the actual received data, so supporting the OSI Layer 3. The OSI Layer 2 handling is supported by the Packet Composition and Protocols. The receiving mode is a default state of the 1B31 On-Board Radio Frequency Module.

4.2.4 Get Received Packet

Provides the received packet to the OBC. The OBC issue the GET_PACKAGE command only if the status of 1B31 On-Board Radio Frequency Module is either RX_OK or RX_WRONG_CRC, otherwise invalid data is provided. Then the 1B31 On-Board Radio Frequency Module responds with a content described in *readCom(N(S), payload_len : ushort, payload)* (see section 3.4.1), generated previously through the Packet Composition and Protocols.
This use case makes use of the Read Slave Data of 1B45_Subsystem_Serial_Data_Bus.

4.2.5 Transmit

Provide transmission of data from the OBC to the Ground station. The OBC first gets the TX/RX status; if status is not RECEIVING, OBC can send a system command CMD_TRANSMIT followed by data formatted as described in *sendMessage(N(R), payload_len : ushort, payload, address)* (see section 3.4.1). After the command, the 1B31 On-Board Radio Frequency Module will format the packet through the Packet Composition and Protocols handling it at the OSI Layer 2, then send it to the antenna. If the status is RECEIVING, the received data will be lost.
This use case uses the Write Slave Data of 1B45_Subsystem_Serial_Data_Bus.

4.2.6 Deploy

The deploy use case will send the opening command to the antenna upond command CMD_DEPLOY from the OBC, in order to open the baffles. This happens only once after the satellite is in orbit and is fundamental for the antenna usage. As the *transmit* use case, the deployment will use the OBC bus interface with the Write Slave Data of 1B45_Subsystem_Serial_Data_Bus. Since the manufacturer does not provide software informations before selling the product, it is not yet defined a proper I2C command to be sent to the antenna control bus.

4.2.7 Get TX/RX status

Gets the status of TX/RX transceiver 1B31 On-Board Radio Frequency Module. The Status can assume either of the following values:

- RX_OK when a whole packet has been received, no other reception is currently going on, packet CRC is correct and it has not been read by OBCÂ (in this case, internal status is RX_OK and status variable **RxStatus : t_RX_STATUS == RX_OK**)

- RX_WRONG_CRC when a whole packet has been received, no other reception is going on but packet CRC is not correct and it has not been read by OBC (in this case, internal status is RX_WRONG_CRC and internal variable **RxStatus : t_RX_STATUS == RX_WRONG_CRC**)

- RX_IDLE when there is no reception in progress and there is no any received packet in memory. Internal status is RX_IDLE and corresponds to internal variable **RxStatus : t_RX_STATUS = RX_IDLE**.

- RX_RAW when a packet has been received but it is not yet processed by the 1B31 On-Board Radio Frequency Module, therefore it is not yet available to OBC. Internal status is RX_RAW and internal variable **RxStatus : t_RX_STATUS = RX_RAW**.

- TRANSMITTING means that there is a transmission in progress, and it is interruptible. Internal status is TRANSMITTING and internal variable **RxStatus : t_RX_STATUS = TRANSMITTING**.

- RECEIVING when there is a reception in progress and it is interruptible for a transmission, despite it will delete the already received data. Internal status is RECEIVING and internal variable **RxStatus : t_RX_STATUS = RECEIVING**.

This use case is supported by Status and configuration 1B31 (section 4.2.8) and makes use of Get Module Status use case of 1B45_Subsystem_Serial_Data_Bus by reading location 1 of **statusRegister : CS_REDUNDANCY [LENGTH_STATUS]**, masked by the **MASK_CS_RX_STATUS : ushor const**.

4.2.8 Status and configurations 1B31

This use case defines the status data, read-only, in the **statusRegister : CS_REDUNDANCY [LENGTH_STATUS]**; and define the configuration data, read-write, in the **configRegister : CS_REDUNDANCY [LENGTH_CONFIG]**.

The configuration part uses the Reset Module Configuration, Set Module Configuration and Write Module Configuration use cases of 1B45 package, while the system status uses the Get Module Status only. The use cases are shown both in diagrams in figures 4.5 and 4.2. In chapter 6 there will be the technical implementation of every kind of communication and data handling for these vectors. The described vectors and the use cases in 1B45_Slave package are taken from an external AraMiS project, with name coded as 1B45 (see section 4.4), a codename different from the OBRF which is 1B31: this separation is made more clear by putting these use cases in a package (the light-blue rectangle in the figure).

The structure of **statusRegister : CS_REDUNDANCY [LENGTH_STATUS]**:

- Location 0
  1B45_Subsystem_Serial_Data_Bus reserved
- Location 1
  (ushort)**(MASK_CS_RX_STATUS : ushor const | MASK_CS_PA_STATUS : ushort const)**

The structure of **configRegister : CS_REDUNDANCY [LENGTH_CONFIG]**:

- Location 0
  (ushort)**(MASK_CS_BAUDRATE : ushort const | MASK_CS_FREQ : ushort const | MASK_CS_MODULATION : ushort const)**
- Location 1
  (ushort)**(MASK_CS_TX_POWER : ushort const)**

The order of byte and bit transmission is 1B45_Subsystem_Serial_Data_Bus defined. Any modification from the OBC should be applied to the HW if the status is **RX_IDLE**, in order to prevent data corruption, since it changes the RF configuration. For this reason, if a modification of the **configRegister : CS_REDUNDANCY [LENGTH_CONFIG]** is detected,

the affected system (1B31 On-Board Radio Frequency Module) or sub-system (1B31A On-Board Radio Frequency Module, 1B31B On-Board Radio Frequency Module) should be reinitialized in the *default* RX mode. In order to modify, for any reason, in a dependable way the various masks, any mask with name **MASK_CS_XXX** is composed by:

- The absolute value of the mask **LOCAL_MASK_CS_XXX**
- The absolute value is shifted of **SHIFT_CS_XXX** to the correct position, obtaining the final mask

4.2.9 Packet Composition and protocols

Assemble the packet to be sent over the Antenna or disassemble it according to the defined protocol. It handle the telecommunication at OSI Layer 2.

While in Transmit mode, generates only the data structure of protocol (overhead) and will be encapsulated in a *frame(destAddr, sourceAddr, N(R), N(S), info, crc)* preparing it for the transmission. When used by RF Beacon generation the behaviour is the same, because the higher OSI layer capability is managed by RF Beacon itself.

While used by Get Received Packet, disassemble the received packet and prepare it for the *readCom(N(S), payload_len : ushort, payload) : bool* (see section 3.4.1).

4.2.10 Backdoor

The 1B31 On-Board Radio Frequency Module can read the content of the received *frame(destAddr, sourceAddr, N(R), N(S), info, crc)* (see section 3.4.1). This allows, when a Receive takes place, a backdoor data recognition which is not addressed to the OBC's Tile Processor, but redirected to a set of pins which are directly connected to a proper connector. These are 6 digital signals, allowing to modify portions of the programmable digital hardware, external to the 1B31 On-Board Radio Frequency Module.

The 7 bits of data in received *info* field from *frame(destAddr, sourceAddr, N(R), N(S), info, crc)*, are redirected to backdoor connection, if the received OSI Layer 3 command **CommandCode : t_OBRF_DEF_COMMAND_CODES = CMD_BACKDOOR**. The backdoor connection is made of 6 pins + 1 of reset, at BACKDOOR() group of pins in class diagrams (see diagram in figure 5.15 in chapter 5). The structure of payload is provided here, since it is handled at OSI Layer 3 by the system, and therefore should be known for the software implementation:

- payload [0][1] = CMD_BACKDOOR (16bits)
- payload [2] contains, from LSB to MSB, the:

- BACKDOOR_0
- BACKDOOR_1
- BACKDOOR_2
- BACKDOOR_3
- BACKDOOR_4
- BACKDOOR_5
- BACKDOOR_INT (reset signal)

The unused bits are 0.

4.2.11 RF Beacon

Send an auto-generated beacon. Is described in figures 4.3 and 4.4. This happens when OBC do not issue the commands CMD_ GET_STATUS, CMD_TRANSMIT, or GET_PACKAGE to the system after an **OBC_ TIMEOUT : byte const** time. In sequence diagram The content of the beacon is put in a packet through the Packet Composition and Protocols handling the OSI Layer 2 which is sent containing in the payload the **CommandCode : t_OBRF_DEF_COMMAND _CODES = RF_BEACON** with the **housekeeping : HK_REDUNDANCY [LENGTH_ HOUSEKEEPING]** and statistics in **statusRegister : CS_REDUNDANCY [LENGTH_ STATUS]** all together. The **CommandCode : t_OBRF_DEF_COMMAND_CODES** is contained in the payload of *sendMessage(N(R), payload_len : ushort, payload, address)* (see section 3.4.1), therefore it is a Layer 3 command in the OSI stack and so the content is described here for the software implementation.

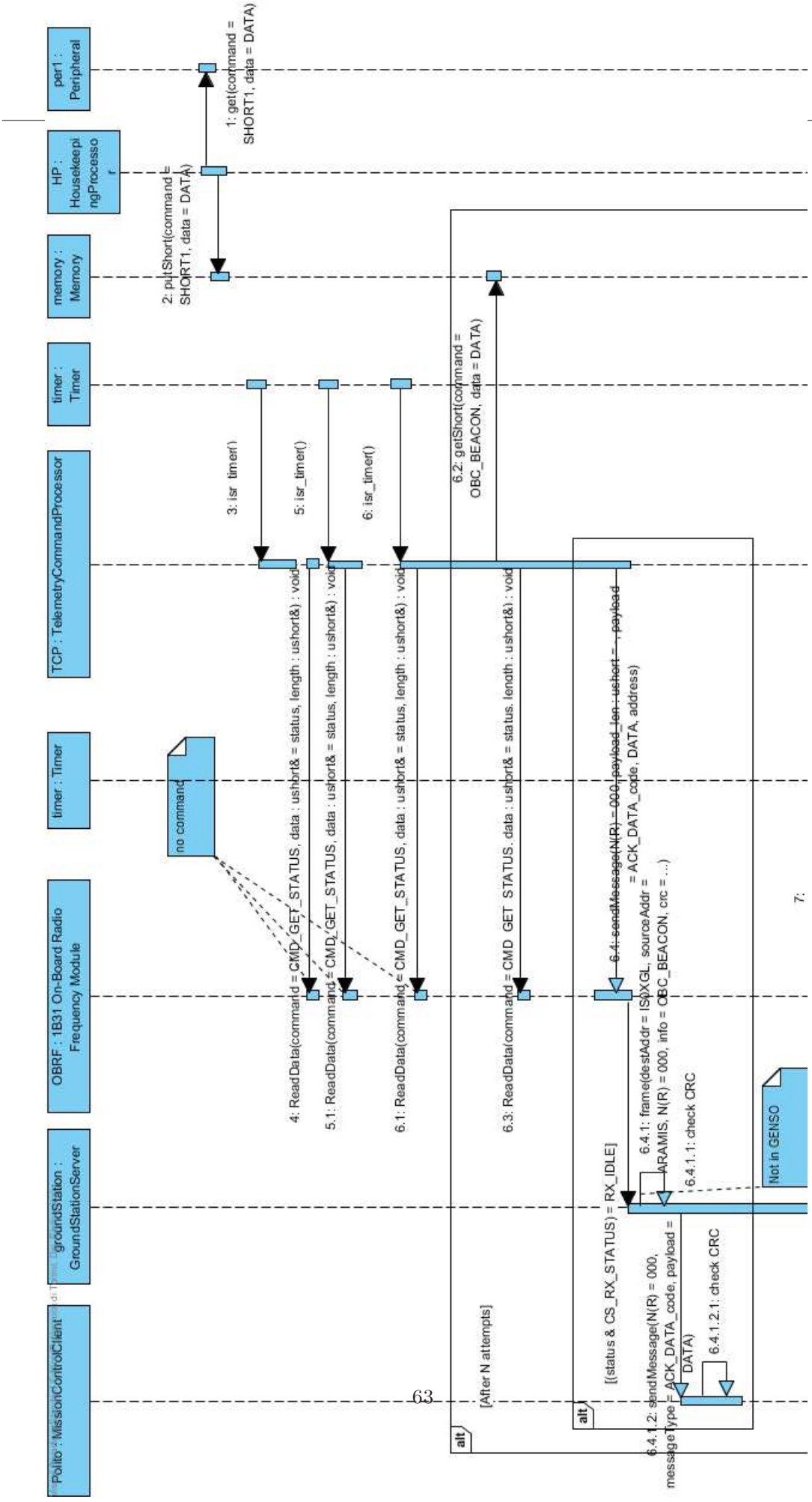

Figure 4.3. Sequence diagram of the RF Beacon, part 1/2

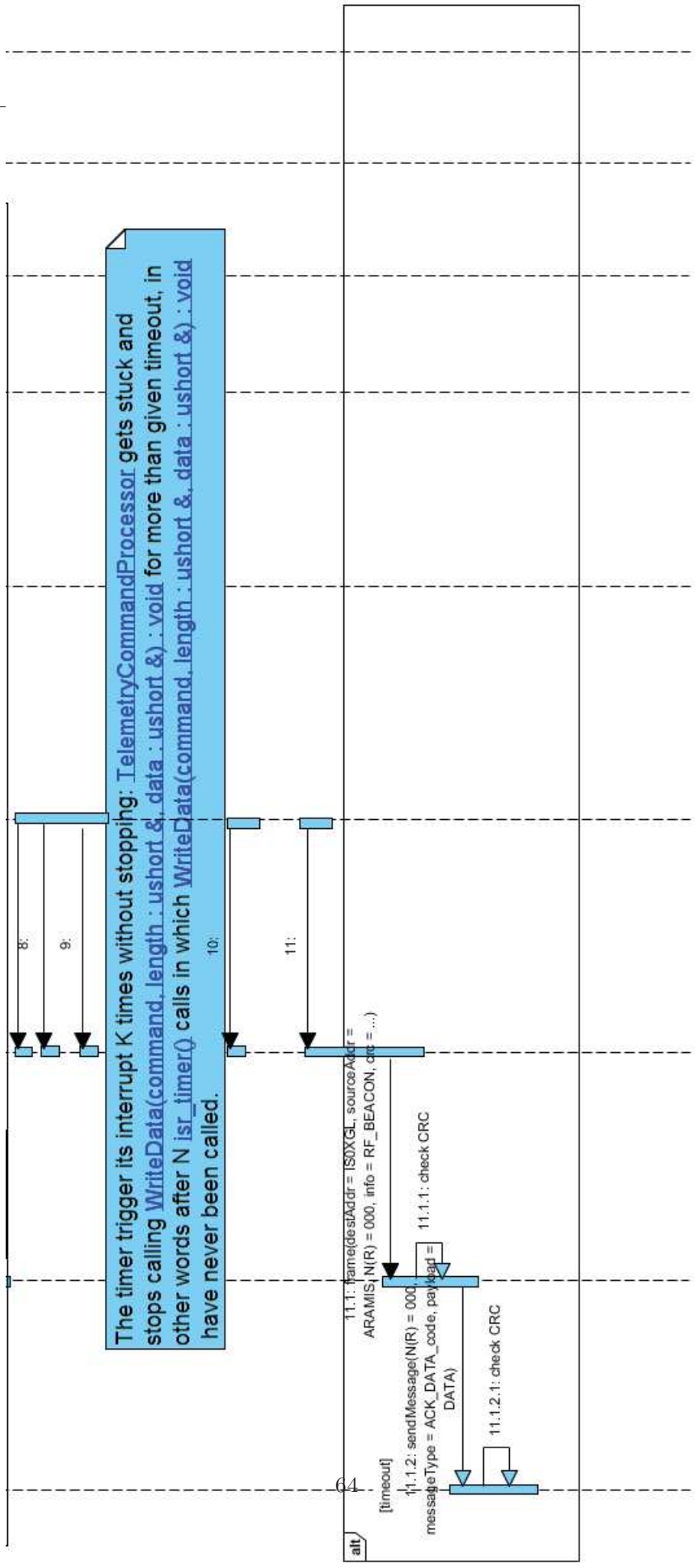

Figure 4.4. Sequence diagram of the RF Beacon, part 2/2

Content of payload

In *sendMessage(N(R), payload_len : ushort, payload, address)* (described in section 3.4.1) the N(R) = 0 and address is the destination one of type **addressGround : uchar[7]**, while the content of the payload is structured as follow, where each location of payload is 8-bit wide:

- payload[0][1] contains the 16bits **CommandCode : t_OBRF_DEF_ COMMAND_ CODES**.
- payload[2] the **housekeeping : HK_REDUNDANCY [LENGTH_ HOUSEKEEPING]** length of words LENGTH_HOUSEKEEPING
- payload[3] to [3+2*LENGTH_HOUSEKEEPING] contains the **housekeeping : HK_ REDUNDANCY [LENGTH_HOUSEKEEPING]**
- payload[3 + 2 * LENGTH _HOUSEKEEPING] the **statusRegister : CS_ REDUNDANCY [LENGTH _STATUS]** length of words LENGTH_STATUS
- payload[4+2*LENGTH_HOUSEKEEPING] to [4+2* LENGTH_HOUSEKEEPING + 2* LENGTH _ STATUS] contains the **statusRegister : CS_REDUNDANCY [LENGTH _STATUS]**

Therefore the payload_len = 4+2*LENGTH_HOUSEKEEPING+2*LENGTH_STATUS, in bytes.

4.3 Housekeeping and module configuration

In figure 4.5 are shown the interactions between the OBC, various sensors, the configuration and status registers, which are used to parse informations from/to the OBC. For what concerns the sensors, here are not described the sensors hardware and software routines (which is done in next chapters), but rather the representation of the logical data when the actors are interfacing with the OBRF in order to read the sensor's value.

4.3.1 Channel selection

The OBC can choose the channel of 1B31 On-Board Radio Frequency Module among some predefined frequencies defined at compile-time among a Frequencies list. This selected channel is present in **configRegister : CS_REDUNDANCY [LENGTH_CONFIG]**, at location 1, in **MASK_CS_FREQ** field.

Values and associated frequencies are listed below:

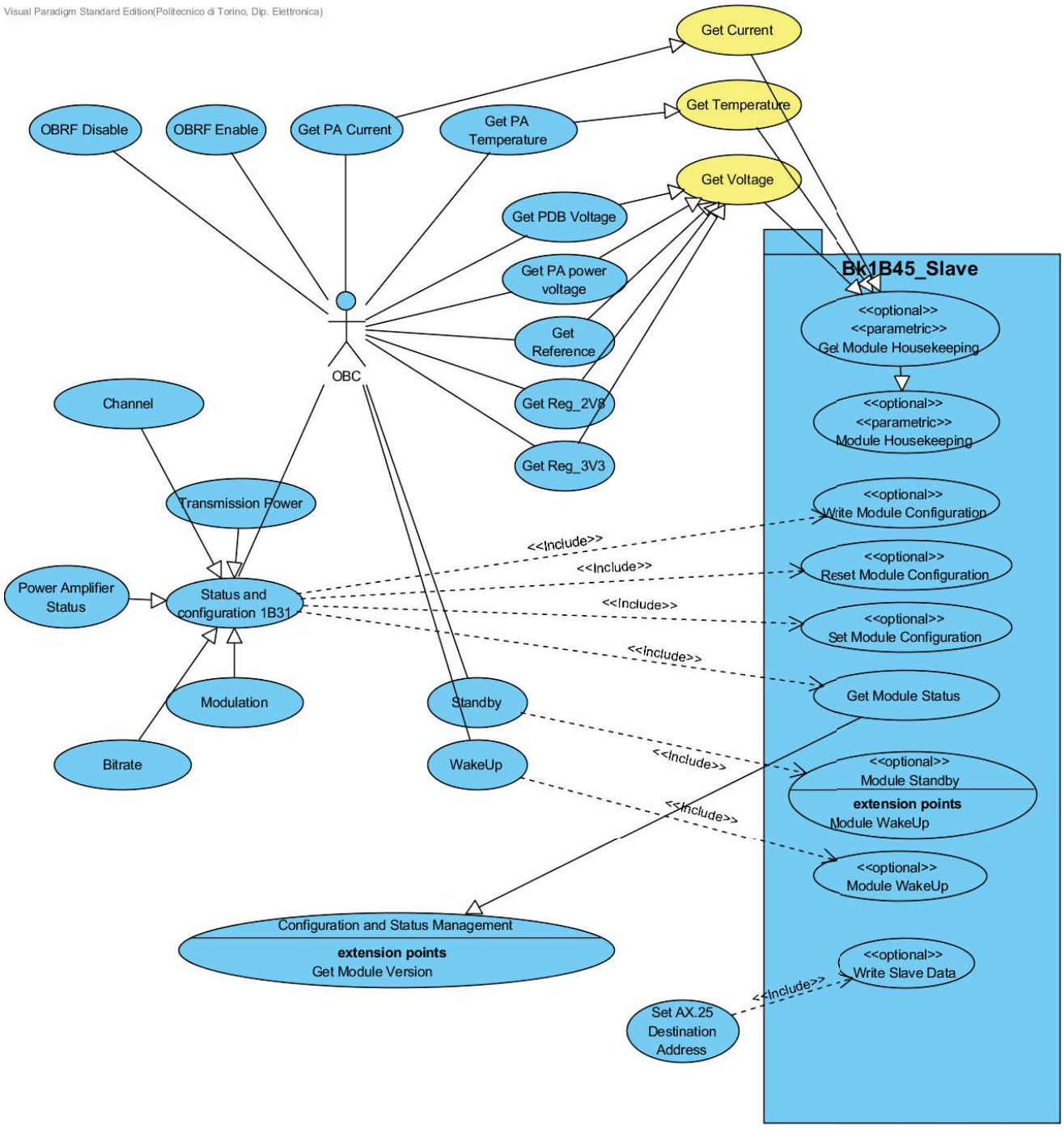

Figure 4.5. Use cases of the On-Board Radio Frequency Module 437MHz, housekeeping and configuration management

- FREQ1 : ulong const = 0;
- FREQ2 : ulong const = 1;
- FREQ3 : ulong const = 2;
- FREQ4 : ulong const = 3;

This use case is supported by Status and configuration 1B31, where provides the structure of registers.

4.3.2 Get Power Amplifier Status

Provide to OBC the status of the power amplifier, usually needed for diagnosis purposes. This value is present in **statusRegister : CS_REDUNDANCY [LENGTH_STATUS]**, location 1, at field MASK_CS_PA_STATUS. The corresponding values are boolean, PA off = 0; PA on = 1. This use case is supported by Status and configuration 1B31.

4.3.3 Set/Get Transmission Power

The OBC can set different RF power levels of the transceiver. These values are only qualitative, because are used to range from the minimum to the maximum allowable settings of the transceiver, so the absolute value is hardware dependent. Their physical meaning is therefore hardware dependent. The absolute outputted power must be considered multiplied by the gain of the power amplifier, if present.

This value is present in **configRegister : CS_REDUNDANCY [LENGTH_CONFIG]**, at location 1, at field masked by **MASK_CS_TX_POWER**. This use case is supported by Status and configuration 1B31.

4.3.4 Set/Get Modulation

The modulation of the RF channel can be changed by the OBC among FSK or GFSK. This value is present in **configRegister : CS_REDUNDANCY [LENGTH_CONFIG]**, location 1, at field **MASK_CS_MODULATION**. The association of the value with the meaning is FSK = 0; GFSK = 1. This use case is supported by Status and configuration 1B31.

4.3.5 Set/Get baudrate

The OBC can specify the baudrate (kBaud per second, kbps) of the RF channel, expressed in bits per second if NRZ Coding, or it is the double of bits per second, if Manchester coded.

This use case allow to use the tile with different specifications with respect to the current usage, since are possible different values than 9600 baud per second. Its baudrate value is mapped in the **configRegister : CS_REDUNDANCY [LENGTH_CONFIG]**, location 1, at field **MASK_CS_BAUDRATE**.

Despite this, *it is not advised to change the baudrate*, because it will change the channel SNR as long as a different need of passive components, leading to a not well configured hardware and different sensitivity of the system. For this reason it cannot be changed at run-time, but only at compile time, modifying the **baud : ulong** variable.

Values and meanings are listed below:

- 2.4 kbps = 0;
- 4.8 kbps = 1;
- 9.6 kbps = 2;
- 19.2 kbps = 3;
- 38.4 kbps = 4;
- 76.8 kbps = 5;
- 153.6 kbps = 6;
- Setting not allowed = 7

This use case is supported by Status and configuration 1B31 at section 4.2.8.

4.3.6 Standby

The system can be put in standby by OBC, with command CMD_STANDBY where internal processor is in sleep mode but is able to listen from the bus; housekeeping sensors are either disabled or shutdown RF hardware is disabled. When in Standby mode, the System can listen any command coming from the OBC. The OBRF can not receive data from Antenna, therefore should be awaken by OBC automatically after some time. It uses the Module Standby use case of 1B45_Subsystem_Serial_Data_Bus.

4.3.7 Wakeup

The system can be awaken from OBC with command CMD_WAKEUP and internal processor start running normally; housekeeping sensors are enabled; RF RX/TX hardware is enabled along with the modules that are put in standby before. It uses the Module WakeUp use case of 1B45_Subsystem_Serial_Data_Bus.

4.3.8 OBRF enabling

The 1B31 On-Board Radio Frequency Module, can be enabled by the OBC, by the enable signal, active high, present in Bk1B481W_Module_Slot, here named MODULE_OBC(). In order to provide selectivity,should be present at least one connector with this signal per sub-module of the 1B31 On-Board Radio Frequency Module. It is equivalent to physically connect the power to the selected sub-module.

4.3.9 OBRF disabling

The 1B31 On-Board Radio Frequency Module, can be disabled by the OBC, by the enable signal, active high, present in Bk1B481W_Module_Slot, here named MODULE_OBC(). In order to provide selectivity, should be present at least one connector with this signal per sub-module of the 1B31 On-Board Radio Frequency Module. It is equivalent to physically disconnect the power from the selected sub-module.

4.3.10 Get PA Current

Get the last acquired value of total current consumption of the board, mainly related to the power amplifier, recording it in **housekeeping : HK_REDUNDANCY [LENGTH_ HOUSEKEEPING]** at HK_CURRENT field, providing it to the OBC. It is supported by the Get Module Housekeeping.

4.3.11 Get PA Temperature

Gets the last acquired value of the board temperature, recording it in **housekeeping : HK_ REDUNDANCY [LENGTH_HOUSEKEEPING]** at HK_TEMPERATURE field, and provides it to the OBC. It is supported by the Get Module Housekeeping. The value is related to the component which can generate more heat than others, which is the power amplifier; to provide this value, the sensor should be placed as near as possible to the component. After production, some tests can be performed to determine what is the relation between the temperature of the sensor and the actual one present on the PA.

4.3.12 Get Voltage

There are few voltages present on board. These can vary a lot between each others, and more different voltage rails should be monitored; for this reason different sensors are required.

Get PDB Voltage

Gets the last acquired value of power distribution bus (PDB) voltage, recording it in **housekeeping : HK_REDUNDANCY [LENGTH_HOUSEKEEPING]** at HK_PDB_VOLTAGE field. The value is provided to OBC upon request. It is supported by the Get Module Housekeeping. The PDB voltage is the power distribution bus voltage, which can be quite high (up to 20V), and the board can absorb a lot of energy from these power rails. It is ideal of using it for the RF transmitter.

Get PA Voltage

Gets the last acquired value of 3V power line voltage, recording it in **housekeeping : HK_ REDUNDANCY [LENGTH_HOUSEKEEPING]** at HK_VPA_VOLTAGE field. The value is provided to OBC upon request. It is supported by the Get Module Housekeeping.

Get Reference Voltage

Gets the last acquired value of reference voltage, recording it in **housekeeping : HK_ REDUNDANCY [LENGTH_ HOUSEKEEPING]** at HK_REF_VOLTAGE field. The value is provided to OBC upon request. It is supported by the Get Module Housekeeping.

Get Reg 2V8

Gets the last acquired value of regulation voltage of the power amplifier, recording it in **housekeeping : HK_REDUNDANCY [LENGTH_HOUSEKEEPING]** at HK_2V8_VOLTAGE field. The value is provided to OBC upon request. It is supported by the Get Module Housekeeping.

Get Reg 3V3

Gets the last acquired value of 3V3 voltage, recording it in **housekeeping : HK_ REDUNDANCY [LENGTH_ HOUSEKEEPING]** at HK_3V3_VOLTAGE field. The value is provided to OBC upon request. It is supported by the Get Module Housekeeping.Â

4.3.13 Set AX.25 Destination Address

The OBC can chose the destination address of the radio link at run-time. Once it has been set, all transmissions are sent with that address upon next change. If not set, a default address is used. It uses a Write Slave Data, with *WriteData(command, length : ushort &, data : ushort &) : void* (see section 3.4.1) when command = CMD_SET_ADDR. The data parameter contains only the

destination address as described in AX.25 protocol. A default address is used if not set. This address could be different from the source address of an incoming packet from Ground Segment. The left-shift, to comply the AX.25 protocol, is performed by the OBRF automatically.

4.3.14 Configurator actor

The person in charge of configuring HW/SW parameters according to spacecraft architecture and mission requirements. In figure 4.6 are described the interactions between the module and the configurator in charge of configuring HW/SW parameters according to spacecraft architecture and mission requirements.

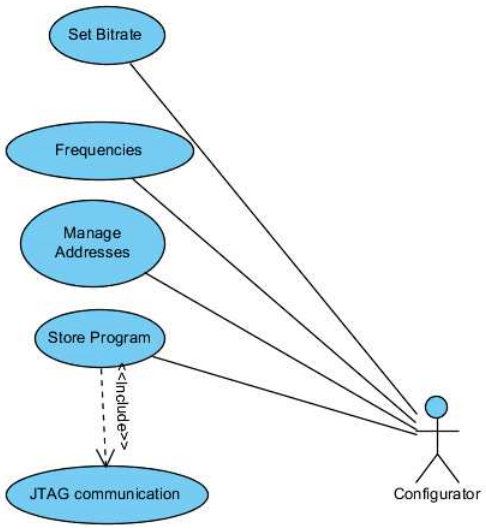

Figure 4.6. Roles of the technician configurator

4.3.15 Frequencies

Here are listed the possible frequencies to be used when designing the satellite, according to the available band, 437MHz nominal. Configures at compile-time the frequencies of up to four channels associated with the satellite (from AllowedFrequencies class, which include a set of usable values), named **FREQ1 : ulong const**, **FREQ2 : ulong const**, **FREQ3 : ulong const**, **FREQ4 : ulong const**.

4.3.16 Manage Addresses

The Configurator at compile time manages the spacecraft addresses. When sending to Ground Segment an auto-generated frame, the destination address used can be a default one AX_DEFAULT_

DEST_ADDR : char const*, while the satellite address AX_SAT_ADDR : char const* cannot be modified once are set. The OBC can change the default destination address.

4.3.17 Firmware storing and JTAG

Is provided the possibility, by means of the configurator in charge, of uploading the firmware to the OBRF board. This is done by equipping the system of a JTAG test/debug interface.

4.4 On-Board communication protocol 1B45 Subsystem Serial Data Bus

This section regards to a serial data exchange organization which follow the same rules for all the AraMiS satellites, therefore it is an external project. It can be applied to various bus protocols, but in the OBRF the I2C is used.

4.4.1 Overview of the 1B45 system protocol

In figure 4.7 are shown functions and flavors of Basic Communication Protocol of the 1B45 Subsystem Serial Data Bus subsystem for the AraMiS architecture. The Basic Communication Protocol supports communication between one Master (usually, either the OBC or the Tile Processor external to the OBRF) and one or more Slave(s) (here the OBRF itself). For the system design it has been used as much as possible the support from this protocol and its support for his specific modules.

As shown, the Basic Communication Protocol provides several basic functions for the AraMiS architecture, which are grouped in at least four groups, which are detailed in the corresponding diagrams and used in digrams in figures 4.5 and 4.2:

- Configuration and Status Management
- Housekeeping Management
- User Defined Messages and Commands
- Supervision and Emergency Recovery

The Basic Communication Protocol has five different options of use: Command Only, Read Data, Write Data, Broadcast Command Only and Broadcast Write Data which differ for the direction of data transfer and the number of Slaves involved. Those are mentioned in section 3.4.1. This communication protocol is here implemented in I2C, but there is support for a lot of other bus

protocols, like: SPI, RS232, IrDA protocol, OBDB, Wireless and IntraBoard protocols, which differ for the details of the physical support and data rate.

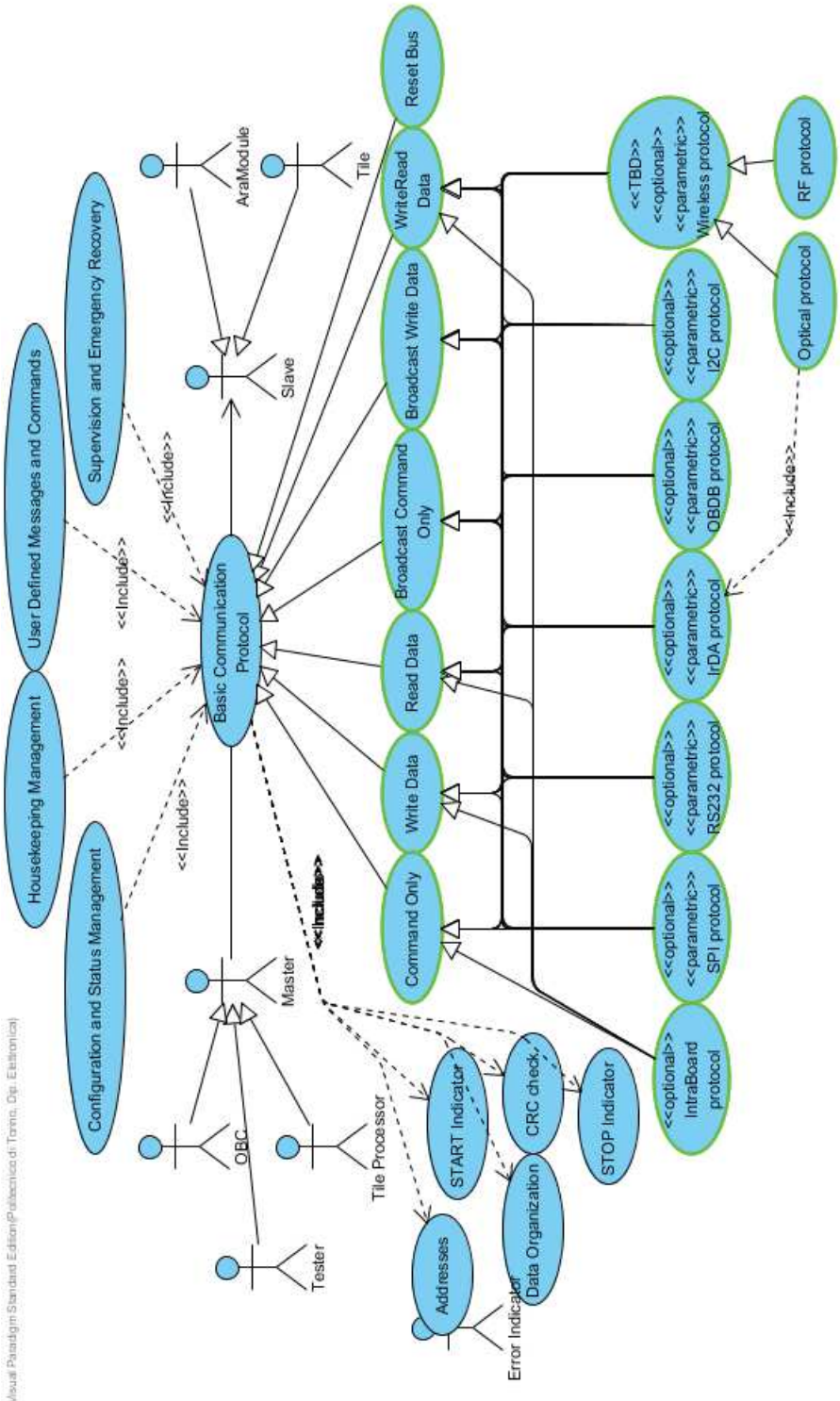

Figure 4.7. The use case diagram of communication protocol adopted in the OBRF

I2C protocol in the 1B45 Basic Communication

This Basic Communication Protocol is implemented using the standard I2C serial protocol for communication. The Tile Processor (here the OBC) actor is the I2C master, while an AraModule (here the OBRF) is a slave. The I2C protocol supports all the non-broadcasting actions of the Basic Communication Protocol, namely: Write Data, Read Data and Command Only. The broadcasting actions (namely, Broadcast Write Data and Broadcast Command Only) are implemented by sending the same message to ALL Slaves in sequence.

I2C protocol complies with the Basic Communication Protocol with:

- START Indicator is implemented by lowering the SDA() signal when SCK() signal is high (I2C START sequence);
- STOP Indicator is implemented by rising the SDA() signal when SCK() signal is high (I2C STOP sequence).

The I2C protocol uses RZ signaling, open-collector TTL logic level, I2C timing, with variable baud rate. Max rate is defined by BAUD_RATE parameter.

1B45 Basic Communication Protocol

Are supported various type of commands, sent in half-duplex mode, between slaves (the OBRF here) and only one master (the OBC, which initiates communication). This communication Protocol supports the following different actions, depending on the command issued. Here are listed only the modes supported by the actual version of the OBRF:

- Write Data mode - when a Master wants to transfer up to 256B of data to a Slave;
- Read Data mode - when a Master wants to read up to 256B of data from a Slave;
- Command Only mode - when a Master issue a command without any other data nor slave response.

Each data transfer follow this protocol, where these four points are common to each mode:

- The master set an appropriate START Indicator; in I2C is a start condition
- (i) Master sends an 8-bit Slave address to address a specific Slave, including the read/write bit;
- (ii) Then master send an 8-bit Master address, which is used by the slave for checking reasons;
- (iii) Master send an additional 16-bits command;

Then, if it is a Write Data mode, the transfer is always from Master to Slave:

- (iv) The master sends again an 8-bit data, indicating the field length in bytes;
- (v) Payload data, 1B to 256B.

If it is Command Only mode, previous two points (iv) and (v) should not considered and only the following two ones are used, which are common to Write Data mode:

- (vi) A 16-bit CRC variable with classic (which means reversed) bit order. CRC algorithm is a CRC-16 of all bytes (including command/ID and, only if Write Data mode, length fields)
- (vii) an appropriate STOP Indicator; in I2C is a stop condition

Or alternatively to the Write Data and Command Only mode, there is Read Data mode which trigger this actions after the 16-bit command, where the transfer is from Slave to Master. Therefore point (iv), (v), (vi) and (vii) are not considered, replaced by the following ones:

- (viii) slave send an 8-bit SlaveID to identify the Slave type;
- (ix) an 8-bit data, indicating the field length in bytes
- (x) Payload data, 1B to 256B
- (xi) a 16-bit CRC check. CRC algorithm is a CRC-16 of all bytes (including command/ID and length fields)

If an error occurs (either wrong CRC or wrong length or no memory available, etc.) the Slave internally sets an **ErrorFlag : bool** and does not send any answer. A particular use case is provided, Get Module Status in section 4.4.2, allowing Slave to read details on the last error and clear the **ErrorFlag : bool**.

The 1B45 Subsystem Serial Data Bus support the whole housekeeping, status and configuration vectors management. In pictures 4.2 and 4.5 is present a light blue folder named *Bk1B45_Slave*. This folder is picked from the 1B45 Subsystem Serial Data Bus, in parts relative to the slave behavior. The AraMiS hierarchy (see section 1.3.7) has brought the possibility to reuse this protocol, already defined, by adapting it to the current 1B31 OBRF project.

4.4.2 Basic functions supported by the 1B45 Slave

As can be seen in pictures 4.2 and 4.5, various use cases are relying on other use cases present in this folder named *Bk1B45_Slave* (also called package), which are briefly listed here. Are supporting the Configuration and Status Management, namely for the exchange of status, configuration and housekeeping bits to/from the central OBC and one or more Tiles (here the OBRF). Due to

modularity, despite this is not the case, for small systems the OBC can coincide with one specific Tile Processor.

Get Module Housekeeping

This use case provides to return the last measured housekeeping data. This data is organized in the following way: the Get Module Housekeeping use case returns the last acquired **housekeeping : HK_REDUNDANCY [LENGTH_HOUSEKEEPING]** vector. The Master shall operate in Read Data mode (see section 4.4) by issuing the **command = CMD_GET_HOUSEKEEPING**. The Slave shall assemble ALL last saved housekeeping data into the response message and return them to the Master. No consistency is guaranteed between sampling time of different housekeeping data; it may therefore happen that some values have been just sampled, while others may be several seconds old, depending on sampling rate of the Module Housekeeping. Here is provided the constant **LENGTH_HOUSEKEEPING** used also in previous chapters, which namely is the number of elements (different sensors, measurements, other data) that will be stored into the **housekeeping : HK_REDUNDANCY [LENGTH_HOUSEKEEPING]** vector.

Write Module Configuration

Overwrites all bits of the internal configuration word (**configRegister : CS_REDUNDANCY [LENGTH_CONFIG]**). The OBC shall send to the Tile (or the sub-module) as many bits as are in its configuration word configRegister : CS_REDUNDANCY [LENGTH_CONFIG]. Each bit will overwrite the corresponding bit in the Tile configuration word. This use case sends configuration bits by communicating in the Write Data mode, by issuing the **command = CMD_WRITE_CONFIGURATION** with the address of the OBRF. All bits are overwritten. Since this payload formatting is not considered in this project because it rely to external code (in 1B45, instead of the 1B31 core code), it is not specified here. A part from these considerations, the structure of the vectors is important because, in one way or another, the final content of the vectors must be the examined by the code present in 1B31 OBRF and therefore should be documented here. Part of it is already described in section 4.2.8. configRegister[0] can be written, as this contains the Designer-defined HW/SW version.

Note that can be used also **command = CMD_SET_CONFIGURATION** and **command = CMD_RESET_CONFIGURATION**: it does not matters what bit exactly the OBC had modified, what is important is that upon these three commands the OBRF check the configuration and eventually update the system accordingly.

Get Module Status

It returns to the OBC the status information (**statusRegister : CS_ REDUNDANCY [LENGTH_STATUS]**) of OBRF. This use case works using only the Read Data mode on the bus, by issuing the **command = CMD_GET_STATUS** with the address of the desired Tile, which returns its **statusRegister : CS_REDUNDANCY [LENGTH_STATUS]** to the Master. Using this use case also clears the Error Indicator signal (if present).

Chapter 5

Hardware

In this chapter is shown the new revision of the designed hardware of the On-Board Radio Freqeuncy Module at 437MHz. Will be provided the hardware description at UML level, in order to keep coherency and modularity with the whole project; to accomplish this, will be provided the sequence and class diagrams that are related to hardware. These diagrams are closely connected with the chapter 6 and 3, since the hardware is dependent on the constraints, use cases and software requirements.

In UML are therefore defined the external interfaces, the relations of the hardware with the other modules and its internal sub-modules. The actual hardware design is then obtained using Mentor Graphics Expedition Enterprise suite. In this work the final hardware design is limited to the PCB, which in AraMiS is considered also a tile of the satellite.

5.1 Hardware organization

The class diagram of the final module implementation which forms the tile is shown in figure 5.2, is called Bk1B31A2M where are instantiated the hardware classes, including the top-level wire schematic, called Bk1B31A2W, shown in figure 5.1. At the Bk1B31A2M level are contained also the interfacing function described in section 3.4.1, for the highest level of logical behaviour, which will be implemented by the complete system. Moreover, the Bk1B31A2M contains also the mechanical connectors and the PCB placement. While the Bk1B31A2W module implementation simply contains the hardware without of external mechanical connectors and with no PCB layout placed in a tile, so that can be reused in different projects. In appendix B is reported the complete BOM contained in this design. The hardware design begins using the UML tool. As stated in previous chapters, the concept of object programming is used also in hardware, where an object of a class is now a physical object.

This hardware revision consist of a complete reorganization of classes in UML, the complete re-organization of components which were already defined in previous thesis works, using Mentor Graphics; therefore is check and eventually re-designed, for each component, the cell and updating the new part numbers. Moreover, all the axternal connectors were revised. Then maximum effort is put towards the schematic reuse, by means of creating the reusable blocks in the central library. In chapter 7 is performed a complete new PCB placement, in order to stay inside the space constraints of the AraMiS-C1 tile, where the tile's space is used also by the 2.4GHz module, named 1B31B-OBRF.

Each class is a main hardware block, referred as a Reusable Block in Mentor Graphics. In figure 5.1, the hardware classes are yellow, in orange the components and in green the software classes and defines. In fact, some hardware is directly dependent in software and in UML this dependence is easily noticeable. Going in the lower layers of the design, the top-level is represented by the *Bk1B31A2W_OBRF_437MHz* class, which contains four objects inside, the *Bk1B4221W_Tile_Processor_4M*, the *Bk1B31A2_Power_Supply*, the *Bk1B31A2_Sensors* and the *Bk1B31A2_Transceiver_437MHz*. In figure 5.1 are shown arrows going towards the Tile Processor's class and the CC1020 class components. These are the connections with the software class diagrams, shown in figure 6.3 and also present here, named *1B31A2S* software class. Therefore, the union between hardware and software is made through the software objects instantiations, that are driving the Tile Processor and the CC1020 chip transceiver.

5.2 Design of OBRF at wire level Bk1B31A2W and the top-level module Bk1B31A2M

The class *Bk1B31A2W_OBRF_437MHz* is also a reusable block in Mentor Graphics, allowing the utilization in more projects with different modules (for example integrating the OBRF with other systems and in various missions). The sign **W** in the name represents the **Wire** level of the schematics.

The *Bk1B31A2M_OBRF_437MHz* carries the connectors for control and power interface, included in MODULE_OBC(), which is connected to an external I2C master (handled by the central OBC of the satellite). The tile can be programmed and debugged with the JTAGPINS() and is connected to an external antenna through the ANTENNA() connector. The BACKDOOR() is used to provide the parallel data received from the ground station to a proper system which needs a backdoor interface, where the backdoor connections are shown in figure 5.15. Finally there is an ANTENNA_CONNECTOR() which is used to control the antenna deployment and its telemetry control.

The sub-modules are instantiated in UML as class' objects, and are the Tile Processor (OBCRF), power supply unit (OBCSupply), the transceiver and the sensor unit. Here each of them is reported.

5.2.1 Schematics

In figure 5.8 is shown the schematic of the OBRF top-level Bk1B31A2M that will be placed in tile, comprehensive of connectors. At this level are visible 4 connectors, 3 of them are a reusable block. The connector on the left is the referred to JTAG() pins in the class in figure 5.1, of type Molex J8 PicoBlade, and it is connected to a MODULE_JTAG bus, that will be used to program and debug the microcontroller using the OBRF power system as a power supply for the debugger's signals. The BACKDOOR() in the class diagram in figure 5.1, is implemented with a type Molex J15 PicoBlade (figure 5.4), where the JTAG one is of same type but with 8 pins. On the right of the schematic, there is a 9 pin connector, the Omnetics A29100-009 (figure 5.3), with 2 redundant I2C buses used to control the antenna deployment, and it is related to ANTENNA_CONNECTOR() in class diagram. The antenna is fed from the coaxial cable coming from the connector SSMCX female type shown in figure 5.5 (in UML named ANTENNA() connector, figure 5.1), both for transmitting and receiving, in half-duplex. All of these blocks, except for the Omnetics connector J3, are packed in a reusable block since can be used for other modules. The connection with the OBC it is of type FFC/FPC Molex connector to save space on PCB (figure 5.6), and in class diagram is named MODULE_OBC().

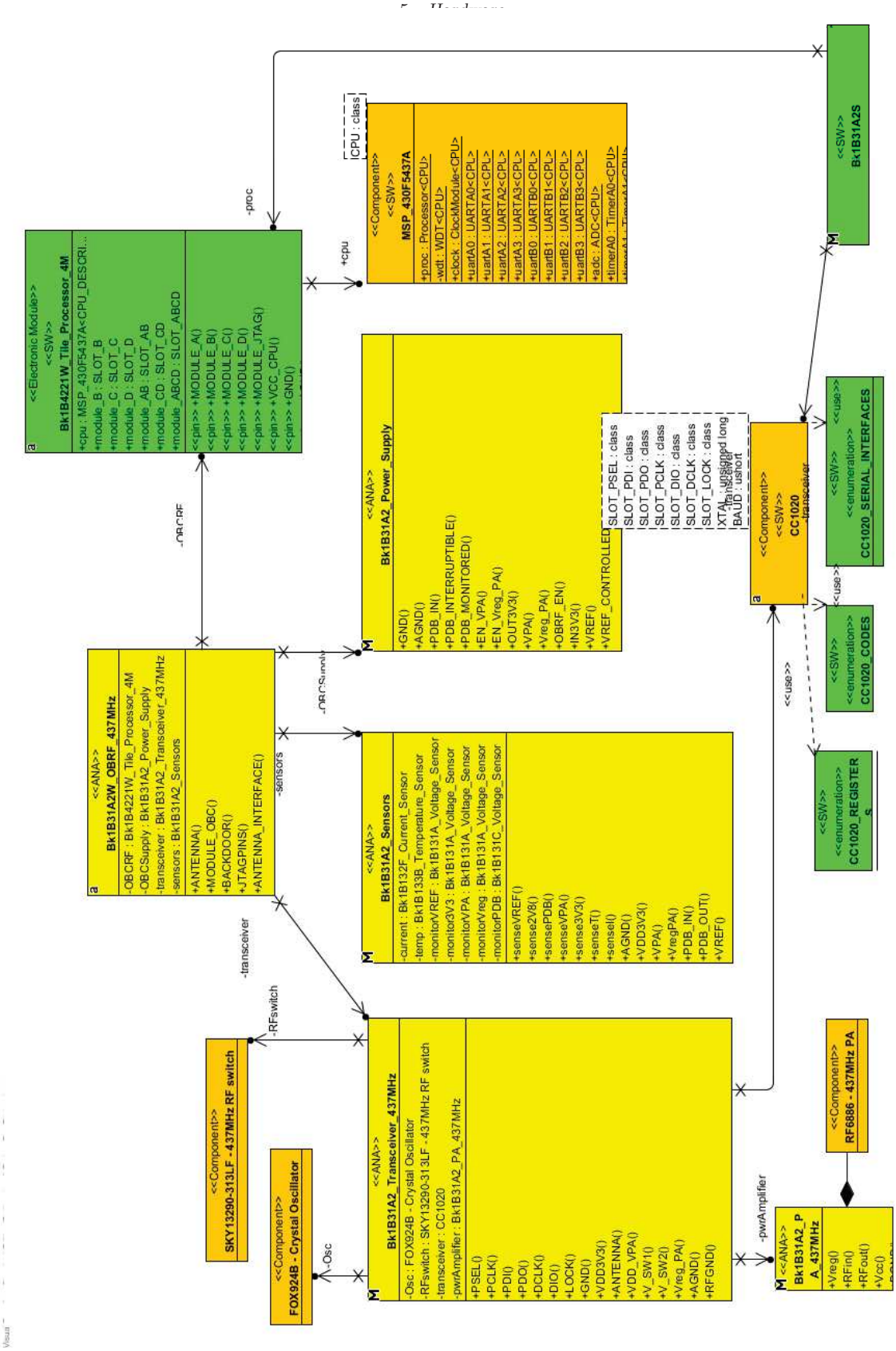

Figure 5.1. Class diagram of the OBRF hardware

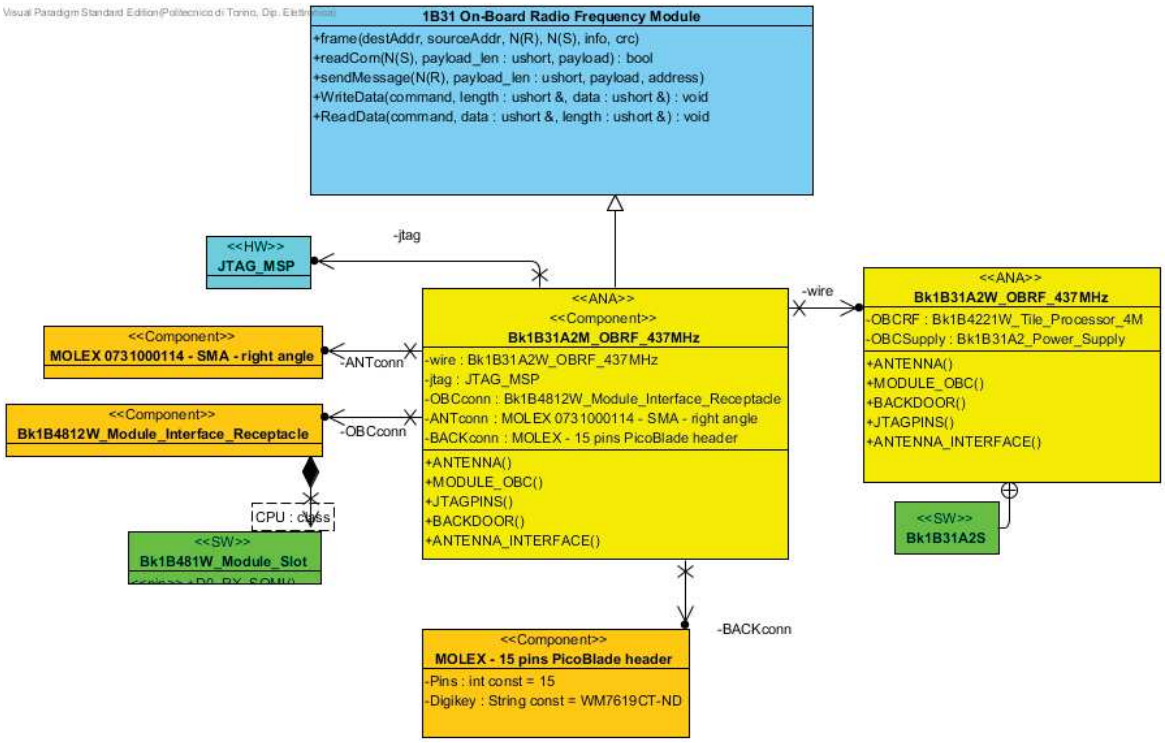

Figure 5.2. Class diagram of the top-level OBRF system

Figure 5.3. A female Omnetics connector

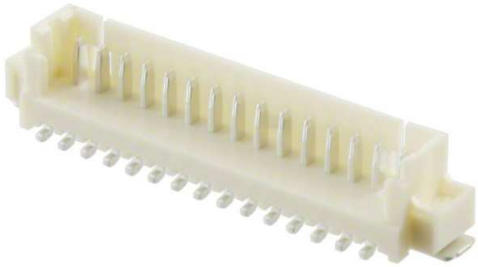

Figure 5.4. Molex PicoBlade, male

Figure 5.5. Molex SSMCX antenna connector, female, straight mounting, SMD

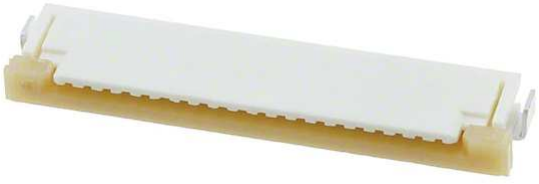

Figure 5.6. Flat OBC Molex connector

The MODULE_OBC() mapping with the connector is shown in table 5.7. This provides the connection with the bus named in the same way, Module Obc, which is used to wire the OBC's satellite bus with the OBRF, using the physical slot.

Signal	Pin	Symbol	I/O	Description
Power Distribution Bus	19,20	PDB	O	Power Distribution bus to supply 12-18V unregulated power supply. This power bus is shared to all AraMiS avionics
+5V	16	5V	O	Positive 5V ±5% power supply
+3.3V	6	3V3	O	Positive 3.3V ±5% power supply
Reference Voltage	8	REF	O	3V±1% reference voltage
Digital I/O	1-5,7,9-12	D0-D9	I/O	Digital Input/Output. Any of these pins can be used as general purpose digital I/O if not configured for other purposes.
Analog I	10,12	A0-A1	I	Analog Input
UART/IrDA Receive	11	RX	I	A standard UART serial line receive, used in RS232 mode and IrDA mode. This is connected to the RX pin of MSP430 UART.
UART/IrDA Transmit	9	TX	O	A standard UART serial line receive, used in RS232 and IrDA mode. .This is connected to the RX pin of MSP430 UART.
SPI		SOMI	I	Slave Out Master In.Used in SPI Bus
		SIMO	O	Slave In Master Out.Used in SPI Bus
		CLK	O	SPI Clock
I²C		SCL	O	Serial Clock.
		SDA	I/O	Serial Data.
Identification	4	ID	I/O	To be connected with 1 wire memory for module identification.
External Signals	13,15	EXT1-EXT2	I/O	External general purpose connectors for routing purposes
Pulse Width Modulation	1,2	PWM,PWM2	O	Pulse Width Modulation
Ground	17,18	GND	-	Common Ground Pin
Analog Ground	14	AGND	-	Analog Ground
Enable	2	EN		Each connected module is enabled by activating this signal.

Figure 5.7. Signals contained in the Bk1B4811_Module_ Interface_ Plug_ V2

In figures 5.9 and 5.10 are shown the complete top level schematic of the Bk1B31A2W only implementation of the OBRF 1B31A, mentioned before in figure 5.1 and being without connectors, because it is not the top level MODULE, will be interfaced with the external world with hierarchical connectors. In the following section will be described each of these blocks, its schematic and its utilization.

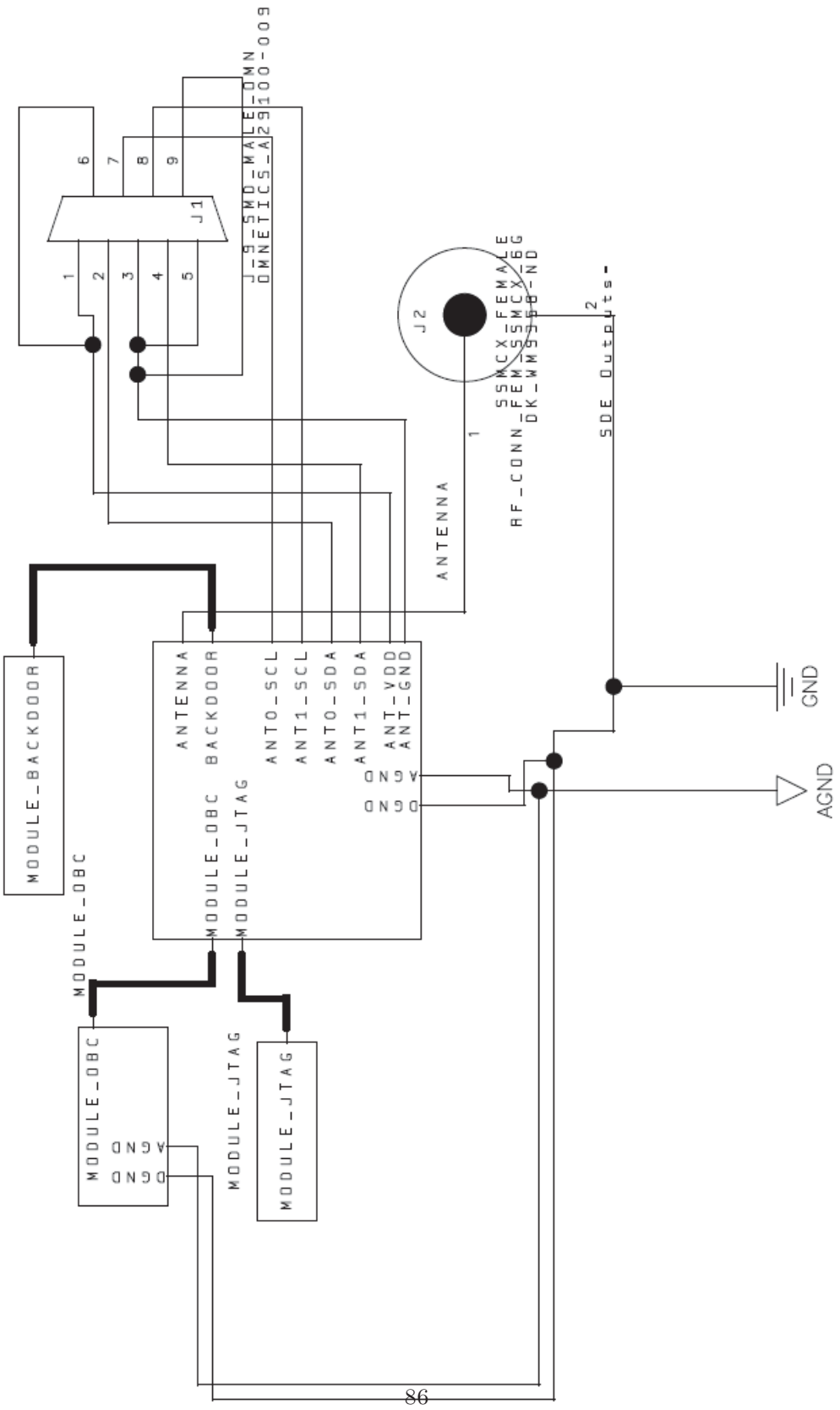

Figure 5.8. Top-level schematic of the OBRF Module

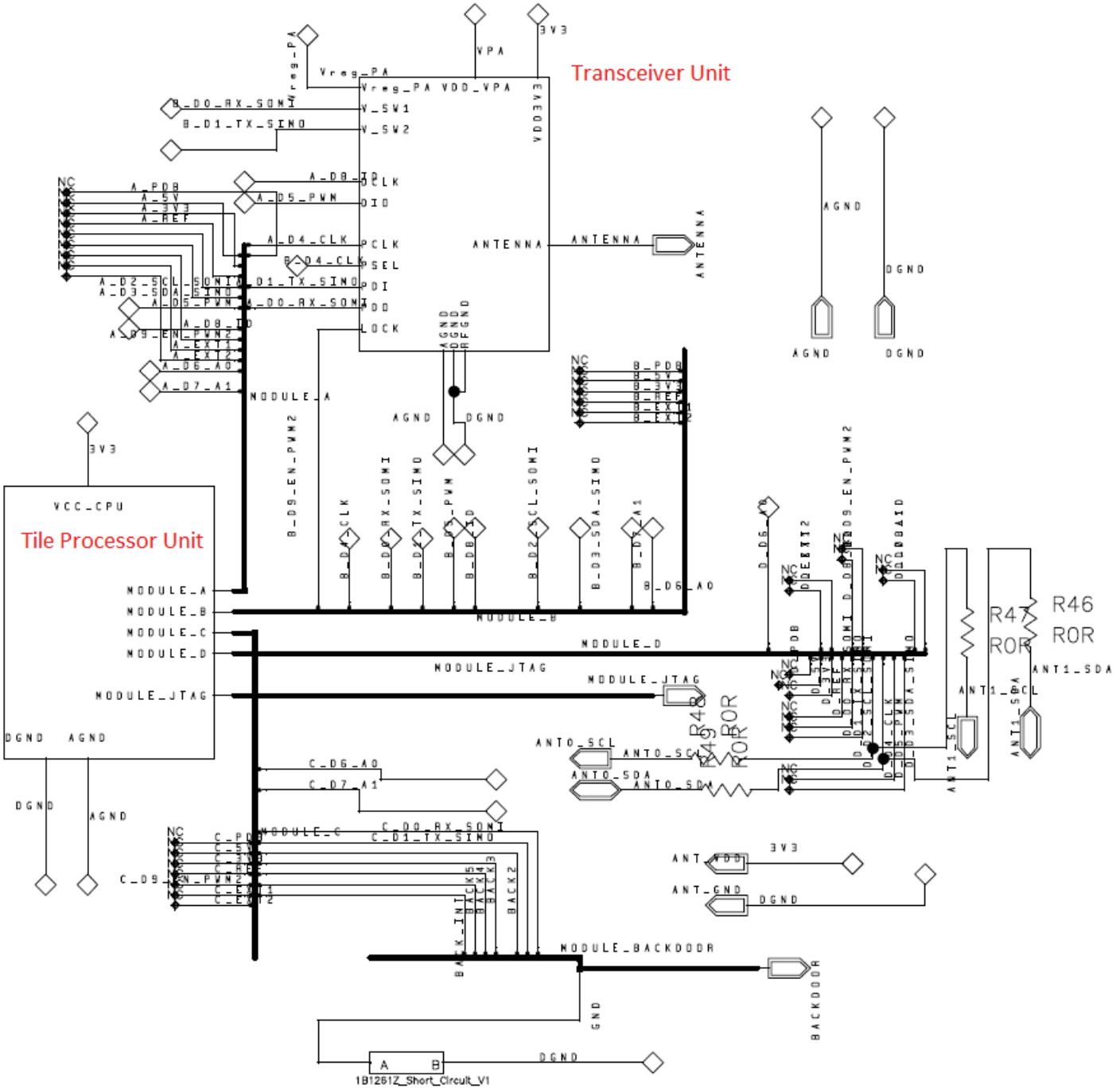

Figure 5.9. Wire level implementation of the OBRF, part 1/2

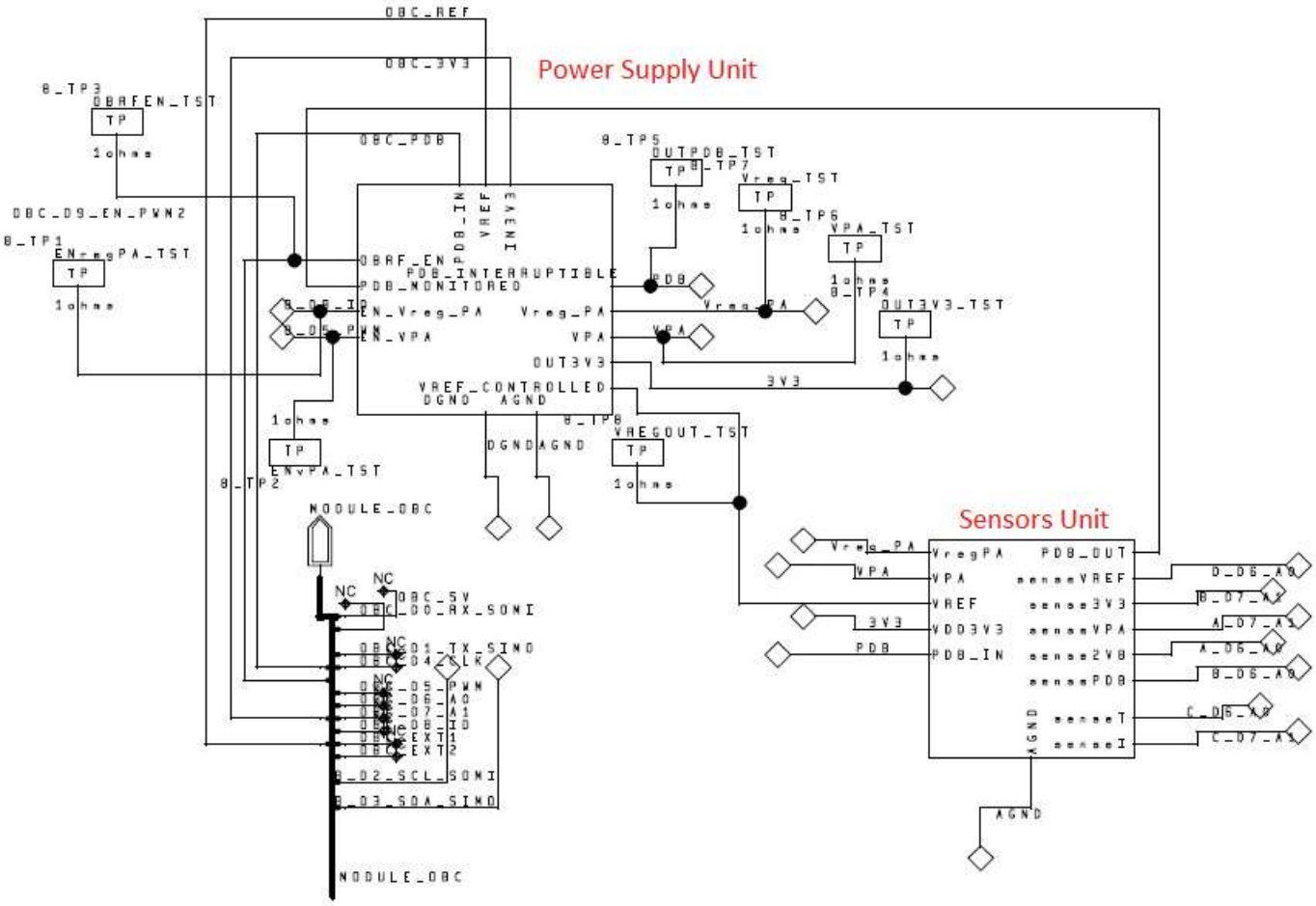

Figure 5.10. Wire level implementation of the OBRF, part 2/2

5.3 Processor unit Bk1B4221W_Tile_Processor_4M

This class contains the tile processor. Consists in an MSP430F5437A microcontroller, with its pins connected directly to modules MODULE_A, MODULE_B, MODULE_C, MODULE_D. It uses a system primary clock based on 4MHz quarts. There is also a secondary low power 32768kHz quartz, which can be not mounted if not needed by the application. The complete firmware will be loaded in this microcontroller through the MODULE_JTAG() connector. When programmed and debugged, should be used the power from the tile. Every switch, sensor or any description of a connection with a MODULE_x, or pin D0, D1 of a certain module, will end up in this hardware block, connecting the wire to the MSP430. This class holds the object **cpu** which represents the MCU, opportunely configured with template CPU_DESCRIPTOR. The objects named *module_x : SLOT_X* are instantiations of the proper software defines used to drive the pins in the module with the same name. See figure 5.11.

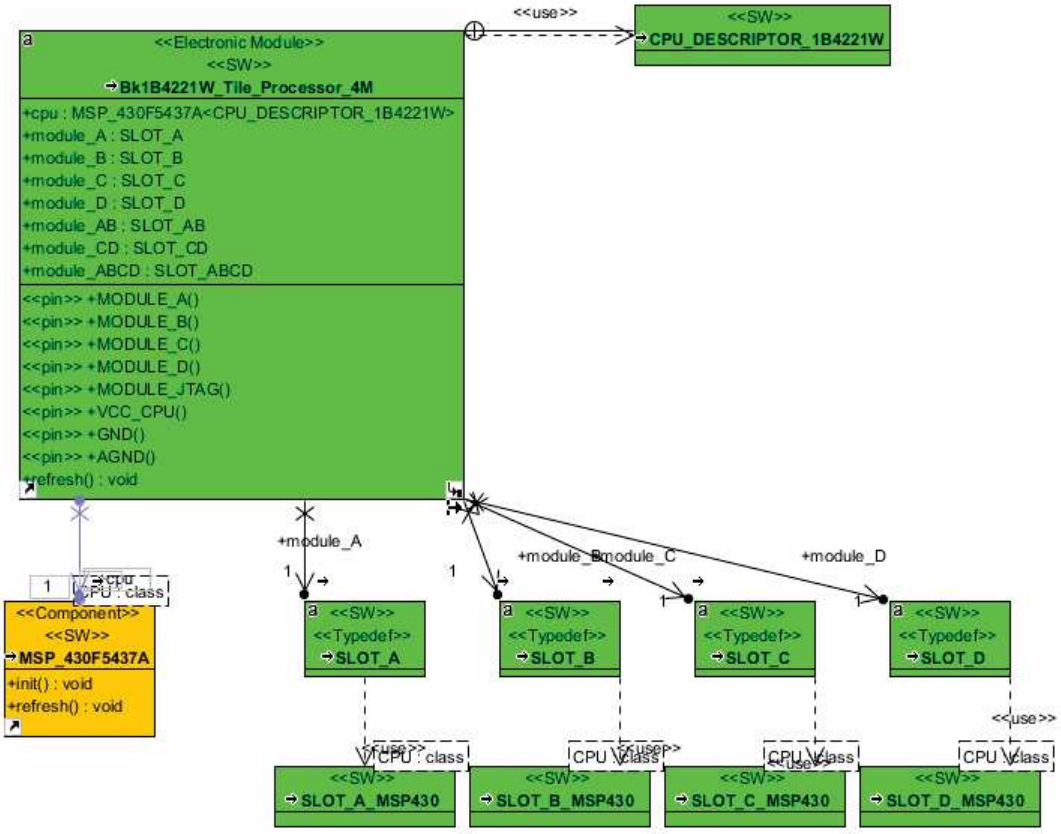

Figure 5.11. Class diagram of the tile processor

5.3.1 Schematics

The schematic in figure 5.12 contains the hardware object, instantiated as **cpu** in figure 5.11, with the required module organization and components for the MCU. Are just present few decoupling capacitors and the jtag bus take the supply from the 3V3 line. The connections with the external hardware are grouped in logical connectors on the left, called from MODULE_A to MODULE_D. The pin usage described in all sequence diagrams and all the buses in the schematic (except for the JTAG), follows the table 6.1 in chapter 6, which organize the logical module connections.

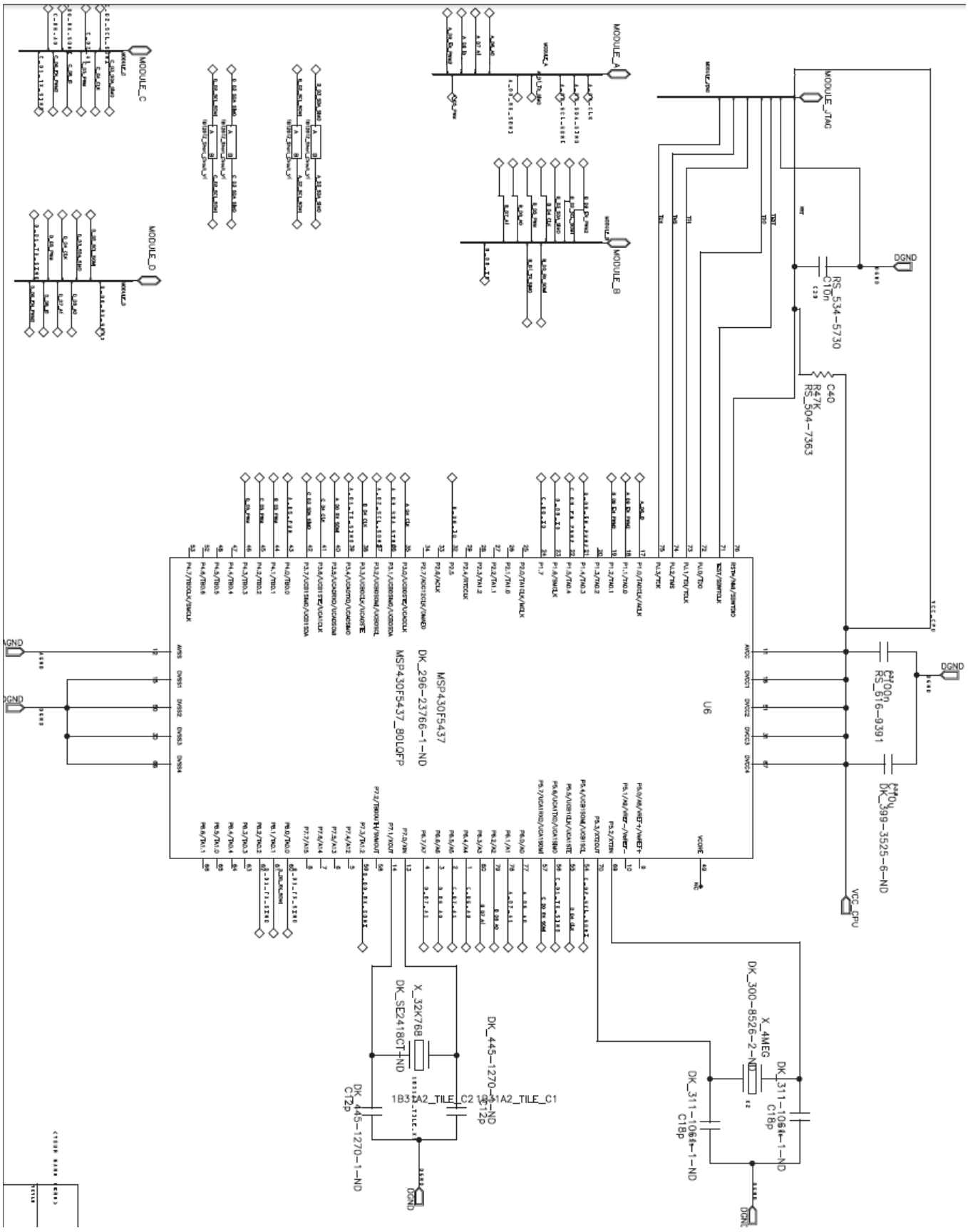

Figure 5.12. Schematic of the tile processor

5.4 Power supply unit Bk1B31A2_Power_Supply

This class, used as a reusable block in Mentor Graphics, represents the power supply unit of the Bk1B31A2W_OBRF_437MHz class and provides the power to the tile, opportunely controlled. The connections of grounds are always not interruptible, while the power supply of the Bk1B31A2W_OBRF_437MHz is composed of more types of voltages. It is represented in UML as in figure 5.13. The tile takes directly the regulated 3.3V and 3V; the power bus, PDB, is less accurate and provides high power and a voltage from 17V to 20V. Moreover, with respect to the previous revision, it is removed the anti latch-up module. This will introduce more space on the board and to countermeasure this change, are modified the supply connections. Now the OBC can directly identify a latch-up on all the power rails, since they are directly connected to the OBC plug, and all of them can be interrupted by a single signal, insulating electrically as much time as required by the module from the rest of the satellite. From this assumption is designed the power supply unit.

The output voltages provided by the unit are:

- PDB_INTERRUPTIBLE(), Imax = 1.3A: Provides the interruptible PDB voltage used to fed the Bk1B31A2_ Sensors' PDB_IN() pin.
- OUT3V3, Imax = 500mA: provides the interruptible 3V3 voltage from the OBC. The OBRF_EN() pin, which is taken directly from MODULE_OBC() connection, interrupt this voltage. This voltage will supply the transceiver, the tile processor and any other 3V3 system.
- VPA(), Imax = 5A: voltage designed to supplt the Power Amplifier. It is controlled by EN_VPA() pin. This regulator is powered from the interruptible PDB _MONITORED(). This VPA() pin is connected to Bk1B31A2_Transceiver_437MHz VDD_ VPA() pin.
- Vreg_PA(), Imax = 100mA: voltage generated from input PDB_IN() and controlled by EN_Vreg_PA(). This Vreg_VPA() pin is connected to Bk1B31A2_Transceiver _437 MHz Vreg_PA() pin.
- VREF_CONTROLLED(): provides the reference voltage coming from the OBC, used for sensors, and it is interruptible from the OBC through the OBRF_EN() pin.

All of these aforementioned voltages can be disabled through the MODULE_OBC() by means of an OBRF enable pin connected to the OBRF_EN() of the unit, even the ones that can be controlled directly by the OBRF tile, therefore the priority is given to enable signal of the OBC. The input voltages of the unit are:

- PDB_IN(), max 15W: Uninterrupted, must be connected to the PDB voltage from the MODULE_OBC().

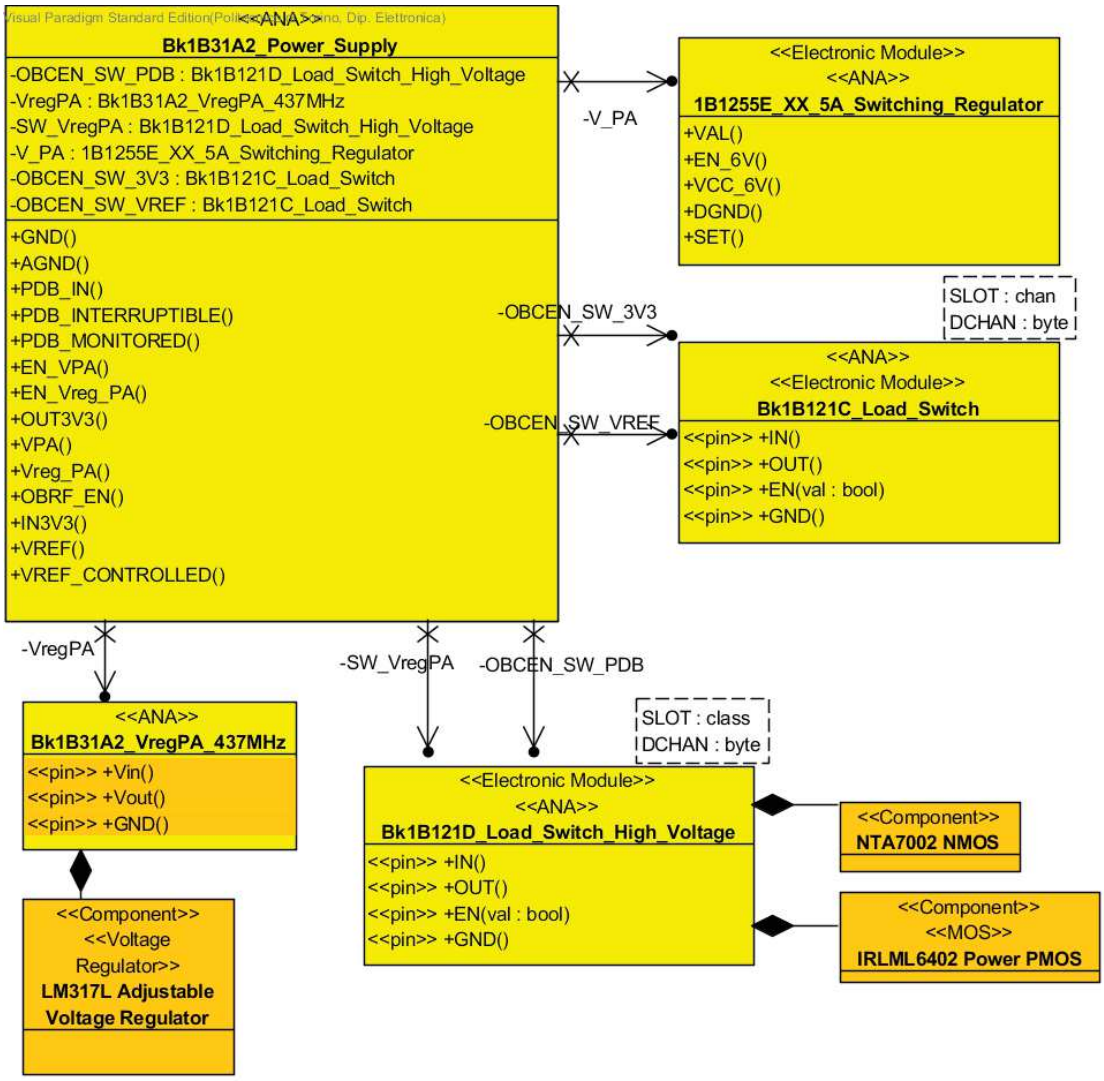

Figure 5.13. Class diagram of the power supply

- PDB_MONITORED(): takes back the PDB voltage from Bk1B31A2_Sensors' PDB_OUT(), which is now monitored. This voltage should be used to connect every on-board system which needs a PDB voltage.
- IN3V3(), max 1W: Uninterrupted voltage taken directly from the MODULE_OBC().
- VREF(): Uninterrupted, reference voltage from the OBC.

The control signals of the unit and voltages are:

- EN_VPA(): Enables the V_PA voltage to VPA() pin.

- EN_Vreg_PA(): Enables the voltage VregPA, to Vreg_PA() pin.
- OBRF_EN(): active high signal, if not used, a pull-up should be present to keep it high. If low, it will disable all the voltage sources of the tile, therefore all the voltages provided by this supply unit. It can be driven directly through the MODULE_OBC().

The objects used in the Bk1B31A2 Power Supply class are instantiated from classes which are described in the following sections.

The PDB_INTERRUPTIBLE() is designed to be provided only to a shunt current sensor, which returns back the monitored voltage to the PDB_MONITORED() which will be used by the internal components of this Bk1B31A2_Power_Supply unit. If an external components to the unit requires the PDB voltage, the PDB_MONITORED() should be used as a voltage source. The connections of this class are also described in figure 5.14 in which the arrows are indicating the output and the input of the power supply unit. The interaction with the rest of the system is described in sequence diagrams in figure 5.15, in figure 6.21 and 6.25.

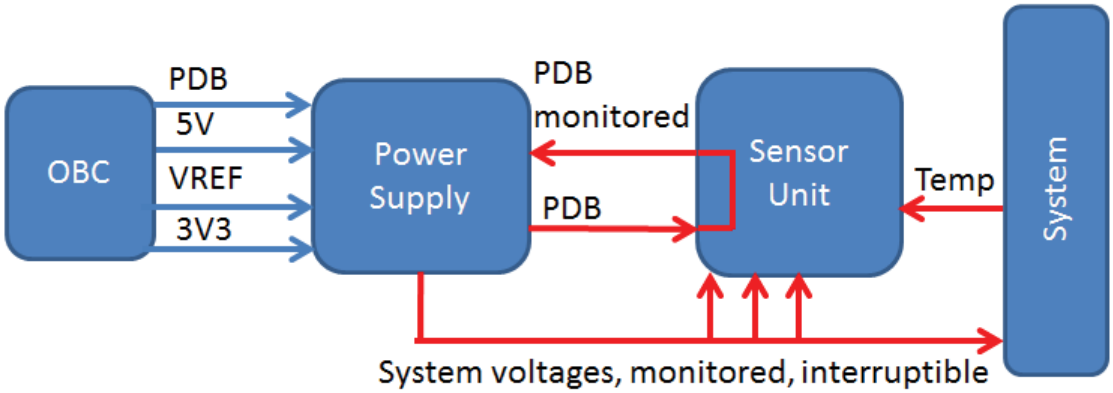

Figure 5.14. Interactions between the sensors and poswer supply unit

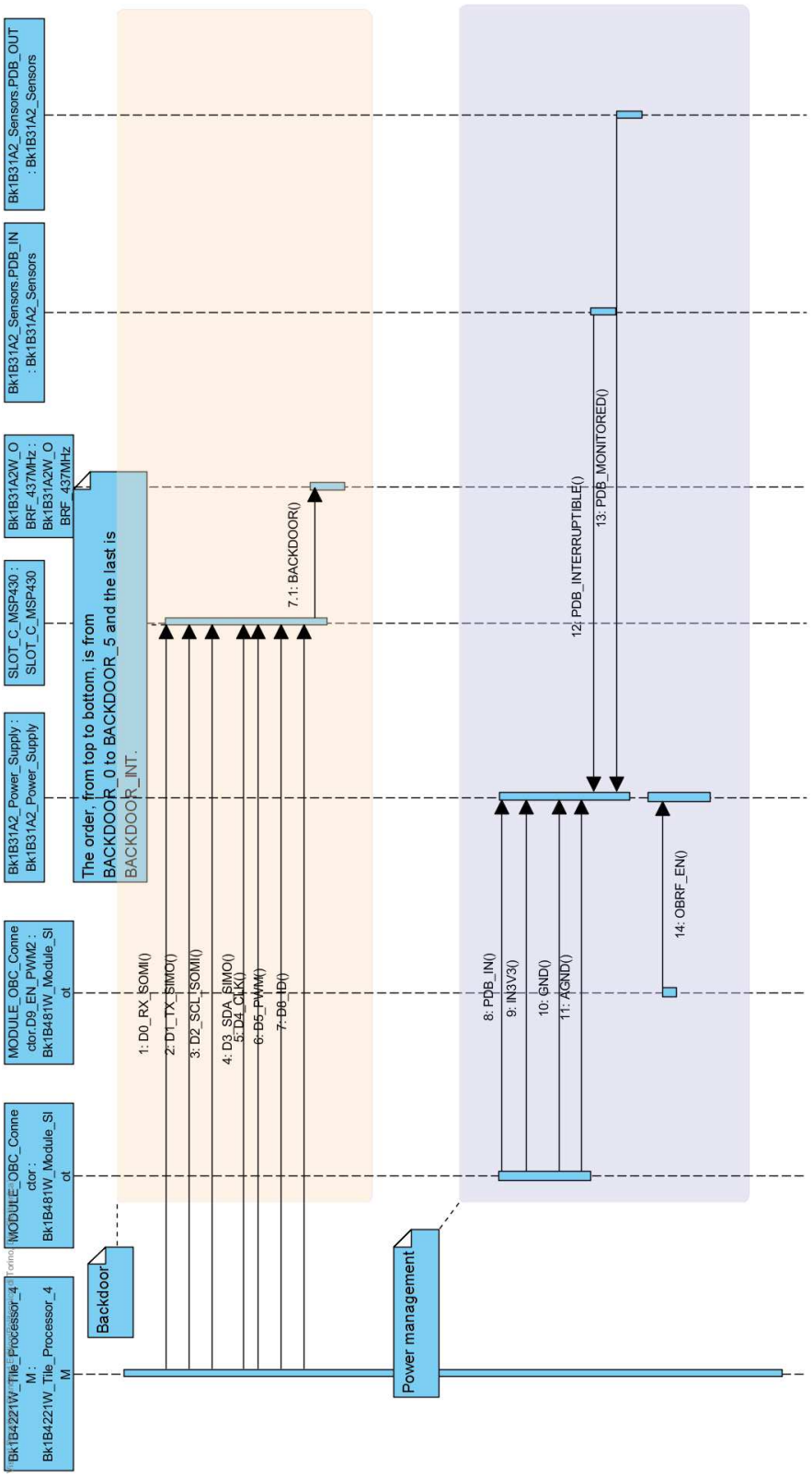

Figure 5.15. Sequence diagram of backdoor and power supply connections

5.4.1 Schematic

In figure 5.16 is shown the top-level of the power supply unit and its relative control signals. Each of its block is described in the following sections.

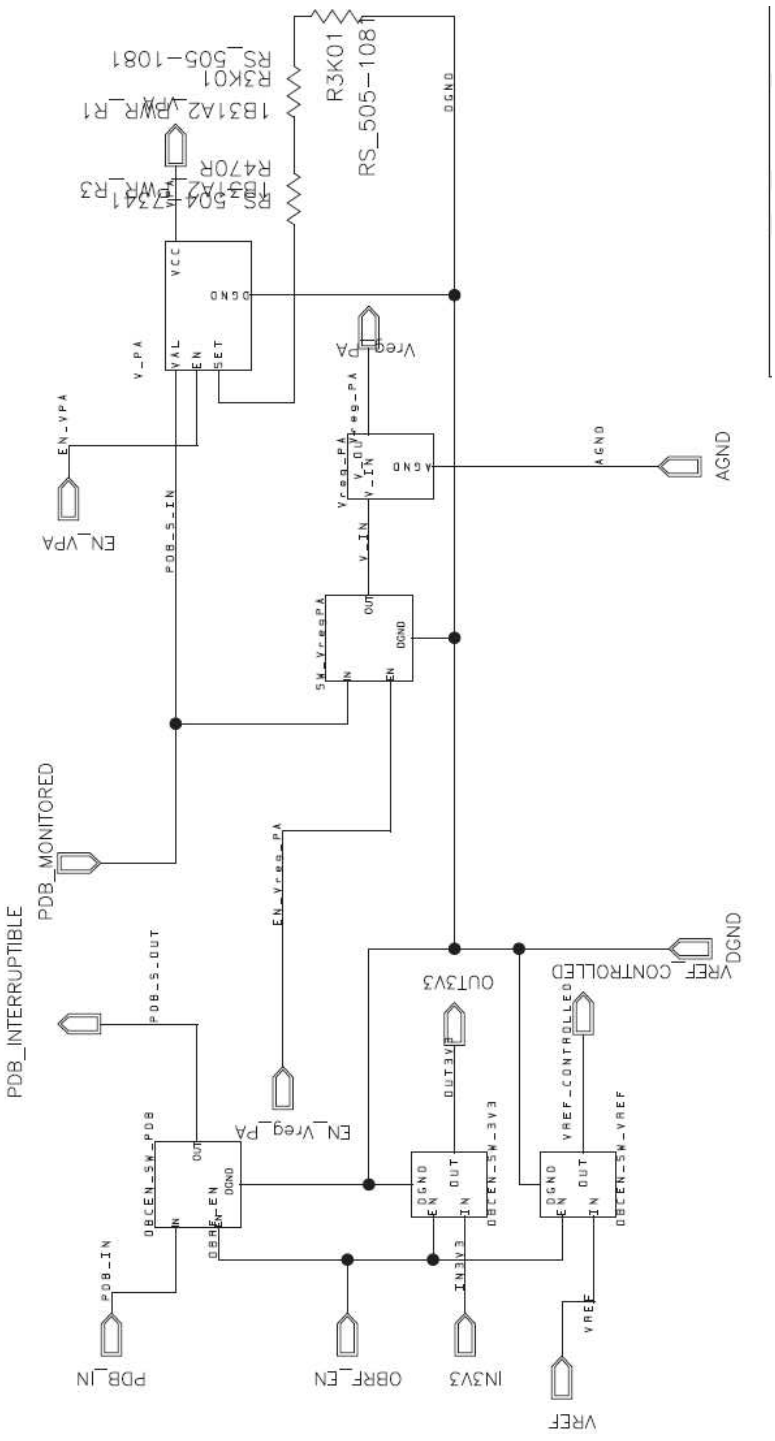

Figure 5.16. Top-level schematic of the power supply

5.4.2 Sub-schematic V_PA block

In figure 5.18 is shown a DC to DC Buck switching regulator that converts VAL() Voltage to a defined one, according to voltage reference at SET() pin. It is a high power regulator that can drive loads up to 5A. It is designed using TPS5450 buck switching regulator. The voltage reference system of the chip produces a precision reference signal by scaling the output of a temperature stable band-gap circuit. The band-gap and scaling circuits are trimmed during production testing to an output of 1.221 V at room temperature.

The output voltage of the TPS5450 is set by a resistor divider (R1 and R2) from the output to the VSENSE pin. The TPS5450 Assuming starting value of R1 = 10kΩ, R2 is given by:

$$R2 = \frac{R1 \cdot 1.221V}{V_{out} - 1.221V} \tag{5.1}$$

To form the R2 value are used resistors with 1% of tolerance (E96 series), allowing to keep the voltage error inside the 2%. Three resistors in series are used, with two of $3.01k\Omega$ and one of 470Ω visible in figure 5.16 connected to V_PA block, obtaining a nominal 3.1V voltage for the PA.

The output capacitor suggested by the manufacturer was of tantalium type, 220μF at 16V. It is changed to a ceramic one, for dependability reasons in a high vacuum environment. Due to the extremely low requirements of capacitor's ESR, the ceramic capacitor used keeps this value inside the boundaries provided. The ceramic capacitors from temperatures below -10°C start almost halving their capacitance, and the same happens when are kept polarized, here with the regulated DC voltage, a phenomenon that is enhanced when the maximum voltage of capacitor is near to the operating one (see figure 5.17). The actual ceramic implementation with high capacity provides not so high voltages, and more than 330 μF are not available with voltages higher than 3V. Moreover, implementing a lot of parallel components to achieve high capacitance with smaller capacitors at higher voltage, lead to an unacceptable low ESR. Therefore two capacitors are used at 220μF each, rated at 6.3V with quality dielectric X5R.

The minimum ESR advised by Texas Instruments is provided by:

$$Resr = \frac{1}{(2\pi \cdot f_z \cdot C_o)} \tag{5.2}$$

where the C_o is the output capacitor and f_z must be higher than the main poles of the compensating network used by the TPS5450, which are at 24kHz and 54kHz, but also not too far from them. Starting with f_z at 54kHz with capacitance of 300μF provide a minimum ESR boundary that can be reasonably lower than the equivalent of the partitor, but further testing should be performed.

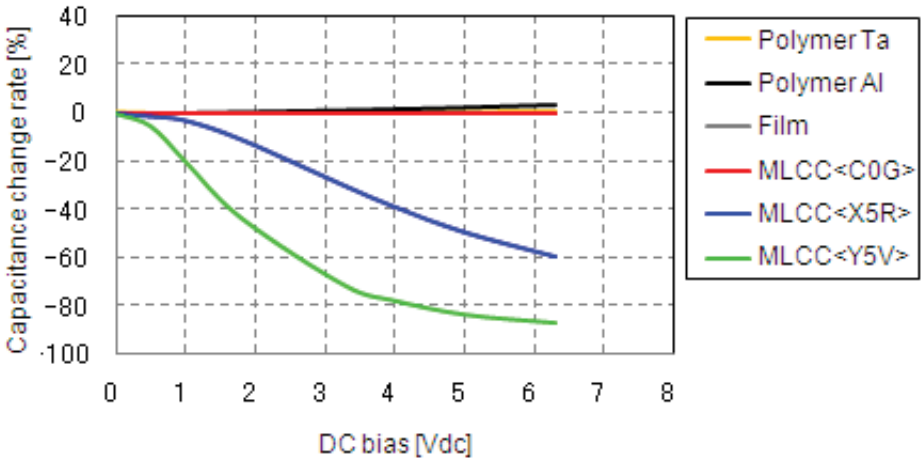

Figure 5.17. Capacitance variation of ceramic capacitors with different dielectric

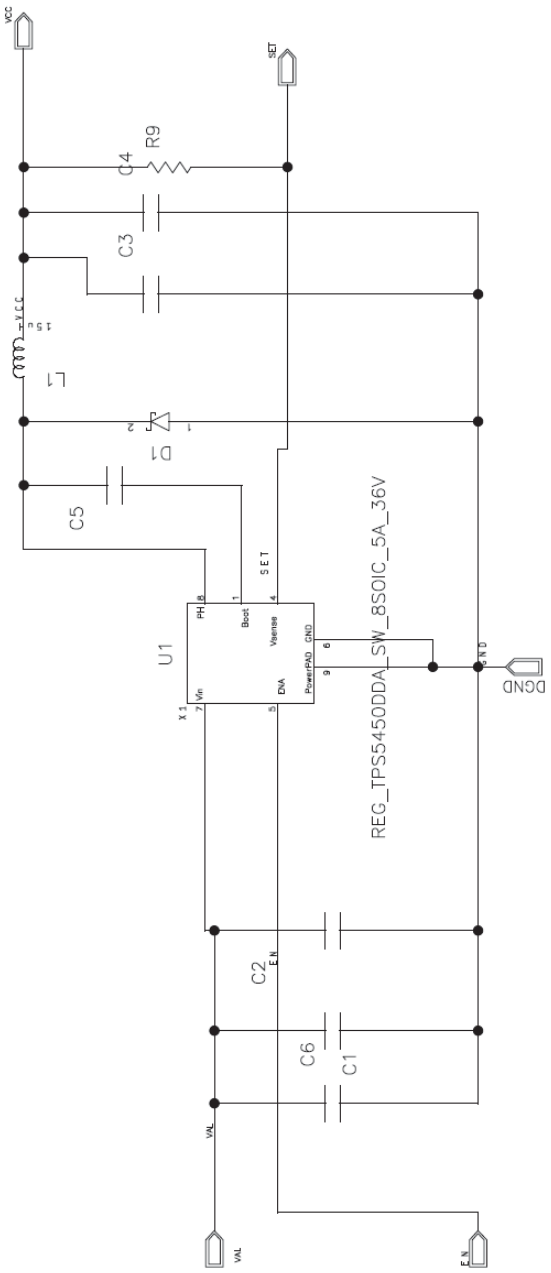

Figure 5.18. Switching regulator for the power amplifier, providing the VPA voltage

5.4.3 Sub-schematic Bk1B121D Load Switch High Voltage

This switch is designed to be compliant with PDB voltage, since the input of the regulator is the PDB and should be interrupted. Its schematic is shown in figure 5.19.

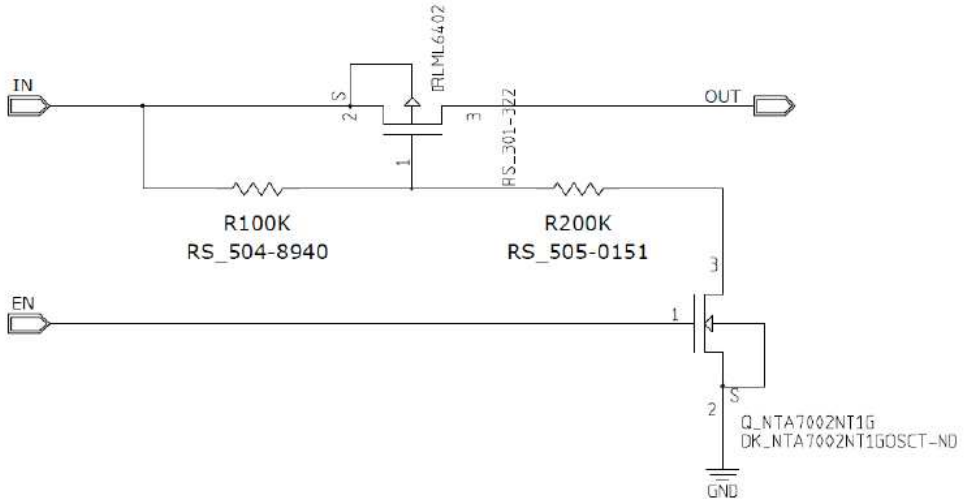

Figure 5.19. A switch type used in the power suply unit, capable of interrupt up to 20V

5.4.4 Sub-schematic Bk1B121D Load Switch

Its schematic is shown in figure 5.20, where the voltage partition is different, so almost all the input voltage is used to drive the IRLML6402 P-MOS, because the input will be always at maximum of 3.3V without the risk of exceeding the maximum allowable Vgs. The N-MOS NTA7002NT1G has a threshold around 1.5V and can be driven by the 3.3V signal from the MCU.

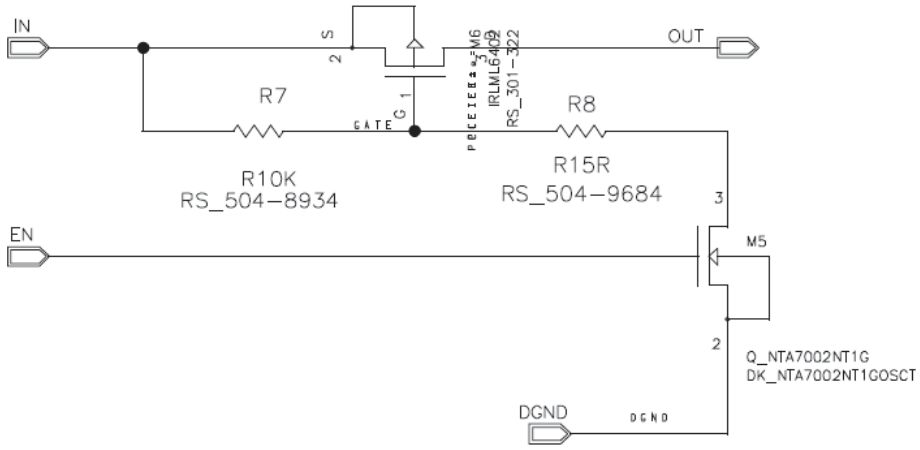

Figure 5.20. A switch type used to interrupt low voltages, up to 5V

5.4.5 Sub-schematic VregPA block

Takes the input from the PDB_MONITORED() pin and can be interrupted by the SW_VregPA. Its output is at pin Vreg_PA(). The input is taken at PDB level, because the other voltages available are not satisfying the minimum drop-out requirement, therefore obtaining the line regulation.

According to the thermal specifications of the voltage regulator LM317L, the spread power is still under constraints even with PDB voltage at its input:

$$(V_{in} - V_{out}) \cdot I_{load} < 200mW < \frac{T_j - T_a}{\theta_{ja}} \tag{5.3}$$

Where the load current $I_{load} \leq 3mA$, voltage difference $(V_{in} - V_{out}) < 17V$, $T_j = 125$ °C, the maximum ambient is considered to be $T_a = 70$ °C, thermal resistance junction to ambient $\theta_{ja} = 165$ °C/W. The output voltage of 2.8V is given by:

$$V_{out} = V_{ref} \left(1 + \frac{R_2}{R_1}\right) \tag{5.4}$$

In figure 5.21 is provided the schematic of the regulator, which uses 4 resistors to provide the required output voltage.

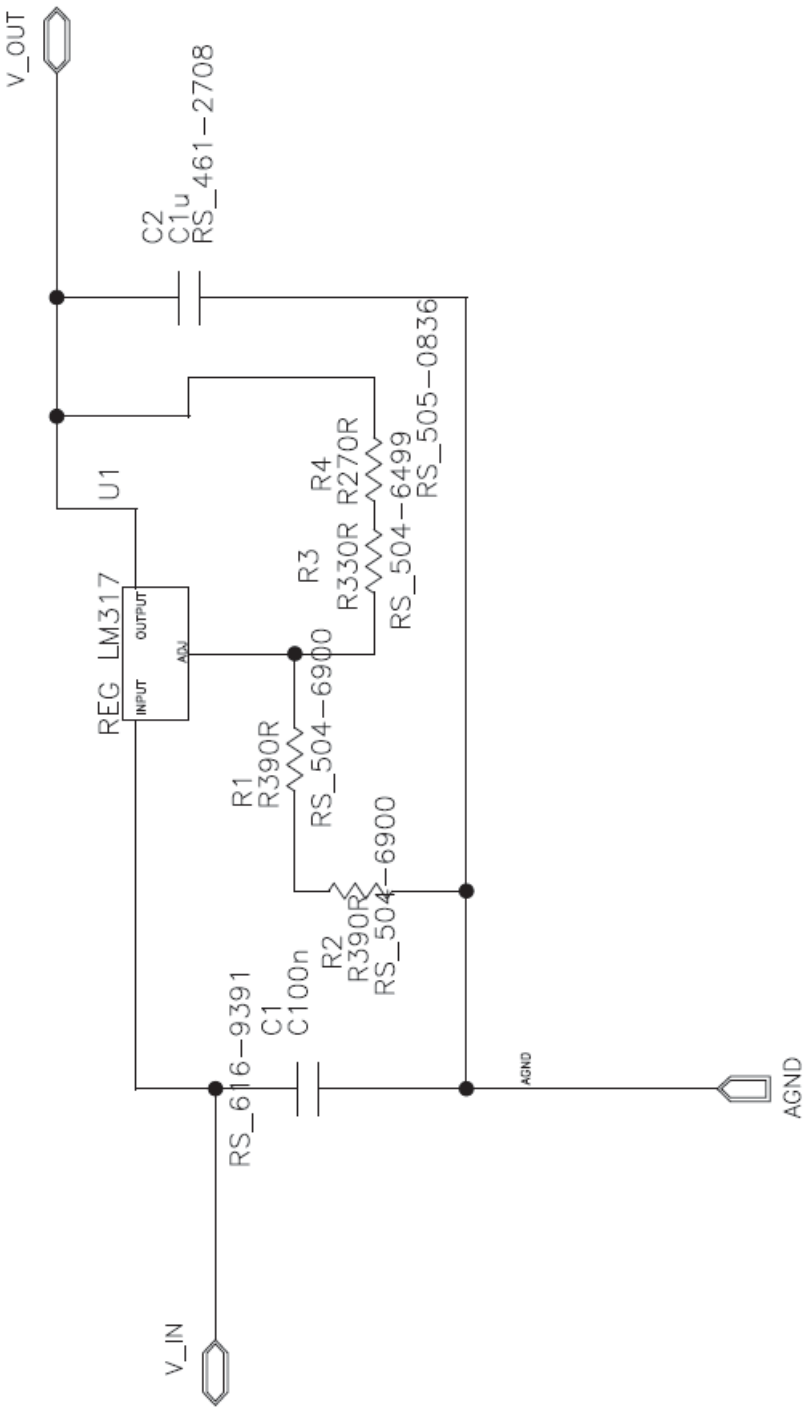

Figure 5.21. Schematic of regulator of reference voltage of power amplifier, the VregPA

5.5 Sensor unit Bk1B31A2_Sensors

This class is used as a reusable block in Mentor Graphics and contains all the system's sensors. Its schematic is provided in figure 5.23. The output range is defined in each sub-block which contains the single sensor. The output analog values are:

- sense2V8(), provides the value of Vreg voltage
- sense3V3(), provides the 3V3 voltage
- senseI(), provides the value of current consumption of the PDB
- sensePDB(), provides the value of PDB voltage
- senseT(), provides the value of the module's temperature
- senseVPA(), provides the value of power amplifier's supply voltage
- senseVREF(), provides the value of reference voltage

The unit need a set of supplies, which are:

- PDB_IN(): takes the interruptible PDB power bus from Bk1B31A2_Power_Supply at PDB_ INTERRUPTIBLE() pin.
- VREF(): receive the interruptible reference voltage from Bk1B31A2_Power_Supply at VREF _CONTROLLED() pin, for reading it.
- VDD3V3(): Take the 3V3 voltage from OUT3V3() of Bk1B31A2_Power_Supply for reading it.
- VPA(): Takes the power amplifier voltage for reading it
- VregPA(): Takes the VregPA voltage for reading it

The objects used in the Bk1B31A2 Sensor Unit class are instantiated from classes which are described in the following sections. Each of these input pins are connected to sources which are interruptible. The PDB_OUT() is the output of a shunt current sensor of this unit, therefore can be used to power up what needs a PDB voltage with a monitored current consumption. The connection of this class is described also in figure 6.31. The connections of this class are also described in figure 5.14 in which the arrows are indicating the output and the input of the power supply unit.

The class diagram of the sensor unit is shown in figure 5.22. The objects instantiated in class Bk1B31A2_Sensors are the sensors' reusable blocks. These objects are containing a float type

SENS_ VOUT value used in software in order to achieve the real read value, so compensating the signal conditioning of the sensor:

$$V_{meas} = \frac{D \cdot S_{max}}{SENS\_VOUT \cdot 2^b} \tag{5.5}$$

Where D is the corresponding digital output from the MCU's ADC, S_{max} is the maximum output sensor voltage (which is the same for all sensors of every kind and magnitudes handled), SENS_VOUT is the aforementioned compensating value and b are the ADC's available bits. The maximum supported analog voltage is 2.5V represented in 12 bits.

Each of these objects refers to an higher level set of classes in UML, which represents the typology of the sensor (voltage, current, temperature...) and are containing the software classes, in green in the class diagram. These software classes are instantiated as objects in chapter 6 in figure 6.3, using the appropriate templates parameters in the *Bk1B31A2S* class. The digital value is stored and brought to OBC when required by the housekeeping functions: the OBC should therefore use the SENS_VOUT and other compensating values in order to read the correct analog value, while the OBRF provides only the raw digital value. Now are provided the descriptions of the various type of sensors used in this unit.

Voltage sensor

The generic voltage sensor mostly converts (through a voltage divider) the input voltage between input pin VIN() and analog ground AGND() to an output voltage between pin VOUT() and analog ground AGND(). Input voltage shall range between 0 and a maximum value which depends on the specific specialization of Bk1B131_Voltage_Sensor (instantiated as object in Bk1B31A2_Sensors), using a defined value INPUT_RANGE, while output voltage is in the range 0 to OUTPUT_RANGE. Output impedance is common for all implementations (namely, OUTPUT_IMPEDANCE). It also contains a first order low pass filter to flattening the output of the sensor. It requires no supply voltage. Output impedance is high. This must be taken into account during sample and hold phase. In next sections are shown the actual quantitative implementation of the sensor.

Current sensor

The current sensor converts positive current flowing from pin I_IN() to pin I_OUT() into an output voltage between pin CS_VOUT() and analog ground AGND(). Input current shall be in a range which depends on the specialization of Bk1B132_Current_Sensor (instantiated as object in Bk1B31A2_Sensors) using a defined value INPUT_RANGE, while output voltage is in the range 0 to OUTPUT_RANGE. Differential input impedance between pins I_IN() and I_OUT() depends

on the actual sensor used. Output impedance is common for all implementations (namely, OUTPUT_IMPEDANCE). It internally takes supply voltage from pin I_IN(), therefore input voltage on this pin shall be in range SUPPLY_VOLTAGE_MIN to SUPPLY_VOLTAGE_MAX appropriately defined. Supply current drawn from pin I_IN() is given by SUPPLY_CURRENT_NOMINAL in sensor class. It also contains a first order low pass filter to flattening the output of the sensor. In next sections are shown the actual quantitative implementation of the sensor.

Output voltage is given by:

$$V_{CS\_VOUT} = I_{I\_IN} \cdot SENS\_CS\_VOUT \tag{5.6}$$

where I_{I_IN} is the current entering from pin I_IN() and exiting I_OUT(), while SENS_CS_VOUT depends on the magnitude required by the sensor.

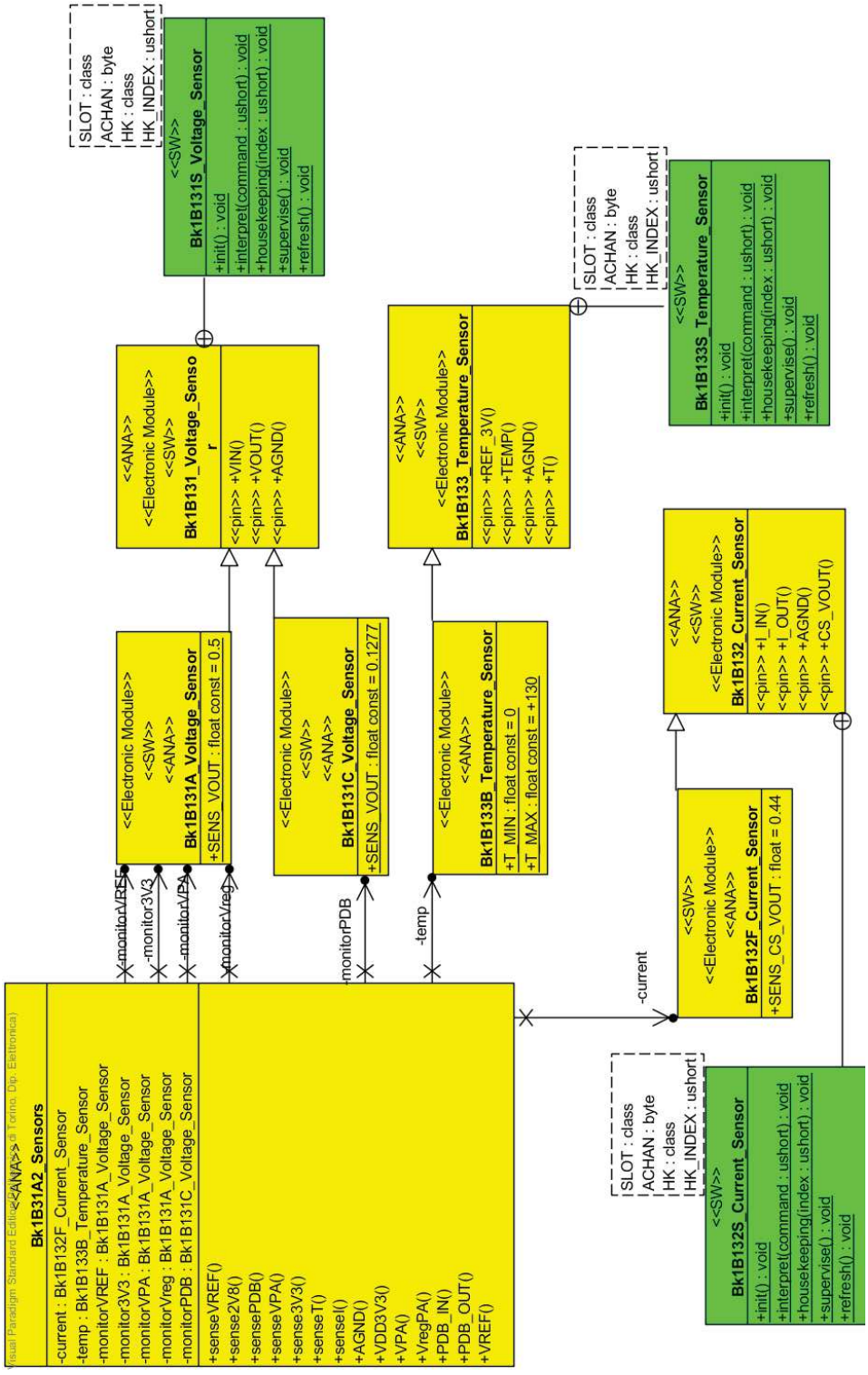

Figure 5.22. Class diagram of the sensor unit

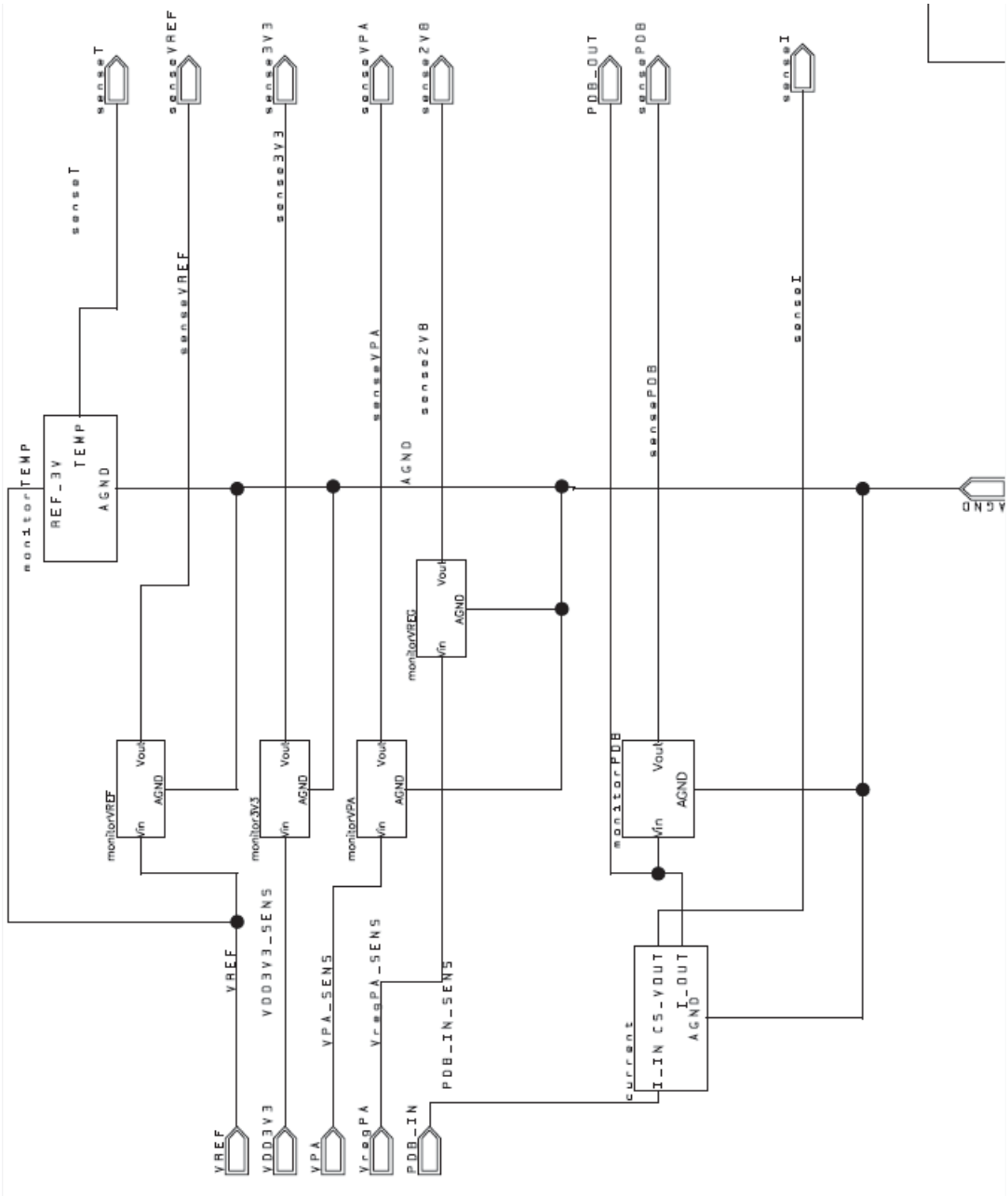

Figure 5.23. Top-level schematic of the sensors unit

Temperature sensor

It is a non-linear temperature sensor for an AraModule. It mostly converts temperature on a transducer (at point T()) to an output voltage between pin TEMP() an analog ground AGND(). Temperature shall be in range T_MIN to T_MAX. The range depends on the specific implementation of Bk1B133_Temperature_Sensor (instantiated as object in Bk1B31A2_Sensors), while output voltage is in the range 0 to OUTPUT_RANGE. It requires a 3V reference voltage between REF_3V() and AGND(). Output voltage is a non linear function of temperature at point T(); this is plotted (for each implementation) in a referenced plot in figure 5.24.

In next sections are shown the actual schematics of the sensors.

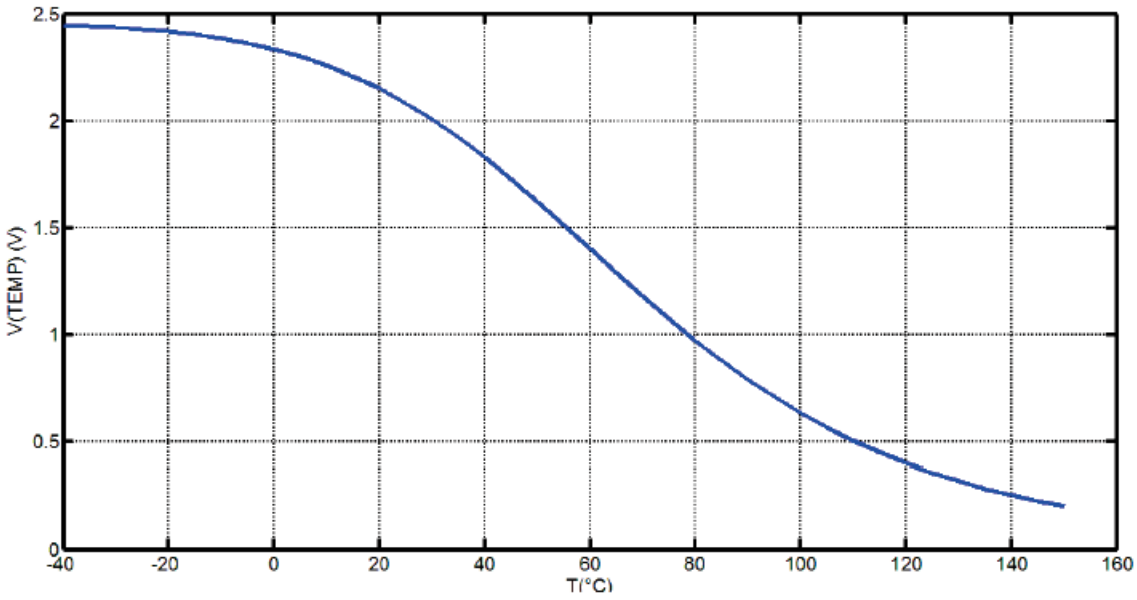

Figure 5.24. Transfer function of the NTC sensor adopted

5.5.1 Sub-schematic Bk1B131A_Voltage_Sensor block

This schematic is istantiated four times because as many voltages in the range of this sensor are need to be monitored. As mentioned before, this is a specific implementation of a Bk1B131_Voltage_Sensor with the INPUT_RANGE=5V. The block provide a maximum voltage of 2.5V, therefore the OBC divides input voltage by a compensating factor of SENS_VOUT = 0.5 (see eq. 5.5) to obtain a maximum range of 5V. The sensor consist of a voltage divider with an high input impedance. In figure 5.25 is provided a schematic of the sensor used and it can measure up to 5V.

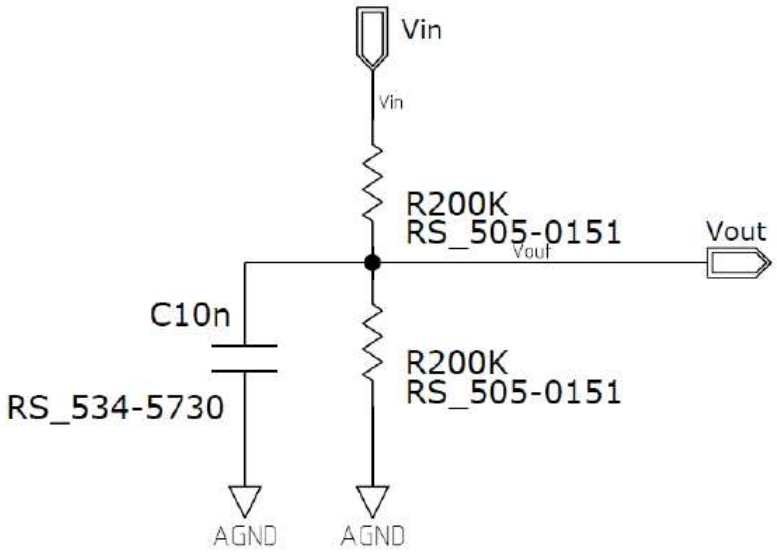

Figure 5.25. 5V range voltage Sensor

5.5.2 Sub-schematic Bk1B131C_Voltage_Sensor block

This is a specific implementation of a Bk1B131_Voltage_Sensor with the INPUT_RANGE=20V. The block provide a maximum voltage of 2.5V, therefore the OBC divides input voltage by a compensating factor of SENS_VOUT = 0.1277 (see eq. 5.5) to obtain a maximum range of 20V.

There is only one instantiation of this class, named **monitorPDB**, measure the PDB_OUT() and provide the analog read to *sensePDB()* pin. In figure 5.26 is provided a schematic of the sensor used and it can measure up to 20V. The conditioning here means varying the resistor partition, in order to keep the output inside the specifications.

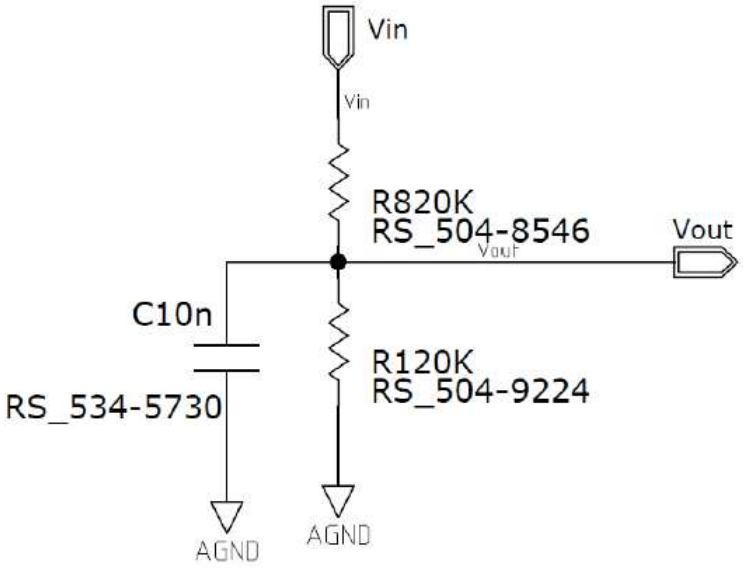

Figure 5.26. 20V range voltage Sensor

5.5.3 Sub-schematic Bk1B132F_Current_Sensor

This is a specific implementation of a Bk1B132_Current_Sensor with the INPUT_RANGE=5.682A, instantiated as a **current**, that will provide maximum analog voltage of 2.5V to maintain the MCU's ADC in its dynamic range. The OBC which reads the raw value should divide the voltage read, which is proportional to input current, by factor SENS_CS_VOUT = 0.44, according to the shunt resistor used, which represents the sensitivity of sensor in V/A. The sensitivity is known given the transconductance, gm, of the TI INA138 device adopted to implement it, of 200 uA/V. Therefore, according to schematic, the output voltage is:

$$VOUT = I_{in} \cdot R2 \cdot gm \cdot R1 \tag{5.7}$$

Figure 5.27 provides the schematic of the current sensor.

5.5.4 Sub-schematic Bk1B133B_Temperature_Sensor

In figure 5.28 is shown the schematic of the temperature sensor. This is a specific implementation of a Bk1B133_Temperature_Sensor with a range from -40 to +130 °C with a thermal constant of 8 s. The circuit acts as a voltage divider with high input impedance, then the output voltage depends on the resistance associated to the NTC, that is a function of temperature. It is placed near the power amplifier, since it is the most critical elements in term of power consumption.

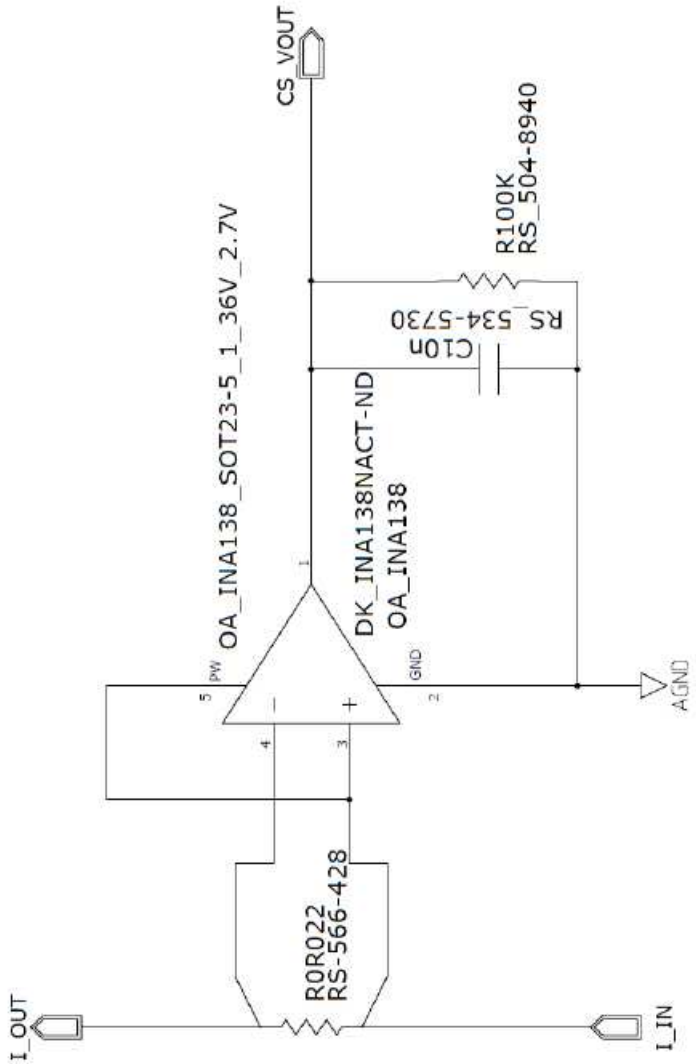

Figure 5.27. Current sensor

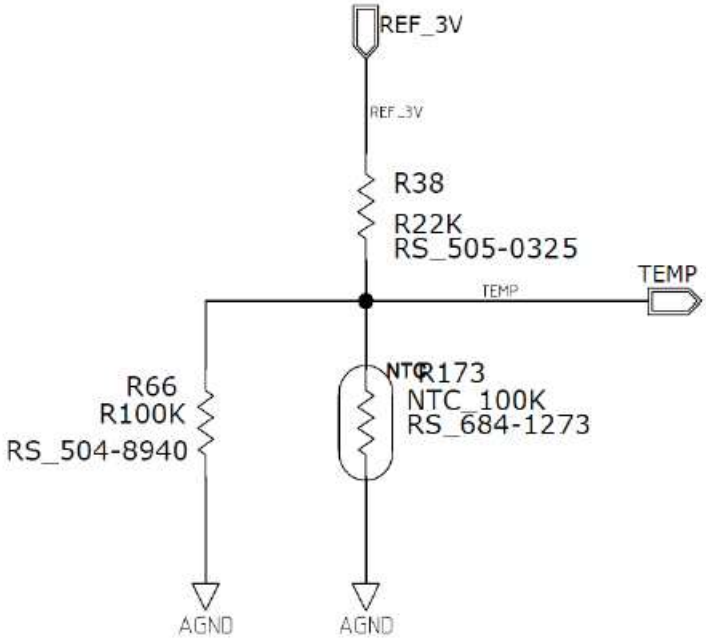

Figure 5.28. Schematic of the temperature sensor

5.6 Transceiver unit Bk1B31A2_Transceiver

This unit implements the transceiver of the Bk1B31A2W_OBRF_437MHz. It contains the digital interface for the data processing, the RF connections for the ANTENNA(), power supplies inputs and control voltages.

The digital programmable interface is SPI compliant, where the Bk1B31A2W_OBRF_437MHz is supposed to be the master:

- PCLK(): It is the clock signal for the SPI-interface.
- PDI(): Data-input pin for transceiver configuration. provides the serial bit-stream of the SPI-interface. This pin is used to write setup information into the transceiver's registers.
- PDO(): Data-output pin for transceiver configuration. The serial bit-stream of the SPI-interface. This pin is used to read setup information from the transceiver's registers.
- PSEL(): It is the slave select signal for the SPI-interface.

The digital pins for the digitalized synchronous RF data are:

- DCLK(): the interface clock, always provided by the transceiver
- DIO(): data pin of the stream, bidirectional
- LOCK(): PLL lock pin by default, its usage is programmable

TX/RX switch control voltages, used to control the RF switch, according to sequence diagram in figure 6.21 and 6.25, named V_SW1() and V_SW2().

Transceiver supplies are:

- VDD_VPA(): the power supply of the power amplifier
- Vreg_PA(): the regulation voltage of the power amplifier
- VDD3V3(): the power supply of the CC1020 chip.

Its connections are described also in figure 6.4.

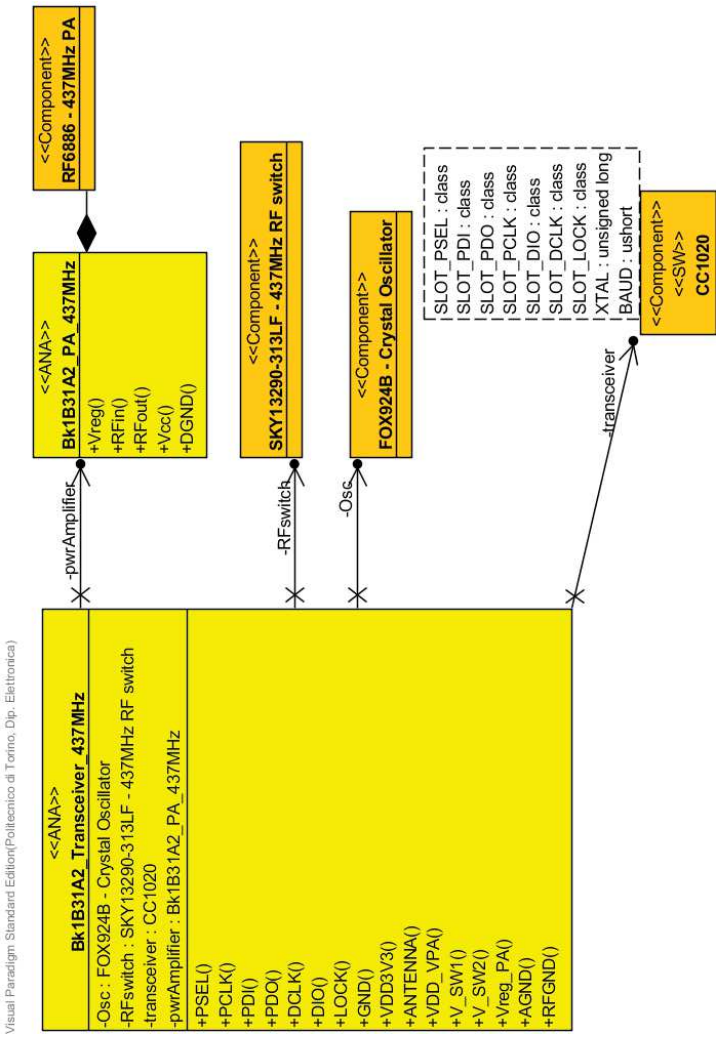

Figure 5.29. Class diagram of the transceiver unit

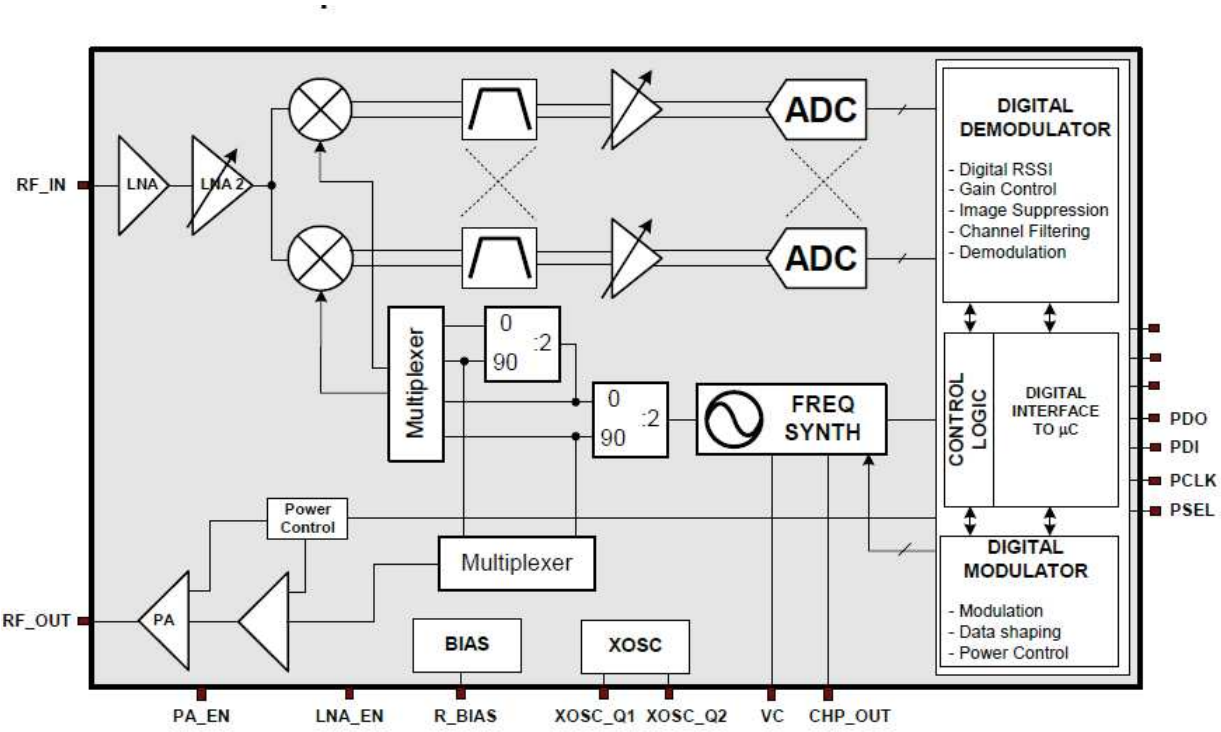

Figure 5.30. Internal block diagram of the transceiver CC1020

This transceiver unit uses a TI CC1020. Its functional block is shows in figure 5.30. In reception, the signal is amplified by two LNAs, where the LNA2 has a variable gain, compensating the power level variations of the input signal. Then it is down-converted at I and Q at IF frequency and after the filtering, is digitized by the ADCs, using a frequency synthesizer shared by the TX circuitry. Before the ADCs there is a variable gain amplfier, in order to stay in the full dynamic range of the ADC, reducing the quantization noise. Automatic gain control, fine channel filtering, demodulation and bit synchronization is performed digitally through the configuration registers. The demodulated digital signal is brought to the DIO pin, and updated at every rising edge of DCLK.

During the transmission, is transmitted the signal present at DIO pin, sampled at every falling edge of DCLK. It is used the frequency synthesizer shared with the RX circuitry. The stream at DIO is shift keyed (FSK) directly on the PA, with variable gain. Optionally can be used a gaussian filter for the shift keying, obtaining a GFSK modulation.

The chip contains two set of configurations for the frequency synthesizer, and registers which are configuring those parts are identified with letters A or B. When properly configured the A and B parameters, is possible to switch very fast between one configuration to another. Could be used for double configuration for either TX and RX, or used to switch rapidly between RX and TX, as in the OBRF.

The frequency synthesizer includes a completely on-chip LC VCO and a 90°phase splitter for generating the LO_I and LO_Q signals to the down-conversioncmixers in receive mode. The VCO operates in the frequency range 1.608-1.880 GHz. The CHP_OUT pin is the charge pump output and VC is the control node of the on-chip VCO. The external loop filter is placed between these pins. It used an oscillator which fed the CMOS output level to the XOSC_Q1 pin (see figure 5.30). A lock signal is available from the PLL, and can be read on PLL pin or digitally from the PLL register. Pin of PLL and enable signal for external LNA and PA are kept disabled and can be used also as a general purpose pins.

A note on the schematics of the CC1020: few components are not the same for all the configurations. According to the bandwidth adopted for the signal, the PLL loop filter components must change accordingly. Moreover, according to the defined carrier frequency, the RF matching components and filtering should change, too. This is accomplished by using the TI SmartRF Studio for devising the optimum values. The components adopted are then compatible with settings provided at the end of section 6.3.5.

5.6.1 Top level schematic of transceiver

In this schematic in figure 5.31 is shown the CC1020 used with the advised reference schematic from TI, except for components of PLL loop filter and matching network. The transceiver should be capable of tracking the frequency variations, due to doppler and other influences (both of transceiver and the ground station). For this reason the internal local oscillator (for the IF stage) and the frequency synthesizer are needing a precise clock source, avoiding large channel bandwidth and so avoid to reduce the sensitivity (see section 6.3.5 to see how sensitivity varies with the bandwidth).

The clock signal is external and so it is not used any resonant crystal. It is used a clock generator of 5ppm accuracy, the FOX924B, powered at 3.3V and providing a HCMOS compliant output. The frequency chosen is 14.7456MHz, in order to use the advised frequency and improve the precision of the chip settings. The interface with the antenna is made through an RF switch, insulating the RX and TX networks. It is a solid state switch, therefore consumes low power, it is small and with higher dependability. The timing for the control voltages are shown in figures 6.21 and 6.25. The component used is a SKY13290_313LF pHEMT single pole double throw switch. In transmission the signal pass through a power amplifier (see next paragraph) before entering in the switch, while in reception the sensitivity is high enough to avoid an external LNA.

The proper decoupling capacitors and the antenna matching circuitry are compatible with the reference schematic provided by TI. Therefore the output towards the PA or coming from the switch are designed to have a 50Ω impedance matching.

5.6.2 Sub-schematic of power amplifier block

In transmission the signal must be amplified. For this purpose is used an RF6886 power amplifier. The device is manufactured on an advanced InGaP HBT process and is provided in a 24-pin leadless chip carrier with backside ground. External matching allows for use in standard bands from 100MHz to 1000MHz, for this reason is followed the reference design for the 433-470MHz. The matching circuits are designed to achieve a 50Ω impedance matching, using the advised capacitors due to a low DC leakage. The schematic is shown in figure 5.32. Components of the RF OUT and IN sections were chosen accordingly to what was suggested by RFMD, as long as the decoupling of the supply.

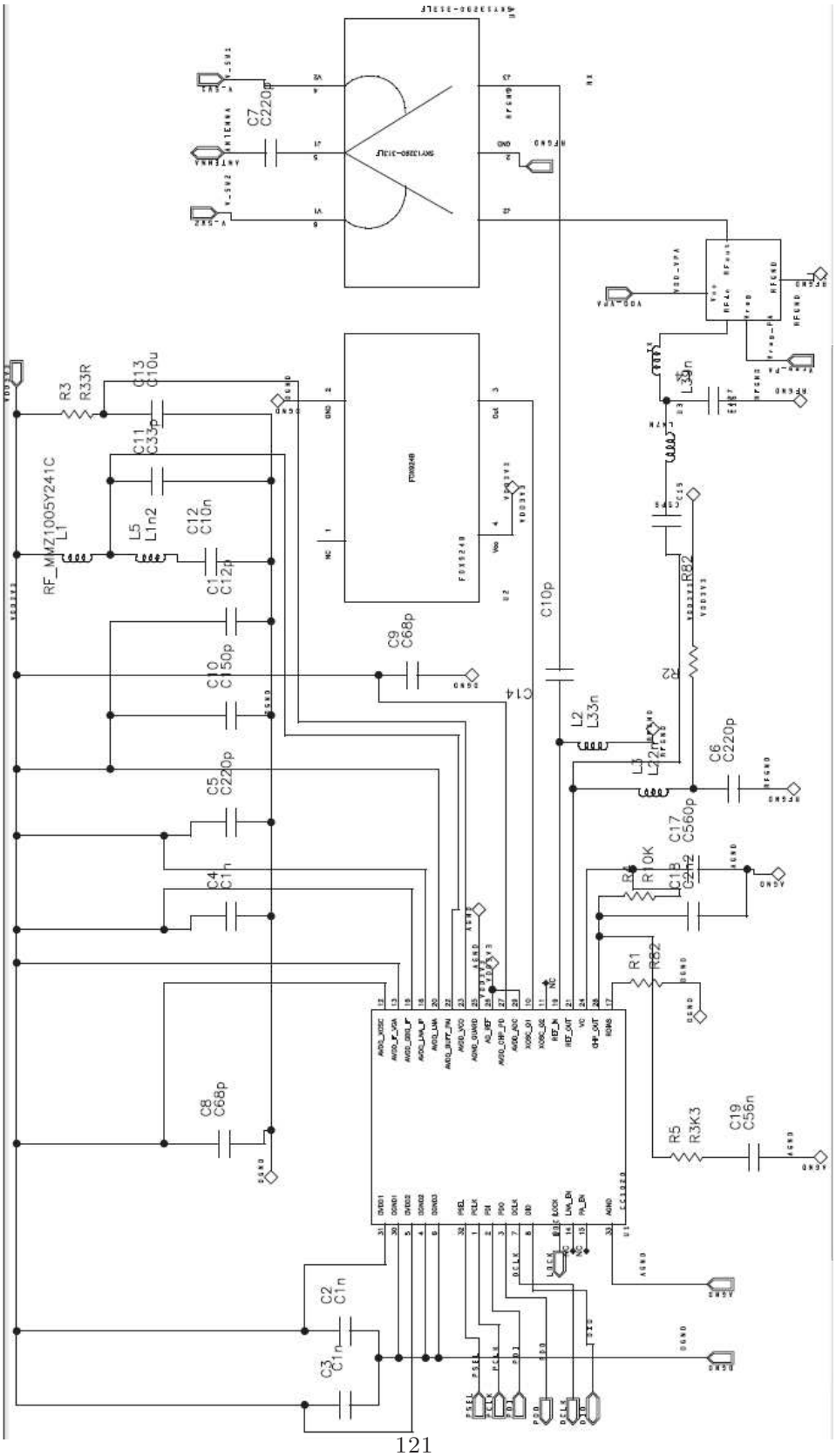

Figure 5.31. Schematic of the transceiver unit

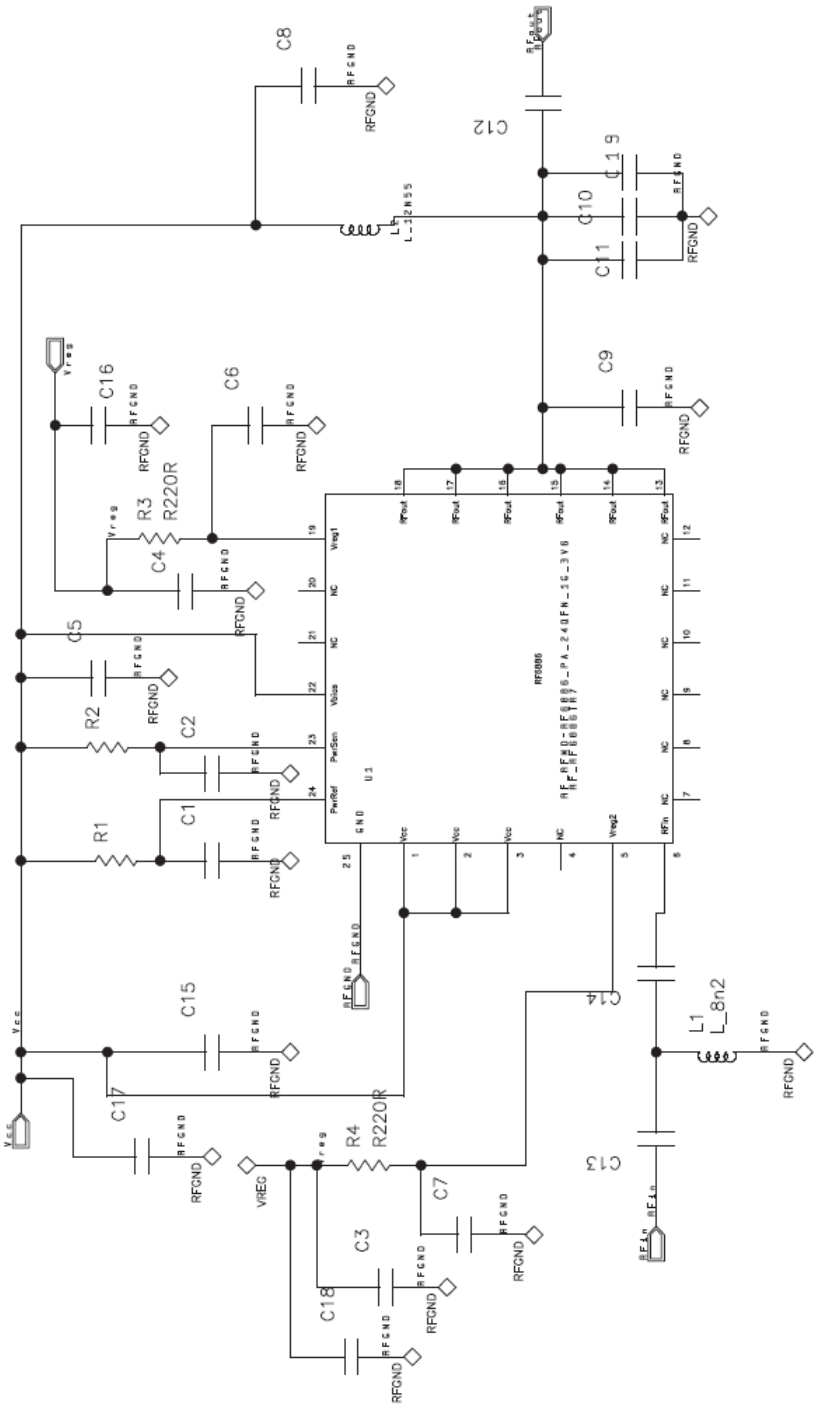

Figure 5.32. Schematic of the power amplifier

Chapter 6

Software

In this chapter is presented the firmware design of the On-Board Radio Frequency Module at 437 MHz, entirely written in C++. Will firstly be shown the complete class diagram, which contains all the main classes of the firmware developed, describing its architecture. A brief description of these classes is provided, but how the firmware will operate on the field is described with sequence diagrams and state machines. The code is kept light without overhead even when using modular and parametric functions and classes, by exploiting the templates of C++. A method can use a particular template in the same way as a parameter, when called in an object instantiated with such template, but therefore with no overhead.

6.1 Software organization

The software of the OBRF is written in C++ and therefore is object oriented. To handle classes and their relations, is used the UML environment in order to ease the development, by connecting classes in a visual way. Each class' object instantiated, is connected to the father class with an arrow. This is shown in figure 6.3.

As shown in figure 6.3, the top-level class is the *Bk1B31A2S_main*, since contains the *main()*. This class provides an object called *OBRF* which contains in turn the *Bk1B31A2S* class, the core of the OBRF firmware. This contains all the methods developed for the tile and a brief description of them is provided later.

There are other classes in this diagram, used in this firmware, developed by others AraMiS projects, and are the *MessageHandler*, *Housekeeping*, the *Bk1B4221W_Tile_Processor_4M* and the *CC1020*. Every class is explained during the chapter.

The *Bk1B31A2S* is interrupt driven, mostly from command generated interrupts from the *MessageHandler* and *Housekeeping* classes. The MessageHandler provides the capability of the tile

to always listening from the bus, which is I2C based, in order to get any command even when in stand-by. After a command has been received, the *intepret()* function is called. Another periodic interrupt source is coming from the Housekeeping, which calls the *housekeeping()* method every fixed amount of time, based on *Timer A0* of the tile processor. In this way all the housekeeping functions of the sensor classes are executed as shown in figure 6.31 and the proper vectors are updated, according to use cases related to the **housekeeping** vector, described in section 4.3.

The remaining interrupt sources are from the transceiver when receive or transmit, at the DCLK pin signal; the last interrupt source is from the *Timer A1* of the tile processor, which handles the timing of beacon, TX and RX timeout, the decision of when search for an incoming message from the antenna. In this chapter all of these mechanisms are going to be described.

Both in software and hardware are used the slots. A slot is an object which contains a well defined group of pins of the MCU hardware, according to an AraMiS protocol defined in 1B48. In classes are defined as operations the slots containing the pins called modules, where the modules of these pins are driven by software objects called SLOT_A, SLOT_B etc. This organization is reported also in chapter 5 since these classes are kept coherent with the hardware connections and the objects are representing also the physical ones.

When it is called *logical* module, it is referred to a slot mapped with the MCU's pins. When called *physical* module, it is referred also to the physical connector, which bring the connections to a mechanical slot. In figure 6.2 there is a class example of a logical module, with the relative objects that are mapping the MCU pins at software level (D0, D1 and so on). Note that the driver is used by the previous mentioned SLOT_A declared as object in the Tile Processor. An example of the organization of the logical slots is provided in 6.1; while a physical connector used for AraMiS module can be of any type containing 20 pins.

Conn	Pin	A	B	C	D
D0/RX/SOMI	11	P3.5/UCA0SOMI /UCA0RXD	P7.3/TA1.2	P5.7/UCA1SOMI/ UCA1RXD	P8.1/TA0.1
D1/TX/SIMO	9	P3.4/UCA0SIMO /UCA0TXD	P8.0/TA0.0	P5.6/UCA1SIMO/ UCA1TXD	P8.2/TA0.2
D2/SCL/SOMI	7	P3.2/UCB0SCL	P3.2/UCB0SCL	P5.4/UCB1SCL	P5.4/UCB1SCL
D3/SDA/SIMO	5	P3.1/UCB0SDA	P3.1/UCB0SDA	P3.7/UCB1SDA	P3.7/UCB1SDA
D4/CLK	3	P3.0/UCA0CLK	P3.3/UCA0STE	P3.6/UCA1CLK	P5.5/UCA1STE
D5/PWM	1	P4.0/TB0.0	P4.1/TB0.1	P4.2/TB0.2	P4.3/TB0.3
D6/A0	12	P6.0/A0	P6.2/A2	P6.4/A4	P6.6/A6
D7/A1	10	P6.1/A1	P6.3/A3	P6.5/A5	P4.7/TB0CLK/ SMCLK
D8/ID/INT	4	P1.0/TA0CLK/ ACLK	P2.5	P1.7	P1.6/SMCLK
D9/EN/PWM2/INT	2	P1.1/TA0.0	P1.2/TA0.1	P1.5/TA0.4	P1.4/TA0.3

Figure 6.1. Mapping of a logical slot

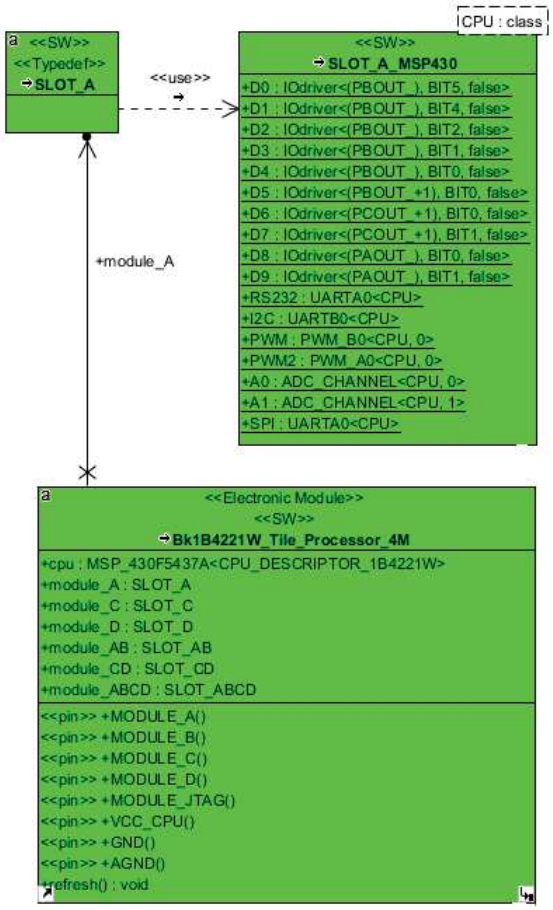

Figure 6.2. Class diagram of a logical slot

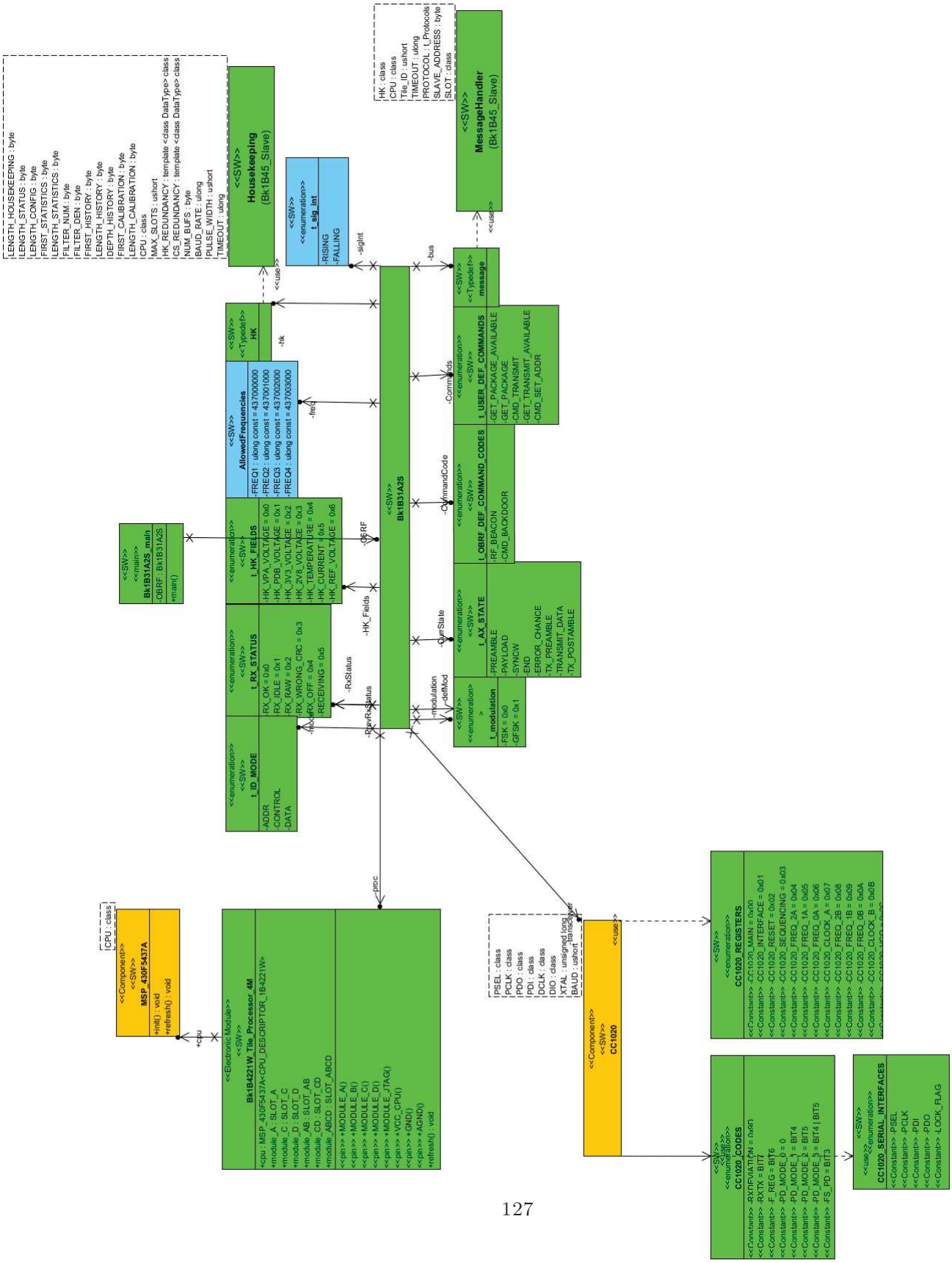

Figure 6.3. Class diagram of the firmware running on the OBRF

6.2 Algorithms and functions of Bk1B31A2S_main class

When the OBRF tile is powered up, will start its execution from the *Bk1B31A2S_main*. As described from figure 6.5, this will initialize all the systems in order to enable the interrupts and prepare the environment variables. In this section is provided a complete description of the algorithms implemented in this class. All the methods involved in sequence diagrams are documented individually at the end of each section, after a description of their interactions at system level.

6.2.1 Algorithm of the main() routine

In figures from 6.5 to 6.7 is shown the behaviour of the tile *main()* function. The part related to electrical connections is highlighted in figure 6.4, which is still part of the entire sequence diagram. In these diagrams (from figure 6.4 to 6.7) is explained how are made the connections between the OBC and the 1B31 On-Board Radio Frequency Module for what concerns the data bus 1B45 and transceiver connections. The I2C protocol is implemented on the logical *module_B : SLOT_B* of the **Bk1B4221W_Tile_Processor_4M** class, connected to the I2C pins of the OBC's module physical connector, named *MODULE_OBC()*.

Steps from 4 to 7 are showing the transceiver's connections under the SPI protocol, even though there will be used the *bit banging*, therefore the SPI hardware is not directly needed, introducing a greater flexibility w.r.t. microcontroller's port used. Those steps are needed to connect the **Bk1B31A2_Transceiver_437MHz** and the logical *module_A : SLOT_A* of the **Bk1B4221W_Tile_Processor_4M**.

In steps from 9 to 11 are used 2 GPIO pins, DIO() and DCLK(), which are respectively the data pin in which the system processor should be able to read from and write to (so to be bidirectional), and the pin that is the transceiver's clock both in TX and RX, in which the system is programmed to trigger an interrupt which executes the *isr_CC1020RxData() : bool* at every rising edge, or the *isr_CC1020TxData()* at every falling edge, if needed.

The execution of the *main()* is described in events starting from step 10 (from figure 6.5) and after the proper initializations, the firmware loops forever. It is interrupted upon a transition on DCLK() or upon interrupts from OBC, which can still issue commands but on a lower priority interrupt w.r.t. to the one related to RF reception and transmission. The AX.25 Unpacking (shown in figure 6.15) is called as soon a raw packet, with the destination address equal to **AX_SAT_ADDR : char const**, is fully received. In this loop is also checked the **SendBeacon : byte** variable, in order to see if it is the time to prepare the beacon packet. If this is the case, after beacon data has been prepared, the variable is reset and the transmission begin, interrupt driven by the transceiver (the TI CC1020).

Steps from 15 to the end (figure 6.7) are showing the possible ISRs that can be triggered. Here are shown the interrupt driven functions *housekeeping(index : ushort)*, *interpret(command : ushort)*, *isr_CC1020RxData()*, *isr_CC1020TxData()* and *isr_timerA1()*.

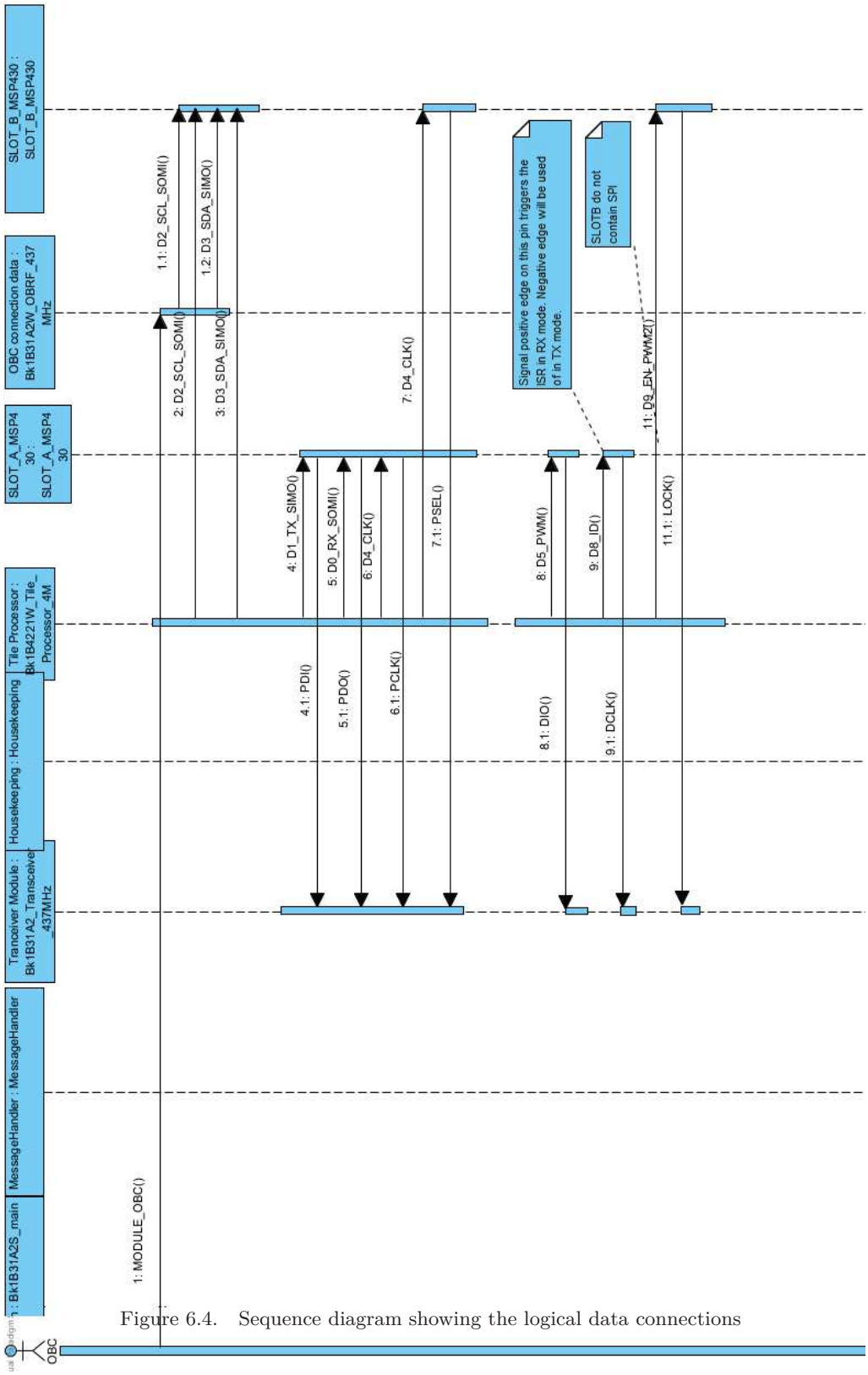

Figure 6.4. Sequence diagram showing the logical data connections

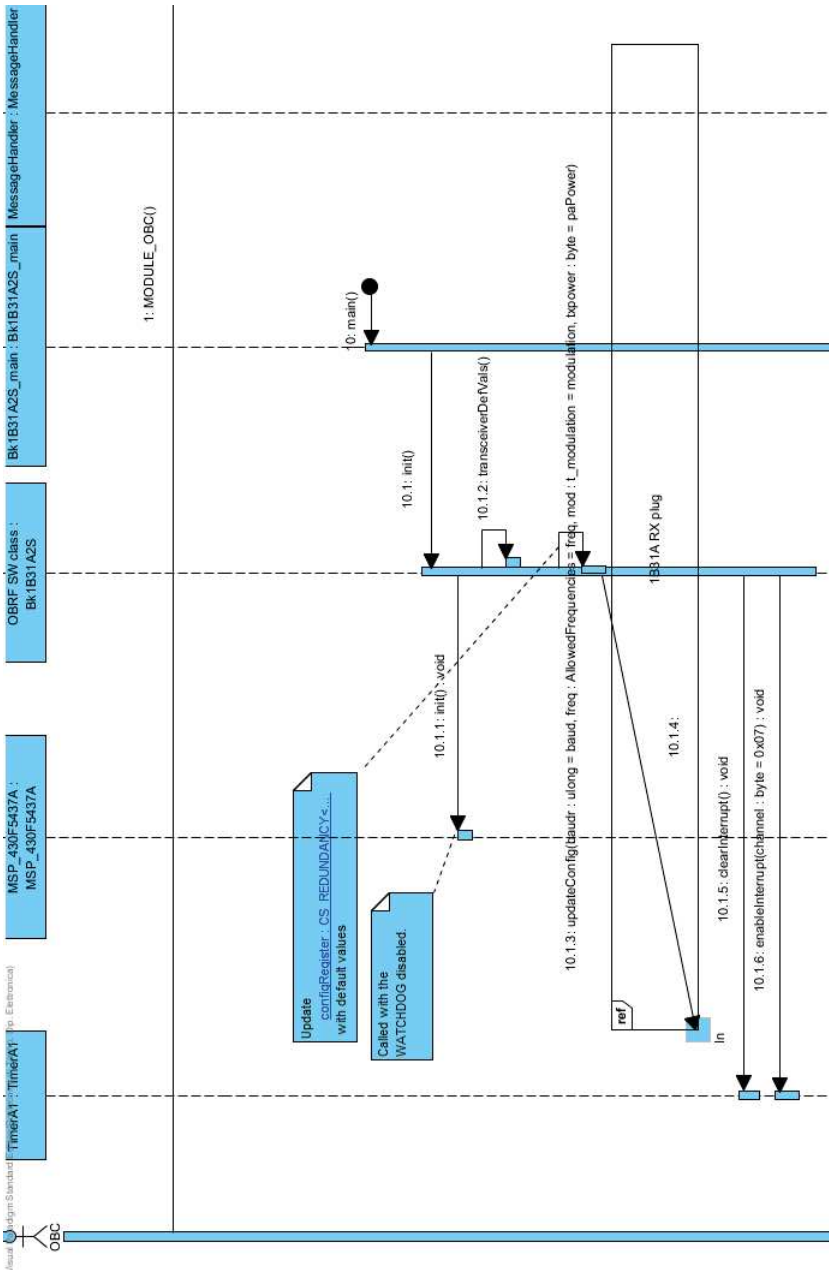

Figure 6.5. Sequence diagram of the main firmware function, 1/3

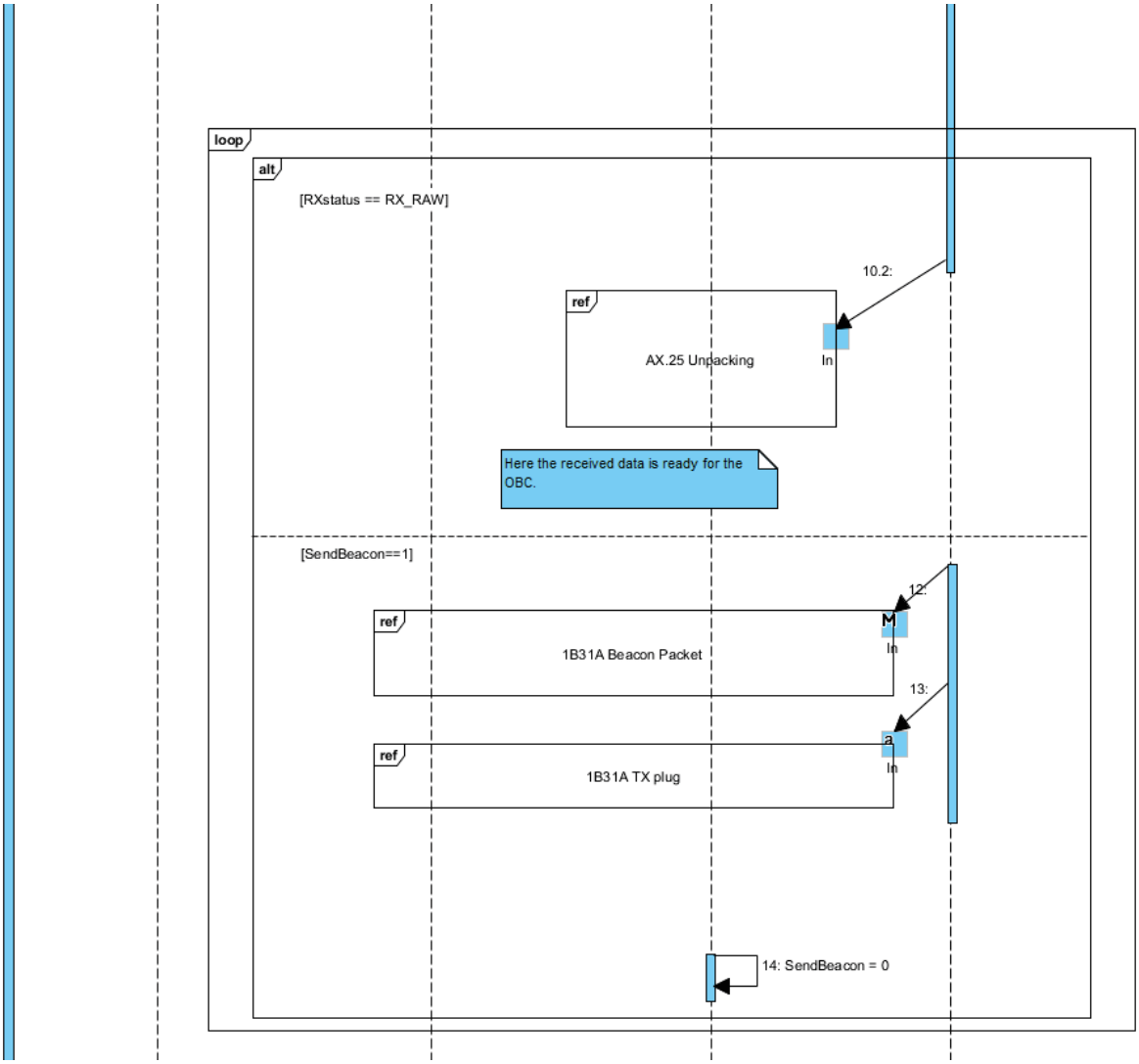

Figure 6.6. Sequence diagram of the main firmware function, 2/3

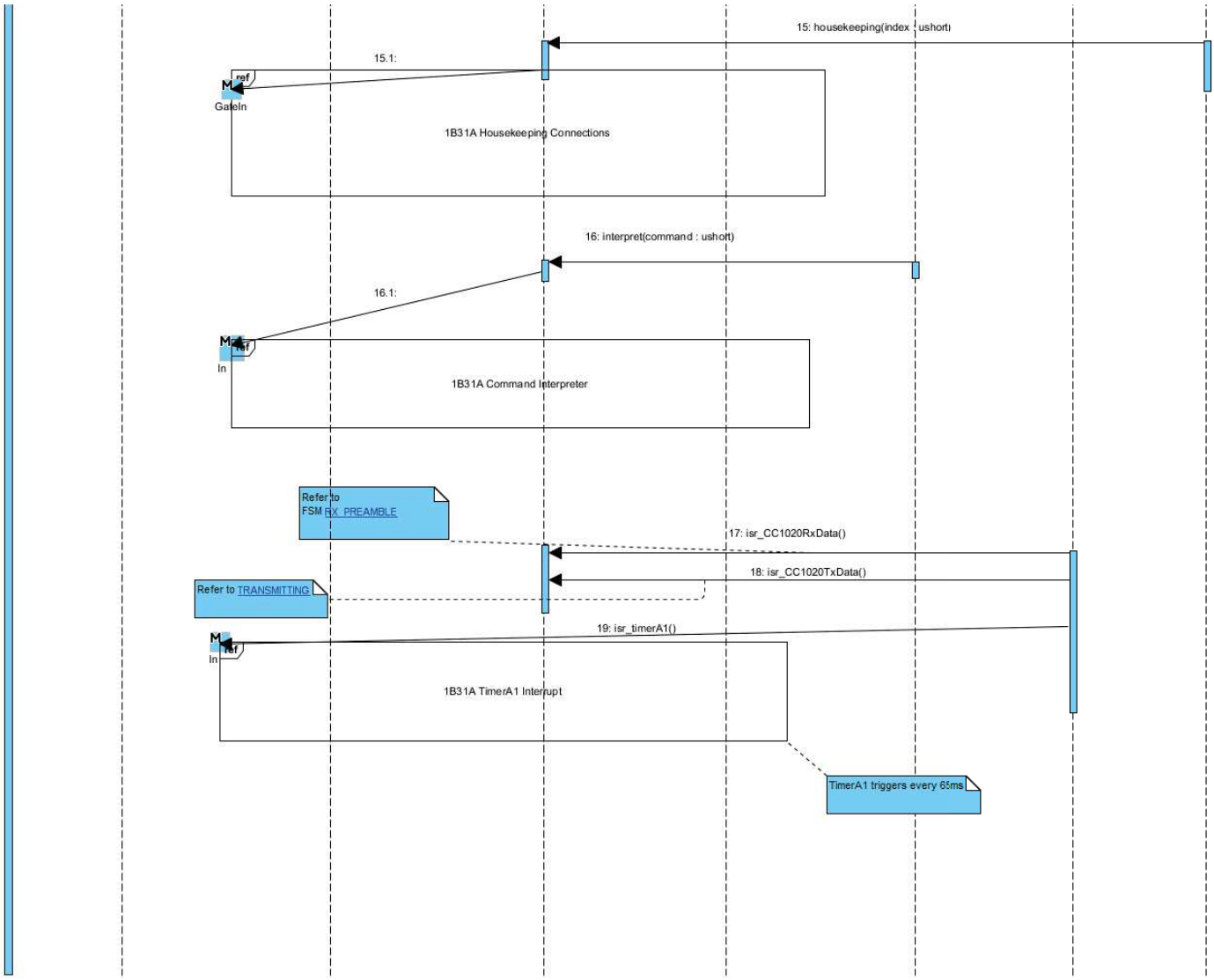

Figure 6.7. Sequence diagram of the main firmware function, 3/3

Here below is shown the description and implementation of attributes and methods of class *Bk1B31A2S_main*, which are developed on the basis of the algorithm described before.

6.2.2 main()

This function initialize the *Bk1B4221W_Tile_Processor_4M* according to defined templates. Then initialize the *Bk1B31A2S* class calling its *init()*.

In main loop are checked changes of the **RxStatus : t_RX_STATUS** to control the packet availability, and if the status is RX_RAW is called the *ax25unpack()* to prepare the received RF data. It is checked also if a RF Beacon must be sent. If so, it is prepared a proper beacon buffer to be sent, through the *beaconPack()*, therefore the *Bk1B31A2_Transceiver_437MHz* unit is initialized for the transmission of the RF Beacon, calling the *CC1020InitTX(baudr : ulong, freq : AllowedFrequencies, mod : t_modulation, txpower : byte)* and resetting the **SendBeacon**. The system will transparently continue the beacon transmission using the interrupt coming from the transceiver, upon the whole buffer has been sent.
Code:

```
#include "Bk1B31A2S_main.h"
#include "Bk1B31A2S.h"
Bk1B31A2S OBRF;
main() {
OBRF.proc.cpu.init();
OBRF.init();

while(1){
if (OBRF.RxStatus == RX_RAW){
OBRF.ax25unpack();
   }

if (OBRF.SendBeacon){
OBRF.beaconPack();
OBRF.CC1020initTX(OBRF.baud, OBRF.freq, OBRF.modulation, OBRF.paPower);
SendBeacon = 0;
}
}
}
```

6.3 Transceiver CC1020 class and algorithms

Here will be described how all the RF parameters which are applied to the transceiver were devised. These values are going to be used in the system through the software class Bk1B31A2S.

The Bk1B31A2_Transceiver_437MHz unit contains at its core a CC1020 transceiver chip. The

comments on component selection are made in chapter 5. The transceiver can controls the communication channel in the Frequency range 402 MHz - 470 MHz. In UML is described the transceiver unit with hardware and software classes together: in this way the internal class of the Bk1B31A2_ Transceiver_437MHz unit, called *CC1020*, contains both hardware characteristics and firmware methods. For this reason this *CC1020* class is instantiated as *Bk1B31A2S*' object, called simply **transceiver : CC1020**. These classes are shown in figure 6.3. The *CC1020* class is instantiated by using template parameters for the pin definitions, and other values like the baudrate and the tranceiver's crystal frequency.

6.3.1 The CC1020 digital interface

The CC1020 transceiver provide a digital interface, SPI-based, with the microcontroller. In classes *Bk1B31A2S* and *CC1020* are present all the methods which are supporting its correct behaviour. Here will be described the configuration procedure adopted and the transceiver specific methods developed after a description of its behaviour.

Through the programmable configuration registers the following key parameters can be programmed:

- Receive / transmit mode
- RF output power
- Frequency synthesizer key parameters: RF output frequency, FSK frequency separation, crystal oscillator reference frequency
- Power-down / power-up mode
- Crystal oscillator power-up / powerdown
- Data rate and data format (NRZ, Manchester coded or UART interface)
- Synthesizer lock indicator mode
- Digital RSSI and carrier sense
- FSK / GFSK / OOK modulation [11]

The SPI implementation on class *CC1020* is currently made with bit-banging, i.e. it is followed the SPI protocol in software in order to make it available at any digital pin of the MCU. The class *Bk1B31A2S* provides the object **transceiver : CC1020** to talk to the transceiver CC1020 chip and methods in *CC1020* class.

CC1020 Serial Peripheral Interface description

CC1020 is configured via a SPI-compatible interface (PDI, PDO, PCLK and PSEL pins in figure 6.4) where CC1020 is the slave. There are 8-bit configuration registers, each addressed by a 7-bit address. A Read/Write bit initiates a read or write operation. A full configuration of CC1020 requires sending 33 data frames of 16 bits each (7 address bits, R/W bit and 8 data bits). In appendix A in figure A.1, are shown the accessible registers used by this software. All registers are also readable.

The time needed for a full configuration depends on the PCLK frequency: this is set inside the *CC1020* class. During each write-cycle, 16 bits are sent on the PDI-line. The seven most significant bits of each data frame (A6:0) are the address-bits. A6 is the MSB (Most Significant Bit) of the address and is sent as the first bit. The next bit is the R/W bit (high for write, low for read). The 8 databits are then transferred (D7:0). During address and data transfer the PSEL (Program SELect) must be kept low. See figure 6.8. The method used to implement this protocol is the *SetReg(data : byte, register : byte)*.

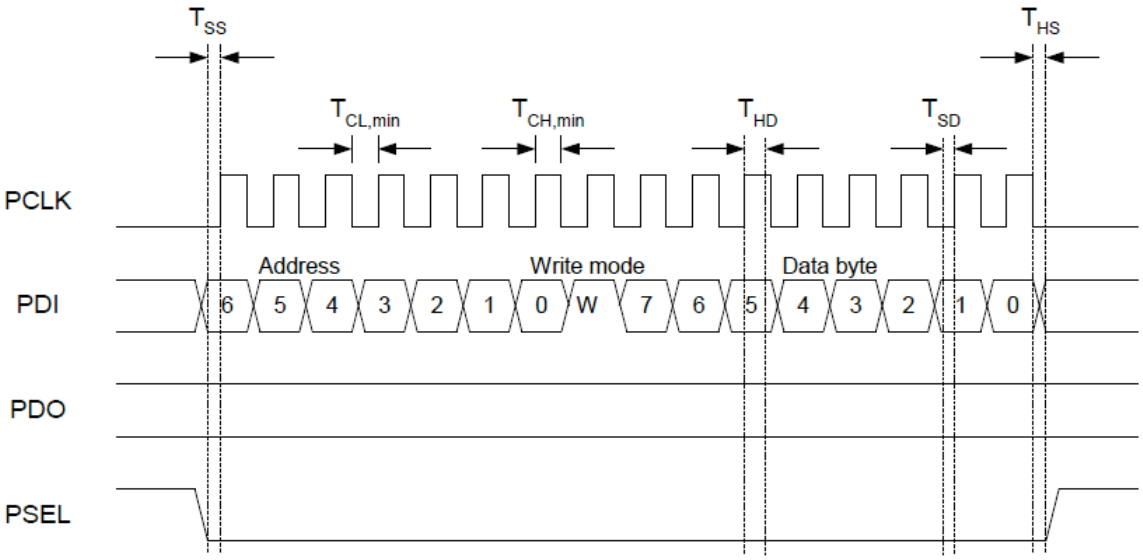

Figure 6.8. Configuration registers write operation

The clocking of the data on PDI is done on the positive edge of PCLK. Data should be set up on the negative edge of PCLK by the microcontroller. When the last bit, D0, of the 8 data-bits has been loaded, the data word is loaded into the internal configuration register. The configuration data will be retained during a programmed power down mode, but not when the power supply is turned off. The registers can be programmed in any order. To increase the dependability, at every new transceiver's configuration all the registers are rewritten even though there was no power down.

The configuration registers can also be read by the microcontroller via the same configuration interface. The seven address bits are sent first, then the R/W bit set low to initiate the data read-back. CC1020 then returns the data from the addressed register. PDO is used as the data output and must be configured as an input by the microcontroller. The PDO is set at the negative edge of PCLK and should be sampled at the positive edge. The read operation is illustrated in figure 6.9. PSEL must be set high between each read/write operation. [11] The method used to implement this protocol is the *ReadReg(register : byte) : byte*.

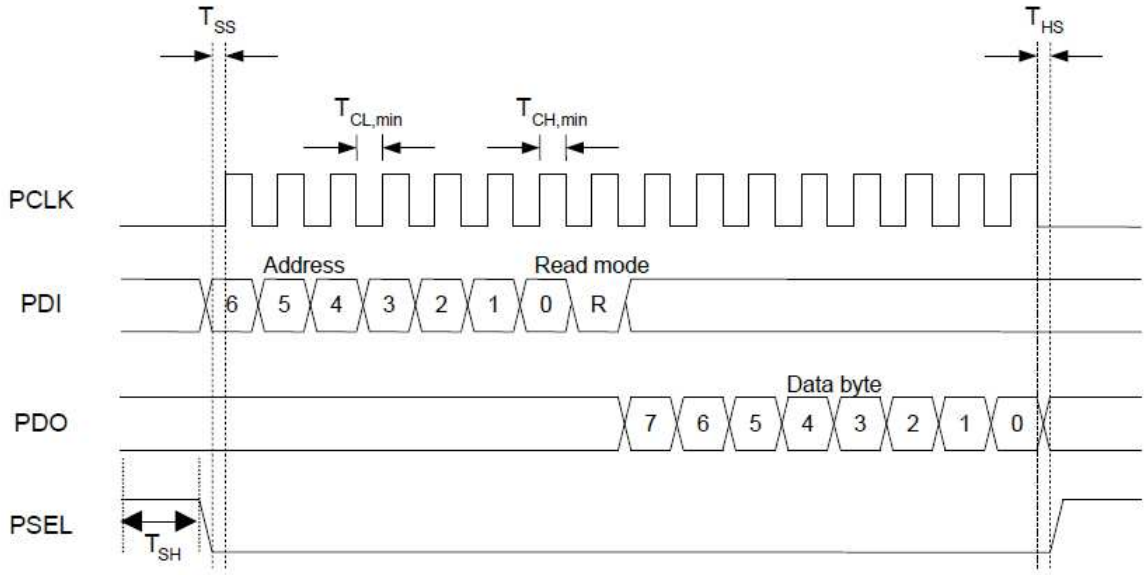

Figure 6.9. Configuration registers read operation

6.3.2 ReadReg() and SetReg()

In these methods of CC1020 class is implemented the bit-banged SPI read and write interface on the MCU. This uses the 4 pins PSEL, PCLK, PDO and PDI. PSEL is reset (where the initial

value of the others don't care) and the interface is active. At every bit sent or received, is toggled the PCLK with a period define by the *WAIT_CYCLE()*. This period must be in the SPI speed bounds defined in the datasheet.

6.3.3 The CC1020 signal interface

The RF data is not redirected in the SPI interface, but on two pins, DCLK and DIO, used for an NRZ synchronous communication with the MCU. The data on the RF side can be configured to be NRZ or automatically transformed in Manchester coding. The data format is controlled by the DATA_FORMAT[1:0] bits in the MODEM register.

CC1020 is used for synchronous NRZ mode, this requires, in transmit mode, the presence of data at DIO pin from the MCU, while the MCU itself will be synchronized on DCLK signal provided by the transceiver. Data is clocked into CC1020 at the rising edge of DCLK. The data is modulated at RF without encoding, if no manchester is used.

In receive mode CC1020 performs the synchronization and provides received data clock at DCLK and data at DIO pins. The data should be clocked into the interfacing circuit at the rising edge of DCLK. Figure 6.10 shows the behaviour. [11]

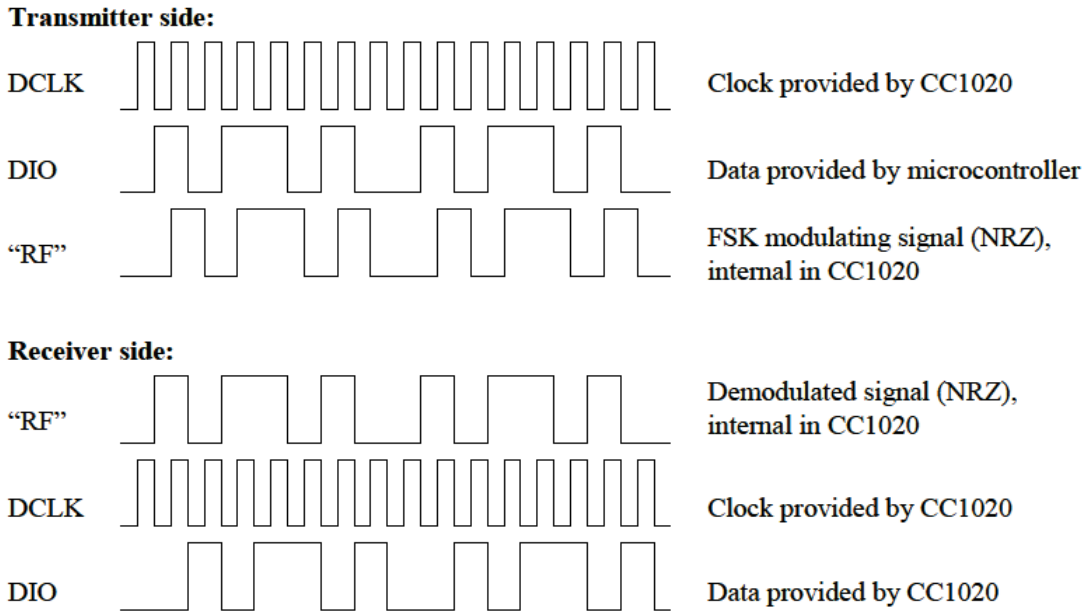

Figure 6.10. Synchronous NRZ mode

Note that in case of Manchester mode, the baudrate of the RF transmission is half of the chosen one, due to the doubled transition when transmitting a single bit. In fact Manchester code is

based on transitions; a “0” is encoded as a low-to-high transition, a “1” is encoded as a high-to-low transition. See figure 6.11. This ensures that the signal has a constant DC component, which is

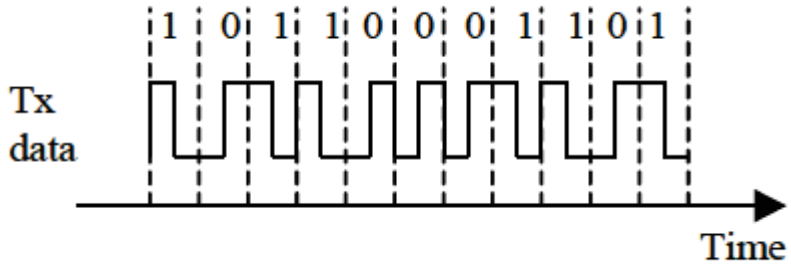

Figure 6.11. Manchester encoding

necessary in some FSK demodulators. At this first progress of the project, the AX.25 protocol is implemented without this coding, because the bit-stuffing already reduces this DC condition.

The data in reception on the MCU side is handled and synchronized on DCLK with the *isr_CC1020RxData()*. In transmission is used the *isr_CC1020TxData()*, synchronized on the same signal.

6.3.4 Transceiver’s configuration

The configuration of the RF circuits and the various transceiver settings are devised also using SmartRF Studio from TI. This chip contains two identical modules for setting up to two parallel configurations, named A and B, for TX and RX modes, allow a fast switch between them without reconfiguring the system. Here the configuration assign the settings A to RX, while B for TX mode.

Here are derived the main parameters for an FM based radio-link. Generally speaking, a frequency modulated carrier is made by a carrier frequency, called center frequency, which carries the baseband signal. This signal is coded using a simple Binary Freqeuncy-Shift Keying modulation, which consists of varying the carrier frequency between two extreme frequencies around the center one. This variation can be instantaneous or smoothed by a gaussian filter, obtaining the Gaussian BFSK, and defines the *frequency deviation*, representing the absolute difference between the carrier and the modulated frequency, as shown in figure 6.12.

The occupied bandwidth is given by the sum of the doubled frequency deviation, which is called the *frequency separation* and it is the bandwidth occupied by the baseband signal (see section 6.3.5). But the filter bandwidth that will need to be defined must be greater than this value,

including all the drifts due to errors and the doppler effect, see figure 6.12. In that picture is shown how the filter is oversized w.r.t. the ideal bandwidth. This is done to cover the system errors and deviation, and to obtain an efficient channel separation with high frequency drifts, a channel should be reasonably outside the filter, a shown again in that figure, pointed out by the Channel Spacing. According with these considerations, using SmartRF Studio and choosing a channel spacing, the filter bandwidth is automatically chosen.

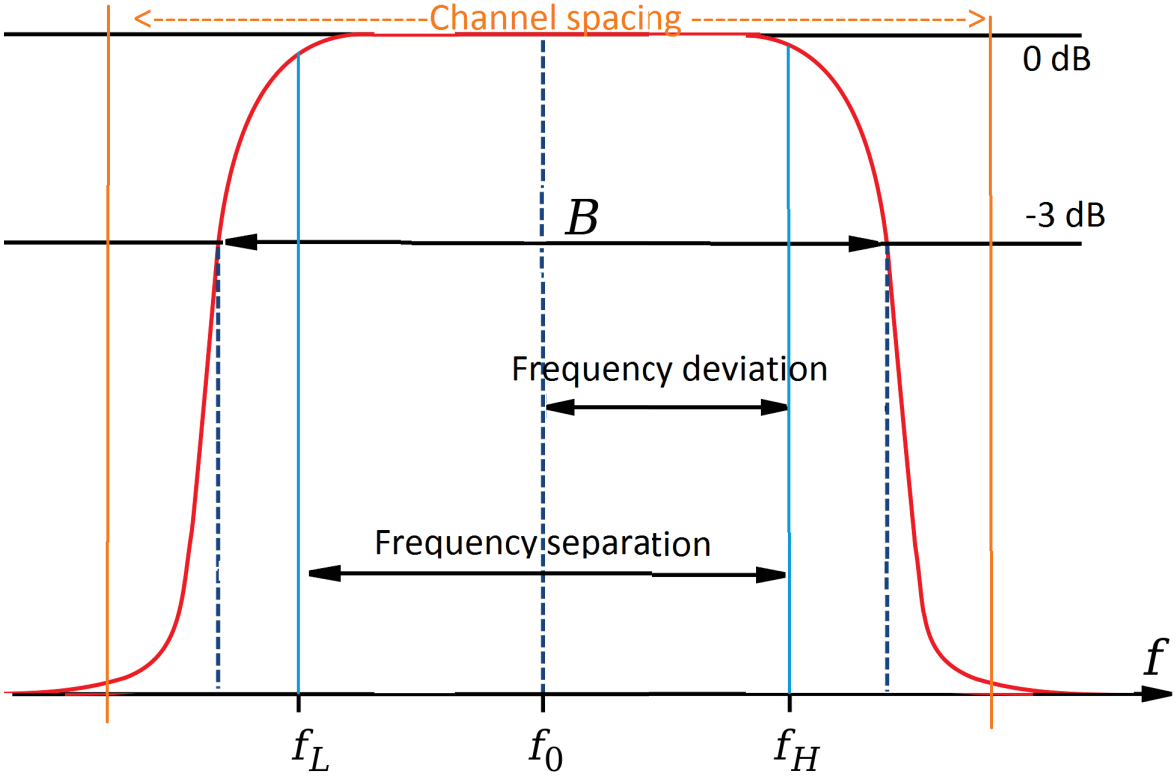

Figure 6.12. FM spectrum organization

The SNR and the bandwidth are also dependent with the ratio of the frequency deviation and the baseband signal frequency. It is expressed as a *modulation index*, here called h, as shown in equation below:

$$h = \frac{\Delta f}{f_m} = \frac{\Delta f}{\frac{1}{T_s}} \tag{6.1}$$

Where Δf is the frequency deviation, f_m is the highest frequency of the digital signal, using T_s as the symbol period [14]. This value is $\ll 1$ for narrowband systems. With transceivers of this family, lowering the modulation index will increase (degrades) the level of sensitivity, and going lower than 0.5 will increase very steeply the sensitivity level. The higher bound of the modulation index is the increase of bandwidth if too high, because every doubling of the filter bandwidth will halve the sensitivity. As a conclusion, a good (and common) design compromise is using modulation index around 1. This is also suggested by SmartRF Studio.

The datarate, referred as *baudrate* in use case in figure 4.5, is chosen to be at 9600 baud per second, which corresponds to the actual bits per second since the NRZ is used. Note that if Manchester is used, the programmed baudrate corresponds to the one on RF side, therefore the bitrate is half of the programmed baudrate. Capital letters of equations of this section refers to digital register values of the CC1020. This is the formula from which the registers values were devised:

$$Baudrate = \frac{f_{xosc}}{8 \cdot (REF\_DIV + 1) \cdot DIV1 \cdot DIV2} \tag{6.2}$$

Where the f_{xosc} is passed to the class as a template. Texas Instruments advise to use a frequency of 14.7456 MHz, therefore is used that due to the factory tests and guaranteed values are devised with that frequency.

Then must be decided the carrier frequency to be generated by transceiver. This is based on a reference frequency for which a timebase crystal of 14.7456 MHz will not introduce any error because values of internal registers are always integer numbers. A dithering can be automatically implemented by the transceiver, modifying the carrier according to DITHER parameter. The formula for carrier frequency is:

$$f_c = f_{ref} \cdot \left(\frac{3}{4} + \frac{FREQ + 0.5 \cdot DITHER}{32768} \right) \tag{6.3}$$

where f_{ref}:

$$f_{ref} = \frac{f_{xosc}}{REF\_DIV + 1} \tag{6.4}$$

The FSK modulation frequency can be set in DEVIATION register, setting the frequency spacing from the carrier. An f_{dev} is set, where $f_0 = f_c - f_{dev}$ when sending a '0' and $f_1 = f_c + f_{dev}$ when sending '1'. Where in TX mode:

$$f_{dev} = f_{ref} \cdot TXDEV\_M \cdot 2^{TXDEV\_X-16} \tag{6.5}$$

In RX mode is programmed the local oscillator in order to achieve $f_{lo} = f_c - f_{if}$, and from datasheet the ideal intermediate frequency is designed to be $f_{if} = 307.2kHz$ and a value as close as possible to that should be used, according to:

$$f_{if} = \frac{f_{osc}}{8 \cdot ADC\_DIV \cdot 2^{TXDEV\_X-16}} \tag{6.6}$$

The further consideration is the bit and word synchronization. This is needed to devise a correct number for the FLAG_THR in reception and FLAG_THR_TX in transmission. The data slicer, to make the bit decision, uses an average value of both maximum and minimum frequency deviations detected: the expected received frequency deviation is set using RXDEV_M and RXDEV_X in the same way as in eq. 6.5. The minimum bit transitions used to made the average are set in AFC_CONTROL register. In this case is set the maximum, four, to achieve better quality of decision.

The AX.25 protocol compatibility requires that between one packet and another is sent the AX_FLAG = 0x7E, this contains 2 transitions on RF side when coded with NRZ. For setting up the AGC, synchronizer and data slicer, are recommended 3 bytes, including the four transitions set in AFC_CONTROL, of values 0xAA or 0x55 coded on RF side (containing 7 transitions per byte). Therefore, if the AX_FLAG is sent, are needed at least 11 bytes for setting the receiver correctly. The word synchronization is then performed on the basis of this flag, when adopting the algorithm in figure 6.26.

6.3.5 Filter parameters selection

The overall signal bandwidth is needed to set accordingly the FILTER register. The signal bandwidth SBW is defined using the Carson's rule:

$$SBW = 2 \cdot fm + 2 \cdot f_{dev} = baudrate + freq\_separation \tag{6.7}$$

where fm is the maximum frequency of modulating signal. In NRZ mode occurs when transmitting a 0-1-0 sequence, therefore $2 \cdot fm$ is the programmed bitrate, because of transmitting two different bits. With Manchester, occurs when transmitting a continuous 1's or 0's.

The filter bandwidth must include the crystal errors and other frequency deviations, like the doppler effect. This bandwidth should be:

$$ChBW > baudrate + frequency\_separation + 2(2 \cdot XTAL\_ppm \cdot f_c + f_{doppler}) \tag{6.8}$$

At 9600 baud per second, the frequency separation suggested as a starting point is 9900 Hz. This is also suggested by the TI SmartRF Studio software. With 11 kHz of doppler and a crystal of 14.7456 MHz +/- 2.5ppm and carrier frequency of 437MHz, the total minimum channel bandwidth is

$$ChBW \geq 45.8kHz \tag{6.9}$$

according to equation 6.8. If too less sensitivity is measured, the bandwidth can be reduced. The filter bandwidth can be adjusted by tuning the bits [0-4] of FILTER register, at compile-time. The channel bandwidth (ChBW) is set using the FILTER register by means of

$$ChBW = \frac{307.2}{DEC\_DIV + 1} \tag{6.10}$$

where the intermediate frequency is 307.2 kHz.

Typical receiver sensitivity values are reported in figure 6.13. At 9.6 kBaud the advised deviation is 9.9kHz with 25.6 kHz of filter BW. Widening to 51.2 kHz reduces sensitivity of $10 \cdot \log\left(\frac{51.2}{25.6}\right) = 3dB$, as proven in figure. Due to the wideness of filter, the sensitivity can be still improved by widening the frequency deviation, therefore obtaining a modulating index M grater than 1, so becoming compliant with the sensitivity specifications. But the advised frequency deviation from SmartRF studio is kept, since the configuration is still under the minimum sensitivity value.

It is important to remind that reducing the baudrate allows to reducing the filter, therefore increasing the sensitivity. These are mission dependent values, and must be re-elaborated the corresponding passive components for the transceiver, possibly using the SmartRF Studio from TI, if the specifications will be changed.

Finally, the adopted characteristics of the channel are 9600bps data rate, $f_{dev} = 9.9kHz$, $BW = 51.2kHz$, GFSK modulation and NRZ coding.

6.4 Algorithms and functions Bk1B31A2S class

This class behave mostly with interrupts, and it is an object of the *Bk1B31A2_main*. Except for few functions called by the *Bk1B31A2_main* for the tile initialization, everything works upon external or timer driven interrupts, which are:

Data rate [kBaud]	Channel spacing [kHz]	Deviation [kHz]	Filter BW [kHz]	Sensitivity [dBm]		
				NRZ mode	Manchester mode	UART mode
2.4 optimized sensitivity	12.5	± 2.025	9.6	-115	-118	-115
2.4 optimized selectivity	12.5	± 2.025	12.288	-112	-114	-112
4.8	25	± 2.475	19.2	-112	-112	-112
9.6	50	± 4.95	25.6	-110	-111	-110
19.2	100	± 9.9	51.2	-107	-108	-107
38.4	150	± 19.8	102.4	-104	-104	-104
76.8	200	± 36.0	153.6	-101	-101	-101
153.6	500	± 72.0	307.2	-96	-97	-96

Figure 6.13. Typical receiver sensitivity as a function of data rate at 433 MHz, FSK modulation

- OBC requests
- Presence of carrier
- Internal timer

Here are implemented a lot of algorithms for as many functions, which are defined in the previous use cases chapter. The diagram from figure 6.5, shows the firsts methods of the *Bk1B31A2S* that are called. The complex ones should follow a proper algorithm, which are described in this section. The algorithms have a C++ implementation which is herein described individually.

6.4.1 init()

Called by the class *Bk1B31A2_main*, initialize the CPU, disabling the watchdog (see the templates used for the **proc : Bk1B4221W_Tile_Processor_4M**). Then initialize the transceiver's parameters, which are:

- power amplifier set to maximum
- baudrate to default value for the AX.25 at 9600 bps
- modulation set to GFSK
- the carrier frequency of default channel chosen

Then the configuration is updated to **configRegister** with the *updateConfig(baudr : ulong, freq : AllowedFrequencies, mod : t_modulation, txpower : byte).* Then the Bk1B31A2_Transceiver_437MHz is initialized with the *CC1020InitRX(baudr : ulong, freq : AllowedFrequencies, mod : t_modulation, txpower : byte)* called with the just set parameters, because the RX mode is a default one.

Finally, all the necessary interrupts are initialized:

- timer : TimerA0 interrupt for housekeeping functions
- timerA1 : TimerA1 interrupt for system tick functions
- uartB0 : UARTB0 for the I2C interrupts from MODULE_OBC(), handled by the MessageHandler's init()
- uartB1 : UARTB1 for the I2C external antenna deploy control, the bus is connected to both of the redundant antenna device bus, see figure 6.14.

In this implementation the Bk1B31A2_Transceiver_437MHz uses bit-banging for the SPI interface, therefore the SPI interrupt of **uartA0 : UARTA0** is not initialized.

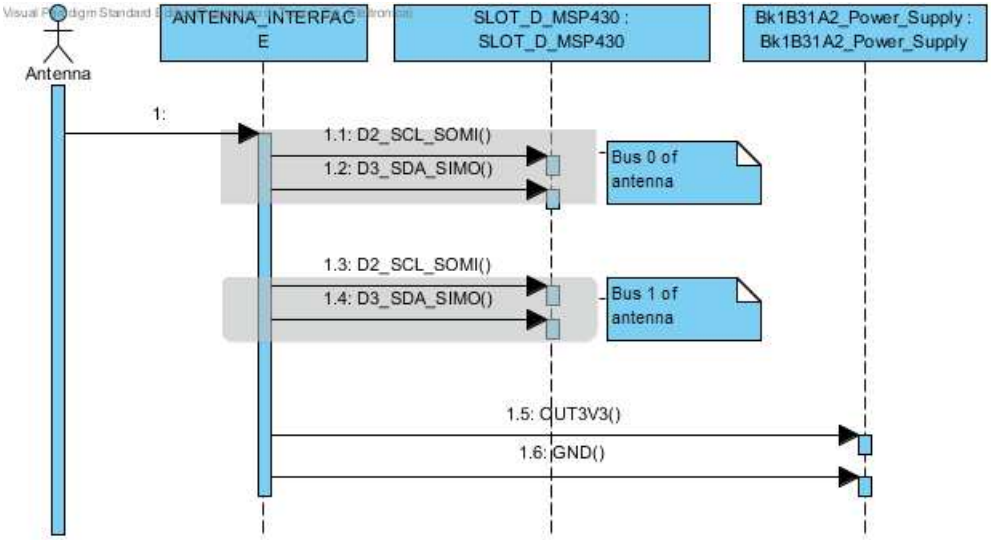

Figure 6.14. External antenna connections

Code:

```
Bk1B31A2S::init() {
proc.cpu.init();

//var inits
transceiverDefVals();
//init pins for RXmode, default
updateConfig(baud, freq, modulation, paPower);
// apply config
CC1020InitRX(baud, freq, modulation, paPower);
// Now if PSEL is toggled, the CC will search for an RSSI

bus.init(); //handler init
hk.init();
```

```
proc.cpu.uartB1.enable(I2C_MASTER_MODE, 9600);
//Main timerA1 already initialized
//Leave A0 for housekeeping interrupt routine, use A1 for another ovflw interrupt
//init what is needed for 1b45 etc but no for DCLK
proc.cpu.timerA1.clearInterrupt();
proc.cpu.timerA1.enableInterrupt(0x07); //ovflw mode
}

 Bk1B31A2S::transceiverDefVals() {
paPower = defPaPower;
baud = defBaud; //def vals
modulation = GFSK;
freq = FREQ1;
memcpy(addressTo, AX_DEFAULT_DEST_ADDR,
 AX_ADX_LEN);
}
```

6.4.2 AX.25 Unpacking algorithm

This algorithm is represented in figure 6.15. It is used by the *Bk1B31A2_main* class when the OBRF status become RX_RAW. This status means that a received frame must be unpacked in order to extract the AX.25 informations from it and this is what the AX.25 unpacking does. This diagram is a visual structure of the *ax25unpack()*. From step 1.1 on, is called the *subfieldID(buffer : char *, subBuff : char *, start : short &, mode : t_ID_MODE) : bool* one time for every field present in the received AX.25 packet: the implementation of this subfield identification method is described later. At the end of the unpacking, the *main()* will have all the useful data from the frame, ready to be sent to the OBC according to use case Get Received Packet in section 4.2.4. In the AX.25 fields documentation are described all of the types of data which is handled by the subfieldID() (read the AX.25 protocol use case). These fields are here implemented in **addressFrom : uchar[7]**, **addressTo : uchar[7]**, **nr : uchar**, **ns : uchar**. The **auxBuff : uchar[BUFFLEN]** contains the payload, while the **crc : ushort** is the FCS of AX.25 protocol.

In step 1.6 if the *subfieldID(buffer : char *, subBuff : char *, start : short &, mode : t_ID_MODE) : bool* returns TRUE, there is a problem with the packet composition, therefore the RX_WRONG_ CRC is set already here, and it is a worst condition than in step 1.10 because the packet can be not formatted right, showing a protocol mismatch. In step 1.8, the **RxStatus : t_RX_STATUS** is set to RX_IDLE if the packet fails the address check, invalidating all the potential data resetting the receiver status. This is the same value which is set after a transmission to the OBC occurs, since the packet now is transferred and should be no more present in the OBRF. Always in step 1.8, the **RxStatus : t_RX_STATUS** is reset because could also be a

CMD_BACKDOOR, therefore should not be available to the OBC. If no error takes place, at the end all the AX.25 parameters are extracted from the receiving buffer.

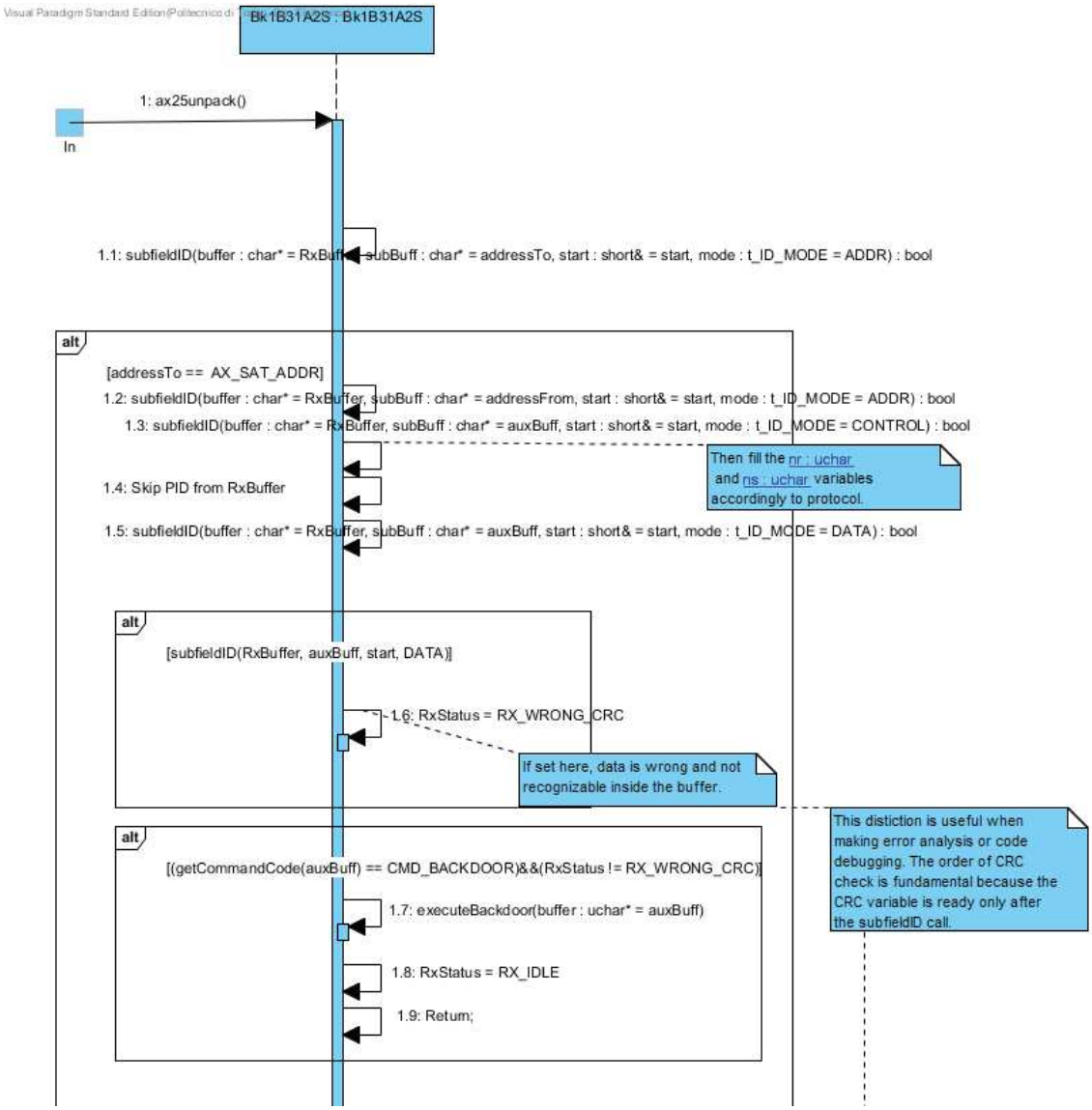

Figure 6.15. Sequence diagram of the AX.25 Unpacking procedure, 1/2

6.4.3 ax25unpack()

This section provides the function implementation of the algorithm in section 6.4.2. This function takes **RxBuffer : uchar[BUFFLEN]** in which a complete AX.25 frame is present, not necessarily CRC correct. The packet is assumed to be fulfilled with the ending **AX_FLAG : byte const**,

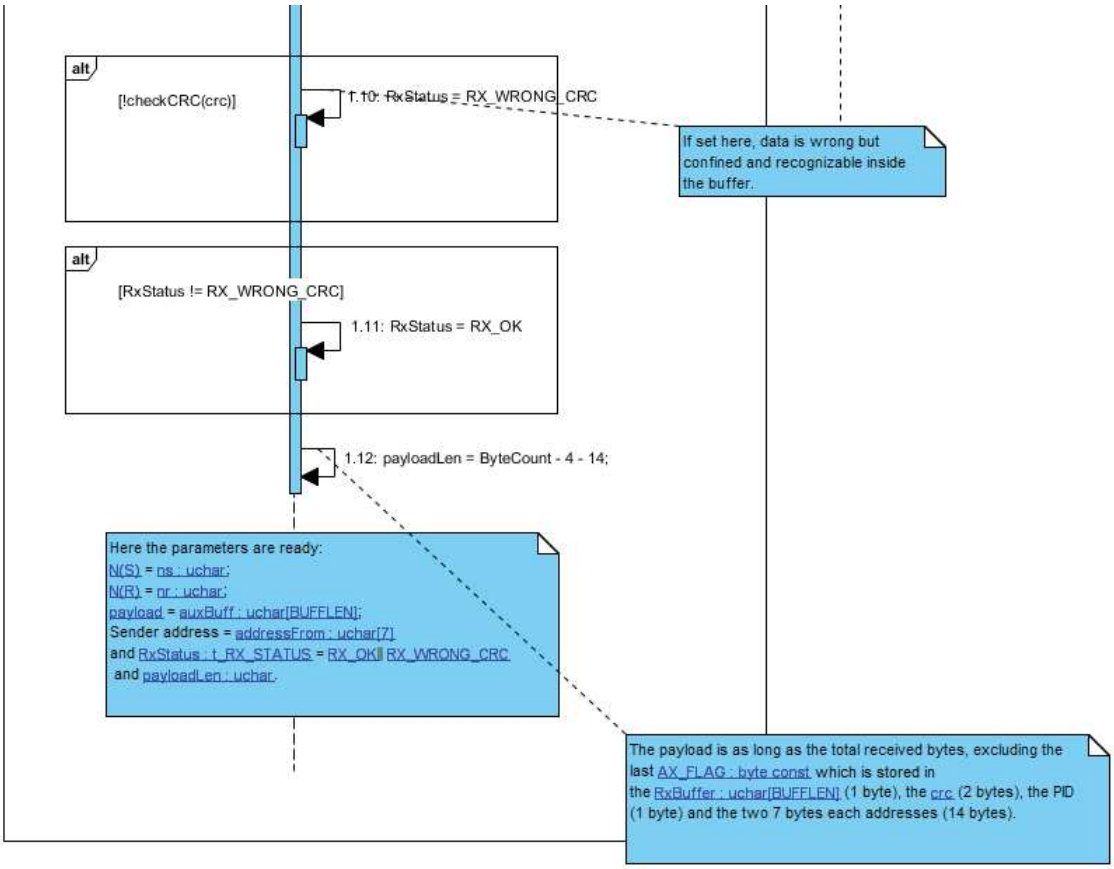

Figure 6.16. Sequence diagram of the AX.25 Unpacking procedure, 2/2

because the starting one was needed only for the synchronization.

Then the *callsign* of the destination of the packet is checked, by reading the **addressTo : uchar[7]**, generated by the call of *subfieldID(buffer : char *, subBuff : char *, start : short &, mode : t_ID_MODE) : bool*. The next step is to find the sender address, in the same way, but storing it in **addressFrom : uchar[7]**.

The CONTROL byte is checked to achieve the sequence numbers that are needed by the OBC and copying them in **nr : uchar** and **ns : uchar**. Note that the PID byte has been trashed since it is not used any layer 3 protocol. See the AX.25 protocol conventions used.

The *checkCRC(crc : ushort)*, if needed, will set a flag **RxStatus : t_RX_STATUS** = RX_WRONG_CRC, otherwise will be RX_OK. It is also checked if it is a backdoor command, but for dependability reasons it is executed only if the packet is correct. On the other hand, if it is a normal command that will be sent to OBC, it is the master (OBC itself) responsibility to deny its execution, for this reason the elaboration continues even if there is a RX_WRONG_CRC

condition.

The OBRF status is updated and the **payloadLen : uchar** will be equal to the **ByteCount : short** (the total length of received data) minus the data that will not be needed by the OBC, which is the FCS, PID, the flag and the addresses.

A note on addresses from/to the satellite: the OBC is not interested in the destination address (which should be of the AraMiS satellite) when there is incoming data, it is a task for the OBRF, because will automatically check if the address received matches the **AX_SAT_ADDR : char const***. In the opposite way, the sender address (identified with the ground segment) does not need to be checked by the OBC itself because the OBRF uses a **AX_DEFAULT_DEST_ADDR : char const*** or a particular **addressGround : uchar[7]** previously set by the OBC.
Code:

```
Bk1B31A2S::ax25unpack() {
subfieldID(RxBuffer, addressTo, start, ADDR);

if (addressTo == AX_SAT_ADDR) {

subfieldID(RxBuffer, addressFrom, start, ADDR);  // static vect
subfieldID(RxBuffer, auxBuff, start, CONTROL);
ns = (auxBuff[0] >> 1) && 0x07;
nr = (auxBuff[0] >> 5);
(*start)++; //PID trashing

if (subfieldID(RxBuffer, auxBuff, start, DATA)) { //crc updated
RxStatus = RX_WRONG_CRC; // something bad happened, wrong data in rxbuffer
}
else if (!checkCRC(crc)) {
RxStatus = RX_WRONG_CRC; // check not passed, writing data in rxbuffer
}

if ((getCommandCode(auxBuff) == CMD_BACKDOOR)&&(RxStatus != RX_WRONG_CRC)){
executeBackdoor(auxBuff);
RxStatus = RX_IDLE;
return;
}

if (RxStatus != RX_WRONG_CRC) {
RxStatus = RX_OK;
}
updateStatus(RxStatus, paStatus);
payloadLen = ByteCount - 4/*FCS, PID, final FLAG*/ - 14 /*addresses from/to*/;
// In cmd interpret all data is put in 1B45 buffer
CC1020InitRX(baud, freq, modulation, paPower);
}
}
```

6.4.4 getCommandCode()

Used to return the type of command from the AX.25 packet. It is used to check if the command should be interpreted by the OBRF, as described in section 6.4.3. The command is related to the OSI Layer 3, therefore is contained inside the received RF *frame(destAddr, sourceAddr, N(R), N(S), info, crc)* at **info** field. According to AraMiS protocol, it is designed to retrieve 16-bit wide commands.

```
 Bk1B31A2W_OBRF_437MHz::t_OBRF_DEF_COMMAND_CODES Bk1B31A2S::getCommandCode(uchar* buffer) {
return ((t_OBRF_DEF_COMMAND_CODES) (auxBuff[0] | auxBuff[1]<<8))
}
```

6.4.5 executeBackdoor()

Executes the backdoor command from the referred vector. Simply apply the bit of a vector location to the assigned digital ports.

```
 Bk1B31A2S::executeBackdoor(uchar* buffer) {
SLOT_C::D0.write(buffer[2]&0x1);
SLOT_C::D1.write((buffer[2]>>1)&0x1);
SLOT_C::D2.write((buffer[2]>>2)&0x1);
SLOT_C::D3.write((buffer[2]>>3)&0x1);
SLOT_C::D4.write((buffer[2]>>4)&0x1);
SLOT_C::D5.write((buffer[2]>>5)&0x1);
SLOT_C::D8.write((buffer[2]>>6)&0x1);
}
```

6.4.6 subfieldID()

This is the description of the implementation of *subfieldID()* used in section 6.4.3. This function is mode-driven and its purpose is to identify and store separately all kind of data present in the AX.25 protocol frame. This frame is stored in a main **buffer** while the **subBuff** buffer is used to store temporarily the searched parameters. A word synchronization is assumed to be present. The **subBuff** vector will contain always only the last data of the last *mode* used. Every subsequent call will overwrite the addressed buffer subBuff from the beginning, while the main **buffer** continues from the last position pointed from **start**.

If **mode** = ADDR, 7 bytes are copied from buffer to subBuff according to the AX.25 address format, start reading from the **start** pointer, which is used for indexing the main **buffer**. As required by the AX.25, the last bit of the field of packet tells if that field is finished. That bit is returned by the function to allow the caller the decision of continue reading or not. Â If **mode**

= CONTROL is copied 1 byte because it should be the control byte of the AX.25, from **buffer** to **subBuff**.

If **mode** = DATA, all the bytes until the end of the main buffer are stored in **subBuff**. The CRC (FCS of AX.25) is stored in the **crc : ushort** variable, reversed as described in AX.25 protocol. Since in this mode the data is retrieved until a final AX_FLAG is found in the main buffer, a length control is implemented to prevent loops in case if flag missing, signalling an error, returning a true boolean value. The final retrieved content is purged from the flag and FCS. This function can be improved to achieve compatibility with other standards out of the AX.25, by adding others modes of type t_ID_MODE.

Code:

```
bool Bk1B31A2S::subfieldID(char* buffer, char* subBuff, short& start,
 Bk1B31A2W_OBRF_437MHz::t_ID_MODE mode) {
switch (mode) {
case ADDR:
memcpy(subBuff, buffer, 7*sizeof(char));
bool _keepGoing = (buffer[(start) + 6] & 0x1); // 1LSB copied
 of the last byte SSID, bool conversion
start += 7; //next subfield
return (_keepGoing);
break;

case CONTROL:
subBuff[0] = buffer[start];
_keepGoing = buffer[start]; //1lsb bitwise
start++;
return(_keepGoing);
break;

case DATA:
short i = start;
short k = 0;
while (buffer[i]!=AX_FLAG && i < 255){
subBuff[k++] = buffer[i++];
}
if (i>=255)
return 1;
subBuff[k-3] = "\0"; // trunc out the FCS and flag, not needed here
start = start + i-3; // final point of the buffer, with no FCS and no flag
crc = (subBuff[k-2]<<8)&0xFF00;
crc |=subBuff[k-1]&0x00FF; // saves the crc, reverse order, see use case
ax.25 protocol
break;
}
return 0;
}
```

6.4.7 Beacon packing

When looping, the *main()* polls the **SendBeacon** variable. This is set accordingly to the sequence diagram in figure 6.17. In this sequence diagram is presented the algorithm of a beacon preparation procedure. Are shown two types of beacons: the OBC beacon (in steps from 6.2 to 6.4) and RF beacon (in steps from 7 to 11). The former has the content completely transparent to the OBRF and it is handled by the On-Board Computer, happens when it is not found any command from ground after a predefined and mission dependent N attempts. What it is interested to the On-Board Radio Frequency system is the RF beacon. This is activated when no command is received from OBC (the kind of commands is defined in the RF Beacon section in chapter 4), letting a control variable to increment without any reset by a mission dependent K times. When reached a certain value, this variable triggers the auto-generation of the content to be transmitted, as described better here below, therefore the OBRF must support the OSI Layer 3 for this use case.

As a consequence, in order to keep trace of the OBC's calls, the OBRF resets a variable every time a request form the OBC takes place and increment it at every system tick of 65ms. When the value **BEACON_TIMEOUT** is reached, the SendBeacon is set and the beacon packing starts. Then it is initialized the transmission procedure like every other normal transmission, described later. The beacon use case described in section 4.2.11 is the guideline for the *BeaconPack()* method.

In figure 6.18 is simply prepared a generic buffer, called **beaconBuff : uchar[BUFFLEN]**, that will be copied in **TxBuffer : uchar[BUFFLEN]** when packing the AX.25 data, as described later, when calling the *ax25pack()*. The first step of the *beaconPack()* is therefore the assignment of the command code RF_BEACON that will be read by the receiving station, which is an OSI Layer 3 command type. Then all the values are copied in the beacon buffer following the order defined in the use case RF Beacon. After this preparation, is called the initialization of the transmitter (described in section 6.4.22) and only there is prepared the TxBuffer.

6.4.8 beaconPack()

This method implements the algorithm in section 6.4.7. According to the use case RF Beacon, this method generates the beacon data and put it in the **beaconBuff : uchar[BUFFLEN]**. This method is visually described in figure 6.18 and its timing is handled by a system tick provided by the *isr_timerA1()*.

Firstly is generated the cose **CommandCode : t_OBRF_DEF_COMMAND_CODES** = RF_BEACON and then is put in the **beaconBuff : uchar[BUFFLEN]**. The same is for the other parameters described by use case. A note on the two loop cycles: since the **beaconBuff :**

uchar[**BUFFLEN**] has 8-bit locations, the vectors copied inside that are 16-bit wide are split in two by checking if the index is odd or even, providing the capability to recognize if the previous copy was the most significant byte or not of the 16-bit vectors, and read the half word accordingly. Code:

```
Bk1B31A2S::beaconPack() {
CommandCode = RF_BEACON;
beaconBuff[0] = (byte) (CommandCode & 0xFF);
beaconBuff[1] = (byte) ((CommandCode>>8) & 0xFF);
beaconBuff[2] = LENGTH_HOUSEKEEPING;

byte index = 0;

for (index = 0; index < 2*LENGTH_HOUSEKEEPING; index++){
if ((index%2)==0){
beaconBuff[3 + index] = (byte) (housekeeping[index/2] & 0xFF);
}
else {
beaconBuff[3 + index] = (byte) ((housekeeping[index/2]>>8) & 0xFF);
}
}

beaconBuff[3+2*LENGTH_HOUSEKEEPING] = LENGTH_STATUS;

for (index = 0; index < 2*LENGTH_STATUS; index++){
if ((index%2)==0){
beaconBuff[4 + 2*LENGTH_HOUSEKEEPING + index] = (byte)
 (statusRegister[index/2] & 0xFF);
}
else {
beaconBuff[4 + 2*LENGTH_HOUSEKEEPING + index] = (byte)
 ((statusRegister[index/2]>>8) & 0xFF);
}
}
}
```

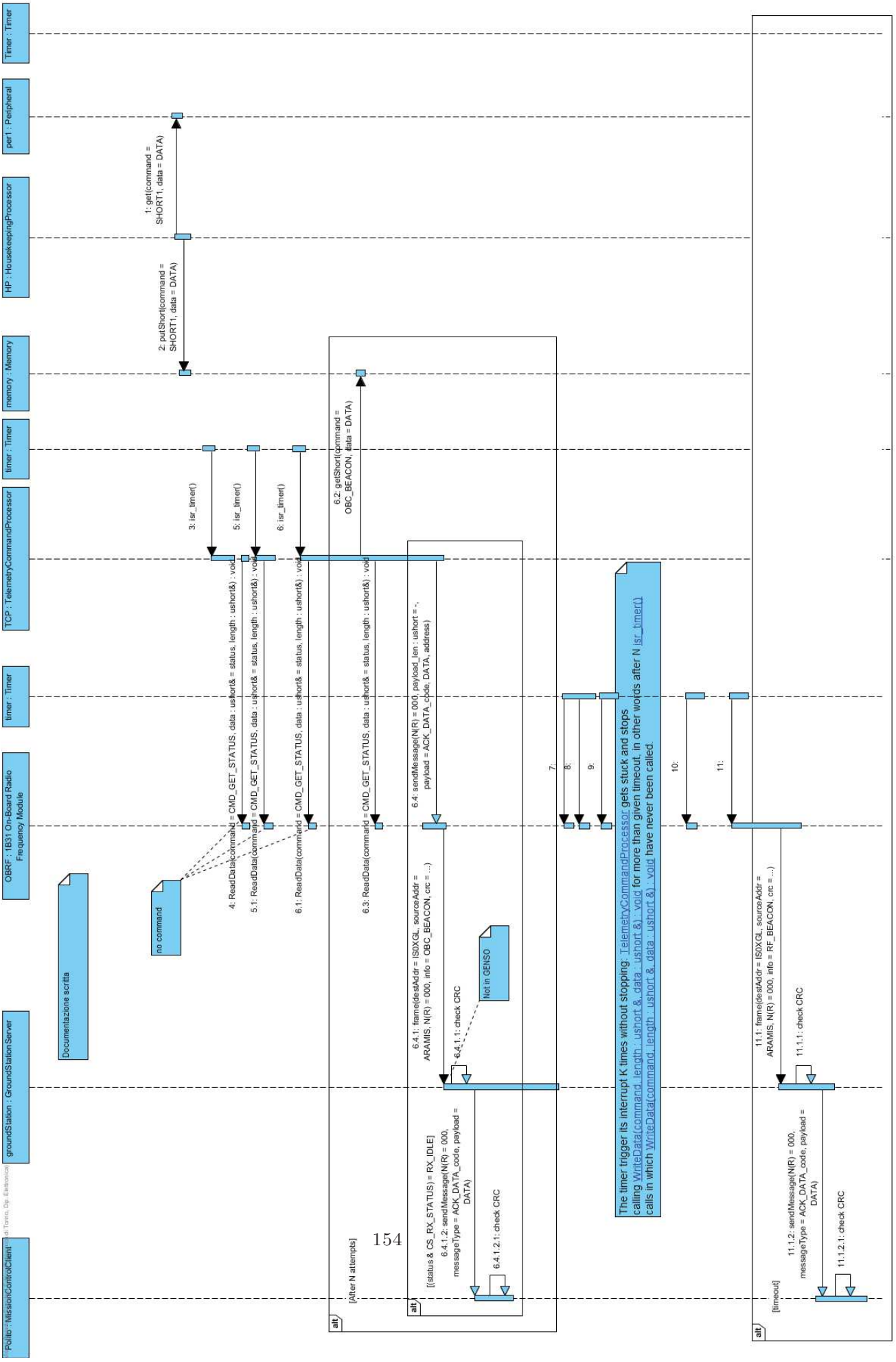

Figure 6.17. Sequence diagram of the beacon system organization

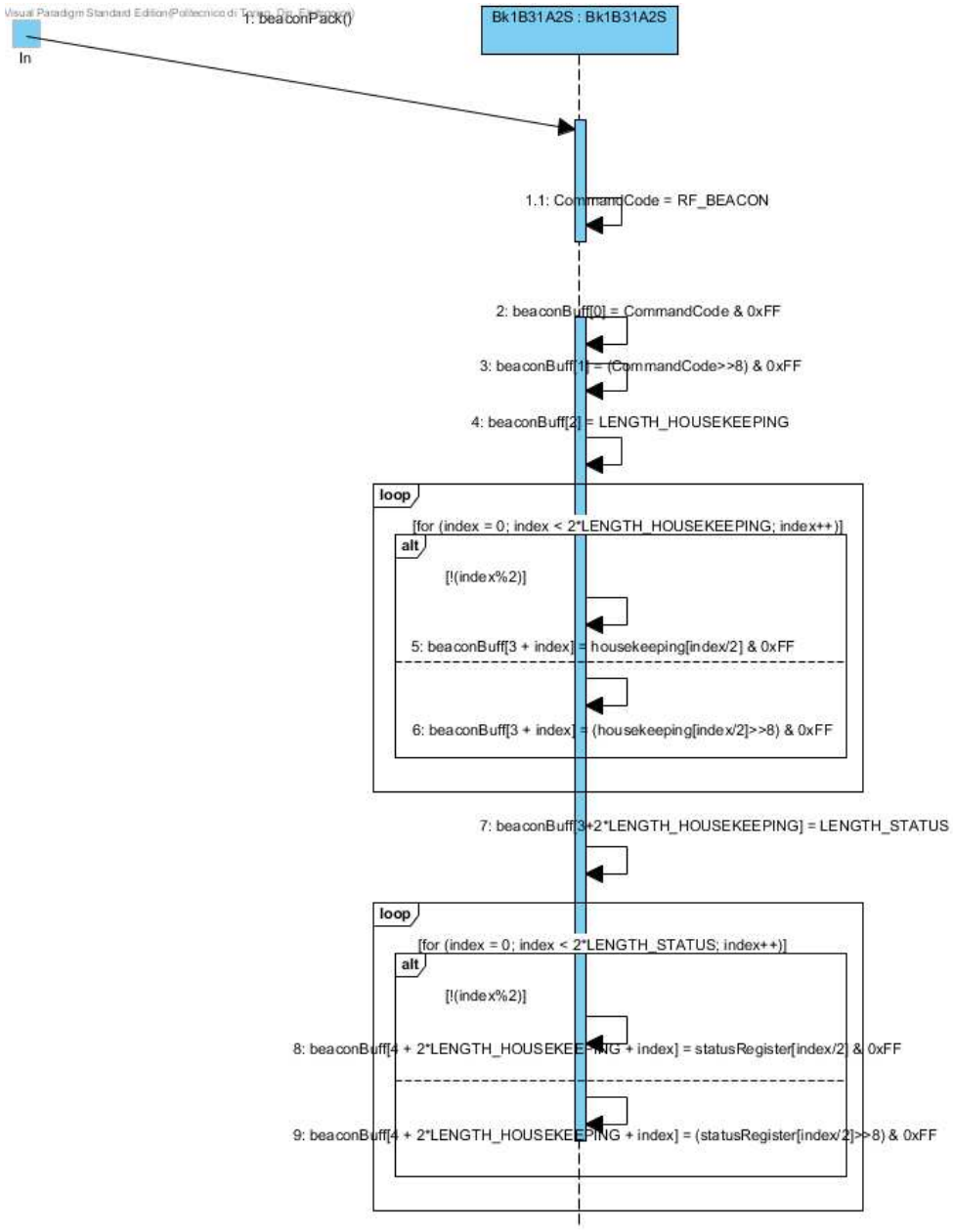

Figure 6.18. Sequence diagram of the beacon preparation

6.4.9 OBRF status and configuration updater concepts

The configuration and status registers of the OBRF are system registers that are containing a defined informations, devised by use cases in section 4.2.8. Here are developed few functions in order to update the **statusRegister : CS_REDUNDANCY** and the **configRegister : CS_REDUNDANCY**. Moreover, the configRegister can be also modified to keep coherence with *Bk1B31A2S*'s global variables. After every modification of the **configRegister**, the OBRF will be reset to the RX mode with the new configurations.

The update is performed by the *updateStatus(rxstatus : t_RX_STATUS, pa : byte)* for the statusRegister and *updateConfig(baudr : ulong, freq : AllowedFrequencies, mod : t_modulation, txpower : byte)* for the configRegister. When called, these methods will put the parameters in the respective register's position, according to the defined use cases in section 4.2.8. These functions are moving the information from a global variable, passed as a parameter in order to achieve more flexibility, to the appropriate register. It is assumed that the value of the variable reflects the actual setting. Therefore, with any modification which touch the register's value, a variable must be updated accordingly to what is inside the register using these methods. In the same way, if a variable is modified, with the appropriate methods the registers must be updated.

According to use cases and the possible OBC's commands, the **configRegister** can be modified by the OBC. When any modification takes place (by checking the last received command), must be called the *writeConfig(baudr : ulong *, freq : AllowedFrequencies *, mod : t_modulation *, txpower : ushort *)* in order to update the system variables and apply their settings to the system. This function takes the data from the register and store it to the addressed parameters. The sequence which actually implements this is shown in figure 6.19 and if followed, a system coherence is guaranteed. Are shown the steps to follow when:

- OBC modify the register (STEP 3)
- The OBRF itself need to modify configuration in registers (STEPS 1-2)
- The OBRF itself need to modify status in registers (STEP 4)

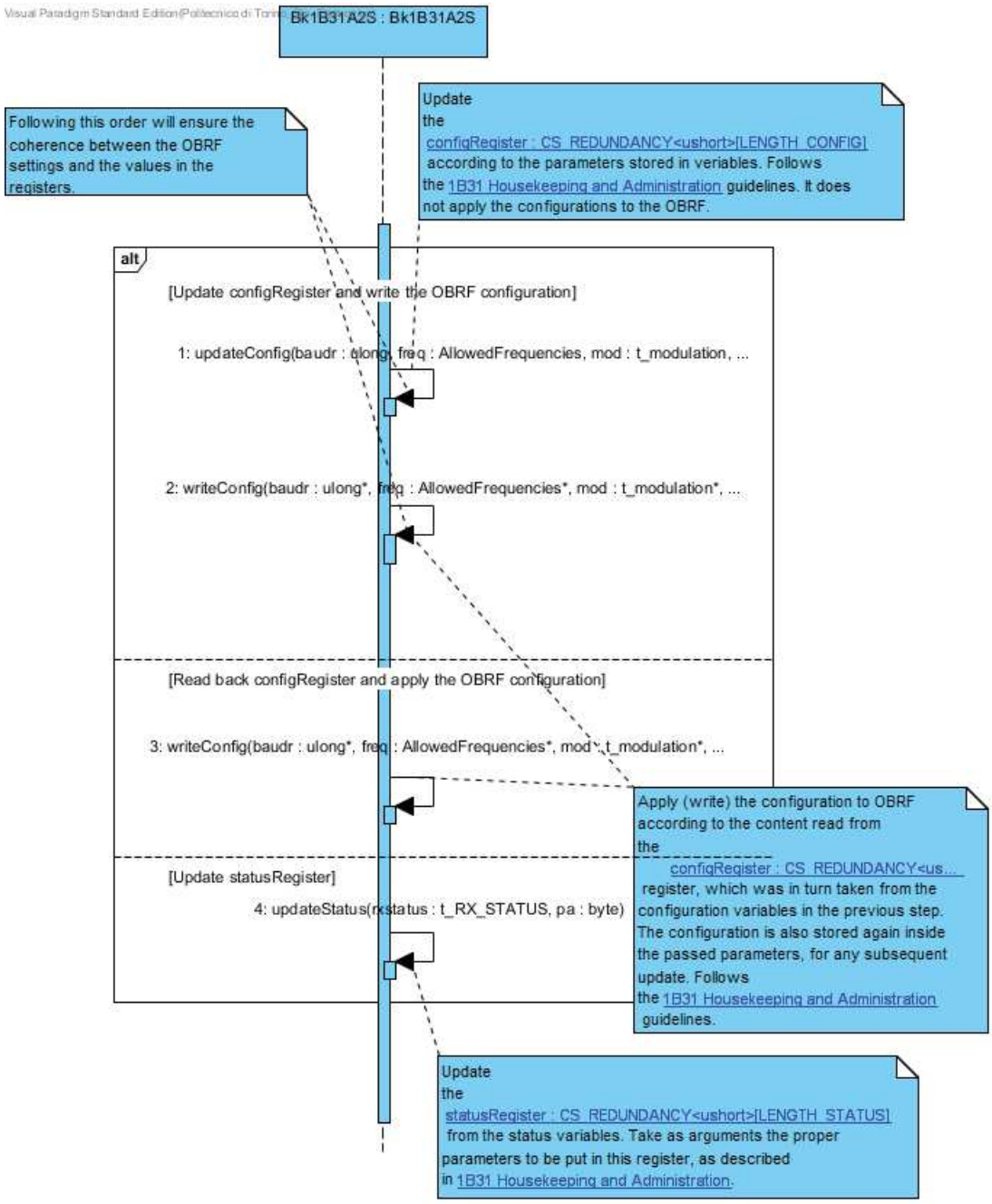

Figure 6.19. The management of status and configuration updating methods

6.4.10 updateStatus()

It implements the algorithms in section 6.4.9. Updates the **statusRegister : CS_REDUNDANCY [LENGTH_STATUS]** register from the passed parameters. These parameters are corresponding to use cases in diagram in figure 4.5. How to use this function is described in figure 6.19. Should be called after every modification of a variable in which its value is present in **statusRegister : CS_REDUNDANCY [LENGTH_STATUS]**.

Code:

```
Bk1B31A2S::updateStatus(Bk1B31A2W_OBRF_437MHz::t_RX_STATUS rxstatus,
byte pa) {
HK::statusRegister[1] = ((ushort)(rxstatus & MASK_CS_RX_STATUS) |
 (ushort)(pa & MASK_CS_PA_STATUS));
}
```

6.4.11 updateConfig()

It implements the algorithms in section 6.4.9. Updates the **configRegister : CS_REDUNDANCY [LENGTH_CONFIG]** register from the passed parameters, and do not modifies them. The parameters are corresponding to use cases in diagram in figure 4.5. Should be called after every modification of a variable in which its value is present in **configRegister : CS_REDUNDANCY [LENGTH_CONFIG]**. It will NOT updates the OBRF settings, use instead *writeConfig(baudr : ulong *, freq : AllowedFrequencies *, mod : t_modulation *, txpower : ushort *)*. How to use this function is described in figure 6.19.

Code:

```
Bk1B31A2S::updateConfig(ulong baudr, Use_Cases::AllowedFrequencies freq,
 Bk1B31A2W_OBRF_437MHz::t_modulation mod, byte txpower) {
HK::configRegister[1] = 0;
switch (baudr) {
case 2400:
HK::configRegister[1] |= (ushort)(0 & MASK_CS_BAUDRATE);
break;

case 4800:
HK::configRegister[1] |= (ushort)(1 & MASK_CS_BAUDRATE);
break;

case 9600:
HK::configRegister[1] |= (ushort)(2 & MASK_CS_BAUDRATE);
break;

case 19200:
HK::configRegister[1] |= (ushort)(3 & MASK_CS_BAUDRATE);
```

```
break;

case 38400:
HK::configRegister[1] |= (ushort)(4 & MASK_CS_BAUDRATE);
break;

case 76800:
HK::configRegister[1] |= (ushort)(5 & MASK_CS_BAUDRATE);
break;

case 153600:
HK::configRegister[1] |= (ushort)(6 & MASK_CS_BAUDRATE);
break;

default:
HK::configRegister[1] |= (ushort)(7 & MASK_CS_BAUDRATE);
break;
}

switch (freq) {
case carrierFreq.FREQ1:
HK::configRegister[1] |= (ushort)(0 & MASK_CS_FREQ);
break;

case carrierFreq.FREQ2:
HK::configRegister[1] |= (ushort)(1 & MASK_CS_FREQ);
break;

case carrierFreq.FREQ3:
HK::configRegister[1] |= (ushort)(2 & MASK_CS_FREQ);
break;

case carrierFreq.FREQ4:
HK::configRegister[1] |= (ushort)(3 & MASK_CS_FREQ);
break;

default:
HK::configRegister[1] |= (ushort)(0 & MASK_CS_FREQ);
break;
}

switch (mod) {
case modulation.FSK:
HK::configRegister[1] |= (ushort)(0 & MASK_CS_MODULATION);
break;

case modulation.GFSK:
HK::configRegister[1] |= (ushort)(1 & MASK_CS_MODULATION);
break;
```

```
default:
HK::configRegister[1] |= (ushort)(1 & MASK_CS_MODULATION);
break;
}
HK::configRegister[2] = (ushort) (txpower & MASK_CS_TX_POWER);
}
```

6.4.12 writeConfig()

It implements the algorithms in section 6.4.9. Updates the OBRF configuration from the content of the **configRegister : CS_REDUNDANCY [LENGTH_CONFIG]**, so it will read it only, updating the system variables which are buffered in this vector. The updated variables are described in use cases in figure 4.5. This function take as argument the parameters and modifies them accordingly. How to use this function is described in figure 6.19.

Code:

```
Bk1B31A2S::writeConfig(ulong* baudr, Use_Cases::AllowedFrequencies* freq,
 Bk1B31A2W_OBRF_437MHz::t_modulation* mod, ushort* txpower) {

switch (HK::configRegister[1] & MASK_CS_BAUDRATE) {
case 0:
*baudr = 2400;
break;
case 1:
*baudr = 4800;
break;
case 2:
*baudr = 9600;
break;
case 3:
*baudr = 19200;
break;
case 4:
*baudr = 38400;
break;
case 5:
*baudr = 76800;
break;
case 6:
*baudr = 153600;
break;
default:
*baudr = 2400;
break;
}
```

```
switch (HK::configRegister[1] & MASK_CS_FREQ) {
case 0:
*freq = FREQ1;
break;
case 1:
*freq = FREQ2;
break;
case 2:
*freq = FREQ3;
break;
case 3:
*freq = FREQ4;
break;
default:
*freq = FREQ1;
break;
}
switch (HK::configRegister[1] & MASK_CS_MODULATION) {
case 0:
*mod = FSK;
break;
case 1:
*mod = GFSK;
break;
default:
*mod = GFSK;
break;

}
// retrieve
*txpower = (uchar)(HK::configRegister[2] & MASK_CS_TX_POWER);
//apply
transceiver.SetReg(CC1020_PA_POWER, *txpower);
// configure
CC1020InitRX(*baudr, *freq, *mod, *txpower);
}
```

6.4.13 Initialization of radio-frequency reception mode

As shown in figure 6.5, the Tile will start automatically in RX mode. Here is therefore provided a description of this initialization procedure and methods used. This starts with the update of **configRegister : CS_REDUNDANCY [LENGTH_CONFIG]** at step 10.1.3, which act as a report of the configuration of the tile that needs to be updated every time the status is modified. Then the system will start the RX configuration procedure, which starts with sequence diagram in figure 6.20.

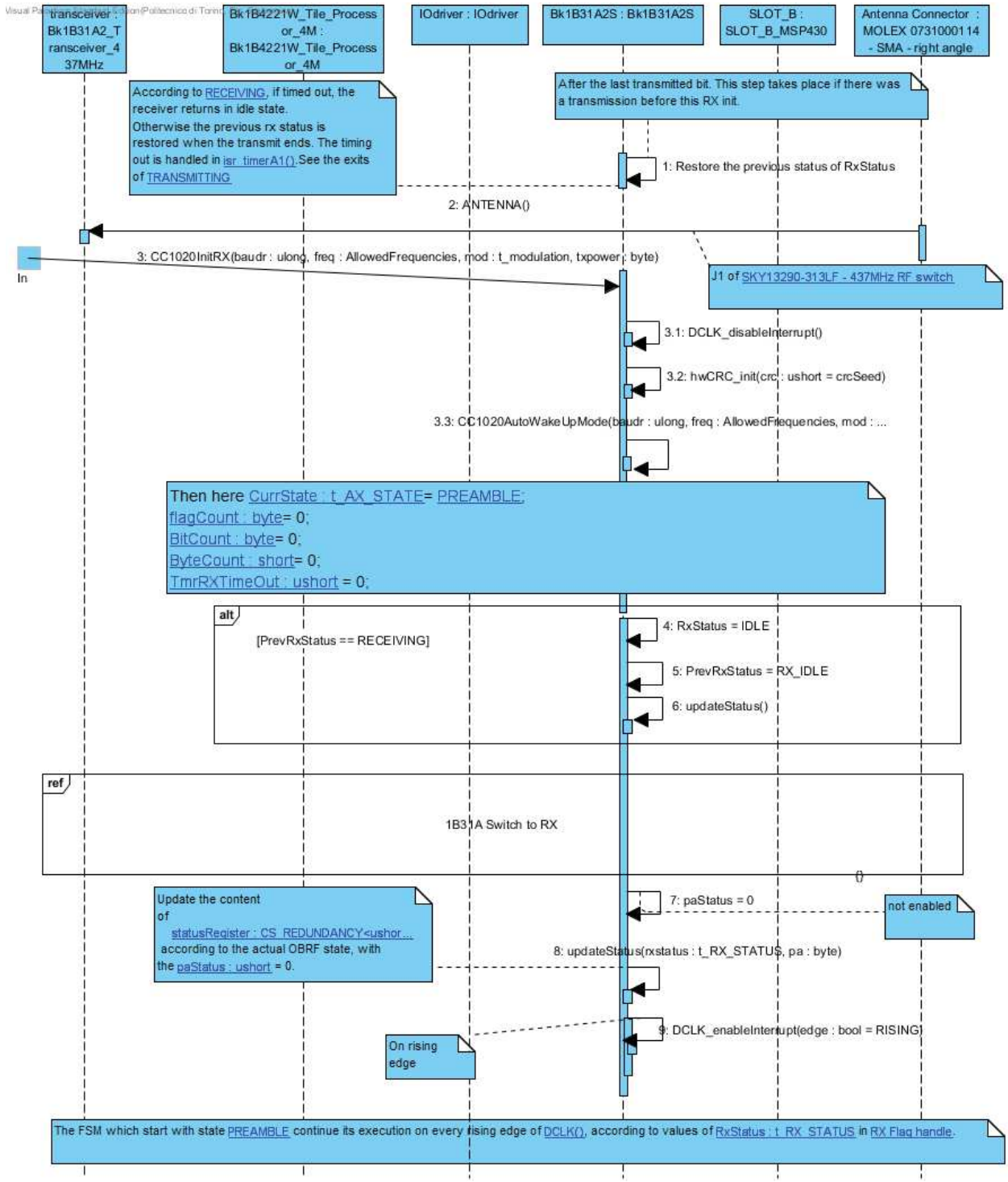

Figure 6.20. The sequence for the transceiver's configuration in RX mode

In step 2 of this diagram is highlighted that the transceiver is connected to the Antenna Connector through an RF switch, described in chapter 5, at pin J1. This switch will put the antenna in communication with the RX pin of the transceiver, which is connected to the J2 of the switch. Then from step 3 basically it describes the structure of the *CC1020InitRX(baud, freq, mod)*. In this function firstly all the interrupts related to the transceiver are disabled and it is initialized the CRC calculation hardware. Then is called the driver function *CC1020AutoWakeUpMode(baud, freq, mod)*, which actually configure the transceiver in a particular mode called Auto WakeUp mode, described better in section 6.3. This mode puts the transceiver in power down and when toggling the SPI bus slave select pin called PSEL, will initialize it automatically in RX mode, searching for an RSSI signal and return in power down if no carrier is detected. The *baud* is the desired **baudrate** chosen among the available ones as shown in use case in figure 4.5 and described from section 4.2.8. It is provided also the available carrier frequency *freq* to be search and modulation *mod* for the incoming RF symbols.

Then before the step 4 is shown what variables are prepared for the RX part, needed for keeping the receiving bus aligned with the transceiver's data. All the checks and assignment about the **RxStatus** and **PrevRxStatus** (steps 4-6) are devised from the FSM which describes the RX mode, shown in figure 6.26, from section 6.4.24. The switch to RX is important, because the switch and power amplifier are needing a proper order of activation. This order is shown in figure 6.21, which is important to avoid dangerous RF reflections, connecting the RX pin of the transceiver through RF switch (path J1 -> J2). Since the power amplifier is now disabled, it is updated the status register accordingly in step 8. The last step calls the *DCLK_enableInterrupt(RISING)*, to trigger the interrupt on the transceiver's clock on the rising edge, how these interrupt signals are generated from the transceiver will be described in section 6.3. The FSM in figure 6.26, which is the backbone of this initialization, will start with state PREAMBLE and continue its execution on every rising edge of DCLK pin, according to values of **RxStatus : t_RX_STATUS** in the secondary FSM called RX Flag handle, in figure 6.27.

Now are going to be described methods used in diagram in figure 6.20, except for the functions already described and the *CC1020AutoWakeUpMode()* which will be introduced in transceiver's section 6.4.43.

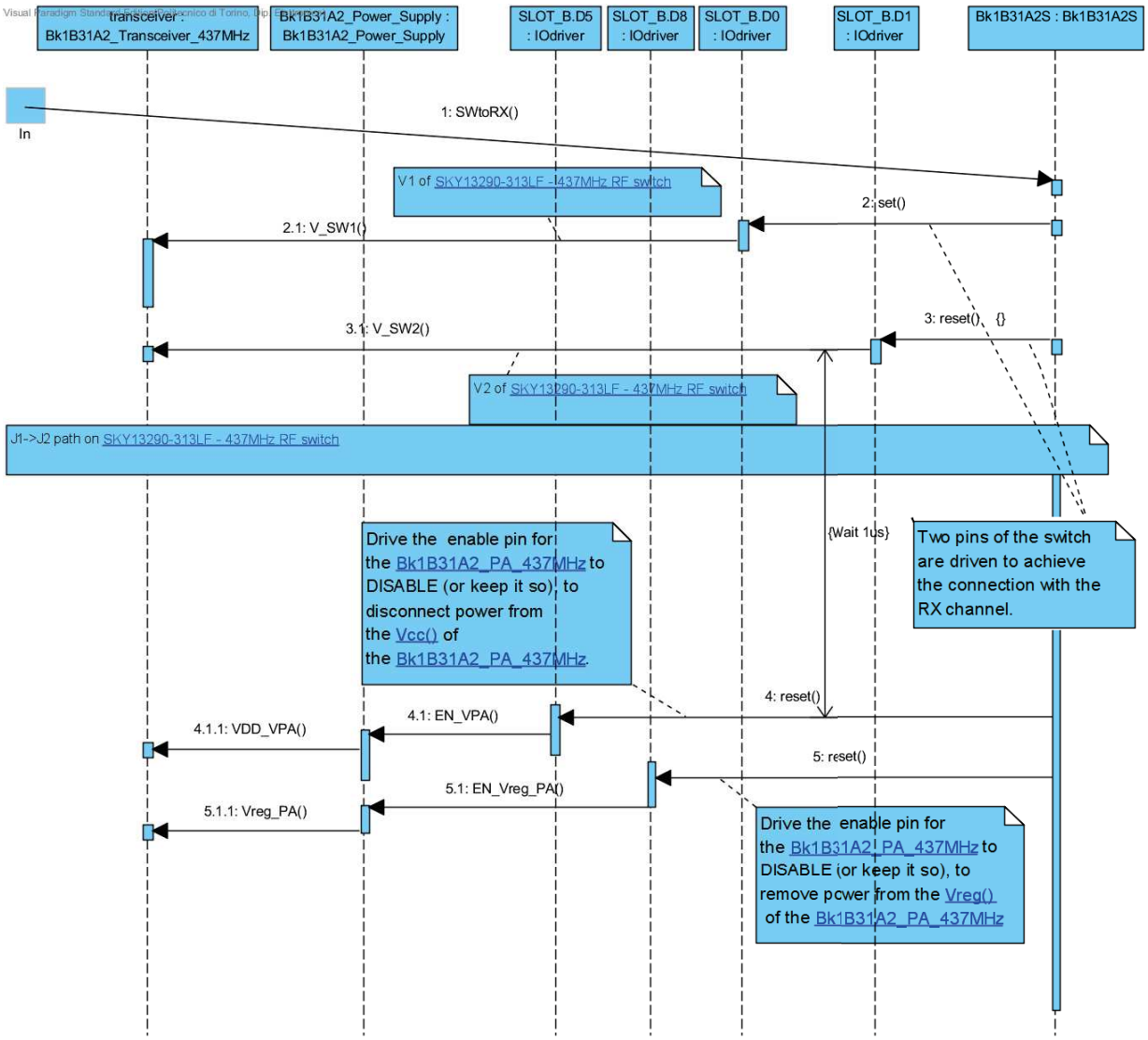

Figure 6.21. The sequence for the connection of antenna to the transceiver RX pin

6.4.14 CC1020InitRX()

It is used to implement the algorithm in section 6.4.13. Set up the transceiver to RX mode, according to the passed parameters. When in this function, the transceiver's interrupts are temporary disabled. Initialize the CRC hardware with *hwCRC_init(crc : ushort)* and initialize the Bk1B31A2_Transceiver_437MHz unit for the reception using *CC1020AutoWakeUpMode(baudr : ulong, freq : AllowedFrequencies, mod : t_modulation, txpower : ushort)*. Then the proper attributes for the **RxBuffer : uchar[BUFFLEN]** are set:

- **CurrState : t_AX_STATE** is the OBRF FSM receiving status
- **flagCount : byte** is the AX.25 flags counter
- **BitCount : byte** the counter of received bits
- **ByteCount : short** the counter of received not dummy bytes
- **TmrRXTimeOut : ushort** is reset to prevent the receiver's timeout condition

The **PrevRxStatus : t_RX_STATUS** contains the last status of the system, and is controlled to check if a useful reception was broken and therefore resets the receiver status to RX_IDLE, if necessary. Otherwise the system's status is kept unchanged.

With *SWtoRX()* the RF hardware is put correctly in the appropriate configuration. After updating the status with *updateStatus(rxstatus : t_RX_STATUS, pa : byte*), the transceiver's interrupts are enabled on RISING edge on transceiver's signal pin DCLK.
Code:

```
Bk1B31A2S::CC1020InitRX(ulong baudr, Use_Cases::AllowedFrequencies freq,
 Bk1B31A2W_OBRF_437MHz::t_modulation mod, byte txpower) {
DCLK_disableInterrupt();
hwCRC_init(crcSeed);
// transceiver init in automatic sequencing
CC1020AutoWakeUpMode(baud, freq, modulation, txpower);

// Variables init
CurrState = PREAMBLE;
FlagCount = 0;
BitCount = 0;
ByteCount = 0;
TmrRXTimeOut = 0;
if (PrevRxStatus == RECEIVING){ // reset if a reception was interrupted
RxStatus = RX_IDLE;
PrevRxStatus = RX_IDLE; // because this if() should not be executed
again if there was no tx
```

```
updateStatus(RxStatus, paStatus);
}

// init pins for RX mode
SWtoRX();
paStatus = 0;
updateStatus(RxStatus, paStatus);
DCLK_enableInterrupt(RISING); // rising
}
```

6.4.15 PAEnable()

Enable the power amplifier, according to figure 6.24.

```
 Bk1B31A2S::PAEnable() {
SLOT_B::D5.set();
SLOT_B::D8.set();
}
```

6.4.16 PADisable()

Disable the power amplifier, according to figure 6.20.

```
 Bk1B31A2S::PADisable() {
SLOT_B::D5.reset(); // PA
SLOT_B::D8.reset();
}
```

6.4.17 SWtoTX()

Switch the RF hardware in the TX mode, according to figure 6.24.

```
 Bk1B31A2S::SWtoTX() {
PAEnable();
SLOT_B::D0.reset(); // Switch
SLOT_B::D1.set();
}
```

6.4.18 SWtoRX()

Switch the RF hardware in the RX mode, according to figure 6.20

```
 Bk1B31A2S::SWtoTX() {
PAEnable();
SLOT_B::D0.reset(); // Switch
SLOT_B::D1.set();
}
```

6.4.19 DCLK_disableInterrupt() and DCLK_enableInterrupt()

Are used to implement part of the algorithm in section 6.4.13. The *DCLK_enableInterrupt(edge : bool)* enables the interrupt on DCLK() pin changes of the Bk1B31A2_Transceiver_437MHz unit. The edge parameter is RISING when interrupt occurs on RISING edge of the signal on pin. Viceversa when FALLING.

The *DCLK_disableInterrupt()* disables all the interrupt related to DCLK pin of the Bk1B31A2_ Transceiver_ 437MHz unit.

6.4.20 AX.25 Packing algorithm

In figure 6.24 at step 1.3 there is the call to the *ax25pack()*. The algorithm of that method is described here, shown in figure 6.22, with the description of its implementation. This packing algorithm consists of creating a complete AX.25 buffer to be sent, auto-generating the needed auxiliary data. Because this method is called also for the beacon, it is checked the **SendBeacon** variable, therefore supports two modes of packing: one for the OBC transmitting commands and one for the RF beacon.

After a CRC initialization, the step 1.2 or 1.3 adapts the length of the packet to the total of the Beacon length or to the one decided by the OBC data. From step 1.5 up to 1.13 are generated the addresses, the AX.25 sequence numbers and the PID byte: for RF beacon the sequence numbers are always zero because it is designed to be sent in a single frame. From step 1.15 it is copied the payload to the **TxBuffer**, which can be taken from the *MessageHandler*'s buffer or from the **beaconBuff**[**BUFFLEN**].

Every byte copied in the TxBuffer is succeeded by an updating of the CRC. This value is calculated bit reversed as well as the byte order, and it is copied in the transmission buffer taking care of keeping the data reversed (steps 1.20 and 1.21). Here there is no modification of the **RxStatus** because this packing algorithm should be called only when the OBRF is in TRANSMITTING status.

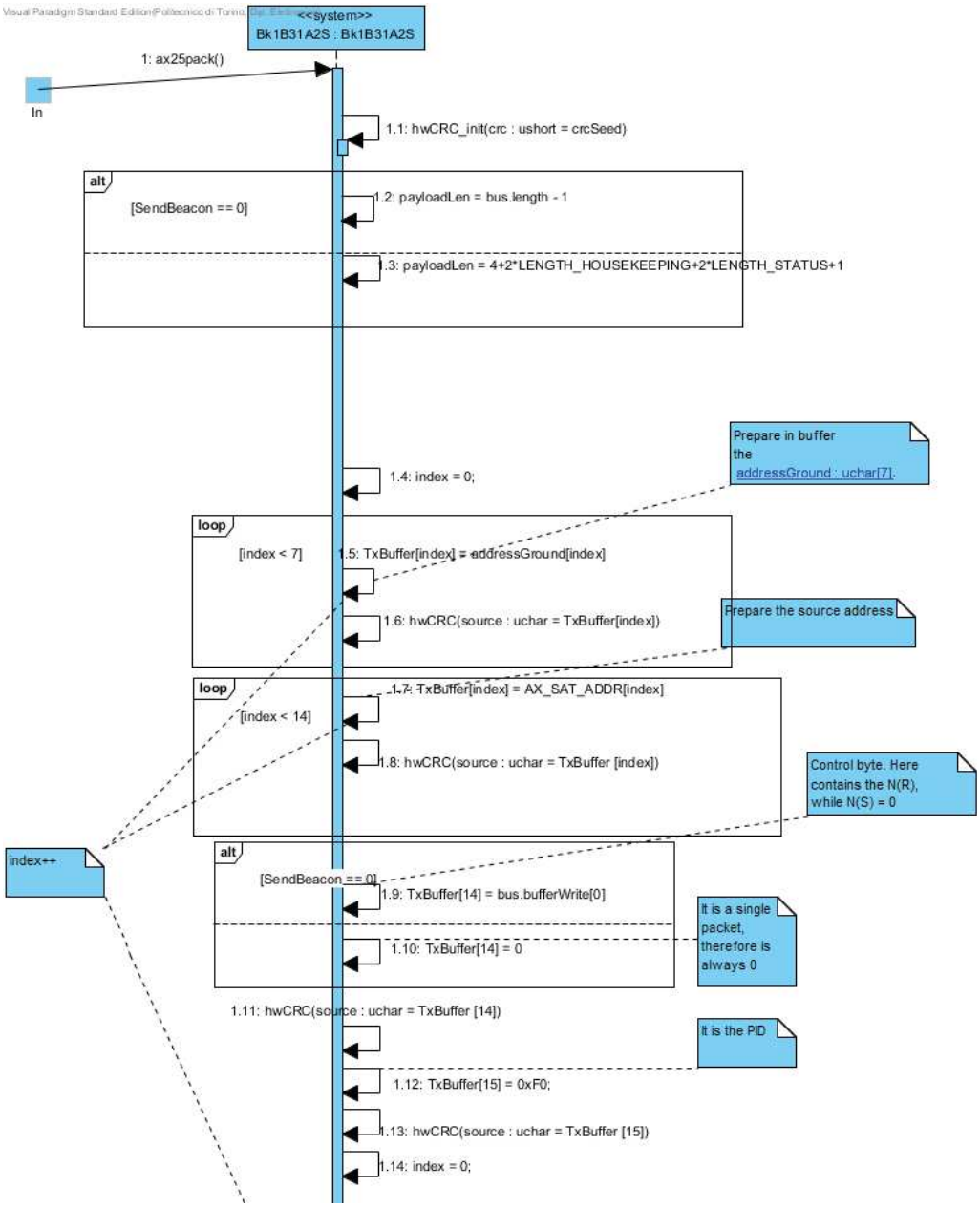

Figure 6.22. Sequence diagram of the AX.25 Packing procedure, 1/2

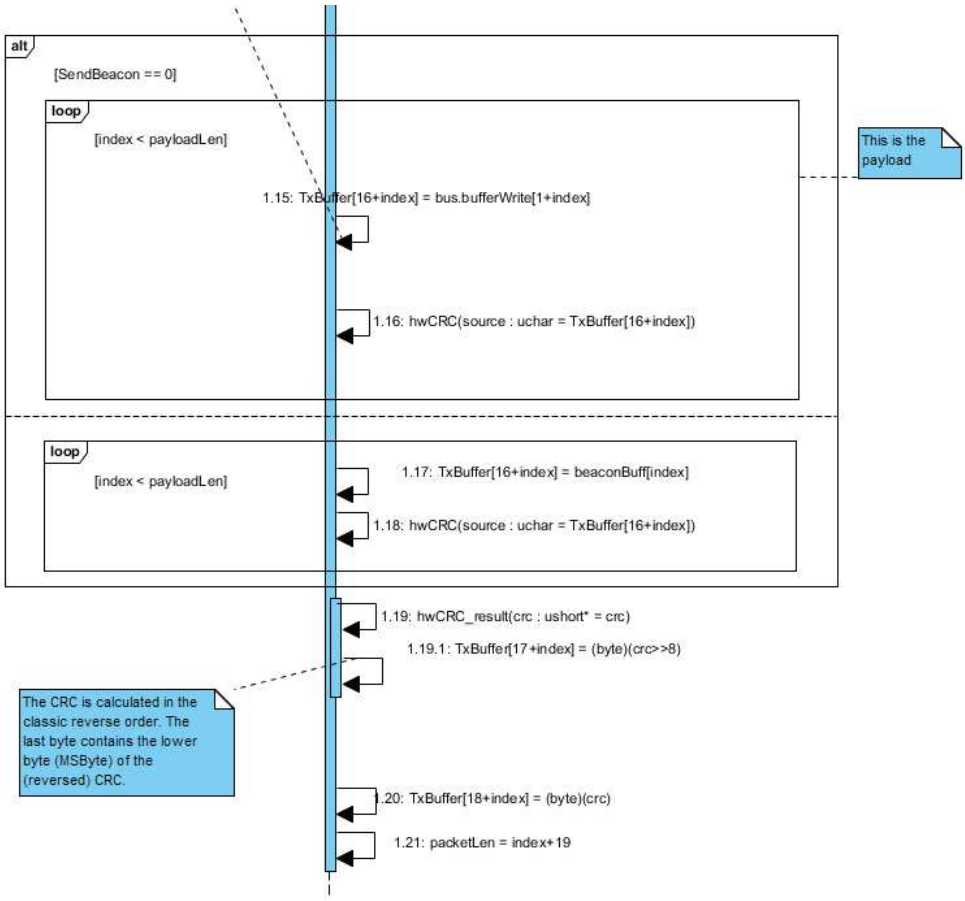

Figure 6.23. Sequence diagram of the AX.25 Packing procedure, 1/2

6.4.21 ax25pack()

It is used to implement the algorithm in section 6.4.20. This function prepare the **TxBuffer : uchar[BUFFLEN]** in which a complete AX.25 frame will be stored. It is made a distinction between the a normal packet composition (if SendBeacon : bool == false) and the beacon packet(if SendBeacon : bool == true), due to the difference of the information sources.

The TxBuffer is filled with:

- the **AX_SAT_ADDR : char const*** (bytes 0 to 6);
- **addressGround : uchar[7]** (byte 7 to 13);
- then with the rest of the AX.25 components. The payload is taken from the remaining OBC data or from **beaconBuff : uchar[BUFFLEN]**, depending if it is a normal packet or a beacon one. Then the CRC is calculated and stored as a FCS field. The **PacketLen : uchar** is updated.

Here there is no modification of the **RxStatus : t_RX_STATUS** because should be called only when it is TRANSMITTING.

Code:

```
Bk1B31A2S::ax25pack() {
byte index = 0, bufIndex = 0;

hwCRC_init(crcSeed);

if (!SendBeacon)
payloadLen = bus.length - 1;
else
payloadLen = 4+2*LENGTH_HOUSEKEEPING+2*LENGTH_STATUS+1;

while (index<7){
TxBuffer[index] = addressGround[index];
hwCRC((ushort)TxBuffer[index]);
index++;
}

while (index<14){
TxBuffer[index] = AX_SAT_ADDR[index];
hwCRC((ushort)TxBuffer[index]);
index++;
}

if (!SendBeacon)
TxBuffer[index] = bus.bufferWrite[0];
```

```
else
TxBuffer[index] = 0; //index = 14

hwCRC(TxBuffer[index]&0x00ff);

index++;

TxBuffer[index] = PID; //index = 15
hwCRC(TxBuffer[index]&0x00ff);

bufIndex++;

if (!SendBeacon){
while(bufIndex < payloadLen){
TxBuffer[index] = bus.bufferWrite[bufIndex];
hwCRC(TxBuffer[index]&0x00ff);
bufIndex++;
index++;
}
}
else {
bufIndex = 0;
while(bufIndex < payloadLen){
TxBuffer[index] = beaconBuff[bufIndex];
hwCRC(TxBuffer[index]&0x00ff);
bufIndex++;
index++;
}
}
hwCRC_result(&crc); // pass by pointer

TxBuffer[index++] = (byte)(crc>>8);
TxBuffer[index++] = (byte)(crc);

packetLen = index;
}
```

6.4.22 Initialization of radio-frequency transmission mode

Here is provided the description of the initialization procedure for the transmission mode. Here is therefore provided a description of the initialization algorithm and the methods used to implement it. This happens only after the need of a beacon (procedure in figure 6.17 and section 6.4.7) or after a transmitting command from the OBC, described later in the *intepret(command : ushort)* method. When this happens the OBRF will initialize the transceiver for the TX mode. This is described by sequence diagram in figure 6.24.

In step 2 of this diagram is highlighted that the transceiver is connected to the Antenna Connector through an RF switch, described in chapter 5, at pin J1. This switch will put the antenna in communication with the TX pin of the transceiver, which connected to the J3 of the switch.

Then from step 1 basically it describes the structure of the *CC1020InitTX(baud, freq, mod)*. In this function firstly all the interrupts related to the transceiver are disabled and the CRC system is initialized for the subsequent elaboration. Then is called the *ax25pack()* which prepares the transmitting buffer with the content received from the OBC or autogenerated (if it was a beacon). All the checks and assignment about the **RxStatus** and **PrevRxStatus** are devised from the FSM which describes the TX mode, shown in figure 6.29, in section 6.4.24.

The switch to TX is made in step 3 and is important, because the switch and power amplifier are needing a proper order of activation. This order is shown in figure 6.25, which is important to avoid dangerous RF reflections, connecting the TX pin of the transceiver through RF switch (path J1 -> J3). Then the RF hardware is configured properly and the driver function *CC1020TxMode(baud, freq, mod, txpower)*, which actually configures the transceiver in transmission mode. How this is made will be described in section 6.3. The *baud* is the desired **baudrate** chosen among the available ones as shown in figure 4.5 and described from section 4.2.8. Are also provided the available carrier frequency *freq* and modulation *mod* that can be chosen.

Since now the power amplifier is enabled, it is updated the status register accordingly, in step 5. The last step calls the *DCLK_enableInterrupt(FALLING)*, to trigger the interrupt on the transceiver's clock on the falling edge, and how the transceiver generates this interrupt is described in section 6.3. The FSM in figure 6.29 will start with state TX_PREAMBLE and continue its execution on every falling edge of DCLK pin, until the values of **RxStatus : t_RX_STATUS** in the secondary FSM (shown in figure 6.27), is kept in TRANSMITTING, or when the whole TxBuffer is sent.

Now are going to be described methods used in diagram in figure 6.24, except for functions already described and the *CC1020TxMode()*, which will be introduced in transceiver's section 6.3.

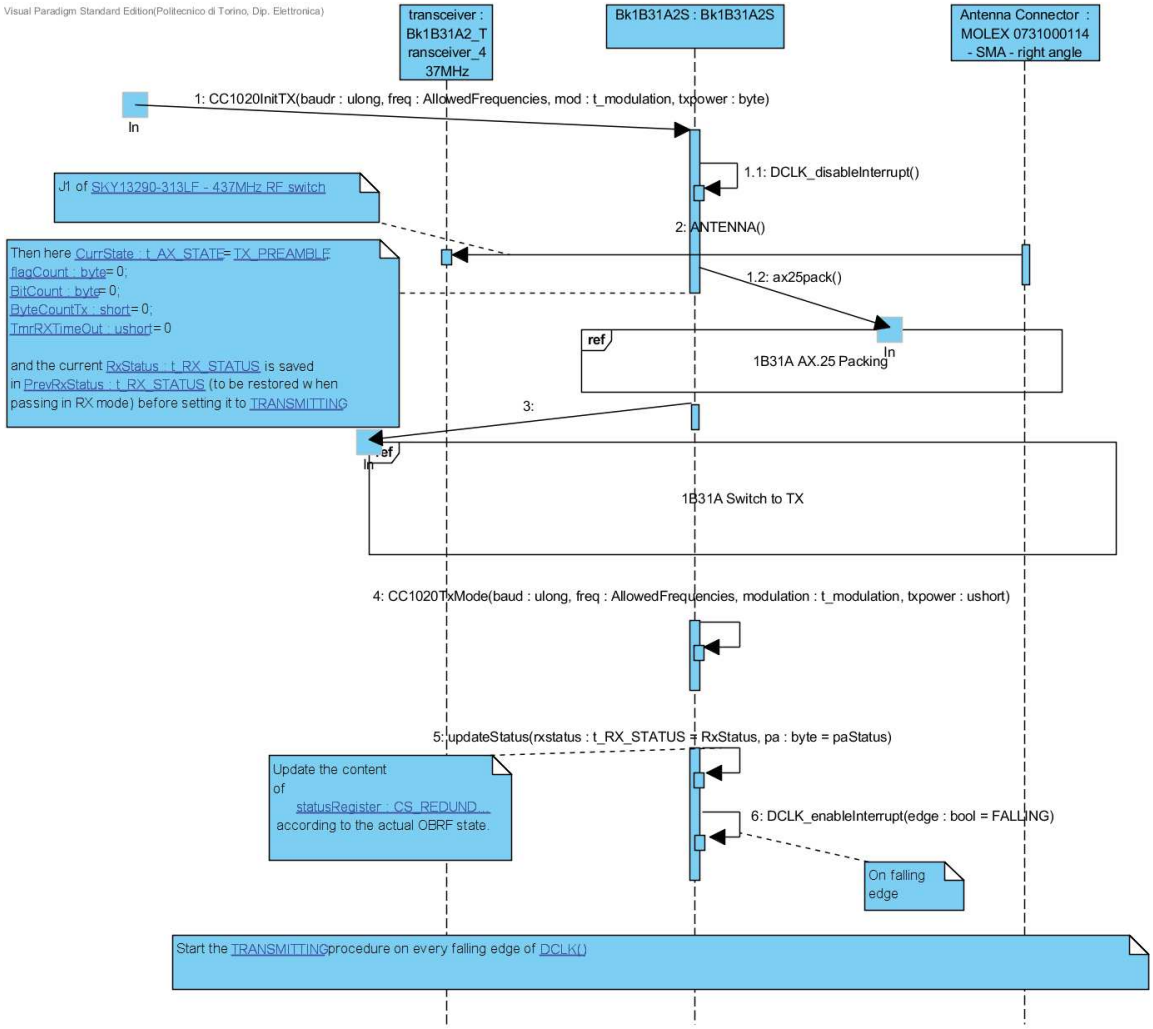

Figure 6.24. The sequence for the transceiver's configuration in TX mode

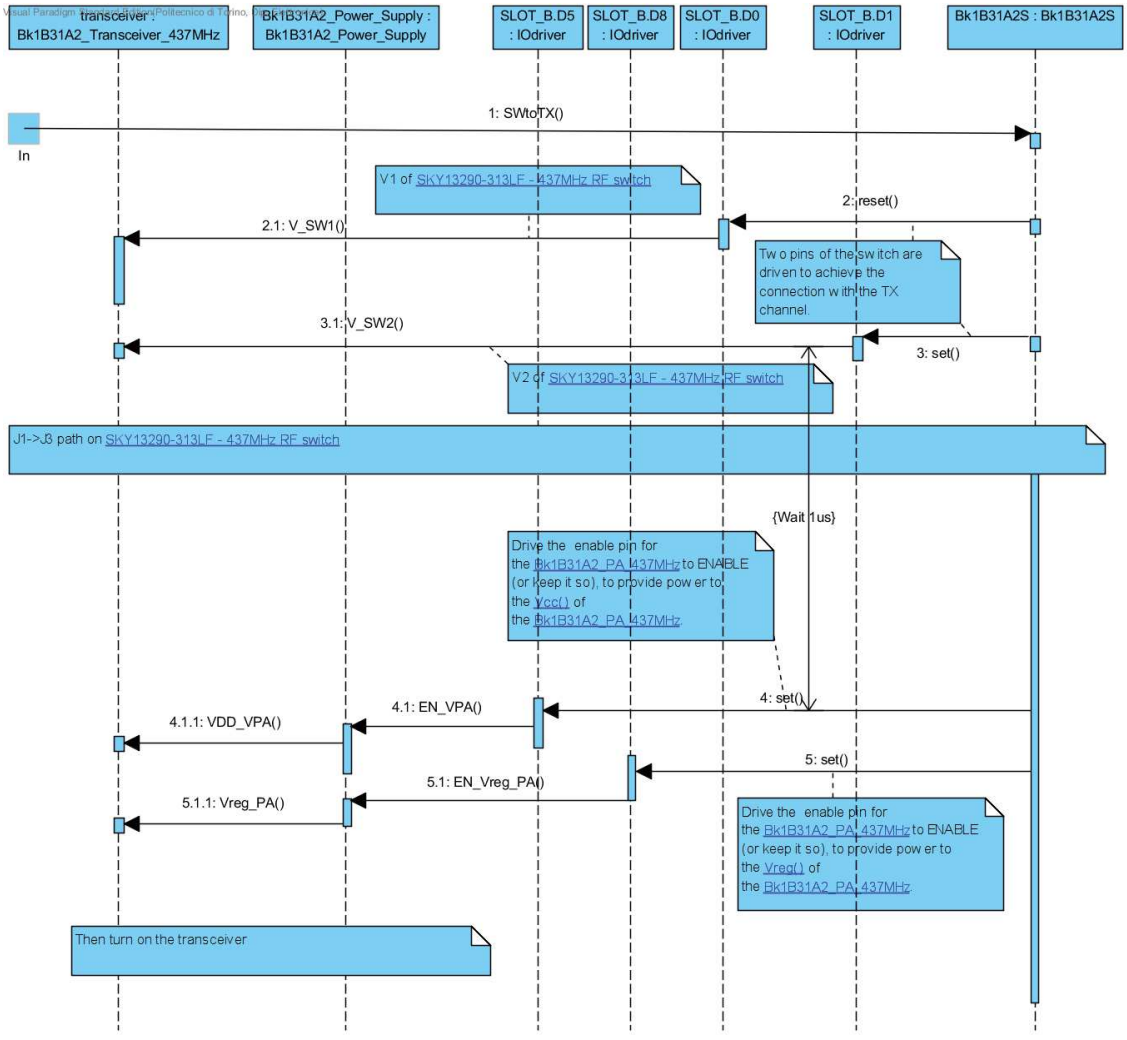

Figure 6.25. The sequence for the connection of antenna to the transceiver TX pin

6.4.23 CC1020InitTX()

It is used in the initialization sequence in section 6.4.22. Set up the transceiver to TX mode, starting from a reset instead of power down assumption, in order to prevent bad configurations that can arise with these COTS components. This is made according to the passed parameters. Calls the *ax25pack()* in order to prepare the **TxBuffer : uchar**[**BUFFLEN**]. Then are prepared the following parameters:

- **CurrState : t_AX_STATE** is the OBRF FSM initial transmitting status
- **flagCount : byte** is the AX.25 flags counter
- **BitCount : byte** the counter of transmitted bits
- **ByteCount : short** the counter of transmitted bytes
- **TmrRXTimeOut : ushort** is reset to prevent the transmitter's timeout condition

The **PrevRxStatus : t_RX_ STATUS** is used to buffer the current **RxStatus : t_ RX_ STATUS** and updating it with TRANSMITTING value. The **PrevRxStatus : t_RX_STATUS** will be used after the transmission to restore the interrupted status of the receiver FSM. With *SWtoTX()* the RF hardware is put correctly in the appropriate configuration. Now the Bk1B31A2_ Transceiver _437MHz unit is put in TX mode by calling the *CC1020TxMode(baud : ulong, freq : AllowedFrequencies, modulation : t_modulation, txpower : ushort)*. Since now the power amplifier is activated, the **paStatus : ushort** is updated accordingly and its value is written inside the status register by using the *updateStatus(rxstatus : t_RX_STATUS, pa : byte)*. The transceiver's interrupts are now enabled on FALLING edge on transceiver's signal DCLK().
Code:

```
Bk1B31A2S::CC1020InitTX(ulong baudr, Use_Cases::AllowedFrequencies freq,
 Bk1B31A2W_OBRF_437MHz::t_modulation mod, byte txpower) {
DCLK_disableInterrupt();
ax25pack();  // implement following s.d.

CurrState = TX_PREAMBLE;
FlagCount = 0;
BitCount = 0;
ByteCount = 0;
TmrRXTimeOut = 0;

PrevRxStatus = RxStatus;
RxStatus = TRANSMITTIG;
```

```
SWtoTX(); //vedi se mettere prima
CC1020TxMode(baudr, freq, mod, txpower); // with config registers

paStatus=1;
updateStatus(RxStatus, paStatus);

DCLK_enableInterrupt(FALLING);
}
```

6.4.24 Data handling of RF data

Until now it has been described the various initialization procedures and the behaviour at boot. Here is now described the algorithm on how the radio data is handled on the OBRF, both in transmission and reception, once the transceiver is correctly initialized. The classes *Bk1B31A2S* and *Bk1B31A2S_main* are developed in order to implement the algorithms herein described.

The default condition of the OBRF is the RX mode, so the description starts with this one. At every clock cycle of the transceiver a bit is brought on the DIO pin from the RF hardware. For this reason the data organization is devised in an FSM where its clock is the DCLK signal, for both RX and TX modes. At the beginning, there is no synchronization of the transceiver with the modulated signal, therefore in order to understand the FSM behaviour, is needed to start with FSM shown in figure 6.26.

At the very first call of the transceiver's interrupt, the *isr_CC1020RxData()*, we are in the RX_ PREAMBLE state, in which every bit is shifted in and it is check continuously the eventual presence of the AX_FLAG (its description is in section 3.3). If this is the case, we are in a condition in which the slicer of the transceiver is synchronized with the carrier, what is missing is the synchronization with the word. Here the SYNCW state could be activated by mistake, so there is the ERROR_CHANCE, where it is controlled if the next byte it is not a flag when it should be: if so, was an error due to the limited BER and the word searching continue. Since the transceiver's slicer need a certain amount of transitions for the synchronization (see section 6.3), it is set a fixed minimum amount of flag that must be received correctly before assuming the transmission to be reliable, stored in variable **FLAG_THR : byte const**.

When the current state is SYNCW, after every 8 bits received is checked if it is part of the flag or it is the first payload data. Because the transceiver is configured to wakeup only in presence of a carrier, and since it can be lost, it is possible to lock the FSM in this state. So it is implemented an automatic reset of the RX status using the ISR of the Timer A1. For this reason, at every call of the *isr_CC1020RxData()*, when in SYNCW or PAYLOAD, the timeout variable must be reset

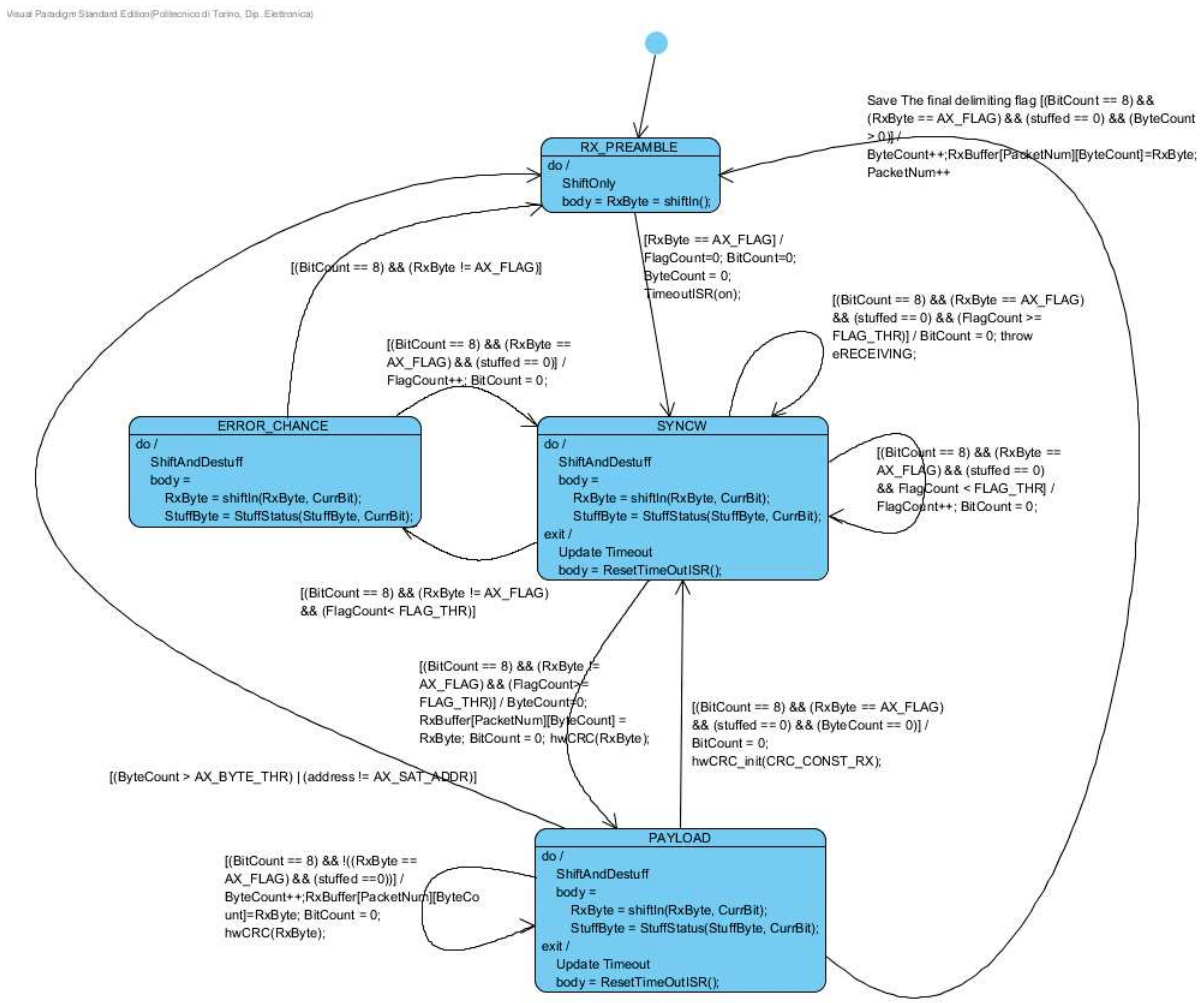

Figure 6.26. FSM for the received data

to avoid to set the default value of the RX status flag.

The subsequent state PAYLOAD is activated if the last byte is not a flag, assuming that a minimum number of flags has been received. But due to the presence of a certain amount of the BER, the transition could be a false result, therefore if a flag is detected, we may come back to the SYNCW state. If more than one byte is different from the flag is received, then the new condition is to keep saving the incoming data, while searching for a closing flag of the same type of one used during the preamble. When in PAYLOAD, is checked in run-time if the destination address match the**AX__SAT__ADDR : char const**, and if not, FSM restart from PREAMBLE state.

When in SYNCW it is possible to take the branch where, after the first data byte, there is the *throw eRECEIVING*. This is useful for a logical parallel FSM which handles the flag **RxStatus**. This is used to coordinate and synchronize all the satellite data handling. Basically, in figure 6.27, it is shown this FSM which in turn contains other 2 FSMs, called *RX Flag Handle* and *TRANSMITTING*. This layering is useful to split the complexity, which is not trivial when the dependability is a priority. When the system boot, this FSM starts with the RX Flag Handle and from there decide the next state upon the OBC's command or beacon necessity. These commands can be a standby or a transmitting request, and this FSM is needed to recognize when the data is no more consistent, because these commands are capable to interrupt the normal execution of receiving process.

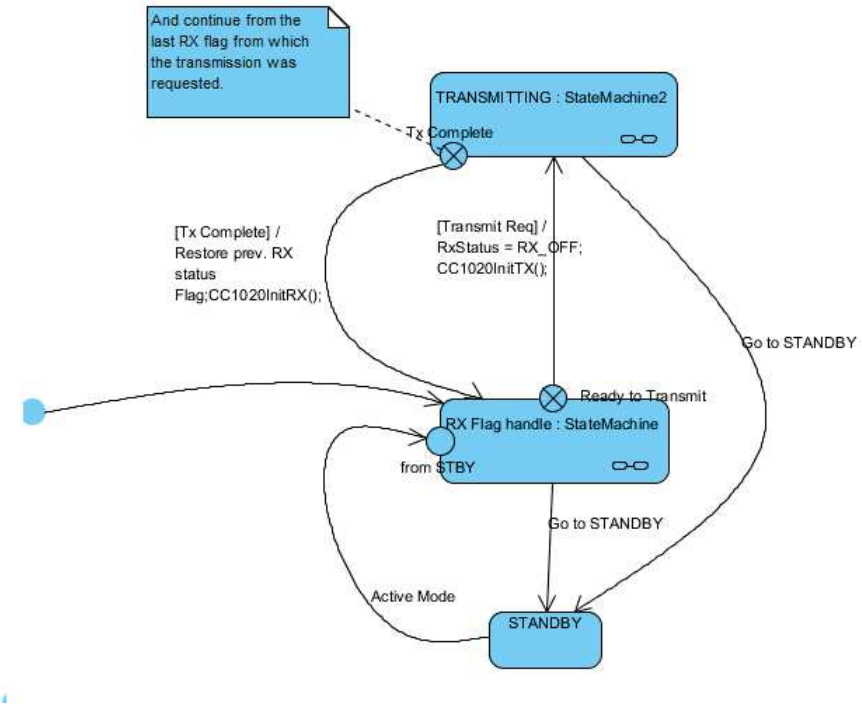

Figure 6.27. FSM for the flag handling

6.4.25 RX Flag Handle State Machine

It is a state machine used to implement the system flags handling, introduced in section 6.4.24. In this FSM are present all the possible values of the **RxStatus** flag. From RX_IDLE to RECEIVING is present the eRECEIVING on the arrow. This transition occur when in figure 6.27 is taken the arrow with the "throw eRECEIVING" from the SYNCW state. By doing so, can be recognizable

when the reception should not be interrupted because of the potentially useful data received. This FSM is the backbone of the OBRF status synchronization.

When the reception is terminated is set the RX_RAW to indicate that the data need a proper elaboration, as described in figures 6.6 and 6.15. Here will be set the RX_WRONG_CRC if the packet's CRC is wrong or viceversa if RX_OK: in any case the packet is sent to OBC, which is able to read the **RxStatus** flag from the **statusRegister** and decide if keep the packet or thrashing it away. Should be quite clear the purpose of the note, which highlight that when coming from TRANSMITTING, the FSM will continue from the last RxStatus: there is only one state which is not kept and is the RECEIVING. When here, any interruption from the receiving process will introduce data loss from the medium and a reset is needed.

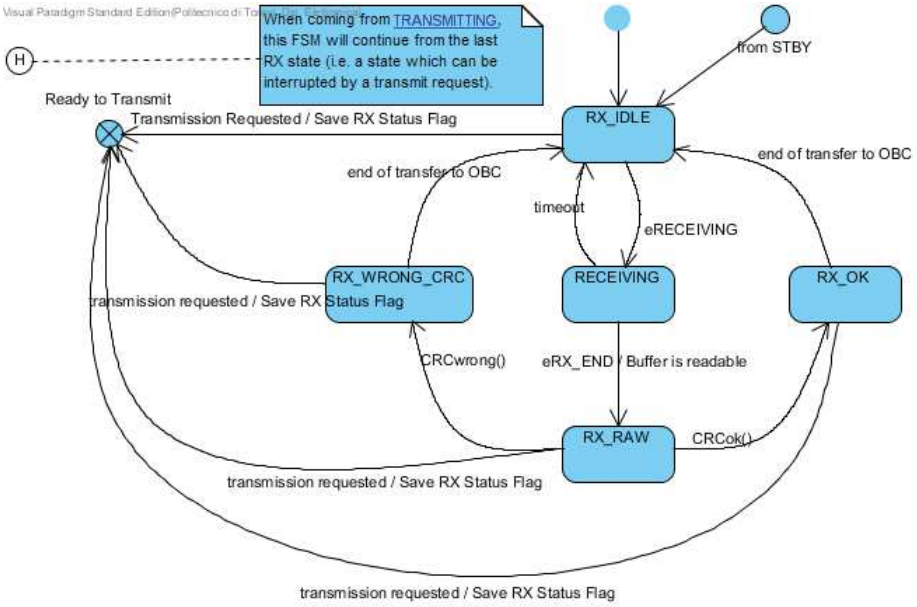

Figure 6.28. FSM for the flag handling

6.4.26 isr_CC1020RxData()

Implements the FSM of the received data sequence in section 6.4.24. Interrupt used to synchronize bits from the transceiver and when there is valid data store it in **RxBuffer : uchar[BUFFLEN]**. This function is state driven and act as an FSM which change at every rising edge of the DCLK() and acting based on the **CurrState : t_AX_STATE** and the value of DIO(). The behavior of states is described by the FSM which starts from RX_PREAMBLE.

The incoming bit is read at SLOT_A::D0 and buffered in **CurrBit : bool**. Then the FSM interpret this bit, by reading the current **CurrState : t_AX_STATE**. If **CurrState : t_AX_STATE** = PREAMBLE the CPU keep to shift in by *shiftIn(data : uchar, _bit : uchar) : byte* the bit from the transceiver whenever an **AX_FLAG : byte const** is found. If so, **CurrState : t_AX_STATE** = SYNCW, **TmrRXTimeOut : ushort** = 0 because the receiver is not blocked, and initialize to 0 all the other counting variables **BitCount : byte**, **ByteCount : short**, **flagCount : byte**.

If **CurrState : t_AX_STATE** = SYNCW is updated the **RxByte : byte** with *shiftIn(data : uchar, _bit : uchar) : byte*, along as the stuffing analysis with **StuffByte : byte** = *StuffStatus(data : uchar, _bit : uchar)*. In every condition is controlled if the **CurrBit : bool** is the 8th with **BitCount : byte** == 8. Then if **FLAG_THR : byte const** is not reached and data is different from **AX_FLAG : byte const**, there is something wrong and ERROR_CHANCE is set for the next cycle. The second condition controls if received byte is a correct **AX_FLAG : byte const** but **FLAG_THR : byte const** is not reached: the flag counter is incremented. The third condition controls if it is a valid data different from **AX_FLAG : byte const**, if the **FLAG_THR : byte const** is reached: if so **RxBuffer : uchar[BUFFLEN]** = **RxByte : byte** and the CRC is updated with *hwCRC(source : uchar)*. The next state now must be PAYLOAD. Since the data should be correct, there is an active reception and **RxStatus : t_RX_STATUS** = RECEIVING. The OBRF status is updated accordingly with *updateStatus(rxstatus : t_RX_STATUS, pa : byte)*. The last fourth condition cover the case in which more correct flags are received. The **TmrRXTimeOut : ushort** is kept reset.

If **CurrState : t_AX_STATE** = ERROR_CHANCE the **RxByte : byte** and **StuffByte : byte** are updated as described previously. Then if RxByte is not a flag, according to FSM in figure 6.26 it is an error and state is reset to PREAMBLE. The last condition handles the case if it is a real AX_FLAG without stuffing executed (so it is not data, with **stuffed : bool** = 0): here the state can be brought again in SYNCW and **TmrRXTimeOut : ushort** is kept reset.

If **CurrState : t_AX_STATE** = PAYLOAD the RxByte and StuffByte are updated as

described previously. Then if it is the 7th byte (the first AX.25 address is received now) is controlled at run-time if the reception is addressed correctly: if not, then the state begin with PREAMBLE. The second condition limits the received bytes to **AX_BYTE_THR : byte const**. The third condition is checked if it is data and not a flag, with the string !(RxByte == AX_FLAG && stuffed == 0), because given A and B boolean values, apply the rule !(A and B) = (!A or !B). In this case the **ByteCount : short** is incremented, the RxBuffer[ByteCount]Â = RxByte and CRC is updated. After 8 increments, the BitCount is always 0. The last fourth condition checks if a real AX_FLAG is received. The RxBuffer update is done as in previous if condition, but now **RxStatus** = RX_RAW (with the consequent system updating *updateStatus(rxstatus : t_RX_STATUS, pa : byte)*, and the receiver begin from PREAMBLE state. The **TmrRXTimeOut** is kept reset.
Code:

```
public: static isr_CC1020RxData() {
if (SLOT_A::D0.read()){ // this ISR is triggered by a
 transition on a different pin, e.g. D1
CurrBit = 1;
}
else{
CurrBit = 0;
}

switch (CurrState) {

case PREAMBLE: //Bit synch made here. Could not happen at first time

RxByte = shiftIn(RxByte, CurrBit);
if (RxByte == AX_FLAG){
CurrState = SYNCW;
TmrRXTimeOut = 0;
FlagCount = 0;
BitCount = 0;
ByteCount = 0;
}
break;

case SYNCW: // word synch made here

RxByte = shiftIn(RxByte, CurrBit);
StuffByte = StuffStatus(StuffByte, CurrBit);

// state branches
if (BitCount == 8 && RxByte != AX_FLAG && FlagCount < FLAG_THR){
CurrState = ERROR_CHANCE;
}
else if (BitCount == 8 && RxByte == AX_FLAG && stuffed == 0 &&
```

```
 FlagCount < FLAG_THR){
FlagCount++;
BitCount = 0;
}
else if (BitCount == 8 && RxByte != AX_FLAG && FlagCount >= FLAG_THR){
ByteCount = 0;
BitCount = 0;
RxBuffer[ByteCount] = RxByte; //it's the first byte
hwCRC((RxByte&0x00FF));
RxStatus = RECEIVING;
updateStatus(RxStatus, paStatus);
CurrState = PAYLOAD;
}
else if (BitCount == 8 && RxByte == AX_FLAG && stuffed == 0 &&
 FlagCount >= FLAG_THR){
BitCount = 0;
RxStatus = RECEIVING;
updateStatus(RxStatus, paStatus);
}

TmrRXTimeOut = 0;
break;

case ERROR_CHANCE:
RxByte = shiftIn(RxByte, CurrBit);
StuffByte = StuffStatus(StuffByte, CurrBit);

if (BitCount == 8 && RxByte != AX_FLAG){
CurrState = PREAMBLE;
hwCRC_init(crcSeed);
}
else if (BitCount == 8 && RxByte == AX_FLAG && stuffed == 0){
FlagCount++;
BitCount = 0;
CurrState = SYNCW;
TmrRXTimeOut = 0;
}

break;

case PAYLOAD:
RxByte = shiftIn(RxByte, CurrBit);
StuffByte = StuffStatus(StuffByte, CurrBit);

     if (ByteCount == 7) { // to avoid the check everytime bitcount is 0
if (BitCount == 0){
if (strncmp(AX_SAT_ADDR, RxBuffer[PacketNum], 7)!=0){
CurrState = PREAMBLE;
}
}
```

```
}
else if (ByteCount > AX_BYTE_THR){
CurrState = PREAMBLE;
}
else if (BitCount == 8 && ~(RxByte == AX_FLAG && stuffed == 0)) {
ByteCount++;
RxBuffer[ByteCount]=RxByte;
BitCount = 0;
hwCRC((RxByte&0x00FF));
}
else if (BitCount == 8 && RxByte == AX_FLAG && stuffed == 0 && ByteCount > 0) {
ByteCount++; // mi salvo il flag
RxBuffer[ByteCount]=RxByte;
RxStatus = RX_RAW; // refer to fsm
updateStatus(RxStatus, paStatus);
CurrState = PREAMBLE;
}

TmrRXTimeOut = 0;
break;
}
return;
}
```

6.4.27 Transmitting State Machine

When a transmission is requested, the TRANSMITTING state is activated, from the FSM in figure 6.27. This will activate the lower level FSM depicted in figure 6.29. The principle is very similar to the one in reception, but the opposite: every bit is shifted out from the last updated **TxByte** through the *shiftOut(data : uchar)*. When start with the TX_PREAMBLE this byte is updated with a **AX_FLAG**, and sent **FLAG_THR_TX** times to allow the receiver the bit and word synchronization.

Then the **TxByte** is updated with the first value of the **TxBuffer : uchar[BUFFLEN]** which was prepared during when the OBC requested a transmission with a command, or upon beaconing. When the whole buffer is sent to the transceiver, a new state called TX_POSTAMBLE is activated, allowing the receiver to know that such transmission has been finished. The principle is the same as TX_PREAMBLE. Then the FSM in figure 6.28 starts again from the last state. This algorithm is implemented in the *isr_CC1020TxData()*.

6.4.28 isr_CC1020TxData()

This method is used to implement the algorithm in section 6.4.27. It is an interrupt routine used to send to the transceiver the content of the **TxBuffer : uchar[BUFFLEN]**. This function simply put on DIO() the single LSB of the **TxByte : byte**, taken from the LSByte of the TxBuffer. This interrupt is triggered on every falling edge of the signal on DCLK(). At the end of the transmission, the **RxStatus : t_RX_STATUS** = **PrevRxStatus : t_RX_STATUS** which was stored during the initialization of transmission. This is described by the relation between FSM RX Flag handle and TRANSMITTING. The **BitCount : byte** and **ByteCount : short** are shared with *isr_CC1020RxData()*. The behavior is described in FSM in figure 6.29.

When **CurrState : t_AX_STATE** = TX_PREAMBLE is shifted out the **AX_FLAG : byte const** value with *SLOT_A::D0.write(shiftOut(data : uchar) : bool)*, the bit counter **BitCount : byte** is incremented if the sent number of AX_FLAG bytes is still under the FLAG_THR _TX number and the **flagCount : byte** is incremented. When a proper FLAG_THR_TX number of flags are sent, the TxByte is updated with the first byte of **TxBuffer : uchar[BUFFLEN]**. Controlling if BitCount = 0 allow to update every time the **TxByte : byte** with AX_FLAG value.

If the **CurrState : t_AX_STATE** = TRANSMIT_DATA, then a certain number of flags are sent and the payload data is going to be sent. Here is shifted out the content of the **TxBuffer : uchar[BUFFLEN]** with with *SLOT_A::D0.write(shiftOut(data : uchar) : bool)*, the bit counter **BitCount** is incremented. It is also performed the stuffing with *StuffStatus(data : uchar, _bit*

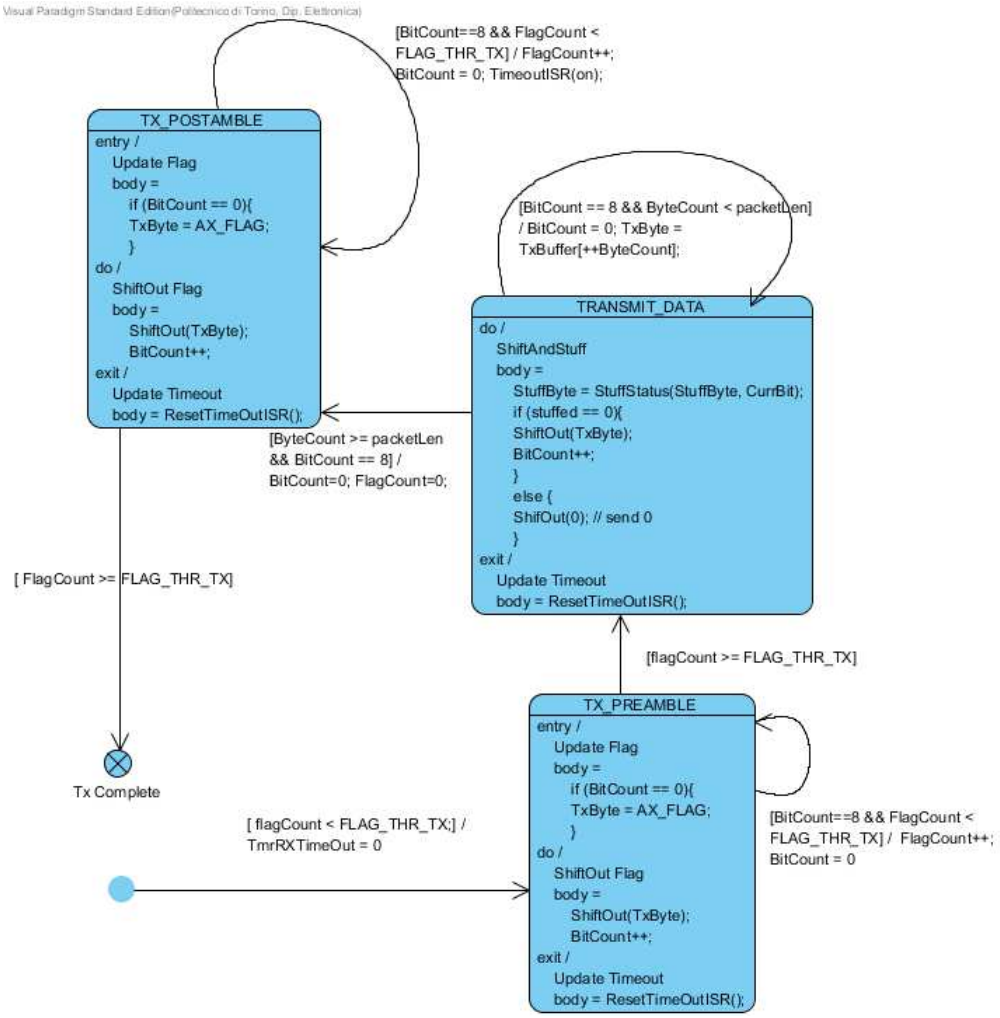

Figure 6.29. FSM for the flag handling

: uchar), and based on its outcome, will be decided if sending a 0 or the actual payload. Then is controlled if a **PacketLen** bytes are sent: if not, the ByteCount continue to index the next value of **TxBuffer : uchar[BUFFLEN]**. But when the PacketLen bytes are transmitted, the TX_POSTAMBLE state is activated and the counters **BitCount : byte** and **flagCount : byte** are reset for the postamble. The **ByteCount : short** value is not used anymore.

If the**CurrState : t_AX_STATE** = TX_POSTAMBLE, the behaviour is the same as the TX_PREAMBLE state, but the difference is when exiting: it is restored the **PrevRxStatus : t_RX_STATUS**, updating the system with *updateStatus(rxstatus : t_RX_STATUS, pa : byte)* and reinitialize everything in default mode with *CC1020InitRX(baudr : ulong, freq : AllowedFrequencies, mod : t_modulation, txpower : byte)*. The **TmrRXTimeOut : ushort** is kept reset.

Code:

```
public: static isr_CC1020TxData() {
if (RxStatus == TRANSMITTING){
switch (CurrState) {

case TX_PREAMBLE:

if (BitCount == 0){
TxByte = AX_FLAG;
}
SLOT_A::D0.write(shiftOut(TxByte));
BitCount++;
if (BitCount == 8 && FlagCount < FLAG_THR_TX) {
FlagCount++;
BitCount=0;
}
else {
CurrState = TRANSMIT_DATA;
BitCount = 0;
ByteCount = 0;
TxByte = TxBuffer[ByteCount];
}
break;

case TRANSMIT_DATA:

StuffByte = StuffStatus(StuffByte, CurrBit);
if (stuffed == 0){
SLOT_A::D0.write(shiftOut(TxByte)); // send first bit
 of TxByte with no added zeros
BitCount++;
}
else {
SLOT_A::D0.reset(); // send 0
}
if (BitCount == 8 && ByteCount < PacketLen){
BitCount = 0;
ByteCount++;
TxByte = TxBuffer[ByteCount];
}
else if (BitCount == 8 && ByteCount == PacketLen){
CurrState = TX_POSTAMBLE;
BitCount = 0;
FlagCount = 0;
}

case TX_POSTAMBLE:
if (BitCount == 0){
TxByte = AX_FLAG;
```

```
}
SLOT_A::D0.write(shiftOut(TxByte));
BitCount++;
if (BitCount == 8 && FlagCount < FLAG_THR_TX) {
FlagCount++;
BitCount=0;
}
else {
RxStatus = PrevRxStatus; // and shutdown the tx_isr triggering
updateStatus(RxStatus, paStatus);
CC1020InitRX(baud, freq, modulation, paPower);
}
break;
}
}
TmrRXTimeOut = 0;
return;
}
```

6.4.29 Bit storing and bit stuffing

Until now is described the process of handling the data in RX and TX mode, but in this section is introduced how bits are stored and how are implemented the techniques required by the AX.25 protocol. The sequence of the initialization in RX mode is described in section 6.4.13 or in 6.4.22 the TX mode. These will setup the MCU in order to trigger the *isr_CC1020RxData()* on rising edge of signal at transceiver's DCLK pin connected to the MCU, or the *isr_CC1020TxData()* on falling. At every call of these ISRs it is executed an algorithm described from 6.4.24. Now are going to be described the main sub-algorithms inside the TX and RX interrupt routines (namely *isr_CC1020TxData()* and *isr_CC1020RxData()*).

In reception

The FSM in figure 6.26, over the various controls to keep aligned bits and bytes received, uses two fundamental functions, the *shiftIn(RxByte, CurrBit)* and *StuffStatus(StuffByte, CurrBit)*. When the ISR is called, the digital value present at the DIO pin is immediately stored in the **CurrBit** variable. Then a buffer byte **RxByte** is filled with CurrBit values, by using the *ShiftIn()* function.

The AX.25 require that the flag must be not present in payload, but the data contained could assume any kind of value, even equal to the flag. For this reason, and fully transparently with respect to the data present in buffers (therefore at OSI Layer 2), is adopted the *bit stuffing*. It is made by the *StuffStatus()* in which check the sequence of the stream. The adopted AX.25 flag is 0b01111110, therefore must be controlled if an incoming transmission has more than 5 ones (which are 6 minus 1 bit for the difference from flag), if so it is a flag or an invalid data (invalid if more than 6 ones are present). The *StuffStatus()* function reuses the *ShiftIn()* to make this control, where the parameters now are not the RxByte, but the **StuffByte** w.r.t. the actual **CurrBit**. With this row in StuffStatus():

```
data = (shiftIn(data, _bit) & (STUFFED+1));
```

where STUFFED = 0b00111110, and STUFFED+1 is a mask which covers 6 LSB bits and it is checked if **data** contains the value STUFFED, that is equivalent from the transmitter point of view to put a 0 value after 5 high bits to avoid equalities with the flag, therefore this bit it is a value that need to be discarded, being part of the stuffing procedure when the stream was transmitted. The "destuffing" (discarding procedure of the current low bit) is made using the *destuff(data : uchar)*, present in StuffStatus(), which bring the actual data window back by one time-slot:

```
return(data >>= 1);
```

These controls are executed if the RxStatus is different from TRANSMITTING, otherwise the *StuffStatus()* will do the opposite task, implementing the stuffing instead of removing it, because in this case is called by the transmitting procedures.

In transmission

As mentioned, the *StuffStatus()* can be used in RX or TX modes, according to the RxStatus. In this case is shifted a **__bit : bool**, taken from the actual **TxByte : byte**, into the **StuffByte** (the same global variable used in reception). But here is made a check on the stream that will need to be transmitted, with a different mask, named TOSTUFF = 0b00011111:

```
data = (shiftIn(data, (bool)(TxByte & 0x1)) & TOSTUFF;
```

If **data** contains 5 ones from the LSB position it is marked the need to implement the stuffing. Since this *StuffStatus()* is called by the *isr_CC1020TxData()*, this ISR is the function which should check the global variable **stuffed : byte** modified by the StuffStatus(), in order to understand if (when stuffed = 1) should transmit a 0 (stuffing) or, viceversa, to use the *ShiftOut(TxByte : byte)* and put on the transceiver the first LSB available for the transmission (no stuffing). The meaning of every byte (therefore the data handling at higher level) is described in section 6.4.24.

6.4.30 destuff()

Undo the stuffing procedure, i.e. shifting out the last bit from **data**, which need to be discarded to compensate the bit stuffing implemented when transmitting the AX.25 packet. It is called by the *StuffStatus(data : uchar, _bit : uchar)*.

Code:

```
 Bk1B31A2S::destuff(uchar data) {
return(data >>= 1);
}
```

6.4.31 StuffStatus()

It is used to implement the algorithm in section 6.4.29. Here is performed a shift in of the **__bit** and masked with a value that covers the presence of 6 bits, here **STUFFED : byte const**+1 (0x3F or 0b0011 1111). So 3 events can occur:

- if the sequence has 6 ones (i.e. **STUFFED : byte const**+1) then is the same as the mask so the incoming packet, except for errors, is an **AX_FLAG : byte const**.

- the sequence has a 0 stuffed inside, so the incoming data (analyzed on an 8-bit window by the *shiftIn(data : uchar, _bit : uchar) : byte)* is, in binary, 0011 1110, or 0x3E (**STUFFED : byte const**). In this case the bit is destuffed using the *destuff(data : uchar)* to the **RxByte : byte** and is not taken into account by **BitCount : byte**, ignoring the last received bit, which was a 0 stuffed.

- any other different sequence will mismatch the byte from the previous two cases, so the data will be shifted in to the **RxByte : byte**.

The variable **stuffed : bool** is needed to know when an AX_FLAG inside the RxByte is really a flag. When TRUE, the caller of this function must ignore the last received bit. This function is also used when in TRANSMITTING. The **_bit** parameter is not used, but instead is directly check if the TxByte will contains five consecutive ones, i.e. 0x1F (**TOSTUFF : byte const**). If it is the case, **stuffed** = TRUE and a 0 value must be sent by the caller of this function, implementing the bit-stuffing in transmission.

Code:

```
public:
#pragma inline = forced
StuffStatus(uchar data, uchar _bit) {
if (RxStatus != TRANSMITTING){
data = (shiftIn(data, _bit) & (STUFFED+1)); // STUFFED+1 = 0x3F
if (data == STUFFED){
RxByte = destuff(RxByte);
stuffed = 1; // to know when an AX_FLAG inside the RxByte is
 really a flag. If 1, the RxByte contains data which is not
}
else{
BitCount++;
stuffed = 0;
}
}

else {
data = (shiftIn(data, (bool)(TxByte & 0x1)) & TOSTUFF; //5 ones, TOSTUFF = 0x1f
if (data == TOSTUFF){
//stuff();
stuffed = 1;
}
else {
BitCount++;
stuffed = 0;
}
}
return data;
```

```
}
```

6.4.32 shiftIn()

Writes the **_bit** value on the LSB (rightmost) position of **data**. Returns the updated **data**.
Code:

```
  byte Bk1B31A2S::shiftIn(uchar data, uchar _bit) {
return((data << 1) | _bit);
}
```

6.4.33 shiftOut()

Will return the LSB shifted out from **data**. The LSB is at right position. **data** is modified.
Code:

```
 bool Bk1B31A2S::shiftOut(uchar data) {
bool _bit = 0;
_bit = (data | ((bool) 0));
data >>= 1;
return (_bit);
}
```

6.4.34 hwCRC_init()

Initialize the CRC hardware of the MSP_430F5437A. The initial value (seed) is in **crcSeed : ushort const**, 16-bit wide and it is applied to the MPU's internal register.

```
public:
#pragma inline = forced
hwCRC_init(ushort crc) {
proc.cpu.crc.init(crc);
}
```

6.4.35 hwCRC_result()

Retrieve the 16bit value of the CRC from the internal registers of the processor MSP_430F5437A, in reversed order.

```
  Bk1B31A2S::hwCRC_result(ushort* crc) {
(*crc) = proc.cpu.crc.crc_result_in_reversed();
}
```

6.4.36 hwCRC()

Generates the parameter's CRC using the hardware of the MSP_430F5437A, obtaining the FCS of the AX.25. Receives in input the source **data** 16bit wide, but the function should be called with the higher byte always 0x00 due to the byte nature of the AX.25 protocol. The CRC is kept in the processor register ready for any further update or check.

If a 16 bit data is processed, the lower byte at the even address is used at the first clock cycle. During the second clock cycle, the higher byte is processed. Thus, it takes two clock cycles to process 16bit data, while it takes only one clock cycle to process byte data. Here are going to be used a byte sized data, in order to keep the AX.25 compatibility.

```
 Bk1B31A2S::hwCRC(uchar source) {
proc.cpu.crc.add_data_in_reversed(source);
}
```

6.4.37 checkCRC()

Checks if the AX.25 CRC corresponds to the actual content of the packet. The FCS comparison should be performed after the last call of the *subfieldID(buffer : char *, subBuff : char *, start : short &, mode : t_ID_MODE) : bool* in DATA mode. In this way the **crc : ushort** variable has been updated, and then can be compared with the processor's calculation using the *hwCRC_result(crc : ushort *)* called by this method.

```
Bk1B31A2S::checkCRC(ushort crc) {
ushort crc_temp = 0;
hwCRC_result(crc_temp);
if (crc_temp == crc) return 1;
else return 0;
}
```

6.4.38 System Timer

The class *Bk1B31A2S* contains a timer, named *Timer A1* for the actual MPU adopted. It used as a system tick, which allow to define a time base interval in this class. This timer is based, in turn, on another class named *TimerA1*. The methods contained here are called by the *Bk1B31A2S*, which initialize the timer on generating interrupt on its overflow. In figure 6.30 is shown when the interrupt flag TAIFG is set, when configured in up-mode.

The system clock is chosen to be 8MHz and divider used on the timer is 8, obtaining, when counting up to 0xFFFF, a system tick which is around 65ms.

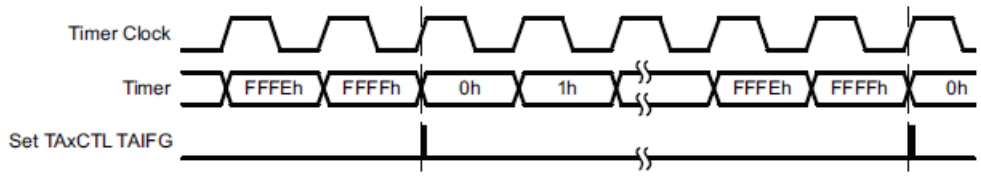

Figure 6.30. Interrupt flag setting of timer in up-mode

6.4.39 isr_TimerA1()

It is a method used to implement the timer handling described in section 6.4.38. This is an ISR which is called when the TimerA1 will overflow. The provided overflow period is around 65ms, therefore this is the system tick for its variables updating.

Every system tick are incremented the:

- **TmrBeacon : ushor** and a missing reset of this variable for 65***BEACON _TIMEOUT : byte const** milliseconds the auto-generating beacon will take place
- **TmrRXCheckCarrier : ushort**, which triggers the search of a RF carrier after 65***RSSI_ CHECK_TIMEOUT : byte const** milliseconds. This search mode is made by calling the *TogglePSEL()*.
- **TmrRXTimeOut : ushort** where a missing reset of this variable lead to a reset of the receiving state machine after 65***RX_TIMEOUT : byte const** milliseconds.

Code:

```
public: static _isr_timerA1() {
//Every system tick 65ms

if (RxStatus != RECEIVING){
TmrRXTimeOut = 0;
}
else if (TmrRXTimeOut++ >= RX_TIMEOUT && (CurrState == SYNCW || CurrState == PAYLOAD)){
if (RxStatus == RX_OFF ) { // ripristina valore precedente se eri in trasmissione,
quindi con RX spento
RxStatus = PrevRxStatus;
} else {
RxStatus = RX_IDLE;
}
}

if (TmrBeacon++ >= BEACON_TIMEOUT) {
SendBeacon = 1;
TmrBeacon = 0;
```

```
}

if (RxStatus == RX_IDLE && TmrRXCheckCarrier++ >= RSSI_CHECK_TIMEOUT) {
transceiver.CC1020.TogglePSEL();
TmrRXCheckCarrier = 0;
}


cpu.timerA1.clearInterrupt();
}
```

6.4.40 Methods based on external classes

In figure 6.3 are shown two classes which are necessary for the functioning of the main class *Bk1B31A2S*. These are the *Housekeeping* (from the external package 1B45) and the *MessageHandler* (from the same 1B45 package).

Housekeeping

This class support various methods and template configurations in order to keep updated the housekeeping register **housekeeping : HK_REDUNDANCY [LENGTH_HOUSEKEEPING]**. The housekeeping vector is triple redundant hardened and can store LENGTH_HOUSEKEEPING different values of different sensors. This class provide just the last read of sensors data, with no other particular statistics. The description of the vector is provided from section 4.3.

This class contains also the **configRegister : CS_REDUNDANCY [LENGTH_CONFIG]**. Its template parameter LENGTH_CONFIG defines the maximum number of configuration words which are available. The last vector provided is the **statusRegister : CS_REDUNDANCY [LENGTH_STATUS]**. Its template parameter LENGTH_STATUS defines the maximum number of status words which are available. The description of the content of these last two vectors is provided before, in section 4.2.8. All the data is triple hardened using the HK_REDUNDANCY template as "tripleData".

The connection of the housekeeping sensors is provided in figure 6.31, in which it is highlighted also the relative functions called when this class requires the periodic update. In fact, each system sensor contain a software class, which is used by this housekeeping to trigger the handling of the sensors: in other words the *housekeeping* class calls the *housekeeping()* functions of the sensors' classes. The connections are useful for the chapter 5, while the index names in figure 6.31 are described in section 4.3. It is configured by the *Bk1B31A2* class through the object **hk**, which provides the required templates. See figure 6.3.

The sensors were read through analog channels of the MCU, assigned using templates, using the *acquire()* method of the *ADC* external class.

Code:

```
Bk1B31A2S::housekeeping(ushort index) {
monitor2V8.housekeeping(index);
monitorVPA.housekeeping(index);
monitor3V3.housekeeping(index);
monitorPDB.housekeeping(index);
absCurr.housekeeping(index);
temperature.housekeeping(index);
monitorVREF.housekeeping(index);
```

```
}
```

MessageHandler

This class support methods and template configurations in order to handle the OBC requests and relative responses. It is the software implementation of the interfacing functions described in section 4.4. It is used the PROTOCOL template as I2C and other templates related to the selected SLOT (see figure 6.4 for the selected slots and pins).

This class contains the methods which are interrupt driven, handling every single byte in reception and in transmission. The synchronization between the request and the eventual further response is described in section 4.4 and implemented in this class, where after a full command has been received, a call of the *interpret(command : ushort)* of *Bk1B31A2S* class is made.

The interpreter starts with check sequence of the command. If corresponds to CMD_GET_ STATUS, CMD_TRANSMIT, or GET_PACKAGE the reset of the beacon timing happens, because of the interaction with the OBC, as depicted before in figure 6.17. If it is acommand which requires to return some data, the buffer of the MesageHandler **bufferRead : byte*** is initialized by a proper **payloadLen** length and filled with data described in use case Get Received Packet described in section 4.2.4. If it is a transmit command from the OBC, it is received a buffer which will be stored in a MesageHandler previously **bufferWrite : byte***. (As provided by 1B45 SubSystem Serial Data Bus).

The CMD_WAKEUP is used to firstly put in active mode the MCU and then initialize the transceiver. Conversely, when CMD_STANDBY the RxStatus is set accordingly as in figure 6.27, the transceiver put in Power Down mode and the MCU put in standby, but capable of listening from the bus, as provided by use cases in section 4.3. Always from these use cases, the CMD_SET_ADDR is used to change the destination address in the tile and keep it until new address is eventually set.

A final remark on the change of the configuration, is related to the nature of the **configRegister** which allows it to be modified. Therefore, any command which can lead to any kind of modification must call the already described *writeConfig()* in order to update the configuration and apply it, by calling the initialization of the transceiver in the default mode RX.

According to figure 6.17 and the other diagrams of the AraMiS protocol, every time a command CMD_GET_STATUS is issued, suddenly the OBC issue commands related to the RF interaction. This means that every time the CMD_GET_STATUS is issued, the beacon timeout counter should be restarted. This is handled in the *interpret()* too.

6.4.41 interpret()

This function is called by the *MessageHandler* class when a command is received.

If **command** = CMD_TRANSMIT, it is initiated the transmission sequence described in section 6.4.22. It is the implementation of use case Transmit (from section 4.3).

If **command** = GET_PACKAGE, is prepared the bus present in the *MessageHandler*, through the object **bus** of the *Bk1B31A2S* class, with the content of the **auxBuff : uchar[BUFFLEN]** prepared autonomously by the system according to section 6.4.2. It is the implementation of the use case Get Received Packet (section 4.2.4).

If **command** = CMD_WAKEUP the Bk1B4221W_Tile_Processor_4M unti is resumed from the low power mode and the Bk1B31A2_Transceiver_437MHz unit is put in RX mode. This implements the WakeUp use case (from section 4.3).

If **command** = CMD_STANDBY the Bk1B31A2_Transceiver_437MHz is put in power down mode, and then also the Bk1B4221W_Tile_Processor_4M unit. Implements the Standby use case (from section 4.3).

If **command** = CMD_SET_ADDR is copied to **addressGround : uchar[7]** the desired address. In order to adapt it to the AX.25 protocol, the single left shift is automatically performed at run-time before storing the new address.

If command = CMD_DEPLOY issue the opening command to the antenna.

It is also checked if the OBC had required any possible modification of the **configRegister : CS_ REDUNDANCY [LENGTH_ CONFIG]**. In this case it is applied the configuration to the tile and reinitialize it in RX mode, because the configuration could affect the channel parameters and therefore the Bk1B31A2_Transceiver_437MHz unit should be reinitialized in the RX (default) mode.

When issued the CMD_GET_STATUS, among with GET_PACKAGE and CMD_TRANSMIT, the beacon timeout counter is reset.

Code:

```
Bk1B31A2S::interpret(ushort command) {
TmrBeacon = 0;

switch (command) {
case Commands.CMD_TRANSMIT:
CC1020InitTX(baud, freq, modulation, paPower);
break;
case Commands.GET_PACKAGE:
bus.lenght = payloadLen; // contains all the payloadLen updated after the unpacking
bus.bufferRead[0] = ns;
```

```
memcpy(bus.bufferRead+1, auxBuff, bus.length);
RxStatus = RX_IDLE;
updateStatus(RxStatus, paStatus);
break;

case bus.message::MessageHandler.command.CMD_WAKEUP:
Wakeup();
CC1020InitRX(baud, freq, modulation, paPower);
break;

case bus.message::MessageHandler.command.CMD_STANDBY:
CC1020PD(); //include lo switch e pa
RxStatus = RX_IDLE; // non si deve perdere il
contenuto in RAM
Standby();
break;

case Commands.CMD_SET_ADDR:
for (byte i = 0; i < 6; i++)
addressGround[i] = (bus.bufferWrite[i] << 1);
addressGround[6] = bus.bufferWrite[6]; //SSID
break;

case (bus.message::MessageHandler.command.CMD_SET_CONFIGURATION ||
 bus.message::MessageHandler.command.CMD_RESET_CONFIGURATION ||
  bus.message::MessageHandler.command.CMD_WRITE_CONFIGURATION)
writeConfig(&baud,  &freq, &modulation, &paPower);
CC1020InitRX(baud, freq, modulation, paPower);
break;

case (bus.message::MessageHandler.command.CMD_GET_STATUS):
TmrBeacon = 0;
break;
}
}
```

6.4.42 CC1020PD()

This function put the Bk1B31A2_Transceiver_437MHz unit in power down mode. A proper configuration of the transceiver's registers is assumed to be already made, allowing to issue only one single command to CC1020.

```
 Bk1B31A2S::CC1020PD() {
transceiver.CC1020.SetReg(CC1020_MAIN, 0x1F);
transceiver.CC1020.SetReg(CC1020_PA_POWER, 0x00);
// p. 55 datasheet
}
```

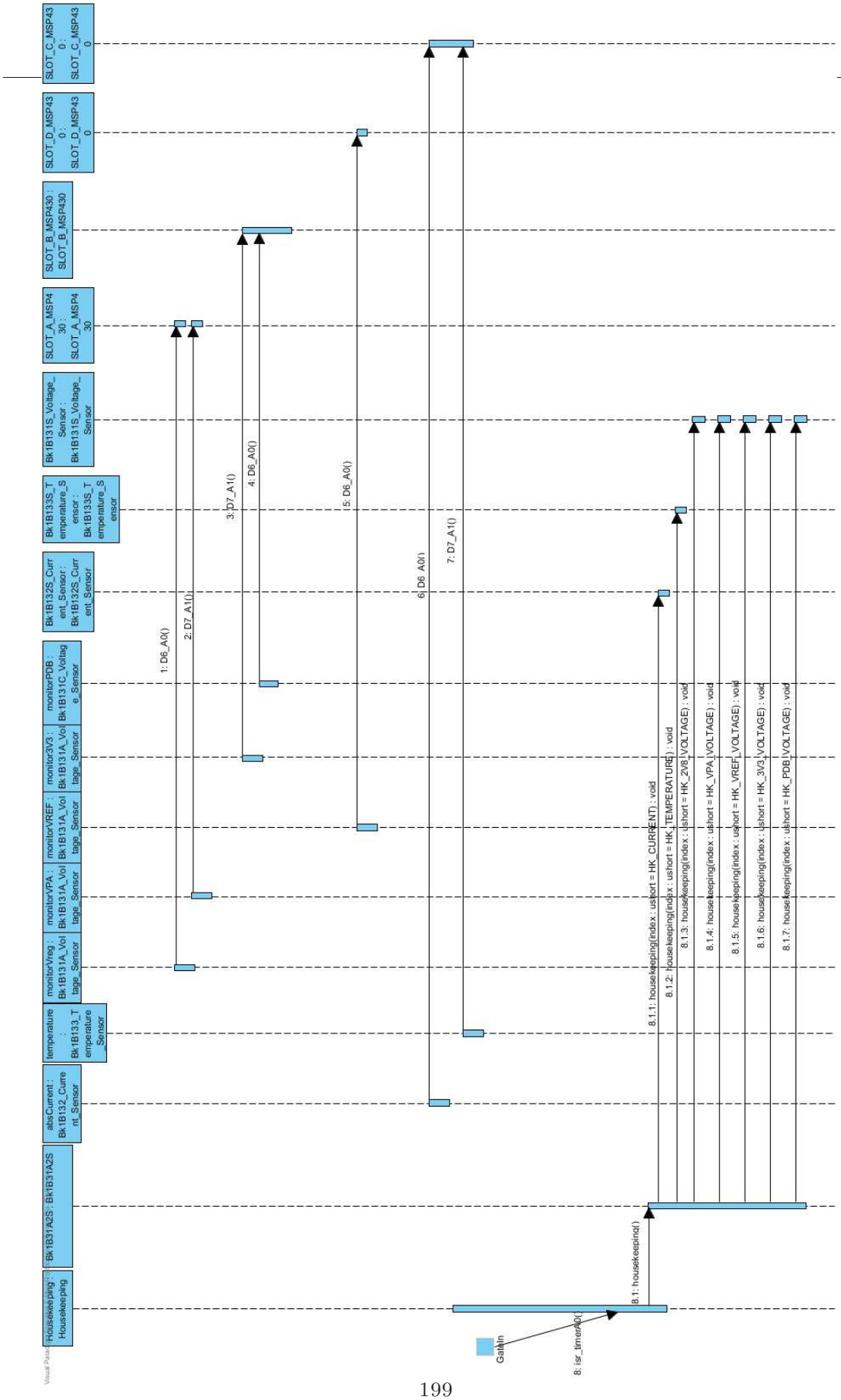

Figure 6.31. Housekeeping connections and functions involved

6.4.43 CC1020AutoWakeUpMode()

The CC1020 when put in RX mode need a proper sequence of configurations. Moreover, in this application is configured to be in Automatic Power-Up Sequencing, in which upon a proper signal in PSEL pin, start searching for a received signal of a strength (RSSI) which is over a defined threshold. The CC1020 provides a reading of the RSSI level from the RSSI register. The RSSI reading is a logarithmic measure of the average voltage amplitude after the digital filter in the digital part of the IF chain and can be referred to the incoming relative power. This is proportional to VGA gain too, therefore should be kept in mind its amplification, if the absolute power reading is needed. Despite this, it is only needed the relative value of the RSSI for the automatic power-up triggering.

According to the required conditions, SmartRF Studio helps in defining the optimum levels of the VGA and a starting point for a minimum carrier sense threshold, considering the bandwidth, frequency deviation and crystal tolerance defined. The threshold value set in the VGA4 register can be offset to obtain an higher or lower threshold. This threshold is used to set the operating point of the gain control, where its hysteresis can be tuned.

The threshold comparison is used for the Automatic Power-up sequencing mode. The initialization for this modality starts from the power off assumption, so resetting all the registers each time, in order to discard the previous values on transceiver's on-chip registers, reducing the variables time life and therefore reducing SEUs events. This mode allow the transceiver to wake up from Power Down mode upon toggling the PSEL pin, this automatically put the RX mode and search for an incoming carrier higher than the threshold level of RSSI and automatically enters in RX mode. If no signal is detected, will turn back in power down automatically.

Note for the testing phase: Activating the reception periodically, assuming that the power down mode lasts less than half of the preamble duration, will introduce a power saving improvement without loosing any information.

Initialization steps

The steps to use this mode are made by a sequence of commands in order to reset the transceiver. Then configuring the RX parameters (associated to transceiver's configuration registers labelled as A) in order to obtain the settings devised in previous chapters. The configuration follow different cases for different baudrates, since each require a fine tuning of the on-chip RF components and a well determined bandwidth (as seen before). Despite a single baudrate is used, a full support is provided for all the possible settings, if needed. The configuration follows the use cases, therefore is activated the NRZ coding, no scrambling is implemented, the ADC frequency is set to the optimal designed value, AFC settling time is set to the slowest, obtaining a more precise frequency control. To ease the development, for some settings are used the already tested methods of *CC1020* class, which are extracting the register values from the required parameters. These decisions were made considering the slow speed of satellite data, which requires a precise control of the transceiver, therefore reducing errors of the transceiver's internal measurements, and not necessarily reducing the attach time in RX mode, which is not so important in this moment.

Then is configured the signalling of the continuous lock of PLL, that will be checked to verify the calibration. This value is present in the LOCK register. The receive chain and PA are put in power down (PD_MODE = 1). The auto-calibration starts and will be performed again if the PLL does not lock, situation that could happens. After this the chip is put in full power up (PD mode 0), then is activated the automatic power-up, which end by putting the chip in full power down. After a PSEL toggle is performed and the RSSI triggers the RX mode, and so a packet is potentially received or the receiver goes in timeout, this sequence should be reinitialized. [12]

This method implements the Automatic Power-Up Sequencing mode mentioned in section 6.4.43. This method can be used for different configurations by using the proper parameters: **baudr : ulong** for the chosen baudrate, **freq : AllowedFrequencies** for the used carrier frequency, **mod : t_modulation** for the selected RF modulation. The MCU's pins connected to the transceiver are initialized according to figures 6.9 and 6.8. Then it is reset, preparing it for the programming.

A set of switch-case is deployed to handle the requested baudrate configuration. Each configuration contains a defined set of **SetReg()** methods for writing a different configuration present in the enumeration *TRANSCEIVER_SETTINGS* while variables **dev** (deviation) and **bw** (bandwidth) are updated.

Then are set the transceiver's internal configuration modules A and B, respectively, for RX and TX modes, using some already tested methods. To avoid possible errors the TX part is configured

too, using *SetFreqA()* and *SetFreqB()* methods. The *Modem()* configures the modem hardware generating variables for the MODEM register. According to *CC1020AutoWakeUpMode()* **mod** parameter, the updated **dev** and **bw** variables, are then set the deviation (with *Deviation((ushort)mod, dev)*), the filter bandwitdth (with *FilterBandWidth(bw, baudr)*) and the automatic frequency control with *AFC_control(CC_SETTLING, dev)*.

After the configuration (which is common with TX mode in next section) is followed the sequence labelled WakeUpCC1020ToRX in figure 6.32 provided by TI, turning on the crystal, bias generator and synthesizer. Now the transceiver is in PD mode 1 (the core is active but separated from the outside, so the receive chain and the internal PA are in power down) and it is performed the next step in figure 6.32, the calibration, using the *CC1020Calibrate()* method. Here the label SetupCC1020PD is skipped and since after the calibration the PLL is in lock, the auto wake-up mode can be activated by putting the CC1020 in PD mode 0 (which means that is fully power up and connected to the internal receive chain); this step consists in activating the on-chip configuration labelled A for the RX mode, by writing PDMODE0_RX_A value in MAIN register. Now after a delay of at least 100 us, it is put in auto powering up mode by writing in MAIN the value AUTO_POWERING_UP. The chip now waits for the PSEL pin to be toggled.

Code:

```
Bk1B31A2S::CC1020AutoWakeUpMode(ulong baudr, Use_Cases::AllowedFrequencies freq,
Bk1B31A2W_OBRF_437MHz::t_modulation mod) {
//init CC to PD.  The CC1020 class uses  bitbanging,
 NOT SPI (is compatible). PSEL is high
transceiver.CC1020_Init(); //PSEL high
// From AN, first reset
transceiver.SetReg(CC1020_MAIN, MAIN_RESET);
transceiver.SetReg(CC1020_MAIN, MAIN_OUT_RESET); //out of reset
//sequence from RF Studio. Now is PD (mode is described in documentation of this class)
// configuration
ulong dev = 0;
ulong bw = 0;
//consigliati per bandw con doppler,
 quindi VGAx giÃ  regolati
// cambiare i VGA se sul campo non funziona correttamente
(che sia sensitivitÃ , selettivitÃ , ecc)
switch (baudr) {

case 2400: //25, ma dev'essere 50
transceiver.SetReg(CC1020_INTERFACE, (XOSC_BYPASS | SEP_TX_RX | NOGATE_DCLK_PLL
 | GATE_DCLK_CS | NO_PA | NO_LNA | PA_LOW | LNA_LOW));
transceiver.SetReg(CC1020_RESET, 0xff);
transceiver.SetReg(CC1020_SEQUENCING, (PSEL_TOGGLE | WAIT_32ADC | MAXCS_WAIT));
transceiver.SetReg(CC1020_CLOCK_A, CONF2400);
```

```
transceiver.SetReg(CC1020_CLOCK_B, CONF2400);
transceiver.SetReg(CC1020_VCO, (VCO_CURR_2_8_A | VCO_CURR_2_8_B));
transceiver.SetReg(CC1020_VGA1, (FREEZE_32ADC | WAIT_16FCLK
| CS_SIG_RESET2CY | CS_SIG_SET2CY));
transceiver.SetReg(CC1020_VGA2, (AGC_AVG_4CY | HYSTER_GAIN | AGC_ON
 | LNA_SETTING | MAX_LNA | MIN_LNA));
transceiver.SetReg(CC1020_VGA3, (VGA_MAX_GAIN | VGA_DOWN));
transceiver.SetReg(CC1020_VGA4, (CS_LEVEL | VGA_UP));
transceiver.SetReg(CC1020_LOCK, (LOCK_CONTINUOUS | LOCK_WINDOW |
 LOCK_RESTART_MODE | LOCK_ACCUR));
transceiver.SetReg(CC1020_FRONTEND, (LNAMIX_CURRENT, LNA_CURRENT,
 MIX_CURRENT, LNA2_CURRENT, SDC_CURRENT, LNAMIX__BIAS));
transceiver.SetReg(CC1020_ANALOG, (BAND0 | LOSC_DC | BLANK_OFF |
 PHASE_SHORT_DELAY | PA_BOOST | DIV_BUFF_CURRENT));
transceiver.SetReg(CC1020_BUFF_SWING, (PRE_SWING |
 RX_SWING | TX_SWING));
transceiver.SetReg(CC1020_BUFF_CURRENT, (PRE_CURR | RX_CURR | TX_CURR));
transceiver.SetReg(CC1020_PLL_BW, PLL_BW);
transceiver.SetReg(CC1020_CALIBRATE, (NO_CAL | CAL_WAIT
 | CAL_SINGLE | CAL_ITER));
transceiver.SetReg(CC1020_MATCH, 0x00);
transceiver.SetReg(CC1020_PHASE_COMP, 0x00);
transceiver.SetReg(CC1020_GAIN_COMP, 0x00);
transceiver.SetReg(CC1020_POWERDOWN, 0x00);
dev = 2025;
bw = 25000;
break;

case 4800: //50kHz
transceiver.SetReg(CC1020_INTERFACE, (XOSC_BYPASS | SEP_TX_RX
| NOGATE_DCLK_PLL | GATE_DCLK_CS | NO_PA | NO_LNA | PA_LOW | LNA_LOW));
transceiver.SetReg(CC1020_RESET, 0xff);
transceiver.SetReg(CC1020_SEQUENCING, (PSEL_TOGGLE | WAIT_32ADC |
 MAXCS_WAIT));
transceiver.SetReg(CC1020_CLOCK_A, CONF4800);
transceiver.SetReg(CC1020_CLOCK_B, CONF4800);
transceiver.SetReg(CC1020_VCO, (VCO_CURR_2_8_A | VCO_CURR_2_8_B));
transceiver.SetReg(CC1020_VGA1, (FREEZE_32ADC |
WAIT_16FCLK | CS_SIG_RESET2CY | CS_SIG_SET2CY));
transceiver.SetReg(CC1020_VGA2, (AGC_AVG_4CY | HYSTER_GAIN |
 AGC_ON | LNA_SETTING | MAX_LNA | MIN_LNA));
transceiver.SetReg(CC1020_VGA3, (VGA_MAX_GAIN1 | VGA_DOWN));
transceiver.SetReg(CC1020_VGA4, (CS_LEVEL1 | VGA_UP));
transceiver.SetReg(CC1020_LOCK, (LOCK_CONTINUOUS | LOCK_WINDOW |
 LOCK_RESTART_MODE | LOCK_ACCUR));
transceiver.SetReg(CC1020_FRONTEND, (LNAMIX_CURRENT, LNA_CURRENT,
MIX_CURRENT, LNA2_CURRENT, SDC_CURRENT, LNAMIX__BIAS));
transceiver.SetReg(CC1020_ANALOG, (BAND0 | LOSC_DC | BLANK_OFF |
 PHASE_SHORT_DELAY | PA_BOOST | DIV_BUFF_CURRENT));
transceiver.SetReg(CC1020_BUFF_SWING, (PRE_SWING
```

```
 | RX_SWING | TX_SWING));
transceiver.SetReg(CC1020_BUFF_CURRENT, (PRE_CURR
 | RX_CURR | TX_CURR));
transceiver.SetReg(CC1020_PLL_BW, PLL_BW);
transceiver.SetReg(CC1020_CALIBRATE, (NO_CAL | CAL_WAIT
 | CAL_SINGLE | CAL_ITER));
transceiver.SetReg(CC1020_MATCH, 0x00);
transceiver.SetReg(CC1020_PHASE_COMP, 0x00);
transceiver.SetReg(CC1020_GAIN_COMP, 0x00);
transceiver.SetReg(CC1020_POWERDOWN, 0x00);
dev = 2475;
bw = 50000;
break;

case 9600: //50kHz
transceiver.SetReg(CC1020_INTERFACE, (XOSC_BYPASS | SEP_TX_RX |
 NOGATE_DCLK_PLL | GATE_DCLK_CS | NO_PA | NO_LNA | PA_LOW | LNA_LOW));
transceiver.SetReg(CC1020_RESET, 0xff);
transceiver.SetReg(CC1020_SEQUENCING, (PSEL_TOGGLE
| WAIT_32ADC | MAXCS_WAIT));
transceiver.SetReg(CC1020_CLOCK_A, CONF9600);
transceiver.SetReg(CC1020_CLOCK_B, CONF9600);
transceiver.SetReg(CC1020_VCO, (VCO_CURR_2_8_A | VCO_CURR_2_8_B));
transceiver.SetReg(CC1020_VGA1, (FREEZE_32ADC
| WAIT_16FCLK | CS_SIG_RESET2CY | CS_SIG_SET2CY));
transceiver.SetReg(CC1020_VGA2, (AGC_AVG_4CY
| HYSTER_GAIN | AGC_ON
 | LNA_SETTING | MAX_LNA | MIN_LNA));
transceiver.SetReg(CC1020_VGA3, (VGA_MAX_GAIN1 | VGA_DOWN));
transceiver.SetReg(CC1020_VGA4, (CS_LEVEL1 | VGA_UP));
transceiver.SetReg(CC1020_LOCK, (LOCK_CONTINUOUS | LOCK_WINDOW
| LOCK_RESTART_MODE | LOCK_ACCUR));
transceiver.SetReg(CC1020_FRONTEND, (LNAMIX_CURRENT, LNA_CURRENT,
 MIX_CURRENT, LNA2_CURRENT, SDC_CURRENT, LNAMIX__BIAS));
transceiver.SetReg(CC1020_ANALOG, (BAND0 | LOSC_DC | BLANK_OFF |
 PHASE_SHORT_DELAY | PA_BOOST | DIV_BUFF_CURRENT));
transceiver.SetReg(CC1020_BUFF_SWING, (PRE_SWING
 | RX_SWING | TX_SWING));
transceiver.SetReg(CC1020_BUFF_CURRENT, (PRE_CURR
 | RX_CURR | TX_CURR));
transceiver.SetReg(CC1020_PLL_BW, PLL_BW);
transceiver.SetReg(CC1020_CALIBRATE, (NO_CAL |
CAL_WAIT | CAL_SINGLE | CAL_ITER));
transceiver.SetReg(CC1020_MATCH, 0x00);
transceiver.SetReg(CC1020_PHASE_COMP, 0x00);
transceiver.SetReg(CC1020_GAIN_COMP, 0x00);
transceiver.SetReg(CC1020_POWERDOWN, 0x00);
dev = 4950;
bw = 50000;
break;
```

```
case 19200: //100khz BW
transceiver.SetReg(CC1020_INTERFACE, (XOSC_BYPASS | SEP_TX_RX
| NOGATE_DCLK_PLL | GATE_DCLK_CS | NO_PA | NO_LNA | PA_LOW | LNA_LOW));
transceiver.SetReg(CC1020_RESET, 0xff);
transceiver.SetReg(CC1020_SEQUENCING, (PSEL_TOGGLE |
WAIT_32ADC | MAXCS_WAIT));
transceiver.SetReg(CC1020_CLOCK_A, CONF19200);
transceiver.SetReg(CC1020_CLOCK_B, CONF19200);
transceiver.SetReg(CC1020_VCO, (VCO_CURR_2_8_A | VCO_CURR_2_8_B));
transceiver.SetReg(CC1020_VGA1, (FREEZE_32ADC |
WAIT_16FCLK | CS_SIG_RESET2CY | CS_SIG_SET2CY));
transceiver.SetReg(CC1020_VGA2, (AGC_AVG_4CY | HYSTER_GAIN | AGC_ON
 | LNA_SETTING | MAX_LNA | MIN_LNA));
transceiver.SetReg(CC1020_VGA3, (VGA_MAX_GAIN2 | VGA_DOWN));
transceiver.SetReg(CC1020_VGA4, (CS_LEVEL1 | VGA_UP));
transceiver.SetReg(CC1020_LOCK, (LOCK_CONTINUOUS | LOCK_WINDOW |
 LOCK_RESTART_MODE | LOCK_ACCUR));
transceiver.SetReg(CC1020_FRONTEND, (LNAMIX_CURRENT, LNA_CURRENT,
 MIX_CURRENT, LNA2_CURRENT,  SDC_CURRENT, LNAMIX__BIAS));
transceiver.SetReg(CC1020_ANALOG, (BAND0 | LOSC_DC | BLANK_OFF |
 PHASE_SHORT_DELAY | PA_BOOST | DIV_BUFF_CURRENT));
transceiver.SetReg(CC1020_BUFF_SWING, (PRE_SWING | RX_SWING | TX_SWING));
transceiver.SetReg(CC1020_BUFF_CURRENT, (PRE_CURR | RX_CURR | TX_CURR));
transceiver.SetReg(CC1020_PLL_BW, PLL_BW);
transceiver.SetReg(CC1020_CALIBRATE, (NO_CAL |
CAL_WAIT | CAL_SINGLE | CAL_ITER));
transceiver.SetReg(CC1020_MATCH, 0x00);
transceiver.SetReg(CC1020_PHASE_COMP, 0x00);
transceiver.SetReg(CC1020_GAIN_COMP, 0x00);
transceiver.SetReg(CC1020_POWERDOWN, 0x00);
dev = 9900;
bw = 100000;
break;

case 38400: //150
transceiver.SetReg(CC1020_INTERFACE, (XOSC_BYPASS | SEP_TX_RX |
 NOGATE_DCLK_PLL | GATE_DCLK_CS | NO_PA | NO_LNA | PA_LOW | LNA_LOW));
transceiver.SetReg(CC1020_RESET, 0xff);
transceiver.SetReg(CC1020_SEQUENCING, (PSEL_TOGGLE | WAIT_32ADC | MAXCS_WAIT));
transceiver.SetReg(CC1020_CLOCK_A, CONF38400);
transceiver.SetReg(CC1020_CLOCK_B, CONF38400);
transceiver.SetReg(CC1020_VCO, (VCO_CURR_2_8_A | VCO_CURR_2_8_B));
transceiver.SetReg(CC1020_VGA1, (FREEZE_32ADC | WAIT_16FCLK |
 CS_SIG_RESET2CY | CS_SIG_SET2CY));
transceiver.SetReg(CC1020_VGA2, (AGC_AVG_4CY | HYSTER_GAIN | AGC_ON |
 LNA_SETTING | MAX_LNA | MIN_LNA));
transceiver.SetReg(CC1020_VGA3, (VGA_MAX_GAIN | VGA_DOWN));
transceiver.SetReg(CC1020_VGA4, (CS_LEVEL3 | VGA_UP));
transceiver.SetReg(CC1020_LOCK, (LOCK_CONTINUOUS | LOCK_WINDOW |
```

```
 LOCK_RESTART_MODE | LOCK_ACCUR));
transceiver.SetReg(CC1020_FRONTEND, (LNAMIX_CURRENT, LNA_CURRENT,
 MIX_CURRENT, LNA2_CURRENT, SDC_CURRENT, LNAMIX__BIAS));
transceiver.SetReg(CC1020_ANALOG, (BAND0 | LOSC_DC | BLANK_OFF |
 PHASE_SHORT_DELAY | PA_BOOST | DIV_BUFF_CURRENT));
transceiver.SetReg(CC1020_BUFF_SWING, (PRE_SWING | RX_SWING | TX_SWING));
transceiver.SetReg(CC1020_BUFF_CURRENT, (PRE_CURR | RX_CURR | TX_CURR));
transceiver.SetReg(CC1020_PLL_BW, PLL_BW);
transceiver.SetReg(CC1020_CALIBRATE, (NO_CAL | CAL_WAIT | CAL_SINGLE | CAL_ITER));
transceiver.SetReg(CC1020_MATCH, 0x00);
transceiver.SetReg(CC1020_PHASE_COMP, 0x00);
transceiver.SetReg(CC1020_GAIN_COMP, 0x00);
transceiver.SetReg(CC1020_POWERDOWN, 0x00);
dev = 19800;
bw = 150000;
break;

case 76800: //200
transceiver.SetReg(CC1020_INTERFACE, (XOSC_BYPASS | SEP_TX_RX
| NOGATE_DCLK_PLL | GATE_DCLK_CS | NO_PA | NO_LNA | PA_LOW | LNA_LOW));
transceiver.SetReg(CC1020_RESET, 0xff);
transceiver.SetReg(CC1020_SEQUENCING, (PSEL_TOGGLE | WAIT_32ADC |
MAXCS_WAIT));
transceiver.SetReg(CC1020_CLOCK_A, CONF76800);
transceiver.SetReg(CC1020_CLOCK_B, CONF76800);
transceiver.SetReg(CC1020_VCO, (VCO_CURR_2_8_A | VCO_CURR_2_8_B));
transceiver.SetReg(CC1020_VGA1, (FREEZE_32ADC |
 WAIT_16FCLK | CS_SIG_RESET2CY | CS_SIG_SET2CY));
transceiver.SetReg(CC1020_VGA2, (AGC_AVG_4CY | HYSTER_GAIN | AGC_ON
 | LNA_SETTING | MAX_LNA | MIN_LNA));
transceiver.SetReg(CC1020_VGA3, (VGA_MAX_GAIN3 | VGA_DOWN));
transceiver.SetReg(CC1020_VGA4, (CS_LEVEL2 | VGA_UP));
transceiver.SetReg(CC1020_LOCK, (LOCK_CONTINUOUS
| LOCK_WINDOW | LOCK_RESTART_MODE | LOCK_ACCUR));
transceiver.SetReg(CC1020_FRONTEND, (LNAMIX_CURRENT, LNA_CURRENT,
 MIX_CURRENT, LNA2_CURRENT, SDC_CURRENT, LNAMIX__BIAS));
transceiver.SetReg(CC1020_ANALOG, (BAND0 | LOSC_DC | BLANK_OFF |
 PHASE_SHORT_DELAY | PA_BOOST | DIV_BUFF_CURRENT));
transceiver.SetReg(CC1020_BUFF_SWING, (PRE_SWING
| RX_SWING | TX_SWING));
transceiver.SetReg(CC1020_BUFF_CURRENT, (PRE_CURR | RX_CURR | TX_CURR));
transceiver.SetReg(CC1020_PLL_BW, PLL_BW);
transceiver.SetReg(CC1020_CALIBRATE, (NO_CAL | CAL_WAIT | CAL_SINGLE | CAL_ITER));
transceiver.SetReg(CC1020_MATCH, 0x00);
transceiver.SetReg(CC1020_PHASE_COMP, 0x00);
transceiver.SetReg(CC1020_GAIN_COMP, 0x00);
transceiver.SetReg(CC1020_POWERDOWN, 0x00);
dev = 36000;
bw = 200000;
break;
```

```
case 153600: //500
transceiver.SetReg(CC1020_INTERFACE, (XOSC_BYPASS | SEP_TX_RX
| NOGATE_DCLK_PLL | GATE_DCLK_CS | NO_PA | NO_LNA | PA_LOW | LNA_LOW));
transceiver.SetReg(CC1020_RESET, 0xff);
transceiver.SetReg(CC1020_SEQUENCING, (PSEL_TOGGLE | WAIT_32ADC | MAXCS_WAIT));
transceiver.SetReg(CC1020_CLOCK_A, CONF153600);
transceiver.SetReg(CC1020_CLOCK_B, CONF153600);
transceiver.SetReg(CC1020_VCO, (VCO_CURR_2_8_A | VCO_CURR_2_8_B));
transceiver.SetReg(CC1020_VGA1, (FREEZE_32ADC | WAIT_16FCLK
 | CS_SIG_RESET2CY | CS_SIG_SET2CY));
transceiver.SetReg(CC1020_VGA2, (AGC_AVG_4CY | HYSTER_GAIN | AGC_ON
 | LNA_SETTING | MAX_LNA | MIN_LNA));
transceiver.SetReg(CC1020_VGA3, (VGA_MAX_GAIN4 | VGA_DOWN));
transceiver.SetReg(CC1020_VGA4, (CS_LEVEL4 | VGA_UP));
transceiver.SetReg(CC1020_LOCK, (LOCK_CONTINUOUS | LOCK_WINDOW
 | LOCK_RESTART_MODE | LOCK_ACCUR));
transceiver.SetReg(CC1020_FRONTEND, (LNAMIX_CURRENT, LNA_CURRENT,
 MIX_CURRENT, LNA2_CURRENT, SDC_CURRENT, LNAMIX__BIAS));
transceiver.SetReg(CC1020_ANALOG, (BAND0 | LOSC_DC | BLANK_OFF
 | PHASE_SHORT_DELAY | PA_BOOST | DIV_BUFF_CURRENT));
transceiver.SetReg(CC1020_BUFF_SWING, (PRE_SWING | RX_SWING | TX_SWING));
transceiver.SetReg(CC1020_BUFF_CURRENT, (PRE_CURR | RX_CURR | TX_CURR));
transceiver.SetReg(CC1020_PLL_BW, PLL_BW);
transceiver.SetReg(CC1020_CALIBRATE, (NO_CAL | CAL_WAIT | CAL_SINGLE | CAL_ITER));
transceiver.SetReg(CC1020_MATCH, 0x00);
transceiver.SetReg(CC1020_PHASE_COMP, 0x00);
transceiver.SetReg(CC1020_GAIN_COMP, 0x00);
transceiver.SetReg(CC1020_POWERDOWN, 0x00);
dev = 72000;
bw = 500000;
break;

default:
break;
}

transceiver.SetFreqA(freq, REF_DIV); //ref div = 2
transceiver.SetFreqB(freq, REF_DIV);

transceiver.Modem(F_ADC, NO_SCRAMB, NRZ); //fadc 1.2288,
 no scrambling 0, NRZ (1-1 = 00)
transceiver.Deviation((ushort)mod, dev);
transceiver.AFC_control(CC_SETTLING, dev); // settling max (3)
transceiver.FilterBandWidth(bw, baudr);
transceiver.SetReg(CC1020_MAIN, RX_A_PDMODE1_XOSC);  //now it is in PD
 mode 1, xosc on
transceiver.WAIT_CYCLE();
transceiver.SetReg(CC1020_MAIN, RX_A_PDMODE1_XOSC_BIAS);  //now it is in PD mode 1, bias on
transceiver.WAIT_CYCLE();  // wait. see p.55 datasheet
```

```
transceiver.SetReg(CC1020_MAIN, RX_A_PDMODE1_ON);  //now it is in PD mode 1,
synth on, FULL ON
transceiver.SetReg(CC1020_PA_POWER, 0x00); // no spurs
CC1020Calibrate(); // AN070
transceiver.SetReg(CC1020_MAIN, PDMODE0_RX_A);  //now it is in PD mode 0, put in RX
transceiver.WAIT_CYCLE();  //at least 100us
transceiver.SetReg(CC1020_MAIN, AUTO_POWERING_UP);  // put in
 auto-powering up, wait for PSEL to toggle
}
```

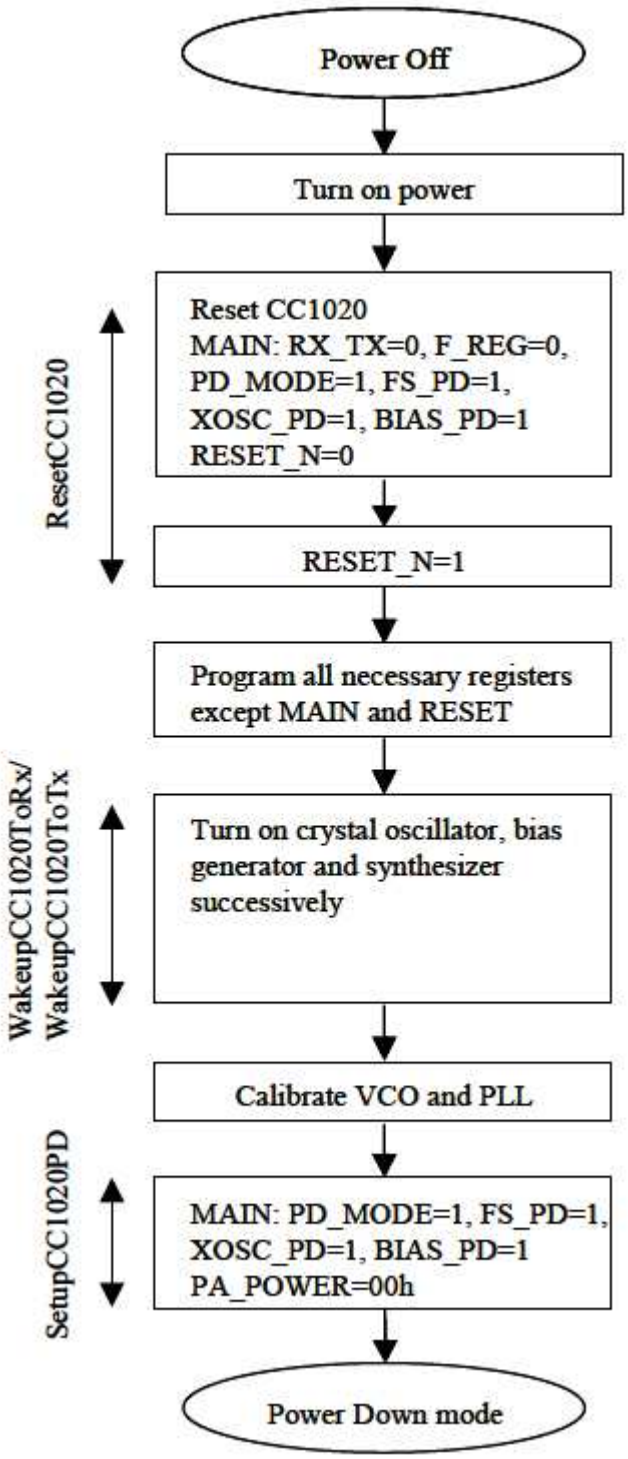

Figure 6.32. Initializing sequence

6.4.44 CC1020TxMode()

When data is ready from the OBC, the OBRF's MCU need to instruct the transceiver to use the TX mode, where the transceiver provides a clock signal in which the MCU will synchronize on it and the MCU provides the bit of the stream to the DIO pin, on the falling edge of DCLK. According to the previous analysis of the parameters, the optimum register values are derived from SmartRF Studio. The initialization starts from the power off assumption, in order to discard the previous values on registers, which can be affected by SEUs events, as seen for RX mode.

Initialization steps

The steps to use this mode are made by a sequence of commands in order to reset the transceiver. Then configuring the TX parameters (associated to transceiver's configuration registers labelled as B) in order to obtain the characteristics shown in previous chapters. The configuration follow different cases for different baudrates, since each require a fine tuning of the internal RF components and a well determined bandwidth (as seen before in RX mode). Up to the calibration step, the configurations are the same for the RX mode, except a different bit indicating the usage of chip module B. Then, the chip is put in full power up (PD mode 0), activating it in the TX mode, and the transceiver starts transmitting data at DIO. To avoid dummy bits, the DIO should be driven as soon as possible from the MCU, therefore after the final initialization there are no other instructions. [13]

This method implements the transceiver initialization in TX mode mentioned in section 6.4.44. This method can be used for different configurations by using the proper parameters: **baudr : ulong** for the chosen baudrate, **freq : AllowedFrequencies** for the used carrier frequency, **mod : t_modulation** for the selected RF modulation and **txpower : ushort** for the intensity of transmitted power. The MCU's pins connected to the transceiver are initialized according to figures 6.9 and 6.8. Then it is reset, preparing it for the programming.

The switch-case implementation is the same for the *CC1020AutoWakeUpMode()* and code is different on the final part only. As for RX initialization, the procedure starts from the assumption of power off, due to the SEU considerations made on the auto wake-up mode, so the sequence follows the flowchart in figure 6.32. After the various registers configurations already made for the RX mode, is followed the sequence labelled WakeUpCC1020ToTX in figure 6.32, turning on the crystal, bias generator and synthesizer. Then is calibrated as for the RX mode. Now, the procedure labelled SetupCC1020PD in figure 6.32 is skipped and since after the calibration the PLL is in lock, the TX mode can be directly activated by putting the CC1020 in PD mode 0, which means that is

in full power up, connected to the internal PA and activated for the TX mode. This configuration uses the on-chip configuration labelled B for the TX mode, writing PDMODE0_TX_B value in MAIN register. The chip now starts to provide a clock signal on the DCLK pin and reads the value on DIO.

Code:

```
Bk1B31A2S::CC1020TxMode(ulong baud, Use_Cases::AllowedFrequencies freq,
Bk1B31A2W_OBRF_437MHz::t_modulation modulation, ushort txpower) {
//init CC to PD.  The CC1020 class uses  bitbanging, NOT SPI (is compatible).
 PSEL is high
transceiver.CC1020_Init(); //PSEL high
// From AN, first reset
transceiver.SetReg(CC1020_MAIN, MAIN_RESET);
transceiver.SetReg(CC1020_MAIN, MAIN_OUT_RESET); //out of reset
//sequence from RF Studio. Now is PD (mode is described in documentation
 of this class)
// configuration
ulong dev = 0;
ulong bw = 0;
//consigliati per bandw con doppler, quindi VGAx giÃ  regolati
// cambiare i VGA se sul campo non funziona correttamente (che sia sensitivitÃ ,
 selettivitÃ , ecc)
switch (baudr) {

case 2400: //25, ma dev'essere 50
transceiver.SetReg(CC1020_INTERFACE, (XOSC_BYPASS | SEP_TX_RX |
NOGATE_DCLK_PLL | GATE_DCLK_CS | NO_PA | NO_LNA | PA_LOW | LNA_LOW));
transceiver.SetReg(CC1020_RESET, 0xff);
transceiver.SetReg(CC1020_SEQUENCING, (PSEL_TOGGLE | WAIT_32ADC | MAXCS_WAIT));
//transceiver.SetReg(CC1020_FREQ_2A, 0x3a);
//transceiver.SetReg(CC1020_FREQ_1A, 0x7a);
//transceiver.SetReg(CC1020_FREQ_0A, 0xf1);
transceiver.SetReg(CC1020_CLOCK_A, CONF2400);
//transceiver.SetReg(CC1020_FREQ_2B, 0x3a);
//transceiver.SetReg(CC1020_FREQ_1B, 0x85);
//transceiver.SetReg(CC1020_FREQ_0B, 0x9d);
transceiver.SetReg(CC1020_CLOCK_B, CONF2400);
transceiver.SetReg(CC1020_VCO, (VCO_CURR_2_8_A | VCO_CURR_2_8_B));
//transceiver.SetReg(CC1020_MODEM, 0x50);
//transceiver.SetReg(CC1020_DEVIATION, 0x09);
//transceiver.SetReg(CC1020_AFC_CONTROL, 0xca);
//transceiver.SetReg(CC1020_FILTER, 0x3f);
transceiver.SetReg(CC1020_VGA1, (FREEZE_32ADC | WAIT_16FCLK |
CS_SIG_RESET2CY | CS_SIG_SET2CY));
transceiver.SetReg(CC1020_VGA2, (AGC_AVG_4CY | HYSTER_GAIN |
AGC_ON | LNA_SETTING | MAX_LNA | MIN_LNA));
transceiver.SetReg(CC1020_VGA3, (VGA_MAX_GAIN | VGA_DOWN));
transceiver.SetReg(CC1020_VGA4, (CS_LEVEL | VGA_UP));
```

```
transceiver.SetReg(CC1020_LOCK, (LOCK_CONTINUOUS | LOCK_WINDOW
 | LOCK_RESTART_MODE | LOCK_ACCUR));
transceiver.SetReg(CC1020_FRONTEND, (LNAMIX_CURRENT, LNA_CURRENT,
 MIX_CURRENT, LNA2_CURRENT, SDC_CURRENT, LNAMIX__BIAS));
transceiver.SetReg(CC1020_ANALOG, (BAND0 | LOSC_DC | BLANK_OFF
| PHASE_SHORT_DELAY | PA_BOOST | DIV_BUFF_CURRENT));
transceiver.SetReg(CC1020_BUFF_SWING, (PRE_SWING | RX_SWING | TX_SWING));
transceiver.SetReg(CC1020_BUFF_CURRENT, (PRE_CURR | RX_CURR | TX_CURR));
transceiver.SetReg(CC1020_PLL_BW, PLL_BW);
transceiver.SetReg(CC1020_CALIBRATE, (NO_CAL | CAL_WAIT
| CAL_SINGLE | CAL_ITER));
//transceiver.SetReg(CC1020_PA_POWER, paPower);
transceiver.SetReg(CC1020_MATCH, 0x00);
transceiver.SetReg(CC1020_PHASE_COMP, 0x00);
transceiver.SetReg(CC1020_GAIN_COMP, 0x00);
transceiver.SetReg(CC1020_POWERDOWN, 0x00);
dev = 2025;
bw = 25000;
break;

case 4800: //50kHz
transceiver.SetReg(CC1020_INTERFACE, (XOSC_BYPASS | SEP_TX_RX |
NOGATE_DCLK_PLL | GATE_DCLK_CS | NO_PA | NO_LNA | PA_LOW | LNA_LOW));
transceiver.SetReg(CC1020_RESET, 0xff);
transceiver.SetReg(CC1020_SEQUENCING, (PSEL_TOGGLE | WAIT_32ADC | MAXCS_WAIT));
//transceiver.SetReg(CC1020_FREQ_2A, 0x3a);
//transceiver.SetReg(CC1020_FREQ_1A, 0x7a);
//transceiver.SetReg(CC1020_FREQ_0A, 0xf1);
transceiver.SetReg(CC1020_CLOCK_A, CONF4800);
//transceiver.SetReg(CC1020_FREQ_2B, 0x3a);
//transceiver.SetReg(CC1020_FREQ_1B, 0x85);
//transceiver.SetReg(CC1020_FREQ_0B, 0x9d);
transceiver.SetReg(CC1020_CLOCK_B, CONF4800);
transceiver.SetReg(CC1020_VCO, (VCO_CURR_2_8_A | VCO_CURR_2_8_B));
//transceiver.SetReg(CC1020_MODEM, 0x50);
//transceiver.SetReg(CC1020_DEVIATION, 0x09);
//transceiver.SetReg(CC1020_AFC_CONTROL, 0xca);
//transceiver.SetReg(CC1020_FILTER, 0x3f);
transceiver.SetReg(CC1020_VGA1, (FREEZE_32ADC | WAIT_16FCLK
| CS_SIG_RESET2CY | CS_SIG_SET2CY));
transceiver.SetReg(CC1020_VGA2, (AGC_AVG_4CY | HYSTER_GAIN
| AGC_ON | LNA_SETTING | MAX_LNA | MIN_LNA));
transceiver.SetReg(CC1020_VGA3, (VGA_MAX_GAIN1 | VGA_DOWN));
transceiver.SetReg(CC1020_VGA4, (CS_LEVEL1 | VGA_UP));
transceiver.SetReg(CC1020_LOCK, (LOCK_CONTINUOUS | LOCK_WINDOW
 | LOCK_RESTART_MODE | LOCK_ACCUR));
transceiver.SetReg(CC1020_FRONTEND, (LNAMIX_CURRENT, LNA_CURRENT,
 MIX_CURRENT, LNA2_CURRENT, SDC_CURRENT, LNAMIX__BIAS));
transceiver.SetReg(CC1020_ANALOG, (BAND0 | LOSC_DC | BLANK_OFF
| PHASE_SHORT_DELAY | PA_BOOST | DIV_BUFF_CURRENT));
```

```
transceiver.SetReg(CC1020_BUFF_SWING, (PRE_SWING | RX_SWING | TX_SWING));
transceiver.SetReg(CC1020_BUFF_CURRENT, (PRE_CURR | RX_CURR | TX_CURR));
transceiver.SetReg(CC1020_PLL_BW, PLL_BW);
transceiver.SetReg(CC1020_CALIBRATE, (NO_CAL | CAL_WAIT
| CAL_SINGLE | CAL_ITER));
//transceiver.SetReg(CC1020_PA_POWER, paPower);
transceiver.SetReg(CC1020_MATCH, 0x00);
transceiver.SetReg(CC1020_PHASE_COMP, 0x00);
transceiver.SetReg(CC1020_GAIN_COMP, 0x00);
transceiver.SetReg(CC1020_POWERDOWN, 0x00);
dev = 2475;
bw = 50000;
break;

case 9600: //50kHz
transceiver.SetReg(CC1020_INTERFACE, (XOSC_BYPASS | SEP_TX_RX
| NOGATE_DCLK_PLL | GATE_DCLK_CS | NO_PA | NO_LNA | PA_LOW | LNA_LOW));
transceiver.SetReg(CC1020_RESET, 0xff);
transceiver.SetReg(CC1020_SEQUENCING, (PSEL_TOGGLE | WAIT_32ADC
| MAXCS_WAIT));
//transceiver.SetReg(CC1020_FREQ_2A, 0x3a);
//transceiver.SetReg(CC1020_FREQ_1A, 0x7a);
//transceiver.SetReg(CC1020_FREQ_0A, 0xf1);
transceiver.SetReg(CC1020_CLOCK_A, CONF9600);
//transceiver.SetReg(CC1020_FREQ_2B, 0x3a);
//transceiver.SetReg(CC1020_FREQ_1B, 0x85);
//transceiver.SetReg(CC1020_FREQ_0B, 0x9d);
transceiver.SetReg(CC1020_CLOCK_B, CONF9600);
transceiver.SetReg(CC1020_VCO, (VCO_CURR_2_8_A | VCO_CURR_2_8_B));
//transceiver.SetReg(CC1020_MODEM, 0x50);
//transceiver.SetReg(CC1020_DEVIATION, 0x09);
//transceiver.SetReg(CC1020_AFC_CONTROL, 0xca);
//transceiver.SetReg(CC1020_FILTER, 0x3f);
transceiver.SetReg(CC1020_VGA1, (FREEZE_32ADC | WAIT_16FCLK
| CS_SIG_RESET2CY | CS_SIG_SET2CY));
transceiver.SetReg(CC1020_VGA2, (AGC_AVG_4CY | HYSTER_GAIN
| AGC_ON | LNA_SETTING | MAX_LNA | MIN_LNA));
transceiver.SetReg(CC1020_VGA3, (VGA_MAX_GAIN1 | VGA_DOWN));
transceiver.SetReg(CC1020_VGA4, (CS_LEVEL1 | VGA_UP));
transceiver.SetReg(CC1020_LOCK, (LOCK_CONTINUOUS | LOCK_WINDOW
 | LOCK_RESTART_MODE | LOCK_ACCUR));
transceiver.SetReg(CC1020_FRONTEND, (LNAMIX_CURRENT, LNA_CURRENT,
 MIX_CURRENT, LNA2_CURRENT, SDC_CURRENT, LNAMIX__BIAS));
transceiver.SetReg(CC1020_ANALOG, (BAND0 | LOSC_DC | BLANK_OFF
 | PHASE_SHORT_DELAY | PA_BOOST | DIV_BUFF_CURRENT));
transceiver.SetReg(CC1020_BUFF_SWING, (PRE_SWING | RX_SWING | TX_SWING));
transceiver.SetReg(CC1020_BUFF_CURRENT, (PRE_CURR | RX_CURR | TX_CURR));
transceiver.SetReg(CC1020_PLL_BW, PLL_BW);
transceiver.SetReg(CC1020_CALIBRATE, (NO_CAL | CAL_WAIT
 | CAL_SINGLE | CAL_ITER));
```

```
//transceiver.SetReg(CC1020_PA_POWER, paPower);
transceiver.SetReg(CC1020_MATCH, 0x00);
transceiver.SetReg(CC1020_PHASE_COMP, 0x00);
transceiver.SetReg(CC1020_GAIN_COMP, 0x00);
transceiver.SetReg(CC1020_POWERDOWN, 0x00);
dev = 4950;
bw = 50000;
break;

case 19200: //100khz BW
transceiver.SetReg(CC1020_INTERFACE, (XOSC_BYPASS | SEP_TX_RX
 | NOGATE_DCLK_PLL | GATE_DCLK_CS | NO_PA | NO_LNA | PA_LOW | LNA_LOW));
transceiver.SetReg(CC1020_RESET, 0xff);
transceiver.SetReg(CC1020_SEQUENCING, (PSEL_TOGGLE | WAIT_32ADC | MAXCS_WAIT));
//transceiver.SetReg(CC1020_FREQ_2A, 0x3a);
//transceiver.SetReg(CC1020_FREQ_1A, 0x7a);
//transceiver.SetReg(CC1020_FREQ_0A, 0xf1);
transceiver.SetReg(CC1020_CLOCK_A, CONF19200);
//transceiver.SetReg(CC1020_FREQ_2B, 0x3a);
//transceiver.SetReg(CC1020_FREQ_1B, 0x85);
//transceiver.SetReg(CC1020_FREQ_0B, 0x9d);
transceiver.SetReg(CC1020_CLOCK_B, CONF19200);
transceiver.SetReg(CC1020_VCO, (VCO_CURR_2_8_A | VCO_CURR_2_8_B));
//transceiver.SetReg(CC1020_MODEM, 0x50);
//transceiver.SetReg(CC1020_DEVIATION, 0x09);
//transceiver.SetReg(CC1020_AFC_CONTROL, 0xca);
//transceiver.SetReg(CC1020_FILTER, 0x3f);
transceiver.SetReg(CC1020_VGA1, (FREEZE_32ADC | WAIT_16FCLK
| CS_SIG_RESET2CY | CS_SIG_SET2CY));
transceiver.SetReg(CC1020_VGA2, (AGC_AVG_4CY | HYSTER_GAIN
| AGC_ON | LNA_SETTING | MAX_LNA | MIN_LNA));
transceiver.SetReg(CC1020_VGA3, (VGA_MAX_GAIN2 | VGA_DOWN));
transceiver.SetReg(CC1020_VGA4, (CS_LEVEL1 | VGA_UP));
transceiver.SetReg(CC1020_LOCK, (LOCK_CONTINUOUS | LOCK_WINDOW
| LOCK_RESTART_MODE | LOCK_ACCUR));
transceiver.SetReg(CC1020_FRONTEND, (LNAMIX_CURRENT, LNA_CURRENT,
 MIX_CURRENT, LNA2_CURRENT, SDC_CURRENT, LNAMIX__BIAS));
transceiver.SetReg(CC1020_ANALOG, (BAND0 | LOSC_DC | BLANK_OFF
| PHASE_SHORT_DELAY | PA_BOOST | DIV_BUFF_CURRENT));
transceiver.SetReg(CC1020_BUFF_SWING, (PRE_SWING | RX_SWING | TX_SWING));
transceiver.SetReg(CC1020_BUFF_CURRENT, (PRE_CURR | RX_CURR | TX_CURR));
transceiver.SetReg(CC1020_PLL_BW, PLL_BW);
transceiver.SetReg(CC1020_CALIBRATE, (NO_CAL | CAL_WAIT
| CAL_SINGLE | CAL_ITER));
//transceiver.SetReg(CC1020_PA_POWER, paPower);
transceiver.SetReg(CC1020_MATCH, 0x00);
transceiver.SetReg(CC1020_PHASE_COMP, 0x00);
transceiver.SetReg(CC1020_GAIN_COMP, 0x00);
transceiver.SetReg(CC1020_POWERDOWN, 0x00);
dev = 9900;
```

```
bw = 100000;
break;

case 38400: //150
transceiver.SetReg(CC1020_INTERFACE, (XOSC_BYPASS | SEP_TX_RX
 | NOGATE_DCLK_PLL | GATE_DCLK_CS | NO_PA | NO_LNA | PA_LOW | LNA_LOW));
transceiver.SetReg(CC1020_RESET, 0xff);
transceiver.SetReg(CC1020_SEQUENCING, (PSEL_TOGGLE | WAIT_32ADC | MAXCS_WAIT));
//transceiver.SetReg(CC1020_FREQ_2A, 0x3a);
//transceiver.SetReg(CC1020_FREQ_1A, 0x7a);
//transceiver.SetReg(CC1020_FREQ_0A, 0xf1);
transceiver.SetReg(CC1020_CLOCK_A, CONF38400);
//transceiver.SetReg(CC1020_FREQ_2B, 0x3a);
//transceiver.SetReg(CC1020_FREQ_1B, 0x85);
//transceiver.SetReg(CC1020_FREQ_0B, 0x9d);
transceiver.SetReg(CC1020_CLOCK_B, CONF38400);
transceiver.SetReg(CC1020_VCO, (VCO_CURR_2_8_A | VCO_CURR_2_8_B));
//transceiver.SetReg(CC1020_MODEM, 0x50);
//transceiver.SetReg(CC1020_DEVIATION, 0x09);
//transceiver.SetReg(CC1020_AFC_CONTROL, 0xca);
//transceiver.SetReg(CC1020_FILTER, 0x3f);
transceiver.SetReg(CC1020_VGA1, (FREEZE_32ADC | WAIT_16FCLK
| CS_SIG_RESET2CY | CS_SIG_SET2CY));
transceiver.SetReg(CC1020_VGA2, (AGC_AVG_4CY | HYSTER_GAIN
 | AGC_ON | LNA_SETTING | MAX_LNA | MIN_LNA));
transceiver.SetReg(CC1020_VGA3, (VGA_MAX_GAIN | VGA_DOWN));
transceiver.SetReg(CC1020_VGA4, (CS_LEVEL3 | VGA_UP));
transceiver.SetReg(CC1020_LOCK, (LOCK_CONTINUOUS | LOCK_WINDOW
 | LOCK_RESTART_MODE | LOCK_ACCUR));
transceiver.SetReg(CC1020_FRONTEND, (LNAMIX_CURRENT, LNA_CURRENT,
 MIX_CURRENT, LNA2_CURRENT, SDC_CURRENT, LNAMIX__BIAS));
transceiver.SetReg(CC1020_ANALOG, (BAND0 | LOSC_DC | BLANK_OFF
 | PHASE_SHORT_DELAY | PA_BOOST | DIV_BUFF_CURRENT));
transceiver.SetReg(CC1020_BUFF_SWING, (PRE_SWING | RX_SWING | TX_SWING));
transceiver.SetReg(CC1020_BUFF_CURRENT, (PRE_CURR | RX_CURR | TX_CURR));
transceiver.SetReg(CC1020_PLL_BW, PLL_BW);
transceiver.SetReg(CC1020_CALIBRATE, (NO_CAL | CAL_WAIT
 | CAL_SINGLE | CAL_ITER));
//transceiver.SetReg(CC1020_PA_POWER, paPower);
transceiver.SetReg(CC1020_MATCH, 0x00);
transceiver.SetReg(CC1020_PHASE_COMP, 0x00);
transceiver.SetReg(CC1020_GAIN_COMP, 0x00);
transceiver.SetReg(CC1020_POWERDOWN, 0x00);
dev = 19800;
bw = 150000;
break;

case 76800: //200
transceiver.SetReg(CC1020_INTERFACE, (XOSC_BYPASS | SEP_TX_RX
 | NOGATE_DCLK_PLL | GATE_DCLK_CS | NO_PA | NO_LNA | PA_LOW | LNA_LOW));
```

```
transceiver.SetReg(CC1020_RESET, 0xff);
transceiver.SetReg(CC1020_SEQUENCING, (PSEL_TOGGLE | WAIT_32ADC | MAXCS_WAIT));
//transceiver.SetReg(CC1020_FREQ_2A, 0x3a);
//transceiver.SetReg(CC1020_FREQ_1A, 0x7a);
//transceiver.SetReg(CC1020_FREQ_0A, 0xf1);
transceiver.SetReg(CC1020_CLOCK_A, CONF76800);
//transceiver.SetReg(CC1020_FREQ_2B, 0x3a);
//transceiver.SetReg(CC1020_FREQ_1B, 0x85);
//transceiver.SetReg(CC1020_FREQ_0B, 0x9d);
transceiver.SetReg(CC1020_CLOCK_B, CONF76800);
transceiver.SetReg(CC1020_VCO, (VCO_CURR_2_8_A | VCO_CURR_2_8_B));
//transceiver.SetReg(CC1020_MODEM, 0x50);
//transceiver.SetReg(CC1020_DEVIATION, 0x09);
//transceiver.SetReg(CC1020_AFC_CONTROL, 0xca);
//transceiver.SetReg(CC1020_FILTER, 0x3f);
transceiver.SetReg(CC1020_VGA1, (FREEZE_32ADC | WAIT_16FCLK |
CS_SIG_RESET2CY | CS_SIG_SET2CY));
transceiver.SetReg(CC1020_VGA2, (AGC_AVG_4CY | HYSTER_GAIN |
AGC_ON | LNA_SETTING | MAX_LNA | MIN_LNA));
transceiver.SetReg(CC1020_VGA3, (VGA_MAX_GAIN3 | VGA_DOWN));
transceiver.SetReg(CC1020_VGA4, (CS_LEVEL2 | VGA_UP));
transceiver.SetReg(CC1020_LOCK, (LOCK_CONTINUOUS | LOCK_WINDOW
| LOCK_RESTART_MODE | LOCK_ACCUR));
transceiver.SetReg(CC1020_FRONTEND, (LNAMIX_CURRENT, LNA_CURRENT,
 MIX_CURRENT, LNA2_CURRENT, SDC_CURRENT, LNAMIX__BIAS));
transceiver.SetReg(CC1020_ANALOG, (BAND0 | LOSC_DC | BLANK_OFF
| PHASE_SHORT_DELAY | PA_BOOST | DIV_BUFF_CURRENT));
transceiver.SetReg(CC1020_BUFF_SWING, (PRE_SWING | RX_SWING | TX_SWING));
transceiver.SetReg(CC1020_BUFF_CURRENT, (PRE_CURR | RX_CURR | TX_CURR));
transceiver.SetReg(CC1020_PLL_BW, PLL_BW);
transceiver.SetReg(CC1020_CALIBRATE, (NO_CAL | CAL_WAIT |
 CAL_SINGLE | CAL_ITER));
//transceiver.SetReg(CC1020_PA_POWER, paPower);
transceiver.SetReg(CC1020_MATCH, 0x00);
transceiver.SetReg(CC1020_PHASE_COMP, 0x00);
transceiver.SetReg(CC1020_GAIN_COMP, 0x00);
transceiver.SetReg(CC1020_POWERDOWN, 0x00);
dev = 36000;
bw = 200000;
break;

case 153600: //500
transceiver.SetReg(CC1020_INTERFACE, (XOSC_BYPASS | SEP_TX_RX
| NOGATE_DCLK_PLL | GATE_DCLK_CS | NO_PA | NO_LNA | PA_LOW | LNA_LOW));
transceiver.SetReg(CC1020_RESET, 0xff);
transceiver.SetReg(CC1020_SEQUENCING, (PSEL_TOGGLE | WAIT_32ADC
 | MAXCS_WAIT));
//transceiver.SetReg(CC1020_FREQ_2A, 0x3a);
//transceiver.SetReg(CC1020_FREQ_1A, 0x7a);
//transceiver.SetReg(CC1020_FREQ_0A, 0xf1);
```

```
transceiver.SetReg(CC1020_CLOCK_A, CONF153600);
//transceiver.SetReg(CC1020_FREQ_2B, 0x3a);
//transceiver.SetReg(CC1020_FREQ_1B, 0x85);
//transceiver.SetReg(CC1020_FREQ_0B, 0x9d);
transceiver.SetReg(CC1020_CLOCK_B, CONF153600);
transceiver.SetReg(CC1020_VCO, (VCO_CURR_2_8_A | VCO_CURR_2_8_B));
//transceiver.SetReg(CC1020_MODEM, 0x50);
//transceiver.SetReg(CC1020_DEVIATION, 0x09);
//transceiver.SetReg(CC1020_AFC_CONTROL, 0xca);
//transceiver.SetReg(CC1020_FILTER, 0x3f);
transceiver.SetReg(CC1020_VGA1, (FREEZE_32ADC | WAIT_16FCLK
 | CS_SIG_RESET2CY | CS_SIG_SET2CY));
transceiver.SetReg(CC1020_VGA2, (AGC_AVG_4CY | HYSTER_GAIN
| AGC_ON | LNA_SETTING | MAX_LNA | MIN_LNA));
transceiver.SetReg(CC1020_VGA3, (VGA_MAX_GAIN4 | VGA_DOWN));
transceiver.SetReg(CC1020_VGA4, (CS_LEVEL4 | VGA_UP));
transceiver.SetReg(CC1020_LOCK, (LOCK_CONTINUOUS | LOCK_WINDOW
| LOCK_RESTART_MODE | LOCK_ACCUR));
transceiver.SetReg(CC1020_FRONTEND, (LNAMIX_CURRENT, LNA_CURRENT,
 MIX_CURRENT, LNA2_CURRENT, SDC_CURRENT, LNAMIX__BIAS));
transceiver.SetReg(CC1020_ANALOG, (BAND0 | LOSC_DC | BLANK_OFF |
 PHASE_SHORT_DELAY | PA_BOOST | DIV_BUFF_CURRENT));
transceiver.SetReg(CC1020_BUFF_SWING, (PRE_SWING | RX_SWING | TX_SWING));
transceiver.SetReg(CC1020_BUFF_CURRENT, (PRE_CURR | RX_CURR | TX_CURR));
transceiver.SetReg(CC1020_PLL_BW, PLL_BW);
transceiver.SetReg(CC1020_CALIBRATE, (NO_CAL | CAL_WAIT
| CAL_SINGLE | CAL_ITER));
//transceiver.SetReg(CC1020_PA_POWER, paPower);
transceiver.SetReg(CC1020_MATCH, 0x00);
transceiver.SetReg(CC1020_PHASE_COMP, 0x00);
transceiver.SetReg(CC1020_GAIN_COMP, 0x00);
transceiver.SetReg(CC1020_POWERDOWN, 0x00);
dev = 72000;
bw = 500000;
break;

default:
//aHK::configRegister[1] |= (ushort)(7 & MASK_CS_BAUDRATE);
break;
}

transceiver.SetFreqA(freq, REF_DIV); //ref div = 2
transceiver.SetFreqB(freq, REF_DIV);
//transceiver.SetReg(CC1020_FREQ_2A, 0x3a);
//transceiver.SetReg(CC1020_FREQ_1A, 0x7a);
//transceiver.SetReg(CC1020_FREQ_0A, 0xf1);
//transceiver.SetReg(CC1020_FREQ_2B, 0x3a);
//transceiver.SetReg(CC1020_FREQ_1B, 0x85);
//transceiver.SetReg(CC1020_FREQ_0B, 0x9d);
```

```
transceiver.Modem(F_ADC, NO_SCRAMB, NRZ); //fadc 1.2288,
no scrambling 0, NRZ (1-1 = 00)
//transceiver.SetReg(CC1020_MODEM, 0x50);


transceiver.Deviation((ushort)mod, dev);
//transceiver.SetReg(CC1020_DEVIATION, 0x09);

transceiver.AFC_control(CC_SETTLING, dev); // settling max (3)
//transceiver.SetReg(CC1020_AFC_CONTROL, 0xca);

transceiver.FilterBandWidth(bw, baudr);
//transceiver.SetReg(CC1020_FILTER, 0x3f);

transceiver.SetReg(CC1020_MAIN, TX_B_PDMODE1_XOSC);  //now it
is in PD mode 1, xosc on
transceiver.WAIT_CYCLE();
transceiver.SetReg(CC1020_MAIN, TX_B_PDMODE1_XOSC_BIAS);  //now
 it is in PD mode 1, bias on
transceiver.WAIT_CYCLE();  // wait. see p.55 datasheet
transceiver.SetReg(CC1020_MAIN, TX_B_PDMODE1_ON);  //now it is
in PD mode 1, synth on, FULL ON
transceiver.CC1020.SetReg(CC1020_PA_POWER, 0x00); // no spurs
CC1020Calibrate(); // AN070
transceiver.CC1020.SetReg(CC1020_MAIN, PDMODE0_TX_B); //powerdown mode 0
transceiver.CC1020.SetReg(CC1020_PA_POWER, txpower);
}
```

6.4.45 CC1020Calibrate()

At every initialization of the CC1020, is performed an internal calibration of VCO and PLL. Is aimed to compensate for supply voltage, temperature and process variations. It is activated by writing the value CAL on the CALIBRATE register, as implemented in the *CC1020Calibrate()* method. Once started the calibration is performed automatically and sets the maximum VCO tuning range and optimum charge pump current for PLL stability. The calibration result is stored internally in the chip, and is valid as long as power is not turned off. To prevent SEUs and compensate for temperature variation, the calibration is performed at every initialisation made.

The calibration starts by writing the bit CAL in CALIBRATE register. After waiting at least 100us, is polled the STATUS register for the CAL_COMPLETE bit to be set, indicating the calibration complete. If the register is polled more than CAL_TIMEOUT times without success, the loop break and do not block the OBRF firmware main execution. Modifying at compile-time the CAL_TIMEOUT, the maximum time of this polling can be adjusted.

Assuming the calibration is successful, is then polled the LOCK_CONTINUOUS bit, indicating that the PLL is locked and stable. The lock signal accuracy is set in LOCK register in *CC1020TxMode()* and *CC1020AutoWakeUpMode()* which are using the calibrate method. Also here there is a limit on the polling, with the LOCK_TIMEOUT parameter.

At this point if the PLL is not stable locked, the calibration restart. This outer loop will be broken if more than CAL_ATTEMPT_MAX calibrations fails. Finally, the PLL continuous lock bit is returned by the method. [13]

Code:

```
bool Bk1B31A2S::CC1020Calibrate() {
//calibrate for the active register (A or B). So not dual.
// Calibrate, and re-calibrate if necessary:

for (nCalAttempt = CAL_ATTEMPT_MAX; (nCalAttempt>0); nCalAttempt--) {
transceiver.SetReg(CC1020_CALIBRATE, (CAL | CAL_WAIT | CAL_SINGLE | CAL_ITER));
transceiver.WAIT_CYCLE();

for(TimeOutCounter=CAL_TIMEOUT; ((transceiver.ReadReg(CC1020_STATUS)&CAL_COMPLETE)
==0x00)&&(TimeOutCounter>0); TimeOutCounter--); // wait for cal

for(TimeOutCounter=LOCK_TIMEOUT; ((transceiver.ReadReg(CC1020_STATUS)&LOCKED_CONTINUOUSLY)
==0x00)&&(TimeOutCounter>0); TimeOutCounter--); // wait to lock after cal

// Abort further recalibration attempts if successful LOCK
if((transceiver.ReadReg(CC1020_STATUS)&0x10) == 0x10) {
break;
}
```

```
}
// Return state of LOCK_CONTINUOUS bit
return ((bool)(transceiver.ReadReg(CC1020_STATUS)
&LOCKED_CONTINUOUSLY)==LOCKED_CONTINUOUSLY);
}
```

Chapter 7

Tile Layout

In this chapter will be shown the physical placement of the OBRF on a CubeSat tile. It consist in the transfer from schematics to a complete PCB which will fit the allowed tile's space. This 1B31A OBRF 437MHz must be mounted in the same tile with the 1B31B OBRF 2.4GHz and half of the tile space is reserved for each design. The starting point of the tile is shown in figure 7.1, where there is room for the UHF section. This space is less than the half tile, but since there will be a further design, correction and integration of the SHF band hardware, it is assumed to have almost half of space available. It is used the tool Mentor Graphics Expedition PCB to design and generate the manufacturing data and GC-Prevue for gerber analysis. Then the RF circuit design has been supported using the AWR TxLine tool.

7.1 Placement criteria

Since in space there is only thermal dissipation via conduction and radiating, thermal considerations are not trivial. The CubeSat structure provide thermal absorption only with four screws, that can be seen at the four edges in figure 7.1. The main heat sources are the power supply and the switching regulator. The PCB that will be used is composed by 4 layers shown in figure 7.2, while using vias will double the external copper thickness during the manufacturing process, extending the external layers to 35μm. In order to spread the heat as much as possible, are placed thermal vias under the critical components, through all the 4 layers and keeping them connected with the ground planes places in all layers.

The thermal path on the copper should not be interrupted otherwise the thermal resistance in the path will increase, being only the FR4 with an higher thermal resistivity (copper ρ =

Figure 7.1. The implementation of half tile with the 1B31B OBRF and the avilable space for the UHF module

$0.00256m \cdot K \cdot W^{-1}$, FR-4 $\rho = 2.9m \cdot K \cdot W^{-1}$). A thermal resistance approximation is given by the formula 7.1, in accordance with figure 7.3:

$$\theta_{th} = \frac{L}{K \cdot S} \tag{7.1}$$

where K is the thermal conductivity, the inverse of thermal resistivity shown above for the copper and FR-4; L is the length of the path and $S = L \cdot d$ is the cross sectional area of the copper on PCB, being L the same value of the length (it is considered a square of copper plane shape) and $d = 35\mu m$ is the copper thickness. For a single layer, on copper, $\theta_{layer} = 72\frac{^\circ K}{W}$. The power

copper - 1	18µm	½oz
Prepreg 7628	180µm	7mil
Prepreg 7628	180µm	7mil
copper - 2	35µm	1oz
Core	710µm	27.95mil
copper - 3	35µm	1oz
Prepreg 7628	180µm	7mil
Prepreg 7628	180µm	7mil
copper - 4	18µm	½oz

Figure 7.2. PCB stack-up adopted

amplifier on both sides presents interruptions on the copper path, due to components or to a NC pins; thermal considerations are then made on 3 layers only.

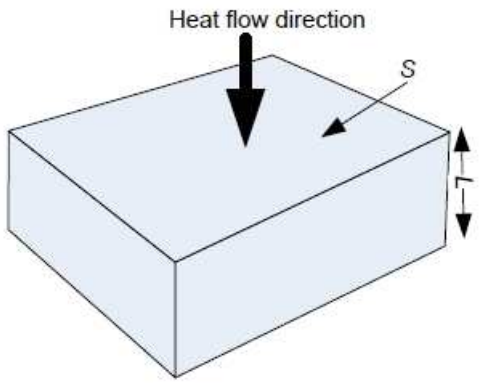

Figure 7.3. Model of PCB copper trace, with cross section S and length L; not in scale

As thermal pads are used 16 vias with 0.125mm internal radius and 0.3 mm external one and their length is about 1.5mm, gold plated ($K = 320m^{-1} \cdot K^{-1} \cdot W$). The total thermal resistance of vias is:

$$\theta_{vias} = \frac{1.5mm}{320m^{-1} \cdot K^{-1} \cdot W \cdot 16 \cdot \pi(0.3^2mm^2 - 0.125^2mm^2)} \approx 1.25°K/W \tag{7.2}$$

Since the layers are 3, the total resistance is their parallel equivalent, obtaining $\theta_{total\_layer} = 24\frac{^\circ K}{W}$. Note that the external radius is the minium distance between the square which contain the via and the via's radius, because the thermal pad is a bigger square composed of 16 of these sub-squares containing hole via, see figure 7.4. The total thermal resistance for the power amplifier is therefore

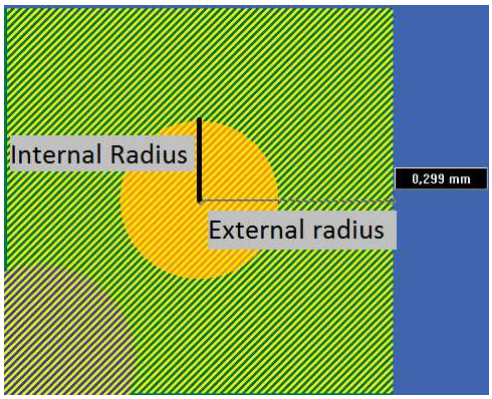

Figure 7.4. Single thermal via placed under the power amplifier. Center yellow circle is an hole, the green square outside is gold plated (worst thermal case).

the parallel of the two above:

$$\theta_{total\_PA} = \frac{1}{\theta^{-1}_{total\_layer} + \theta^{-1}_{vias}} \approx 6^\circ K/W \tag{7.3}$$

The TPS5450 will use 8 different vias, with difference between internal and external circle area difference of $0.18mm^2$. This bring to a:

$$\theta_{vias\_TPS} = \frac{1.5mm}{320m^{-1} \cdot K^{-1} \cdot W \cdot 8 \cdot \pi(0.18mm)^2} \approx 1.1^\circ K/W \tag{7.4}$$

The total is:

$$\theta_{total\_TPS} = \frac{1}{\theta^{-1}_{total\_layer} + \theta^{-1}_{vias_TPS}} \approx 2.6^\circ K/W \tag{7.5}$$

where the switching efficiency is grater than 85%.

The power amplifier RF6886 works with a maximum efficiency of $\eta = 53\%$ at 35dBm ($P_{out} = 3.16W$) of output RF power. This means a heat dissipation of:

$$P_d = P_{out} \cdot \left(\frac{1-\eta}{\eta}\right) = 2.8W \tag{7.6}$$

The RF switch, provides 0.4dB of insertion loss. This is the 10% of the maximum transmitted power, dissipating 300mW on the ground plane. Here there are few vias, but there are no thermal vias underneath the switch. The thermal resistance is more near to 70 °K/W. [15][16]

As a conclusion, these thermal considerations were made neglecting the radiated heat and the conductivity of the FR-4, in order to understand roughly the upper thermal bounds; in order to

achieve the worst conditions, the first ground plane is not considered, but in practice it is expected that will help in reducing the thermal resistance. With these assumptions, some further verification on board testing should be perfomed. The final placement decision is then to put the RF and power supply components on the edges, while the low-power digital parts kept in middle.

7.2 Traces

The main power source is provided by the PDB pins, which are feeding the switching regulator in which provides in output 3.1V at 3A peak. From figure 7.5 are derived the trace widths, chosen where looking for the lowest possible temperature increase. Power amplifier traces are then chosen to be from 1 to 2 mm. The others are not an issue.

The RF traces are treated separately, in order to obtain 50Ω of characteristic impedance. The gerbers of the reference design of CC1020 has been analysed and measured with GC-Prevue, considering the reference evaulation board stackup. These values are derived using AWR TxLine tool, where with that reference was of a microstrip type (figure 7.6). All the RF nets connected to the chip and used to connect all the matching components are shown to have a line impedance of 80Ω, while, as suggested by the datasheet, the lines connected directly to the antenna are 50Ω, proving the correctness of the analysis.

The power amplifier analysis was not so trivial, because the high RF output power os exiting from the whole side of the chip, using all the 6 pins. For this reason it has been chosen the lowest impact under impedance mismatching terms, with constantly varying width, to a standard width of 0.2mm, corresponding to more then 70Ω impedance. This is confirmed by reference designs of other equivalent chips of the same manufacturer, since the actual layout model was not available. As explicitly sugested by datasheet, the lines outside the matching networks are designed to be 50Ω.

The 50Ω impedances width are not considered as a microstrip, because the ground plane sorrounding them. Are considered as a groundwd coplanar waveguide (figure 7.7) and the proper width is derived using AWR TxLine tool, again.

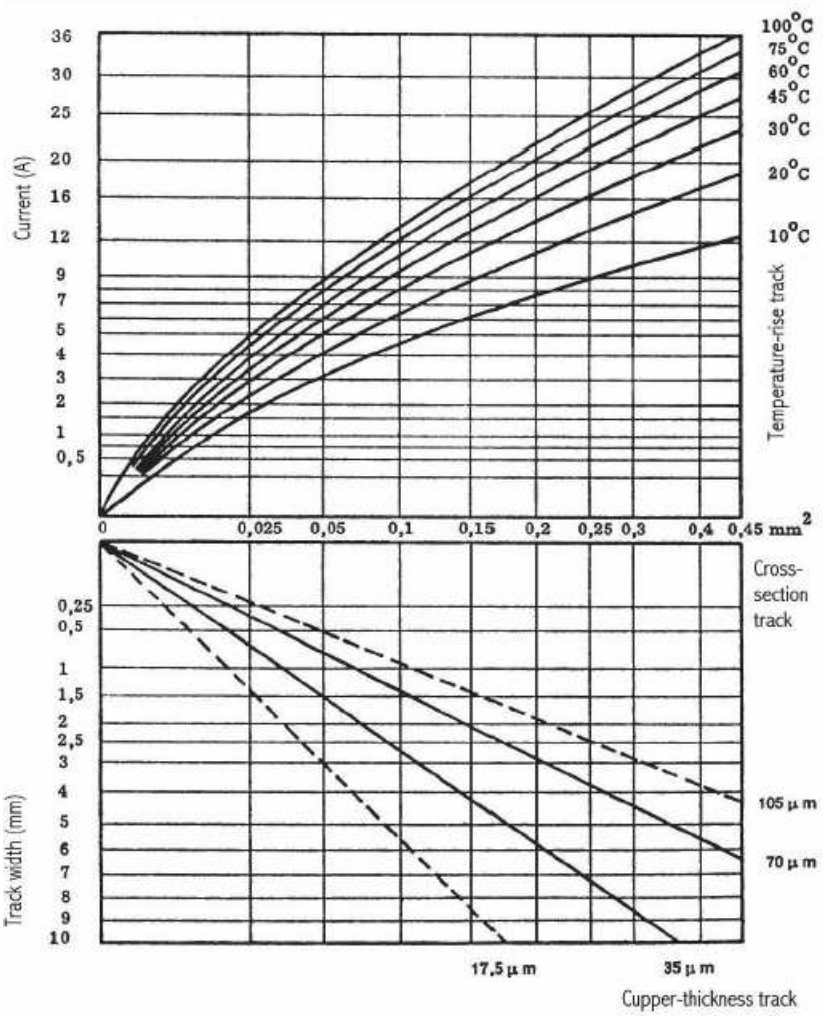

Figure 7.5. Standard PCB width analysis graphs, from PCB manufacturer

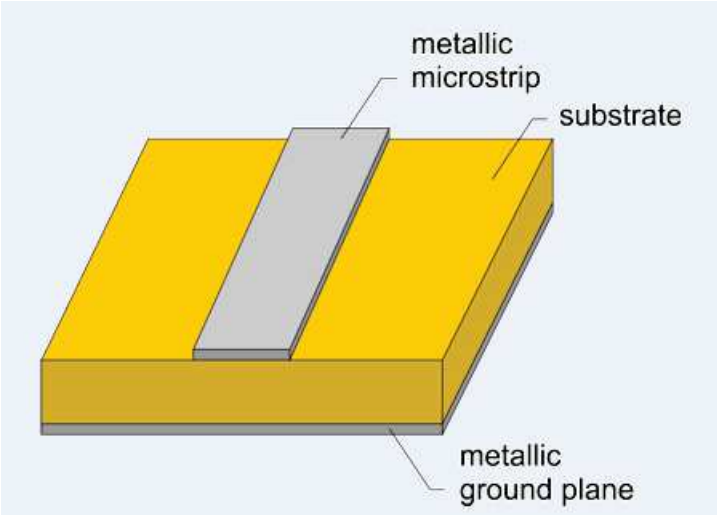

Figure 7.6. Microstrip

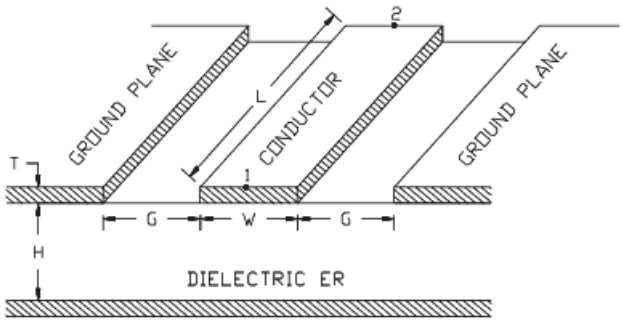

Figure 7.7. Grounded coplanar waveguide

7.3 PCB implementation

The final tile with the 1B31A OBRF 437 MHz module is shown in figure 7.8, without the SHF module (1B31B OBRF 2.4GHz). Are visible the 4 screws used to fix the PCB, each connected to

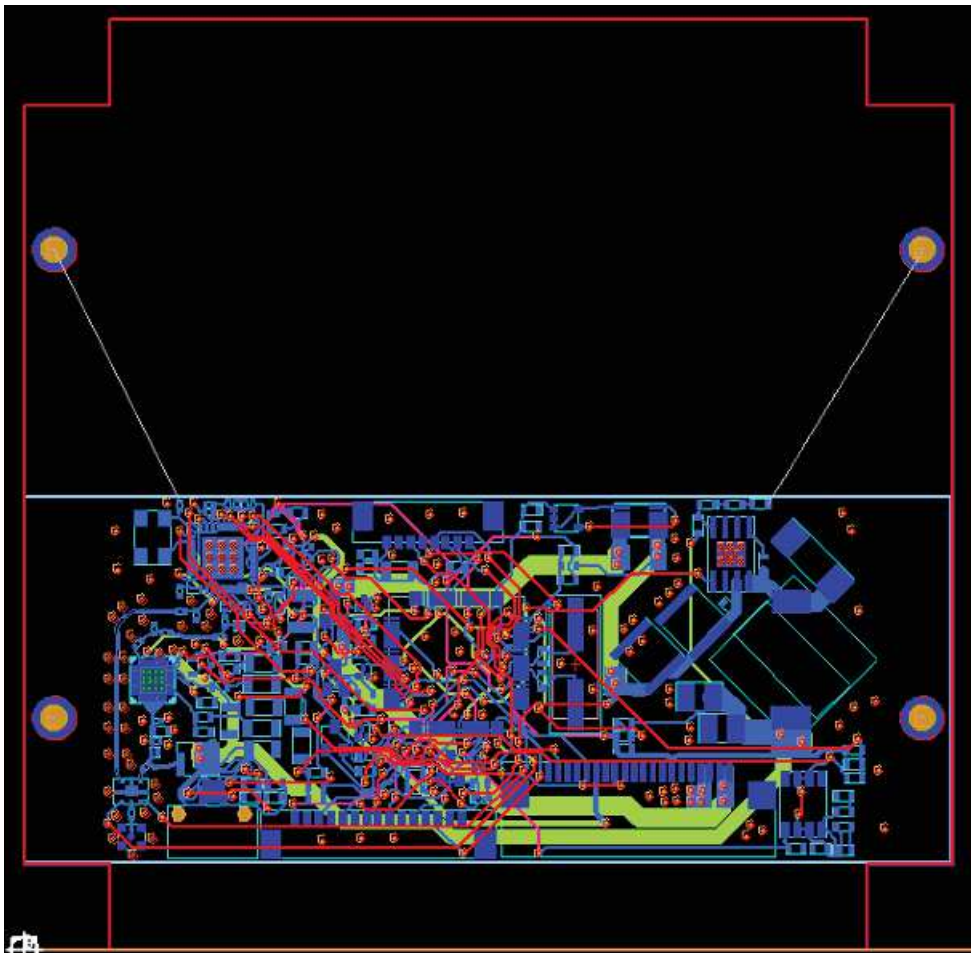

Figure 7.8. Tile with 1B31A OBRF module placed

the 4 grounded layers.

7.3.1 Layer organization

As mentioned before, are used 4 layers. In each of them is placed a digital/RF GND mostly for shielding all traces and for heat spreading purposes. The result of the whole module engineering left enough space for a PCB stack-up of 4 layers instead of the 8 already defined, without affecting performances and lowering the costs.

The layer 1 in blue (figure 7.9) contains all the components, the silkscreen, soldermask and solderpaste; are also shown the 3 mains locations of the RF subsystem, the digital subsystem and power supply subsystem. The RF traces are kept here to avoid vias and are shielded with ground planes. Is then used this layer to wire as mush as possible all the signals. The analog ground (AGND) plane shapes are drawed here, under the CC1020 (on RF side), the current sensor and the LM317L.(in the power supplies side).

Layer 2 (figure 7.10), is used mainly for grounding purposes, but few traces are still placed, mainly due to AGND distribution and few traces that were not placeable elsewhere. Layer 3 (figure 7.10), is used to place the power nets, the supply distribution of the possible voltages. Even here some signal traces are routed, but shielded by the GND plane. Finally, the fourth layer (figure 7.12) provides connections mainly for digital signal and sensor traces, all shielded between them.

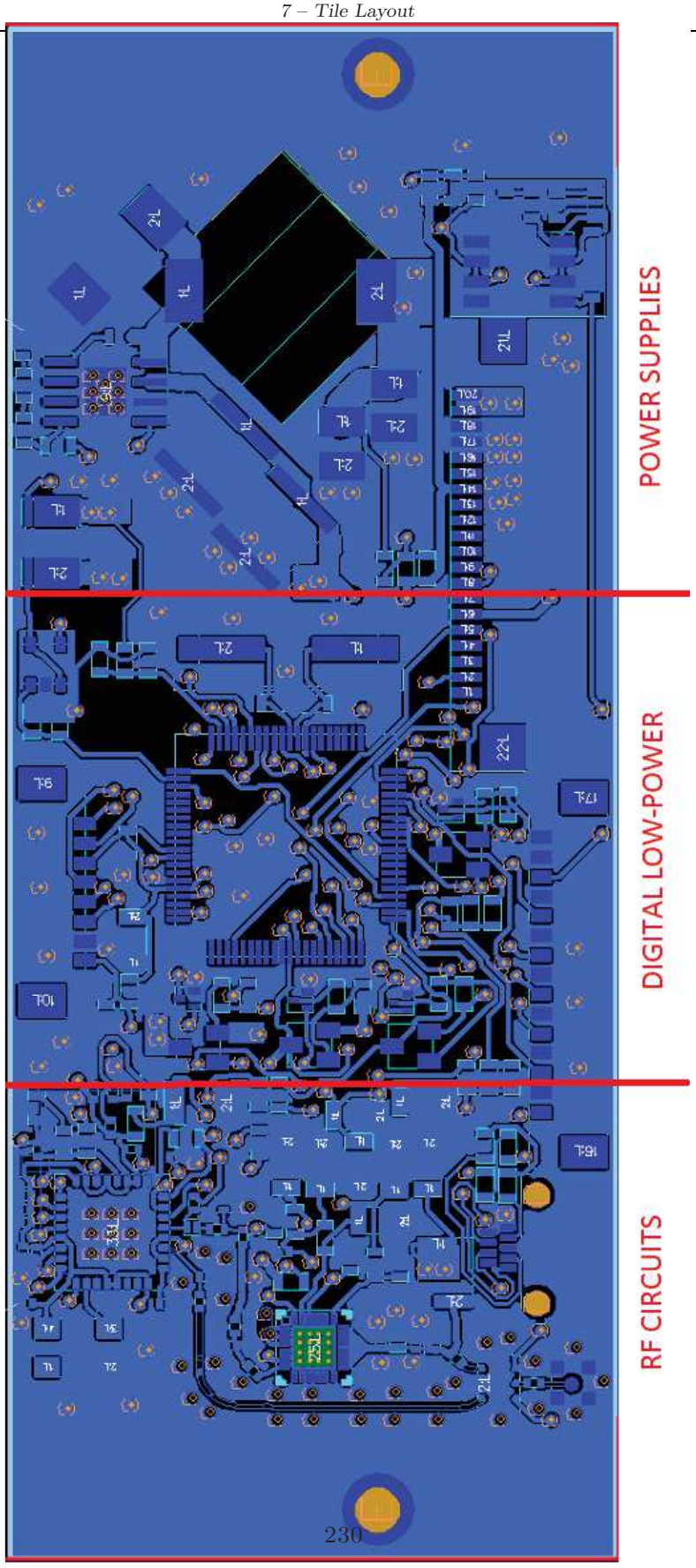

Figure 7.9. PCB Layer 1 and the 3 subsystems placement highlight

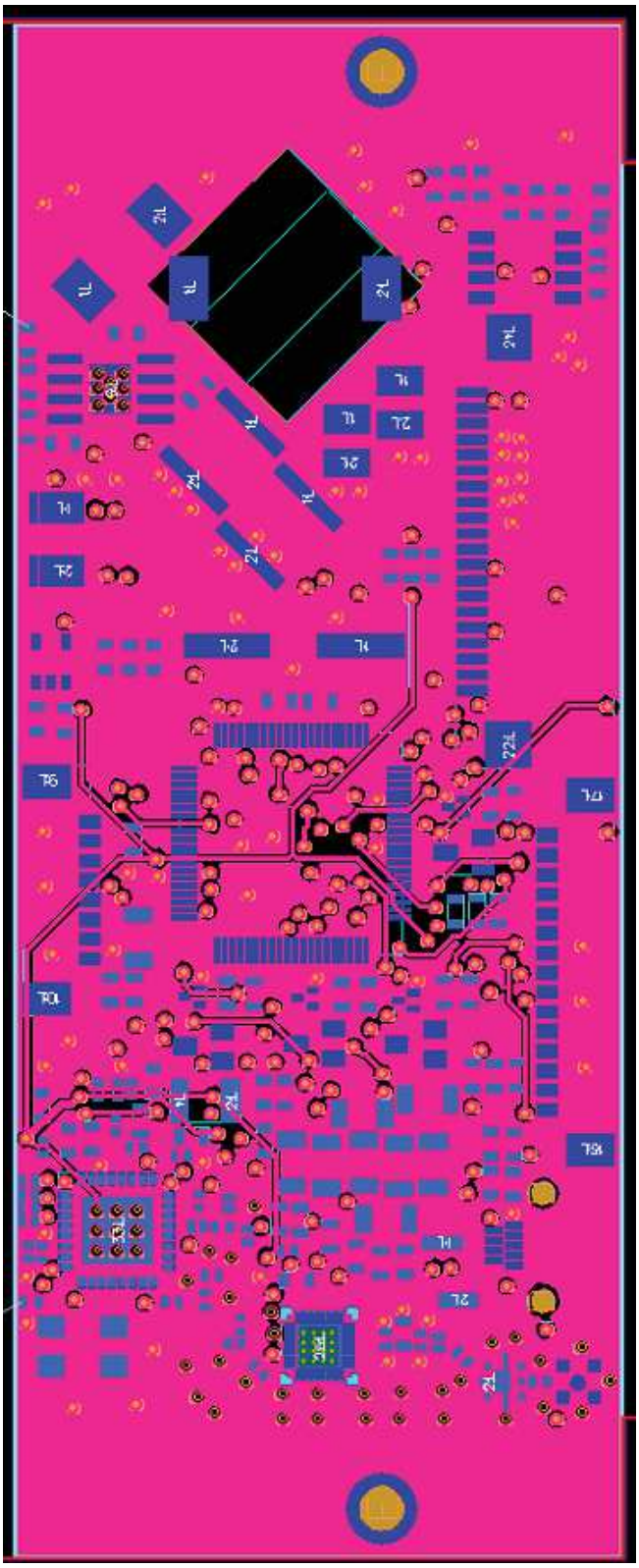

Figure 7.10 PCB Layer 2

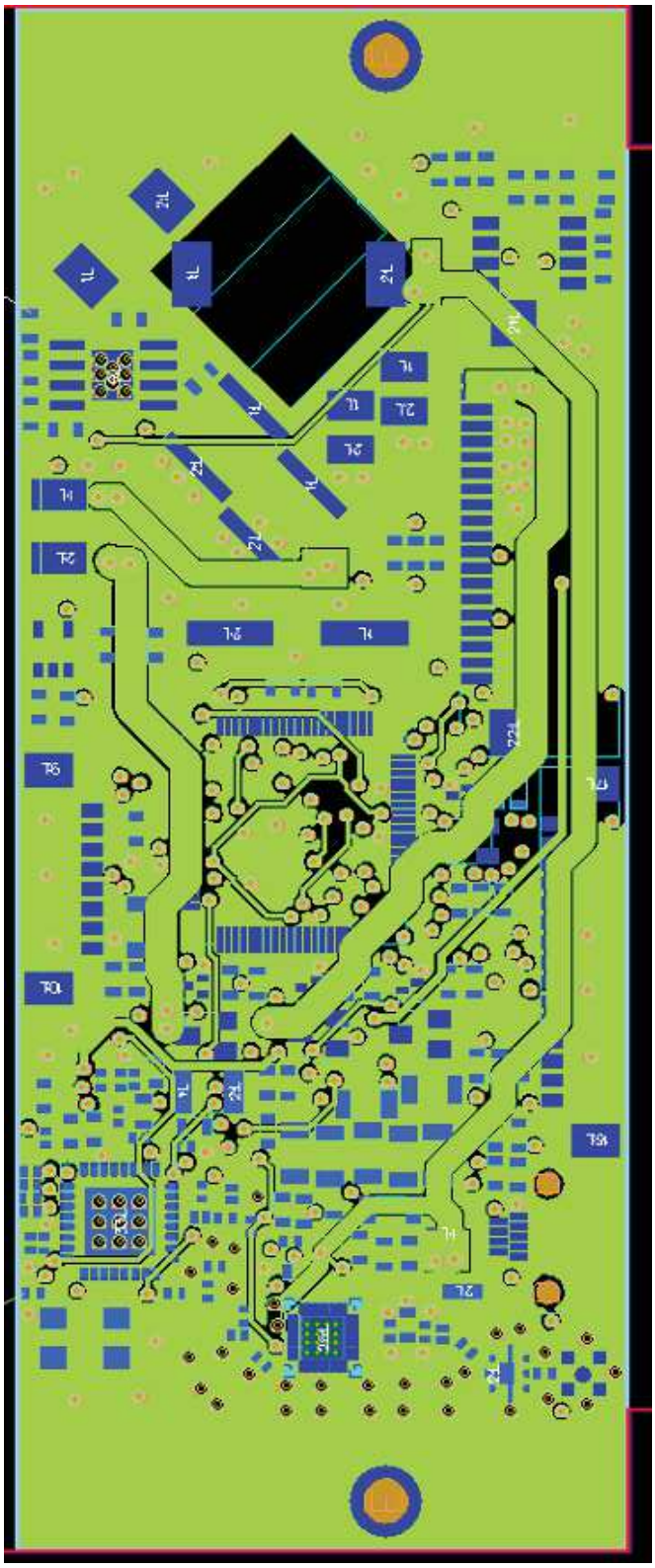

Figure 7.11. PCB Layer 3

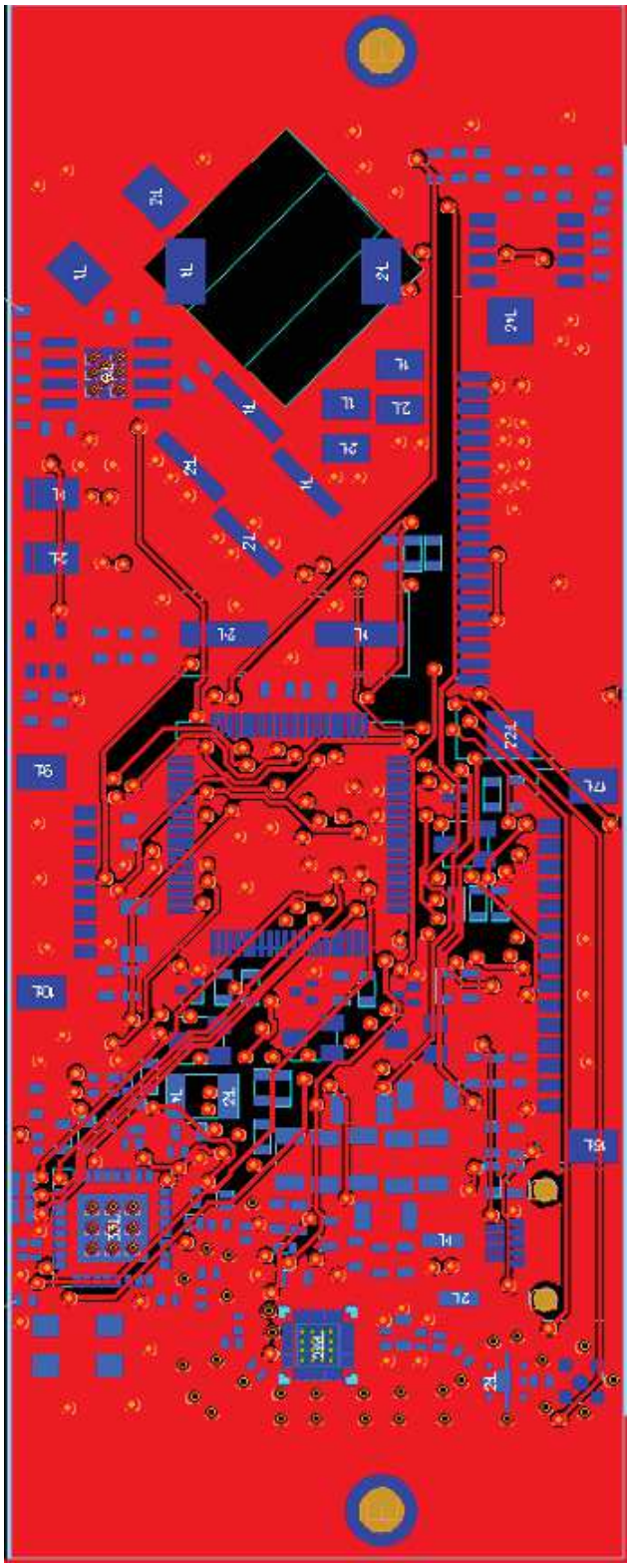

Figure 7.12. PCB Layer 4

Chapter 8

Conclusions

This work consisted in the completion of a previous and only partially developed AraMiS On-Board Radio Frequency module, in the UHF band, in which were analyzed its feasibility in terms of power and link budgets, and was performed the main hardware components selection. Here the whole project is organized and documented using the UML Visual Paradigm tool, from the use cases definitions to the hardware.

Were defined all the timings and the specifications in which the system must be compliant with, and are rearranged the AraMiS telecommunication protocol at low level, without changing the interaction, therefore not affecting the dependability.

Therefore, after showing the specification of the system, in this thesis the environment constraints were verified under the worst case conditions and starting from them, have been devised the use cases of the module. These were necessary in order to manage the design inside a well defined boundaries and develop an affordable system, with hardware and software tightly interconnected.

Since the main component selection was already performed, the hardware needed only a reorganization in UML class diagrams to keep coherency with the use cases and the logical behaviour defined. The hardware is redesigned using the already selected components, reorganizing its hierarchy and modularity. Are used tool such as SmartRF Studio from TI, Mentor Graphics Expedition Enterpise, AWR TxLine. Are also used a some application and development notes from the manufacturers. The design improves the power supply system and the housekeeping sensors.

The software was instead completely developed from scratch, with the help of some already developed modular software in the AraMiS project library. According to use cases and timings devised, the high level software organization is defined, according with the available hardware components selected. Then are defined the proper algorithms and their implementations for the

RF protocol adopted, for the on-satellite communication and the module housekeeping. A complete integration of the CC1020 handling is devised, supported by the TI application notes.

Once the whole integration is completed, is devised the physical implementation of the tile. A thermal analysis is performed, in order to devise a good PCB placement. The PCB manufacturer is Eurocircuits and the design tool used were AWR TxLine and again the Mentor Graphics Expedition Enterprise suite. The final PCB is not yet manufactured.

A further work could be mainly devoted to software development and refinement, since various calibration parameters and templates needs to be defined and tested on the microcontroller, possibly using a development board for the transceiver. There is also a lack on the documentation of the antenna control system, that will be available after purchasing the hardware; therefore are instantiated the methods related to it, but are not developed as well as the set of commands. Then the PCB manufacturing can be issued and the final tile could be tested. Therefore, the OBRF at SHF band can be integrated with this telecommunication module, so testing and troubleshooting will be possible also to the complete CubeSat AraMiS Telecommunication system, by placing both modules on the same tile.

Appendix A

CC1020 Registers

ADDRESS	Byte Name	Description
00h	MAIN	Main control register
01h	INTERFACE	Interface control register
02h	RESET	Digital module reset register
03h	SEQUENCING	Automatic power-up sequencing control register
04h	FREQ_2A	Frequency register 2A
05h	FREQ_1A	Frequency register 1A
06h	FREQ_0A	Frequency register 0A
07h	CLOCK_A	Clock generation register A
08h	FREQ_2B	Frequency register 2B
09h	FREQ_1B	Frequency register 1B
0Ah	FREQ_0B	Frequency register 0B
0Bh	CLOCK_B	Clock generation register B
0Ch	VCO	VCO current control register
0Dh	MODEM	Modem control register
0Eh	DEVIATION	TX frequency deviation register
0Fh	AFC_CONTROL	RX AFC control register
10h	FILTER	Channel filter / RSSI control register
11h	VGA1	VGA control register 1
12h	VGA2	VGA control register 2
13h	VGA3	VGA control register 3
14h	VGA4	VGA control register 4
15h	LOCK	Lock control register
16h	FRONTEND	Front end bias current control register
17h	ANALOG	Analog modules control register
18h	BUFF_SWING	LO buffer and prescaler swing control register
19h	BUFF_CURRENT	LO buffer and prescaler bias current control register
1Ah	PLL_BW	PLL loop bandwidth / charge pump current control register
1Bh	CALIBRATE	PLL calibration control register
1Ch	PA_POWER	Power amplifier output power register
1Dh	MATCH	Match capacitor array control register, for RX and TX impedance matching
1Eh	PHASE_COMP	Phase error compensation control register for LO I/Q
1Fh	GAIN_COMP	Gain error compensation control register for mixer I/Q
20h	POWERDOWN	Power-down control register

Figure A.1. CC1020 Register Overview

Appendix B

Bill Of Material

1B31A2M_OBRF module

Part Lister output for Bk1B31A2M_OBRF_437MHz

#	QTY	Part Number	Ref Designator
1	1	DK-631-1070-2-ND	X1
2	1	DK-863-1174-1-ND	U4
3	1	DK_296-21715-5-ND	U1
4	1	DK_296-23766-1-ND	U6
5	1	DK_300-8526-2-ND	X2
6	2	DK_311-82JRCT-ND	R17,R18
7	1	DK_311-1011-1-ND	C23
8	3	DK_311-1014-1-ND	C22,C49,C50
9	1	DK_311-1016-1-ND	C9
10	2	DK_311-1024-1-ND	C52,C53
11	1	DK_311-1025-1-ND	C18
12	4	DK_311-1026-1-ND	C13-C15,C55
13	2	DK_311-1061-1-ND	C35,C36

```
          -ND
14    1   DK_399-1278-1 C20
          -ND
15    1   DK_399-3525-6 C38
          -ND
16    3   DK_399-4937-1 C56-C58
          -ND
17    2   DK_445-1245-1 C16,C17
          -ND
18    2   DK_445-1270-1 1B31A2_TILE_C1,
          -ND           1B31A2_TILE_C2
19    1   DK_445-2153-1 L2
          -ND
20    2   DK_445-3486-1 C1,C2
          -ND
21    1   DK_490-1125-1 L6
          -ND
22    1   DK_490-1283-1 C54
          -ND
23    3   DK_490-1303-1 C10-C12
          -ND
24    1   DK_490-1305-1 C26
          -ND
25    1   DK_490-1530-1 C27
          -ND
26    3   DK_490-1586-1 C46-C48
          -ND
27    1   DK_497-1572-1 U2
          -ND
28    1   DK_541-33.0SC R19
          T-ND
29    1   DK_587-1523-1 L4
          -ND
30    1   DK_587-1525-1 L3
          -ND
31    1   DK_587-1526-1 L5
          -ND
32    1   DK_587-1527-1 C28
          -ND
33    1   DK_689-1091-6 U7
          -ND
34    1   DK_712-1333-1 C51
          -ND
35    2   DK_1276-3375- C3,C4
          1-ND
36    1   DK_B550C-FDIC D1
          T-ND
37    1   DK_INA138NA/2 U5
          50G4-ND
38    2   DK_NTA7002NT1 M2,M5
```

		GOSCT	
39	2	DK_NTA7002NT1	Q1,Q2
		GOSCT-ND	
40	1	DK_PCC2308CT-	C21
		ND	
41	1	DK_PCD2154CT-	L1
		ND	
42	1	DK_SE2418CT-N	1B31A2_TILE_X1
		D	
43	1	DK_WM7612CT-N	J3
		D	
44	1	DK_WM7619DKR-	J4
		ND	
45	1	DK_WM9358-ND	J2
46	1	DK_WM10423CT-	J5
		ND	
47	1	FR_2285536	L8
48	1	OMNETICS_A291	J1
		00-009	
49	1	RS-624-2222	C24
50	1	RS-698-2731	C59
51	4	RS_301-322	M1,M3,M4,M6
52	1	RS_461-2708	C8
53	1	RS_504-6499	R12
54	1	RS_504-6506	R21
55	4	RS_504-6900	R10,R11,R42,R43
56	1	RS_504-7341	R16
57	1	RS_504-7363	C40
58	1	RS_504-8546	R30
59	4	RS_504-8827	R46-R49
60	4	RS_504-8934	R3,R7,R9,R20
61	3	RS_504-8940	R2,R6,R22
62	1	RS_504-8956	R32
63	1	RS_504-9224	R31
64	2	RS_504-9684	R4,R8
65	10	RS_505-0151	R1,R5,R34-R41
66	2	RS_505-0303	R44,R45
67	1	RS_505-0331	R33
68	1	RS_505-0836	R13
69	2	RS_505-1081	R14,R15
70	9	RS_534-5730	C5,C6,C29-C34, C39
71	1	RS_545-4115	C19
72	1	RS_566-428_K	R23
73	2	RS_616-9391	C7,C37
74	1	RS_626-3954	U3
75	5	RS_648-0733	C41-C45
76	1	RS_669-8808	C25
77	1	RS_684-1273	NR1
78	1	RS_725-4901	L7

```
79    8   TP            8_TP1,8_TP2,
                        8_TP3,8_TP4,
                        8_TP5,8_TP6,
                        8_TP7,8_TP8
```

Bibliography

[1] Cubesat Specification. Available at http://cubesat.calpoly.edu/images/developers/cds_rev13_final.pdf

[2] Passerone C., Tranchero M., Speretta S., Reyneri L., Sansoe C., Del Corso D., Design Solutions for a University Nano-satellite, Aerospace Conference, 2008 IEEE , vol. no. pp.1,13, 1-8 March 2008.

[3] Speretta S., Reyneri L. M., Sansoe C., Tranchero M., Passerone C., Del Corso D., Modular architecture for satellites. 58th International Astronautical Congress, Hyderabad, India, 2007

[4] Source at http://www.planet.com

[5] http://en.wikipedia.org/wiki/Miniaturized_satellite

[6] Stefano Speretta, Project solutions for low cost space missions, PhD thesis, March 2010

[7] Alessandro Matheoud, UHF Radio Frequency Modules for Satellite-Ground Communication, MS Thesis, 2012

[8] Haider Ali, Telecommunication Subsystem Design for Small Satellite, PhD thesis, March 2014

[9] http://en.wikipedia.org/wiki/OSI_model

[10] William A. Beech, Douglas E. Nielsen, Jack Taylor, AX.25 Link Access Protocol for Amateur Packet Radio, Version 2.2, July 1998

[11] TI, CC1020 Datasheet, April 2013

[12] TI, AN070 CC1020 Automatic Power-Up Sequencing

[13] TI, AN023 CC1020 Microcontroller Interfacing

[14] https://en.wikipedia.org/wiki/Frequency_modulation

[15] Hittite Microwave Corporation, Thermal Management for Surface Mount Components, 2012

[16] Cree, Optimizing PCB Thermal Performance, CLD-AP37 Rev 2E, 2014