Using a radio transceiver for CubeSat communications

David Soldevila Casanovas

Resum– Aquest projecte forma part de la iniciativa portada a terme per l'Institut de Ciències Espacials de Catalunya, la Universitat Politècnica de Catalunya i l'Insitut de Ciències de l'Espai [5]. Aquesta iniciativa persegueix desenvolupar un microsatèlit CubeSat des de zero sense dependre de tercers. Un CubeSat és un microsatèlit format per un o més cubs de mida estàndard (10 x 10 x 11,35 cm) que funciona amb components comercials, és a dir, components produïts en massa disponibles en qualsevol botiga, que no estan pensats per funcionar a l'espai [1]. El seu avantatge principal és el baix cost no només de la seva construcció, donat el baix cost dels components i la possibilitat de reutilitzar-los, amortitzant el desenvolupament, sinó també en el llançament, ja que son enviats com a carga extra de llançaments més grans. El seu principal inconvenient és la seva curta vida, degut als components utilitzats. Aquest projecte va ser ofert per l'ICE i el seu propòsit és el de desenvolupar el *driver* pel transceptor del CubeSat perquè es pugui comunicar des de i a la Terra. En aquest informe es mostra el procés que s'ha seguit per desenvolupar aquest driver.

Paraules clau– cubesat, satèl·lit, cc1101, stm32, nucleo, f446ze, spi, sdr, gnuradio, radio, receptor, transmisor, transceptor, arm, encastat

Abstract– This project is part of the initiative carried out by Institut de Ciències Espacials de Catalunya, Universitat Politècnica de Catalunya and Institut de Ciències de l'Espai [5]. This initiative aims to develop a Cubesat microsatelite from the ground without depending on third parties. A Cubesat is a microsatellite formed by one or more cubes of standard size (10 x 10 x 11.35 cm) that work with commercial components, that is, mass produced components available at any store, not made specifically for the space environment[1]. Their main advantage is the low cost not only of the manufacturing, due to the low cost of the components and the possibility of reusing them, thus amortizing the development, but also in the launching, since they are sent as an extra cargo of other bigger launches. Their main disadvantage is their short lifespan, because of the components used. This project was offered by the ICE and its purpose was to develop a driver for the radiofrequency transceiver of the CubeSat to make it able to communicate to and from earth. In this paper the process followed to develop this driver is shown.

Keywords– cubesat, satellite, cc1101, stm32, nucleo, f446ze, spi, sdr, gnuradio, radio, receiver, transmitter, transceiver, arm, embedded

1 Introduction

This project aimed not only to develop a driver for the CC1101 radio transceiver, but also, with the help of this paper, to shed some light on other projects of this kind, to offer some advice and to prevent others from making the same mistakes. The driver developed during the realization of this project was made to run on a STM32 Nucleo-F446ZE development board and thus compatible, in principle, with any STM32 board. Nevertheless, its layered nature eases its porting to other platforms. It has 3 main layers: The first layer, which is platform-dependent, is the SPI communication, controlled as explained further on by the STM32 HAL Driver. The second one abstracts the data flow between the board and the CC1101 transceiver. The third one is compound by high level functions that handle the configuration of the transceiver and control the logic to perform the reception and transmission of data. For the most part of this paper, the CC1101 transceiver is called just

• E-mail de contacte: dsoldevila@protonmail.com
• Menció realitzada: Enginyeria de Computadors
• Treball tutoritzat per: Marius Montón Macián (Microelectrònica i sistemes electrònics)
• Curs 2018/19

as transceiver and the Nucleo-F446ZE development board as board.

2 OBJECTIVES

1. Developing a functional driver for the board to enable the transmission and reception of data.
   (a) Learning about the board and the development environment.
   (b) Learning about the transceiver functionality.
   (c) Establishing a communication with the transceiver using SPI.
   (d) Generating signals with the transceiver and capturing them with an SDR (Software-Defined Radio).
   (e) Achieving reception of data.
   (f) Accomplishing data transmission between devices.
2. Implementing additional functionalities.
   (a) Frequency set-up (carrier base frequency, frequency offset, channel shifting, channel spacing, etc.)
   (b) Making the driver failure-proof.
   (c) Modulation mode set-up (OOK, GFSK, MSK, etc.)*
   (d) Wake On Radio (WOR)*
3. Impelementing CCSDS Standard layers.*
4. Making the driver proof to inconvenient effects such as Doppler's.*

* The feature has not either been implented or tested.

3 STATE OF THE ART

In this section the state of the art is introduced as well as other key concepts that appear later in this document.

3.1 CubeSat

The summary of the state of the art of the CubeSat is that is all done. As said in the abstract, the main goal is to develop a home made CubeSat, but there are already many institutions which have launched successfully this type of microsatellite and even companies that sell them.

CC1101

CC1101 is a popular inexpensive flexible and highly configurable transceiver designed for low-power wireless applications. It features a variety of frequencies to use within the sub 1Ghz range, various modulation modes, data rates from 0.6Kbits/s up to 600Kbit/s and so on.

Other RF transceivers on the market

The transceiver used is not the only one on the market, there are other chips such as the AX5042. The AX5042, just as the CC1101 , is a highly customizable transceiver and offers pretty much the same features. Choosing the CC1101 was a matter of availability.

CCSDS Standard

CCSDS stands for Consultative Committee for Space Data Systems, which is a "a multi-national forum for the development of communications and data systems standards for spaceflight, founded by the major space agencies of the world"[6]. This turns it into the de facto standard in the sector. In Figure 1 the different layers of the standard given by the CCSDS can be seen.

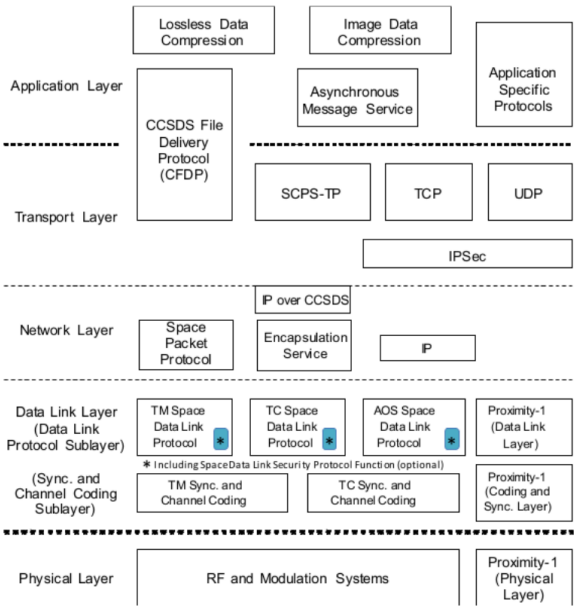

Fig. 1: CCSDS Standard layers [4]

3.2 SDR

To check that the transceiver is transmitting something, an easy approach is to use and SDR. An SDR consist of that certain parts are implemented on software, so that a particular device can be used to work with Bluetooth or any radio-frequency-based technology. That is to say, via software, it can be configured to "understand" the transceiver. On the hardware part, there are various USB dongles on the market, and the one used has been the RTL-SDR. These dongles have an antenna with which they capture the raw signal to then send this signal to the PC for a later post-processing. Its set-up is simple and straightforward. In Figure 15, a photography of the dongle can bee seen.

3.3 SPI

SPI stands for Serial Peripheal Interface. On this project this bus is used to communicate the transceiver with the board. An SPI bus follows a master-salve hierarchy, where there is one master and several slaves, and requires, at least, 4 wires: The Clock, the MOSI (Master Out Slave In), the

MISO and the Chip Select. The later is used to select a particular slave to communicate to, for that reason, there are as many Chip Selects as slaves.

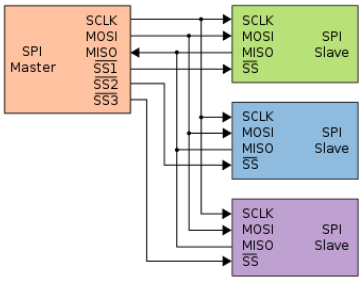

Fig. 2: SPI with 1 master and 3 slaves [14]

3.4 UART

UART stands for Universal Asynchronous Receiver-Transmitter. The UART bus is used on this project to communicate the board with the PC via the USB port.

3.5 Telecommunication concepts

It is not the purpose of this document to get in much detail into telecommunication aspects, but it has been considered appropriate to explain a general idea of some of them, since they appear in this paper.

Carrier Signal

The carrier signal is a signal with constant frequency, typically a sinusoid, that carries the signal containing the actual data to be transmitted. See Figure 3.

Modulation

The modulation is a set of techniques that are used to vary the properties of the carrier signal with the signal containing the information to be transmitted. In Figure 3, AM and FM modulations can be seen.

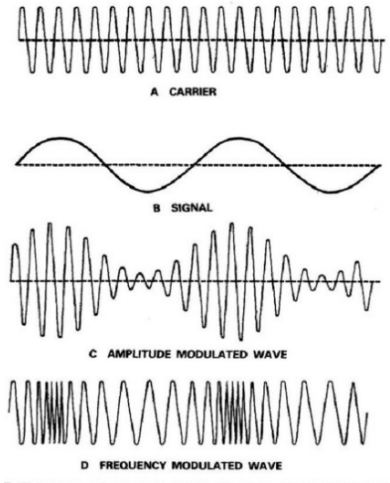

Fig. 3: Amplitude and frequency modulation [15]

The are various kinds of modulation, the ones offered by the transceiver are On-Off Keying, Amplitude-shift Keying, Frequency-shift Keying, Gaussian Frequency-shift Keying and Minimum-shift Keying. ASK has been the one used in the most part of this project. With this modulation the radio signal changes its amplitude according to if it's sending a 1 or 0. A graphical representation of real data can be seen further on in Figure 5.

4 METHODOLOGY

1. An agile methodology has been used, as the scope of the project was never intended to be closed. The first objective is the core of the project, but from that on the intention has been to advance what time allows. In fact, the third and the forth objective have been left as future work. The former aimed to integrate the CCSDS communication standard layers. The latter, to make the communication proof to Doppler effects and so on. Furthermore, taking into account the lack of experience in this field, and agile methodology eases the adaptation to unexpected issues during the project.

2. All the needed resources, such as the repository, the board, the transceiver, etc, were provided by Institut de Ciències de l'Espai.

3. During the realization of this project a development board was used when possible, since it was forbidden to move the final board outside ICE and was not fully functional.

4. Instead of doing periodic feedback meetings from time to time, as agile methodologies require, the feedback was done as needed in form of casual conversations.

5 DEVELOPMENT

The development of this project has been divided into two parts, following the two first objectives.

5.1 Transceiver driver

The development environment chosen for this project has been System Workbench for STM32, which is an open-source IDE based on Eclipse ready to work with STM32 boards out-of-the-box. Note that there were problems with System Workbench when debugging on the board, so depending on the Operating System and distribution it may not work. Using an external debugger solved the issue [9].

Another tool called STM32CubeMX has also been used. This tool allows users to , through a graphical interface, select the board components needed and configure them comprehensively. Then it generates a project frame with the components ready to be used. Not only initializes them, but also includes the corresponding HAL (High Abstraction Layer) drivers in case they are necessary. A HAL Driver is basically a set of high level functions to use the corresponding module in a transparent manner. For example, the SPI HAL Driver includes functions such as transmit or receive.

To transmit and receive data with the transceiver the following steps were followed:

5.1.1 Pinout

As said above, the transceiver uses a SPI bus to communicate with the board.

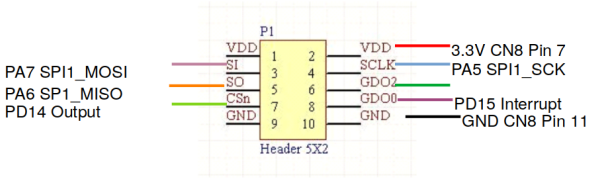

Fig. 4: Pinout configuration. The yellow rectangle represents the transceiver.

There were no pin restrictions in this project, so they were selected for their convenient position on the board [3] [7]. In Figure 4 we can see the pin-out configuration and in Figure 12 (A.3) a real photography of it. The additional pins GDO2 and GDO0 have the same multiple uses (see Table 41 in the datasheet [2]). For this project, the GDO0 pin was used to generate interruptions given particular events. The GDO2 pin was not considered necessary, since both pins share the same functions and it is not needed to use two of them at once, so a single pin can be reconfigured at different stages in the code.

5.1.2 SPI Communication

To make the SPI communication work, apart from configuring the SPI HAL driver properly (clock speed, bit order, etc.), a normal GPIO (General Purpose In Out) pin had to be used instead of the default Chip Select. To impersonate the Chip Select, the state of the GPIO pin has to be manually changed before and after initiating the SPI communication, that is to say, before and after calling SPI's HAL functions. In the transceiver datasheet it is specified that Chip Select must be set to low during the transmission and reception of data.

5.1.3 Turning on the transceiver

Before testing the correct behaviour of the SPI communication the transceiver must be explicitly turned on. To do so the following sequence must be used:

1. Set CS to High
2. Wait 10 microseconds
3. Set CS to LOW
4. Wait 40 microseconds
5. Send command strobe SRES

The command strobes are 1-byte instructions used to change the state of the transceiver internal state-machine.

5.1.4 Testing the SPI communication

To test the correct behaviour of the SPI communication a special register was read. This register, called version in the transceiver datasheet and common in every MCU (Micro Computer Unit), has a known immutable value. Obtaining the expected value when reading this register means that the communication works. Furthermore it is also advisable to read the state-machine state, which should be IDLE after turning on.

5.1.5 Serial Communication

For debug purposes an UART bus was also configured to communicate the board with the PC via USB in order to print data directly to the terminal [8]. This comes handy, as it is not always possible to use the debugger: The transceiver is not affected by it, only the board, which can break the driver during transmission and reception of data.

5.1.6 Debugging the signal

To check that the transceiver is transmitting, first of all is recommended to check that the RTL-SDR works. To do that an open-source software called Gqrx SDR was used, which is one of the programs recommended in the RTL-SDR Install Guide [13]. It does not meet the features to decode the data sent by the transceiver, but it can be used to listen to the radio, for example, without requiring any further configuration.

Once assured that the reception end, the RTL-SDR, works fine, the next step is to verify the transmission end. To check that the transceiver was transmitting, the RTL-SDR was used with the help of a program called GNURadio, which is also open-source. This software can be programmed using diagrams, which are made of blocks, each one performing a certain task. Custom blocks can be written, although it was not needed for this project. Two diagrams were made to perform this post-processing, being its main objective to transform the analog signal sent by the transceiver back into binary data, to compare if the received data was the same as the original data. The first diagram records the raw data received by the RTL-SDR (as can be seen in Figure 5) and stores it into a file. The second one loads this file and does the actual post-processing [11] (as can be also observed in Figure 6. A deeper explanation of these diagrams can be seen in appendices A.1 and A.2 respectively, as well as the reason for this division in two.

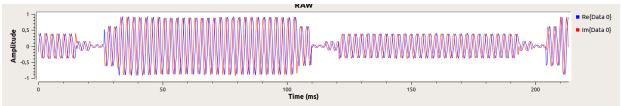

Fig. 5: Raw ASK signal captured with GNURadio.

5.1.7 Signal Generation

After achieving an SPI communication, the next step is to make the transceiver generate some dummy data in order to check that the transceiver is doing something. To do that, its 47 register were configured to set the desired modulation, signal frequency, signal intensity and so on. In this stage the goal is to achieve to do something with the transceiver, so it is convenient not to get too complicated and not to configure manually every register by reading the datasheet, as it is easy to make a mistake. The approach taken was to get a reportedly working configuration from the Internet[10], but there is also a tool made by the transceiver manufacturer

called SmartRF Studio, which generates the respective register values given the desired signal properties. Within the modulations available, ASK (Amplitude-shift keying) was the one to be chosen, because of its simplicity to be decoded.

Furthermore, the transceiver was configured to transmit at 2.4 Kbaud/s. To generate this dummy data a particular characteristic of the transceiver was used. When sending data, the transceiver first sends a preamble, which is a string of bits that toggles ones and zeros. If it has entered transmission mode, but there is not data to be sent, the preamble is sent indefinitely, until there is. Figure 5 is in fact a visual representation of the preamble.

5.1.8 Data transmission (I)

Once assured the transceiver works, the next step is to check that the data is being correctly transmitted. As said previously, the transceiver was configured to transmit an ASK signal at 2.4Kbaud/s. Furthermore the transceiver was configured to, when there is data, send a 4-byte preamble, followed by a synchronization word repeated twice, for a total of 4 bytes, to finally send the payload, that is, the actual data.

Before testing, the infinite preamble trick was again used to check that the second GNURadio diagram was also working properly, as can bee seen in Figure 6.

To perform the data transmission test, an array containing 6 bytes of data (in addition to the preamble and the sync. word, for a total of 12 bytes, as can be seen below) was sent.

```
PREAMBLE
1010 1010 1010 1010 1010 1010 1010 1010

SYNC WORD (x2)
0101 0111 0100 0011 0000 0111

DATA(0x07 x6)
0000 0111 0000 0111 0000 0111 0000 0111
0000 0111
```

The reception on the RTL-SDR side was successful as can be seen in Figure 7. To print the bit string the post-processed data (the output file of the second diagram) was fed into python script [12], which was later updated to perform the check automatically.

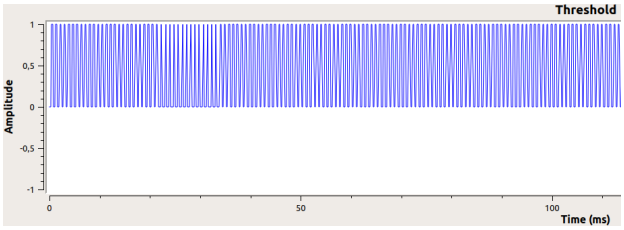

Fig. 6: Preamble representation on GNURadio after post-processing

5.1.9 Data transmission (II)

The transceiver has a buffer for transmitting data of 64 bytes, although the chip has no packet length restriction. To send data packets longer than the buffer size, the data must be written into the buffer as it's being sent, checking at every moment that there is neither overflows (the transceiver receives more data than it can send over the radio) nor underflows (it doesn't get new data in time and stops the transmission. In order to control this flow of data, there are 3 different approaches: Polling, which consist of asking the chip periodically, interruptions or via DMA (Direct Memory Access). The second way has been chosen since it is a midpoint between ease of implementation and efficiency.

Fig. 7: Screenshot of the received data.

To explain in more detail how this interruption driven solution has been implemented, the transceiver has two additional pins called GDO0 and GDO2. These ones can be configured to be toggled in certain events, such as an incoming reception or when passing a certain threshold in the buffer. When this happens, an interrupt is captured in the corresponding pin on the board. The base logic used for data transmission is to write to the buffer 32 bytes of data, every time that the buffer gets half empty, which is the event that triggers the GDO0 pin and therefore, the interruption.

Although the chip has no data length restriction, it does require a special procedure for sending packets longer than 255 Bytes. This is because to send data, the data length must be written to the Packet Length register. The chip sends bytes until the internal counter reaches the Packet Length register value. This register is 1-byte long though, so 255 is the maximum value it can contain. To overcome this limitation, the chip can be configured to use the so called "Infinite Packet Length" mode to spin the counter, although it must return to "Finite Packet Length mode before ending the transmission, since the counter must reach the configured packet length to finish correctly. What the datasheet recommends to do is to set the packet length to $length\%255$ and activate the "Finite Packet Length" mode when there are $length\%255$ bytes of remaining data.

In the event of data transmission failing it is important to check the various registers to determine that the transmission has actually finished. For example, by verifying that the transceiver is in the correct state or that the Packet Status register tells so.

To test the correct functioning, the SDR was used again, but due to the post-processing approach taken, only up to 30-40 consecutive bytes of data could be verified before getting a mismatch (A.2). Because of this, the testing was par-

tially ignored to implement the reception of data instead, to then test the transmission and reception at once, using two boards to establish an end-to-end communication. The second board being a prototype of the final board to be used in the CubeSat, also called On Board Computer (OBC). In Figure 14 (A.3) a photography of it can bee seen.

5.1.10 OBC Setup

The STM32CubeMX tool was used again to generate a project for the OBC. Furthermore, to work on the development of the driver without duplicated files, the following file scheme was used. The “Nucleo and ”OBC“ folders

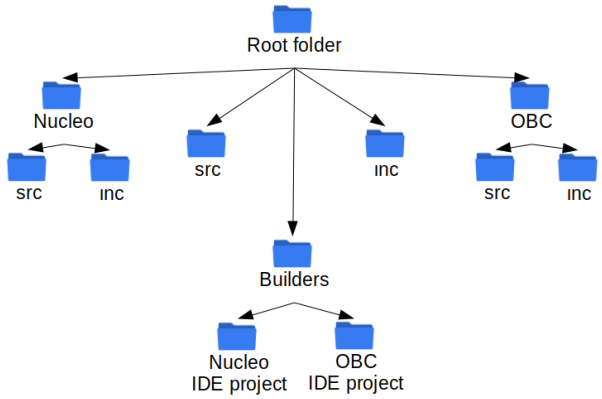

Fig. 8: File scheme used for the project. Nucleo refers to the development board.

contain source and include folders, which were moved from their respective project folders. ”inc“ and ”src“ contain the header and the source files of the driver respectively.

This was done because linking the driver files from an external location worked for the IDE, but not for compiling, since the compiler was not able to find them. With this scheme, the files can be retrieved using short relative paths from the main include locations. On the IDE the moved source and include folders must be added to the project as source locations under ”Path and Symbols“, the driver files just need to be linked. A Photography of the setup used at this stage can bee sen in Figure 14 (A.3).

5.1.11 Data reception (I)

As occurred with data transmission, the data reception was also divided into two parts, in order to not go into much detail into the control logic and to focus only in the correct reception of data instead. In other words, a small packet of data was used, specifically 20 bytes, in order to not cause any overflow in the 64-byte buffer.

Although both transceivers were equally configured with a carrier signal frequency of 433.98MHz, there was actually a considerable difference in the emitted frequency between them, so it had to be tweaked at both sides with an offset to make the communication work. The difference can be seen in Figure 9. The modulation used was also changed from ASK at 2.4Kbit/s to GFSK at 1.2Kbit/s, due to the fact that the transceiver does not support frequency offset compensation for ASK (and OOK). What frequency offset compensation does is to mitigate the difference in frequency between the transmitter and the receiver, which made possible to configure the offset without the need to be extremely precise.

Furthermore it is important to configure the output signal power according to the distance between the two transceivers, since a too high output power can saturate the transceiver at short distances.

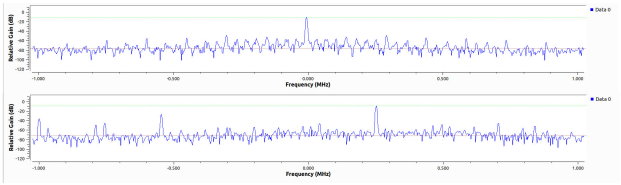

Fig. 9: Difference in frequency between boards. Captured with GNURadio. Top: Board. Bottom: OBC.

5.1.12 Data reception (II)

The transceiver has also another buffer of 64 bytes for receiving data. As occurred with data transmission, the control logic for receiving packets longer than 64 Bytes had to be implemented. The approach of interruption was also taken, by using the same base logic of reading in chunks of 32 Bytes, although reversed. That is, reading the buffer when it is, at least, half full, which is not the same as half empty. At least half-full implies that it is fine to read 32 Bytes when there are 42 in the buffer, an event that would cause and overflow if a writing was performed instead. But there is a problem with that logic, which is that the interruption will not be triggered with the last chunk of data, unless the packet is multiple of 32 Bytes. For example, with a packet of 300 Bytes, the last 12 would be left ($300\%32 = 12$). That’s why the reception is not full interruption-driven, since the last bytes are retrieved using polling.

Moreover, unlike data transmission, a mechanism for reading the length of the packet had to be also implemented, as it was key to make the reception work. The logic behind this mechanism is to initialize the Packet Length register to a number great enough to read the data length field and, when retrieved, overwrite the Packet Length register with the field value.

2 Bytes of the packet had to be reserved to store the length of it, since 1 Byte was neither enough to test packets transmitted in Infinite Packet Length mode nor enough for most communication protocols. The position of the length was not hard-coded, since it should be decided by the communication protocol on top and not the driver. The length of the data length field is not that flexible, although the field can be masked to recover a custom number of bits. For example, the first layer of the CCSDS standard, the Data Link layer, uses 10 bits to store the length of the packet. To read the length field, polling is also used for the same reason described in the paragraph above.

As a side note, due to the fact that the OBC had not the external interrupt enabled at the beginning, the data transmission code had to be temporarily modified to use polling instead. The roles of the two boards were arbitrarily decided, so the first board could have been used as the sender.

As it was expected, debugging the transmission and the reception at once to patch errors was more complicated, but by testing different critical data lengths and watching the symptoms one can get close to the cause. To illustrate that, if the error appears at 65 bytes, it is reasonable to think that the error appears when transmitting, since 65 is the bare minimum to use the interruption system. If it appears at 67, it is quite plausible that the error is located in the reception, since 67 is no different from 66 and 65 from the transmission perspective, but it is on the reception: If the data length is located at the beginning (position 0 of the array of data), once retrieved 65 bytes are left, which is the minimum to trigger the interruption system.

5.1.13 CRC Checksum

The transceiver can calculate automatically and transparently a 16-bit CRC checksum of the whole sent packet to verify that the data has been correctly transmitted. Although it is attached at the end of the packet when transmitting, it is automatically removed by the transceiver in the reception. If the data received is correct, but the CRC tells otherwise, a probable cause is that the reception or the transmission is not finishing correctly.

5.1.14 Making the driver failure-proof

The driver needs to handle also interrupted transmissions. By using a timer the driver can be prevented from waiting for the data indefinitely. An interrupted transmission can cause also to make the transceiver read data where there is not, but this is handled by the CRC checksum.

5.2 Additional functionalities

Apart of implementing the transmission and reception of data, it was also considered useful to make a transparent configuration for various settings of the transceiver as well. Also, the Wake On Radio functionality is mentioned, although it has not been properly implemented.

5.2.1 Carrier Frequency Configuration

The transceiver has 3 registers for setting up the base carrier frequency, using the mathematical function below:

$$f_{base} = \frac{f_{XOSC}}{2^{16}} \times FREQ\_REG$$

Where f_{XOSC} is the frequency of the crystal oscillator. In order to make it more transparent, a code to perform the reverse function was made, so that the user only needs to introduce the desired frequency. To check the configured frequency corresponded to the frequency of the output signal, the RTL-SDR was used again along GNURadio. This approach was used also in the following subsections.

5.2.2 Carrier Frequency Offset

It is usual the appearance of a discrepancy between the configured frequency and the actual frequency. To mitigate this, an offset can be added to adjust the frequency, which also uses a mathematical function.

$$f_{offset} = \frac{F_{XTAL}}{2^{14}} \times OFFSET\_REG$$

Where f_{XTAL} is the frequency of the crystal.

5.2.3 Channel Configuration

A channel is an offset added to the base frequency of the carrier. This offset depends on the channel number selected and the spacing, which is the jump between channels.

$$f_{carrier} = f_{base} + Ch_n \times Ch_{spc}$$

The channel number is written directly in a register, so its configuration is straightforward, but just as the carrier base and offset frequencies, the channel spacing uses a mathematical function.

$$\triangle f_{channel} = \frac{f_{XOSC}}{2^{18}} \times (256 + {Ch_{spc}}^M) \times 2^{{Ch_{spc}}^E}$$

5.2.4 Data Rate Configuration

Data rate can also be configured and ranges from 0.6Kbit/s to 500Kbit/s. The data rate is given by the mathematical function.

$$R_{data} = \frac{256 + DRATE^M \times 2^{DRATE^E}}{2^{28}} \times f_{XOSC}$$

The transceiver datasheet offers also 2 additional functions to find suitable values for a given data rate:

$$DRATE^E = log_2 \times \frac{R_{data} \times 2^{20}}{f_{XOSC}}$$

$$DRATE^M = \frac{R_{data} \times 2^{28}}{f_{XOSC} \times 2^{DRATE^M}} - 256$$

5.2.5 Modulation Configuration

The configuration of the modulation is straightforward as it is only needed to write a particular value to a register to swap within the different modes available. Nevertheless, other registers must be taken into account when changing the modulation. For example, the MSK modulation only supports data rates starting from 26KBit/s.

5.2.6 Power Output Configuration

The transceiver datasheet does not give any formula to calculate the power register values, but offers a table with recommended settings for common frequency bands. For a custom configuration, the use of the tool SmartRF Studio is explicitly recommended.

5.2.7 Wake On Radio

The Wake On Radio functionality enables the transceiver to periodically wake up from the sleep state to check for incoming packets without the interaction of the board. This feature can reduce significantly the power consumption of the chip, since the power consumption in sleep mode is an order of magnitude lower than idle's, although, evidently, the power consumption peak is given by transmit and receive modes.

6 RESULTS

The original general objective of this project was to develop a driver to control the CC1101 transceiver for a space-to-earth communication. Due to either a lack of time or an excess of ambition at the beginning, the final result is an all-purpose driver for the CC1101 transceiver, not related to space communication, as it does not have any specific features related to it, nor does include any layer of the CCSDS Standard. Reviewing the objectives:

1. I have learnt about the main features of the development board and the STM32 development suite.
2. I have got a failry deep understanding on the functioning of the CC1101 chip.
3. I have been able to communicate to the transceiver using SPI.
4. I have been able to decode amplitude-shift keying modulated signals with a software-defined radio.
5. I have achieved the reception of data.
6. I have been able to send data wirelessly from a chip to another.
7. I have implemented a transparent configuration for the various frequency parameters.
8. I have taken into account possible errors during the transmission, such as data corruption or an interruption of it.
9. I have not tested every modulation offered by the chip and the modulation mode set-up is still quite restrictive.
10. I have reviewed how to implement Wake On Radio and which parameters have to be taken into account, but it has not been properly implemented.
11. I have documented myself about the CCSDS standard and I had started to stack an implementation of the Data Link Layer on the driver, but it was finally discarded.
12. I have ignored how to make the driver proof to Doppler and other effects.

To summarize, the key objectives have been accomplished, but there is still a great deal of work to do. The code and other resources can be found at [16].

7 CONCLUSIONS

In this paper firstly some concepts of telecommunication have been introduced to have minimum understanding of the transceiver features as well as an introduction to the CCSDS standard and to other key technologies used during the development of this project such as the SPI and UART buses and the SDR. Secondly the basic set-up has been shown, that is the IDE, the pinout and so on. After that, the procedure for how to turn on the transceiver has been explained as well as how to validate that it has been turned on correctly. Then it has been told the first step to generate an output signal with it and how to decode it with the help of an SDR. It has also been explained the logic and the procedures used in the code to control the flow of data in the transmission and in the reception. Moreover, fault tolerance has been taken into account by checking the CRC checksum and by using a timer to force the transceiver to exit the transmission on reception if it takes too long. Finally, other additional features have been implemented, such as a transparent configuration for the frequency.

8 FUTURE WORK

The driver, although it abstracts the configuration of the transceiver, some registers still must be manually configured. Furthermore the driver depends on the knowledge of the user, as it does not check for conflicts in the configuration yet.

As said in the results section, this driver does not include space-related features. Hence, remains pending to stack the multiple layers of the CCSDS standard on top of it to make it compatible with other devices of the aerospace sector as well as to take advantage of its features, such as error correction or message acknowledgement.

This driver has neither into account unwanted effects such as the Doppler effect yet, which has the potential to break the communication, since it modifies the configured and expected frequency.

GREETINGS

I want to thank my tutors: Marius Montón Macián and Lluís Gesa Boté to bring me the opportunity to take part on the development of a CubeSat satellite and to support me during the realization of the project. I want to thank too other members of ICE's staff and each one of the anonymous heroes on the Internet, that were also of great help during the development of this project, some of which are mentioned in this paper.

REFERENCES

[1] CubeSat. April 2019. `www.cubesat.org`.

[2] Texas Instruments Inc.. *CC1101 Low-Power Sub-1 GHz RF Transceiver datasheet (Rev. I)*. November 2013.

[3] STMicroelectronics. *STM32F446xC/E datasheet (Rev. 6)*. September 2016.

[4] The LibreCube Initiative. *CubeSat Standards Handbook. A Survey of International Space Standards with Application for CubeSat Missions*. January 2017.

[5] Institut de Ciències de l'Espai. April 2019. `https://www.ice.csic.es`.

[6] CCSDS. April 2019. `https://public.ccsds.org/default.aspx`.

[7] STMicroelectronics, *STM32 Nucleo-144 User Manual (Rev 7)*. December 2017.

[8] EMCU. *How to implement printf for sending messages via USB*. April 2019. http://www.emcu.eu/how-to-implement-printf-for-send-message-via-usb-on-stm32-nucleo-boards-using-atollic.

[9] OpenSTM32 Community. *Could not determine GDB version*. April 2019. http://www.openstm32.org/forumthread3279.

[10] SpaceTeddy. *CC1101*. Github. April 2019. https://github.com/SpaceTeddy/CC1101.

[11] Cyrill Brunschwiler. *Software Defined Radio and Decoding On-Off Keying. Compass Security Blog*. September 2016. https://blog.compass-security.com/2016/09/software-defied-radio-sdr-and-decoding-on-off-keying-ook.

[12] Mike Czumak. *ASK/L(OOK)/Listen!*. Security Shift. September 2017. https://www.securitysift.com/ook-signal-decoding-replay.

[13] RTL-SDR. *Quick Start Guide*. May 2019. https://www.rtl-sdr.com/rtl-sdr-quick-start-guide.

[14] Cburnett. *SPI three slaves* Wikipedia. December 2006. https://en.wikipedia.org/wiki/File\:SPI_three_slaves.svg

[15] Ibukun Oluwayomi. *What is carrier frequency?*. Quora. https://www.quora.com/What-is-carrier-frequency

[16] David Soldevila Casanovas. *CC1101*. June 2019. https://github.com/dsoldevila/CC1101

APPENDIX

A.1 GNURadio Diagram 1

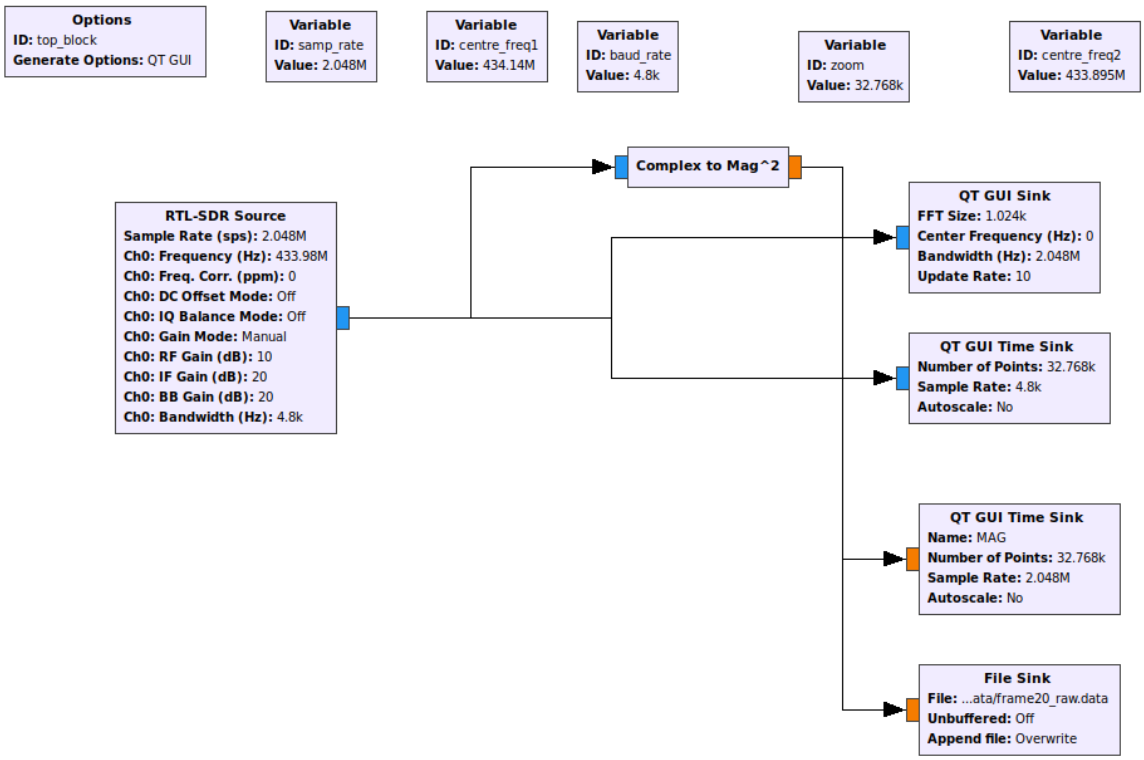

Fig. 10: GNURadio diagram used for capturing the raw radio signal

This diagram (Figure 10) gets the raw data from the RTL-SDR as complex numbers to then transform them into real numbers (floats) with the block “Complex to Mag” to finally save them in a file. The GUI blocks are used as a debugger, to check that the data received seems correct.

A.2 GNURadio Diagram 2

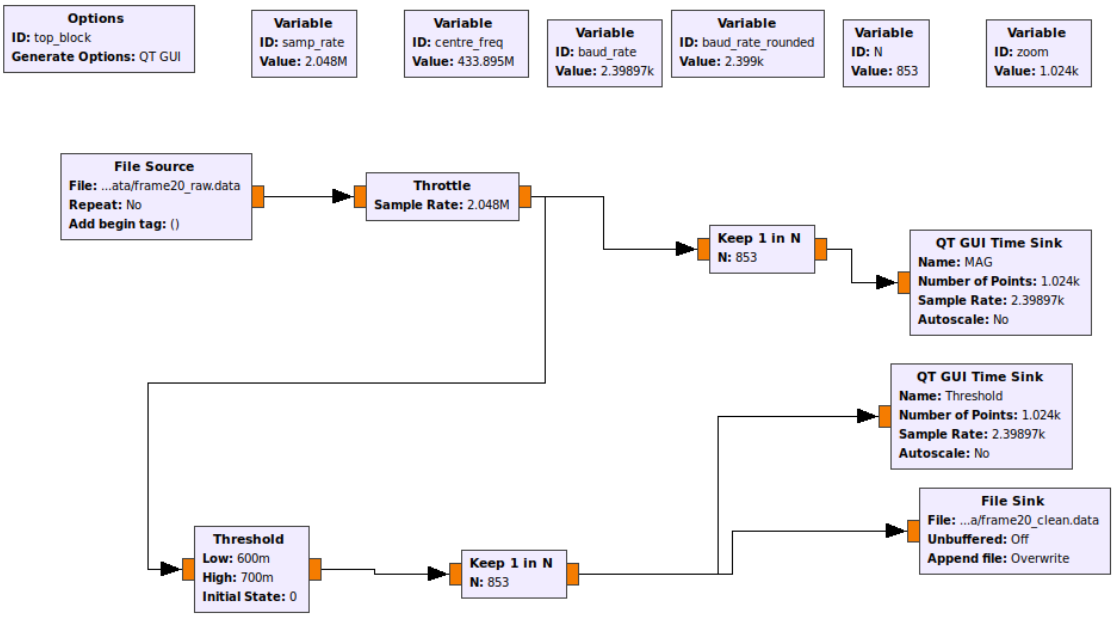

Fig. 11: GNURadio diagram used for post-processing the signal

This second diagram (Figure 11) takes the data generated by the former diagram and, with the block “Threshold” turns each sample into a 1 or 0. The correct behaviour of the threshold block depends deeply on the amplitude of the signal, which easily varies with the distance between the sender (the transceiver) and the receptor (the RTL-SDR) and the obstacles within. Dividing the diagram in two assures that the data, once recorded, will be always the same, hence it allows not to have to check in each execution whether the threshold fits the signal. Finally the sample rate is adjusted. The RTL-SDR captures 2.048 million samples per second, but the signal to be read has a different baud rate, in this case, 2.4Kbit/s. With the block “Keep 1 in N” the program is told to take 1 sample every 853 (RTL-SDR sample rate divided by baud rate), to read thereby 2.4 thousand samples per second.

Later in the project, this approach was found to not be optimum, due to the fact that the block “Keep 1 in N” is not accurate enough. N muss be an integer (it does not make sense to take a sample every 2.5 for example), but the division performed is not. This causes a periodic desynchronization.

A.3 Development setup

In this section a collection of photographies of the development setup used can bee seen.

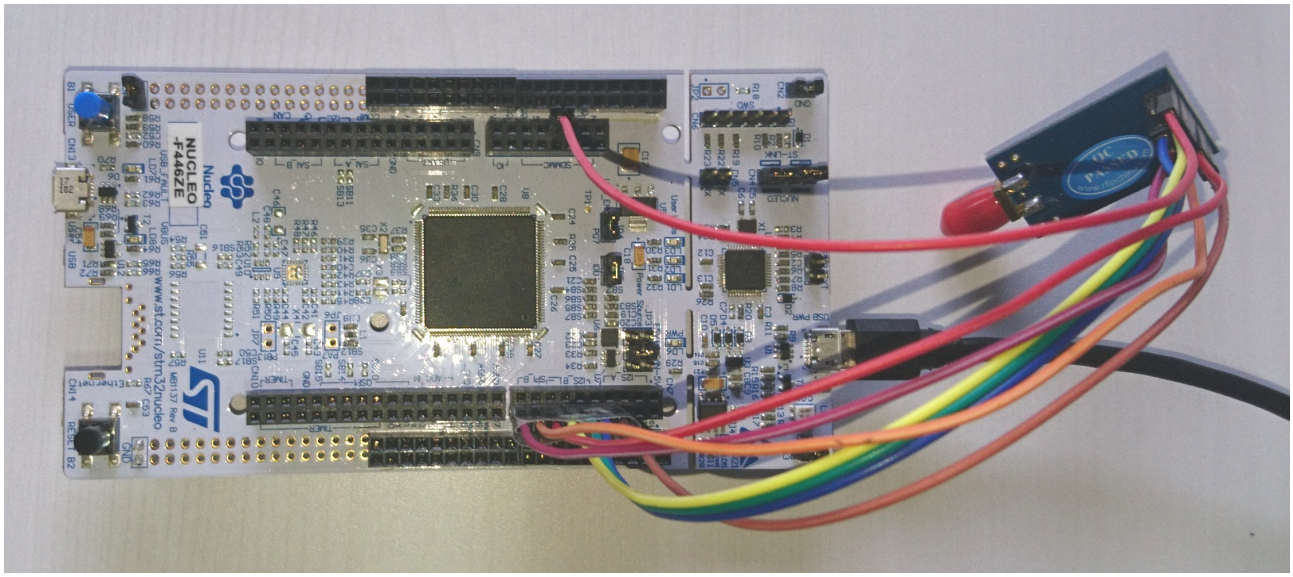

Fig. 12: The development board Nucleo F446ZE and CC1101 transceiver.

Fig. 13: Setup used during the final stages of the project. On the left the OBC can be seen, on the right, the Nucleo development board.

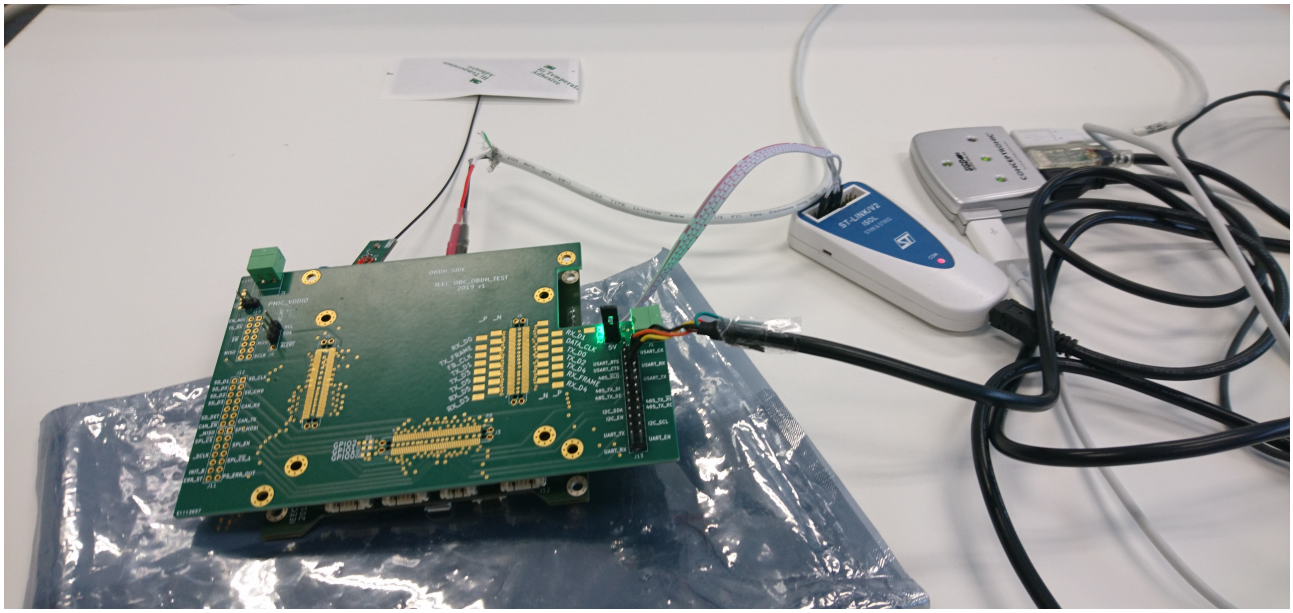

Fig. 14: A closer look at the On Board Computer prototype.

Fig. 15: RTL-SDR