University of Liège - Faculty of Applied Sciences

Optimization of the AX.25 and D-STAR telecommunications systems of the OUFTI-2 nanosatellite

Graduation Studies conducted for obtaining the
Master’s degree in ”Electrical Engineering” by

FRANÇOIS PIRON

Academic year 2018-2019

University of Liège - Faculty of Applied Sciences - Electrical Engineering

Optimization of the AX.25 and D-STAR telecommunications systems of the OUFTI-2 nanosatellite

François Piron

Supervisor: Jean-Michel Redouté

Academic year 2018-2019

Abstract

This work focuses on the development of the telecommunication systems of the OUFTI-2 educational CubeSat, based on its predecessor OUFTI-1. Both satellites embed 3 communication channels: control by the ground station with the AX.25 protocol, a repeater for the digital amateur radio D-STAR protocol (its primary payload) and a continuously emitting beacon.

The AX.25 uplink presented a known (but so far unsolved) issue in OUFTI-1. The On-Board Computer (OBC), which handles the packets, had no external way to detect their presence, and was thus interrupted 9600 times per second to decode the incoming data and look for a valid frame. The Sync Word Detection (SWD) feature of the ADF7021 transceiver could not be used because of a random scrambling process in the protocol. However, it is observed to actually be predictable and therefore the SWD mechanism can notify the OBC upon receiving a packet. The fix is then tested with a similar power level to what the ADF7021 will experience in space.

The D-STAR relay is implemented on a separate microcontroller. Due to the extent of the required modifications and the microcontroller model change for the FRAM-equipped MSP430FR5962, its source code was entirely rewritten during this work. New features were added, such as a logging system, and the parrot mode, allowing to transmit pre-recorded D-STAR messages stored in memory on command.

The modified telecommunications systems were thoroughly tested on their own, but further work will still have to test these at space-like RF power levels, and validate their interactions with the other subsystems, before OUFTI-2 can be put in orbit.

Université de Liège - Faculté des Sciences Appliquées - Ingénieur Civil Electricien

Optimisation des systèmes de télécommunication AX.25 et D-STAR du nanosatellite OUFTI-2

François Piron

Promoteur : Jean-Michel Redouté

Année académique 2018-2019

Résumé

Ce travail porte sur le développement des systèmes de télécommunication du CubeSat éducatif OUFTI-2, basé sur son prédécesseur OUFTI-1. Les deux satellites intègrent 3 canaux de communication : le contrôle par la station au sol avec le protocole AX.25, un répéteur pour le protocole radioamateur numérique D-STAR (sa charge utile principale) et une balise émettant en continu.

La liaison montante AX.25 présentait un problème connu (mais non résolu) dans OUFTI-1. L'ordinateur de bord (OBC), qui traite les packets, n'avait aucun moyen externe de détecter leur présence et était donc interrompu 9600 fois par seconde pour décoder les données entrantes et y chercher une trame valide. La fonction de détection de séquence (SWD) du (dé)modulateur ADF7021 ne pouvait pas être utilisée en raison d'un mélange aléatoire dans le protocole. Cependant, il est observé qu'il est en fait prévisible et, par conséquent, le mécanisme SWD peut notifier l'OBC à la réception d'un paquet. Le correctif est ensuite testé à un niveau de puissance similaire à celui que l'ADF7021 rencontrera dans l'espace.

Le relais D-STAR est implémenté sur un microcontrôleur séparé. Au vu de l'étendue des modifications nécessaires et du changement de modèle de microcontrôleur pour le MSP430FR5962 avec FRAM, son code source a entièrement réécrit dans ce travail. De nouvelles fonctionnalités a été ajoutées, telles qu'un système de journal et le mode perroquet, permettant de transmettre sur commande des messages D-STAR préenregistrés stockés en mémoire.

Les systèmes de télécommunication modifiés sont testés, mais il faudra encore le faire à des niveaux de puissance RF similaires à ceux de l'espace et valider leurs interactions avec les autres sous-systèmes avant de pouvoir mettre OUFTI-2 en orbite.

Acknowledgements

I would like to thank Prof. J.-M. Redouté for his follow up and enthusiasm for this project. I am also grateful to V. Broun for his valuable advice, and the ISIL laboratories that he made available for us to work in ideal conditions. I want to thank X. Werner as well for his help during the first half of this thesis, and Prof. J. Verly and S. De Dijcker for their interest in my work and useful remarks.

I thank my OUFTI-2 teammates François Grosjean (ULiège), Florian Radelet (ULiège), David La (ULiège), Soulaimane Harika (ULiège) and Guillaume Martin (INPRES) for our collaboration and the good atmosphere in the lab. Thank you to all the previous students that worked on OUFTI-1 and OUFTI-2 whose work I benefited. Thank you to my promotion mates and everyone I worked with and learned from during these studies.

I also want to thank my godfather Stéphane, Joachim and Samy for their proofreadings and suggestions, Robin, Gaël and Loïc for our library sessions (and breaks), Lucas, Gaëtan and Gilles. Thank you to my family for their support, my grandfather and grandmother, my godmother, Léon and Jean-Christophe. Finally, I would like to thank my mom and dad, to whom I owe everything, and who, I hope, would be proud of me.

Contents

Acronyms

ACK	Acknowledgement
AX.25	Amateur X.25 protocol
BCN	Beacon subsystem
BER	Bit Error Rate
COMM	Communications subsystem
CRC	Cyclic Redundancy Check
D-STAR	Digital Smart Technologies for Amateur Radio protocol
DV	Digital Voice D-STAR mode
FCS	Frame Check Sequence
FRAM	Ferroelectric Random Access Memory
FSK	Frequency Shift Keying modulation
GMSK	Gaussian Minimum-Shift Keying modulation
LNA	Low-Noise Amplifier
MUX	Multiplexer
OBC	On-Board Computer
OUFTI	Orbital Utility For Telecommunication Innovations
PA	Power Amplifier
PCB	Printed Circuit Board
PLL	Phase-Locked Loop
RF	Radio Frequency
RSSI	Received Signal Strength Indication
RX	Receive
SPI	Serial Peripheral Interface protocol
SWD	Sync Word Detect ADF7021 feature
TNC	Terminal Node Controller
TX	Transmit
UART	Universal Asynchronous Receiver Transmitter protocol
UHF	Ultra High Frequency, between 300 MHz and 3 GHz
VHF	Very High Frequency, between 30 MHz and 300 MHz

Chapter 1

Introduction

OUFTI-2 is an educational CubeSat mainly developed by students from the University of Liège and the Haute Ecole de la Province de Liège (ISIL and INPRES). The project started in 2016 after the launch of its predecessor OUFTI-1.

A CubeSat is a nanosatellite[1] of dimensions 10 cm × 10 cm × 10 cm, weighing around 1 kg and consuming around 1 W. It was developed in 1999 as a specification for educational satellites, by the *California Polytechnic State University* and *Stanford University.* They are usually launched either in batch by CubeSats deployment modules installed alongside bigger satellites in a launcher rocket, or from the International Space Station (ISS). More than 1000 CubeSats have been launched to this day [1].

OUFTI-2 embeds 3 payloads [2]. The main one is a repeater for the amateur radio protocol D-STAR. The second one (RAD) is a test board for new kind of multilayer electronics-shielding material. The last one (IMU) is an inertial measurement unit designed by high-school students.

The development of OUFTI-1 started in 2007. Tens of students took part in its design, implementation and tests, yielding more than 40 Master's theses. Its payloads were a D-STAR repeater like OUFTI-2 and high-efficiency solar panels from *Azur Space* [3]. It was launched in 2016 aboard the Soyuz VS14 launcher, with the 2 other CubeSats of ESA's[2] Fly Your Satellite program. For undetermined reasons, OUFTI-1 became silent after 12 days in orbit.

[1] Satellite with a mass between 1 and 10 kg

[2] European Space Agency

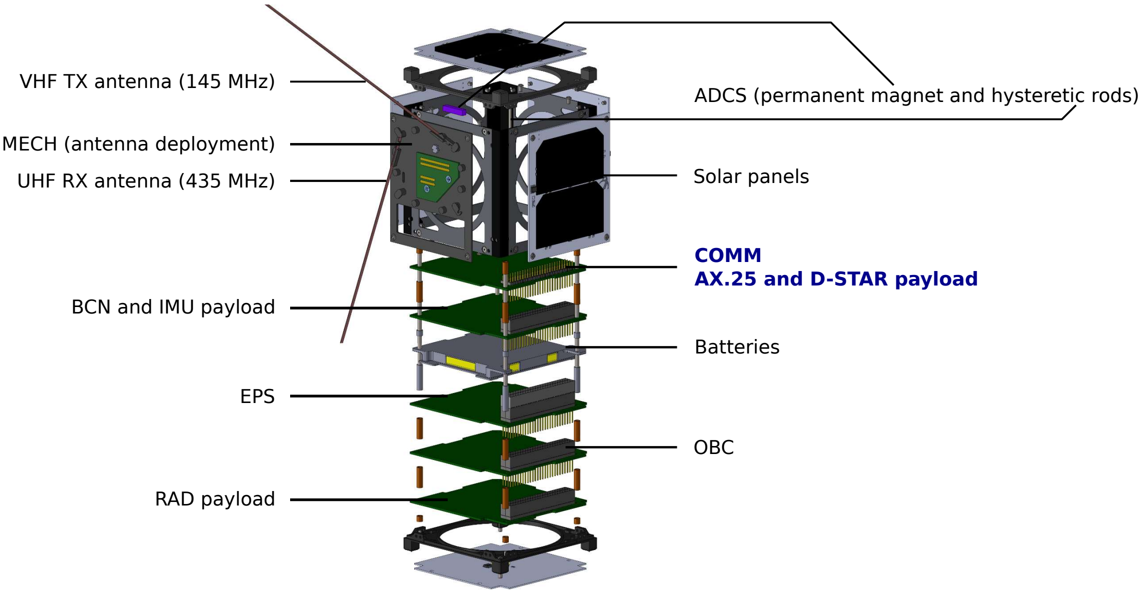


Figure 1.1 – OUFTI-2 CAD model exploded view (the antennas are truncated)

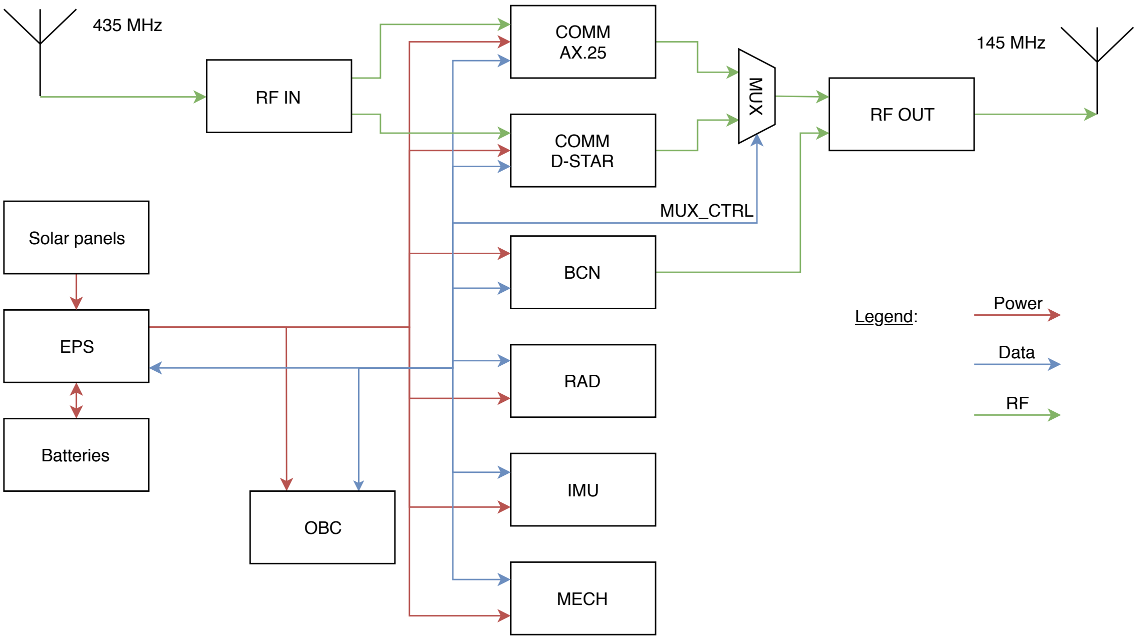


Figure 1.2 – Simplified overview of the interactions between the different subsystems

1.1 OUFTI-2 details

The different subsystems making up OUFTI-2, presented in Figures 1.1 and 1.2 are [4]:

COMM This subsystem includes the AX.25 and D-STAR transceivers, their associated circuits, all the other required RF components (amplifiers, filters, ...), as well as the D-STAR relay microcontroller.

OBC The core or the nanosatellite is its On-Board Computer (OBC). It controls all the other subsystems, communicates with the ground station through AX.25 telecommands and telemetries, schedules received orders execution and logs the important events and measurements of the satellite.

For OUFTI-2, the OBC is based on a new experimental 3 core controller provided by *Thales Aliena Space*, with a high resistance to radiations, a large temperature range (-55 – +125 °C) and embedded RAM error correction and detection. The software is currently being entirely rewritten. Unlike OUFTI-1, it will not use a Real-Time Operating System (RTOS) anymore and there will not be any backup OBC.

BCN The beacon continuously transmits measurements and information about the state of the satellite. The beacon is as independent from the rest of the satellite as possible. Its goal is to be able to collect some data even if other subsystems fail. Its Morse transmissions can be decoded by amateur radio operators listening from all around the world, which can pick up the satellite signal even when it would be impossible from Liège. It also transmits the same data at high speed (2400 bits/s) for machine decoding.

EPS The Electrical Power Supply tracks the maximum power point from the solar panels, handles the charge and discharge of the batteries, and provides 3.3 V and 5 V power buses to the rest of the satellite.

RAD This payload embeds 3 identical circuits, respectively with a new kind of multilayer electronics-shielding material, a conventional shielding and no shielding at all. It will measure and compare the degradation of the components due to radiation [5].

IMU This payload, designed by Belgian high-school students, will perform magnetic and inertial measurements in order to determine the attitude of the satellite.

ADCS This Attitude Determination and Control System is a passive way of controlling the orientation of satellite, and try to keep it stable relative to Earth. This is necessary to optimize RF communications and make sure all solar panels have similar exposure to the sun.

MECH The mechanical subsystems handles the releasing of the antennas when the appropriate signal is sent by the OBC, 30 minutes after the launch of the satellite in orbit.

STRU, THER and VIB Respectively stand for the structure, thermal and vibrations studies and ratings.

1.2 Communications

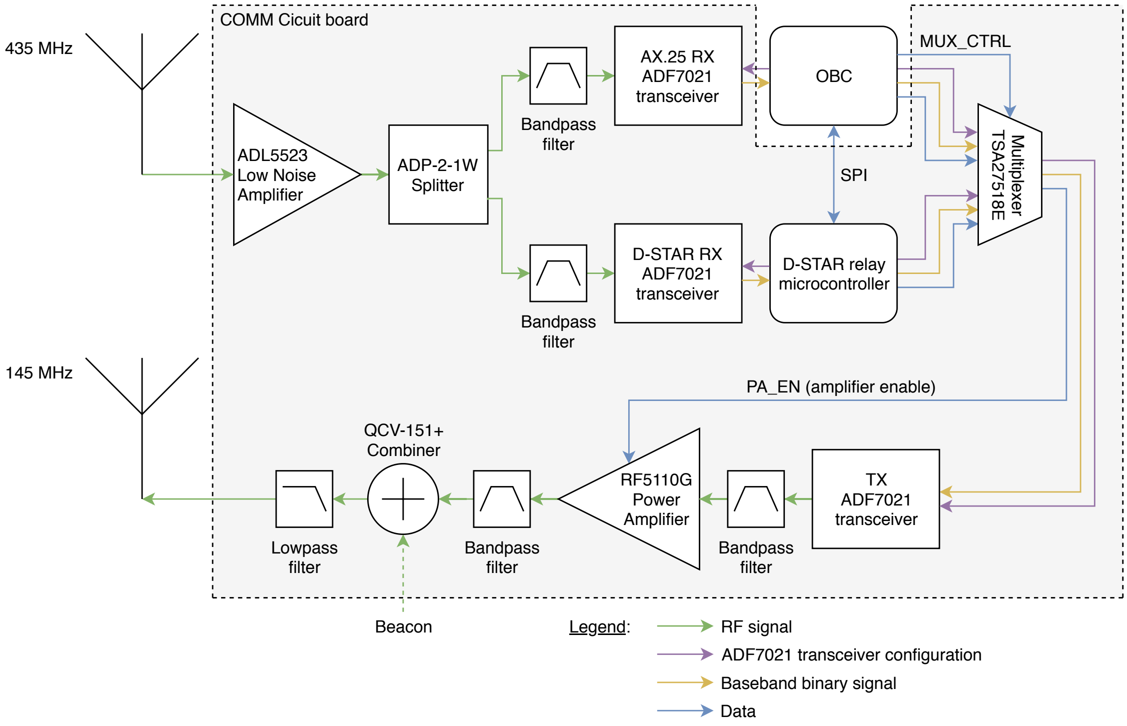


Figure 1.3 – Communication signals flow path. Impedance matching is not shown for clarity reasons

The radio communications of the satellite are divided in 3 channels:

- The AX.25 channel, used to control the satellite from the ground station and retrieve the on-board measurements, with uplink at 435.015 MHz and downlink at 145.950 MHz, modulated in 2-FSK at 9600 bits/s with a bandwidth of 9.6 kHz. It is entirely controlled by the On-Board Computer. The ground to satellite link will be developed in depth in Chapter 2.
- The D-STAR amateur radio channel, repeating from 435.045 MHz to 145.950 MHz, modulated in GMSK at 4800 bits/s with a bandwidth of 6 kHz. It is handled by the COMM microcontroller, which is the subject of Chapter 3.
- The beacon, continuously emitting measurements in OOK/Morse code at 12 words per minute, as well as 2400 bits/s, at 145.876 MHz.

Their interactions and the RF components of the COMM subsystem are shown in Figure 1.3.

This work will focus on the AX.25 and D-STAR communications, which are handled by 3 ADF7021 transceivers on the satellite. The 2 channels have separate reception (RX) ADF7021s, and share the same transmission (TX) ADF7021. The reason for the RX separation is that when the D-STAR repeater is active, the OBC **must** still be able to receive commands. It is required by law that the satellite transmissions can be shut down at any time when ordered [6]. On the other hand, AX.25 and D-STAR transmissions are using the same ADF7021.

The OBC decides which of the D-STAR repeater or itself has control over the TX ADF7021 through a digital multiplexer. Each time it switches, the TX ADF7021 is reconfigured either for AX.25 by the OBC or for D-STAR by the D-STAR microcontroller. As the enabling of the D-STAR circuit is also controlled by the OBC, this multiplexing process is effectively transparent for the D-STAR relay. When powered on, it always has control over the TX ADF7021. If the OBC needs to transmit while the D-STAR relay is active, it will shut it down and take control of the TX ADF7021 first.

The ADF7021 is a digital transceiver working from 80 MHz to 650 MHz and 862 MHz to 950 MHz [7]. It supports multiple FSK modulation configurations and filtering options, data rates up to 32.8 kHz and many other features and settings. It can be configured through the modification of 15 registers, 4 bytes long each. The communication channel for setting these registers is a simple 4-wires serial bus, with a protocol very similar to SPI (`SLE` = Load Enable = Chip Select, `SCLK` = Clock, `SDATA` = MOSI, `SREAD` = MISO) for which the ADF7021 would be the slave. The TX or RX data is transferred on 2 wires (`TxRxCLK`, `TxRxDATA`), whose clock is given by the ADF7021. Some of its specific features that are useful for OUFTI-2 will be explained later when appropriate.

The beacon emits in parallel with the other subsystems. Its RF signal is combined with the amplified output of the TX ADF7021 before going into the antenna.

1.3 Goals of this work

The aim of this work is to optimize some specific points and finalize the communications part of OUFTI-2, based on what has already been done for OUFTI-1. The first part is of course to familiarize with OUFTI-1, OUFTI-2 and their telecommunications systems, study previous work, their components and protocols. The two main tasks are then the following:

- Chapter 2: Fix a known AX.25 synchronization issue. The incoming telecommand packets could not be detected by the OBC, which forced it to continuously listen on the channel to check for the reception of valid data. The OBC was thus interrupted 9600 times per second, most of the times for nothing.

- Chapter 3: Re-develop the D-STAR repeater microcontroller code in a robust and reliable manner, and add a few features. The main addition is a ”parrot” mode, which consists in the transmission of pre-recorded D-STAR frames.

Before this work, the state of the COMM subsystem was the following:

- The required hardware modifications had already been performed. The COMM board schematics only needed some small corrections and there was already a partial PCB layout.
- The D-STAR microcontroller software had not been touched since OUFTI-1. Given the extent of the required changes, it had to be rewritten from the ground up.
- The ADF7021 configurations could be kept as the carrier frequencies and RF parameters for the AX.25 and D-STAR channels are the same as OUFTI-1.
- The ground station hardware (radios, TNC) of OUFTI-1 can be kept as well. The software will need to be adapted for the new AX.25 command set.

Fixing the AX.25 synchronization issue could have implied changes to the ground or satellite hardware, depending on the solution implementation. It was not necessary though, as the fix only requires changes to the ADF7021 configuration and thus the OBC software.

1.4 Activities

I took part in the following activities during this academic year:

- Visit of the CSL (Centre Spatial de Liège / Liège Space Center) on October 19th, 2018. Discussion with some employees followed by a visit of their clean room.
- Amateur radio HAREC exam passed on December 19th, 2018. The HAREC license is mandatory to use the radio equipment on the frequencies of this project. My callsign is ON9PF.
- Lecture on space radiations by Mr. Carapelle from CSL on April 4th, 2019.

Chapter 2

AX.25 synchronization

2.1 Description of the issue

As previously explained, there are 3 telecommunication channels in OUFTI-2. Each of them has a different purpose and is managed by a different part of the nanosatellite. One of them is the AX.25 channel, which allows for the remote control of the On-Board Computer (OBC), the brain of the satellite, from the ground station. Uplink AX.25 packets are called telecommands, and downlink ones are called telemetries. The OBC only transmits telemetries as a response to a telecommand.

In OUFTI-1, the On-Board Computer was constantly listening on the AX.25 channel, waiting to receive a command. In other words, it was constantly interrupted 9600 times per second to read an incoming bit, even when no packet was sent. This was a significant loss of processor time and power.

A much wiser approach would be to have some kind of external signal, notifying the OBC when a packet is detected. That way, it could listen on the channel only when necessary. It just so happens that a Sync Word Detect (SWD) mechanism is present in the ADF7021 transceiver, which is used both in OUFTI-1 and its successor. It could very well solve this issue, but it was not be used in OUFTI-1 due to a scrambling in the modem protocol [8]. The details of this SWD, the reasons why it could not be used and the solution are described in the following sections.

2.1.1 AX.25 protocol

AX.25 is an amateur radio protocol for the data link layer (layer 2 of the OSI model), derived from X.25. It was initially developed to allow amateurs to send data packets using regular radios. Devices that allow AX.25 communications are typically called Terminal Node Controllers (TNC). The main features of AX.25 are [9]:

- Synchronization flags `0x7E`, bit stuffing to avoid having flags inside the packet
- Persistent connection or packet transmission
- Source and destination addressing, with variable addresses length, multiple repeaters and basic routing capabilities

- Control packets: acknowledgement, ready, reject packet, ... possibly leading to retransmission
- Different possible frame types: Supervisory (control), Information (data with sequence numbers), Unnumbered (data without sequence number)
- Frame Check Sequence (FCS)
- Multiple protocols running on top of AX.25, using Protocol Identifier field (PID)

The AX.25 specification does not impose any physical layer (layer 1 of the OSI model) protocol or modulation. However, it is most of the times frequency modulated and transmitted in the amateur radio frequency bands available between 30 MHz and 3 GHz.

Any additional layers or application can be used on top of AX.25. Some common use cases are:

- Direct communication between 2 radio amateurs, without additional encapsulation
- Internet Protocol (IP) tunneling
- Automatic Packet Reporting Systems (APRS), usually to send GPS location at regular intervals. Often used in mobile vehicles like cars and boats. Can also transmit other information: status, weather, telemetries, ...

For OUFTI-1 and OUFTI-2, only the UI frame type is used (Unnumbered Information). Data inside AX.25 frames is encoded following the Telemetry and telecommand Packet Utilization Standard (PUS). It specifies the format for additional information, such as the service type, a timestamp, ... The exact format is not important for this work so it will not be detailed further. The address, control and PID fields are fixed. The AX.25 UI frame structure is shown in Figure 2.1.

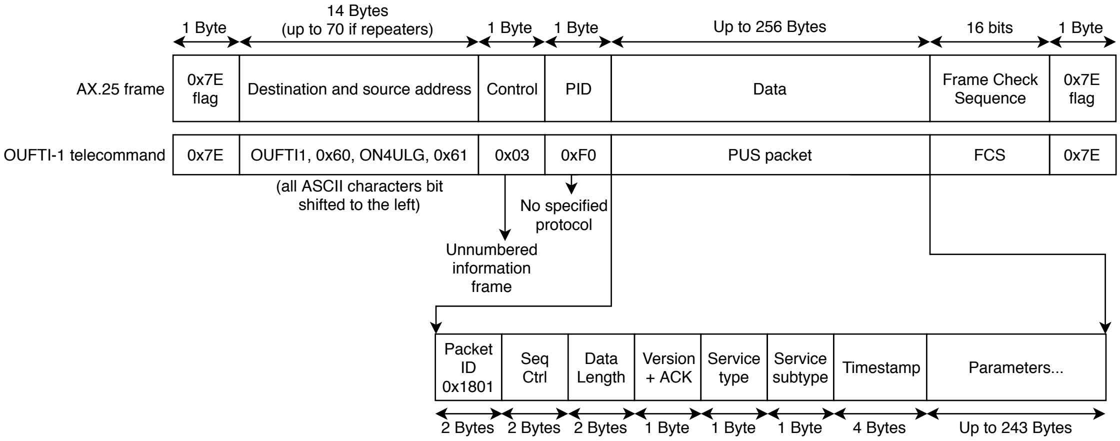


Figure 2.1 – Structure of the AX.25 Unnumbered Information frame in the case of OUFTI-1 and OUFTI-2 [8–11]

The FCS is a Cyclic Redundancy Check computed with the following 16 bits CCITT polynomial:

$$x^{16} + x^{12} + x^5 + 1$$

AX.25 bytes are sent Least Significant Bit (LSB) first, except for the Frame Check Sequence (FCS) which is sent MSB first.

Bit Stuffing

The `0x7E` flag serves as a synchronization pattern. It helps the receiver Phased Lock Loop (PLL) to synchronize its output with the incoming data. But it also acts as a Start Of Frame and End Of Frame delimiter. Therefore, it is important that it is not accidentally present inside the AX.25 packet itself. To avoid that, after every sequence of 5 `'1'`s, an extra `'0'` is added. As the flag consists of 6 consecutive `'1'`s, it is ensured to be the End Of Frame when came across. The reverse process is performed on the receiver side.

2.1.2 Uplink chain

Before tackling the specific synchronization issue, it is important to understand all the elements that make up the ground-to-satellite link. The complete data flow chain is shown in Figure 2.2. Each of the elements are described in details below.

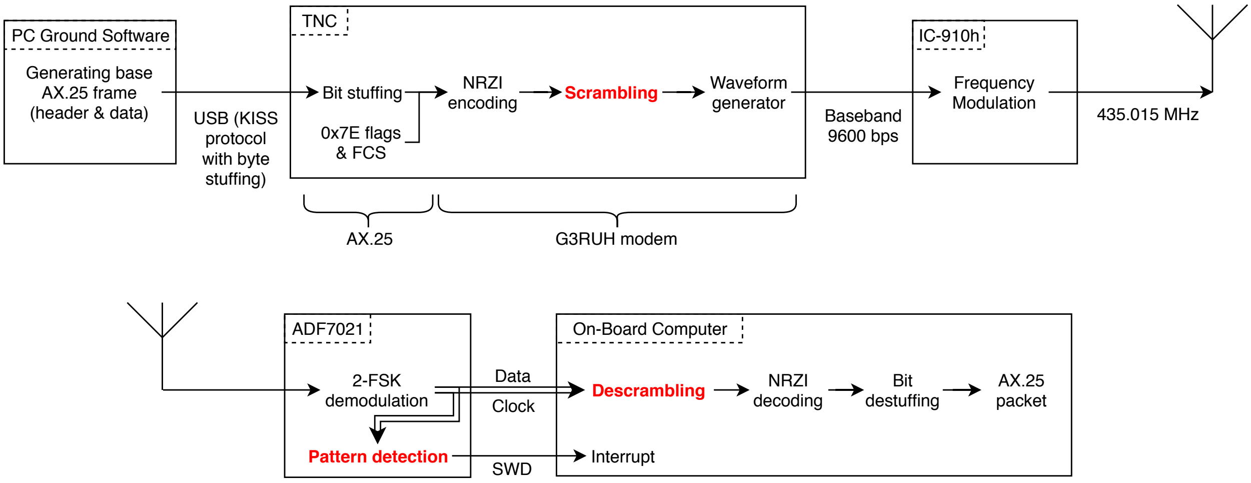
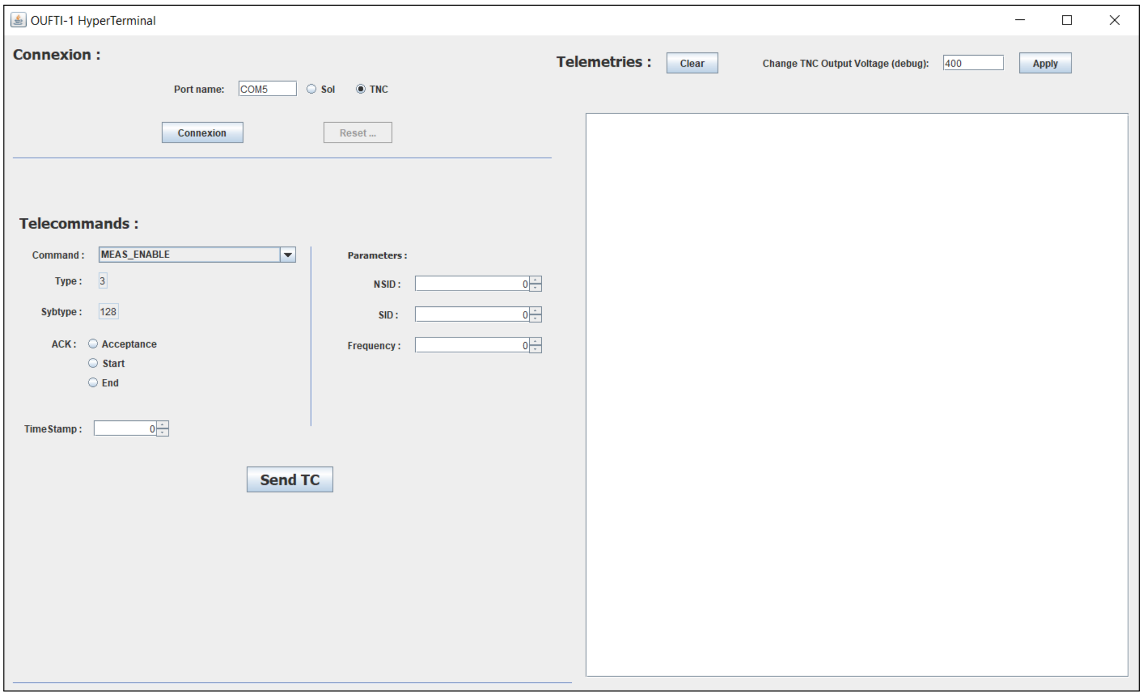


Figure 2.2 – Complete AX.25 uplink data flow (filters, amplifiers etc have been omitted for simplicity)

Computer

First, the useful data frame is created on a computer. A custom software has been developed to automate the creation of a valid packet [12]. It contains a Graphical User Interface (GUI), shown in Figure 2.3, on which the user can specify the type and parameters of the desired command. When the user presses the ”Send TC” button, the software makes the appropriate bit sequence and sends it through a USB port to the TNC. The configuration of the TNC is also determined and sent by the software.

For OUFTI-2, a very similar software is developed to accommodate for the new telecommands and telemetries. The communications with the TNC are identical.



OUFTI-1 HyperTerminal

Connexion :

Port name: COM5 ○ Sol ● TNC

Telecommands :

Command : MEAS_ENABLE

Type : 3

Sybtype : 128

ACK : ○ Acceptance ○ Start ○ End

TimeStamp : 0

Parameters :

NSID : 0

SID : 0

Frequency : 0

Telemetries :

Change TNC Output Voltage (debug): 400

Figure 2.3 – OUFTI-1 AX.25 ground software

TNC

A Terminal Node Controller (TNC) is a device that does the interface between the digital payload and the transceiver. The goal is to use a traditional analog radio to transmit digital information. It wraps the data inside an AX.25 packet and encodes it following a configurable operating mode. It then transmits the packet at a selectable baudrate and baseband modulation (On-Off Keying OOK, Audio Frequency Shift Keying AFSK, ...).

Some TNCs are also able to transmit APRS/GPS information, as explained in 2.1.1.

Figure 2.4 – Picture of the SCS DSP TNC. Image source: [13]

The TNC that was selected is the SCS Tracker / DSP TNC [8], presented in Figure 2.4. It is configured to transmit 9600 bits/s. At that baudrate, its user manual specifies that it will encode the AX.25 frame according to the G3RUH modem [14]. Basically, reversible operations are performed in order to improve the transmission. These are explained in section 2.1.3. The documentation available for this TNC is limited. Although it contains all the information required to configure and use it, it is missing implementation details that were important for this issue. These were obtained through observations, the results of which are presented in 2.3.

The settings, sent by the computer software to the TNC, are listed in Table 2.1.

Command	Description
@D1	Full duplex mode
@F1	Do not send flags during the pause
%B9600	9600 baud FSK mode (G3RUH modem)
X1	Push To Talk (PTT) enabled
%X400	Output of 400 mV peak-to-peak
T100	TX delay of 1 second
P128	Persistence (for CSMA)
W20	Slot time of 20 ms
O2	Maximum of 2 unacknowledged packets
R0	Digipeating disabled
@V0	Callsign check disabled
@K	Switch to KISS mode

Table 2.1 – TNC configuration parameters, unchanged from OUFTI-1 [10]

The following communications between the computer and the TNC are performed in "KISS" mode[1]. This format is common for that kind of equipment, so that they can be operated manually through a simple terminal. The frame is wrapped into `0xC0` flags and the "data frame" command (for port 0, as only one port is used) `0x00` is added at the beginning.

[1] Keep It Simple, Stupid

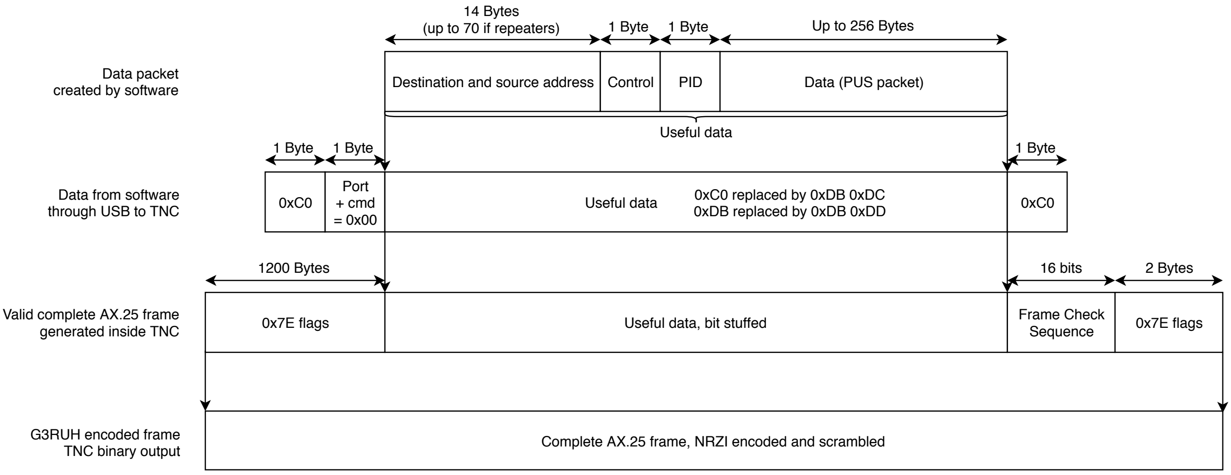
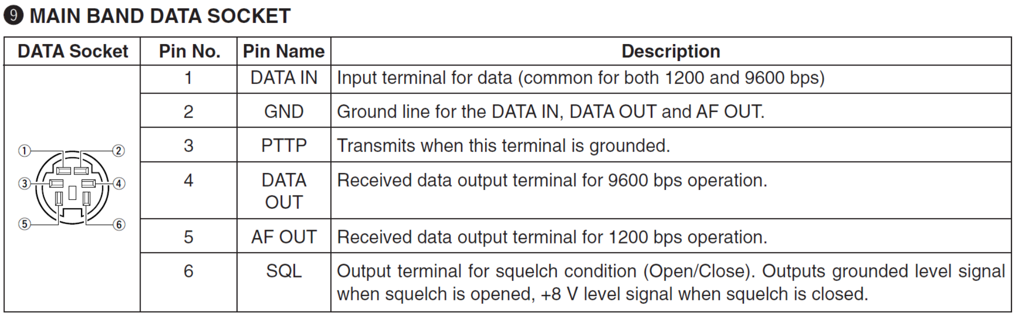


Figure 2.5 – Structure of the telecommand AX.25 frame at the different sending stages

All the manipulations performed on the packet are illustrated in Figure 2.5. The TNC outputs the encoded signal to a mini-DIN cable, which is standard for TNC to radio data links. The port 6 pins, see from the radio side, are detailed in Figure 2.6. When transmitting, the Push To Talk pin is grounded and the data is sent on DATA IN. A lot commercial amateur radio transceivers have this kind of connector, that can be selected instead of the traditional audio interfaces.



❾ **MAIN BAND DATA SOCKET**

DATA Socket	Pin No.	Pin Name	Description
	1	DATA IN	Input terminal for data (common for both 1200 and 9600 bps)
	2	GND	Ground line for the DATA IN, DATA OUT and AF OUT.
	3	PTTP	Transmits when this terminal is grounded.
	4	DATA OUT	Received data output terminal for 9600 bps operation.
	5	AF OUT	Received data output terminal for 1200 bps operation.
	6	SQL	Output terminal for squelch condition (Open/Close). Outputs grounded level signal when squelch is opened, +8 V level signal when squelch is closed.

Figure 2.6 – Pinout of the mini-DIN input of the IC-910h radio [15]

Radio

The radios used are ICOM's IC-910h and IC-9100, shown respectively in Figures 2.7 and 2.8. These amateur radio transceivers can handle AM, FM, SSB, CW and RTTY in the 144 MHz, 430 MHz and optional 1200 MHz bands [15, 16].

To send AX.25 telecommands to the satellite, the transceiver must be configured at 435.015 MHz, in FM data mode (FM-N on the IC-910h, FM-D on the IC-9100). The 9600 bps option must also be selected. Even though it is in data mode, the radio does not really expect binary data. Just like the regular FM mode, the input signal is used to directly modulate the frequency. The only effects of these 2 options (data mode and 9600 bps) are the input/output port selection, and some differences in the filters and amplifiers to optimize that kind of communications [15–17]. The RF output power level of the transceiver must also be appropriately tuned.

The IC-9100 carrier frequency can be tuned down to 10 Hz steps, which allows for precise Doppler effect compensation when communicating with a satellite. Both transceivers contains a satellite mode to easily control the carrier frequency correction. There exists various computer programs that can compute the position, relative velocity and subsequently Doppler effect of any given satellite.

Figure 2.7 – IC-910h radio. Image source: [18]

Figure 2.8 – IC-9100 radio. Image source: [18]

The output of the transceiver is then connected to a power amplifier and finally to the antenna.

ADF7021

The satellite's receiving antenna captures the signal, which goes through filters and the ADL5523 Low Noise Amplifier (LNA). It is then split in 2, filtered again and goes into the 2 ADF7021 receivers, respectively for AX.25 and D-STAR. The reason for this separation is explained in 1.2.

The ADF7021 is configured, in the AX.25 case, by the OBC. It performs the FSK demodulation and outputs the bit stream to the OBC.

The ADF7021 also contains a Sync Word Detect (SWD) circuit [7]. Basically, an output pin of the ADF7021 is set to high when a given sequence is received. By appropriately changing the contents on register 11 and 12, one can configure:

- The pattern to look for, which can be 12, 16, 20, or 24 bits long
- The amount of tolerated errors in the received sequence to trigger the SWD, between 0 and 3 inclusive
- The length during which the SWD pin will be held high after a sync word detection: 1 bit or a given ”packet length” (from 1 to 255 bytes)
- If the envelope detector threshold must lock after the reception of a sync word, and for how much time: 1 bit or a given ”packet length” (from 1 to 255 bytes)

The exact contents and their final value is given in Figures 2.23 and 2.24 of 2.3.4.

On-Board Computer

The On-Board Computer is responsible for the reception of telecommands, their scheduled execution, the gathering and transmission of telemetries, logging, and the general control of the other subsystems operations [12].

The On-Board Computer receives the data line and the clock from the ADF7021. It can then reverse all the encoding performed in the TNC to access the original data. Ideally, one would want the OBC to use the SWD output from the ADF7021 as an input pin, which would enable the decoding operations. But because of the scrambling, detailed in 2.1.3, it could not be done for OUFTI-1.

2.1.3 G3RUH modem

The G3RUH 9600 baud modem was developed by James Miller in 1988. Its goal is to make 9600 bauds data communication possible over a radio channel intended to transmit voice. It features 2 encoding steps, as well as a waveform generator [19].

Non Return to Zero Inverted (NRZI) encoding

First, the bits are modified as such: a `'1'` bit is encoded as a transition, and a `'0'` bit is encoded as no transition. With this simple encoding scheme, even if the signal is flipped, the data stays identical. It removes one source of error when using different equipment for FM communications. It also helps removing the DC component by balancing the 2 levels if there are more `'0'`s than `'1'`s or vice-versa in the original data. However, transmitting a long sequence of `'0'`s will not create any transition. This is why the signal is also scrambled.

Scrambling

The scrambler uses a 17 bits long shift register and 3 XOR gates, as illustrated in Figures 2.9 and 2.10. It can be summarized into the following polynomial:

$$1 + x^{12} + x^{17}$$

Its goal is to further reduce the DC component by avoid long sequences of unchanged level, in the case where the useful data contains a large amount of consecutive `'0'`s. It statistically ensures that the output signal does not stay too long without transition. Decreasing the DC component as much as possible is very important in FSK, as the receiver's Phase-Locked Loop (PLL) output frequency may drift from the real carrier frequency if the signal does not oscillate enough around it.

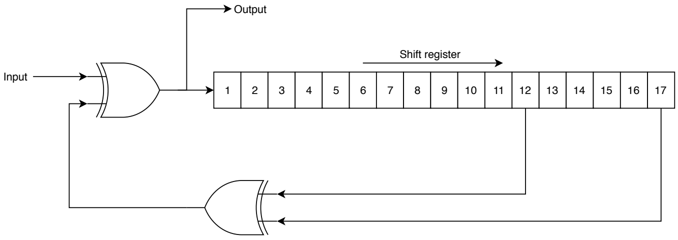


Figure 2.9 – Scrambler diagram

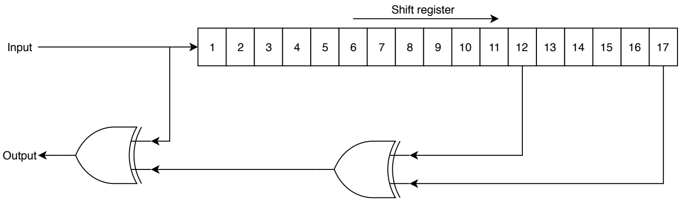


Figure 2.10 – Descrambler diagram

The shift registers of the scrambler and descrambler will naturally synchronize after having received at most 17 bits. When the shift registers are synchronized, the descrambler can work properly. This means that the descrambler needs at least 17 synchronization bits before outputting the correct data. At that point, it can already be assumed that the TNC adds synchronization bits before the flag for that purpose. In any case, synchronization bits are required for the receiver PLL to match the transmitter carrier frequency. The presence of a large amount of flags to perform both these synchronizations is confirmed to be sent for the downlink communications, as it was verified in OUFTI-1's OBC source code. The same behavior will also be observed from the TNC in 2.3.

Waveform generator

After these encoding steps, the TNC could naively output the binary stream to the transmitter, which would modulate it in 2-FSK. However, the theoretically instantaneous changes in the signal frequency would create a lot of higher frequency components, resulting in a much larger signal bandwidth than acceptable and distortion in the channel.

To prevent that, the TNC digitally filters the signal. It modulates each bit using a shape that compensates for the response of the channel, and that should theoretically result in a Nyquist pulse at the output of the receiver. This means that there should be no interference between bits (InterSymbol Interference ISI) [22, 23]. Examples of Nyquist pulses are the raised cosine impulse and the cardinal sine (sinc). A signal which is the sum of Nyquist pulses will theoretically be bounded in the frequency domain and stay inside a limited bandwidth, as showed in Figure 2.11. Because a Nyquist pulse has no ISI, the received signal should always be equal to +1 or -1 at the sampling points, which is verified in Figure 2.12.

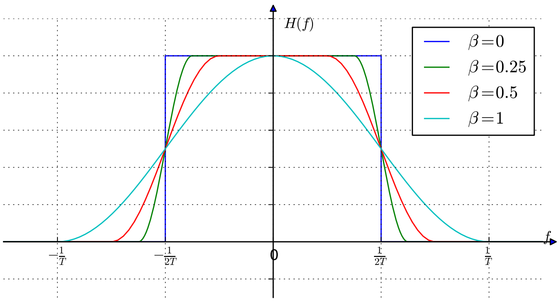


Figure 2.11 – Raised cosines in the frequency domain. Image source: [20]

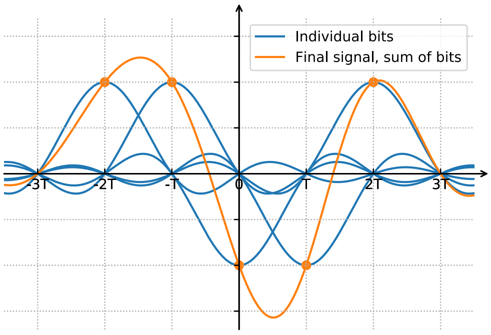


Figure 2.12 – Example of Nyquist pulses signal illustrating the absence of ISI. Image source: [21]

As the channel and receiver compensation cannot be predicted in advance, TNCs often offer multiple configurable waveform options. The result will never be perfect but the quality of the signal is still improved.

2.1.4 The scrambled SWD issue

The issue in our case is that the scrambler's output in unpredictable. The only way to predict what its output will be is to know the contents of the shift register, which is normally obtained by receiving at least 17 bits. Its could be deducted by knowing its initial value and all the sent data. The sent data is constant as it contains AX.25 flags, but the G3RUH modem specifies that the initial value can be arbitrary. Therefore, it can be random, and that's what was assumed in [8].

So even though one wants to use the AX.25 flags as a pattern, it is theoretically not possible to predict what they will be transformed to when scrambled. That's why some processing is required to continuously descramble incoming data in order to detect the pattern, which was previously done by the OBC. But this means that the OBC is repeatedly interrupted 9600 times per second to perform a task that is useful only a handful of times per day, which is not optimal power-wise.

2.2 Considered potential solutions

During the research, many ideas emerged to find a workaround. The most sensible ones are listed below.

- Predict or force the initial value of shift register
  Maybe the initial value of the shift register is kept or reset to a constant between different transmissions or when the TNC is powered off. Some dummy data could be sent to put the shift register to a known state before sending the real data for

example. Or it is possibly not random at all, therefore make the solution to this problem straightforward. It is discovered to actually be the case in 2.3.3.

- Replace the TNC by another available hardware TNC, a software TNC, or a custom solution
  The only feature of the TNC that makes specific hardware useful is the Nyquist pulses generator. Other than that, the TNC only performs bit operations that can easily be implemented on the computer software.
  There exists many software TNCs that take advantage of the sound card of the computer to output the signal. This solution was rejected in 2008 [8] because existing software at that time was not easily configurable and did not perform well, and because basic sound cards were not expected to be able to produce an output signal of the same quality as hardware TNCs.
  However, software TNCs have evolved a lot since then. A good example is Direwolf [24]. It is an open-source TNC that includes the necessary features for this application. It is clear in the source code that the scrambler shift register is always initialized to 0, which makes its output totally predictable and solves the issue.

- Bypass the TNC to send the synchronization pattern that wakes up the OBC, then use the TNC to send the effective packet. This would have been simpler than a complete custom hardware TNC. As the pattern can be repeated a high number of times to be detected, this solution would not require the same level of quality as a TNC, and could potentially work without a wavefunction generator.

Some alternative solutions, that also have been considered at some point, are listed for the sake of completeness.

- Capture the TNC output continuously to retrieve the shift register value, and use as described in the first possible solution
  This is not really possible or useful, as the TNC waits to have received the complete frame from the computer before starting the transmission

- If the TNC uses a hardware shift register, like the original G3RUH modem [19], it could be probed or reset externally. It is very unlikely though, as the SCS DSP TNC contains a microcontroller that most probably performs all the bit encoding operations

- Access or change the firmware of the TNC. Asking the TNC's manufacturer for information on how the shift register is initialized was possible.

- Having a microcontroller, CPLD/FPGA, or combinational circuit on-board, that continuously descrambles the data between the ADF7021 receiver and the OBC, and performs the pattern detection. But that would add unnecessary complexity to the satellite and would defeat the purpose of removing it from the OBC.

2.3 Observations and solution

Before implementing any solution, a better understanding of the practical behavior of the current TNC is required.

For the purpose of this work, the computer software has been modified in the following ways:

- The packets contain an auto-incrementing sequence number. It was kept to a constant value to be able to send exactly the same data multiple times
- The useful frame is outputted before being sent, so that it can be compared to decoded values during the tests
- Automation of the sending of a large amount of identical packets in a row

The output of the TNC has been observed using an oscilloscope, by stripping down one end of a mini-DIN cable. The beginning of a frame is shown in Figure 2.13. Although nothing can really be concluded from that observation, one can still note the Nyquist pulses that are clearly visible, and the 9600 bits per second frequency.

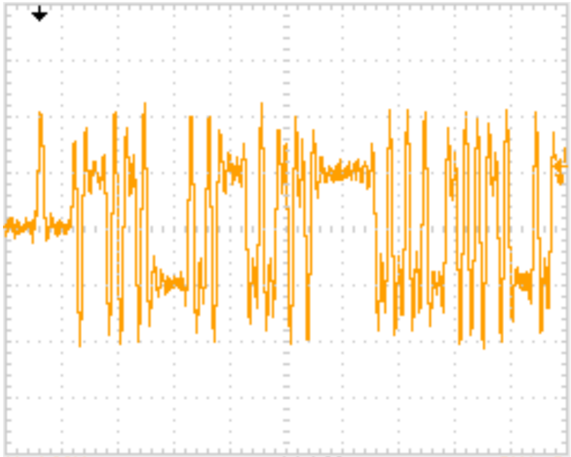


Figure 2.13 – Observation of the TNC output at the beginning of a frame

2.3.1 Sampling

To get real insight on this particular TNC scrambler implementation, a complete frame should be analyzed. The cut end of the mini-DIN cable was attached via the DATA IN and GND wires to a 3.5 mm jack connector, as shown in Figure 2.14. The sound card of a computer was then used to record the signal. With its sampling frequency of 48 kHz, it should be able to decode the signal without any issue. The setup is shown in Figure 2.15 and the result in Figures 2.16 and 2.17. The complete frame is around 1 second long.

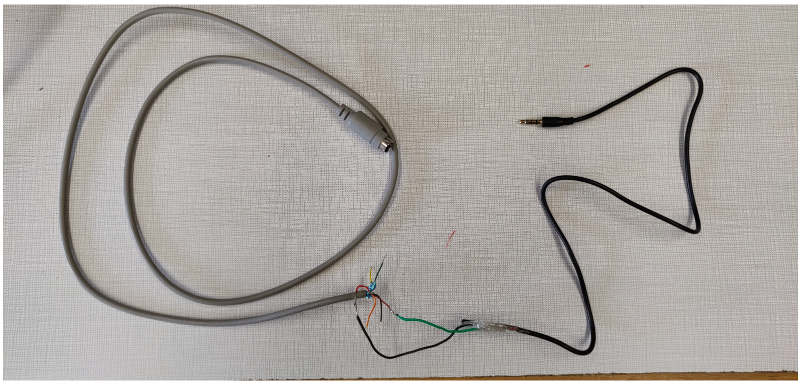


Figure 2.14 – Home-made mini-DIN to 3.5 mm audio jack adapter

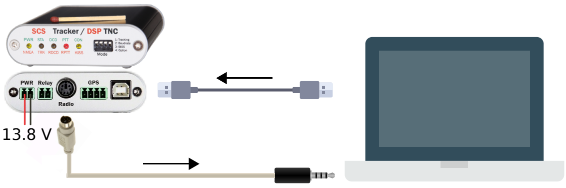


Figure 2.15 – TNC output sampling setup. Image sources [13, 25–27]

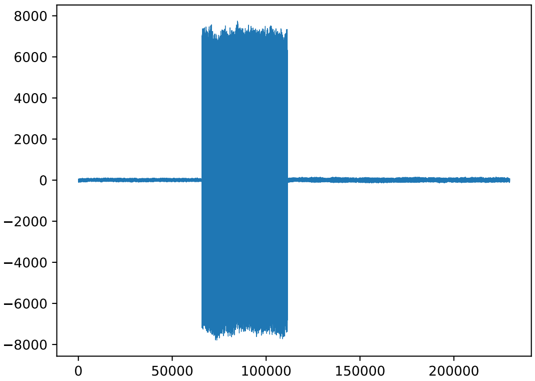


Figure 2.16 – Full recording of the TNC output

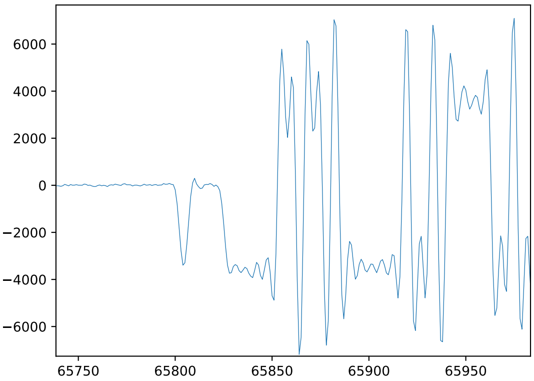


Figure 2.17 – Beginning of the transmission in the TNC output recording

This signal must then be sampled to recover each individual bits. A simple Python script has been written, that samples the signal at 9600 bits/s, effectively taking one out of $\frac{48000}{9600} = 5$ samples. The sign of the samples gives the corresponding bit value. It takes as input the audio file and the offset of the first bit, and returns the raw bit sequence. A few sampling points are shown in Figure 2.18.

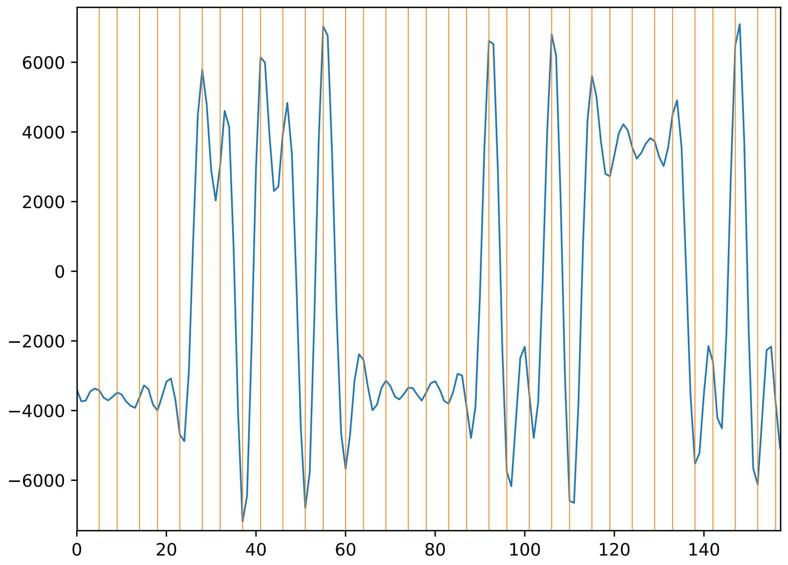
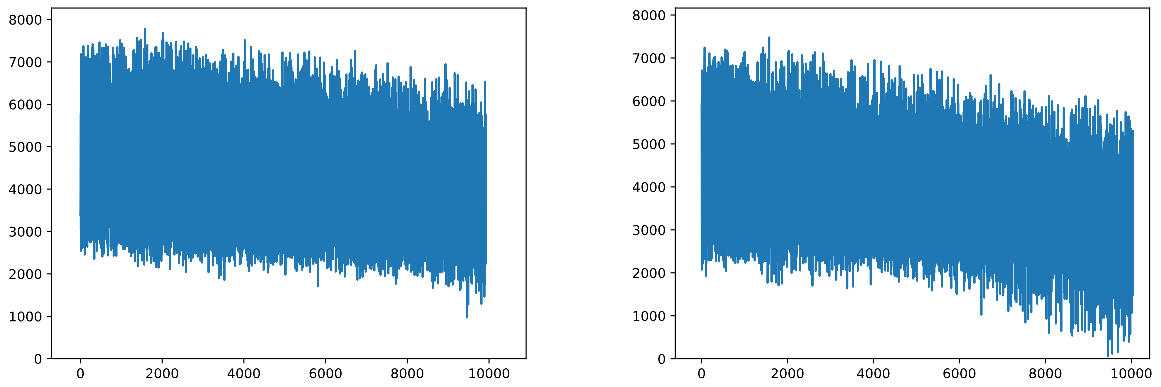


Figure 2.18 – Sampling of the recorded output signal from the TNC

Because the sound card real sampling frequency is not a perfect multiple of the TNC real bit rate, the sampling times eventually drift. After a certain amount of bits, the sign of the sampled values may no longer reflect the correct bit. To check that this doesn't happen during the sampling of the frame, one could visually inspect the sampling points over the whole signal. Though, a much more reliable approach is to compute the absolute value of each sample, shown in Figure 2.19. The margin must be sufficient to ensure the sampled data is correct. This must be verified for each test recording.



(a) Valid sampling

(b) Invalid sampling

Figure 2.19 – Absolute value of the signal at the sample points, showing drift over time

2.3.2 Decoding

The raw bits can then be descrambled, NRZI decoded and bit destuffed. The initial packet is successfully recovered. The transmitted and decoded received frame are showed respectively in Figures 2.20 and 2.21.

As expected, the TNC sends around 1200 `0x7E` flags before the beginning of the data. This is consistent with the 1 second TX delay in the TNC settings shown in Table 2.1. It also sends 2 flags at the end. There are a few bits at the beginning and at the end of the transmission which are not well understood. They are variable even when the frame is exactly the same. They are not part of the scrambled sequence, as the following flags are scrambled independently of them. They will be ignored.

2.3.3 Shift register value

As the descrambling is performed by a custom script, the contents of the shift register can be easily accessed at any point during the process. The comparison point is chosen at the first useful bit, just after the flags. This point is not chosen at the beginning of the flags as at least 17 bits are required to synchronize the shift register. Moreover, this reference point will is more consistent when the amount of unknown starting bits changes.

A total of 14 different recordings were taken, in different situations: after a power-down or after other transmissions, with packets of different length and contents. Each time, the shift register at the reference point had the same value: `110111100011111`.

The state of the shift register, as well as the output, only depend on the initial shift register value and the previous sent data. Before the reference point, the data already sent is always the same as it is just composed of 0x7E flags. This means that the initial value of the shift register is always the same for that TNC.

With that in mind, it is then clear that the flags are always scrambled the same way. So even though the 0x7E flags can't directly be used as a pattern to recognize, the 9600 bits long sequence of the scrambled flags is constant and known. It is indeed verified on the raw sampled test packets. So any 12, 16, 20 or 24 bits long pattern can be selected inside it.

This pattern should preferably be chosen near the end of the flags sequence, to let enough time for the ADF7021 receiver's Phased Lock Loop to synchronize with the signal. At least 17 bits should also be left before the last flag, so that the OBC descrambler can get the correct descrambler shift register value beforehand. A possible pattern verifying these conditions has been selected: `010001010010100011110000`.

```
Sending 1/1
RAW frame to send :
0x9EAA8CA89262609E9C68AA988E6103F01801C000000A1003800000000000101000165AD
frame sent :
0xC0009EAA8CA89262609E9C68AA988E6103F01801C000000A10038000000000001010001
65ADC000000000000000000000000000000000000000000000000...
```

Figure 2.20 – Console output of the modified software sending the frame to the TNC

```
test1
9920 samples, min abs 971, max abs 7783, min index 9456
Raw [0:100]
000000110110100000001001011111001001101101111111111100100100100011010101
1010000100111000110010110001
Descrambled [0:100]
111010101011011000111110010010000000100000001000000010000000100000001000
0000100000001000000010000000
NRZI [0:100]
110000000010010110111101001001111110011111100111111001111110011111100111
1110011111100111111001111110
1199 flags, 9592 flags bits, first flag index 28
Shift register data beginning 110111100011111
After flags
011110010101010100110001000101010100100101000110000001100111100100111001
000101100101010100011001011100011000011011000000000011110001100010000000
000000110000000000000000001010000000001000110000000000000010000000000000000
000000000000000010000000100000000000000010000000101001101011010101111110
011111100110
Data length 280, remaining flags bits 20
Destuffed
011110010101010100110001000101010100100101000110000001100111100100111001
000101100101010100011001011100011000011011000000000011110001100010000000
000000110000000000000000001010000000001000110000000000000010000000000000000
0000000000000000100000001000000000000000100000001010011010110101
Flipped (LSB first)
100111101010101010001100101010001001001001100010011000001001111010011100
011010001010101010011000100011100110000100000011111100000001100000000001
110000000000000000000000000101000010000000000011100000000000000000000000
0000000000000000000000001000000010000000000000000101100101101011 01
Final data
0x9eaa8ca89262609e9c68aa988e6103f01801c000000a1003800000000000101000165ad
```

Figure 2.21 – Console output of the Python script sampling and decoding the recorded frame

2.3.4 Test of the solution

The ADF7021 is configured using a simple Arduino script. Wiring is done following the development board pinout of Table 2.2. The contents of the registers to send is copied from OUFTI-1's code. Additionally, the selected sync word has been placed inside Register 11 in the ADF7021's configuration, as shown in Figures 2.23 and 2.24. Figure 2.22 shows the test setup, where the computer, TNC and IC-910h are used to generate, encode and transmit real AX.25 frames. Meanwhile, the ADF7021's Sync Word Detect output pin is monitored.

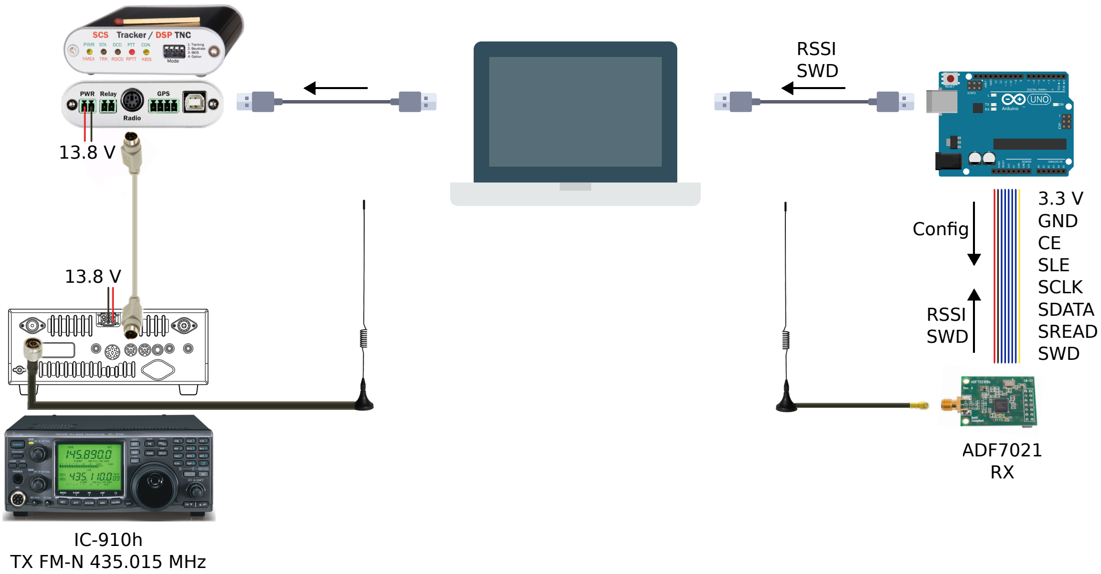
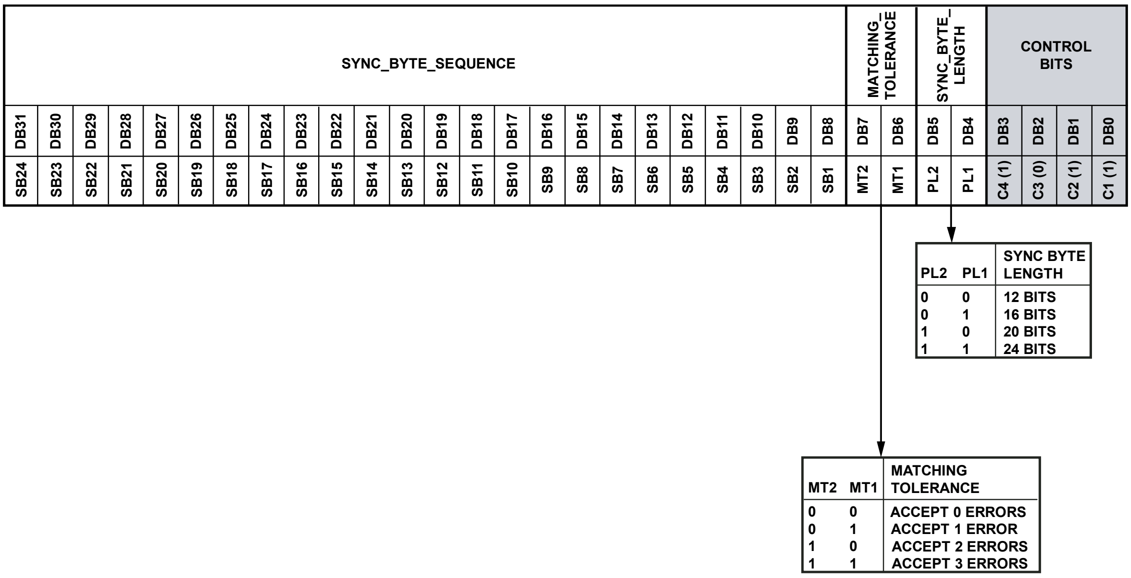


Figure 2.22 – SWD test setup (signals from Arduino to ADF7021 must be shifted from 5V to 3.3V). Image sources: [13, 18, 26–30]

After an unsuccessful attempt, it eventually worked with all the bits of the pattern flipped. This is because the output of the TNC has been chosen, arbitrarily, to be interpreted as a `'1'` when positive and `'0'` when negative. After being modulated in FSK, this is the opposite as what the ADF7021 considers as a `'1'` or a `'0'`. It is not an issue with the useful data, which is encoded in NRZI so only the transitions are important, but it has to be taken care of for the raw pattern detection.

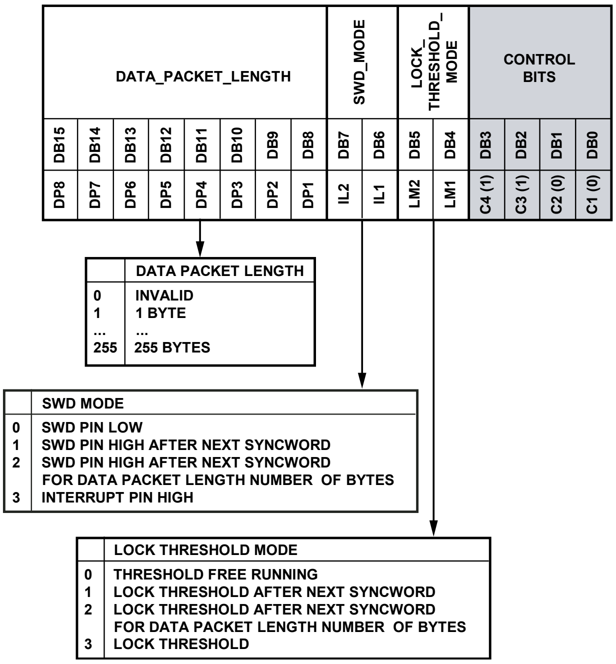
The same test was also successfully performed with another TNC of the same model, to make sure that the scrambling behavior was consistent and could be reproduced without trouble.


SYNC_BYTE_SEQUENCE
MATCHING_ TOLERANCE
SYNC_BYTE_ LENGTH
CONTROL BITS
DB31 DB30 DB29 DB28 DB27 DB26 DB25 DB24 DB23 DB22 DB21 DB20 DB19 DB18 DB17 DB16 DB15 DB14 DB13 DB12 DB11 DB10 DB9 DB8 DB7 DB6 DB5 DB4 DB3 DB2 DB1 DB0
SB24 SB23 SB22 SB21 SB20 SB19 SB18 SB17 SB16 SB15 SB14 SB13 SB12 SB11 SB10 SB9 SB8 SB7 SB6 SB5 SB4 SB3 SB2 SB1 MT2 MT1 PL2 PL1 C4 (1) C3 (0) C2 (1) C1 (1)

PL2	PL1	SYNC BYTE LENGTH
0	0	12 BITS
0	1	16 BITS
1	0	20 BITS
1	1	24 BITS

MT2	MT1	MATCHING TOLERANCE
0	0	ACCEPT 0 ERRORS
0	1	ACCEPT 1 ERROR
1	0	ACCEPT 2 ERRORS
1	1	ACCEPT 3 ERRORS

Figure 2.23 – Register 11 - Sync Word Detect Register, from ADF7021 datasheet [7] Contents for OUFTI-2: `10111010 11010111 00001111 00111011`


DATA_PACKET_LENGTH
SWD_MODE
LOCK_ THRESHOLD_ MODE
CONTROL BITS
DB15 DB14 DB13 DB12 DB11 DB10 DB9 DB8 DB7 DB6 DB5 DB4 DB3 DB2 DB1 DB0
DP8 DP7 DP6 DP5 DP4 DP3 DP2 DP1 IL2 IL1 LM2 LM1 C4 (1) C3 (1) C2 (0) C1 (0)

	DATA PACKET LENGTH
0	INVALID
1	1 BYTE
...	...
255	255 BYTES

	SWD MODE
0	SWD PIN LOW
1	SWD PIN HIGH AFTER NEXT SYNCWORD
2	SWD PIN HIGH AFTER NEXT SYNCWORD FOR DATA PACKET LENGTH NUMBER OF BYTES
3	INTERRUPT PIN HIGH

	LOCK THRESHOLD MODE
0	THRESHOLD FREE RUNNING
1	LOCK THRESHOLD AFTER NEXT SYNCWORD
2	LOCK THRESHOLD AFTER NEXT SYNCWORD FOR DATA PACKET LENGTH NUMBER OF BYTES
3	LOCK THRESHOLD

Figure 2.24 – Register 12 - SWD/Threshold Setup Register, from ADF7021 datasheet [7] Contents for OUFTI-2: `11111111 10001100`

Dev. board pin no.	Chip pin no.	Pin function
1		VDD (3.3V)
2		GND
3	37	MUXOUT
4	35	TxRxCLK
5	34	TxRxDATA
6	33	SWD
7	28	SCLK
8	27	SREAD
9	26	SDATA
10	25	SLE
11	24	CE
12		GND

Table 2.2 – ADF7021 development board pin attribution

2.4 Validation

The system works as expected, but the test explained in 2.3.4 was performed in laboratory conditions, which are very different from what the satellite will encounter in space. Namely, the reception power will be much lower in space, in the order of -80 dBm instead of -50 dBm.

It should be noted that this evaluation is only performed as a verification. The ADF7021's Sync Word Detect (SWD) system is purely digital. The ADF7021 continuously demodulates the data, which is always available on the RX output pin. The SWD circuit only pushes the same data into a shift register, which is then compared to the stored pattern. It is safe to assume that if the SWD system has been verified to work at high power, and if the ADF7021 is able to correctly receive some data at some lower power, then the SWD will also work at said lower power. The proper functioning of the SWD depends on the quality of the demodulated output signal of the ADF7021, and not directly on the input signal. If we take into account that the ADF7021 reception has already been thoroughly tested in space conditions for OUFTI-1 [31], the SWD is ensured to work. However, to be on the safe side, it has been decided to perform this evaluation anyway.

The maximum RF output power of the IC-9100 is 75W in the 430 MHz band [16]. Using an external power amplifier, it was estimated that the final output power would be 100 W. In the AX.25 uplink power budget presented in [32], the power level received by the spacecraft is computed. This gives a resulting power at the input of the satellite Low Noise Amplifier of -129.1 dBW = -99.1 dBm. The receiver noise power is also given and is -165 dBW = -135 dBm.

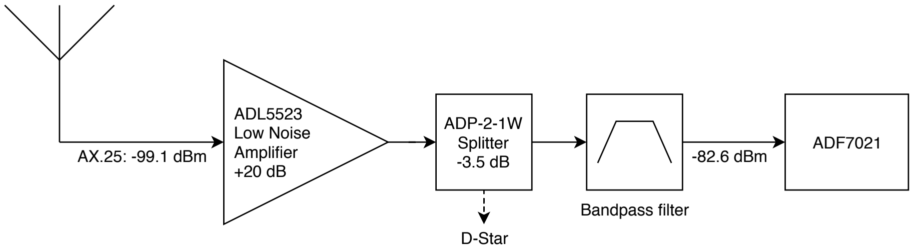


Figure 2.25 – AX.25 reception RF path, with power levels for 100 W emission [32], see Figure 1.3 for complete RF path

The different stages amplifying and attenuating the signal are depicted in Figure 2.25. The ADL5523 Low Noise Amplifier (LNA) amplifies the signal by 20 dB [33]. It then goes through a splitter, of which the 2 outputs respectively go into the AX.25 and the D-STAR ADF7021s. It attenuates the signal by 3.5 dB. The reason for this splitter was already explained in 2.1.2. When entering the ADF7021, the level of the signal is -82.6 dBm and the level of the noise is around 118.5 dBm. To perform the test on ground, multiple setups have been used but the received signal always directly goes in the ADF7021's input. The goal is to match the -82.6 dBm power level. For information, the sensitivity of the ADF7021 in these conditions is -114 dBm [7].

2.4.1 RSSI measurement

The ADF7021 is able to measure the received power level. To access it, one must send the appropriate value to Register 7, and then read what that ADF7021 send back on the SREAD pin. After a few simple calculations explained in the datasheet, the RSSI can easily be retrieved. The Arduino script that already configures the ADF7021 has been modified to regularly take an RSSI measurement and send it to the computer. The resulting graph is shown in Figure 2.26.

The RSSI given by this method has been compared against the one given by a spectrum analyzer, as shown in Figure 2.27. And this at several occasions during the following tests, in wire and antenna configurations and at different power levels. The difference between the levels was at most of the order of 3 – 4 dBm. Therefore, the RSSI given by the ADF7021 can be considered as reliable.

2.4.2 Sync Word Detection rate test setup

A simple way to properly evaluate if the SWD works properly is to take the setup of 2.3.4. The software is now configured to send a series of 100 packets with a 2s interval. The Arduino script, after having configured the ADF7021, regularly takes an RSSI measurement and continuously monitors the SWD pin to count the number of times it is triggered.

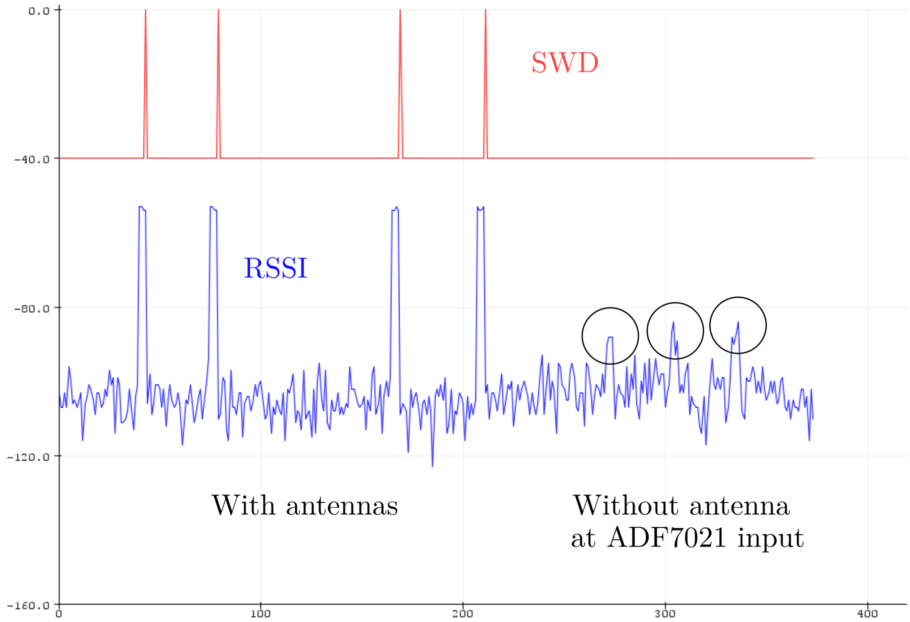


Figure 2.26 – Arduino serial tracer showing the RSSI and SWD in real-time

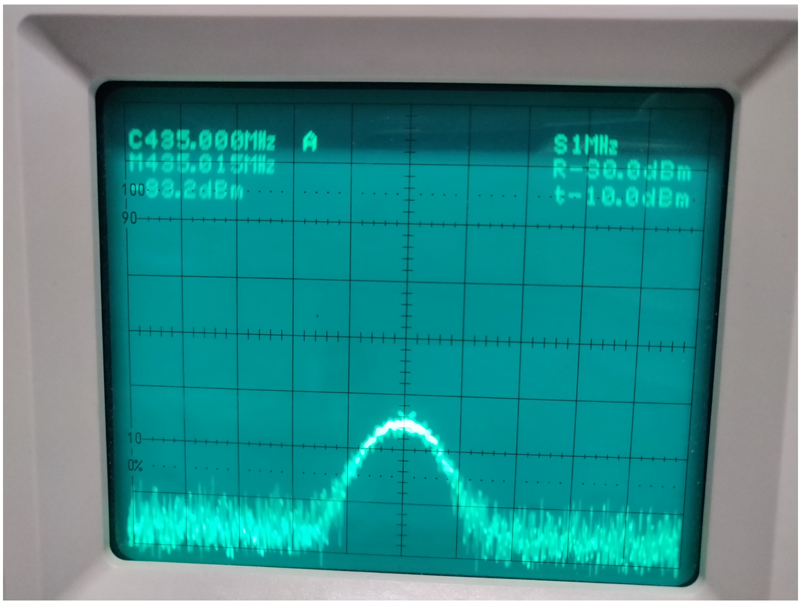


Figure 2.27 – Spectrum analyzer measurement of the RSSI in the configuration of Figure 2.32, the ADF7021 outputting approximately the same RSSI value

Many tests were performed using different setups to try to match the -82.6 dBm theoretical power at the input of the ADF7021. Some of them showed a very variable measured RSSI, or the same results as with similar power/setups. That is the reason why only part of the tests were reported to illustrate each method.

Test with antennas in the same room

To have a working reference as a starting point, tests were performed using close antennas for both transmission and reception. Even with the RF output power of the IC-910h at the minimum, the ADF7021 input power is still obviously very high. These first results, presented in Table 2.3, show that in ideal RF conditions, the SWD works every time.

Setup	Measured RSSI	Detection rate	
Antenna - Antenna in the same room	-50 dBm	100/100	100%
Receiving antenna without ground plane	-63 dBm	100/100	100%
With -20 dB attenuator	-68 dBm	100/100	100%
With -40 dB attenuator	-80 dBm	87/100	87%

Table 2.3 – Sync Word Detection rate test results for antennas in the same room (IC-910h)

An unexpected phenomenon can already be observed. The measured RSSI only decreases by 12 dB when a -20 dB attenuator is added. This kind of non-idealities will be encountered again and some possible explanations will be given in the wired tests below.

Wired test with attenuators

The power obtained using antennas varies a lot over time and when moving the antennas around. The RF characteristics of the signal are difficult to monitor and control. Unknown multi-path effects, due to reflections on the nearby environment, are non-negligible at that small scale. To correctly assess the detection rate to input power relation, a reproducible, consistent, test setup is preferable. A chain of attenuators, shown in Figure 2.28, was used to achieve a completely wired configuration, illustrated in Figure 2.29. As most attenuators can only withstand up to 1 W and the minimum output power of the IC-910h is 5 W, the first one is a *R415610000* -10 dB attenuator with a maximum input power of 15 W. Then, several attenuators (up to -20 dB each) are combined to get to the desired power level.

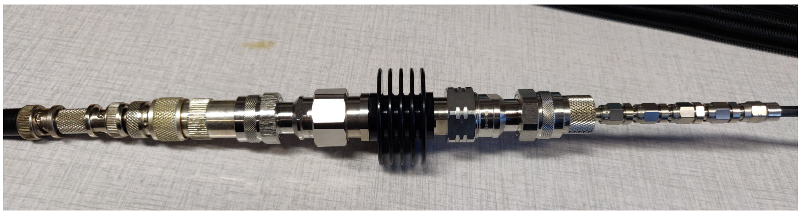


Figure 2.28 – Attenuator chain with the necessary adapters

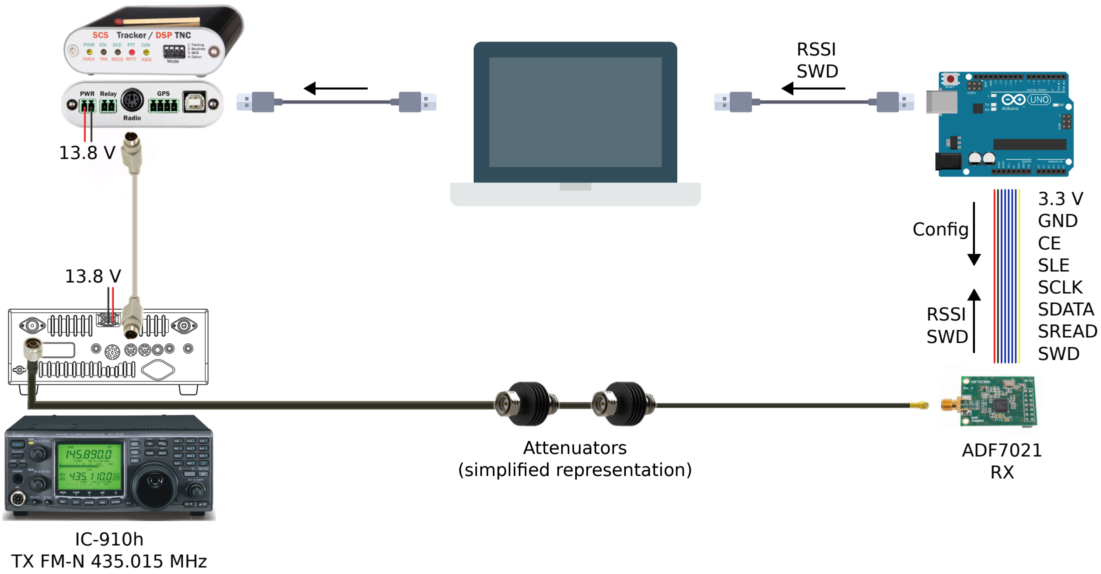


Figure 2.29 – Sync Word Detect wired test setup (signals from Arduino to ADF7021 must be shifted from 5V to 3.3V). Image sources: [13, 18, 26–29]

As shown in Table 2.4, these tests could not confirm that the system works. As soon as -60 dBm, the SWD starts to fail. An interesting observation is that the measured RSSI begins to rise again when adding more attenuators at some point. This probably means that there are RF interactions between the attenuators or with the transmitter or receiver. For example, the first -10 dB attenuator may radiate some power that will be strong enough to be picked up by another one that works at much lower power. The measured RSSI is indeed very unstable, and varies when the attenuators are moved around and when they are held in hand, sometimes changing from around -70 dBm to -100 dBm. This is not a normal behavior for these attenuators, but it is difficult to diagnose its cause. If one of the attenuators, adapters or cable was malfunctioning, it could have been identified using a VNA (Vector Network Analyzer) to measure the S parameters of the chain. These checks were not performed.

Setup	Theoretical RSSI	Measured RSSI	Detection rate	
Wired with -90 dB	-53 dBm	-55 dBm	100/100	100%
Wired with -100 dB	-63 dBm		96/100	96%
Wired with -110 dB	-73 dBm	-72 dBm	194/200	97%
Wired with -120 dB	-83 dBm	-65 dBm	281/300	94%
Wired with -123 dB	-86 dBm		188/200	94%
Wired with -126 dB	-89 dBm		97/100	97%
Wired with -130 dB	-93 dBm		89/100	89%

Table 2.4 – Sync Word Detection rate test results for wired link with attenuators (IC-910h)

Wired test in a TEM-Cell

In order to mitigate these RF interactions that are the likely cause of the bad results, they were performed again in the TEM (Transverse Electromagnetic) cell of the Electronics Department of the HEPL Industrial School (ISIL). This cell works for electromagnetic waves up to 300 MHz. Therefore, it is not supposed to work perfectly at 435 MHz but should still attenuate the RF signal. As it was easily available, it was tried anyway, with the setup of Figure 2.30.

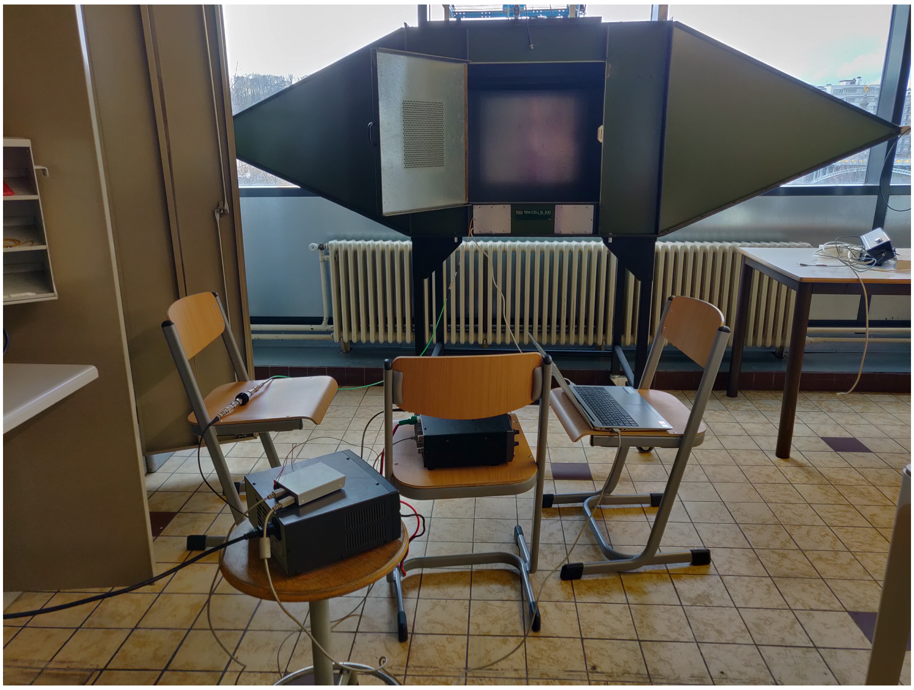


Figure 2.30 – Setup for the wired test in a TEM-Cell (the cell was closed during the tests)

Multiple configurations have been tested:

- Only all attenuators inside
- Only some (the first or last few) attenuators inside
- Only the receiving ADF7021 inside
- Only the receiving ADF7021 inside, with all or the last few attenuators

None of them worked any better than without the TEM-Cell. Changing the position of the attenuators inside the cell was still creating huge differences in the measured power.

Test with antennas at different locations

In order to avoid the interactions and the multi-path effect happening because of the proximity of the transmitter and receiver, the 2 antennas approach was brought back. This time, the low power levels were obtained by physically separating the antennas in different parts of the building.

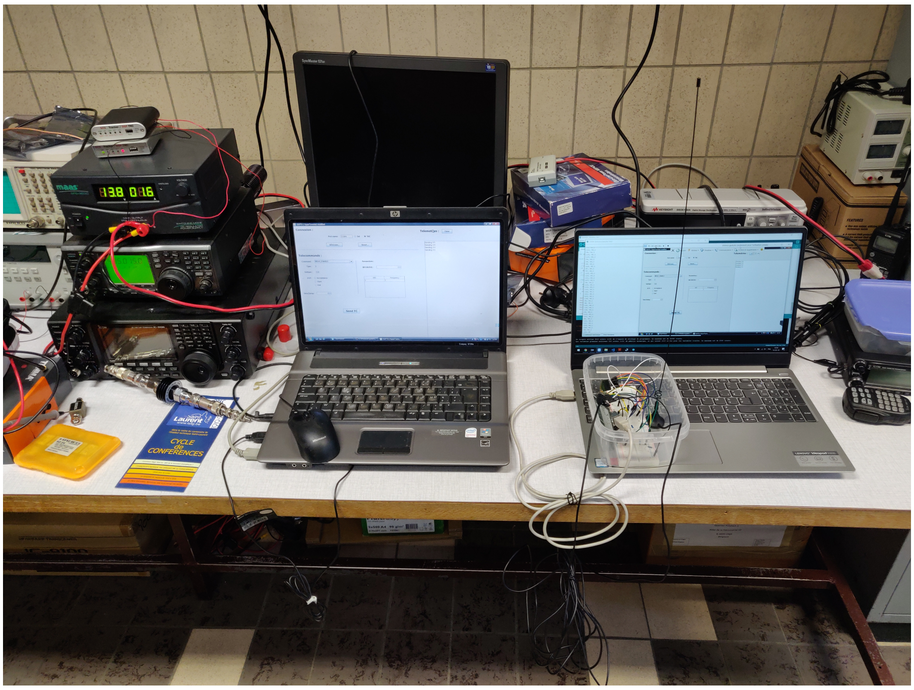


Figure 2.31 – Setup for the test with antennas at different locations. Left: Transmitting PC. Right: Moving receiving PC, controlling the left one

The test setup is, this time, made up of 2 different computers, as shown in Figure 2.31. The one which will be moving with the receiving ADF7021 has control over the transmitter one through *TeamViewer*. This allows to operate the transmission from the mobile location. The measurements of Table 2.5 were taken from different places in the HEPL (ISIL) building, and from different floors. The maximum distance is estimated to be around 60 – 80 meters[2]. In most cases, there were several walls between the transmitter and receiver. Again, due to the configuration of the building, the RSSI was varying a lot, probably indicating strong reflections. The results seem more reliable but there are still a lot of missing SWDs. The detection rate does not seem to be as dependent on the RSSI as before.

[2] Distance measured on Google Maps satellite view

Measured RSSI	Detection rate	
-47 dBm	95/100	95%
-55 dBm	98/100	98%
-60 dBm	92/100	92%
-70 dBm	94/100	94%

Table 2.5 – Sync Word Detection rate test results for antennas at different locations (IC-910h)

To decrease further more the received power level, the transmitting antenna was replaced with a dummy 50 Ω load. The power levels were lower of a few dB while still way above -80 dBm. The detection rates were similar.

2.4.3 Bit Error Rate hypothesis verification

After the unsuccessful described above, it was unclear what the exact cause of the bad detection rates was. Is is correlated with the RSSI, which theoretically isolates the analog parts (transmitter, channel and receiver) as the only possible suspects. The goal is to characterize the SWD system of the ADF7021, which relies on the receiver. The Bit Error Rate (BER) must be measured to verify that the errors are already present at the output of the receiver, and thus are not caused by SWD system.

This is even more concerning knowing that the descrambling process will inevitably multiply the amount of errors, and that AX.25 does not include any error correction mechanism.

As the D-STAR repeater was developed in parallel, the new tests were performed using the D-STAR microcontroller, instead of the Arduino. The of the microcontroller was modified to configure the ADF7021 with the AX.25 registers, count the number of Sync Word Detections, monitor the RSSI and compare the first 336 bytes received with the given reference AX.25 packet.

As a side note, this test also successfully checks that the packet can be correctly received after a SWD interrupt, and that there is enough data to initialize the descrambler and detect an AX.25 flag before the frame.

The tests with two antennas and with a charge at the transmitter's output were performed again. It is clear that the SWD success rate is function of the BER.

Even though the test setup was probably the cause of the issue, and the SWD system was ruled out, this is still concerning as such a BER in space would make any communication with the satellite impossible. More investigation was thus absolutely necessary.

Test with IC-9100

A frequency mismatch was another possible cause for this issue. To be able to tune more precisely the carrier frequency of the transmitter, the IC-910h was replaced by the IC-9100. The IC-9100 minimal output power is 2 W instead of 5 W for the IC-910h, already reducing the level by around 4 dB and possibly avoiding some of the unwanted RF effects. These tests could also eliminate the hypothesis of a defective IC-910h radio. The modified test setup is shown in Figure 2.32.

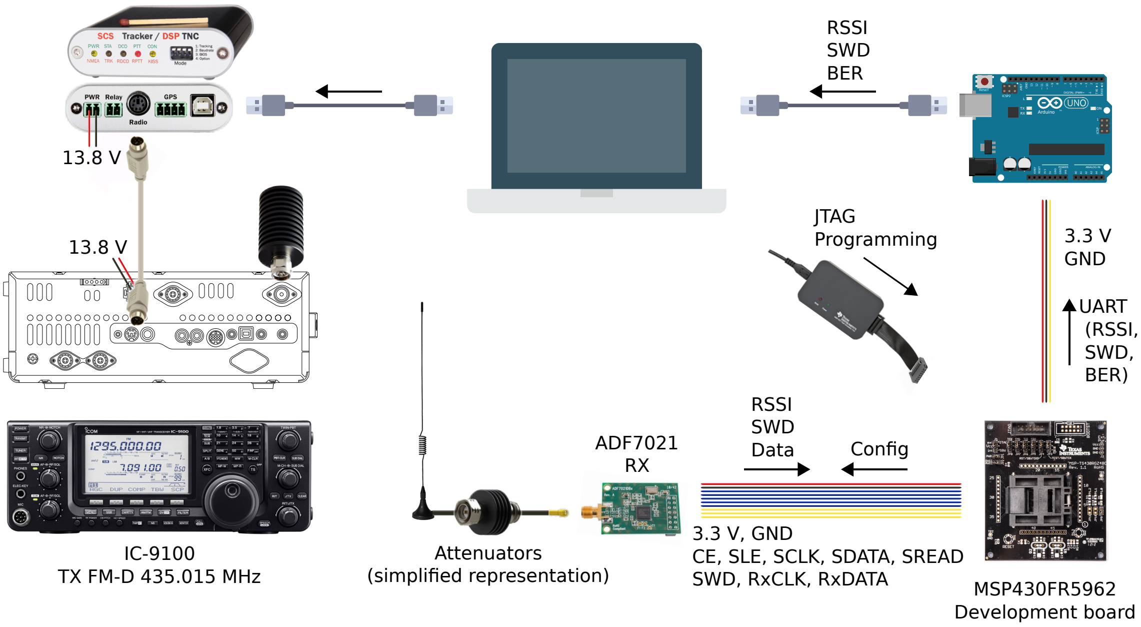


Figure 2.32 – Bit Error Rate test setup (signals from Arduino to ADF7021 must be shifted from 5V to 3.3V). Image sources: [13, 18, 26–30, 34]

Because the RSSI reading, as well as the power output tuning are not that precise, a large number of packets were sent and processed, with very small changes to the power output. The results in Table 2.6 are not sufficiently accurate to determine the limit power level, but it can be assumed that the system works as it should at power levels close to the -82.6 dBm target. Some tests were also performed with another nearly identical antenna. For unknown reasons, it showed a lower RSSI, so the output power of the IC-9100 was increased to stay around -83 dBm again. In practice, the measured RSSI was varying between -81 dBm and -85 dBm during the test but they were not discriminated. The results were even closer to the goal.

Setup	Radio	Measured RSSI	Bit error rate		Packet error rate	
Charge TX, antenna RX	IC-910h	-64 dBm	$0/(10*336)$	0	0/10	0%
Idem with -20 dB	IC-910h	-71 dBm	$0/(10*336)$	0	0/10	0%
Idem with -20 dB and tuning the radio RF output power knob	IC-9100	-81 dBm	$0/(13*336)$	0	0/13	0%
		-82 dBm	$1/(15*336)$	$2.0*10^{-4}$	1/15	7%
		-83 dBm	$3/(16*336)$	$5.6*10^{-4}$	3/16	18%
		-84 dBm	$15/(21*336)$	$2.1*10^{-3}$	10/21	48%
		-85 dBm	$10/(17*336)$	$1.8*10^{-3}$	6/17	35%
		-86 dBm	$14/(14*336)$	$3.0*10^{-3}$	9/14	64%
		-87 dBm	$12/(8*336)$	$4.5*10^{-3}$	7/8	88%
		-88 dBm	$2/(1*336)$	$6.0*10^{-3}$		
Idem with other antenna	IC-9100	$\sim$ -83 dBm	$2/(100*336)$	$6.0*10^{-5}$	2/100	2%

Table 2.6 – Bit Error Rate on 336 bits long packets

2.4.4 Conclusion

In the end, the best Bit Error Rate that has been experimentally achieved for the aimed -82.6 dBm power level is $\sim 6*10^{-5}$, which is not the predicted 10^{-5} [32] but still in the same order of magnitude. The test conditions of [31], with few errors[3] at -105 dBm could unfortunately not be reproduced. However, the results obtained here are judged good enough to guarantee that the satellite will eventually be able to receive telecommands without issue, even though it may sometimes need re-transmission. Moreover, the tests were performed in more difficult conditions than in space regarding noise and multi-path. Overall, it was proven that the Sync Word Detect system in itself works, as long as the reception works as well. Because evaluating the telecommand transmission quality was not part of the statement of this thesis, there was no further investigation. The initial issue is considered fixed and validated.

As a reminder, the proposed solution is to set the AX.25 RX ADF7021 configuration register 11 to `10111010 11010111 00001111 00111011` and register 12 to `11111111 10001100`. In short, these value will allow AX.25 uplink frames to be detected by the ADF7021, which will then set the SWD pin to high. It does not require any other hardware or software change. It should be noted that the configuration registers bytes are given in reverse order in OUFTI-1's OBC and D-STAR source codes, as well as in the Excel register value computation spreadsheet. For a concrete example and the complete ADF7021 AX.25 configuration, see in file `adf7021.c` of OUFTI-2's D-STAR repeater source code.

[3] But no Bit Error Rate measurement

Chapter 3

D-STAR relay

3.1 D-STAR

D-STAR[1] is an amateur radio protocol for digital voice and data transmission [35, 36]. It was introduced in 2001 by the Japanese Amateur Radio League (JARL). It is usually used in the 145 MHz, 430 MHz and 1.2 GHz ITU amateur radio bands. The main D-STAR compatible radio manufacturer is Icom. D-STAR is encoded in GMSK (Gaussian Minimum Shift Keying) and occupies a bandwidth of 6.25 kHz per channel. D-STAR offers 2 types of packets: Digital Voice (DV) and Digital Data (DD), which is less common and not supported by the repeater. The structure of a DV D-STAR frame is shown in Figure 3.1. The transmission alternates between 72 bits long voice slots and 24 bits long data slots. The 1^{st}, 22^{nd}, 43^{rd}, ... data slots are replaced with a synchronization flag containing a fixed sequence. This mode works at 4800 bits/s with 3600 bits/s of audio encoded with the AMBE codec, 1200 of which are dedicated to Forward Error Correction (FEC), and the remaining 1200 bits/s available for data. The AMBE (Advanced Multiband Excitation) audio encoding protocol is developed by Digital Voice Systems (DVSI) and is proprietary. The contents of the audio stream is not important for this work as it does not need to be decoded but only repeated. A D-STAR frame can have any arbitrary length after the header but the End Of Frame must be placed in place of a synchronization time slot[2].

D-STAR provides an addressable repeater feature, typically working on multiple frequency bands, and frame routing between different repeater stations and even through gateways connected to the internet. OUFTI-1 and OUFTI-2 repeaters are not traditional D-STAR repeaters though, in the sense that they repeat every incoming frame, without taking the repeater callsigns of the header into account.

The various modes of operations available with D-STAR are presented in Figure 3.2. The satellite provides the additional options of Figure 3.3. A D-STAR repeater was installed in the University in 2008 [37].

[1] Digital Smart Technologies for Amateur Radio

[2] This was not taken into account in OUFTI-1's D-STAR relay

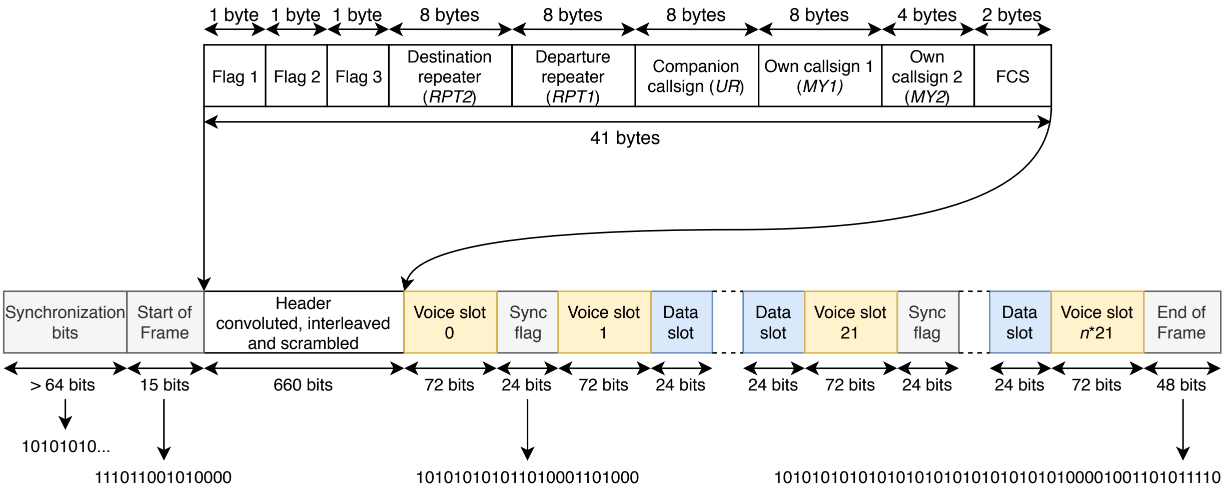
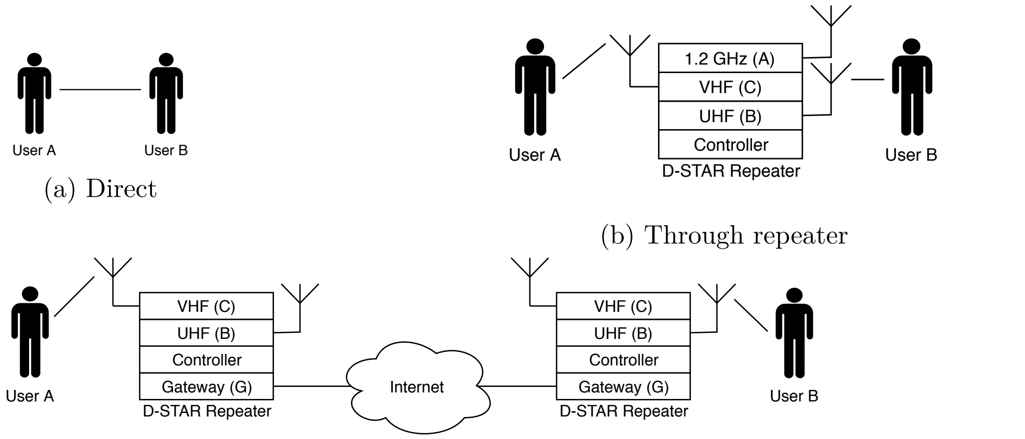


Figure 3.1 – D-STAR Digital Voice frame structure [38]

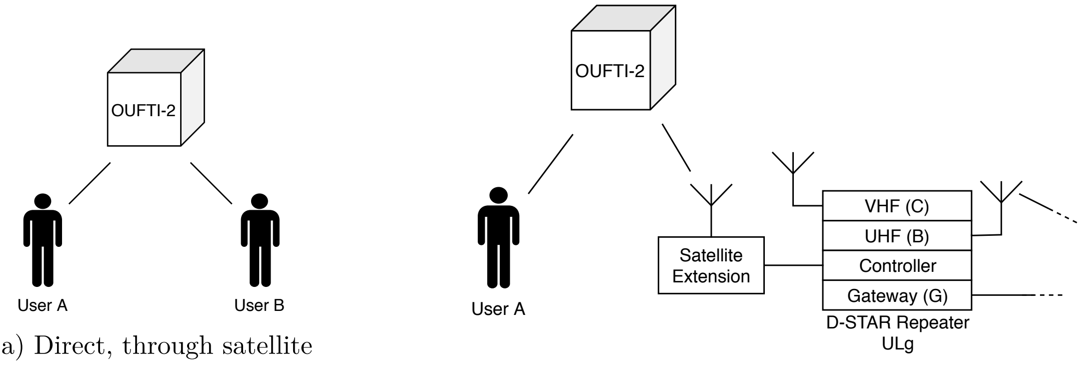


(a) Direct

(b) Through repeater

(c) Through different repeaters connected to the internet

Figure 3.2 – D-STAR normal communication modes [35]



(a) Direct, through satellite

(b) Through satellite and repeater

Figure 3.3 – D-STAR communication modes with OUFTI-2 [37]

3.1.1 D-STAR header

The header contains some control flags, the address callsigns and a Frame Check Sequence (FCS). The FCS is a Cyclic Redundancy Check computed same 16 bits CCITT polynomial as AX.25:

$$x^{16} + x^{12} + x^5 + 1$$

The header is first convoluted, doubling in length, going from 41 Bytes = 328 bits to 660 bits. The difference filled with padding `'0'`s. Odd bits are generated with the polynomial $1 + x + x^2$ and even bits with $1 + x^2$, as illustrated in Figure 3.4.

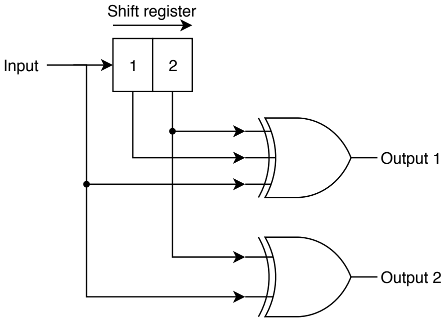


Figure 3.4 – Convolution diagram

It is then scrambled to eliminate the DC component of the signal, using a 7 bits long shift register and the following polynomial:

$$x^7 + x^4 + 1$$

Unlike AX.25 G3RUH scrambling, detailed in 2.1.3, the generated pseudo-random sequence is independent of the data. The shift register is initialized to `0xFF`. Descrambling is identical to scrambling. This process is showed in Figure 3.5.

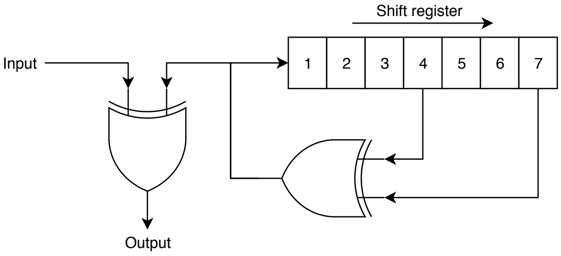


Figure 3.5 – Scrambler/descrambler diagram

Is is finally interleaved with a depth of 24, to spread error bursts.

The decoding process is straightforward for de-interleaving and descrambling. The deconvolution is performed using the Viterbi algorithm [39], which use a state representation of the convolution to return the most likely initial bit sequence, and correct some errors of the header introduced during the transmission.

3.2 Specifications

The initial required specifications for the new repeater as well as the additional modifications are:

- Repeater mode: as in OUFTI-1, the microcontroller must have a mode in which it receives D-STAR frames at 435.045 MHz and repeats them at 145.950 MHz. The header must be decoded and re-encoded to correct the errors, and the *MY2* callsign (see Figure 3.1) must be set to ”`OUFT`” to show that the packet was processed and repeated by the satellite. The repeater operations are detailed in 3.5.

- Parrot mode: in this new mode, the microcontroller must transmit a frame stored in memory when told by the OBC. The goal is to be able to test the transmission independently from the reception. The frame may be transmitted a given amount of times with a given delay. It was not initially planned but the parrot frames may be pre-recorded on ground and there is also a re-writable slot, which can be overwritten in flight. The parrot operations are detailed in 3.6.

- Communications with the OBC must now be done on an SPI bus instead of the USART channel of OUFTI-1. The bus is shared with other subsystems. The OBC is the master and sends commands which must contain some kind of checksum and be acknowledged by the D-STAR microcontroller. Unlike OUFTI-1, the repeater does not request the OBC to enable or disable the Power Amplifier, but the enable signal goes through the multiplexer, as explained in 1.2. The SPI communications with the OBC are detailed in 3.8.

- The repeater must log the callsigns of the processed frames and the number of frames. When ordered so by the OBC, it must respond with those data. The final implementation logs much more data than that for each packet, and all of this is transmitted to the OBC and then to the ground in chunks of length determined by the OBC. The logs contents and SPI command are detailed in 3.8.3.

- As for OUFTI-1, the configuration registers of the ADF7021s must be alterable from the ground. Unlike OUFTI-1 though, the configuration can be sent as a whole and not necessarily one register at a time. Also, the configuration is stored and applied at each ADF7021 startup. In OUFTI-1, it was actually impossible to modify the TX configuration, as it was only applied once and the TX ADF7021 is reset at each transmission. The ADF7021 configuration register modification SPI command is detailed in 3.8.2.

- Like OUFTI-1, the D-STAR microcontroller outputs debug information to an UART serial bus. However, it does not receive anything on that bus, and can be controlled only through SPI. Thanks to the Arduino debug tool presented in 3.9, which relays the UART and emulates the OBC through SPI, receiving debug commands in UART was not necessary anymore.

The previous code also contained a Doppler correction feature. Basically, the frequency setting of the ADF7021s was adjusted to compensate for Doppler effect due to the velocity of the nanosatellite. The exact values were sent from the ground and 2 different zones could be intermittently listened to [6, 40]. When it was first discussed when developing OUFTI-1, the amateur radio transceivers that had both D-STAR and fine carrier frequency tuning for Doppler correction were rare and very expensive. These are now more accessible (like the IC-9100) and other solutions were developed since then. For example, a solution to transmit D-STAR on an usual transceiver was presented in [41]. Therefore, this feature was dropped in OUFTI-2, as it was not judged useful enough anymore and introduced a lot of complexity.

The plan was to take the code from OUFTI-1 [42] and modify it to the new specifications, but the new code was rewritten from scratch for several reasons. First, the code was written by multiple people, which added new features over many years [6, 31, 37, 40, 43, 44]. This made it a bit messy and it definitely needed to be cleaned. The obsolete Doppler correction mechanism was deeply integrated. Plus, the new features would not really incorporate well in the current code without a lot of re-organization. Some parts of the repeater code were inaccurate or missed some special cases checks[3]. The microcontroller model was also changed, as described in 3.3. Finally, another IDE (Code Composer Studio) was adopted, which made impossible the use of the existing Assembly language code for D-STAR header encoding and decoding. Overall, in order to have a clear, robust and reliable code, everything was rethought with the new specifications and constraints in mind. Only the repeater finite state machine was kept to be used as a starting point during the development, although it was then subject to substantial modifications. The ADF7021 configurations registers contents were copied as-is.

3.3 Microcontroller selection

MSP430 is a family of 16 bits low-power, inexpensive microcontrollers from *Texas Instruments*. The MSP430F1612 [45] had been selected for both the OBC and the D-STAR repeater in OUFTI-1. There are 2 reasons for which it has to be changed. First, an FRAM non-volatile memory is more appropriate than a flash one. Second, the new microcontroller must have enough memory space to store the parrot messages.

Ferroelectric Random Access Memory (FRAM) is a kind of non-volatile memory. Its main advantages for our use case compared to flash memory are a lower power consumption and a better resistance against radiation. It is typically more expensive and less dense but it is not really an issue [46]. Some MSP430s exist with that kind of memory, with the prefix MSP430FR, so it was decided to pick one of these microcontrollers.

[3] Examples: Race conditions with the interrupt when changing mode, End Of Frame pattern that should only be detected in a synchronization time slot instead of anywhere in the frame [36], absence of timeout resulting in a possible lock

If the parrot frames are stored without any kind of compression, the required memory is 4800 bits/s, or 600 Bytes/s. The MSP430F1612 had 55 kB of non-volatile memory, but there is no information on how much was really used by the program itself. It is just stated that the available memory size is ”not restrictive” [43]. MSP430FRs are available with 64, 128 or 256 kB of FRAM memory. Even in the worst case scenario where the code was consuming all the 55 kB, it would still let space for respectively 15 s, more than 2 minutes and more than 5 minutes of D-STAR. The 128 kB option seems safe and more than enough for the purpose of the parrot mode.

The microcontroller was finally selected with the following criteria: 128 kB of FRAM memory, at least the same amount of RAM than the previous one and at least the same amount of GPIO than the amount that were used in OUFTI-1. No other particular characteristics are necessary, all the candidates having the required SPI and UART peripherals. This leads to the MSP430FR5962, shown in Figure 3.6. Its relevant characteristics compared with the MSP430F1612 are presented in Table 3.1. The package that will be used on OUFTI-2 will be RGZ[4]. It is the only option available on *Farnell* with only pins on the sides which makes soldering easier compared to ball grid arrays

Microcontroller	MSP430F1612 [45]	MSP430FR5962 [47]
Non-volatile memory type	Flash	FRAM
Non-volatile memory size	55 kB	128 kB
RAM size	5 kB	8 kB
Available GPIO pins	48 (only 32 connected)	68 (40 for RGZ package)

Table 3.1 – OUFTI-1’s and OUFTI-2’s D-STAR microcontrollers comparative table

The MSP430FR5964 was also a viable option, having the same characteristics as the MSP430FR5962 but with 256 kB of FRAM. For the reasons explained above, it was considered overkill for this application.

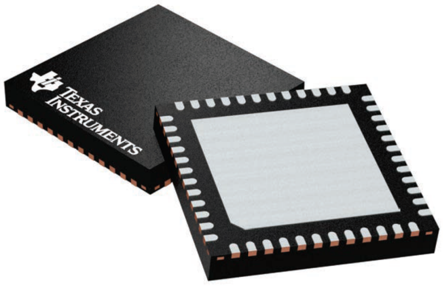


Figure 3.6 – Microcontroller RGZ package picture [47]

[4] The RGZ package from *Texas Instruments* is a 48 pins VQFN (Very thin Quad Flat No lead) package

3.4 General software architecture

After having setup all the ports, interrupts, peripherals, timers and clock settings, the main code enters an infinite loop. This loop continuously executes the 3 following tasks:

- Checks the `mode_change_flag()` from the SPI file. If it is set, identifies the mode (REPEATER, PARROT, CAPTURE or OFF), extracts the parameters if the mode is PARROT, sets the `buffer_ptr` to the right memory, sets the correct starting `state` and enables the appropriate interrupts (SWD or TX for PARROT).
- Because printing debug messages to UART must be avoided in the interrupt routine for timing reasons, the interrupt code sets a debug flag instead. The main loop checks for these debug flags and prints to UART the appropriate messages. This is only for debug purposes and not necessary for the relay operations.
- The computations that need to be performed on the D-STAR header are too heavy to be handled live during the repetition. When a header is received, the `state` is modified in the interrupt to notify the main code that a header is ready to be processed. During these operations, the received bytes are buffered, which is explained in more details in 3.5. Once this is complete, the main code changes the `state` again to begin the transmission.

The Port 1 interrupt routine contains 3 interrupt sources for the repeating operations: SWD and RX_CLK coming from the RX ADF7021 and TX_CLK coming from the TX ADF7021. These interrupts are enabled or disabled when the `state` changes. More details are presented in the sections below.

Another code running in parallel is the SPI reception. The SPI module is enabled or disabled through interrupts on `Chip Select` edges. When active, the SPI interrupt routine fetches the received byte, processes it, and sets the following byte to send. It interacts directly with the ADF7021 and logs when receiving SPI commands that are executed immediately. When receiving a mode change command, it is stored in the `next_mode` buffer until the main loop applies the mode change. During that time, the `mode_change_flag()` cancels all operations in the SWD, RX and TX interrupts. That way, there is no race condition between the main loop and the interrupts code to modify the `state` or enable/disable the interrupts. In all possible situations, only either the main code or the interrupt routine can change the `state`, never both.

The ADF7021 file is responsible for sending the correct configurations to the ADF7021s transceivers. The enable functions have hard-coded delays of 10's of milliseconds during the transmission of the configuration registers. This is not an issue as they are only called by the main code.

The Logs functions can be called from anywhere in the code. The UART functions as well with the exception of timing-critical interrupts, as they are blocking.

The code architecture is summarized in Figure 3.7.

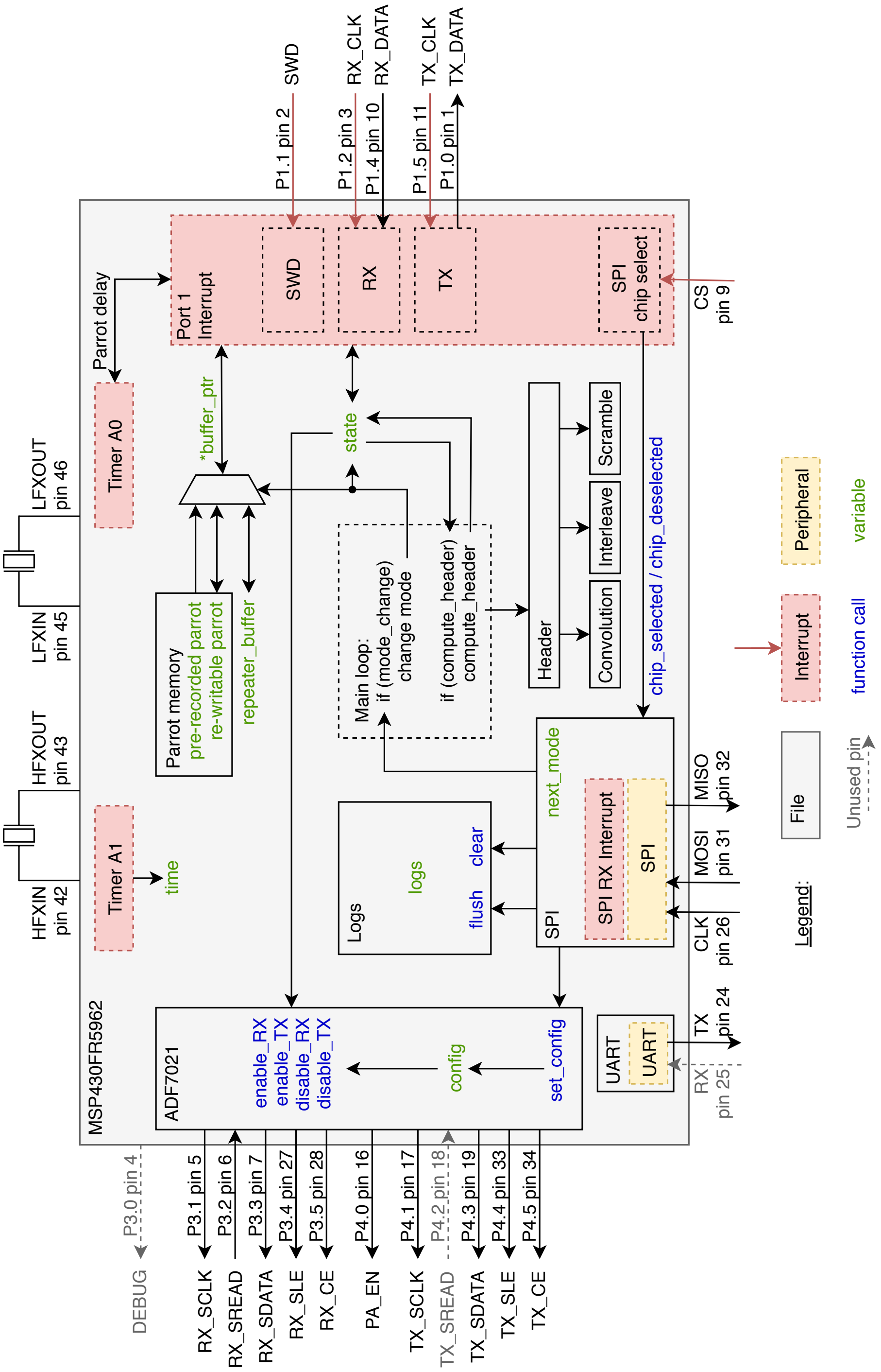
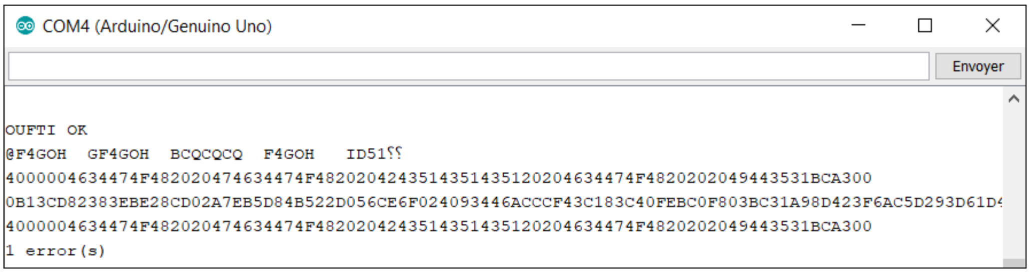


Figure 3.7 – D-STAR microcontroller code block diagram with source files, pinout, main variables and main function calls

3.5 Repeater mode

In repeater mode, OUFTI-2 listens continuously for incoming D-STAR frames. When it receives one, it decodes its header, sets the *MY2* callsign part to `OUFT` to show that the frame went through the satellite. Then, it updates the FCS, encodes the modified header and starts transmitting the frame. While the header is being processed, the received data is buffered. The transmission is therefore delayed by the time needed to complete all the header operations.

The decoding and encoding algorithms were entirely implemented in Assembly language in OUFTI-1, due to timing constraints [42]. It was quickly discovered while developing the new relay that this was not necessary anymore. The new algorithms are, in most part, taken from a library developed by the french amateur radio operator F4GOH [48]. In Figure 3.8, they are tested on an artificial D-STAR header given in the library.



```
OUFTI OK
@F4GOH  GF4GOH  BCQCQCQ  F4GOH   ID51ʕʕ
4000004634474F48202047463447 4F4820204243514351435120204634474F4820202049443531BCA300
0B13CD82383EBE28CD02A7EB5D84B522D056CE6F024093446ACCCF43C183C40FEBC0F803BC31A98D423F6AC5D293D61D
4000004634474F48202047463447 4F4820204243514351435120204634474F4820202049443531BCA300
1 error(s)
```

Figure 3.8 – Arduino debug tool serial output (see 3.9) showing D-STAR header encoding and decoding with a voluntarily flipped bit, corrected by the Viterbi algorithm

The operations of the microcontroller in REPEATER mode are shown in Figure 3.9. In summary, a Finite State Machine goes through the following states:

INACTIVE The SWD interrupt is enabled, waiting for an incoming packet

WAIT_SOF A potential packet has been detected, received data is pushed into a shift register, which is compared to the 15 bits Start of Frame (SOF) pattern indicating the beginning of the header

RX_HEADER The next received 660 bits are stored in a header variable

RX_BUFFER_COMPUTE_HEADER Following data is stored into the final buffer, leaving enough space before for the header once it is computed. The main code checks for this state and begins the header modifications process. The next state will be set by the main code when it is done

RX_BUFFER_TX Once the new header has been placed in the final buffer, alongside the synchronization bits and the Start of Frame, the transmission interrupt is enabled, in parallel with the reception. When the reception ends, either because the End of Frame has been detected or because the length has reached the timeout limit, the reception interrupt is disabled but the states stays unchanged. When all the data has been transmitted, the state goes back to INACTIVE

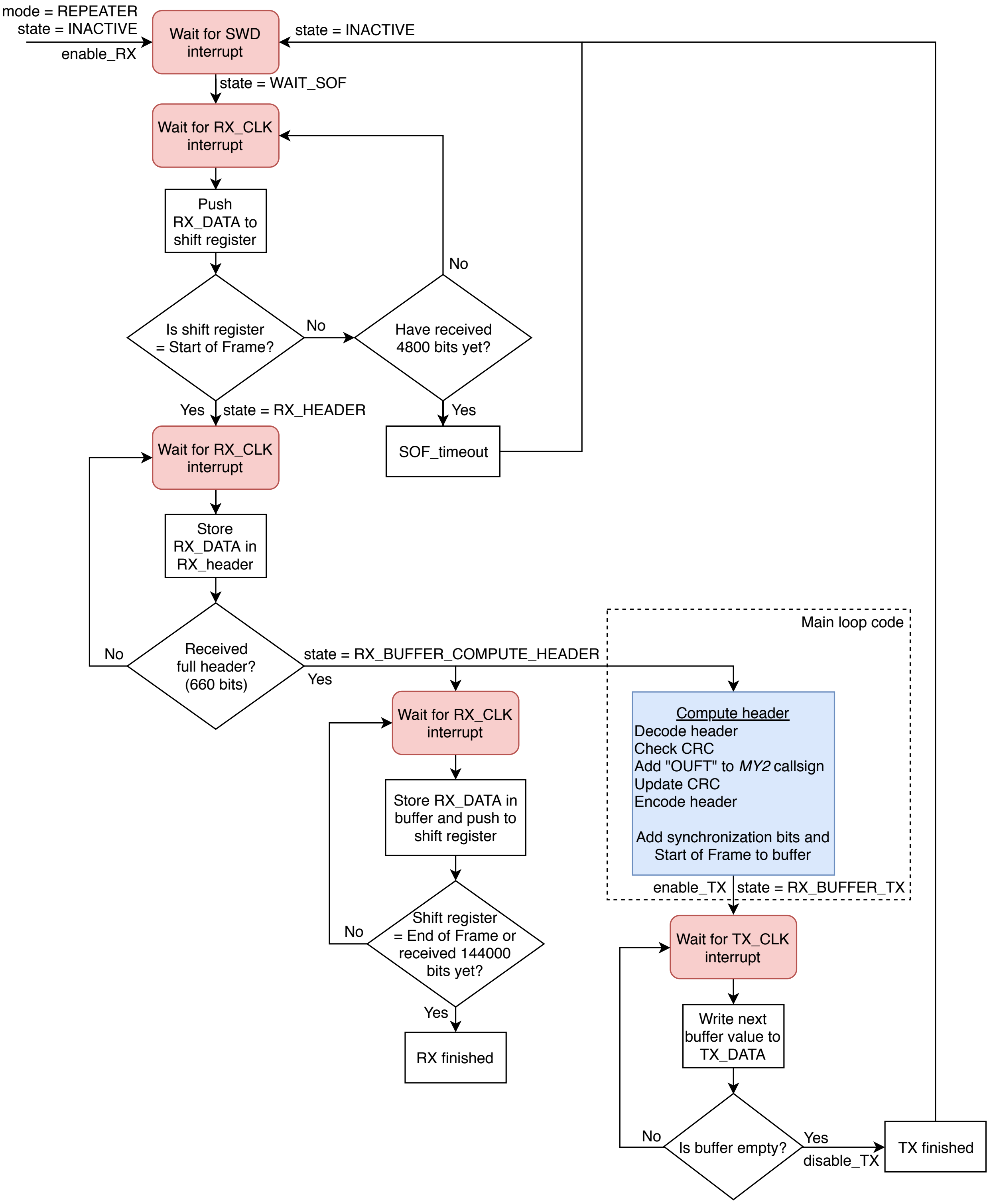

mode = REPEATER
state = INACTIVE
enable_RX
Wait for SWD interrupt
state = INACTIVE
state = WAIT_SOF
Wait for RX_CLK interrupt
Push RX_DATA to shift register
Is shift register = Start of Frame?
No
Have received 4800 bits yet?
No
Yes
state = RX_HEADER
Yes
SOF_timeout
Wait for RX_CLK interrupt
Store RX_DATA in RX_header
Received full header? (660 bits)
No
state = RX_BUFFER_COMPUTE_HEADER
Yes
Main loop code
Wait for RX_CLK interrupt
Store RX_DATA in buffer and push to shift register
Shift register = End of Frame or received 144000 bits yet?
No
Yes
RX finished
Compute header
Decode header
Check CRC
Add "OUFT" to MY2 callsign
Update CRC
Encode header
Add synchronization bits and Start of Frame to buffer
enable_TX
state = RX_BUFFER_TX
Wait for TX_CLK interrupt
Write next buffer value to TX_DATA
Is buffer empty?
No
Yes
disable_TX
TX finished

Figure 3.9 – Repeater mode flow chart (does not illustrate header CRC error nor mode change cases)

The buffer holds the received data while the header is being processed. Then, the transmission begins at the beginning of the buffer, in a FIFO manner (First In, First Out). Transmission and reception happen at the same rate, so the amount of data remains constant once transmission has started. The buffer is stored in a cyclic structure in memory. The length of the buffer must be large enough for the received data during the header processing. The minimum size was obtained by outputting the amount of received bits at the end of these operations, as shown in Table 3.2.

Clock configuration	Received bits	Minimum buffer size
DCO at 8 MHz, divider 8 ⇒ 1 MHz	∼ 17000 bits	∼ 2200 Bytes
DCO at 8 MHz, divider 1 ⇒ 8 MHz	∼ 3200 bits	∼ 400 Bytes

Table 3.2 – Minimum size of buffer to store received data during header processing

The final buffer is 2048 Bytes long, out of the 5 kB total available RAM. It provides more than enough margin for the final clock frequency of 7.3728 MHz. It could have been reduced a lot but the memory space was not needed for something else so it was kept this way.

It should be noted that the repeater code does not access directly its buffer, but a pointer to it. This allows for this pointer to be changed to the parrot buffers, which are in other parts of memory, and transmit or record using essentially the same code.

The modified header is directly stored into the buffer, before the received data. Synchronization bits and the Start Of Frame pattern are also added. That way, the buffer contains all the required parts of the beginning of the D-STAR frame. Only one transmission state is required, which can directly send its contents. There are 640 synchronization bits, which is more than the minimum 64 of the D-STAR specification. Repeating the synchronization pattern is very common to ensure the receiver's Phased Lock Loop (PLL) is well synchronized before the first data. It was measured that the traditional commercial radios used during the development typically transmit around 600 bits, which was also observed in [31].

Unlike OUFTI-1, this repeater also includes a timeout feature which cuts any repeated frame after a given amount of time, set to 30 seconds. It also timeouts if the Start Of Frame has not been detected 4800 bits after the Sync Word Detect.

Commercial D-STAR radios check for the regular synchronization time slots. After a certain amount of missing ones, the frame is cut. It is useful in the case of an abruptly terminated transmission. The D-STAR relay does not implement this verification for the following reasons. First, it already contains a timeout feature, preventing any deadlock and cutting any frame after 30 seconds. This made the synchronization check not that useful anymore, and because any unnecessary complexity should be avoided, it was not implemented. In the worst case, less that 30 seconds of garbage will be repeated by the relay, which will occupy the bandwidth. The radios receiving on that frequency will ignore that noise as they will detect the absence of valid synchronization patterns. The same thing will happen if a valid frame is timed-out after 30 seconds.

3.6 Parrot mode

The parrot mode allows to transmit pre-recorded frames, to test the transmission independently from the reception. To make good use of the available FRAM memory while keeping the system simple, there are 2 kinds of parrot memory slots. First, there is a single re-writable slot able to store a 60 seconds frame. The modification of this message can be done with the CAPTURE command, after which the next received frame will be recorded. This mode was the simplest to implement, as the microcontroller FRAM can easily be written to, and thus the data does not have to leave the memory or be generated by a computer. However, to allow for more flexibility and to avoid breaking the parrot mode in case of a problem with the CAPTURE command once in flight, some frames can also be permanently programmed into the microcontroller by putting them directly in the source code. To do that, the frames can be extracted from the re-writable memory to a computer with the DUMP command. See 3.8.4 for more details. This second approach allows for up to 255 frames, with a total maximum length of more than 2 minutes. The PARROT command used to play back a parrot message takes the ID of the message as parameter. The special ID value `0xFF` is used to play the re-writable frame. The amount of times the frame will be repeated and the delay between each each transmission can also be specified in parameters. See Table 3.3 in 3.8 for more details.

Figure 3.10 shows how the microcontroller handles the play back PARROT mode. Before it is enabled, the `buffer_ptr` variable is set to the correct buffer address. It can be the re-writable parrot variable in FRAM or one of the constant pre-recorded parrot frame.

Only one state is used, the TX interrupt is simply disabled during the delays and the buffer index reset before the next transmission. Other than that, the repeater code can be reused.

Figure 3.11 shows how frames are recorded. The repeater code is reused again but without the transmission part. The RX_BUFFER_TX state is replaced by RX_BUFFER_PARROT, which is equivalent for the reception. The capture is complete when the reception of the packet is finished and the header has been computed. It is the end of the reception that changes the mode to OFF, but it is not an issue if it happens before the header processing, as the following actions will always be performed in the main code anyway.

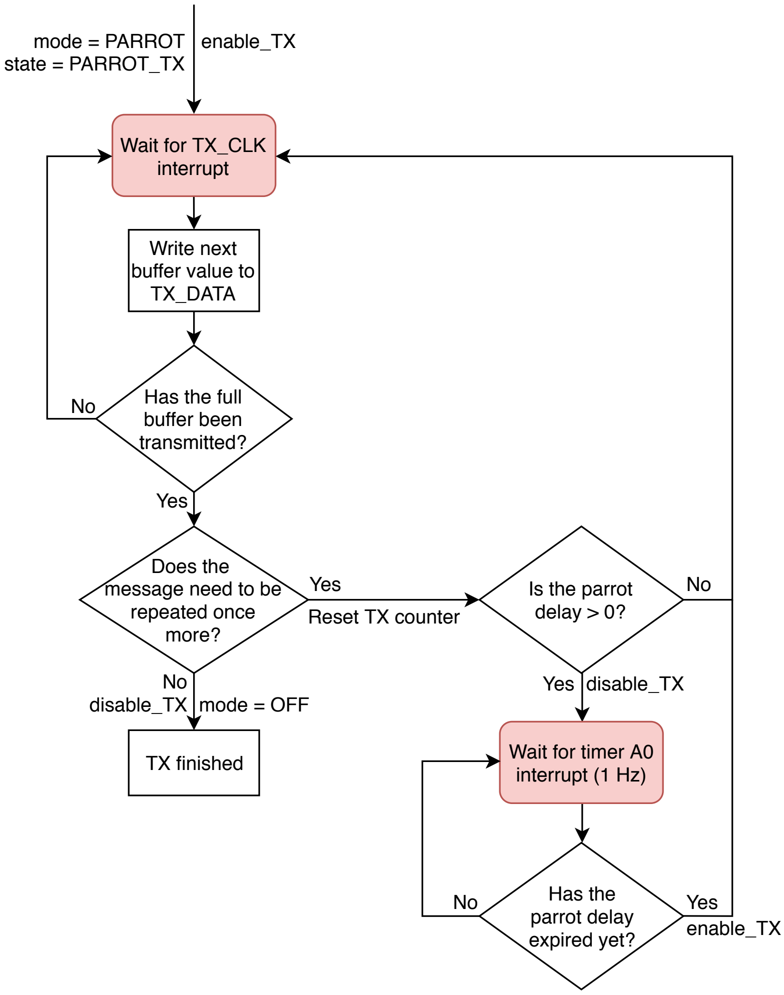

mode = PARROT
state = PARROT_TX
enable_TX
Wait for TX_CLK interrupt
Write next buffer value to TX_DATA
Has the full buffer been transmitted?
No
Yes
Does the message need to be repeated once more?
Yes
Reset TX counter
Is the parrot delay > 0?
No
No
disable_TX
mode = OFF
TX finished
Yes
disable_TX
Wait for timer A0 interrupt (1 Hz)
Has the parrot delay expired yet?
No
Yes
enable_TX

Figure 3.10 – Parrot mode flow chart (does not illustrate mode change case)

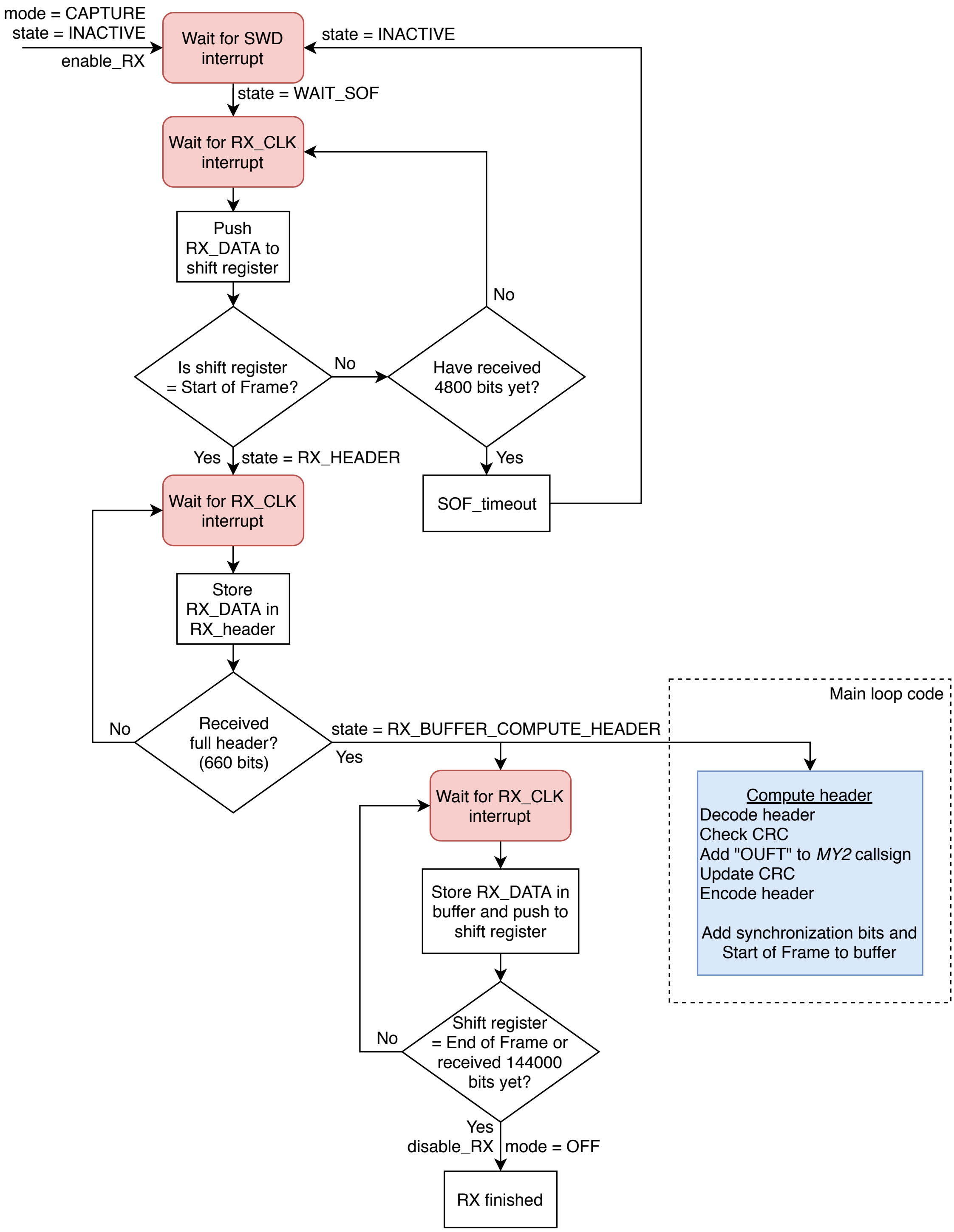
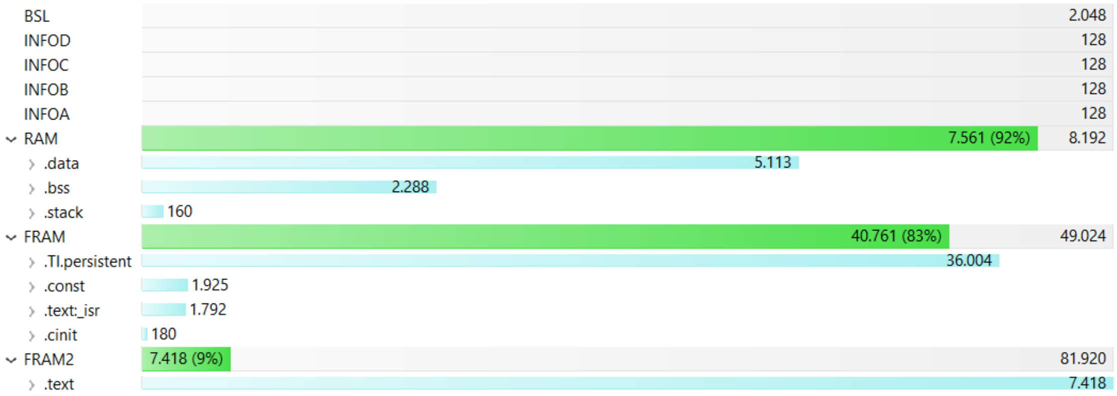

mode = CAPTURE
state = INACTIVE
enable_RX
Wait for SWD interrupt
state = INACTIVE
state = WAIT_SOF
Wait for RX_CLK interrupt
Push RX_DATA to shift register
No
Is shift register = Start of Frame?
No
Have received 4800 bits yet?
Yes
state = RX_HEADER
Yes
Wait for RX_CLK interrupt
SOF_timeout
Store RX_DATA in RX_header
Main loop code
No
Received full header? (660 bits)
state = RX_BUFFER_COMPUTE_HEADER
Yes
Wait for RX_CLK interrupt
Compute header
Decode header
Check CRC
Add "OUFT" to MY2 callsign
Update CRC
Encode header
Add synchronization bits and Start of Frame to buffer
Store RX_DATA in buffer and push to shift register
No
Shift register = End of Frame or received 144000 bits yet?
Yes
disable_RX
mode = OFF
RX finished

Figure 3.11 – Capture mode flow chart (does not illustrate header CRC error nor mode change cases)

3.7 Memory usage



BSL		2.048
INFOD		128
INFOC		128
INFOB		128
INFOA		128
RAM	7.561 (92%)	8.192
.data	5.113	
.bss	2.288	
.stack	160	
FRAM	40.761 (83%)	49.024
.TI.persistent	36.004	
.const	1.925	
.text:_isr	1.792	
.cinit	180	
FRAM2	7.418 (9%)	81.920
.text	7.418	

Figure 3.12 – Memory allocation, without any pre-recorded parrot frame, from Code Composer Studio

Figure 3.12 shows the organization of the microcontroller memory using the default linker file. The RAM mainly stores the logs of 4856 bytes in the `.data` segment, and the repeater buffer of 2048 bytes in `.bss` (uninitialized data). The FRAM is divided into 2 partitions: `FRAM` and `FRAM2`. The former stores the re-writable parrot message slot of 36000 bytes in `.TI.persistent`, while the latter is mostly empty and only contains the 7418 bytes long main code in the `.text` segment (without the interrupts). It should be noted that the memory allocation can be tweaked by modifying the linker file, which will certainly not be necessary in this case.

When adding pre-recorded messages in the source code, they will be automatically placed by the linker in either of these 2 partitions. So the maximum length of a single parrot message is $\sim$ 74500 bytes, or 124 seconds. The maximum total length of the pre-recorded messages is $\sim$ 82700 bytes, or 137 seconds.

The parrot pre-recorded messages could be compressed, for example by only storing the voice part of the frame and not the data, which is currently empty. This would save 25% of the memory space but would make the code more complicated. It is judged unnecessary in this case, the available memory being largely enough for this application.

The MSP430FR5962 FRAM has a Memory Protection Unit (MPU) feature, which prevents any write access to parts of memory that should not be written to, only allowing it in `.TI.persistent` in this case. This protects the code from being unintentionally overwritten, but also data to be executed in case of an error. It also has bit error detection and correction codes built-in [49].

3.8 SPI communications with the On-Board Computer

The D-STAR microcontroller is inactive by default. It needs to be explicitly turned on by an AX.25 telecommand, which will be received and processed by the OBC. After powering on the microcontroller, the OBC will send the appropriate command on an SPI bus, shared between the OBC and D-STAR microcontroller, but also other subsystems of the satellite (IMU, RAD, beacon, some ADCs, ...).

The OBC is the master of the SPI bus, and communications with the D-STAR microcontroller are performed in `"mode 3"`: $CPOL = 1$, $CPHA = 1$ (clock high by default, signal modified on clock falling edge and sampled on clock rising edge) [50]. Bytes are sent Most Significant Bit (MSB) first. The maximum speed, tested experimentally to be stable with some margin, is 125 kHz. There must be at least 20 μs of delay between the moment the `Chip Select` pin is pulled down and the beginning of the transmission, between the end and the rise of `Chip Select`, as well as between each transmission.

The available commands are listed in Table 3.3. Most commands (except the `LOG` request) do not expect an answer from the D-STAR microcontroller. After having pulled down the `Chip Select` pin, the OBC sends the command, its parameters if needed and the Cyclic Redundancy Check (CRC). As an acknowledgement, every transmitted byte will be repeated. The command will be executed when the OBC releases the `Chip Select` pin back to high. All the commands are represented by a single ASCII byte. All the parameters are single byte unsigned numbers unless stated otherwise. Figures 3.13 and 3.14 show some time diagrams of typical commands.

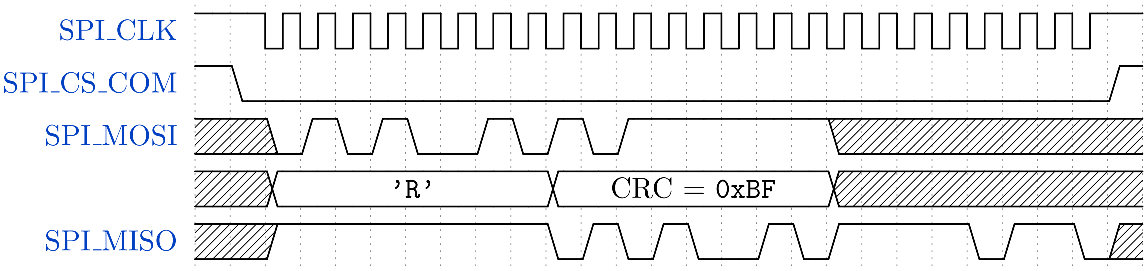


Figure 3.13 – SPI time diagram of the REPEATER command

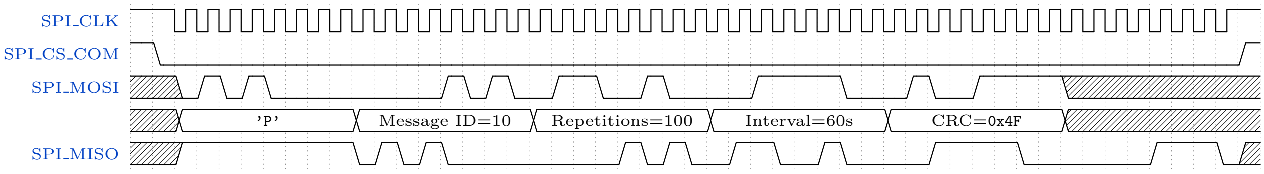


Figure 3.14 – SPI time diagram of an example PARROT command

Command	Byte	CRC	Parameters	Description
OFF	`'F'`	`0xA4`	*None*	Disables all D-STAR operations (default mode)
REPEATER	`'R'`	`0xBF`	*None*	Repeater mode
PARROT	`'P'`	Yes[5]		Transmits a parrot message
			Message ID:	Pre-recorded frame ID, or `0xFF` for re-writable frame
			Repetitions:	Number of additional transmissions[6]
			Interval:	Time in seconds between each transmission[7]
CAPTURE	`'C'`	`0x32`	*None*	Records next frame into re-writable parrot slot
SET_CONFIG	`'S'`	Yes		Changes ADF7021 configuration registers
			Nb config TX:	Number of TX configuration registers
			Config TX:	Variable length, see 3.8.2
			Config RX	Variable length
CLEAR_LOGS	`'Z'`	`0xB1`	*None*	Resets and unlocks the logs
LOGS	`'L'`	No		Flushes the logs on the SPI bus, see 3.8.3
			Offset:	Index of the first returned byte, on 16 bits
(DUMP)	`'D'`	No	*None*	Dumps the re-writable parrot frame Should only be used on ground, see 3.8.4

Table 3.3 – SPI commands summary

When entering REPEATER mode, it will stay active until another command is received. On the other hand, the microcontroller will automatically go back in OFF mode after having completed a PARROT or CAPTURE command. The 4 mode changing commands (OFF, REPEATER, PARROT and CAPTURE) will override the previous mode when received, even if the operation was not yet completed. Changing mode will immediately interrupt any ongoing reception or transmission. The SET_CONFIG command will be effective starting from the next ADF7021 power on.

If the first byte received on the SPI bus after `Chip Select` has been pulled down is different from the ASCII characters presented above, the microcontroller will go into OFF mode and the `FLAG_SPI_UNKNOWN_MODE` error flag will be raised in the logs.
If a command requiring a certain amount of parameters is issued and not enough parameters are sent before `Chip Select` is set back to high, the mode will also be set to OFF and the `FLAG_SPI_TOO_SHORT` error flag will be raised.
If the message ID received in a PARROT command is greater or equal to the number of pre-recorded messages, and is not equal to `0xFF`, no message will be transmitted, the mode will be set to OFF and the `FLAG_SPI_INVALID_PARROT_ID` error flag will be raised.

[5] Depends on the parameters, see 3.8.1

[6] The message will be transmitted a total of *Repetitions* + 1 times

[7] More precisely, between the end of a transmission and the beginning of the next one

3.8.1 Acknowledgement and CRC

Except when transmitting LOGS or the re-writable parrot frame with the DUMP command, each byte sent by the OBC will be immediately transmitted again by the microcontroller. The OBC must verify them to ensure the command has correctly been received. If the OBC detects an error, it informs the ground but takes no action. If the microcontroller has received the error, it will detect it thanks to the CRC and safely go back in OFF mode. To retrieve the last byte of the command, which is the CRC, the OBC must keep on transmitting the SPI clock for an additional byte, as shown in Figure 3.13. If it does not, the command will not be recognized. This acknowledgement does not guarantee that the command was valid and that it was executed, but only that it has been correctly received and will be processed. Information about invalid SPI commands are available in the logs (see 3.8.3).

The commands, with the exception of LOGS and DUMP, must end with a Cyclic Redundancy Check (CRC) byte. This ensures that the repeater will never start transmitting when not allowed and that the ADF7021s configuration will not be broken, even in the case of a bit error in an SPI command. This CRC is computed against the command byte and all the parameters, following the CRC-8-CCITT polynomial. Here is an implementation of this algorithm:

```
char spi_crc(char *array, unsigned int len) {
  char crc = 0xFF;
  for (int n=0; n<len; n++) {
    crc ^= array[n];
    for (int m=0; m<8; m++) {
      if (crc & 1)
        crc = (crc >> 1) ^ 0xE0;
      else
        crc = crc >> 1;
    }
  }
  return crc ^ 0xFF;
}
```

In the case the CRC is incorrect, the command will be discarded, the microcontroller will go back in OFF mode and the `FLAG_SPI_BAD_CRC` error flag will be raised. The optimal CRC polynomial choice is a complex research topic, but in the case of an 8 bits CRC for messages of only a few bytes with an expected error rate of almost 0, it does not really matter.

3.8.2 ADF7021 configuration

The SET_CONFIG command allows for a complete custom re-configuration of both the TX and RX ADF7021s. The amount of registers for the TX ADF7021 is sent as the first parameter, followed by the TX registers contents, then the RX registers and finally the CRC. The amount of RX registers is inferred from the total size of the command. Each register is simply the raw 4 bytes that will be transmitted to the ADF7021, in the same order. The maximum amount of registers per ADF7021 is 15. The new configuration is applied independently at the next corresponding ADF7021 reset. The TX configuration will thus be updated before the next transmission and the RX configuration at the next mode change. A time diagram with the different fields is shown in Figure 3.15.

If the amount of registers of one of the configuration is 0, the default configuration for that ADF7021 will be used. Thus, it is possible to modify only one of the two configurations, or even to reset both configurations by sending no register at all. The configurations are not kept in persistent memory and will be back to default at power down.

A special value for the number of TX configurations parameter is `0xFF`. If used, the default TX configuration will be set, but the Power Amplifier will be disabled. This may be useful for RX-only operations if so desired. As if the parameter was set to 0, all the following registers will be stored for the RX configuration.

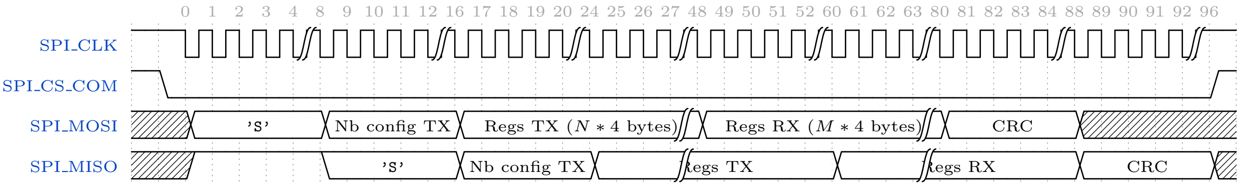


Figure 3.15 – SPI time diagram of the SET_CONFIG command

3.8.3 Logs

The LOGS command will send back any part or all the stored logs on the SPI bus. When a LOGS command is issued, all further logging is blocked until the next CLEAR_LOGS call. This ensures the data is not modified and stays consistent during a multiple parts transfer. The logs are not stored in persistent memory and erased when the microcontroller is powered down. Due to tight SPI timing constraints, it is strongly recommended to disable the repeater before pulling the logs.

After the `'L'` byte and the offset have been received and sent back (see 3.8.1), the microcontroller will respond with the amount of bytes of the remaining logs. Both the offset and the length are 2 bytes long and sent in big-endian (most significant byte first). As long as the SPI master continues to set the SPI clock and keeps `Chip Select` low, the logs will be sent until the end. However, the transmission can be terminated at any point. This allows for the logs to be retrieved in chunks of any length, depending on the OBC memory and telemetry constraints. Figure 3.16 shows the time diagram of a LOGS command.

The LOGS command does not end with a CRC, unlike all the other commands that the OBC has to handle. This CRC would add unnecessary complexity, as in the case of an error in the offset, the only thing that can happen is that wrong logs will be returned. It will immediately be detected on ground due to incoherent data and bad logs CRCs, leading to a re-transmission request for this chunk.

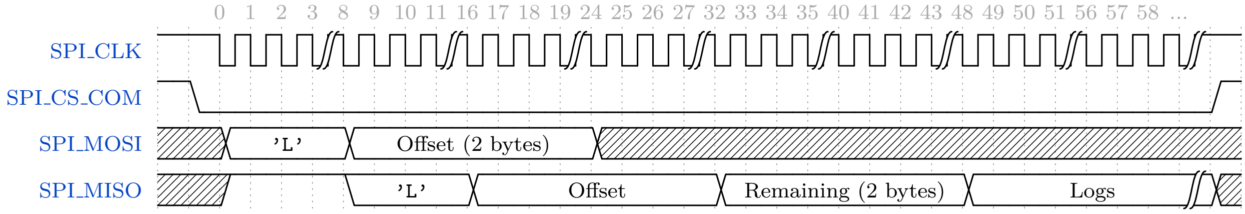


Figure 3.16 – SPI time diagram of the LOGS command

The first 40 bytes after the remaining length contain some global debugging information, detailed in Table 3.4. Then, chunks of 56 bytes contain information about each received D-STAR frame. Their format is detailed in Table 3.5. All integers are unsigned, multiple-byte long integers are sent in big-endian (most significant byte first). Every chunk (global logs and each packet logs) is ended with a single-byte CRC. This CRC is computed using the same algorithm as described in 3.8.1. The global logs CRC is computed during the first logs transmission, as they can be modified up to that point and the full CRC takes too much time to be computed at once. This means in practice that the global logs (first 40 bytes) must be retrieved in order the first time[8]. The packet logs CRC is computed when the packet operations are finished and its logs will not be modified anymore. The debug Arduino program implements an example of logs decoding. It is described in 3.9 and shown in Figure 3.18.

The maximum amount of logged packets is 85, which gives a maximum total logs length of 4800 bytes[9]. If more packets are received, the oldest ones will be overwritten and the `FLAG_LOGS_OVERFLOW` will be raised. During the transmission, the packet logs are internally re-ordered from oldest to latest.

Thanks to the fact that there are individual CRCs for the global logs and each packets logs, the logs are still exploitable even if only part of them have been received.

When doing a CLEAR_LOGS, part of the global logs are reset to 0 (see column "Reset" of Table 3.4) and all the packet logs are erased.

[8] The first LOGS command must have a 0 offset and bytes cannot be skipped until the global logs have been read at least once

[9] Global logs 40 bytes + 85 × packet logs 56 bytes = 4800 bytes

Name	Length	Reset	Description
FLAG_LOGS_OVERFLOW	1 bit	Yes	If received packets while logs were full, meaning that oldest logs were overwritten
FLAG_CAPTURE_USED	1 bit	Yes	If capture mode has been enabled
FLAG_PENDING_CONFIG_TX	1 bit		If a SET_CONFIG has not been applied to TX yet
FLAG_PENDING_CONFIG_RX	1 bit		If a SET_CONFIG has not been applied to RX yet
FLAG_SPI_BAD_CRC	1 bit	Yes	If the CRC of a SPI command was incorrect
FLAG_SPI_UNKNOWN_MODE	1 bit	Yes	If received invalid first byte in SPI
FLAG_SPI_TOO_SHORT	1 bit	Yes	If did not receive enough parameters in SPI
FLAG_SPI_INVALID_PARROT_ID	1 bit	Yes	If received invalid PARROT message ID
GBLLOG_TIME	4 bytes		Time of the first LOGS command, in seconds, starting from the power on of the microcontroller
GBLLOG_LAST_CLEAR	4 bytes		Time of the last CLEAR_LOGS command
GBLLOG_LAST_CONFIG	4 bytes		Time of the last SET_CONFIG command
GBLLOG_NB_CONFIG_TX	1 byte		Number of TX config registers, see 3.8.2
GBLLOG_NB_CONFIG_RX	1 byte		Number of RX config registers
GBLLOG_LAST_MODE_CHANGE	4 bytes		Time of last mode change
GBLLOG_LAST_MODE	1 byte		Last mode (or current) in ASCII format
GBLLOG_SPI_NB	1 byte	Yes	Number of SPI commands received
GBLLOG_PACKETS_NB	4 bytes	Yes	Number of received packets
GBLLOG_TOTAL_LEN	4 bytes	Yes	Sum of all the transmitted packets lengths
GBLLOG_STARTUP_RSSI	1 byte		Noise RSSI at last repeater mode change (absolute value, in dBm)
GBLLOG_PARROT_STARTUP_LEN	4 bytes		Re-writable parrot frame length since last reset
GBLLOG_PARROT_LEN	4 bytes		Current length of re-writable parrot frame
GBLLOG_PARROT_TX_NB	1 byte	Yes	Number of parrot transmissions
GBLLOG_CRC	1 byte		CRC of global logs, computed on the previous 39 bytes like in 3.8.1
Total:	40 bytes		

Table 3.4 – Global logs format
The reset column indicates if the log is set to 0 with the CLEAR_LOGS command

Name	Length	Description
FLAG_ABORTED	1 bit	If the mode was changed during the reception of this packet, interrupting this packet reception
FLAG_SOF_TIMEOUT	1 bit	If no header found after Sync Word Detect
FLAG_CRC_ERROR	1 bit	If the header's checksum was incorrect
FLAG_CAPTURE_PKT	1 bit	If CAPTURE mode enabled
FLAG_PACKET_TIMEOUT	1 bit	If this packet was too long and was thus truncated
	3 bits	Padding `'0'`s
PKTLOG_TIME	4 bytes	Time of Sync Word Detect reception
PKTLOG_MODE_TIME	4 bytes	Time of current mode change
PKTLOG_HEADER	36 bytes	Decoded callsigns, in D-STAR header order
PKTLOG_HEADER_CORRECTED_ERRORS	1 byte	Number of errors corrected by the Viterbi algorithm during deconvolution
PKTLOG_RSSI	1 byte	RSSI (absolute value, dBm)
PKTLOG_SYNC_LEN	4 bytes	Bits between Sync Word Detect and Start Of Frame
PKTLOG_FRAME_LEN	4 bytes	Total frame length in bits (4800 bits/s)
PKTLOG_CRC	1 byte	CRC of this packet logs, computed on the previous 55 bytes like in 3.8.1
Total:	56 bytes	

Table 3.5 – Packet logs format

3.8.4 Parrot dump command

The DUMP command allows to extract the frame stored into the re-writable parrot memory back to a computer. Its purpose is to provide an easy way to record D-STAR frames which can then easily be copied and stored in the pre-recorded messages memory. This command should only be used on ground.

To pre-record a message, the user connects the microcontroller SPI and UART pins to an Arduino running the debug tool (see 3.9). Then, they set the mode to CAPTURE, and transmit a message using a conventional D-STAR radio. This frame is stored in the re-writable parrot memory. The user then issues the DUMP command, which prints back to the Arduino serial terminal what has just been recorded. The debug tool conveniently prints the frame in a format which can directly be copied into the `parrot_memory` source file. The modified code can be compiled and re-programmed into the microcontroller.

After having received the `'D'` byte, the D-STAR microcontroller will send it back. The length of the dump in bytes in then sent on the next 2 bytes (big-endian: most significant byte first). As long as the SPI master continues to set the SPI clock and keeps `Chip Select` low, the data will be sent until the end. Unlike the LOGS command, it is not possible to specify an offset. If the DUMP command is issued again after an incomplete transfer, it will come back to the beginning. A result example is shown in Figure 3.19.

3.9 Arduino debug tool

To help during the development of the repeater and to test thoroughly its features independently, it is connected to an Arduino Uno board as illustrated in Figure 3.22 of 3.10. The Arduino serves 2 purposes: relay the UART debug output and emulate the On-Board Computer to send SPI commands and receive the logs. The Arduino is connected to a computer through USB and the communication can be done with a simple serial terminal, such as the one provided with the Arduino IDE. The voltage difference between the logic level of the Arduino (5V) and the microcontroller (3.3V) must be taken care of. Simple $\frac{2}{3}$ voltage dividers for the Arduino to microcontroller signals will work fine.

The UART debug output is a single wire 9600 bits/s[10] communication channel. It is one-way only, without parity bit, LSB first, with 8 data bits and 1 stop bit. It is used to send debugging data about the state and mode changes, the received and transmitted frames, the errors and basically all the information that ends up in the logs but in real time.

The Arduino script accepts simple commands to simulate SPI commands from the OBC. The parameters are separated by spaces, and can be decimal or hexadecimal with the prefix `0x`. The commands are terminated by a new line. They are listed in Table 3.6. The script will also compute and add the correct CRC and check the acknowledgement.

An output example is shown in Figure 3.17.

When issuing the LOGS command, the Arduino will test the offset feature, check basic rules on the total length, the remaining length field, the CRCs, and that the data corresponds between different requests. It will then display the information to the user in human-readable text, as can be seen in Figure 3.18.

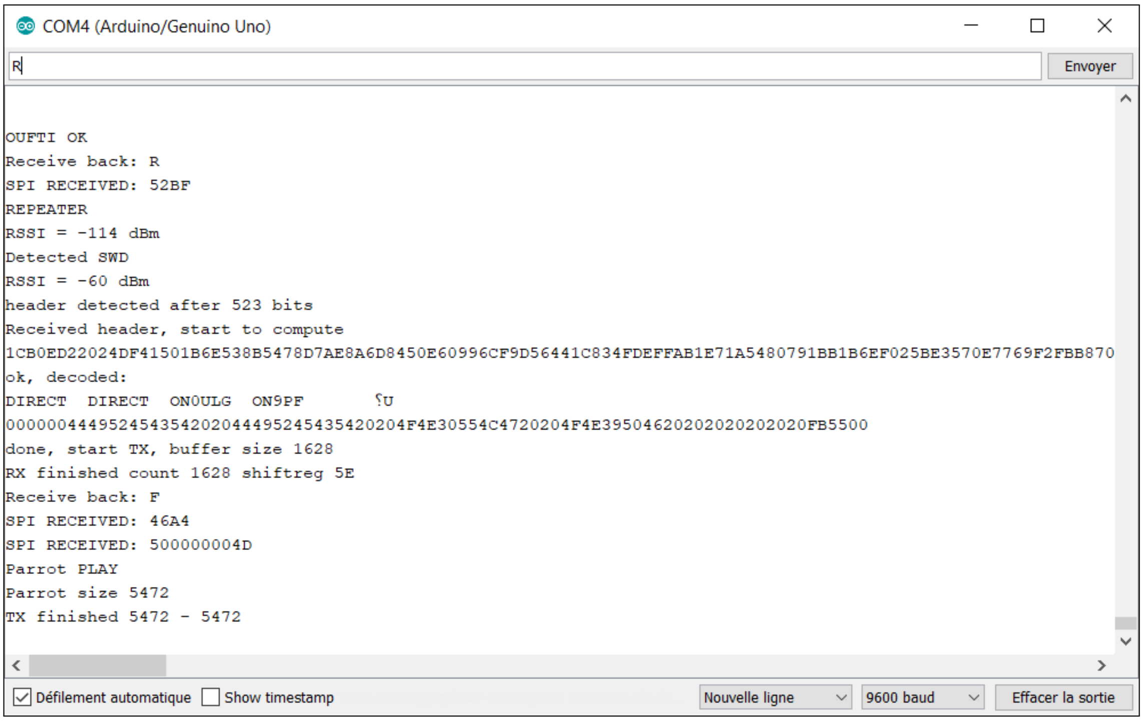
When using the DUMP command, it will display the contents of the parrot memory such that it can be copied directly into the source code, like in Figure 3.19.

The source code of this tool can be used in the future development of the ground station user interface, in order to encode the SPI commands, and decode and present the logs to the user.

[10] Generated from the 32768 Hz ACLK, with a modulation pattern to correct for the inexact division. When using the internal VLO as a source for ACLK, this correction does not seem to work and the UART peripheral on the Arduino must be configured at the corresponding integer frequency of 10923 Hz

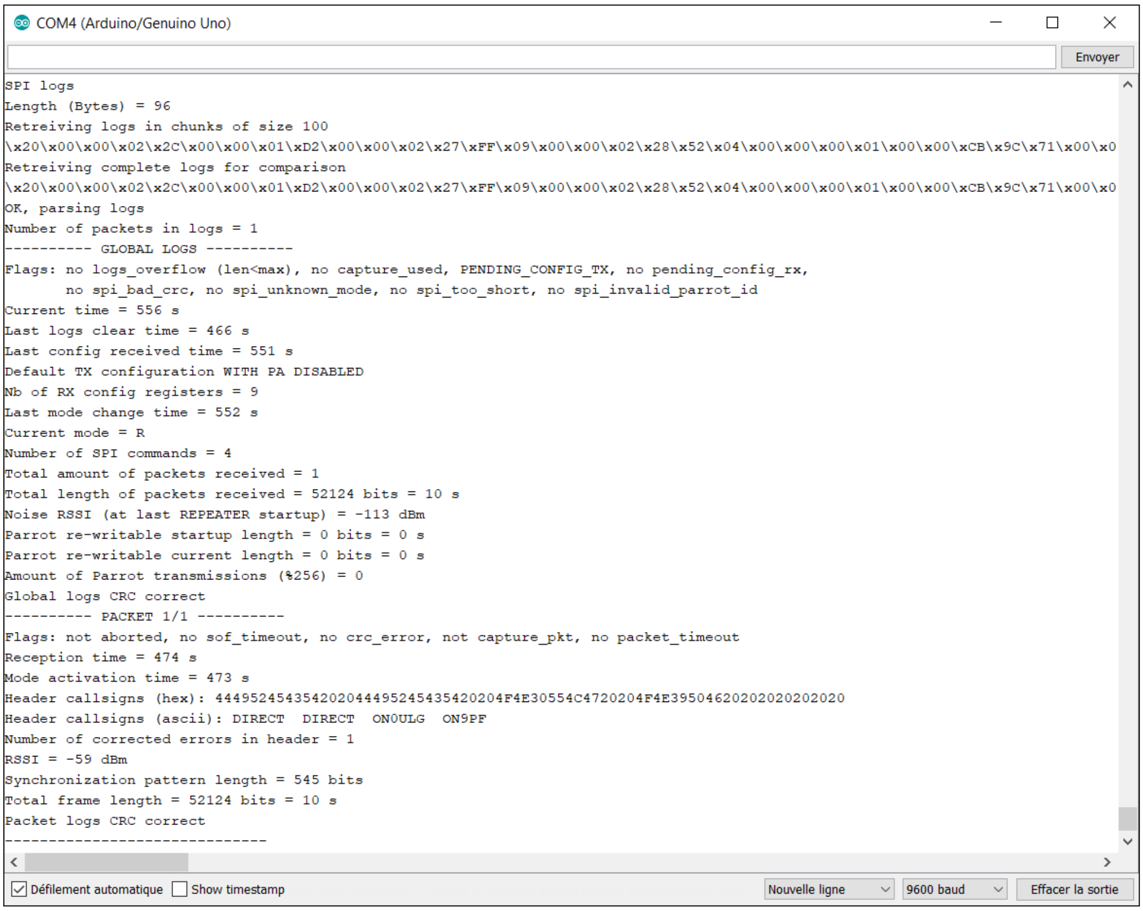
Commands	Corresponding SPI command
`"R"`	Sends the REPEATER command
`"C"`	Sends the CAPTURE command
`"F"`	Sends the OFF command
`"P -1 0 0"`	Sends the PARROT command to play the re-writable frame once
`"P 0 2 3"`	Idem, play the pre-recorded frame ID 0 with 2 repetitions and a 3 s delay (parameters can be replaced by any value, see Table 3.3 for more details)
`"Z"`	Sends the CLEAR_LOGS command
`"L"`	Retrieves and displays the logs (LOGS command)
`"D"`	Retrieves and displays the re-writable parrot memory (DUMP command)
`"S 0"`	Sends the SET_CONFIG command to reset to default configuration
`"S 1"`	Idem, default configuration with disabled Power Amplifier
`"S 2"`	Idem, manual configuration with same registers as default one
`"S 3"`	Idem, manual RX configuration with disable Power Amplifier
`"S 4"`	Idem, manual configuration with voluntarily broken RX registers
`"S 5"`	Idem, manual configuration with voluntarily broken TX registers
`"S 6"`	Idem, manual configuration with AX.25 RX registers, that can be verified with the detection of the SWD pattern when sending an AX.25 packet

Table 3.6 – Commands that can be sent to the Arduino debug tool



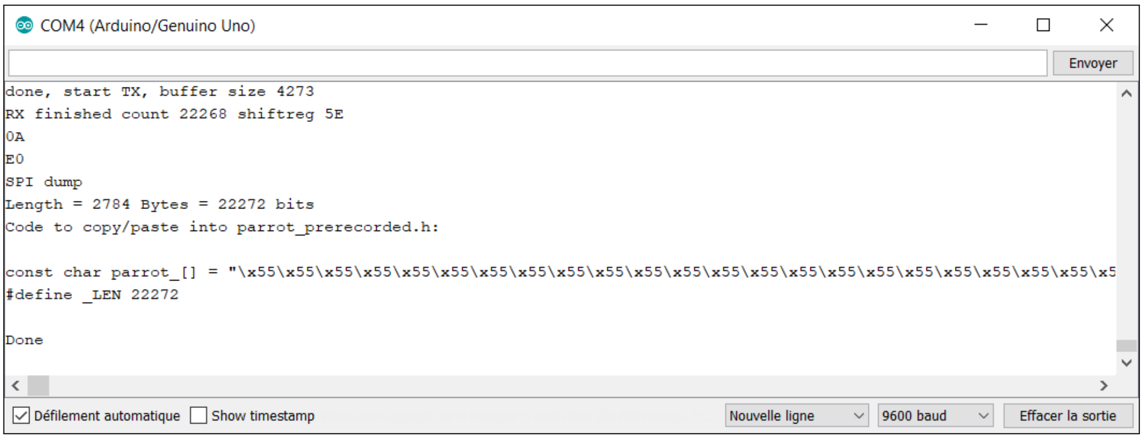
```
OUFTI OK
Receive back: R
SPI RECEIVED: 52BF
REPEATER
RSSI = -114 dBm
Detected SWD
RSSI = -60 dBm
header detected after 523 bits
Received header, start to compute
1CB0ED22024DF41501B6E538B5478D7AE8A6D8450E60996CF9D56441C834FDEFFAB1E71A5480791BB1B6EF025BE3570E7769F2FBB870
ok, decoded:
DIRECT  DIRECT  ON0ULG  ON9PF       ʕU
000000444952454354202044495245435420204F4E30554C4720204F4E3950462020202020202020FB5500
done, start TX, buffer size 1628
RX finished count 1628 shiftreg 5E
Receive back: F
SPI RECEIVED: 46A4
SPI RECEIVED: 500000004D
Parrot PLAY
Parrot size 5472
TX finished 5472 - 5472
```

Figure 3.17 – Arduino debug tool serial output when issuing SPI commands and receiving a D-STAR frame in repeater mode



```
SPI logs
Length (Bytes) = 96
Retreiving logs in chunks of size 100
\x20\x00\x00\x02\x2C\x00\x00\x01\xD2\x00\x00\x02\x27\xFF\x09\x00\x00\x02\x28\x52\x04\x00\x00\x00\x01\x00\x00\xCB\x9C\x71\x00\x0
Retreiving complete logs for comparison
\x20\x00\x00\x02\x2C\x00\x00\x01\xD2\x00\x00\x02\x27\xFF\x09\x00\x00\x02\x28\x52\x04\x00\x00\x00\x01\x00\x00\xCB\x9C\x71\x00\x0
OK, parsing logs
Number of packets in logs = 1
---------- GLOBAL LOGS ----------
Flags: no logs_overflow (len<max), no capture_used, PENDING_CONFIG_TX, no pending_config_rx,
       no spi_bad_crc, no spi_unknown_mode, no spi_too_short, no spi_invalid_parrot_id
Current time = 556 s
Last logs clear time = 466 s
Last config received time = 551 s
Default TX configuration WITH PA DISABLED
Nb of RX config registers = 9
Last mode change time = 552 s
Current mode = R
Number of SPI commands = 4
Total amount of packets received = 1
Total length of packets received = 52124 bits = 10 s
Noise RSSI (at last REPEATER startup) = -113 dBm
Parrot re-writable startup length = 0 bits = 0 s
Parrot re-writable current length = 0 bits = 0 s
Amount of Parrot transmissions (%256) = 0
Global logs CRC correct
---------- PACKET 1/1 ----------
Flags: not aborted, no sof_timeout, no crc_error, not capture_pkt, no packet_timeout
Reception time = 474 s
Mode activation time = 473 s
Header callsigns (hex): 444952454354202044495245435420204F4E30554C4720204F4E395046202020202020
Header callsigns (ascii): DIRECT  DIRECT  ON0ULG  ON9PF
Number of corrected errors in header = 1
RSSI = -59 dBm
Synchronization pattern length = 545 bits
Total frame length = 52124 bits = 10 s
Packet logs CRC correct
------------------------------
```

Figure 3.18 – Arduino debug tool serial output when retrieving the logs



```
done, start TX, buffer size 4273
RX finished count 22268 shiftreg 5E
0A
E0
SPI dump
Length = 2784 Bytes = 22272 bits
Code to copy/paste into parrot_prerecorded.h:

const char parrot_[] = "\x55\x55\x55\x55\x55\x55\x55\x55\x55\x55\x55\x55\x55\x55\x55\x55\x55\x55\x55\x55\x55\x55\x5
#define _LEN 22272

Done
```

Figure 3.19 – Arduino debug tool serial output when dumping the parrot frame

3.10 Tests

3.10.1 Compilation and programming

The Integrated Development Environment (IDE) used to develop the code is *Code Composer Studio* from Texas Instruments. This software allows to develop, compile, program and debug for TI's embedded processors, such as the MSP430. It is based on the Eclipse IDE and is completely free since version 7.

Programming is done through the JTAG interface, with the MSP-FET programmer and debugger. The development board includes a full JTAG connector. The final boards of the satellite are programmed using a *08FMS-1.0SP-TF* connector carrying the 4 programming wires, power, and the reset signal. Small and cheap adapter boards make the transition with the JTAG connector. During testing, the JTAG connection is not necessary, except when modifying the parrot pre-recorded messages.

3.10.2 Test setup

For OUFTI-1, a prototype PCB, shown in Figure 3.20, had been made to connect the D-STAR microcontroller to the ADF7021 development boards and to the OBC through a single connector. It also included the multiplexer to be able to test the interaction between the subsystems of the communications part of the satellite, excluding the beacon. However, this card was made for the previous MSP430F1612 microcontroller, now replaced by the MSP430FR5962, for the reasons developed in 3.3. The schematics of the card could not be found in the archives so it could not easily be modified to accommodate for this new microcontroller. Therefore, it was decided to skip the prototype PCB step, and develop using the MSP-TS430RGZ48C development board, connected with the ADF7021 development boards with wires. The setup is presented in Figure 3.21 and its diagram in Figure 3.22. The IC-9100 [16] is used to transmit at 435.045 MHz but cannot receive in D-STAR mode at a different frequency, so the IC-E2820 [51] is used for the reception at 145.950 MHz. It indeed looks messy due to all the wires connected to the ADF7021 development boards, but they do not need to be touched anymore once installed. Further testing can be performed directly on the final CubeSat PCB, addressed in 3.11.

When using the MSP-FET, the Arduino and the microcontroller grounds are connected and the MSP-FET provides 3.3V. When testing without the MSP-FET, the power came from the 3.3V output of the Arduino Uno board. The SPI signals coming from the Arduino to the microcontroller need to be lowered, using a voltage divider. The signals from the microcontroller can directly be fed into the Arduino as it will recognize 3.3V as a logic high without issue.

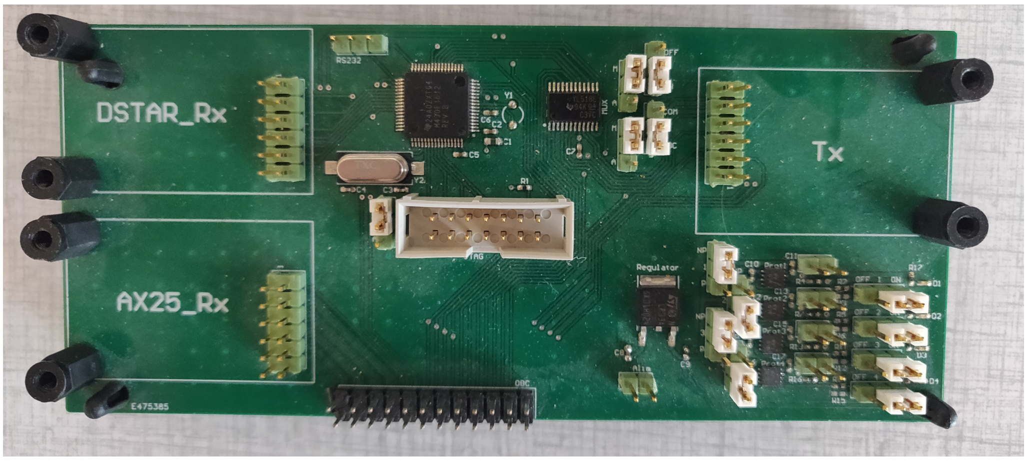


Figure 3.20 – PCB for the OUFTI-1 D-STAR microcontroller tests, with power regulators, slots for the ADF7021 evaluation boards, JTAG connector and pins to connect to the OBC

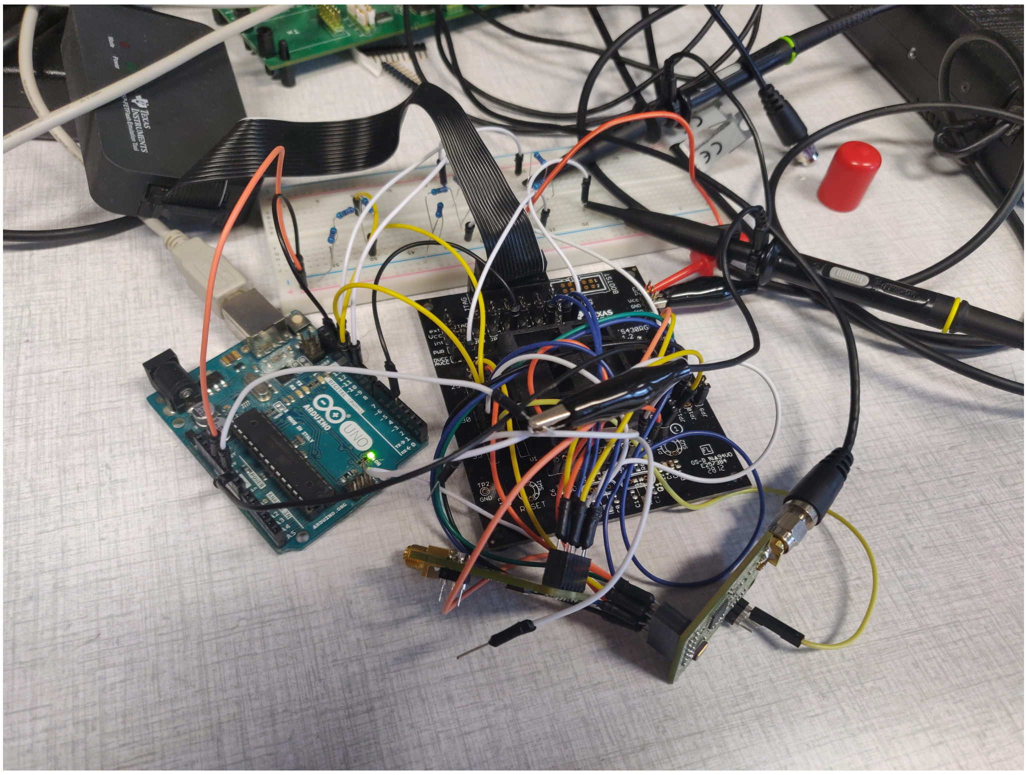


Figure 3.21 – Setup for the OUFTI-2 D-STAR microcontroller independent tests using the Arduino debug tool

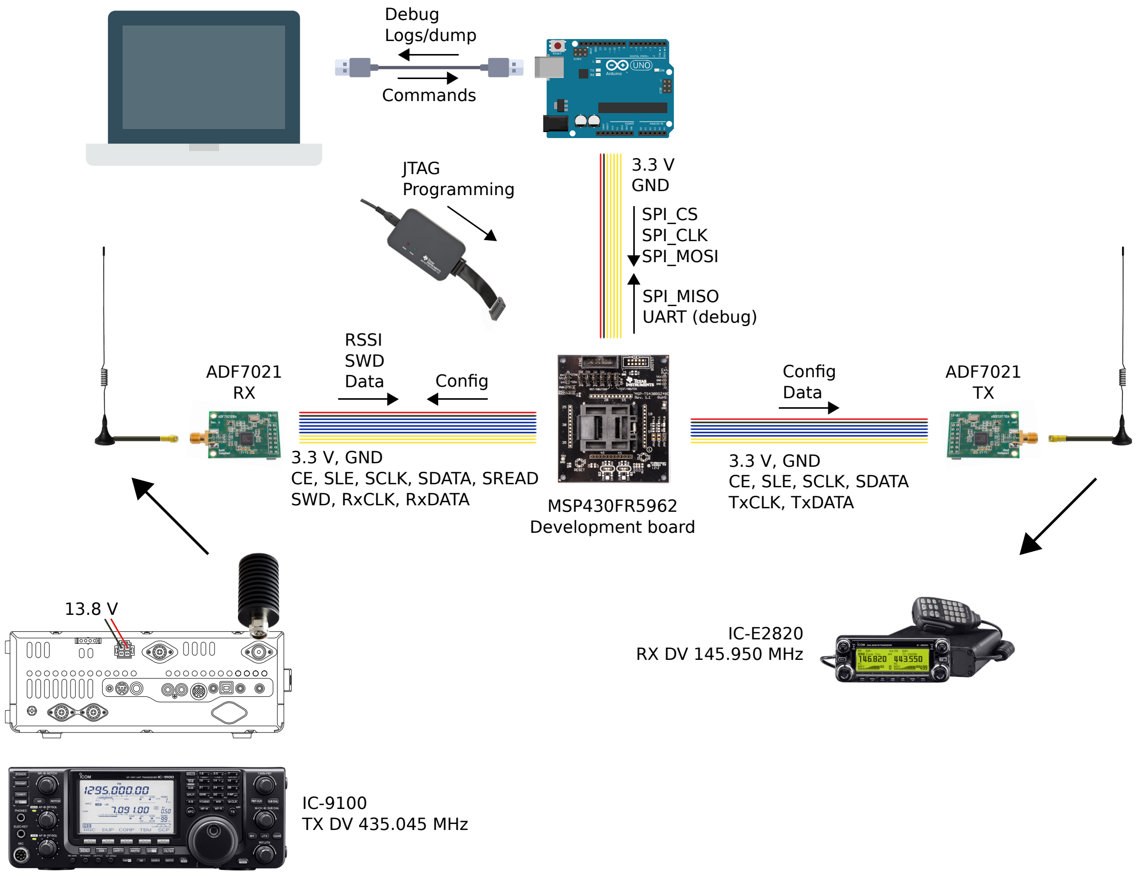


Figure 3.22 – D-STAR relay test setup (signals from Arduino to ADF7021 must be shifted from 5V to 3.3V)

Many tests were performed during and after the development of the microcontroller code, reproducing normal usage situations as well as edge cases. All the features have been deeply tested, including but not limited to:

- Every mode independently, and their timeouts
- Parrot messages with various lengths
- D-STAR header decoding and encoding, with manual bit error insertion to assess the Viterbi error correction, see 3.5 and Figure 3.8
- SPI commands, also with possible errors in the parameter amount or contents
- SPI CRC and acknowledgement
- SPI commands during RX or TX operations, or during parrot delay
- Every log and their different values
- Log overflow, cyclic log storing and packet logs re-ordering
- Log transmission, full or by chunks, also with invalid offset values, see 3.9
- The SET_CONFIG command in all the possible combinations, see Table 3.6
- Pending ADF7021 configuration and the moment it takes effect
- Different clock configurations, detailed in 3.10.3
- SPI communications with the actual OBC, still in development, detailed in 3.10.4

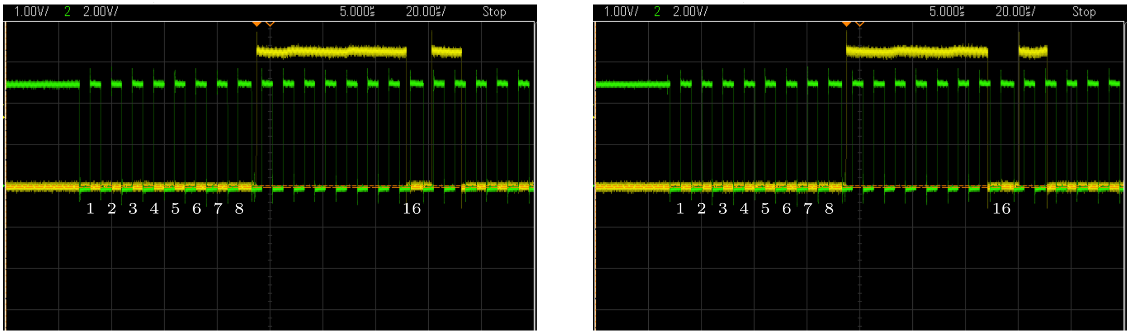
3.10.3 Clock tests

The majority of the development was done with the Master Clock MCLK driven by the internal DCO (Digitally Controlled Oscillator) of the microcontroller, configured at 8 MHz with a divider of 1. The Auxiliary Clock ACLK, used by the timers, was driven by the internal VLO (Very Low power frequency Oscillator) at 32768 Hz. The DCO was then set to 7 MHz when performing the majority of the tests, as the external oscillator specified in the schematics is at 7.3728 MHz. Finally, some equivalent oscillators were installed on the evaluation board to test the clock configuration. The pins of the external oscillators can be probed to check if they are correctly driven by the microcontroller. The clocks frequencies can also be observed on output ports to make sure they have the expected frequency.

The parts of the repeater with critical timing are the header processing and the SPI communications. The header processing is not really an issue though, as the buffer size is well over the limit. On the other hand, SPI communications face 2 major constraints:

- Repeating the byte that has just been received before the beginning of the next one, for SPI acknowledgements as explained in 3.8.1. This lets less than a full SPI bit period (at maximum 125 kHz) to trigger the interrupt, read the RX buffer, check weather it must be repeated or another byte must be sent (e.g., for the logs), and write to the TX buffer
- Computing in advance the following byte to send. Especially when receiving the LOGS command, as a series of additional operations have to be performed: add the current time to the logs, lock further logs updates, prepare the global CRC computation and update the internal state

To check that these constraints are always verified and ensure stable SPI communications, a debug output pin was used. It is set to high after the TX buffer has been written and back to low after all the operations in the interrupt have been completed. This verification can be performed in Figure 3.23. The debug signal rising edge must happen before the second byte starts to be sampled by the OBC, which happens on the 9th rising edge of the clock. The debug signal falling edge must happen before the second byte RX interrupt is triggered, which happens when the its bit is sampled, on the 16th rising edge of the clock. These 2 conditions are met in the worst case of the LOGS command. Other cases like when sending the logs remaining length or receiving other SPI commands, have been verified to be quicker with the same method.



(a) Internal 7 MHz DCO clock

(b) External 7.3728 MHz oscillator clock

Figure 3.23 – Observation of the time spend in the SPI interrupt. Yellow: debug signal high after TX buffer has been set until the end of SPI interrupt. Green: SPI clock.

3.10.4 SPI communications with the On-Board Computer

SPI communications were also tested with the On-Board Computer, being developed at the same time. A temporary ground software communicates in AX.25 with the OBC through a wired link, as the ADF7021 is not yet supported by the OBC. There are 2 telecommands dedicated to the D-STAR microcontroller. One of which sends an SPI

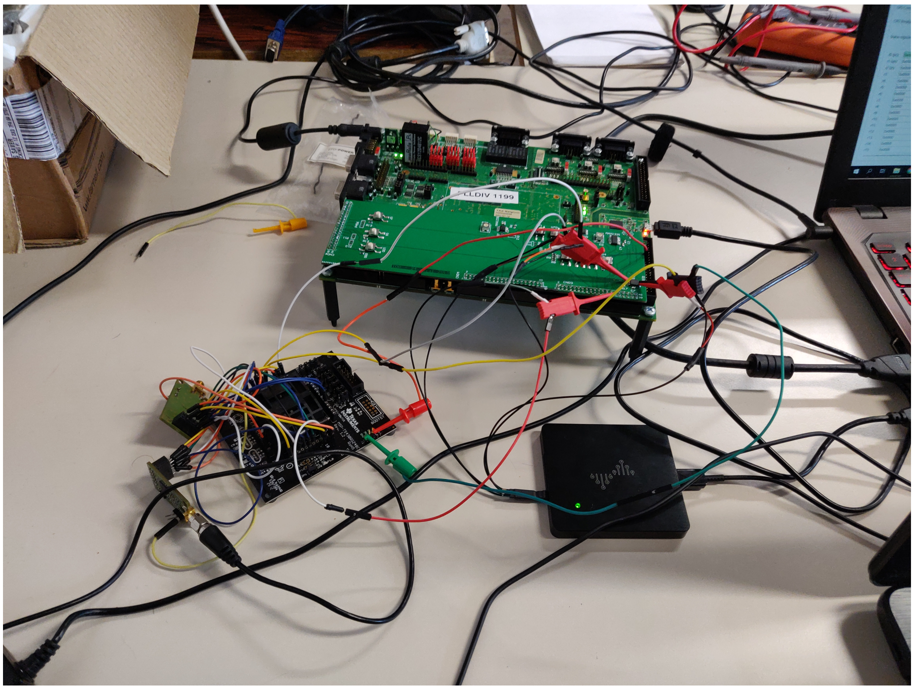


Figure 3.24 – Setup for SPI communications test with the OBC

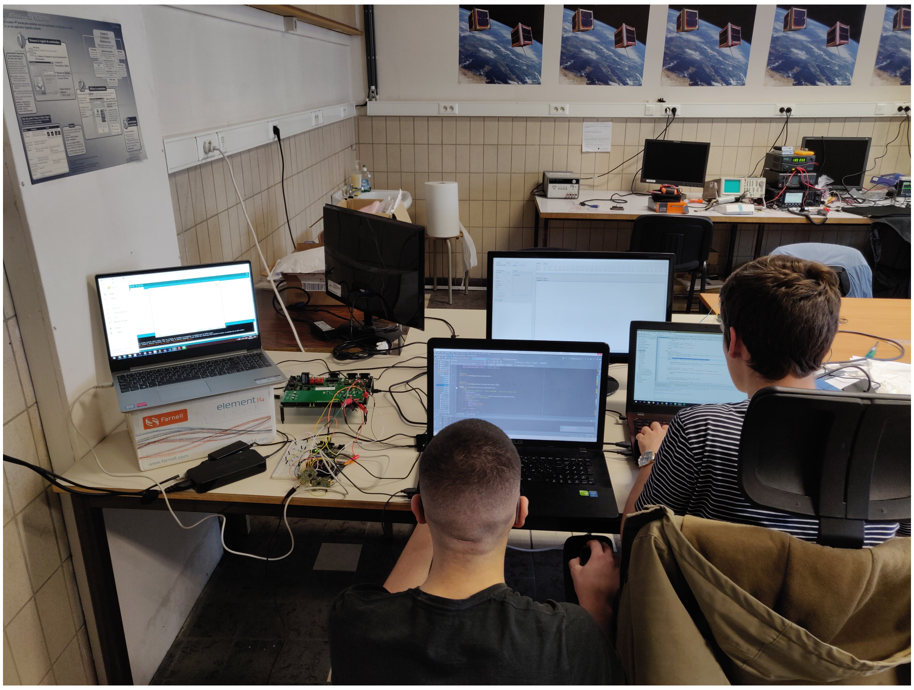


Figure 3.25 – Setup for SPI communications test with the OBC. The left computer shows the UART output from the Arduino, Soulaimane on the central laptop configures and runs the bus sampling with *PulseView* and Guillaume on the right debugs the OBC code

command, the other retrieves the log. To keep the flying software as simple as possible, it only relays the commands and logs respectively to and from the SPI bus. The SPI commands CRC is computed on ground. The OBC only checks for the acknowledgement and reports the result back to ground. For the logs, the AX.25 telecommand takes the offset as a parameter, so that a specific re-transmission can be initiated from ground in the case of an incorrect logs CRC. Figures 3.24 and 3.25 show pictures of the test setup and Figures 3.26 and 3.27 show the SPI bus during the communications. Multiple different commands have been tested with and without parameters.

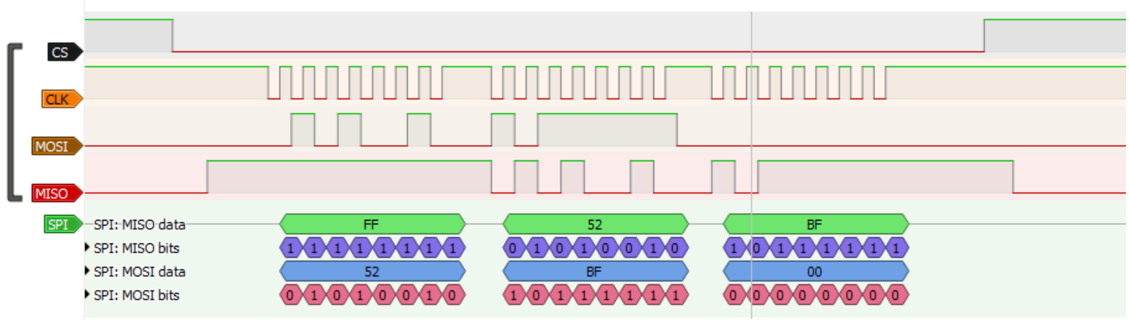


Figure 3.26 – Measurement of the SPI bus during the transmission of a REPEATER command, displayed on *PulseView*

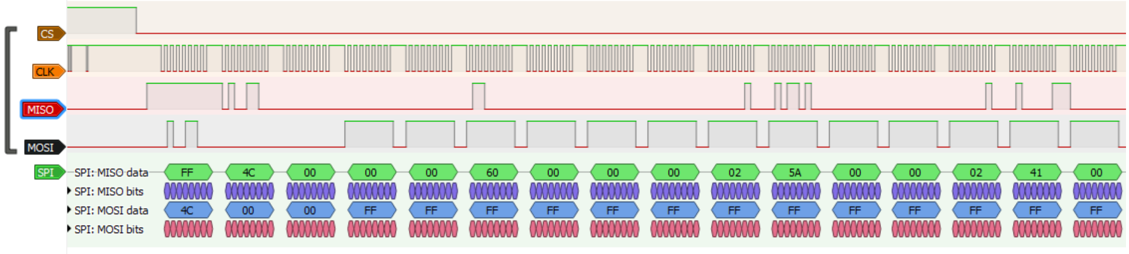

CS
CLK
MISO
MOSI
SPI
SPI: MISO data
SPI: MISO bits
SPI: MOSI data
SPI: MOSI bits
FF 4C 00 00 00 60 00 00 00 02 5A 00 00 02 41 00
4C 00 00 FF FF FF FF FF FF FF FF FF FF FF FF FF

Figure 3.27 – Measurement of the SPI bus during the transmission of a LOGS command and its response, displayed on *PulseView*

The good reception of the command by the D-STAR microcontroller was checked on the UART debug output. The good transmission of the retrieved logs were checked on the temporary ground software.

3.11 Communications circuit board

The communications circuit board contains the ADF7021 transceivers, the amplifiers, filters, splitter and combiner for the AX.25 and D-STAR communications. It is connected to both antennas, as well as the beacon output by 3 HFL connectors. The general RF components are depicted in Figure 1.3. But the board also contains current protectors, current and voltage measurement circuits and ADCs.

All OUFTI-2 boards follow a specific format called PC104, often used with CubeSats. It was not directly designed for that purpose but its form factor fits well into the cube. Multiple boards are stacked, and they are connected together by a 104 pins bus that goes through them all. This bus transports power and signals between the different subsystems.

The schematics were already drawn and some parts of the layout were already routed by X. Werner. During this work, a few corrections and modifications were performed relative to the PC104 connector attributions and the D-STAR microcontroller. The current schematics are attached in Appendix A and the board layout is shown in Figure 3.28.

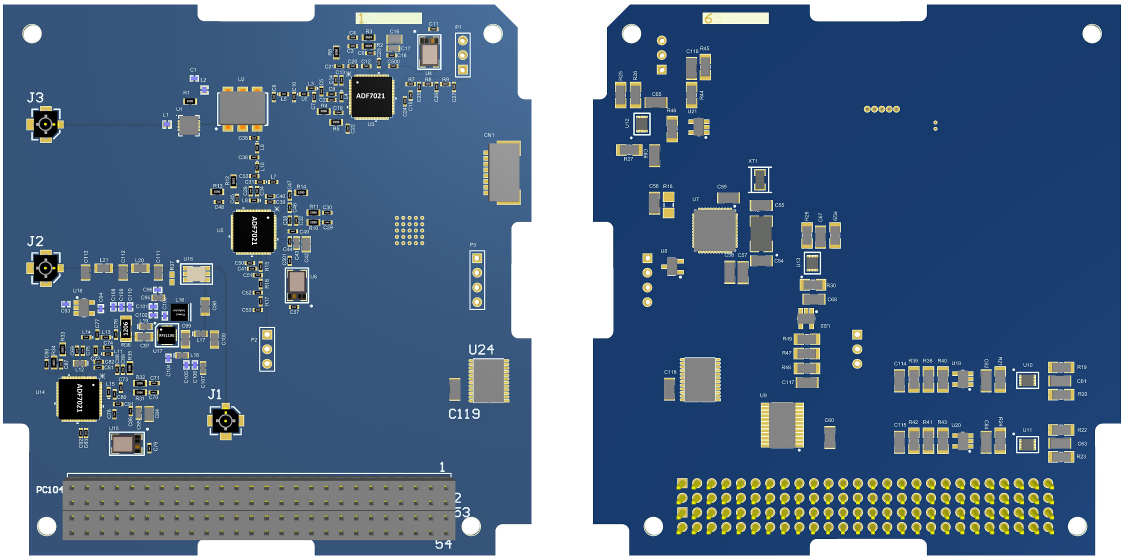
This board creates 4 different 3.3 V power buses from the general 3.3 V supply. Each of these is controlled by an enable signal and includes current protection (with a monitored fault signal), and voltage and current measurement circuits.

VCC_PA controlled by **EN_PA** powers the Power Amplifier. It is enabled only when transmitting, either by the OBC or the D-STAR repeater, depending on the multiplexer setting **MUX_CTRL** determined by the OBC. The Power Amplifier is the most consuming component of the nanosatellite. It is therefore very important to limit its running duration.

VCC_TX powers the transmission ADF7021 and associated circuitry. It is controlled by the OBC

VCC_AX25 powers the AX.25 reception ADF7021 and associated circuitry. It is controller by the OBC

VCC_DSTAR powers the D-STAR repeater microcontroller, its reception ADF7021 and associated circuitry. It is controlled by the OBC.



(a) Front side

(b) Back side

Figure 3.28 – Communications board 3D view (incomplete routing)

3.12 Conclusion

The D-STAR microcontroller implementation is finished and has been thoroughly tested with the ADF7021 development boards. Its interactions with external components, such as the OBC and the Power Amplifier have also been simulated, and even directly tested in the OBC case.

The code is ready to fly as is, even though some things could still be optimized. The use of a low-power mode of the microcontroller can be added when the repeater is idling. It was not judged necessary as the microcontroller is already completely shut down by the OBC when inactive, and its current draw is negligible compared to the Power Amplifier when active. The current pre-recorded parrot messages were added for testing purposes. New ones with more meaningful or useful messages can be easily recorded and added to the microcontroller, as there is a lot of memory space left.

Further work includes testing with all the external RF components (namely the amplifiers, splitter, combiner, filters, ...), testing the repeater at space-like power levels and testing the final PCB. Special attention should be paid to the RF5110G Power Amplifier operations, as it is controlled by the D-STAR repeater but they have never been tested together. Its datasheet specifies a turn on time of at most 100 ns [52], so there should not be any issue as the transmission begins several tens of microseconds after the **PA_EN** pin is set high.

It should be noted that the clock system of the D-STAR microcontroller was configured to provide the lowest drive current to both oscillators, in order to reduce consumption (see LFXTDRIVE and HFXTDRIVE in `main.c`). It worked without issue during testing. The final oscillators will be different. If the clocks seem to malfunction with the new oscillators, this drive current can easily be increased. However, there is no reason it should be necessary.

All the tools necessary for further development, message recording and testing have been provided. The Arduino debug program not only allows for individual testing of all the relay features and parrot message programming, but its source code can also serve as a basis for the D-STAR SPI command encoding and logs decoding in the ground software.

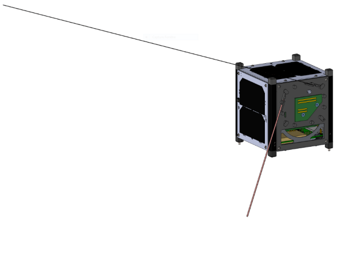
Chapter 4

Conclusion

OUFTI-2 slowly materializes and will soon come to a conclusion. All the different subsystems are at different levels of progress. Lessons learned from OUFTI-1 have turned into change lists, of different sizes for each subsystem, gathered in Master's thesis subjects. OUFTI-2 is the continuation of OUFTI-1, benefiting from the experience of 10 generations of students. However, every choice made for OUFTI-1 had to be reconsidered, and many parts redesigned or reimplemented, to make OUFTI-2 a truly new, hopefully better, nanosatellite.

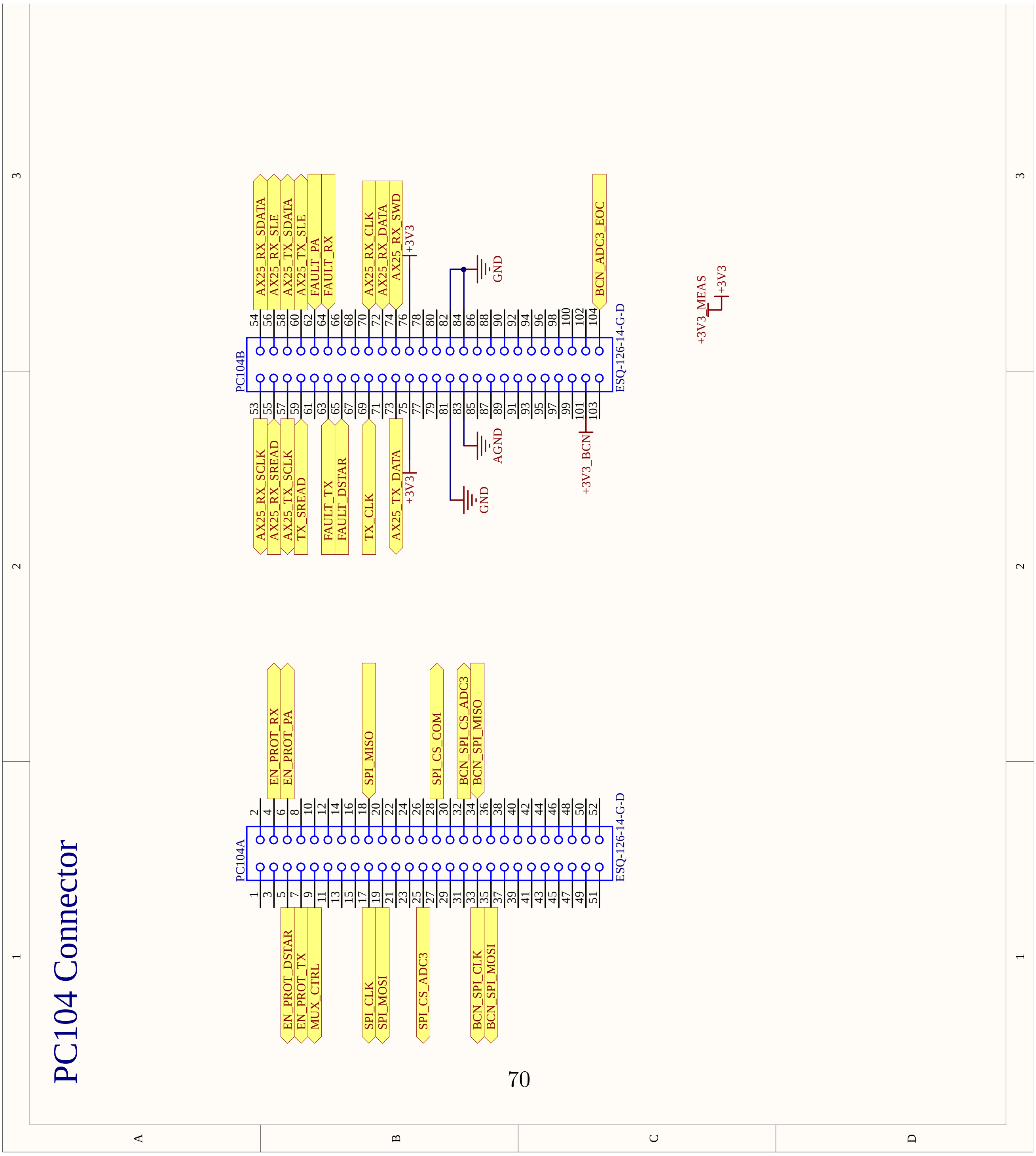
This work concentrates on its telecommunications. A major issue in the AX.25 uplink transmission process, that made it into the flying version of OUFTI-1, was fixed. It avoids the On-Board Computer to be interrupted 9600 times per second even when there is no useful data to receive. The D-STAR repeater was re-developed with various fixes, a cleaner, stable and reliable code as well as new features. The main addition is the parrot mode, which allows to transmit pre-recorded D-STAR messages, keeping the relay usable even if reception is impossible. A full logging system was also implemented.

The communications part is almost ready to be launched. Though, one of the most important phase still remains: the complete testing of all the systems, on their final PCB, in the harsh conditions that they will encounter in space, especially in terms of RF power, which has proven not to be trivial. The full satellite will also need to be tested with all the systems interacting with each other. Besides the RF inputs and outputs, the communications subsystem interfaces with the On-Board Computer. The SPI link between the D-STAR repeater and the OBC has already been verified. All the necessary tools and information are given to help with further testing and integration of this subsystem.

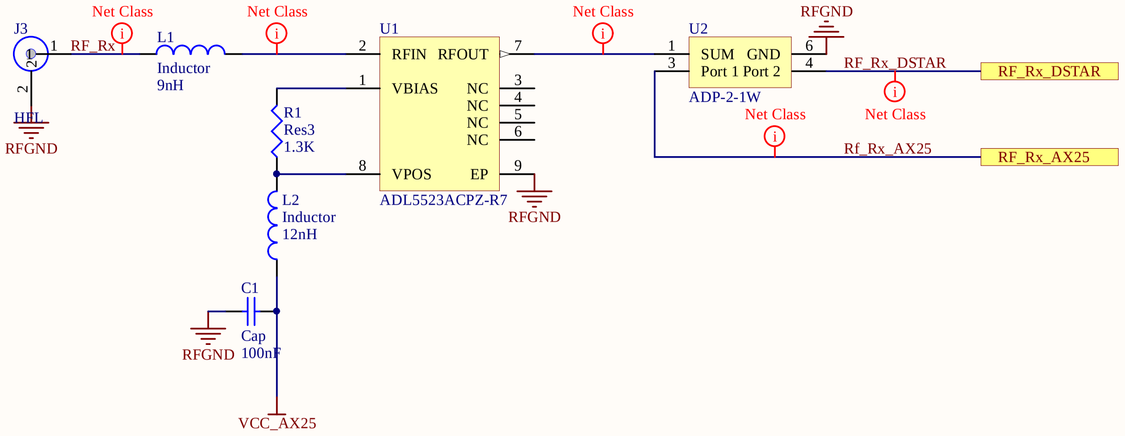


Appendix A

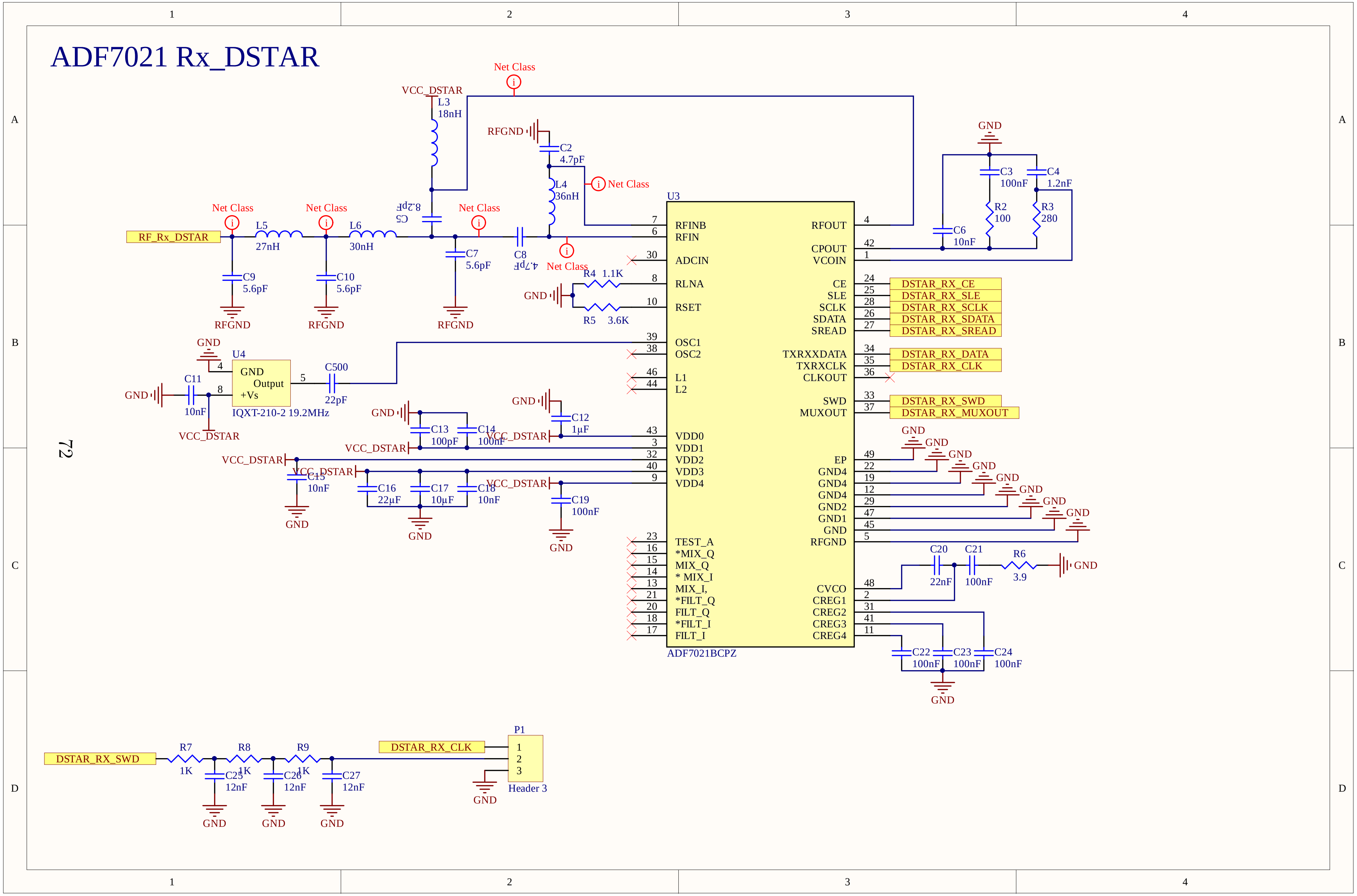
Communications circuit schematics



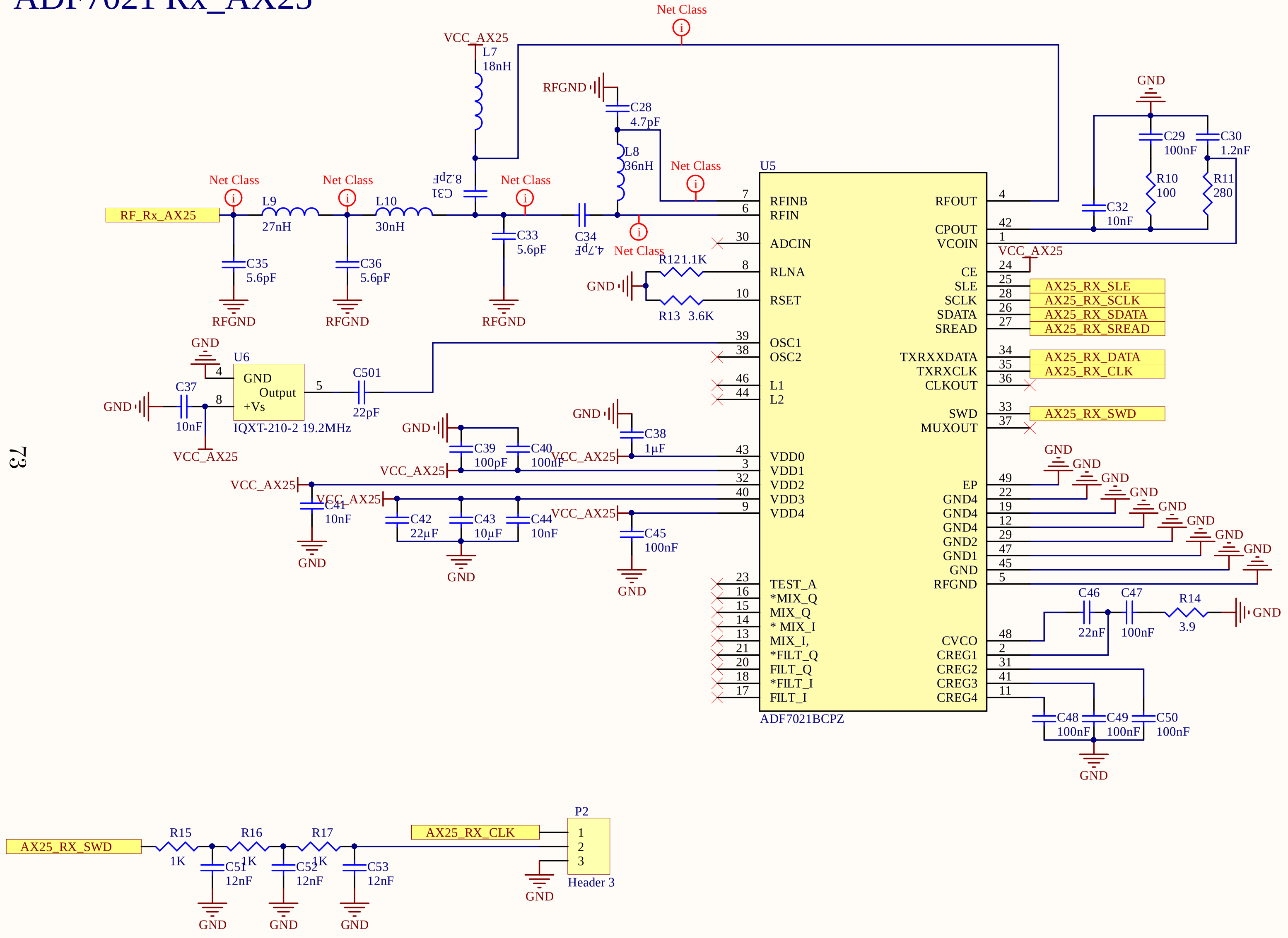
Low Noise Amplifier



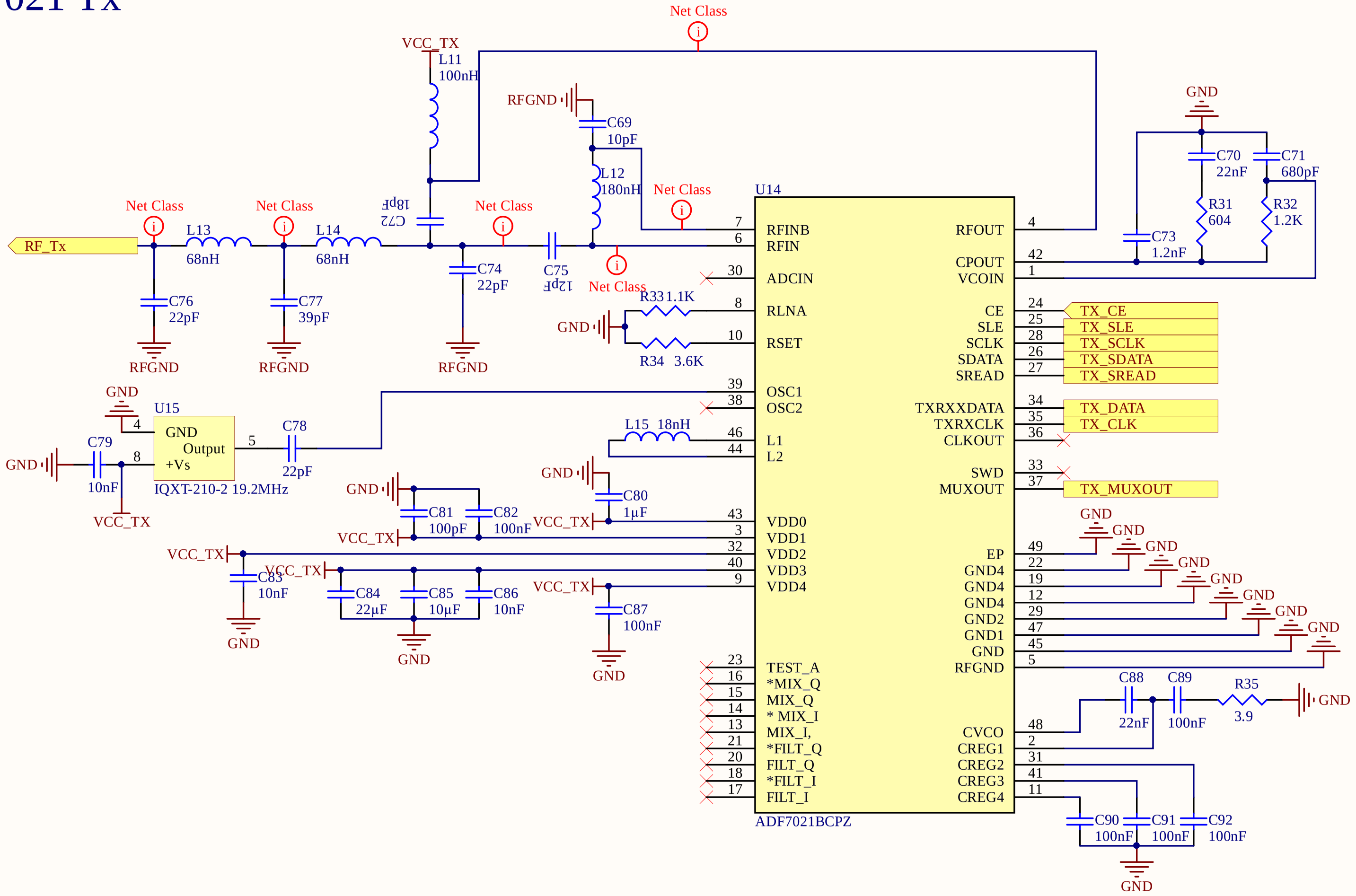
ADF7021 Rx_DSTAR



ADF7021 Rx_AX25

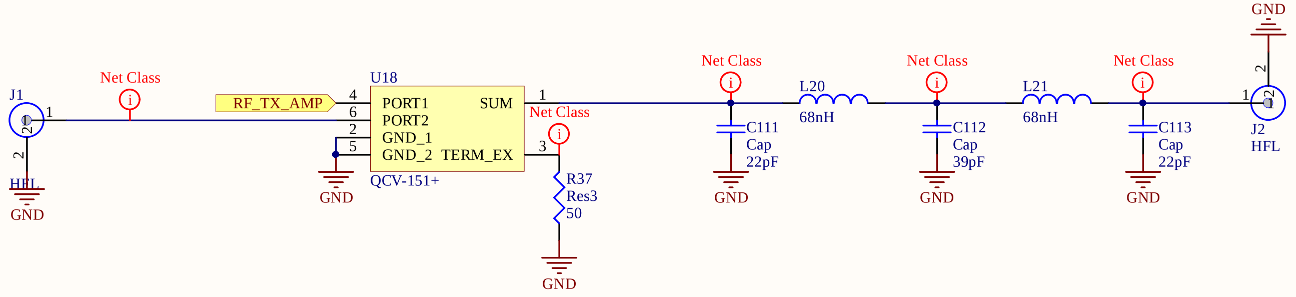


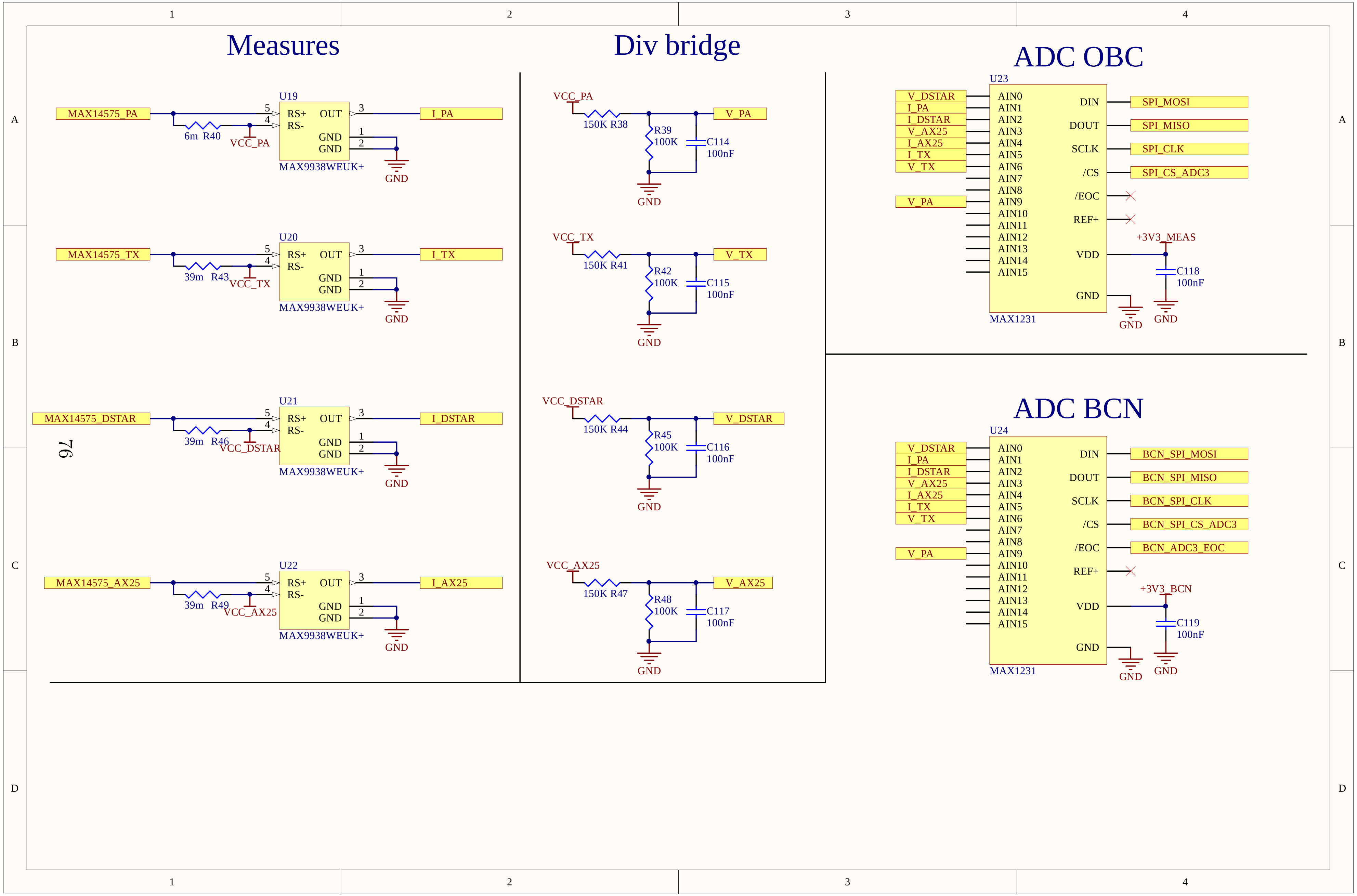
ADF7021 Tx

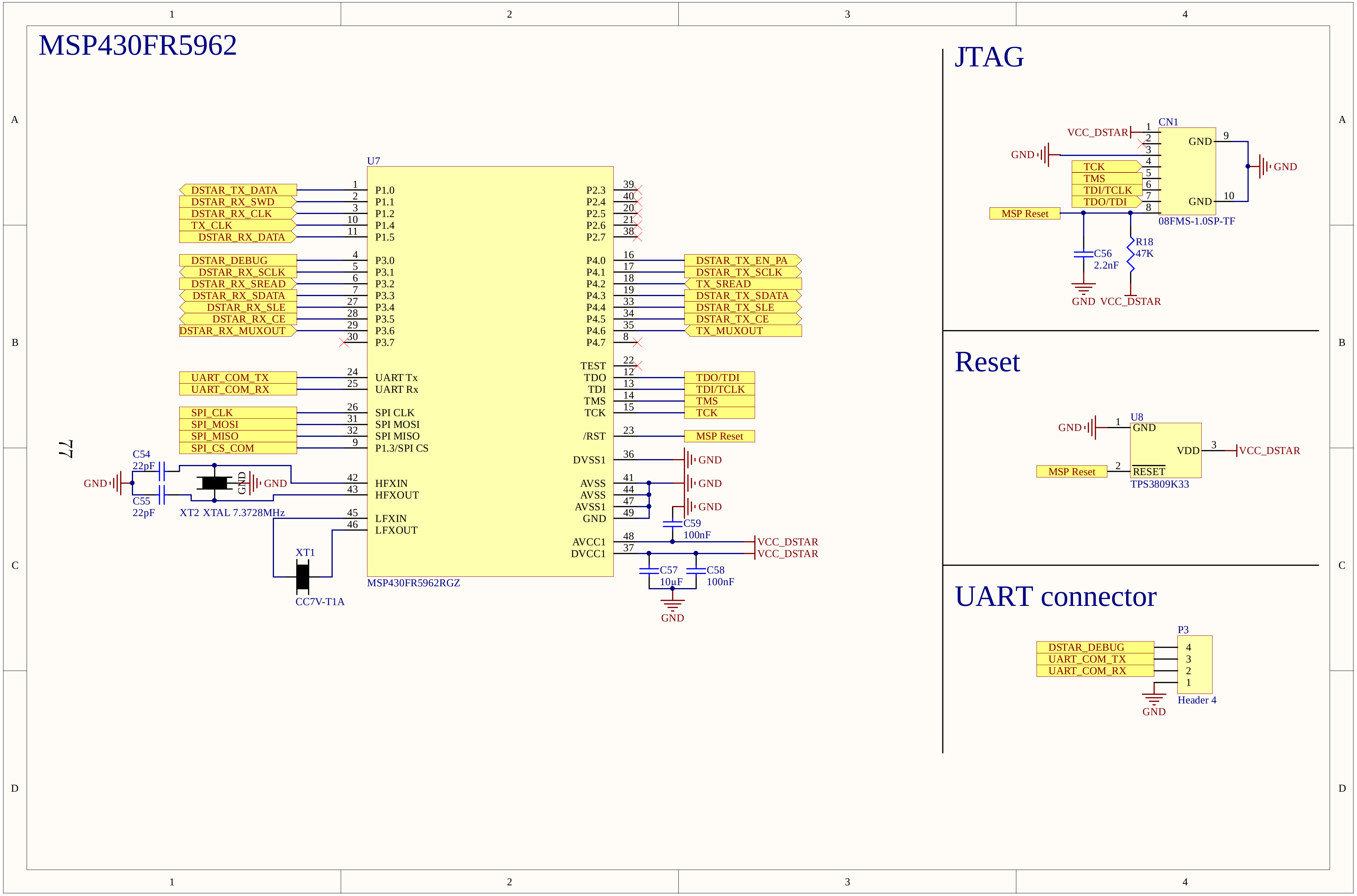


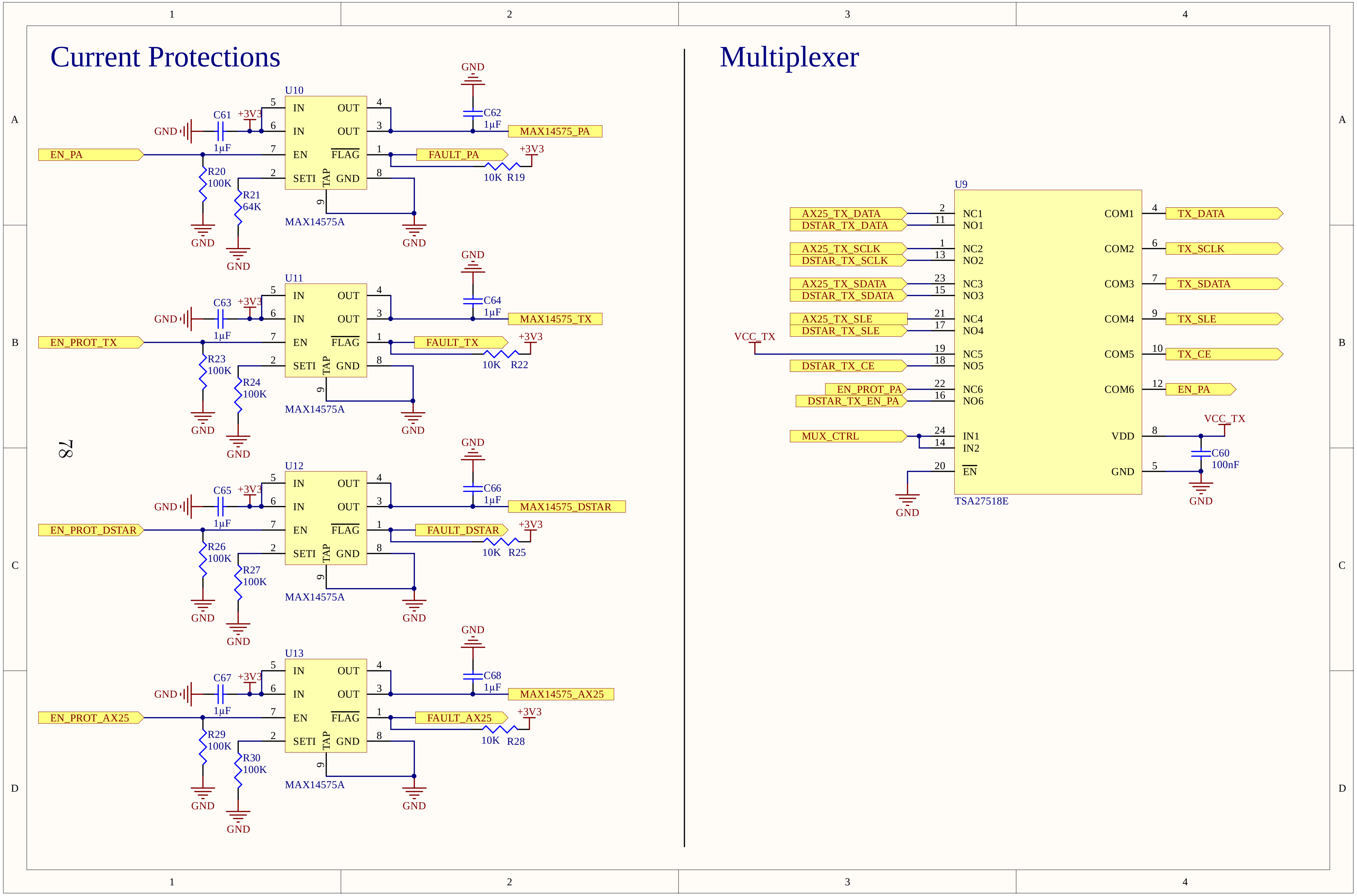
Net Class
VCC_TX
L11
100nH
RFGND
C69
10pF
L12
180nH
Net Class
U14
RF_Tx
Net Class
L13
68nH
Net Class
L14
68nH
C72
18pF
Net Class
C74
22pF
C75
12pF
Net Class
C76
22pF
C77
39pF
RFGND
RFGND
RFGND
R33 1.1K
GND
R34 3.6K
GND
U15
4
GND
Output
+Vs
5
8
C78
22pF
C79
10nF
GND
VCC_TX
IQXT-210-2 19.2MHz
L15 18nH
GND
C80
1µF
GND
C81
100pF
C82
100nF
VCC_TX
VCC_TX
VCC_TX
C83
10nF
VCC_TX
C84
22µF
C85
10µF
C86
10nF
VCC_TX
C87
100nF
GND
GND
GND
7 RFINB
6 RFIN
30 ADCIN
8 RLNA
10 RSET
39 OSC1
38 OSC2
46 L1
44 L2
43 VDD0
3 VDD1
32 VDD2
40 VDD3
9 VDD4
23 TEST_A
16 *MIX_Q
15 MIX_Q
14 * MIX_I
13 MIX_I,
21 *FILT_Q
20 FILT_Q
18 *FILT_I
17 FILT_I
ADF7021BCPZ
RFOUT 4
CPOUT 42
VCOIN 1
CE 24
SLE 25
SCLK 28
SDATA 26
SREAD 27
TXRXXDATA 34
TXRXCLK 35
CLKOUT 36
SWD 33
MUXOUT 37
EP 49
GND4 22
GND4 19
GND4 12
GND2 29
GND1 47
GND 45
RFGND 5
CVCO 48
CREG1 2
CREG2 31
CREG3 41
CREG4 11
TX_CE
TX_SLE
TX_SCLK
TX_SDATA
TX_SREAD
TX_DATA
TX_CLK
TX_MUXOUT
GND
C70
22nF
C71
680pF
R31
604
R32
1.2K
C73
1.2nF
GND
GND
GND
GND
GND
GND
GND
GND
C88
22nF
C89
100nF
R35
3.9
GND
C90
100nF
C91
100nF
C92
100nF
GND

Combiner


J1
1
2
2
HFL
GND
Net Class
RF_TX_AMP
U18
4
6
2
5
PORT1
PORT2
GND_1
GND_2
SUM
TERM_EX
QCV-151+
GND
1
3
Net Class
R37
Res3
50
GND
Net Class
C111
Cap
22pF
GND
L20
68nH
Net Class
C112
Cap
39pF
GND
L21
68nH
Net Class
C113
Cap
22pF
GND
GND
2
1
2
J2
HFL

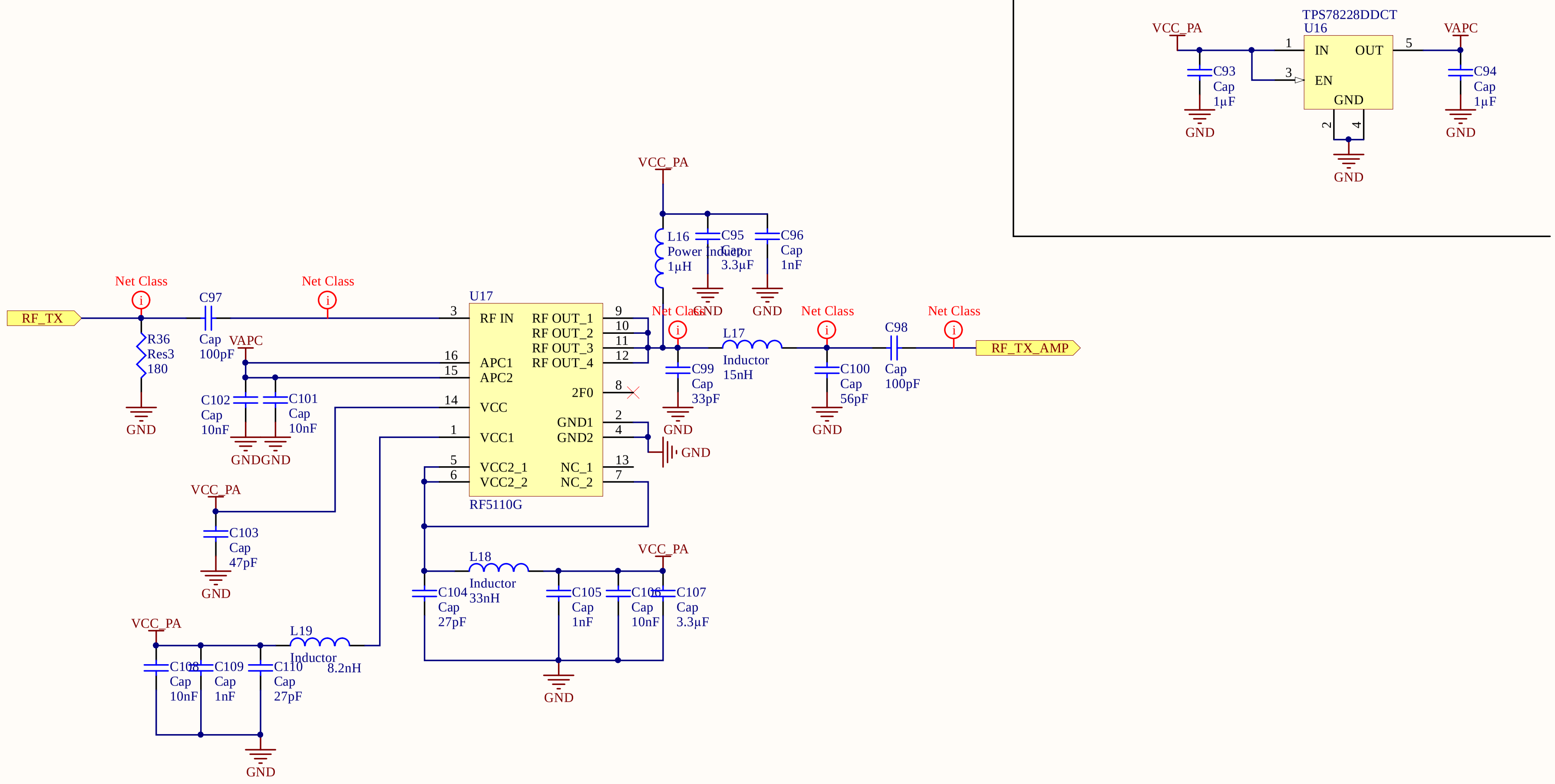

Measures
Div bridge
ADC OBC
ADC BCN
1
2
3
4
A
B
C
D
MAX14575_PA
6m R40
VCC_PA
U19
RS+
RS-
OUT
GND
GND
MAX9938WEUK+
I_PA
GND
MAX14575_TX
39m R43
VCC_TX
U20
MAX9938WEUK+
I_TX
MAX14575_DSTAR
39m R46
VCC_DSTAR
U21
MAX9938WEUK+
I_DSTAR
MAX14575_AX25
39m R49
VCC_AX25
U22
MAX9938WEUK+
I_AX25
5
4
3
1
2
VCC_PA
150K R38
R39
100K
C114
100nF
V_PA
VCC_TX
150K R41
R42
100K
C115
100nF
V_TX
VCC_DSTAR
150K R44
R45
100K
C116
100nF
V_DSTAR
VCC_AX25
150K R47
R48
100K
C117
100nF
V_AX25
U23
V_DSTAR
I_PA
I_DSTAR
V_AX25
I_AX25
I_TX
V_TX
V_PA
AIN0
AIN1
AIN2
AIN3
AIN4
AIN5
AIN6
AIN7
AIN8
AIN9
AIN10
AIN11
AIN12
AIN13
AIN14
AIN15
DIN
DOUT
SCLK
/CS
/EOC
REF+
VDD
GND
MAX1231
SPI_MOSI
SPI_MISO
SPI_CLK
SPI_CS_ADC3
+3V3_MEAS
C118
100nF
U24
MAX1231
BCN_SPI_MOSI
BCN_SPI_MISO
BCN_SPI_CLK
BCN_SPI_CS_ADC3
BCN_ADC3_EOC
+3V3_BCN
C119
100nF
GND


MSP430FR5962
U7
MSP430FR5962RGZ
DSTAR_TX_DATA 1 P1.0
DSTAR_RX_SWD 2 P1.1
DSTAR_RX_CLK 3 P1.2
TX_CLK 10 P1.4
DSTAR_RX_DATA 11 P1.5
DSTAR_DEBUG 4 P3.0
DSTAR_RX_SCLK 5 P3.1
DSTAR_RX_SREAD 6 P3.2
DSTAR_RX_SDATA 7 P3.3
DSTAR_RX_SLE 27 P3.4
DSTAR_RX_CE 28 P3.5
DSTAR_RX_MUXOUT 29 P3.6
30 P3.7
UART_COM_TX 24 UART Tx
UART_COM_RX 25 UART Rx
SPI_CLK 26 SPI CLK
SPI_MOSI 31 SPI MOSI
SPI_MISO 32 SPI MISO
SPI_CS_COM 9 P1.3/SPI CS
42 HFXIN
43 HFXOUT
45 LFXIN
46 LFXOUT
P2.3 39
P2.4 40
P2.5 20
P2.6 21
P2.7 38
P4.0 16 DSTAR_TX_EN_PA
P4.1 17 DSTAR_TX_SCLK
P4.2 18 TX_SREAD
P4.3 19 DSTAR_TX_SDATA
P4.4 33 DSTAR_TX_SLE
P4.5 34 DSTAR_TX_CE
P4.6 35 TX_MUXOUT
P4.7 8
TEST 22
TDO 12 TDO/TDI
TDI 13 TDI/TCLK
TMS 14 TMS
TCK 15 TCK
/RST 23 MSP Reset
DVSS1 36 GND
AVSS 41 GND
AVSS 44
AVSS1 47 GND
GND 49
C59 100nF
AVCC1 48 VCC_DSTAR
DVCC1 37 VCC_DSTAR
C57 10µF
C58 100nF
GND
C54 22pF
C55 22pF
GND
GND
GND
XT2 XTAL 7.3728MHz
XT1
CC7V-T1A
JTAG
CN1
VCC_DSTAR 1
2
GND 3
TCK 4
TMS 5
TDI/TCLK 6
TDO/TDI 7
MSP Reset 8
GND 9
GND 10
GND
08FMS-1.0SP-TF
C56 2.2nF
R18 47K
GND VCC_DSTAR
Reset
U8
GND 1 GND
VDD 3 VCC_DSTAR
MSP Reset 2 RESET
TPS3809K33
UART connector
P3
DSTAR_DEBUG 4
UART_COM_TX 3
UART_COM_RX 2
1
GND
Header 4
1
2
3
4
A
B
C
D


Current Protections
Multiplexer
1
2
3
4
A
B
C
D
U10
U11
U12
U13
MAX14575A
IN
OUT
EN
FLAG
SETI
TAP
GND
EN_PA
EN_PROT_TX
EN_PROT_DSTAR
EN_PROT_AX25
MAX14575_PA
MAX14575_TX
MAX14575_DSTAR
MAX14575_AX25
FAULT_PA
FAULT_TX
FAULT_DSTAR
FAULT_AX25
C61
C62
C63
C64
C65
C66
C67
C68
1µF
R19
R20
R21
R22
R23
R24
R25
R26
R27
R28
R29
R30
10K
100K
64K
+3V3
GND
U9
TSA27518E
NC1
NO1
NC2
NO2
NC3
NO3
NC4
NO4
NC5
NO5
NC6
NO6
IN1
IN2
EN
COM1
COM2
COM3
COM4
COM5
COM6
VDD
AX25_TX_DATA
DSTAR_TX_DATA
AX25_TX_SCLK
DSTAR_TX_SCLK
AX25_TX_SDATA
DSTAR_TX_SDATA
AX25_TX_SLE
DSTAR_TX_SLE
VCC_TX
DSTAR_TX_CE
EN_PROT_PA
DSTAR_TX_EN_PA
MUX_CTRL
TX_DATA
TX_SCLK
TX_SDATA
TX_SLE
TX_CE
EN_PA
C60
100nF

Power Amplifier

LDO 2.8V



Bibliography

[1] E. Kulu, *Nanosats database*, www.nanosats.eu (accessed May 17, 2019).

[2] X. Werner, V. Broun, S. De Dijcker, S. Habraken, G. Kerschen, and J. Verly, "Cubesats activities at the university of liège", hdl.handle.net/2268/212433, Paper presented at UBA National Congress - 2017, Redu (Euro Space Center), Belgium, May 2017.

[3] X. Werner, S. De Dijcker, V. Broun, G. Kerschen, and J. Verly, "Architecture of educational oufti-1 nanosatellite of university of liège, as tested in preparation for space flight", hdl.handle.net/2268/185918, Poster session presented at 9th European CubeSat Symposium, Liège, Belgique, Sep. 2015.

[4] S. De Dijcker, V. Broun, X. Werner, and J. Verly, "Oufti-2: status report on the design and construction of the second educational nanosatellite featuring d-star amateur-radio communications", hdl.handle.net/2268/210408, Poster session presented at 2017 CubeSat Developers Workshop, Cal Poly Performing Arts Center, San Luis Obispo, CA, USA, Apr. 2017.

[5] V. Broun, S. De Dijcker, X. Werner, A. Carapelle, and J. Verly, "Description of secondary payload of upcoming educational oufti-2 1u cubesat for testing a new multilayer shield for protecting electronics against space radiations", hdl.handle.net/2268/235121, Poster session presented at 10th European Cubesat Symposium, Dec. 2018.

[6] N. Marchal, *Implémentation, design et test de la carte électronique de télécommunication du cubesat oufti-1*, Available at www.leodium.ulg.ac.be/cmsms/index.php?page=technical-documents (accessed October 9, 2018), Haute Ecole Libre Mosane - Gramme, 2010.

[7] *High performance narrow-band transceiver ic datasheet*, ADF7021, Rev. D, Analog Devices, www.analog.com/media/en/technical-documentation/data-sheets/ADF7021.pdf (accessed October 26, 2018).

[8] J. Hardy, *Implémentation du protocole ax.25 à bord du nanosatellite oufti-1*, Available at www.leodium.ulg.ac.be/cmsms/index.php?page=technical-documents (accessed October 9, 2018), Haute Ecole Libre Mosane - Gramme, 2009.

[9] W. A. Beech NJ7P, D. E. Nielsen N7LEM, and J. Taylor N7OO, *Ax.25 link access protocol for amateur packet radio*, Tucson Amateur Packet Radio Corporation, www.tapr.org/pdf/AX25.2.2.pdf (accessed October 26, 2018), Jul. 1998.

[10] *Oufti-1 ground software source code.*

[11] *Oufti-1 on-board computer software source code.*

[12] S. De Dijcker, *Implémentation de la gestion des télécommandes et des télémétries au sein de l'ordinateur de bord du nanosatellite oufti-1*, Available at `www.leodium.ulg.ac.be/cmsms/index.php?page=technical-documents` (accessed October 9, 2018), Haute Ecole de la Province de Liège - Catégorie Technique (ISIL), 2011.

[14] *Instruction manual / command description for the tracker/ dsptnc*, Firmware Version 1.7, SCS, `www.p4dragon.com/download/SCS_Manual_DSPTNC_1.7.pdf` (accessed October 26, 2018), Jan. 2018.

[15] *Vhf/uhf all mode transceiver instruction manual*, IC-910H, ICOM, `www.icom.co.jp/world/support/download/manual/pdf/IC-910H_9a.pdf` (accessed October 26, 2018).

[16] *Hf/vhf/uhf transceiver instruction manual*, IC-9100, ICOM, `www.icom.co.jp/world/support/download/manual/pdf/IC-9100_ENG_2a.pdf` (accessed October 26, 2018).

[17] *Vhf/uhf all mode transceiver service manual*, IC-910H, ICOM, `www.radiomanual.info/schemi/ICOM_VU/IC-910H_serv.pdf` (accessed October 26, 2018).

[19] J. Miller G3RUH, "9600 baud packet radio modem design", *ARRL 7th Computer Networking Conference (US)*, Oct. 1988, `www.amsat.org/amsat/articles/g3ruh/109.html` (accessed October 26, 2018).

[22] A. V. Oppenheim, A. S. Willsky, and S. H. Nawab, *Signals & Systems (2Nd Ed.)* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996, ISBN: 0-13-814757-4.

[23] M. A. Soto, M. Alem, M. Amin Shoaie, A. Vedadi, C.-S. Brès, L. Thévenaz, and T. Schneider, "Optical sinc-shaped nyquist pulses of exceptional quality", *Nature Communications*, vol. 4, p. 2898, Dec. 2013, Article. [Online]. Available: `doi.org/10.1038/ncomms3898`.

[24] wb2osz, *Direwolf github repository*, `github.com/wb2osz/direwolf` (accessed November 23, 2018).

[31] R. Henrard, *Réalisation du système de télécommunicatio du nano-satellite oufti-1*, Available at `www.leodium.ulg.ac.be/cmsms/index.php?page=technical-documents` (accessed October 9, 2018), Haute Ecole de la Province de Liège - Catégorie Technique (ISIL), 2009.

[32] A. Hay, *Nanosatellite oufti-1 : conception et implémentation des étages de réception et d'émission radiofréquences*, Available at `www.leodium.ulg.ac.be/cmsms/index.php?page=technical-documents` (accessed October 9, 2018), Haute Ecole Libre Mosane - Gramme, 2012.

[33] *400 mhz to 4000 mhz low noise amplifier*, ADL5523, Rev. C, Analog Devices, `www.analog.com/media/en/technical-documentation/data-sheets/ADL5523.pdf` (accessed November 23, 2018).

[35] Japan Amateur Radio League, *Translated d-star system specification*, `www.jarl.com/d-star/shogen.pdf` (accessed February 21, 2019), 2005.

[36] P. Loveall AE5PL, *D-star uncovered*, `www.aprs-is.net/downloads/DStar/DSTARUncovered.pdf` (accessed February 21, 2019), 2008.

[37] J. Pisane, *Design and implementation of the terrestrial and space telecommunication elements of the student nanosatellite of the university of liege*, Available at www.leodium.ulg.ac.be/cmsms/index.php?page=technical-documents (accessed October 9, 2018), University of Liège, 2008.

[38] D. Bederov DL3OCK, *Dstar radio frame structure in dv mode*, db0fhn.efi.fh-nuernberg.de/lib/exe/fetch.php?media=projects:dstar:ircddb:dstar_dv_frame3_en.pdf (accessed February 21, 2019), 2008.

[39] A. Gerstlauer, *Convolutional (viterbi) encoding*, The University of Texas at Austin, 2009.

[40] N. Crosset, *Implémentation du relais d-star à bord du nanosatellite oufti-1*, Available at www.leodium.ulg.ac.be/cmsms/index.php?page=technical-documents (accessed October 9, 2018), Haute Ecole Libre Mosane - Gramme, 2010.

[41] X. Werner, N. Vetcour, and J. Pisane, "Implémentation du protocole d-star sur un transceiver radioamateur classique", *Revue Scientifique des ISILF no. 26*, 2012, www.isilf.be/Articles/ISILF12p225gramme.pdf (accessed April 26, 2019).

[42] *Oufti-1 d-star relay software source code.*

[43] F. Mahy, *Design and implementation of on-board telecommunication system of student nanosatellite oufti-1 of university of liège*, Available at www.leodium.ulg.ac.be/cmsms/index.php?page=technical-documents (accessed October 9, 2018), University of Liège, 2009.

[44] R. Labeye, *Design, implémentation et test du système électronique de télécommunication du nano-satellite étudiant oufti-1*, University of Liège, 2011.

[45] *Mixed signal microcontroller datasheet*, MSP430F1612, Rev. G, Texas Instruments, www.ti.com/lit/gpn/msp430f1612 (accessed November 23, 2018).

[46] Texas Instruments, *Fram faqs*, www.ti.com/lit/ml/slat151/slat151.pdf (accessed November 23, 2018), 2014.

[47] *Mixed-signal microcontroller datasheet*, MSP430FR5962, Rev. C, Texas Instruments, www.ti.com/lit/gpn/msp430fr5962 (accessed November 23, 2018).

[48] Anthony F4GOH, *Dstar arduino header encoding and decoding library*, github.com/f4goh/DSTAR (accessed April 26, 2019), 2015.

[49] *Msp430fr58xx, msp430fr59xx, and msp430fr6xx family user's guide*, MSP430FR5962, Rev. O, Texas Instruments, www.ti.com/lit/pdf/slau367 (accessed February 21, 2019).

[50] P. Dhaker, *Introduction to spi interface*, www.analog.com/media/en/analog-dialogue/volume-52/number-3/introduction-to-spi-interface.pdf (accessed February 21, 2019), Analog Dialogue, Sep. 2018.

[51] *Dual band fm transceiver instruction manual*, IC-E2820, ICOM, www.icom.co.jp/world/support/download/manual/pdf/IC-E2820.pdf (accessed February 21, 2019).

[52] *3v general purpose/gsm power amplifier datasheet*, RF5110G, Qorvo, www.qorvo.com/products/d/da000459 (accessed April 26, 2019), May 2018.

Image sources

[13] *Tracker/dsp tnc picture*, `www.scs-ptc.com`, With written consent, SCS.

[18] *Ic-910h, ic-9100 and ic-e2820 pictures*, `www.icom-france.com`, With written consent, ICOM.

[20] Krishnavedala, *Raised cosine filter*, `commons.wikimedia.org/w/index.php?title=File:Raised-cosine_filter.svg&oldid=198726847` (accessed May 4, 2019), Wikimedia Commons, Jun. 2011.

[21] Modified script from Chris828, *Raised cosine isi*, `commons.wikimedia.org/w/index.php?title=File:Raised-cosine-ISI.svg&oldid=212639954` (accessed May 4, 2019), Wikimedia Commons, Mar. 2015.

[25] Benedikt Seidl, *3.5mm jack plug*, `commons.wikimedia.org/w/index.php?title=File:3.5mm_jack_plug_4.svg&oldid=278814312` (accessed May 4, 2019), Wikimedia Commons, Jun. 2008.

[26] *Usb cable icon*, `www.flaticon.com/free-icon/usb-cable_196513` (accessed May 4, 2019), Icon made by Freepik from www.Flaticon.com.

[27] *Laptop icon*, `www.shareicon.net/laptop-technology-computer-computing-electronic-800724` (accessed May 4, 2019), Icon from www.shareicon.net.

[28] *Arduino uno board image*, `fritzing.org`, Fritzing.

[29] *Adf7021 development board picture*, `www.analog.com`, Analog Devices.

[30] *Antenna picture*, `www.telcoantennas.com.au`, With written consent, Telco Antennas.

[34] *Msp-ts430rgz48c and msp-fet pictures*, `www.ti.com`, Texas Instruments.