UAB

Treball de Fi de Grau

Grau en Enginyeria de Sistemes de Telecomunicació

Control and evaluation of the AD-FMCOMMS5-EBZ software-defined radio

Guillem Foreman-Campins

Director: Dr. Josep Parrón Granados

Departament de Telecomunicació i Enginyeria de Sistemes

Universitat Autònoma de Barcelona

Bellaterra, Juliol 06, 2020

Abstract

Software-defined radios (SDR) have presented a new way to do telecommunication systems in a configurable, efficient and portable manner. With the rapid evolving capabilities of these systems, the traditional non-software-configurable ones are being left behind. This project explores the concept of software-defined radio in a theoretical and practical way using the AD-FMComms5-EBZ, a SDR provided by Analog Devices with 4 transceivers.

A complete analysis is done on the AD-FMComms5-EBZ, understanding all its physical components and its digital interface that make it software-configurable. Furthemore, its interaction with MATLAB/Simulink and with the IIO Oscilloscope application is studied, providing a profound explanation of the board's capabilities on these software. Finally, a few algorithms we found interesting are designed with MATLAB to enable all the potential of the board by reaching MIMO capabilities, analysing the phase of each transmitter and the reception order of symbols.

Resum

Les ràdios definides per software (SDR) s'han presentat com a una nova manera de fer sistemes de telecomunicacions gràcies a la seva configurabilitat, eficiència i portabilitat. La seva evolució ha sigut ràpida i contínua, fent que els sistemes tradicionals que no son configurables en software s'hagin quedat enrere. Aquest projecte explora el concepte de ràdio definida per software de forma teòrica i pràctica utilitzant l'AD-FMComms5-EBZ, una SDR dissenyada per Analog Devices que incorpora 4 transceptors.

En el projecte s'hi fa una anàlisi completa de l'AD-FMComms5-EBZ, entenent tots els seus components físics i la seva interfície digital que en permet la configurabilitat per via de software dels seus components. A més, s'estudia la seva interacció amb MATLAB/Simulink i amb l'aplicació IIO Oscilloscope, donant una explicació profusa de les capacitats de la placa amb aquest programari. Finalment, es dissenyen alguns algoritmes que hem trobat interessants amb MATLAB per a habilitar tot el potencial de la placa, aconseguint que funcioni com a un sistema MIMO, analitzant la fase ens els diferents transmissors i l'ordre de la recepció dels símbols.

Acknowledgments

I would like to express my gratitude to professor Josep Parrón for his help during this project and for his patience and serenity. I'd also like to thank all the public entities that have made this project possible with their financial support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, the most relevant factors to take into account when innovating a telecommunication system are velocity, effectiveness and simplicity. The software-defined radio technology (SDR), first mentioned in 1984[P.J85], drastically improves simplicity in comparison with other traditional telecommunication systems by digitalising all the components which were traditionally hardware while keeping high velocity and effectiveness.

As a historic introduction, the term software-defined radio was first introduced by a team from E-Systems Inc when refering to a bandbase digital receiver they made in 1984[P.J85]. This receiver was designed out of necessity for an ultra-fast processor to test a new signal processing technique they had come up with, and it was able to detect signals with a strength as low as -20 dBW. The SDR technology continued developing up to the point where, in 1991, the defense department of the United States created SpeakEasy I [Lac95], the first big scale software-defined radio project. This project was focused on unifying all the many types of radios and waveforms the US military used into only one, since using different kinds of radios prevented them from communicating in real time. This was a key step for software-defined radio since this feature is still one of the many advantages SDR has. The US defense department also wanted the SDR to be easily reconfigurable. It wasn't until 1994 when the SpeakEasy I first worked properly with carrier frequencies ranging from 2 Hz to 2 GHz, whilst also enabling software updates to include new features such as modulation or codification schemes. In 1993, J.Mitola described how SDRs should be in the future [Mit92]: a set of Digital Signal Processing (DSP) algorithms, a system to implement these algorithms into communications systems functions and a set of processors on which the software-defined radio works to provide real-time communications.

This definition is still eloquent today as software-defined radios have progressed into being applicable for a great range of communication applications, way beyond its initial military purpose.

In 1996 the first SDR forum was created to ensure the advancement of SDR-related technologies and one year later the US defense department creates the Joint Tactical Radio System (JTRS) to increase interoperability and waveform portability, which greatly stimulated advancements for the next decade. A great step was made by Eric Blossom when he founded GNU Radio in 2001, an open-source SDR development toolset that provides signal processing blocks to implement SDR platforms and signal processing systems. Its combination with external RF hardware create an SDR. Finally, in 2004 the first SDR was comercialised by Vanu Inc and, in 2009, Lime Microsystems comercialised the first radio frequency integrated circuit (RFIC) which could tune from 400 MHz to 4 GHz and supported a bandwidth of up to 28 MHz.

With the advancement of RFICs, cost-effective Field Programmable Gate Array (FPGA) that included Digital Signal Processors (DSP) and Electronic Design Automation (EDA), came the second of (until now) four generations of SDR innovation, which was driven by 4G LTE infrastructre. Practically all LTE base stations were developed with RFICs and FPGAs.
With low-power and high-performance DSP cores came the third generation of SDR advancement, when 4G LTE handsets moved to SDR architectures. Hence, SDR platforms increased in volume and became the industry standard for all radios. The fourth generation is expected to come with the arrival of 5G and IoT technologies which will further increase the volume and magnitude of SDR platforms. The evolution of SDR technologies is shown in figure 1.1.

This project focuses on the evaluation of Analog Devices' AD-FMComms5-EBZ SDR[Anad], the basis of which is explained profusely in Chapter 2. This SDR platform will be used for this project because it's the one we had in the lab, but there are many other good options like National Instruments' USRP 2922 [Nat], the Wireless Open Access Research Platform (WARP) [Man], Ettus' b210 [Ett], Nuand's bladeRF [Nua], Nutaq's PicoSDR [Nut] or Lime Microsystems' LimeSDR [Lim].

1.1 Introduction to Software-Defined Radio

The combination of digital signal processing and analog radiofrequency systems (RF) has always made up communication systems. Nowadays, the signal processing systems have evolved in such a way that the vast majority of the baseband functionality is implemented in software and RF hardware is reconfigurable enough so that one radio front-end handles most RF systems. The combination of the RF flexibility and the implementation of signal processing in software has lead to the creation of software-defined radio (SDR).

Software-defined radio can be defined as a wireless device that is capable of implementing both the functionality of the transmitters and the receivers with software, without making any changes to the hardware. It was initally developed to replace radio tonifiers and certain filters.

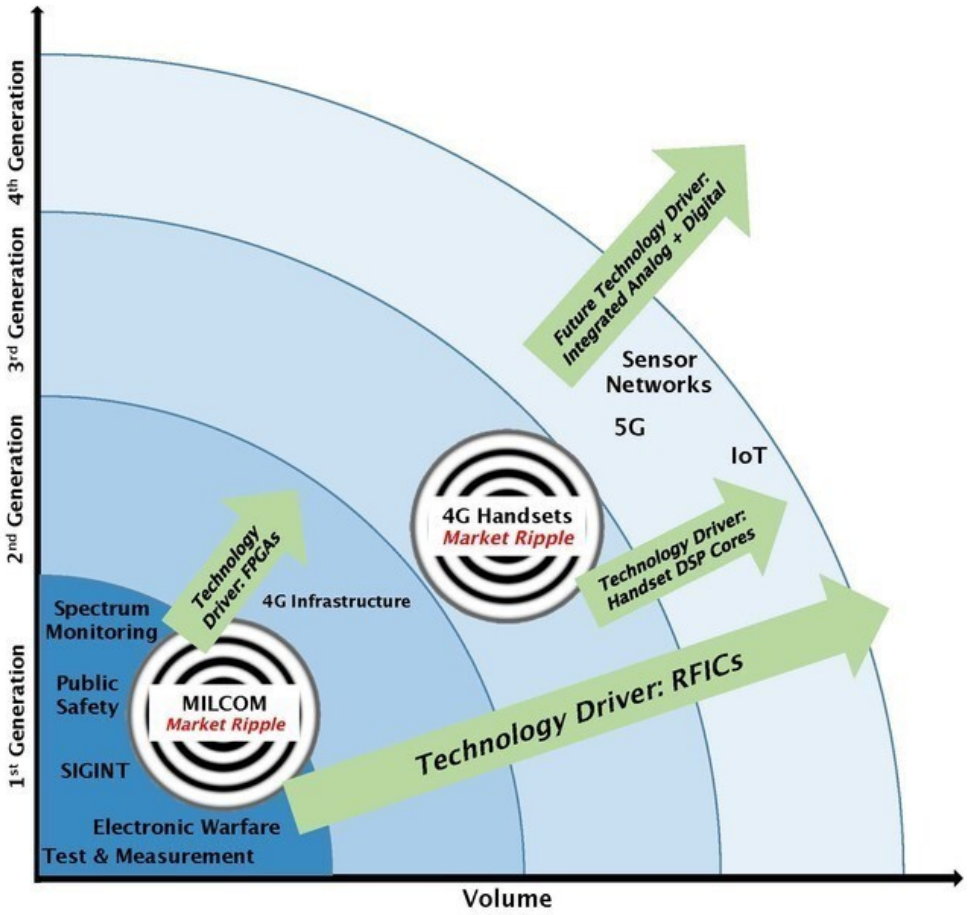

Figure 1.1: Increase in volume of SDRs throughout different generations, showcasing the technologies in each one.

SDR eliminates the need to use resistors and capacitors whilst being able to apply certain filtering algorithms to specific frequencies. The flexibility SDR has is very useful on a large range of applications and designs.

Because of the definition of software of the previously hardware-defined components of the communications system, codifications and modulations can now be also defined with software. This is a major improvement since they were very hard to implement in hardware. With an SDR platform, all these functionalities can be simply changed by changing the software that defines them.

Given the complexity of an SDR platform and its components, it's important to understand its limitations and how its design will impact performance. For it to work properly, it's necessary to use a bandwidth big enough at the front-end for the user to be able to move between different subchannels and to have the possibility of using more than one antenna (for example, in systems

that may have MIMO). Taking into account the codependency between the different layers in the OSI model, it's necessary for an SDR to integrate the physical, link and network layers using a real time protocol in order to have all the functionalities it's meant to have. Interaction with other layers must also be done, but it's done outside the SDR itself. For an even more software-defined communications system, an SDR could be combined with a Software-defined Network (SDN), the SDN being the device that combines higher layers with the layers implemented by the SDR. Finally, an SDR has to be implemented so that the user can do controlled experiments in different environments to demonstrate its fiability, whilst being heavily reconfigurable.

When these criteria are met, SDR platforms become very powerful and useful communication systems because of the functionality they provide, the advantages they bring and the applications where they can be used.

1.1.1 Advantages and inconvenients of Software-Defined Radio

Software-defined radio has many advantages in comparison with a traditional communication system:

- Portability: SDR platforms are the most portable existing communication system since their weight and dimensions are practically negligible. This allows for much quicker testing in various environments since the assembly is much simpler and it also makes it suitable for a wide range of applications discussed in 1.1.2.

- Configurability: The essence of software-defined radio is that all the components can be changed via software. This means all the components traditionally assembled with hardware no longer exist and neither do the problems they carried. Instead of having to change the components manually, they can now be changed using an enhanced software, bringing vast configurability to the communications system. This especially reduces the time spent on testing any functionalities the user wants to implement.
  Configurability is also an advantage in terms of implementation. Usually SDR platforms allow for various transmitters and receivers, enabling MIMO functionalities. The user can configure what transmitters or receivers they use whilst only having to connect antennas to the SDR ports to send/receive information.

- Possibility to move between frequencies. This was one of the main reasons to create SDR platforms when SpeakEasy I was created by the US military. Most SDR platforms have a wide range of frequencies they work on. The AD-FMComms5-EBZ, for example, is tunable from 70 MHz to 6 GHz. Hence, a SDR is suitable for working with many subcarriers and moving between them when necessary. This allows these devices to be used in new technologies such as LTE.

- Robustness: SDR platforms provide reliable and effective communication whilst having all the other advantages.

Yet Software-defined radio also has inconvenients:

- Energy consumption: They consume a considerable amount of energy.
- They are difficult to control. Many specific libraries and toolboxes need to be installed for the SDR to work, especially if the user wants to use the SDR alongside MATLAB or other external tools. Throughout this project, many issues have been faced regarding this matter, and they are thoroughly explained in many sections. For this particular project, the involvement with MATLAB and Linux has been particularly tough to deal with.

1.1.2 Applications for Software-Defined Radio

The advantages explained in 1.1.1 imply that SDR platforms can be used in any communication application and all the phases its development goes through. It's very useful for testing purposes since its portability make it relatively easy to test different environments and its configurability make it even easier because no changes in hardware are needed once it's assembled in the environment. Hence, many environments with different parameters changed by software can be tested. The deployment of any of the applications for which it's tested should be as easy as the testing itself, again, because of its portability. Although SDR platforms could be used for any communications functionality, some general applications where it can be used are:

- Education: SDR platforms can be very useful for educational purposes. Again, they're very portable whilst implementing all the functionalities of a communications system, making them useful in any educational environment. It's especially useful when the lecturer wants to demonstrate a point at a practical level. For example, most of this project could be shown in a lecture as a demonstration for what SDR platforms are capable of.
- LTE: An SDR platform can be used as an eNodeB (which is the node that connects different users), as the core of the network (EPC) or as a User Equipment (UE). When having multiple SDR platforms, a user could simulate a whole LTE network by using each SDR platform as the core network, an eNodeB and a UE, respectively. This also implies LTE networks can be implemented in more applications thanks to portability, such as aerospacial projects.
- Military: It's the application for which SDR was initially created. Since SDR platforms don't weigh a lot and they can be tuned in different frequencies, they are useful for real-time military communication.

1.2 Motivation and Objectives

One of the motivations for doing this project is the innovation the SDR platform implies and the array of possibilities it opens up. The main objective is to understand, evaluate and control the behaviour of the AD-FMComms5-EBZ SDR, an SDR platform developed by Analog Devices, implementing physical layer algorithms to prove its functionality. Another objective for this project is understanding how SDR platforms work in general and what motivates its usage. Furthermore, the use of Software-Defined Radio in lectures could be possible in the future since they're very portable and it would provide students with a more visual and intuitive understanding of certain concepts that could be visualised with SDRs. Knowing this project could be useful for this purpose is also motivational.

Specifically, we aim to cover the following points:

- Documentation
- System description
- Connectivity with MATLAB and installation of the appropriate components.
- Experiments we find interesting.

1.3 Methodology

The methodology for this project is based on the objectives described in 1.2. When evaluating the AD-FMComms5-EBZ SDR by Analog Devices, Mathworks' MATLAB and Simulink will be used to write the code to evaluate the device, although a certain assembly around the SDR will be needed to simulate the code. Use of the anechoic chamber available will be made to test the functionality of the code. Further explanaition about the assembling can be found in chapter 2, whilst all the code is explained in chapters 3 and 4.

From the specific points meant to cover described in 1.2, we'll go through the documentation, a description of the hardware, the structure and the components of the SDR, while the interface provided by Analog Devices will also be described, as well as its functionalities. Furthermore, the interaction with MATLAB and Simulink will be tested and some experiments using the available anecoic chamber will be made.

The research for the AD-FMComms5-EBZ SDR is found in chapter 2 while the general SDR research is found in 1.1.

1.4 Structure of the document

The document is structured to first provide the reader with all the theory related to the project, mainly an introduction to SDR, an explanation of the FMComms5 architecture and a profuse explanation of the AD9361 devices that run it and a description of the interfaces provided by Analog Devices to work with the FMComms5. This occupies chapters 1, 2 and some portions of chapters 3 and 4. The experiments and practical utilities are described in chapters 3 and 4, regarding the IIO Oscilloscope software and MATLAB/Simulink, respectively. Finally, chapter 5 is added to conclude the project and take a look at the future work.

Chapter 2

Hardware Description

The AD-FMComms5 Evaluation Board [Anad] is a general purpose platform suitable for any software-defined radio application. It's tunable accross a wide frequency range (from 70 MHz to 6 GHz) with a channel bandwidth ranging from 200 kHz to 56 MHz. In comparison with other evaluation boards from Analog Devices, such as the AD-FMComms2[Anaa], the AD-FMComms3 [Anab] or the AD-FMComms4 [Anac], the AD-FMComms5 [Anad] has four transmitters and four receivers, allowing for a 4x4 MIMO platform, which maximises throughput and uses the spectrum efficiently. It also has two AD9361 devices, each one controlling two transceivers.

The AD9361 device[Anaf] is a high performance radio frequency (RF) 2x2 transceiver (so it includes two transmitters and two receivers) with integrated Digital-to-Analog Converters (DAC) and Analog-to-Digital Converters (ADC). It supports both Time-Division Duplexing (TDD) and Frequency-Division Duplexing (FDD), it has fully integrated synthesizers in both the receiver and the transmitter and allows for digital filtering. Using the device with an FPGA with the control components the FMComms5 provides allows full SDR capabilities.

2.1 AD9361 Architecture

The basic architecture of the AD9361 device is shown in figure 2.1. This section describes how it works, covering all the elements of the device: the RF and Baseband Phase-Locked Loop (BBPLL) synthesizer, Enable State Machine (ENSM) commands, gain control, Rx and Tx signal paths, filtering, the digital interface and others.

In general terms, the AD9361 integrates all RF, mixed signal and digital blocks necessary to provide all TX functions and allows various baseband processors (BBP) for the device to be interfaced to using low voltage differential signaling interface (LVDS) and be able to transmit data. It has self-calibration and automatic gain control (AGC), which can be set by the user

and includes several test modes.

The receive chain contains all blocks necessary to receive RF signals and convert them to digital data that is usable by a BBP in two independently controlled channels that receive signals from sources. The receiver is a direct conversion system that contains a low-noise amplifier (LNA), matched in-phase (I) and quadrature (Q) amplifiers, mixers and band shaping filters that will be covered later on. The signal path is also composed of two programmable analog low-pass filters, a 12-bit Analog-To-Digital Converter (ADC) and four stages of digital decimating filters.
Gain control is achieved by following a preprogrammed gain index map that distributes gain among the blocks for optimal performance at each level and the BBP makes the gain adjustments as needed.

The transmit chain also consists of two independent channels that provide all digital processing, mixed signal and RF blocks necessary to implement a direct conversion system, and is basically the inverted reception chain.

All these blocks and others are explained in detail in this section.

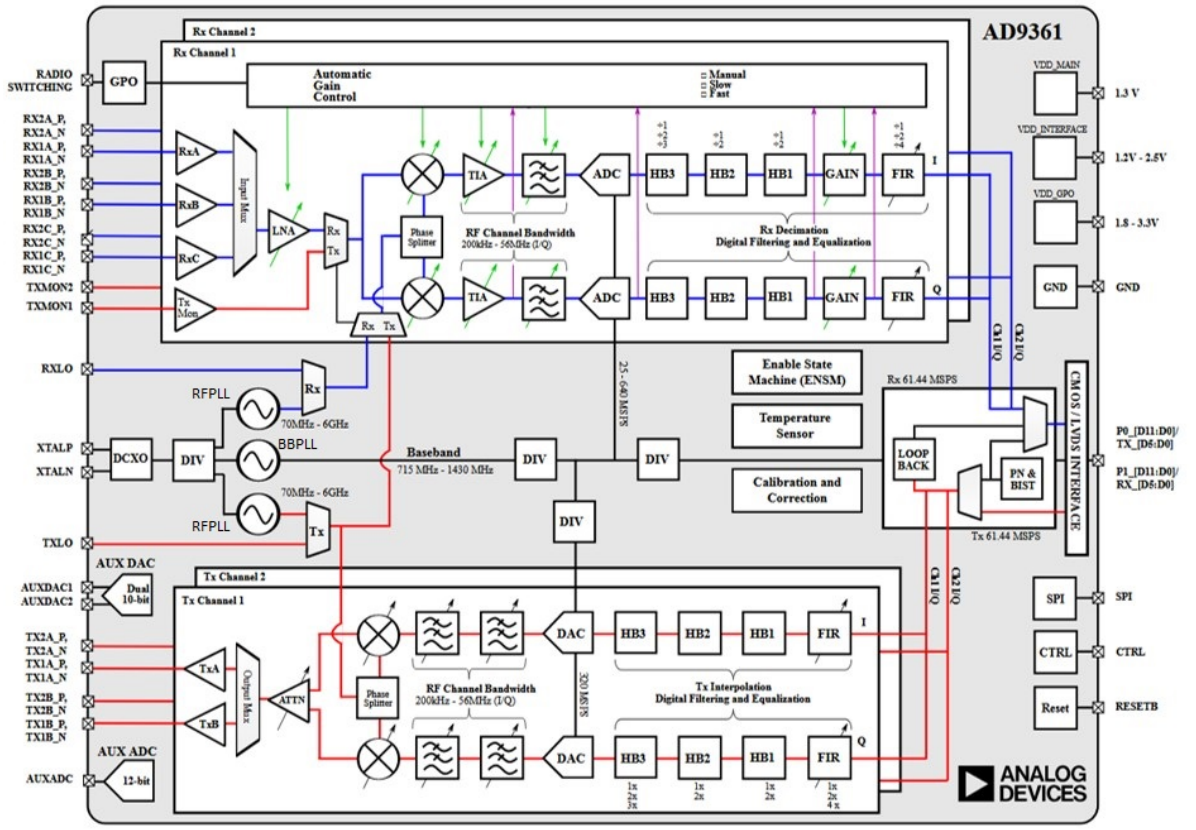

Figure 2.1: AD9361's architecture [Anae]

2.1.1 Clocking, the RFPLL and the BBPLL synthesizers

The AD9361 device uses fractional-N phase locked loops (PLL), which allows frequency resolution that is a fractional portion of the reference frequency, to generate the local oscillator (LO) frequencies for the transmitter and the receiver, as well as to generate the main oscillator. All these PLLs require a reference clock as an input to generate certain frequencies, which can be provided with an external oscillator, as discussed in the previous section, or by a crystal oscillator (XO), which is provided by the FMComms5. Each AD9361 includes its own baseband PLL.

The RFPLL synthesizer is the PLL that generates the LO signals for the Rx channel and for the Tx channel, independently, which should'nt have a similar frequency or the Tx LO may leak into the Rx path. Since each AD9361 has four channels (2 for Rx and 2 for Tx), two RFPLLs are required, and they're identical. The BBPLL synthesizer (baseband PLL) is used to generate all baseband required clocks, for example, to generate the required sampling rate or the DATA_CLK signal, which is further reviewed later. Both have integrated Voltage Controlled Oscillators (VCO) and loop filters, the latter smoothing the input signal to the VCO to not have rapid changes. They generate any convenient reference frequency to be used for any operation. The RFPLLs work from 6 GHz to 12 GHz, which give the frequency range of the AD9361 (from 70 MHz to 6 GHz) by dividing the frequency by either two or four with the VCO divider block, which can be set as is convenient to the user. The BBPLL operates between 700 MHz and 1.4 GHz, from which any sample rate can be generated. The reference clock the BBPLL receives should be between 35 and 70 MHz for optimal performance.

The AD9361 has a mode called "fast lock" which makes it possible to change the frequencies faster by eliminating most of the overhead used by the synthesizer and creating profiles for the Tx and Rx, which makes it more suitable for certain applications. The user can choose whether to use this mode or not but, if used, profiles must be created following a procedure described in the AD9361 Reference Manual [Anag] in page 23.

2.1.2 Tx signal path and filtering

The 8 transmitters the FMComms5 has, 4 for each AD9361 device, all receive data in in-phase/quadrature (I/Q) format from the digital interface of the device. In these channels, the received data goes through four digital interpolating filters (the programmable TX FIR, HB1, HB2 and HB3, as shown in figure 2.2), a 12-bit DAC and two-low pass filters (the BBLPF and the 2nd LPF in figure 2.2) before arriving to the RF mixer.

The digital filters provide the bandwidth limiting required before the DAC and interpolation to translate from the input data rate to the rate needed for the DAC. They are fully configurable

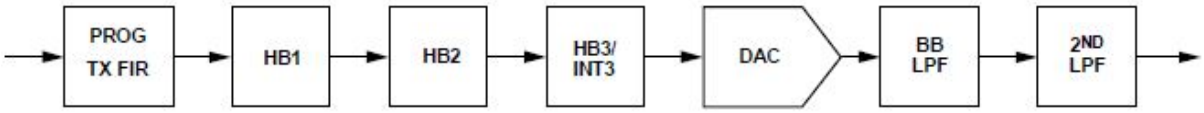

Figure 2.2: Filtering in the transmission signal path [Anag]

by the user.
The first filter the I and Q components of the data encounter is a Finite Impulse Response (FIR) filter. This filter is customizable with a filter wizard in MATLAB. The total number of taps that can be used for this filter depends on the input data rate and the clock of the DAC, since the FIR calculates 16 taps per clock cycle. It also interpolates the data, allowing for more taps if needed.
Then come two half-band interpolating filters (HB1 and HB2) that can interpolate by a factor of 2 and a third one (HB3) by a factor of 2 or 3. Each digital filter adds a delay equivalent to the number of taps of the filter divided by two times the sampling frequency.

After these digital filters and the DAC come two analog filters (BBLPF and 2nd LPF), which reduce spurious outputs by removing sampling artifacts and providing general low-pass filtering. The first filter, the baseband low-pass filter, which is a third-order Butterworth filter[But30] with a programmable 3 dB cutoff frequency, rejects the unwanted frequencies and tries to be as flat as possible to keep the desired ones (where the sent signal is), decreasing -18 dB/octave in gain once in the stopband. In comparison with other similar filters like the Chebyshev, the Butterworth doesn't decrease as quickly in gain but has no ripple, making it better for this application. The second filter is the Secondary Low-pass filter (LPF) which is a single-pole low-pass filter that helps sharpen the decay of the Butterworth filter gain, since it's calibrated at 5 times the bandwidth whilst the Butterworth is calibrated at 1.6 times the BW.
After these filters there are two mixers that transform the I and Q channels into one Rx output that has both in a single I+jQ format. This needs a phase splitter to create a phase of 90^{o} between them (it's the phase difference between real and imaginary components). Finally there's an attenuator that brings the signal back to its initial levels since in the receiver there's an amplifier to be able to work better with the signal, explained in the Rx signal path.

2.1.3 Rx signal path and filtering

The Rx channel follows a similar and inverted path to the Tx channel. First, this data is amplified by a low-noise amplifier (LNA) which has programmable gain (explained in the following section) that grants better control of the received signals by amplifying them without degrading the

Signal-to-Noise Ratio (SNR). Then two mixers are used to convert the signal into its I and Q components, allocating a channel for each, with a phase splitter, as can be seen in figure 2.3.

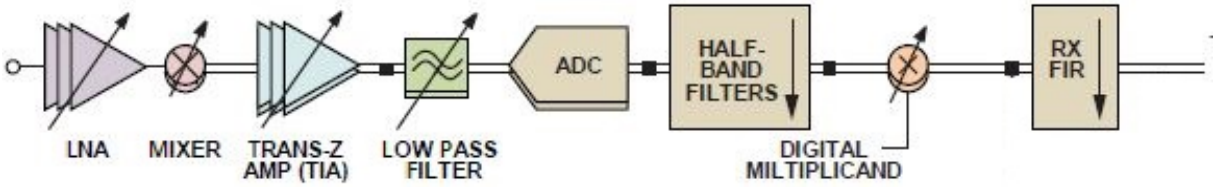

Figure 2.3: General reception signal path [Anag]

The filtering in the Rx signal path is shown in figure 2.4, passing downconverted signals (I and Q) to the baseband receiver section. It has two programmable analog filters (TIA LPF and BBLPF), an ADC and four stages of digital decimating filters (HB3, HB2, HB1 and the programmable RX FIR). As in the transmission path, the I and the Q paths are identical.

Figure 2.4: Filtering in the reception signal path[Anag]

The analog filters are meant to reduce spurious signals by removing mixer products and giving general low-pass filtering. First we find the Trans Impedance Amplifier (TIA) filter, which is a single-pole low-pass filter as in transmission, with the exception that it's calibrated to 2.5 times the baseband channel bandwidth. Then follows another third-order Butterworth filter calibrated at 1.4 times the BW.
The digital filters that follow the ADC are like the ones in the transmission path, although they are meant to decimate.

One can notice in figure 2.1 that each receive path has three LNA inputs (A, B and C) and every transmission path has two Tx outputs (A and B). These LNA inputs and Tx outputs require external impedance matching networs and the LNA devices are functional between 70 MHz and 6 GHz, as is the general device, but they work better below 3 GHz. The two Tx outputs, though, work similarly for all the range of frequencies the board supports. Even so, not all these inputs are available for the FMComms5, since they aren't connected. This can be observed in the FMComms5 schematic [Devd] provided by Analog Devices, where it's shown that only the A and B LNA inputs are available for the receiving path and only A for the transmission path.

2.1.4 Gain control

The AD9361 device has various gain control modes which allow the user to control the gain in a manner that suits the most the application. The gain can be controlled in the receiver signal path, specifically for the LNA, the mixer, the TransImpedance Amplifier (TIA) and the low-pass filters, as is shown in figure 2.5, where the gain-controllable components in the Rx path are crossed with a green arrow. The AD9361 can either be set to Automatic Gain Control (AGC), which has some modes in itself, or Manual Gain Control (MGC), which allows the baseband processor (BBP), and, hence, the user, to control the receiver's gain The AGC evaluates the digital signal level at the input/output ports and then adjusts the gain appropriately, using a table that can be found in the reference guide [Anag]. The BBP can control the gain in two ways: the SPI method or pulsing the control input pins to move the gain indices, which will be discussed later.

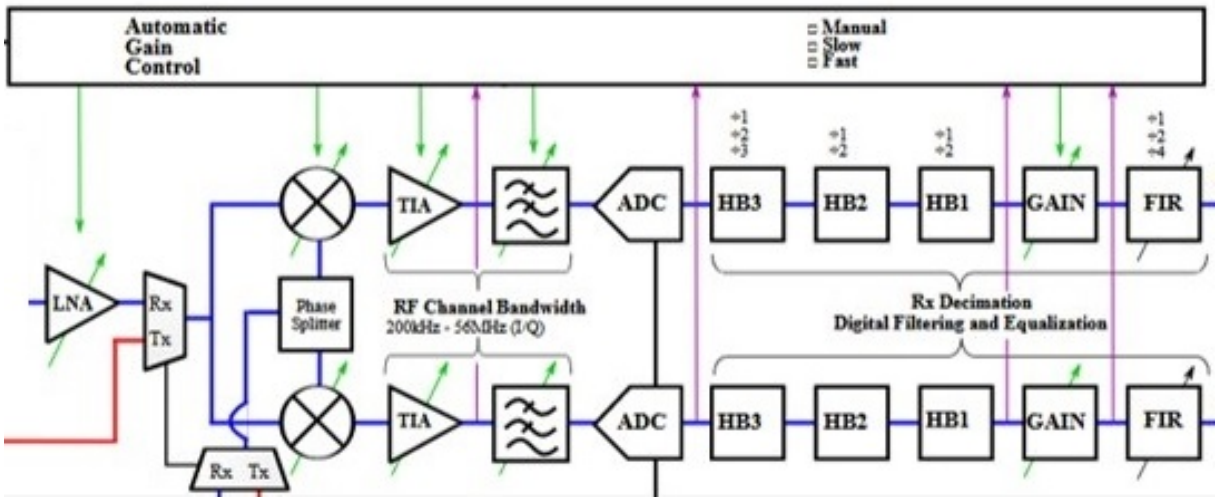

Figure 2.5: Components in the Rx path that are gain-controllable are crossed with a green arrow[Anag].

The AGC has three available modes: slow attack, fast attack and a hybrid mode.

- Slow attack mode: It's intended for slowly changing signals such as those found in Frequency Division Duplexing (FDD) applications. It keeps changing the gain to keep the signal power usable, which is mesured at the receiver.
- Fast attack mode: It's intended for waveforms that turn on and off, usually in TDD applications or GSM FDD applications. The slow attack wouldn't work well with these signals since it would be too slow to adjust the gain, so the fast attack mode reacts very quickly to the changes in power of the signal, while not optimising the gain as well as the slow attack mode.

- Hybrid mode: It's the same as the slow attack but the BBP takes some control, so it's not fully automatic.

These automatic modes and the manual one can be enabled with the IIO Oscilloscope application, with MATLAB or even with Linux commands. The slow attack and the fast attack work fine although we ended up using manual control and setting a gain ourselves. The automatic mode is also a good indicator as to whether the board is actually receiving a signal or not, since the gain that will generate will have to be very big, so when it does, you know it's not working properly.

2.1.5 Power control

There are a few ways to control the power of the transmitters and the receivers of the AD9361 device. The device allows the user to know the Received Signal Strength Indicator (RSSI) of all the receivers in the system. The AD9361 measures the RSSI by measuring the power level and compensating for the reception gain, for both FDD and TDD applications. As the receive path changes, the gain calculated by the algorithm may differ from the real one, causing some error in the RSSI value, which is tipically around 2 dB. For the RSSI algorithm to be more accurate, there are calibration possibilities, although they won't be needed for this project.

There's also a way to control the transmission power. The device uses a method of transmit power control (TPC) that has minimum interaction with the BBP. It has a single 9-bit parameter that controls the attenuation of a transmission path which can be controled via Linux commands.

Finally, the AD9361 has a transmitter power monitor (TPM) only available in TDD mode. It works with the two TPM inputs the device has: the TX_MON1 and the TX_MON2. Since the Tx monitor signal is quite higher with respect to the Rx signal, it's multiplexed into the receiver chain after the LNA. It allows us to know the RSSI values for the Tx by using the circuitry that calculates the Rx RSSI values when it's not being used, by multiplexing the power detector into the receive path. The circuitry is turned on during the transmission of the signal (transmission and reception can't work at the same time, so the reception circuitry is free to be used when transmitting), getting the RSSI for the transmission. Enable State Machine (ENSM) plays a crutial role during this process, as will be explained later. The TX_MON inputs require a matching network as the input.

2.1.6 Digital Interface: CMOS and LVDS

The transfer of data and control information between the AD9361 and the baseband processor (BBP) is done via the serial peripheral interface (SPI) and the parallel data ports. The BBP

can send various signals to the AD9361, some of them being bidirectional so the device can also interact with the BBP. The most important signals are:

- SYNC_IN: It's the signal that the BBP processor sends to the two AD9361 devices to synchronise them using the same reference clock, usually the clock provided by the FM-Comms5.
- DATA_CLK: This signal is sent by the device to the BBP as a clock for the RX data path. The BBP uses it as a timing reference for data transfers and baseband data processing.
- FB_CLK: It's a feedback version of the DATA_CLK sent by the BBP to the device which allows synchronization with bursting control signals like ENABLE or TXNRX.
- P0_D and P1_D: Bidirectional buses for ports 0 and 1 that transfer data between the BBP and the device.
- ENABLE and TXNRX: They are control signals.

The data interface has two operational modes: the Complementary Metal-Oxide-Semiconductor (CMOS) compatible mode and the low-voltage differential signal (LVDS) compatible mode. Both use the signals mentioned above but only CMOS allows single port half duplex mode, single port full duplex mode, dual port half duplex mode and dual port full duplex mode, which are different data transfering modes used on different applications. LVDS only supports dual port full duplex mode and both modes support FDD and TDD operations, a representation of which is shown in figure 2.6.

The basic difference between these two modes is the way they handle the signals, but their functionality is pretty much the same. The maximum data rate is the same for both (61.44 MSPS) and so is the BW (56 MHz).

2.1.7 Programming with ENSM states, SPI and ENABLE/TXNRX

The AD9361 device includes an enable state machine (ENSM) which allows real time control over the state of the device. The state can either be controled via the Serial Peripheral Interface (SPI) or with ENABLE/TXNRX pin control. SPI control is recommended when real time control of the synthesizers is not necessary, which is when the ENABLE/TXNRX control signals are useful. The idea is that ENSM has state values, shown in table 2.1, and SPI or ENABLE/TXNRX change the state as the user requests.

The ENSM states are automatically set by the BBP, but the user can create a set of orders to control them if necessary. The user can also just change whether it automatically uses SPI or ENABLE/TXNRX, depending on the application used.

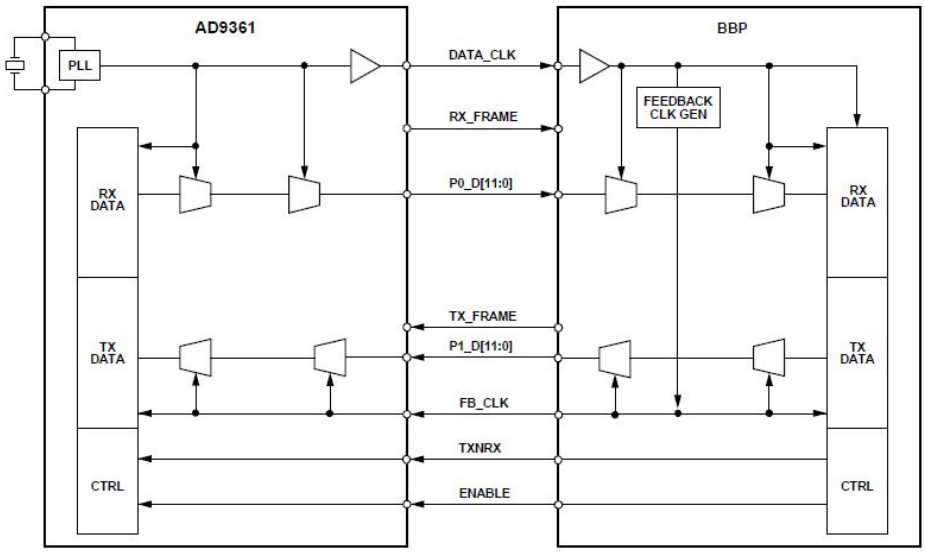

Figure 2.6: Dual Port Full Dupplex mode for the data interface between the AD9361 and the BBP, where both CMOS and LVDS are compatible [Anag].

ENSM State	ENSM State Value	Description
SLEEP	0	Everything is disabled
WAIT	0	Disables the synthesizers
CALIBRATION	1,2,3	Used to calibrate certain components
ALERT	5	Enables the synthesizers
Tx	6	Enables the Tx path
Tx FLUSH	7	Sets Tx filters to 0
Rx	8	Enables the Rx path
Rx FLUSH	9	Sets Rx filters to 0
FDD	10	Enables FDD
FDD FLUSH	11	Sets all filters to 0 in FDD mode

Table 2.1: ENSM states and description[Anag]

The SPI control method works by writing SPI registers to advance the current state to the next state, and it's controled asynchronously by the BBP using the DATA_CLK signal as the reference clock. The ENABLE/TXNRX method uses the FB_CLK instead, and the ENABLE signal is set to 1 during a FB_CLK cycle for the ENSM to advance to the next programmed state. The TXNRX signal is 0 when using the Rx chain and 1 for the Tx chain, following the FB_CLK too. The ENABLE signal is kept to 1 during all the cycles that the FDD state is

used. These signals are sent from the BBP to the AD9361, the BBP essentially controls all the ENSM states changes, whether it's using SPI or ENABLE/TXNRX. It's the AD9361 that handles the received pulses (in the ENABLE/TXNRX case) to change the ENSM states, while the SPI registers are saved in the device directly.

2.2 AD-FMComms5-EBZ SDR Architecture

The AD-FMComms5 Evaluation Board (shown in figure 2.7) consists, basically, of the two AD9361 chips, the input and output ports for the transmitters, receivers and other necessary connections, an interface for all these ports, and connectors to be able to use it with a Field-Programmable Gate Array (FPGA). In comparison with the other evaluation boards from Analog Devices, the main purpose of this board is to provide a platform to connect and synchronise the AD9361 chips, for them to be used in MIMO applications. The FMComms5 also includes other components to control the device. In itself, the AD9361 chips do all the work and provide for the usual SDR capabilities, such as configuration, transmission and reception of signals or all the data interface.

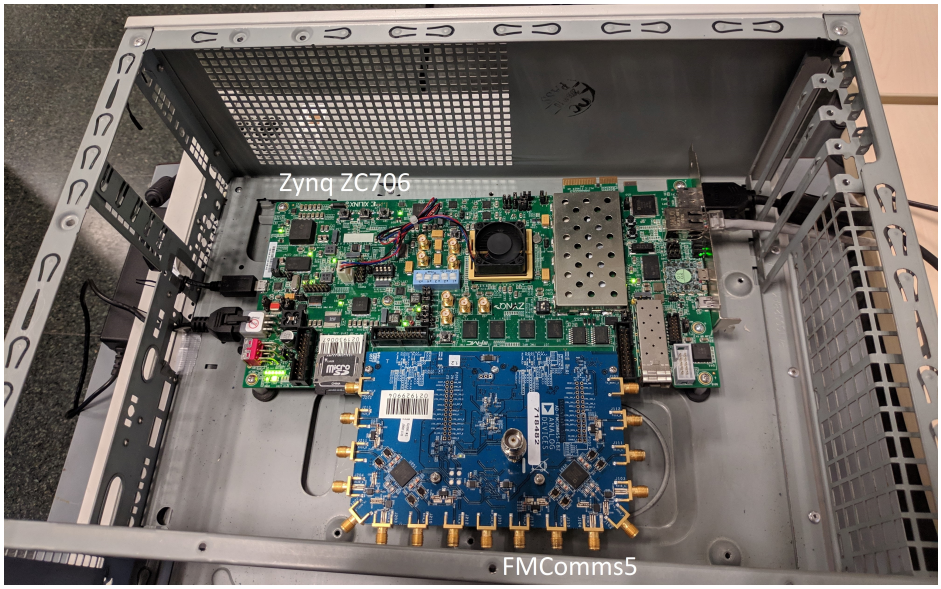

Figure 2.7: Picture of the FMComms5 board connected with the ZC706 Zynq

For this matter, how the connectivity of the FPGA is accomplished, how the AD9361 chips are used and its signals processed and controlled, and how the FMComms5 achieves overall SDR capabilities are explained in this section.

2.2.1 Connectivity with an FPGA

The AD-FMComms5 Evaluation Board is a Field-Programmable Gate Array (FPGA) Mezzanine Card (FMC) board for the AD9361. FMC is an ANSI/VITA standard that defines how the input and output between the daughterboard and an FPGA (in this case, the FMComms5 and the Xilinx Zynq ZC706 [Xil], respectively) are designed. For this matter, the FMComms5 board has two adjacent FMC connectors with Low-Pin Count (LPC) used to interact with the Zynq ZC706, which has its own FMC-LPC connector.

The Zynq ZC706 provides a hardware environment for developing and evaluating the FMComms5 board. It provides component memory, a four lane PCIe interface, an Ethernet PHY, general purpose input and output and two UART interfaces. Basically, it provides all the power, memory and processing capacity the AD9361 needs while also enabling an Ethernet connection and an HDMI video output to use the IIO Oscilloscope application.

In figure 2.8, a schematic of the system (the evaluation board connected with the Zynq) is shown. Although the schematic is for the FMComms2, instead of the FMComms5, the schematic would be the same for the FMComms5 but, instead of only one AD9361 chip, it would have two. In this figure one can see that the FMComms5 is merely an FMC connector for the AD9361 chips with some control functions over it, while it's the Zynq that provides all the power, processing and memory necessities.

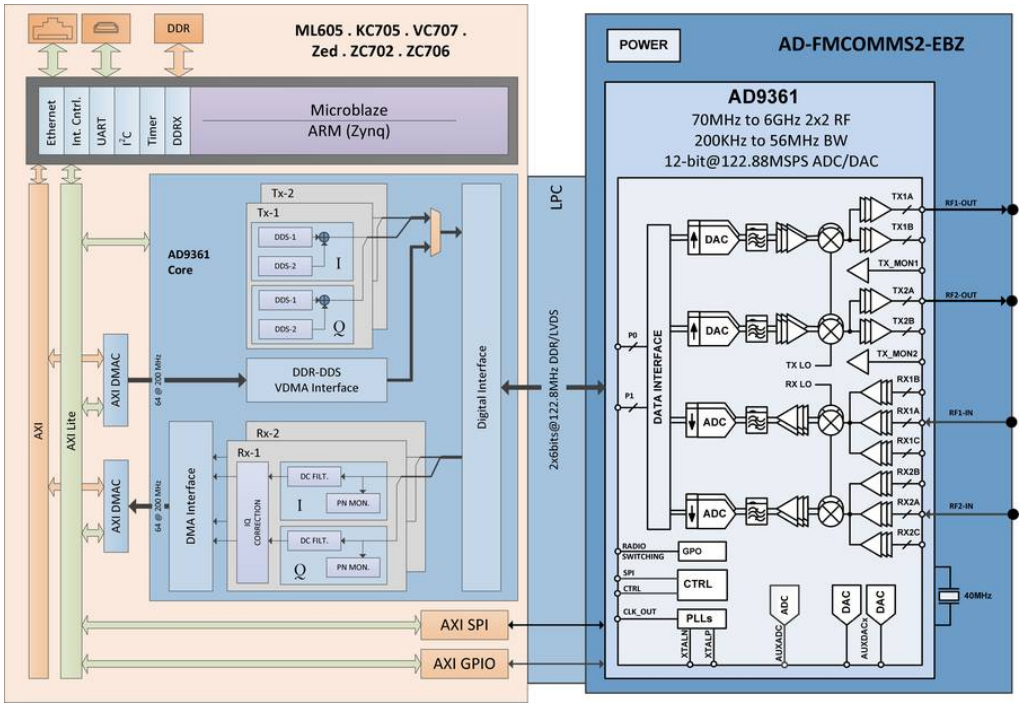

Figure 2.8: Schematic of the FMComms2 connected via FMC with the Zynq ZC706 FPGA. [Anaa]

2.2.2 Connectivity with the AD9361

The FMComms5 has a total of 17 SMA input ports, eight of which are for RX, four for TX, four for TX monitoring and one as the reference clock. It also has an SMA connector to control the evaluation board with an external local oscillator, which can be used as a reference clock, even though the FMComms5 already contains a reference clock (a crystal oscillator, which is an electronic oscillator circuit that uses the mechanical resonance of a vibrating crystal of piezoelectric material to create an electrical signal with a precise frequency) which works at 40 MHz. The output of the clock is distributed to the ADCLK846 [Anah], which distributes the clock to the two AD9361 devices, as well as to the FMC connector and to the ADF5355, which also generates LO signals for the AD9361 device. The TX monitoring inputs are meant to be used to detect the TX power. Neither the TX monitoring inputs nor the reference clock are strictly necessary, since the FMComms5 already has its own reference clock and monitoring mechanisms. The AD9361 accurately measures the level of a received signal in a Received Signal Strength Indicator (RSSI), the process used to obtain it being explained before in section 2.1. As mentioned before, the FMComms5 has two AD9361 devices, each having two transceivers. This begs the question as to how the evaluation board couples with the usage of the two chips at the same time and synchronises them to be used in MIMO applications. The internal reference clock is used to synchronise the two AD9361 devices, which is a feature that the user would want to control if necessary. The device not only uses Phase-Locked Loops (PLL) to generate LO frequencies for the RF paths, but also to generate the oscillator used for the data converters, digital filters and I/O ports. The external oscillator comes in handy for these PLLs, since they require a reference clock to work. Since the FMComms5 is software-configurable, the usage of an external reference clock or of the internal clock can simply be switched by the user thanks to the IIO library made by Analog Devices explained in 2.3.

The FMComms5 has a baseband processor that helps syncing the two AD9361 devices along with the common reference oscillator the board provides. Each AD9361 includes its own baseband PLL (as was explained in 2.1), which generates the internal sampling and data clocks from the given reference clock. A logical syncing input pulse is needed to align the device's own clock to the reference clock, which is sent by the baseband processor, which sychronizes the two devices. These components are shown in figure 2.9.

2.3 Libiio

This is a short introduction to Libiio, which is the basis for both the IIO Oscilloscope and the MATLAB/Simulink interaction with the board, explained in chapters 3 and 4, respectively, and is further explained in these chapters in relation with the applications discussed in each one.

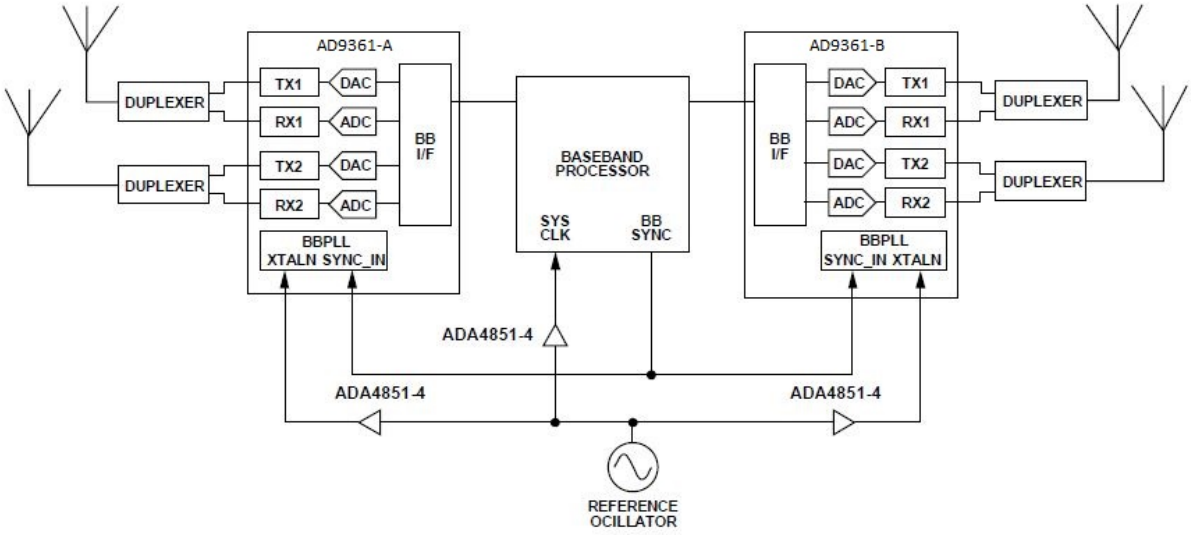

Figure 2.9: Main syncing scheme for the AD9361 devices. [Anag]

Libiio is a library developed by Analog Devices to ease the development of software interfacing Linux Industrial I/O (IIO) devices. IIO is a framework in the linux kernel first introduced in 2009 and is useful because it allows the sharing of an infrastructure, provides fast and efficient data transport and simplifies the management of devices for the user. This section is included in this chapter to explain the various devices of the FMComms5 since they can be seen in the interface that libiio provides. The basic components of the libiio interface are the iio_context, iio_device, iio_channel and iio_buffer classes, which follow a heriarchical order in the manner described in figure 2.10. For our SDR, the iio_context is automatically created by the library and it would represent the FMComms5 in itself. The devices, channels and attributes are described in the listing below. The attributes describe hardware capabilities and they can be changed, and the channels are the actual physical channels in the system. Finally, the buffers are used for continuous data transmission and are binary to a device, they can either have one or have none. Specifically, the FMComms5 uses Direct Memory Acces (DMA) buffers.

Using this library is useful considering all the parameters shown can be changed by the user, enabling one of SDR's principal advantages: configurability.

When inspecting the libiio files, we can observe the devices that it detects when it's connected to the evaluation board. Each device has a number of channels which have attributes that can be changed manually with Linux using certain commands, and each device has its own general attributes. The devices we see are the following:

- ad7291: It's a chip used for voltage monitoring, which has 9 channels, all attributed to certain measured voltages.

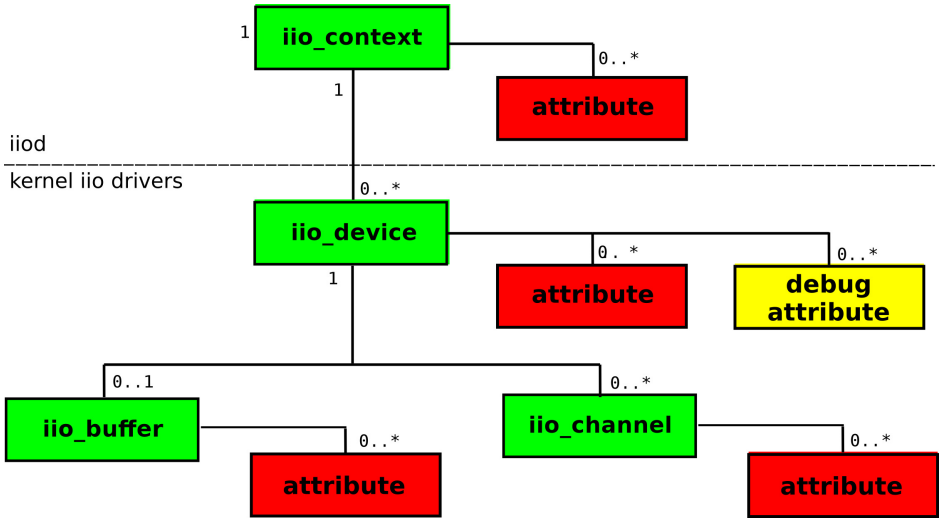

Figure 2.10: Heriarchical distribution of libiio components [GNU19].

- ad9361-phy: This device represents one of the AD9361 chips and it includes valuable information related to it. It uses 11 channels, which were further explained in section 2.1, each of them having certain attributes (as the sample rate or bandwidth) that the user can change.
- ad9361-phy-B: It has the same information as the previous device, but specific to the second AD9361 device.
- xadc: It's a device specific to the Xilinx ZC706 Zynq FPGA that monitors alimentation voltages related to the FPGA.
- cf-ad9361-dds-core-lpc: The "cf" prefix means "core FPGA", so it's a device related to the Zynq FPGA. Specifically, it's related to the receive DMA block of the Zynq; it manages the processing of the reception paths from the AD9361 device and its memory allocation.
- cf-ad9361-dds-core-lpc-B: It's identical as the previous device but for the second AD9361 chip.
- cf-ad9361-A and cf-ad9361-B: Devices related to the Zynq's processing in the TX DMA block for each AD9361 device, respectively.

2.3.1 Linux commands

There are three possible ways to change the value of the attributes of each channel or device: with Linux commands, with the IIO Oscilloscope (chapter 3) or with MATLAB/Simulink (chapter 4).
Libiio provides an interface for all the channels and their attributes of the FMComms5's devices.

There are two kinds of attributes: a channel attribute and a device attribute. On the one hand, a channel attribute belongs to a channel which, in itself, belongs to a device. On the other hand, a device attribute pertains to the whole device and all of its channels.

For example, among many other attributes, channels and devices, the user can modify the sampling frequency (attribute) of the ad9361 (device) in the receiving path (channel), which is a channel-belonging attribute, but the user can also change the value of the ensm_mode (attribute) of the ad9361 (device), which is common for all the channels and is, hence, a device-belonging attribute. All the attributes, channels and devices that are configurable are included in Appendix A.

To change the value of these attributes with basic Linux commands, libiio must have been set up beforehand so that the user can see all the configurability options in its Linux platform. Libiio organises its components in folders, so that the user can access the different devices and channels by moving through the libiio folders (each device is in a folder and each channel in a subfolder).

Each device is contained in the `/sys/bus/iio/devices/iio:device` folder. Once in the desired folder/subfolder, the user can see all the attributes related to that device or channel by listing all the items in the folder/subfolder (with the `ls` command). To see the value of one of these attributes, the `cat` command can be used and to actually change it, the `echo` command is used.

This allows some configurability of the FMComms5, although it only allows the user to change the already-defined parameters, yet the user can't actually choose what to use the FMComms5 for (unless bash scripts are used). The IIO Oscilloscope and MATLAB/Simulink provide this extra layer that allows full configurability and usability of the FMComms5 SDR. Although the three configurability methods use libiio and are used for the same purpose, the user can't rely on them interacting together, since changing an attribute with Linux commands doesn't necessarily work well with MATLAB, where this attribute may not have changed. Hence, it's more efficient to just use one of these platforms and not use them simultaneously, and just change all these attributes in the same place.

Chapter 3

Interaction with the SDR platform: IIO Oscilloscope

This chapter covers the explanation of the IIO environment, working with the IIO Oscilloscope application and an evaluation of possibilities of this application on an educational scope. A first experiment about synchronising the four transmitters will be designed and carried out using the IIO Oscilloscope, evaluating the capabilities of the application.

3.1 Description of the environment

Libiio, the library by Analog Devices on which the IIO Oscilloscope application is built, was already introduced in section 2.3 to show the devices of the FMComms5 and a general vision of the capacities of this library. Any parameter of the board was shown to be configurable using some simple Linux commands when libiio is enabled, but the complexity is bigger when the user wants to generate signals to transmit with the FMComms5. Although one could do this with bash scripts, the IIO Oscilloscope application provides an easier way.

With the IIO Oscilloscope application, the user can change any of the parameters that could be changed with Linux commands, but they're sorted in a different way. Instead of using libiio's components, which sorted the attributes according to the channel and the device they belonged to, the IIO Oscilloscope sorts them in four categories: AD9361 Global Settings, AD9361 Receive Chain, AD9361 Transmit Chain and FPGA settings. The configurable attributes of each one are shown in section 3.1.1. Even though the order of the attributes is different to libiio's hierarchical order (shown in figure 2.10), the IIO Oscilloscope application still relies on libiio, and is built using this library. Hence, to install the IIO Oscilloscope into a Linux platform, libiio must be installed for the application to work, and the user should test beforehand that libiio detects

the FMComms5 and its devices. In conclusion, regarding the configurable attributes of the FMComms5, the IIO Oscilloscope simply sorts them in a more intuitive way and with a simpler interface. Linux commands are no longer needed, and the value of the attributes can be set with this application.

As mentioned before, though, the IIO Oscilloscope is not only useful to configure certain attributes, but also to generate signals to transmit through the FMComms5, to set the FIR filter in any way the user desires and to plot (in real time) the reiceved signals. Hence, it provides the full configurability that SDRs are advantageous for. Signal generation and FIR filter configurations are mixed with the configuration of the FMComms5's attributes configuration in the same interface, explained in section 3.1.1, whilst the IIO Oscilloscope has a separate window for plotting, that allows plotting of the data in time domain, frequency domain and as a constellation, and adds configurability to these options. The application also has a debugging window that helps the user debug any device of the FMComms5 and its attributes. The plotting window is further explained in section 3.2.1 and the debugging window in section 3.1.1.

A first experiment will be explained in section 3.2, where we'll try to synchronise the four transmitters for them to be used at the same time, trying to accomplish efficient data transmission. Firstly, though, a description of all the categories and attributes configurable with the IIO Oscilloscope is done (in section 3.1.1) showing their correlation with some of the concepts explained in chapter 2.

3.1.1 Description of the interface

The IIO Oscilloscope GUI application has an interface that allows configuration of all the capabilities of the board, distributed in four different categories: AD9361 Global Settings, AD9361 Receive Chain, AD9361 Transmit Chain and FPGA settings.
The AD9361 Global Settings contains the following components (shown in figure 3.1):

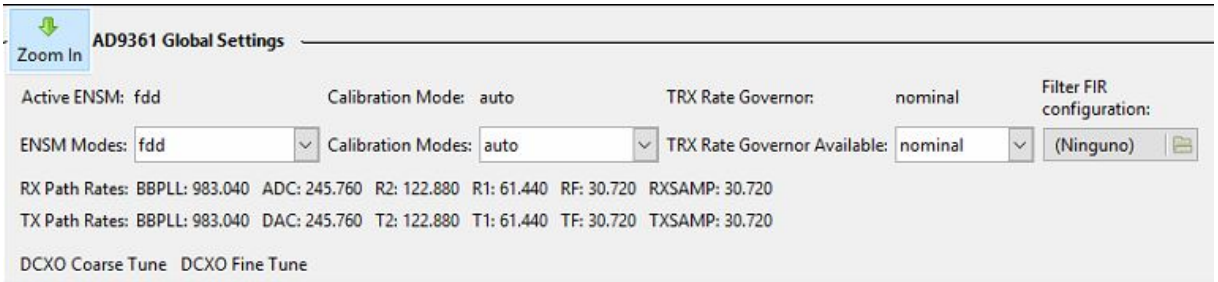

Figure 3.1: "AD9361 Global Settings" configuration category in the IIO Oscilloscope application

- ENSM Modes: Allows the user to choose between FDD and TDD mode. For the purposes of this project, this setting must always be set to FDD mode. There's also a parameter named Active ENSM which shows the current ENSM mode being used.
- Calibration mode: It should always be set to automatic. The AD9361 device already has good calibration algorithms, heavily explained in the ad9361 guide [Anag], which are not covered in this project.
- TRX Rate Governor Available: This parameter is used to configure the decimation and interpolation rates that the digital filters in the reception path and the transmission path have, respectively. These can either be set to a factor of 2 or 3 and they influence the sample rate of the ADC. It can either be set to 'nominal' or 'highest_osr', the last one providing the highest over-sampling rate. We set this parameter to 'nominal'.
- Filter FIR configuration: Allows the user to use a created FIR Filter. Its creation is explained in chapter 4. This filter is shown in figures 2.2 and 2.4 as the programmable transmission filter (Prog TX Filter) and the programmable receiving filter (Prog RX Filter), respectively.

In this category, rates of certain elements like the BBPLL or the ADC/DAC are also displayed. These rates, shown for the reception and transmission path, are all multiples of the sample rate that was set in the receive and transmit chains (which was of 30.72 MSPS). This sample rate, as explained in chapter 2, is created with a driving clock, and is then multiplied by a factor of 2 or 4 to generate the equivalent sample rates for the other components, which are the ones displayed in this category.

The AD9361 Receive Chain control category includes the configurable parameters listed below and shown in figure 3.2

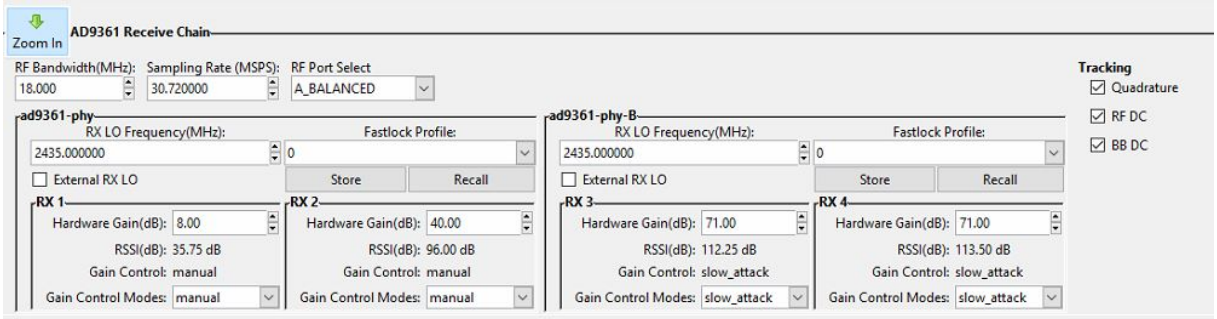

Figure 3.2: "AD9361 Receive Chain" configuration category in the IIO Oscilloscope application

- RF Bandwidth: Sets the bandwidth for the whole chain. In our project, it is 18 MHz. The device configures it via the filters in its reception and transmission paths.

- Sampling rate: We use 30.72 MSPS since it's the maximum the SDR allows.
- RF Port Select: The options for the AD9361 RF port are A, B or C. As shown in the FMComms5 schematic [Devd], not all these ports are connected in the FMComms5. For the reception, only A and B are connected, but C is not, so either one of the two options is viable. We used 'A_BALANCED'.
- RX LO Frequency: Sets the frequency at which the device works. The user should bear in mind that the transmission and reception frequency should be within the range of each other, so they can be different within the range of the bandwidth. Otherwise the receiver can't process the transmitted signals. In our project we set this to around 2.4 GHz.
- Hardware gain: This parameter can set the gain for each receiver independently, depending on the chosen gain control mode.
- Gain control mode: This allows the user to choose how the AD9361 controls the gain in reception. The available options are the ones described in section 2.1.4. We chose the slow_attack mode since it control the receive gain effectively depending on the received signal. The RSSI (explained in section 2.1.5)is also shown in this category in dB.

The parameters in the AD9361 Transmission Chain are shown below and in figure 3.3, most of them being similar to the ones in the reception chain:

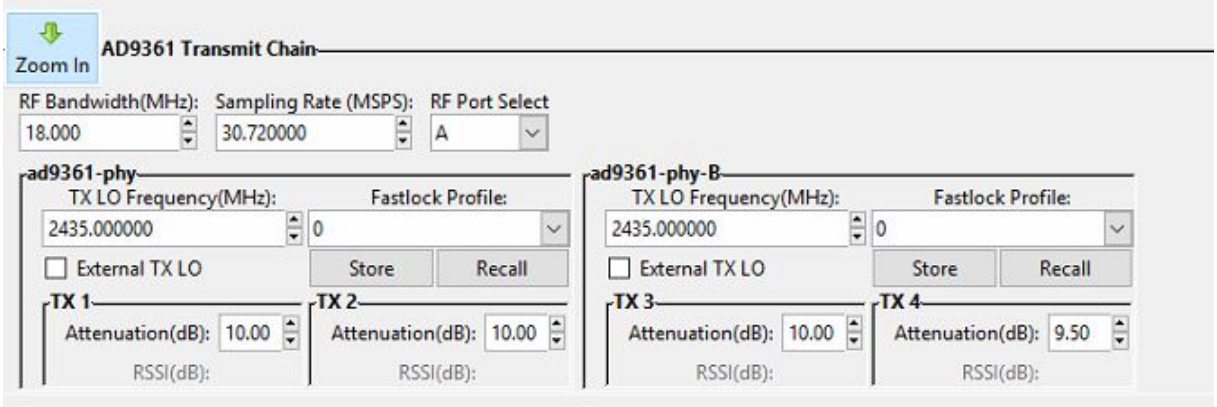

Figure 3.3: "AD9361 Transmit Chain" configuration category in the IIO Oscilloscope application

- RF Bandwidth: It's also 18 MHz in the transmission path.
- Sampling rate: The transmission chain also doesn't allow a sampling rate higher than 30.72 MSPS, so the sampling rate doesn't change either from the RX chain.
- RF Port Select: The options for the AD9361 RF port are A or B. As shown in the FMComms5 schematic [Devd], only port A is available in transmission, so the chosen option is "A".

- TX LO Frequency: Sets the frequency at which the device works. In our project we set this to around 2.4 GHz.
- Hardware attenuation: This parameter can set the gain for each transmitter independently.
- Fastlock Profile: Finally, there's the option to load a previously made profile (explained in 2.1.1) which won't be used in this project.

Finally, there are the parameters related to the FPGA, shown in figure 3.4, which allow configurability on signal generation.

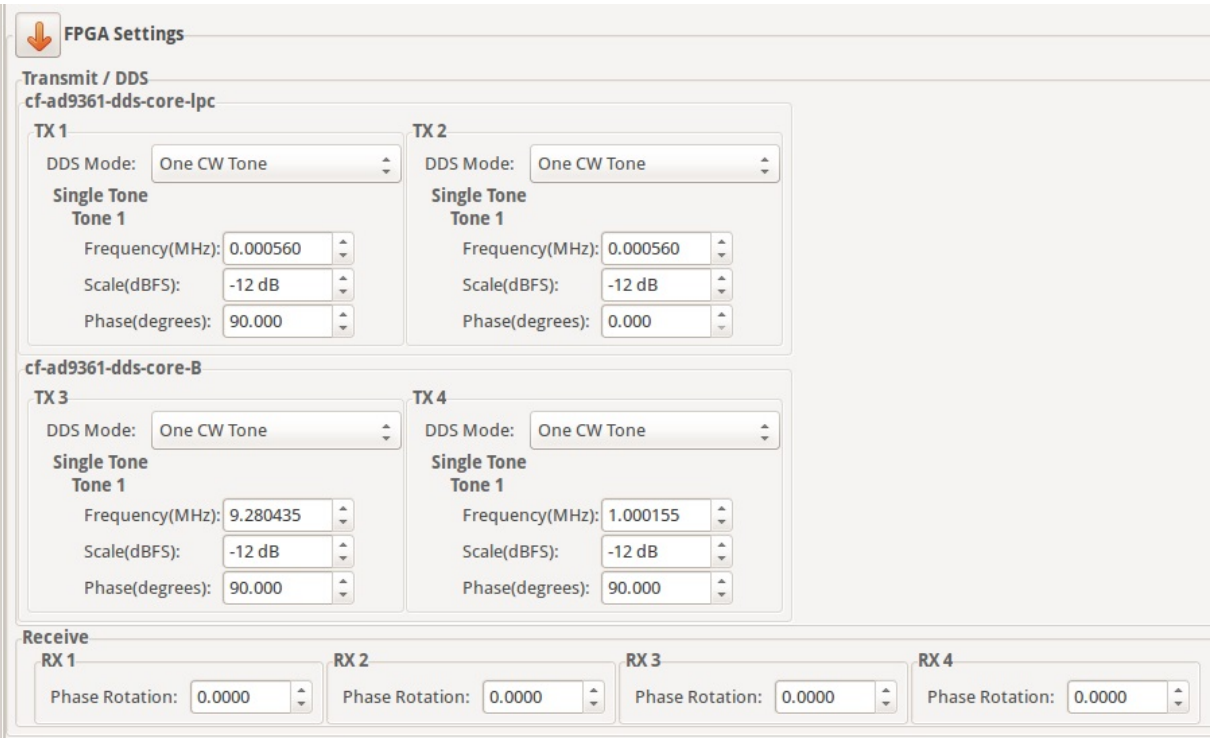

Figure 3.4: "FPGA Settings" configuration category in the IIO Oscilloscope application

The user can choose the Direct Digital Synthesizer (DDS) Mode, which chooses how DDS creates a waveform from the fixed-frequency reference clock, and can also choose the frequency of the waveform, its scale and its phase, for each transmitter.
This configurability is available for each transmitter, independently, so that a different signal can be sent through each transmitter, with a different frequency, scale and phase. There are four different DDS modes:

- One CW Tone: Sends one Continuous Wave tone through the selected transmitter. The frequency of the tone and the scale are configurable.
- Two CW tones: Sends two Continuous Wave tones, both scale and frequency configurable.

- Independent I/Q Control: Allows independent configuration of up to two tones through the I and Q components channels. The configurable options are frequency, scale and phase.

- DAC Buffer Output: This is the most interesting option, since it allows the user to input a file with the values of the amplitude of the wave to be transmitted, and is configurable for each transmitter independently. This option is the one that will be used in the following section to test and synchronise the four transmitters so that they can be used at the same time. This option widens the signal generation capability of the IIO Oscilloscope since virtually any signal can be generated as a text file and, thus, can be sent with the FMComms5.

There are some other control functions in the advanced tab, which provide further control over some of the parameters discussed previously.
For ENSM, TXNRX control can be enabled (explained in 2.1.7), among other options. The clocks can be individually managed by enabling/disabling them, which allows better control of the external LO (which isn't used in this project) by being able to use it only for reception or transmission, and also allows the option to disable the external LO and only use the XO provided by the FMComms5, which is suitable in our case.
Among other options, the gain mode can be further configured by choosing how, for example, the slow attack mode functions and the thresholds it uses to decide the gain.

The debugging mode shows the low-level registers of a certain device, and also allows read/writing of device attributes in a debugging fashion, shown in figure 3.5.

3.2 Synchronising the four transmitters

As mentioned before, the FPGA Settings category in the IIO Oscilloscope application allows the transmission of any desired signal. The only requirement is that this signal is inputted as a text file.

With this feature, any signal can be created and transmitted through the FMComms5. The text file must be written by columns, each column representing one component (I or Q) of one transmitter (in order), and each value of each column must be the amplitude of the signal for that sample. Although the file can go on for as many samples as desired, this implementation doesn't seem very optimal for very large data, but it serves the purpose of this experiment. In our case, we sent 768 samples (256+128x4) through each transmitter, so the text file is a

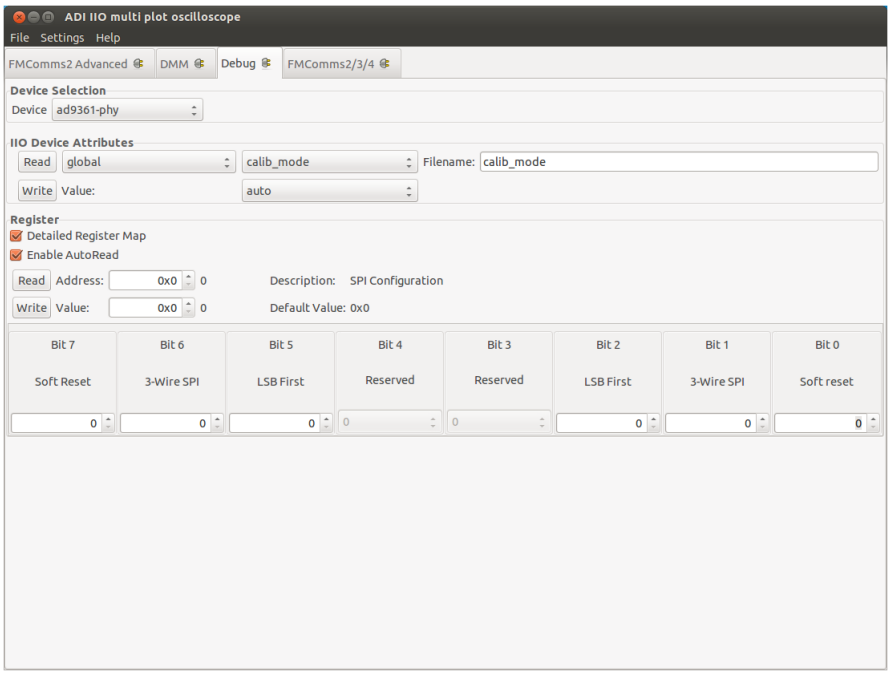

Figure 3.5: "Debug" configuration category in the IIO Oscilloscope application

matrix of 768x8 values.

The purpose of this experiment is to monitor if all the transmitters are synchronised and can be used at the same time. Multiple Input Multiple Output (MIMO) systems are very efficient because the same signal is sent through various transmitters. When the receiver gets this signal, it will have a higher amplitude (the sum of the amplitude of the signal in each transmitter) and, hence, the system will be more efficient. If these transmitters aren't synchronised, though, this efficiency gain is lost and the FMComms5 won't serve the purpose of a MIMO system. Hence, we found that the experiment serves (partly) to the objective of evaluating and understanding the FMComms5 (in this case, evaluating if it serves the purposes of MIMO).

To carry out this experiment, we send QPSK-modulated symbols through each transmitter and through all the transmitters at the same time, separetely. If the transmitters are synchronised, the amplitude of the received constellation will be the sum of the amplitude of each transmitter but, if they aren't, the asynchronisation may cause destructive interference and the received signal may end up being even worse that if only one transmitter was used. The constellation can be seen using the plotting abilities of the IIO Oscilloscope explained in section 3.2.1. To generate the QPSK-modulated symbols, we must take into account the I and Q amplitude values to generate such symbols, described by the ETSI 136.211 standard and shown in table 3.1.

Through each transmitter, we send 128 samples of each symbol and 256 initial zeros, which are sent to have a reference point in the received samples and to know if the samples are received in the same order they were sent, giving a total of 768 samples. This number of samples will be enough to see whether the synchronisation between transmitters is correct or not.

Bits	I	Q
00	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$
01	$\frac{1}{\sqrt{2}}$	$-\frac{1}{\sqrt{2}}$
10	$-\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$
11	$-\frac{1}{\sqrt{2}}$	$-\frac{1}{\sqrt{2}}$

Table 3.1: I and Q QPSK symbol amplitudes following ETSI 136.211 v12.09 standard [Eur]

To suit these amplitude values for the IIO Oscilloscope's range, we multiply these values for a constant so that they're higher and meet the oscilloscope's range.

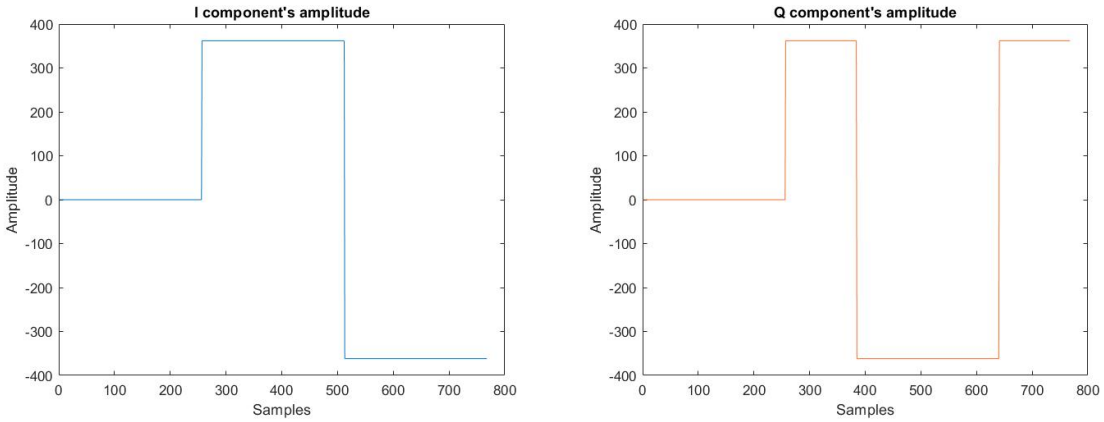

Figure 3.6: Sent I (left) and Q (right) amplitude values to generate QPSK symbols with the first 256 samples set to zero

The setup used to do this experiment was to connect all the transmitters to the same receiver, so that we could measure the effects each transmission path has on the system and separate them from any problems there may be in the reception path. To connect all the transmitters to one receiver we used a four-port Wilkinson power divider, which adds the signals of the transmitters and divides it in four. Each transmitter is connected with the Wilkinson power divider with a coaxial cable, which is then connected to the receiver using also a coaxial cable.

3.2.1 Plotting with the IIO Oscilloscope

The FMComms5 transmits and receives signals using voltages. These can be seen in Appendix A in the channels for the ad9361-phy device (device 1), which shows the physical channels the ad9361 uses. Some of these channels are shown as voltages and represent the voltages of the reception and transmission channels. This is the same nomenclature that's used in the plotting window of the IIO Oscilloscope, where the user can choose what voltages to plot and the manner in which they're plotted. For example, we can choose to plot voltages 0 and 1, which are the equivalent of the I and Q components of the received signal when using TX 1.

Regarding the plotting options, the signal can be plotted in time domain, frequency domain or as a constellation.

When plotting with time domain, the user can only configure how many samples can be plotted. In our case, the capturing of the received data in time domain is interesting to see whether the received data is sorted or not by checking if the zeros we placed at the start of the transmitted data are still there in the received data or not. The number of samples used is 768, as is shown in figure 3.7, where the time domain of the received samples when transmitting through TX1 is shown. This time domain changes over time and the figure only shows a capture of a random position of the samples of the received QPSK symbols in the time domain.

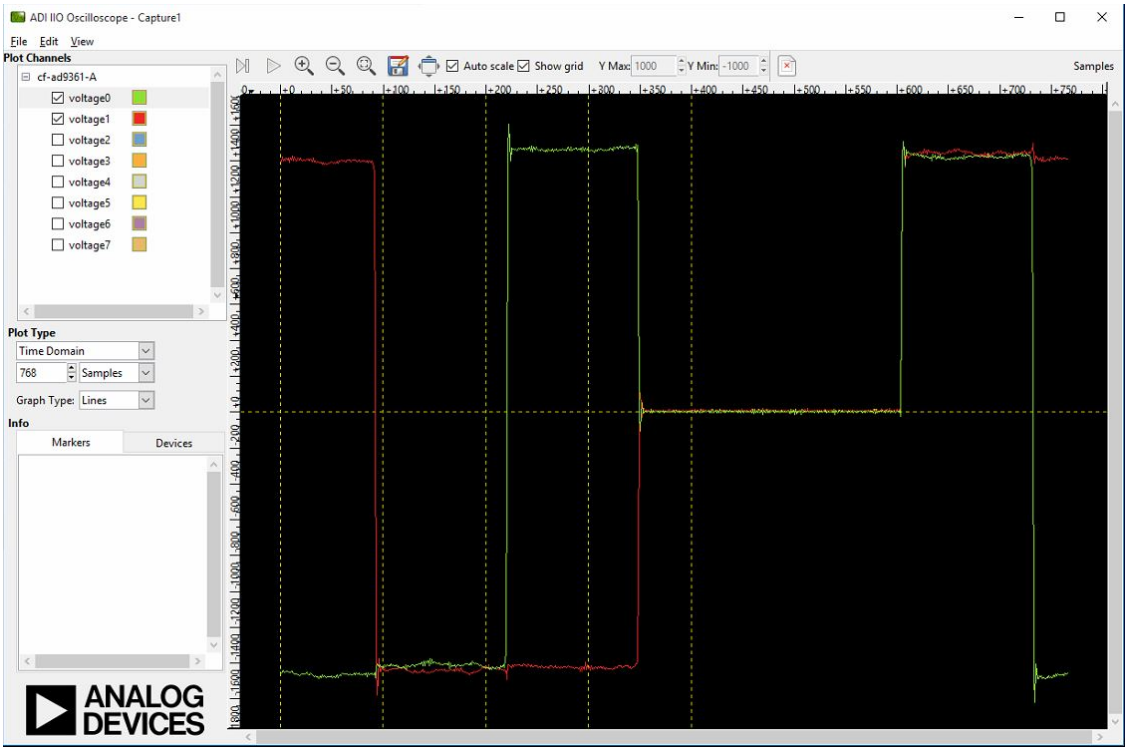

Figure 3.7: Time domain of the I and Q components of the received signal using only TX1.

Frequency domain plotting has a few more configurable options: the user can choose the FFT size, the FFT average and the offset of the FFT graph. In our experiment, the frequency domain is interesting to see if the spectrum matches the theoretical spectrum of a QPSK or not. As is shown in figure 3.8, it does match the QPSK theoretical spectrum, so the QPSK is received correctly at a frequency level. This functionality may be more useful to see the spectrum of, for example, a certain filter or when using the option to send CW tones in the FPGA settings. The higher the FFT average value is, the lower the noise is, which may be useful to observe where the spurious of a certain signal are. In our case, we've used an averaging of 32, which is good enough to reduce the noise levels and see the spectrum in more detail. Again, only the frequency domain of TX1 is shown in this figure, but it's the same for the other transmitters if send the same data.

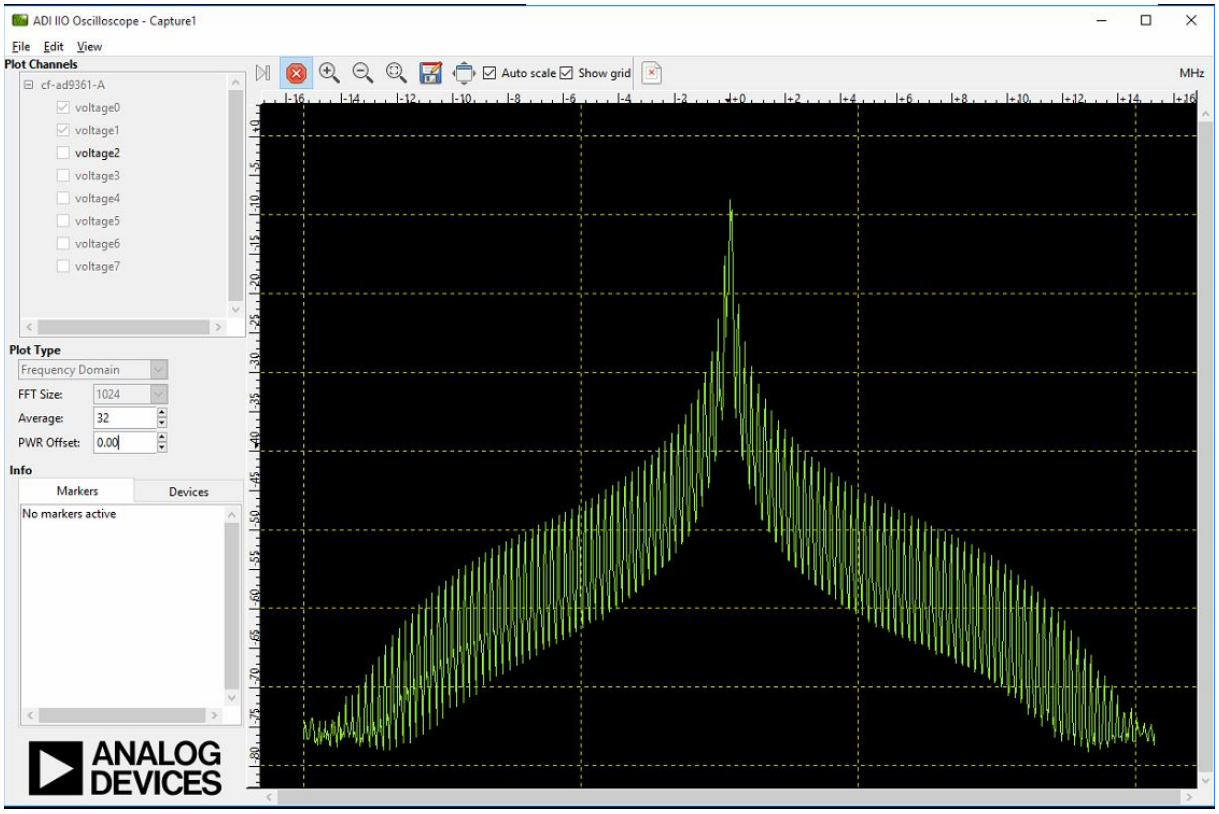

Figure 3.8: Frequency domain of the received signal using only TX1

The most interesting plotting option for our experiment is the constellation. With this option the I and Q components are plotted and we can observe the received symbols and whether the transmitters are actually well synchronised or not, the I channel being plotted on the X axis and the Q channel on the Y axis. The samples plotted are also configurable in this option. The constellation of each transmitter is shown in figure 3.9 where, as in the time domain plot, the number of samples has to be set to 768 for our experiment. These constellations give a lot of information on the quality of the received data, especially in terms of its phase. We can

see that each transmitter has a different phase, an effect that will be explained in section 3.3.2.

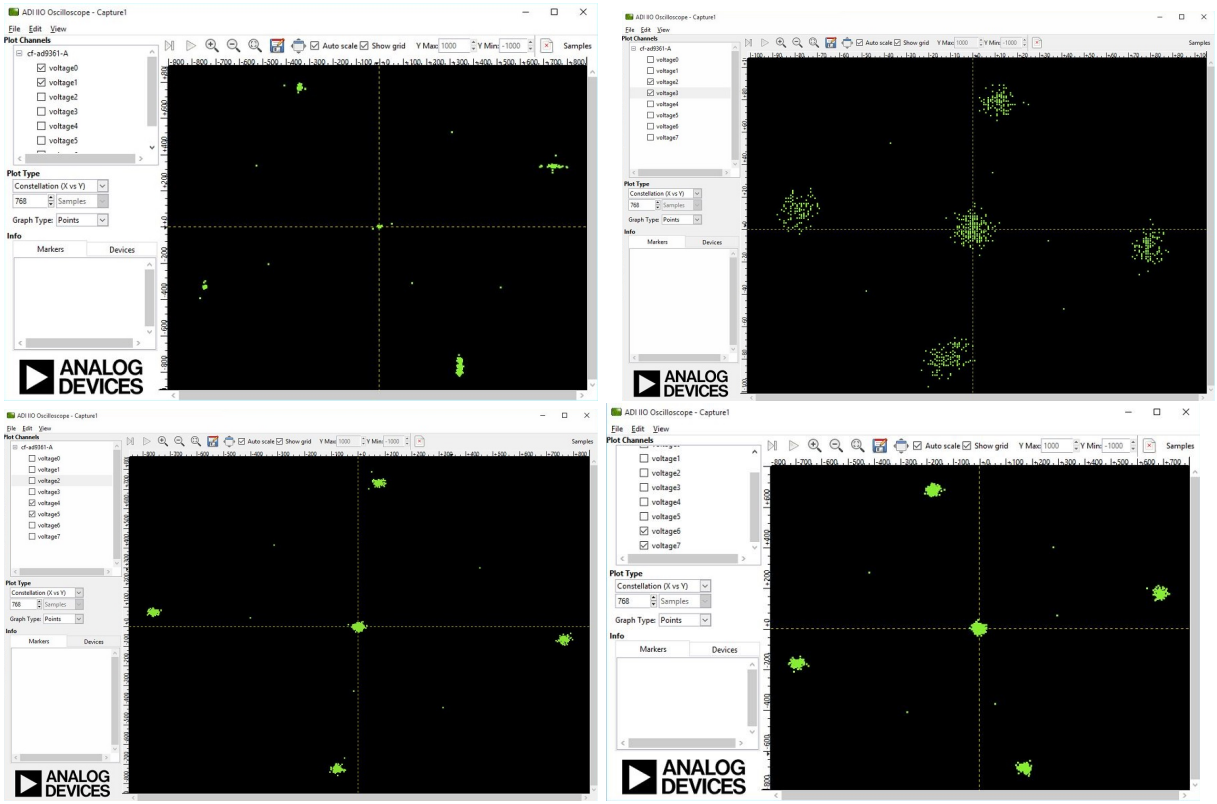

Figure 3.9: Constellation of the received signal for each transmitter (TX1 is top-left, TX2 is top-right, TX3 is bottom-left and TX4 is bottom-right)

Finally, for each of these options, there's also a *math channel* option that allows the user to enter any equation and plot it. There's also an already designed math equation which shows the cross-correlation of the complex signals.

Once the signals have been received, the IIO Oscilloscope allows to save the received signals in MATLAB format, which can be useful to further process the signal with MATLAB.

3.3 Results and problems during synchronisation of transmitters

The results of the experiment are based on whether it accomplishes its goal: to see if the transmitters are synchronised and if MIMO efficiency is accomplished. Too see this, we use IIO Oscilloscope's capture window in time domain and as a constellation, which display the

received data in real time. The basis on which MIMO is efficient was that the amplitudes of the transmitters are summed, which only happens when they're synchronised, so the time domain of the received data will show us information on whether the symbols are in order or not, and the constellation will show if the amplitudes are, indeed, the sum of the amplitude of each transmitter.

In these terms, we found that the symbols don't arrive in the order they were sent and the constellation doesn't show that the transmitters are synced because the amplitudes do not sum. The problems found are explained in the following subsections. To solve them, further signal processing will be necessary, which will be explained in chapter 4.

3.3.1 Reception of unsorted symbols

When observing the time domain of the received data, the zeros we added at the start of the sent data don't appear to be in the position they were sent, but rather in a random position. Thus, the conclusion is that the received symbols are unsorted. The FMComms5 sends the data continuously; it sends the whole frame (the 768 samples) and then it starts again and sends the frame again, repeating this process until the user stops it. This causes the problem because, when the user stops the process, the IIO Oscilloscope captures the received data at a random point, not after the whole frame is sent. Hence, when the data is captured, it's unsorted and the zeros that were supposed to be at the start of the receiving data are at a random section of the captured data.

For this reason, the symbols in themselves are sorted, but the whole block of symbols (the frame) is not.

The IIO Oscilloscope doesn't provide a solution to this since the user can't configure it so that it stops after it has sent the whole frame.

The originally sent data (in I and Q components) is shown in figure 3.10 and the received data in figure 3.11. In these figures, the displacement of the zeros in reception is clear, the zeros being sent at the start (samples 1 to 256) of the frame and being received at samples 395 to 651.

These figures were captured while only using one of the transmitters instead of the four transmitters at the same time, but the displacement of the zeros would be the same for all transmitters, since the problem is only related to the user stopping the process. Still, the figure of the received I and Q components would be unclear if all the transmitters were used, because each transmitter introduces a phase error that is explained in the following subsection.

In this case, the setup used was to connect the four transmitters directly to one receiver

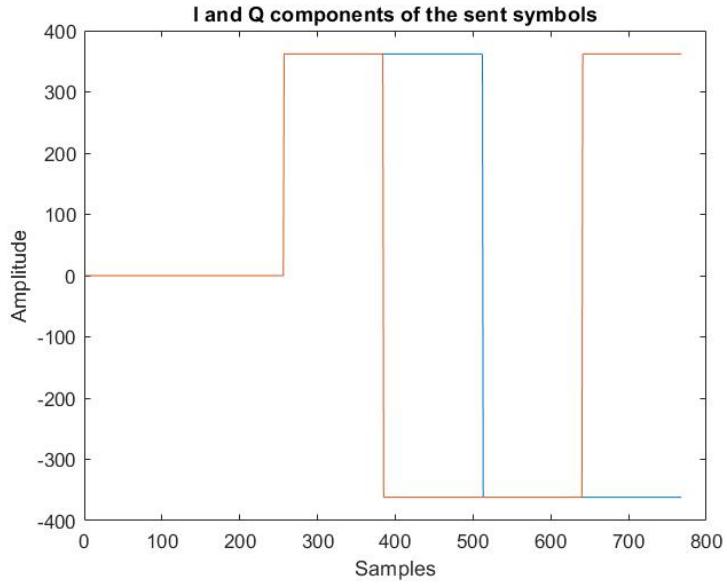

Figure 3.10: Time domain of the sent I (in blue) and Q (in orange) amplitudes

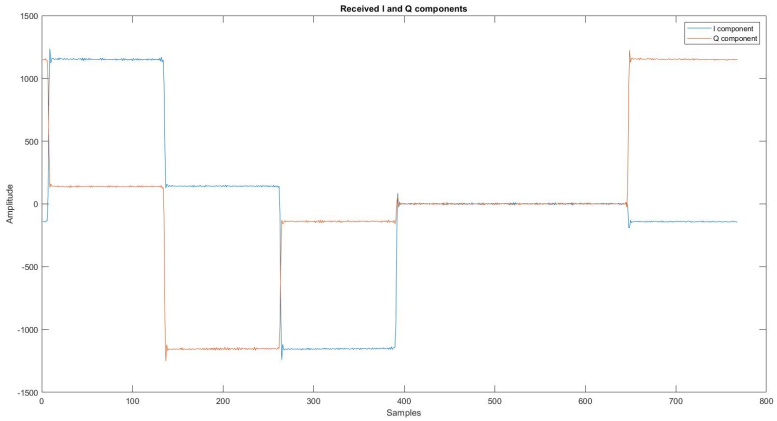

Figure 3.11: Time domain of the received I (in blue) and Q (in orange) amplitudes, where the zeros are clearly displaced

using coaxial cables, although using only one transmitter would've worked too to see the effects of this problem since the problem is the same for the four transmitters because their data is captured at the same time.

How the disordered symbols problem is eventually resolved using MATLAB is explained in section 4.2.2.

3.3.2 Phase error in the received symbols

By plotting the constellation of the received data having sent data through all the four transmitters, we could see whether the transmitters where synchronised or not, by checking if the amplitudes of each transmitter had been summed correctly and had given a QPSK constellation with a bigger amplitude than when using only one transmitter.

This was not the case and we observed that the transmitters weren't synchronised because each of them introduced some phase error.

This effect can be seen in the time domain plot in figure 3.11, where, even if the zeros hadn't been displaced, the received amplitudes would'nt resemble the transmitted amplitudes because the phase isn't the same. For example, if one looks at the data that comes right after the array of zeros, the I and Q components in the transmitted data are at their highest amplitude but, in the received data, the Q component does have the highest amplitude but the I component doesn't. This is because of the introduced phase error, which changes the values of the I and Q component's amplitude.

The effect, though, is seen better when plotting the constellation.

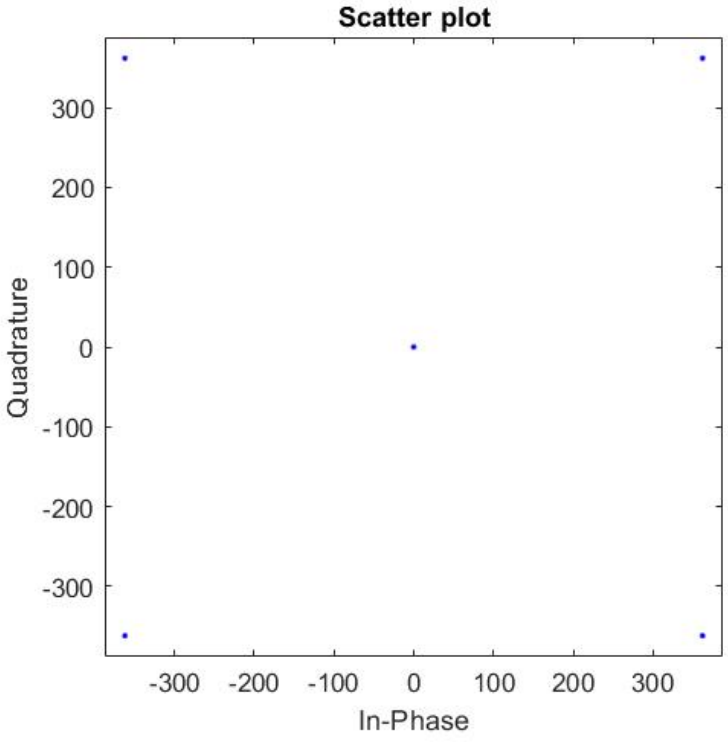

Figure 3.12: Constellation of the transmitted symbols

As one can see in figure 3.12, the phase values for each transmitted symbol follow table 3.2,

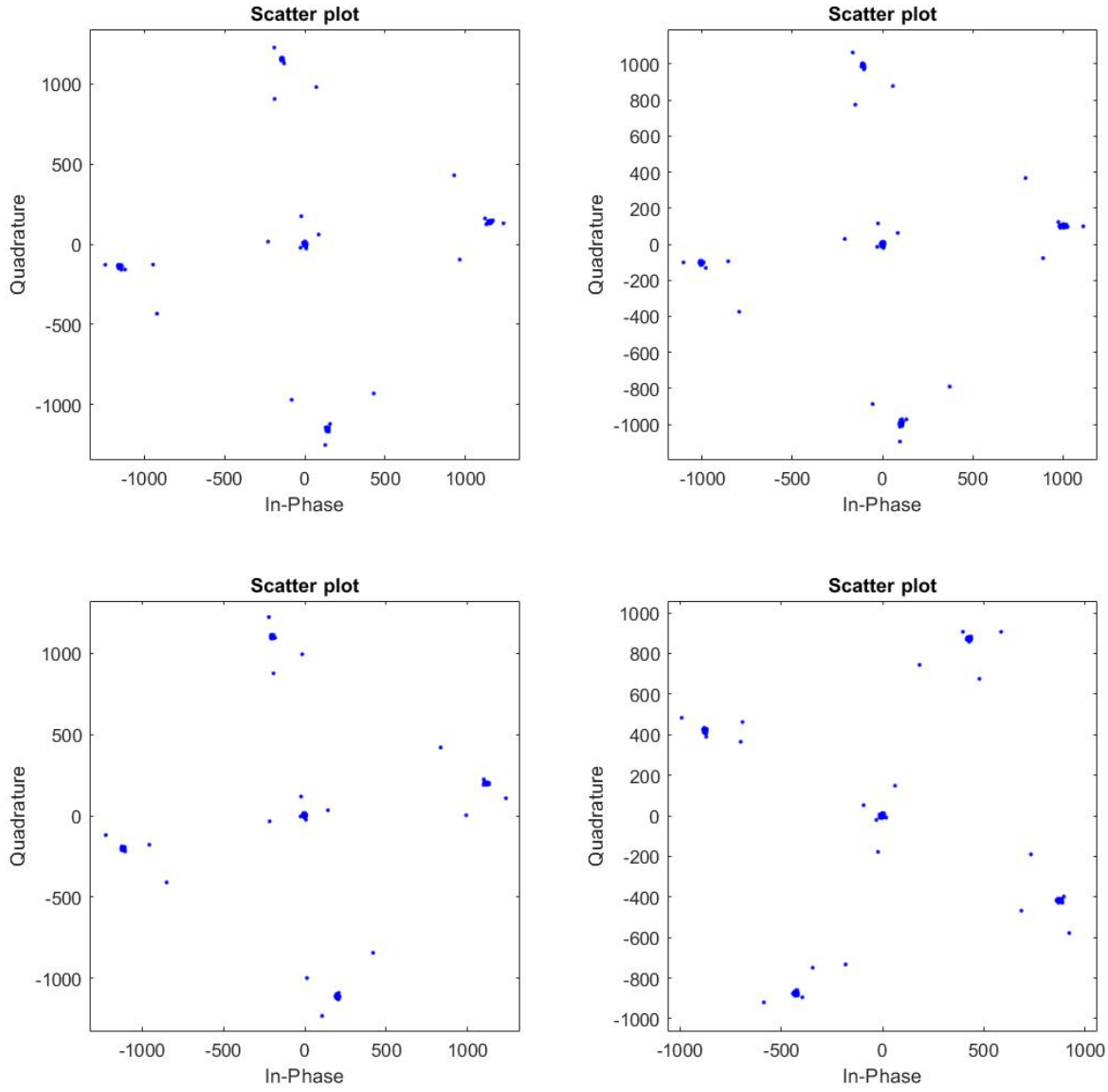

Figure 3.13: Constellation of the received symbols using only TX1 (top-left), TX2 (top-right), TX3 (bottom-left) and TX4 (bottom-right).

but these values greatly change when observing the constellation of the received data in figure 3.13. Although the phase between each symbol hasn't changed (it's still 90°), the whole data has suffered a phase change induced by the FMComms5 in itself and the path the data goes through. Figure 3.13 shows the received data when only transmitting with each transmitter, where the different phase error that each transmission path introduces can be seen.

This error in phase prevents the synchronisation of transmitters because their amplitudes can't be summed. If they were synchronised, each amplitude would have its own phase and the

Bits	Phase
00	$\frac{\pi}{4}$
01	$-\frac{\pi}{4}$
10	$\frac{3\pi}{4}$
11	$-\frac{3\pi}{4}$

Table 3.2: QPSK symbols' theorical phase

end result would cause destructive interference to the constellation. Furthemore, the rotation isn't the same for each transmitter, even if we connect each one to the same receiver using the same cable, the phase difference isn't the same. The rotation appears because the wave travels a certain distance through the coaxial cable with which we've connected the transmitters with the receiver and each transmitter's wave travels through it differently. If the rotation for each transmitter was the same, there wouldn't be destructive interference but, since they aren't, it's necessary to synchronise the four transmitters. This effect isn't only caused by the cable or by some external factor, but by the FMComms5 board in itself too.

To obtain the desired result, the phase error has to be calculated for each transmitter independently and, once all the transmitters' phase errors have been figured out, they should be corrected in order to be able to sum each transmitter's wave amplitude without causing destructive interference and achieving MIMO efficiency. The phase error calibration process is explained in section 4.2.3, where the problem is solved using MATLAB.

To research the effects of this problem, the four transmitters needed to be connected to the receiver to understand the nature of the problem. Since we found that each transmission path has a different phase error, we connected the four transmitters to a Wilkinson power divider in the same way explained in section 3.2. When using MATLAB to solve this problem, the same setup will be used.

3.3.3 Other problems

Difficulty generating the symbols in file format: While other applications like MATLAB provide intuitive symbol generation, either with MATLAB code in itself or using Simulink, the IIO Application's input format is uncomfortable and is a disadvantage in comparison with other applications. While data can still be sent, its generation is uncomfortable and mandates the user to stick to limited amounts of data. The required format for the IIO interface is explained in the FMComms5 user guide [Deva].

3.3.4 How to solve these problems

The IIO Oscilloscope doesn't provide a platform from which these problems can be solved, since no processing of data is available.

Because of this, MATLAB will be used to further extend the research on the FMComms5 SDR, where post-processing of data is available and signal generation is much easier. Besides, Analog Devices provides a good amount of resources to enable MATLAB interaction with the board. The problems encountered related to phase-error and unsorted symbols are still found when simulating with MATLAB since they're caused by the FMComms5's design and the travel paths, not because of the IIO Oscilloscope, but this project aims to design a model that reduces them and allows the capabilites of the FMComms5 to reach their full extent.

Still, the IIO Oscilloscope is a good tool to introduce the user to the FMComms5, especially because it shows its configurability, and could also have an educational purpose which will be explained in section 3.4.

3.4 Evaluation of educational possibilities

As mentioned in the Applications of SDR section in Chapter 1 (1.1.2), software-defined radios can be potentially used for educational purposes because they can implement all the functionalities of a communication system while not being too heavy and being portable. Even though they may be difficult to deal with initially in terms of making it run properly, once the user manages it solvently they are relatively easy to use and transport. For these reasons, SDRs may come in handy when the lecturer may want to show a theoretical explanation at a practical level, displaying the concepts in a more intuitive way for the students. The IIO Oscilloscope application can serve this purpose since the input file for a demonstration needn't be very large and it has a good intuitive interface to show these concepts. Some examples of possible concepts that could be explained in a lecture will be explained below, although the FMComms5 has many more possibilities and could be used to explain any given concept, even if some more complex concepts may require using MATLAB instead of the IIO Oscilloscope application (especially if it is meant to show some sort of signal processing).

Some of the concepts that could be taught using the IIO Oscilloscope application are:

- Spectrum: To introduce a general idea and nuances of the frequency spectrum the IIO Oscilloscope is ideal, since one can see a desired signal in time and frequency domain, as well as in constellation format, giving certain configurability options for all of them. Although a theoretical explanaition for the spectrum is necessary, a practical and visual one would be of help for the student. Since adding to or changing the signal can be done

with relative ease, the lecturer could demonstrate various nuances of the spectrum in a short span of time while keeping the attention of the class, and show its comparison in time domain. The Fourier transform could also be easily demonstrated using this application. While the IIO Oscilloscope shows the spectrum of a certain signal sent and received by the FMComms5, there are other applications with which, using the SDR, the user can see any desired band of the spectrum. An example of the spectrum representation is shown in figure 3.14, where one Continuous Wave (CW) signal has been sent through the first transmitter, although any of the other transmitters could have been used. To do this,

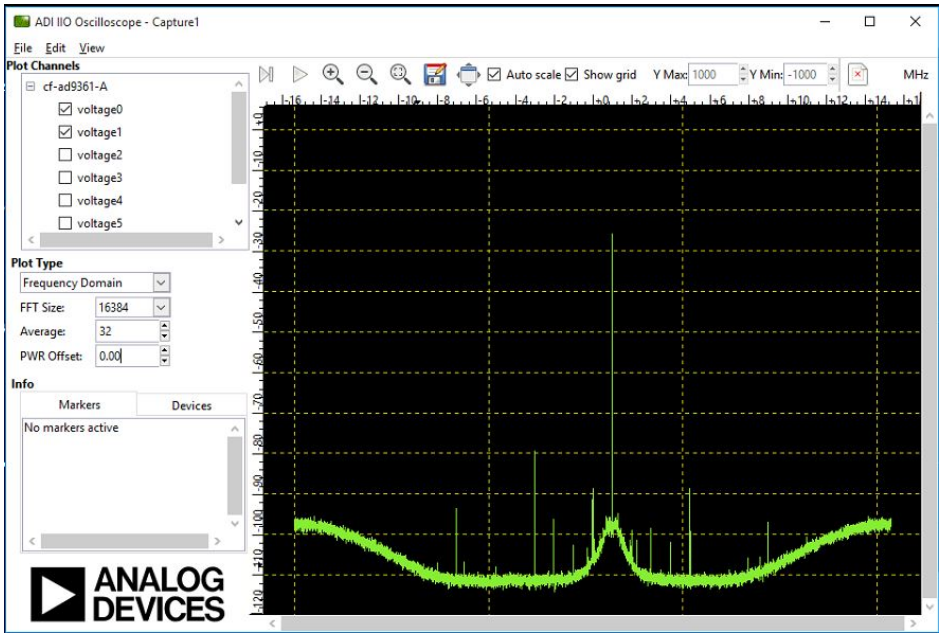

Figure 3.14: Spectrum of the received data when sending one CW tone.

at the FPGA Settings of the FMComms5's control window (shown in section 3.1.1), the option 'CW tone' is chosen in one transmitter while disabling the others. In our case, we used a frequency of 1 MHz, which will generate the CW tone at one MHz from the TX LO Frequency, so this tone is centered at 2436 MHz if the TX LO Frequency is 2435 MHz. If we set a high averaging, the noise is heavily reduced and the harmonics of the tone can be seen more clearly.

- Modulations: Since the application shows the constellation of the received data, modulations could be easily shown and tested in the classroom. The input file the application needs could be designed with ease, just by typing the amplitudes of each in-phase (I) and quadrature (Q) components for each transmitter, each in a separate column, for as many bits as desired following ETSI's 136.211 standard for each modulation. The constellation of any modulated symbols could be shown to increase the student's understanding of it, as well as the corresponding time domain signal. While showing the constellation, the

student could understand the role of each component (I and Q) in the modulation.

- Intermodulation: It's the effect caused by a non-linear system's behaviour that generates undesired amplitude modulation. Intermodulation between each compoent generates additional spurious signals in non-harmonic frequencies (not multiple of the frequencies) but rather the sum or difference between the original frequencies.
  As mentioned in section 3.1, the FMComms5 can be set to send Continuous Wave (CW) tones in each transmitter independently. We can send two CW tones to see the intermodulation between them in the IIO Oscilloscope and measure their value. To do so, we can choose the option to send two tones through one transmitter, as is shown in figure 3.15. We set one tone at 1 MHz from the LO frequency and the other at 5 MHz from it, so their center frequency will be at 2436 MHz and 2440 MHz, respectively, and both are set at the same scale (-7 dB)

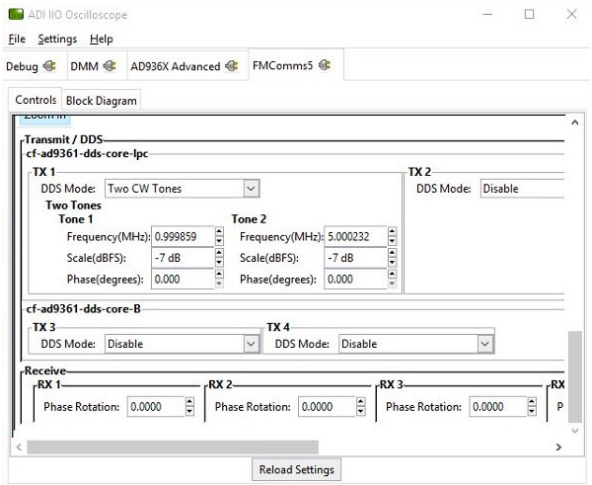

Figure 3.15: FPGA Settings used to send two CW tones with the IIO Oscilloscope

If f_1 is the frequency of the first tone and f_2 is the frequency of the second tone, the third-order intermodulation spurious will show up at $2f_1 - f2 = 2432MHz$ and at $2f_2 - f1 = 2444MHz$, which are shown in figure 3.16, at markers P2 and P4, respectively (the first and second order intermodulation spurious are the regular harmonics). They are at a frequency of 2432 MHz and 2440 MHz, which coincides with the theoretical value. Other intermodulation spurious can also be seen at $f_1 - 2(f_2 - f_1) = 2428MHz$ and at $f_2 + 2(f_2 - f_1) = 2444MHz$, the first one being marked by the P3 marker. In order to see them well and without noise, an averaging of 16 has been done with the IIO Oscilloscope. Hence, the intermodulation effects can be seen clearly using this application.

- Multiple Input Multiple Output (MIMO): The FMComms5 is especially designed to work for MIMO applications since it has four transceivers. MIMO designs are more effective than Single Input Single Output (SISO) designs, an this could be proven and shown by

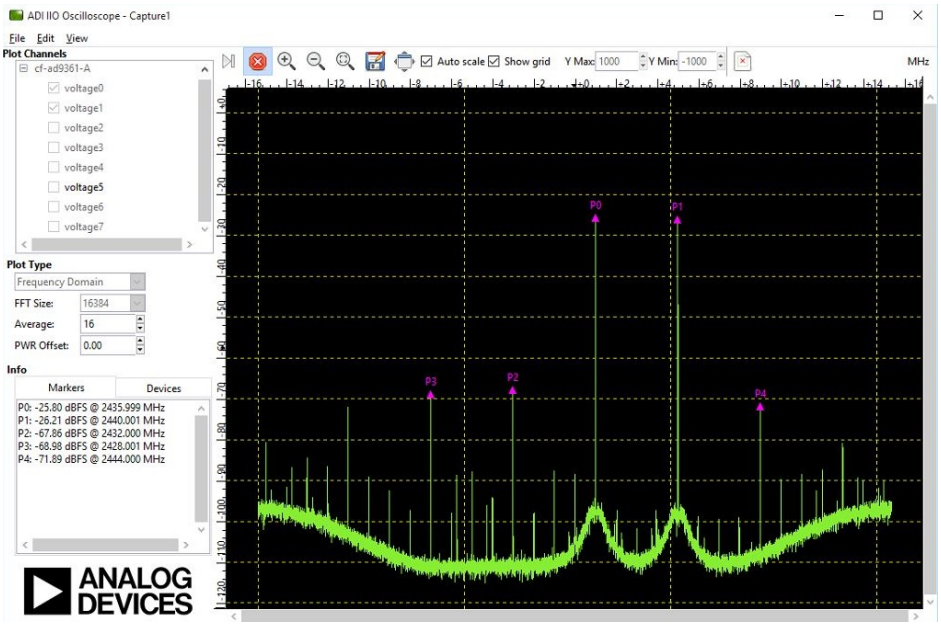

Figure 3.16: Intermodulation between two CW tones.

using one or various transceivers. For this case, MATLAB may be better to measure the effectiveness of MIMO. The transmitter synchronisation experiment explained in this chapter and in chapter 4 is designed to enable MIMO capabilities with the FMComms5 by using the four transmitters at the same time. In chapter 4, where the experiment is completed with MATLAB, the differences between using multiple transmitters or not are shown and, with it, the efectiveness of MIMO is shown by comparing the received amplitude of the symbols between transmitters. In this experiment it's shown that MIMO capabilities are reachable with the FMComms5.

- Filtering: Filters can be designed with MATLAB using a filter wizard and then inputted to the IIO Oscilloscope application. The response of this filter can be seen in the receiving data, so its usages can be demonstrated. A FIR filter can be designed with MATLAB and input into the IIO Oscilloscope application (at the AD9361 Global Settings section) to apply it to the data. An example of a created filter inputted to the IIO Oscilloscope is shown in figure 3.18, where a filter has been applied to two CW tones.

  To generate the filters with MATLAB, the Signal Processing Toolbox and the DSP System Toolbox need to be installed on a MATLAB version higher than 2015b. The filter wizard was designed by Analog Devices and can be installed as a MATLAB app from their github page [Devb]. The filter wizard is shown in figure 3.17, where the passband and the stopband's frequency and amplitude can be designed and the decimation rate of the digital filters and the PLLs can also be set (explained in 2.1.3 and in 2.1.1, respectively). Its response type can also be chosen (lowpass, root raised cosine, bandpass or equaliser) and

a set of already-designed filters for LTE come with the application, its specifications being explained in Analog Device's page [Devh], and can be chosen in the "Device Settings" tab. The only problem with this filter wizard is that we can't enable it in the AD9361

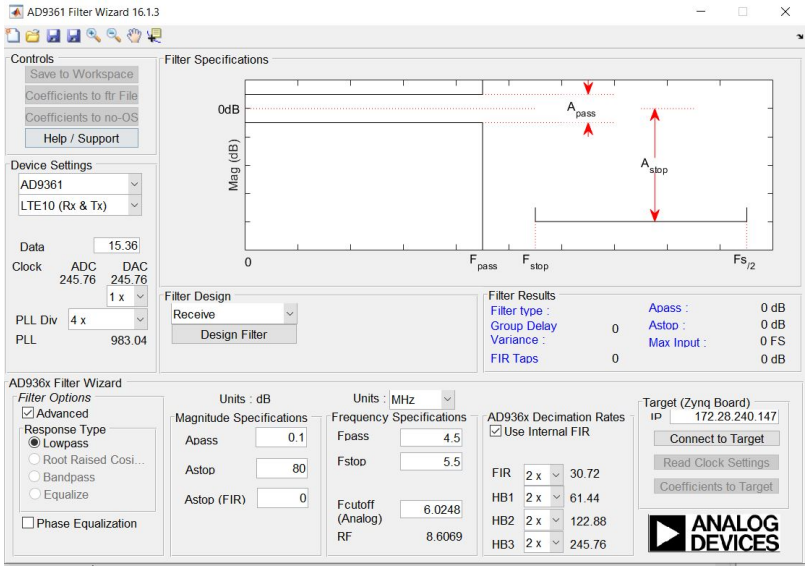

Figure 3.17: Analog Devices' filter wizard for the AD9361

Global Settings tab of the IIO Oscilloscope [Devc] and can only be used if we connect it directly with the board through the wizard itself (the tab at the bottom left), using the 'Coefficients to Target' element.

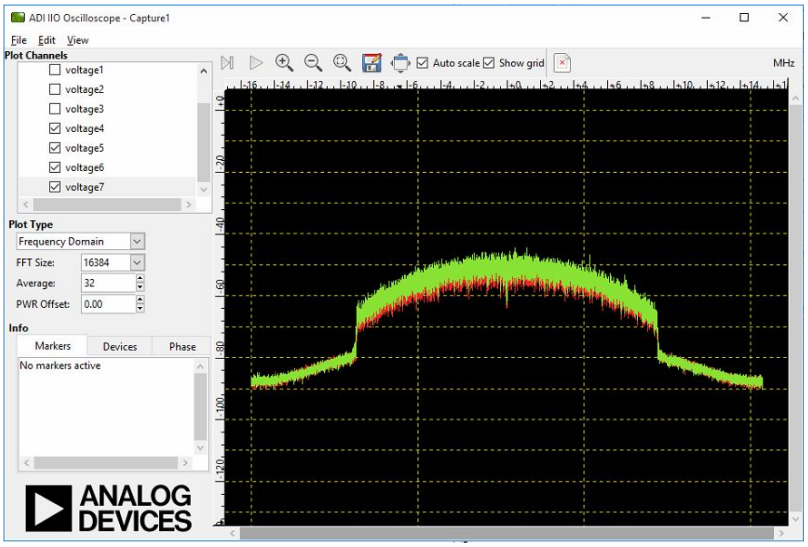

Figure 3.18: Frequency domain of a 1.4 MHz LTE signal with a Butterworth filter applied on it.

Still, there's another option to design filters and use them with the IIO Oscilloscope:

creating a MATLAB file with the filter values and inputting them to the application. Although this option is rather inconvenient, it can be useful in certain instances. For example, we've created an LTE signal with a bandwidth of 6 PRB (1.4 MHz) with a Butterworth filter applied on it, shown in figure 3.18. To do this, we've used the Wireless Waveform Generator with which we can create the LTE signal with the desired bandwidth and we can set the filter we want to use with it. Then, we can save this as a MATLAB file and normalise it, following the steps on the FMComms5 user guide [Devf].

Chapter 4

Interaction with the SDR platform: MATLAB/Simulink

This fourth chapter covers the evaluation of the FMComms5 using MATLAB and Simulink, with which transmission with the four transmitters at the same time with maximum throughput is accomplished. This is achieved by solving the phase and disorder complications in the received symbols that arised in the previous chapter. Besides the completion of the synchronisation of the transmitters, all the MATLAB interfacing with the FMComms5 is covered extensively, describing how they connect and how data transfering is done.

4.1 Connectivity with MATLAB

Using MATLAB with the FMComms5 board allows the user to configure all the elements of the board, to generate the signals to be transmitted with the AD9361 devices and to process the received signals. In comparison with the IIO Oscilloscope, the configurability is relatively similar but MATLAB has a much wider array of possibilities to process the input and output signals. This will allow the problems the IIO Oscilloscope had (explained in sections 3.3.1 and 3.3.2) to be solved.

Hence, we use MATLAB to further extend the capabilities of the FMComms5 and to reach the goal of synchronising the four transmitters. Analog Devices provide extensive documentation on how to connect the board with MATLAB and how to use MATLAB to work with the board effectively.

The way in which the FMComms5 connects to a computer with MATLAB installed on it is with the IIO system object, which is based on the MATLAB System Objects specification.

4.1.1 IIO System object

First of all, to be able to use this object in MATLAB, libiio must be installed, because this object relies on libiio to work. The main idea of libiio is that it's a library used to configure all the attributes of the board (as it was explained in section 2.3), which is what the IIO System Object does but interfaced for MATLAB. Hence, the object is built on libiio and needs libiio to work. This object also needs the Communications Toolbox (from MATLAB) to be installed on version 2015a or higher. The IIO System Object in itself isn't installed, but instead the user calls it, for which the user needs the source code that can be found in github [Devg].

The IIO System Object is based on the MATLAB System Objects specification, which is a MATLAB class with specific methods and properties for modeling an algorithm. They're specifically designed to simulate systems whose inputs change over time, where the output signals depend on both the instantaneous values of the signals and the past behaviour of the system [Mata]. Because they're dynamic and they're configurable, creating an object for the FMComms5 based on a MATLAB System Object makes sense because it provides all the possibilities the FMComms5 has and configurability.

Hence, the IIO System object is built as a MATLAB System Object on libiio and is designed to exchange data over Ethernet with the FMComms5, allowing data streaming from MATLAB to the FMComms5 and viceversa, configurability of the FMComms5 and monitoring of the attributes of the FMComms5 (a list of all the attributes is found on Annex A). To do so, the platform where MATLAB runs must be connected to the Ethernet port on the Zynq ZC706 and, when configuring the object in either MATLAB or Simulink, the IP field has to be filled with the IP Adress of the Zynq ZC706. On figure 4.1 the architecture and the path of the data from MATLAB to the FMComms5 is shown.

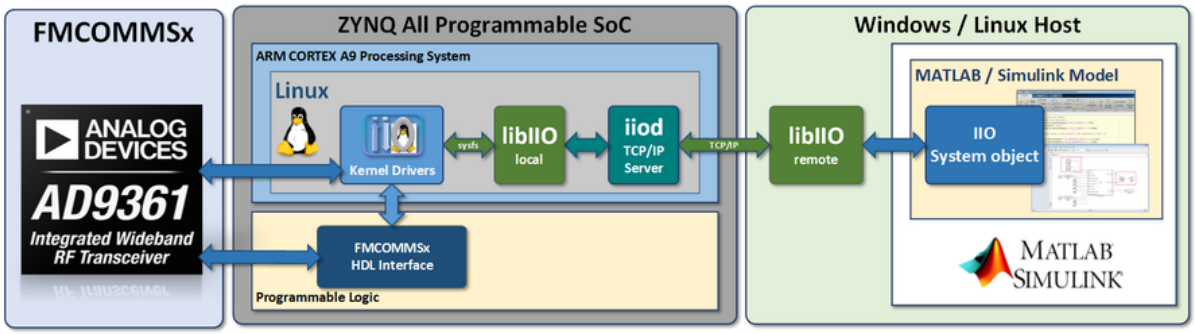

Figure 4.1: Path the data follows from the MATLAB platform to the AD9361 device and viceversa [Deve]

Analog Devices have created a specific IIO System Object for MATLAB and for Simulink,

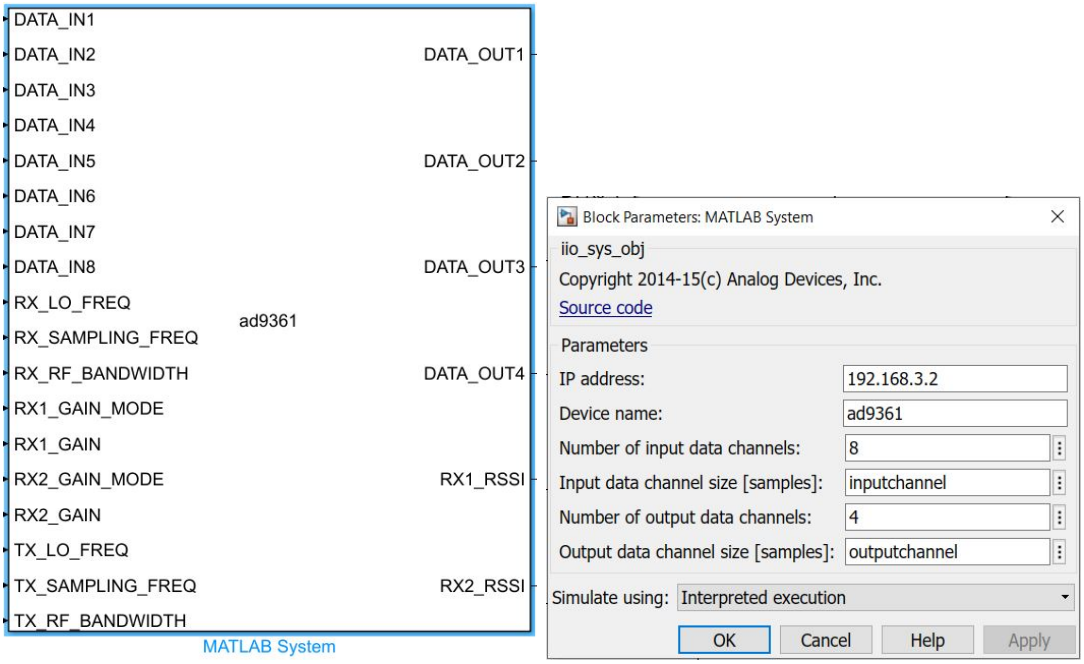

Figure 4.2: Figures of the Simulink System with all its inputs and outputs (left) and its configurable properties (right)

each serving the same purpose and having the same capabilities. The objects are designed to configure solely the AD9361 devices, specific attributes of, for example, the Zynq, can't be configured using these objects, although we don't need to for the purposes of our project. The MATLAB object is designed as a class that has configurable attributes, which also receives data from a configuration file (called ad9361.cfg). The attributes defined in the class are shown in figure 4.2 while in the configuration file, the channels and devices are defined (in this case, including the Zynq ZC706). The object relies on libiio, so there are two other files on which the object is built that give the object libiio capabilities (libiio_if.m and libiio_if_daq2.m). When using the object, these files must be in the same directory of the object, but otherwise the user doesn't have to change anything on these files.

For Simulink, the object is designed to be used as a Simulink system. This block can be found in Simulink's Library browser by the name of "MATLAB System". When double-clicking on it, it opens a prompt window where the user has to browse for the file that contains the Simulink object (in our case, the file's named iio_sys_obj.m) and, if libiio was installed correctly and all the files mentioned before are in the same path, the Simulink system should represent one or both of the AD9361 devices, as shown in figure 4.2.

In this figure, the configurable properties are shown. The system object in itself has several inputs (at the left of the system) and various outputs (at the right) which the user can choose by inputting the desired values. The "DATA_INx" inputs are used to choose what data to send

and through which transmitters. Each transmitter has one input for the in-phase component (I) of the data to be transmitted and one for the quadrature component (Q) (hence, the 8 DATA_IN inputs), and they're sorted so that the first DATA_IN is for the I component of the first transmitter and the last DATA_IN is for the Q component of the fourth transmitter. Besides these inputs, there're also inputs for the frequency, the sampling rate and the RF bandwidth for the receivers and the transmitters, and there's also an input for the gain value and the gain mode (explained in 2.1.4) for each of the two receivers. The values for each of these inputs that we chose are shown in table 4.1, which are roughly the same we used to control the FMComms5 with the IIO Oscilloscope application.

Input	Value
RX_LO_FREQ	2435 MHz
RX_SAMPLING_FREQ	30.72 MSPS
RX_RF_BANDWIDTH	18 MHz
RX1_GAIN_MODE	uint8('manual')
RX1_GAIN	8 dB
RX2_GAIN_MODE	uint8('manual')
RX2_GAIN	40 dB
TX_LO_FREQ	2435 MHz
TX_SAMPLING_FREQ	30.72MSPS
TX_RF_BANDWIDTH	18MHz

Table 4.1: Inputs for the Simulink system and the values we used

Although in figure 4.2 there's only one object, it accounts for both AD9361 devices as it has 8 data inputs (for 4 transmitters) and 4 data outputs (for 2 receivers). This configuration can be changed in the block parameters shown in figure 4.2, where both the number of input and output data channels can be changed if the user doesn't want to use all the available channels. The other properties that are configurable are the IP address, which has to be the one of the Zynq ZC706 to which the MATLAB platform is connected with Ethernet, the device name, which has to be the same as the configuration file name in order to read this one well, and the size of the input and output data channels in samples. This size has to be the same size as the data the user inputs (in our case, it will be 768 samples) or the Simulink system will crash. In our case, then, the variables 'inputchannel' and 'outputchannel' were both 768. We set the gain to "manual" but any of the other modes (especially the slow attack mode) would've worked well too.

These same options are configurable with the MATLAB object, although we used the Simulink model because it seemed easier and more intuitive, but both are equally useful and the user can choose which one to use without any drawbacks. For this reason, the rest of the explanation and the simulations in itselves are all done using Simulink. Further explanation on the Simulink model is done in section 4.2.1

4.1.2 Timeseries

This subsection is added to explain the format in which the data must be inputted into the DATA_IN ports of the IIO System Object in Simulink and how the object outputs the data. Basically, this object acts as the AD9361 devices, so it includes all the transmission and reception paths and all the physical components of the AD9361, all of them integrated into a MATLAB System Object. Hence, the data the user introduces to the DATA_IN ports is the data that the AD9361 devices will transmit and the DATA_OUT is the data the AD9361 will receive.

The sent and received data's format can be set by the user, and we found that the most useful is the timeseries format. This format consists on a time-ordered structure of the data, meaning that each sample has their own time value and each sample is sent according to it. For example, if you want to send ten samples of data consecutevely (in time) but not at the same time, a timeseries object can be created from these samples and they will be sent one at a time in the order the user wants. MATLAB can generate these time values automatically (for which the command `timeseries(data);` is used) or the user can create a time vector that defines the times for the samples (using the command `timeseries(data, time_vector);`, where 'time_vector' contains the time the user has defined for each sample. Since this is the format we'll use for the experiment in section 4.2, a clearer example is found in it. It's important to notice the range of the data that the FMComms5 accepts. Initially, we sent the same data that we used for the IIO Oscilloscope, the amplitudes of which were too high for MATLAB. In MATLAB, the optimal range for the amplitudes of the sent symbols is between 0 and 1, or else it doesn't process the information correctly.

Finally, whilst the timeseries object has to be created with a command when sending data, it can be set directly in Simulink for the received data. This will be further explained in section 4.2.1.

4.2 Transmission of symbols with QPSK modulation

In section 3.2 we explained the first experiment we did: monitoring the four transmitters and the synchronisation between them using the IIO Oscilloscope. The conclusion (explained in 3.3)

was that the transmitters were not synchronised because the received symbols weren't sorted (section 3.3.1) and each transmitter had a phase error (section 3.3.2) that prevented reaching MIMO capabilities with the FMComms5.

We couldn't solve these problems with the IIO Oscilloscope because the application doesn't provide means to do so and, because of this, we decided to move to MATLAB to design the necessary algorithms to achieve transmission synchronisation. With the tools explained in 4.1 (mainly the IIO System Object used in Simulink) we can do the same experiment that we did on the IIO Oscilloscope on MATLAB. With this, we enable all the capabilities of the IIO Oscilloscope and all of its configurability options while also enabling signal processing of the received data to solve the problems we found when using the IIO Oscilloscope application.

This section covers the design of the Simulink model and the algorithms used to solve the problems we've found, correcting the error in phase for each transmitter and the unsorted reception of symbols. To do so, we use the same file that was used as an input for the IIO Oscilloscope, which contains symbols with a QPSK modulation. In this file, there're 128 samples for each symbol (a total of 512 samples) plus 256 zeros at the beginning, that were used to know whether the received data was sorted or not and that will be also used to correct the problem of the unsorted data. The whole experiment will be performed using a Simulink model that is explained in detail in the following subsection.

To perform this experiment, we'll transmit with every transmitter independently, but using the same receiver, to make sure that the receivers in itself don't produce any other problem and the transmitters are synchronised. For this reason, first we simulate using each transmitter independently, calculating the necessary coefficients to solve the errors, and then we simulate using all the transmitters at the same time, each transmitter with each calibrating coefficient applied, to finally achieve transmitter synchronisation and be able to use all of them at the same time.

4.2.1 Simulink model

The main goal of this experiment is to synchronise the four transmitters so that the amplitude of the symbols of each transmitter adds up together and gives one constellation of symbols with an amplitude equal to the sum of all the amplitudes of the transmitters. If they aren't synchronised, the amplitudes can't be summed and the ending result may not even be better that if only one transmitter was used. Hence, synchronisation is vital to achieve MIMO capabilities. First of all, an explanation on how to replicate with MATLAB and Simulink the experiment that was done with the IIO Oscilloscope is done and, after we've achieved correct transmission and reception of data, we proceed to design the algorithms that synchronise the transmitters.

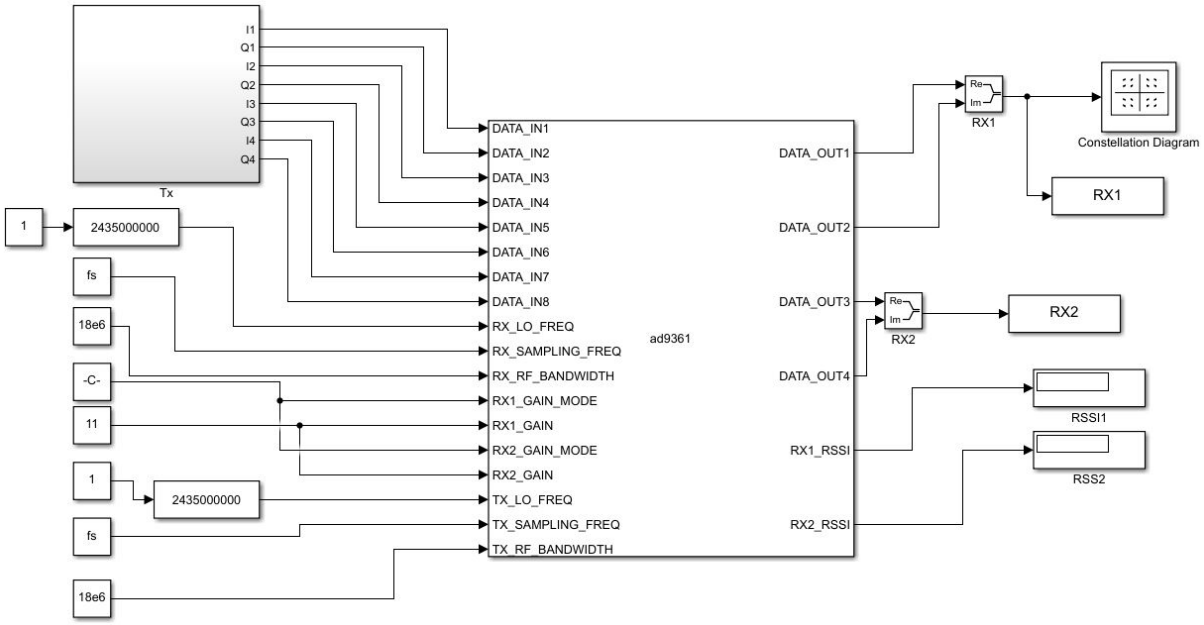

Figure 4.3: Simulink model to simulate transmission and reception with the FMComms5

The purpose of the Simulink model we've designed is to transmit and receive data in the most efficient way possible, but the correction of the phase error and the unsorted data isn't done in this Simulink model and will be implemented later with MATLAB code. Hence, the Simulink model only includes the IIO System object that simulates the AD9361 devices, the input values for this object and the formatting of the output values to be able to use them with MATLAB later on.

The whole Simulink model is shown on figure 4.3, where we identify the IIO System Object that represents both AD9361 devices, all the input values for the object shown in table 4.1, a MATLAB subsystem for the TX data and all the output formatting elements. As we mentioned before, both the inputted and outputted data from the IIO System Object have to be in time-series format, so some of the objects that are used in the Simulink model have to comply with this. Specifically, the sample times for each element have to be set according to this for the elements that transform the outputted data (which is in I and Q components independently) to I/Q data. Their sample time must be set to `1/fs*outputchannel`, which is the time that each sample that outcomes the element will have, so that it can be succesfully transformed to a timeseries object with the same format as the timeseries object that was inputted to the IIO System Object. All the other elements receive data from the inputted timeseries object, so their sample time can be set to '-1' so that they inherit it from this object.

The subsystem that outputs the data to transmit is shown in figure 4.4, where data is

received from the MATLAB workspace.
First of all, how this data has been created will be explained. From the same file we used for the IIO Oscilloscope, which contains all the data to transmit a QPSK constellation through all the transmitters, we have created eight variables, each containing the samples for one of the transmission channels (which are the I and Q components of each of the four transmitters, giving a total of 8 channels). Initially, these variables had the 768 samples of data to be sent through the channel they represent, which was equivalent to one frame of data, but we decided to send more frames of data (we ended up sending 15), because for the first 4 frames the transmission didn't work due to a delay and because it allowed more data to be sent. The Simulink model has two variables named 'inputchannel' and 'outputchannel' which must be set to the value of samples per frame we are using (in our case, 768), because the IIO System Object uses them to process the data. The following commands are used to create the data to be sent, showing only the data for the first transmitter:

```
test_ch=load('test_allch.dat')/362/sqrt(2);
n_frames = 15;
ch1i_data = repmat(test_ch(:,1),[1 1 n_frames]);
ch1q_data = repmat(test_ch(:,2),[1 1 n_frames]);
```

The 'test_allch.dat' is the file that contains the amplitude values for each transmission channel by rows, that are saved into the 'test_ch' variable. It's divided between 362 and $\sqrt{2}$ to keep it between the range of values the FMComms5 accepts, which is optimally between 0 and 1. '362' is the amplitude value of the symbols in the file, so dividing it by this value and by the $\sqrt{2}$ generates the symbols with the amplitude set by the ETSI standard [Eur] for a QPSK constellation. We choose which chunk of data to save into the ch1i_data and ch1q_data variables by copying all the elements of row 1 and row 2 of the file into them, respectively, and we create repetitions of these elements in 15 frames with the MATLAB command `repmat`.

As we mentioned in section 4.1.2, the most effective way to send data with MATLAB is by formatting the data as a timeseries object. To do this, the following commands can be used for each transmitter:

```
Ts = 1/30.72e6; % Sampling time
n_samples = length(test_ch); % 768 samples
time_vector = [0:n_frames-1]*n_samples*Ts;
test_ch1i = timeseries(ch1i_data, time_vector);
test_ch1q = timeseries(ch1q_data, time_vector);
```

These commands create timeseries objects from specific data (ch1i_data and ch1q_data) and from a time vector. Hence, a time is set for each sample in each frame and all samples are sent in order. The time vector has 15 values, one for each frame, each value lasting the number of samples in the frame multiplied by the sample time.
By using the timeseries object, the sampling time is defined for every element and won't need to be defined again. For this reason, in some of the Simulink blocks the sampling time can be set, but we choose to inherit it from the time vector that we've created (to inherit it, the sample time is set to -1).

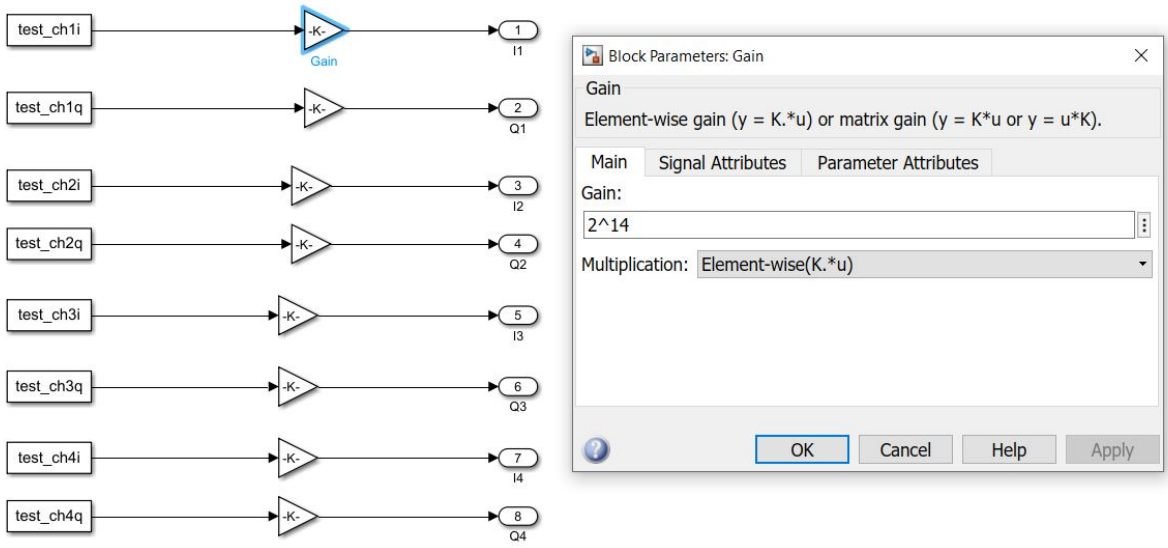

Figure 4.4: Subsystem that outputs the data to transmit

Finally, these variables have to be multiplied by a gain of 2^{14}, as is shown in figure 4.4, or else the IIO System Object isn't capable of identifying them and the outcome will only show noise. Instead of using a file to input the data the user wants to transmit, the data can also be directly generated with MATLAB.

Since we designed the transmitted data to have a timeseries format, we can also choose this option at the receiving end. The IIO System object outputs the received data in four blocks: the I component and the Q component of the first receiver and I component and the Q component of the second receiver, which correspond to the DATA_OUT1, the DATA_OUT2, the DATA_OUT3 and the DATA_OUT4, respectively. We use the 'Real-Imag to Complex' blocks to put together the I and Q components and we save the conjunction to two workspace variables named RX1 and RX2. It's in these variables where we have the option to format the outputting data, where we choose the timeseries option, which will inherit the initial time values of the sent data. With the received data saved in these variables, we'll be able to work with them later.

There are two more outcomes of the IIO System Object, that are the RSSI values for the first and second receiver, that will show the RSSI as it was explained in section 2.1.5.

As it was explained before, though, we don't transmit directly with all the transmitters, but we transmit with each transmitter independently only to one of the receivers. To do so, we send zeros through all the transmitters we don't want to use to disable them, and we send the QPSK symbols through the transmitter we are calibrating. Hence, we first initialise all the transmitters to zero, by using the following commands:

```
zeros_matrix = repmat(zeros(n_samples,1),[1 1 n_frames]);
test_ch1i = timeseries(zeros_matrix, time_vector);
test_ch1q = timeseries(zeros_matrix, time_vector);
...
test_ch4q = timeseries(zeros_matrix, time_vector);
```

Then, when we want to use one of the transmitters, we overwrite the zeros with the corresponding data, without changing the other transmitters. Finally, when changing the transmitter, we fill the transmitter we were using back to zeros (as it can be seen in Appendix B, where the entire code is displayed). This way we can calculate the correction coefficients for all the transmitters independently and achieve transmission synchronisation.

Since the data is received in timeseries format, it will reach the MATLAB workspace with all the data and its time value. To see only the data of, for example, RX1, the command RX1.data shows it all, and to see the time values, the command RX1.time is used. There're many other properties of the timeseries object, which can be found in its documentation [Matb].

4.2.2 Correction of unsorted reception of symbols

An initial explanaition on the basis of the problem of the unsorted reception of symbols is found in section 3.3.1. This section only covers how this problem is corrected using MATAB.
Once we've designed the Simulink model and the data to send with the FMComms5, we proceded to test each transmitter independently. We observed that the same problems that ocurred with the IIO Oscilloscope (unsorted reception of symbols and phase error) still happen when we simulate with MATLAB because these problems are induced by the the FMComms5 in itself. As mentioned before, we test each transmitter independently to be able to correct its own problems and, once all the transmitters have been calibrated, we proceed to simulate with all of them at the same time. When simulating the first transmitter (TX1) with the Simulink model using the commands mentioned before, the reception in the time domain has the zeros

displaced, as shown in figure 4.5. In this figure, the fifth frame of received data is plotted since, as mentioned before, the first four frames don't work well. This data is saved in a variable named `RX1_data` by using the command `RX1_data = RX1.data(:,:,5)`, RX1 being the timeseries object with the received data in the first receiver.

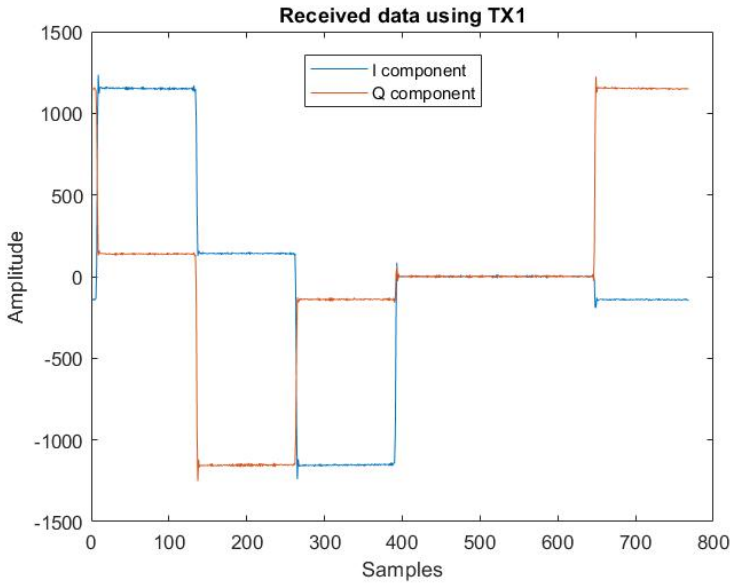

Figure 4.5: Received data at RX1 sent with TX1

We used the zeros as a reference to know where the data started, so we now have to put the zeros back to the start of the sequence. Since the position of the zeros is random everytime we simulate, we have to generate an algorithm that works no matter the position the zeros are in. To do this, we make use of the circular convolution to find the position of the zeros and we use the `circshift` command to reorder the data as it was sent, with the following commands:

```
n_zeros = 256;
ref_conv = zeros(n_samples,1); ref_conv(1:n_zeros) = 1;
conv_data = cconv(abs(RX1_data), ref_conv, n_samples);
[min_data, pos] = min(conv_data);
shift_data = circshift(RX1_data, -pos+n_zeros);
```

The convolution is the integral of the product of two functions after one is reversed and shifted. In our case, the product that is reversed and shifted is the `ref_conv`, which is the function we use as a reference and is shown in figure 4.6.

When we convolute the received data with the reference signal (conv_data = RX1_data(t) $\circledast$ ref_conv (τ-t)), the minimum point of the convolution (or the integral) will be the point where

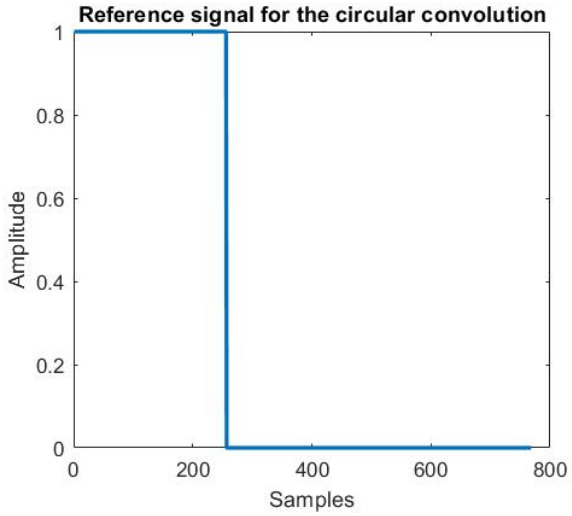

Figure 4.6: Reference signal for the circular convolution

the 256 ones of the reversed and shifted reference signal match the 256 zeros of the received data, shown in figure 4.7, because the integral is virtually zero at this point.

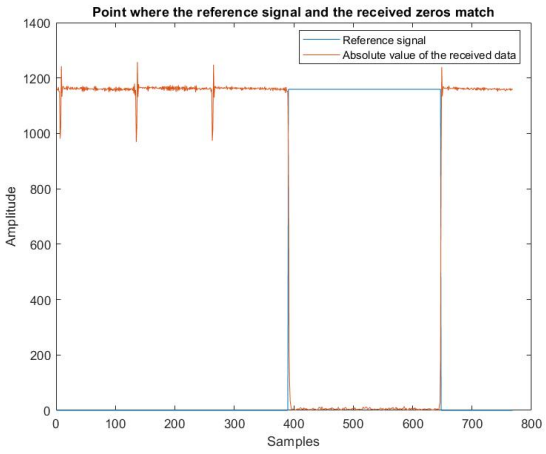

Figure 4.7: Plot of the reference signal and the received data at the matching point

The integral will be at its minimum value only on this point, where the reference signal and the received data match perfectly. With this process, we will know exactly where the zeros are in each simulation, the last zero of the received data will be at the position of the minimum value of the convolution. If we save the position of this value (at variable `pos`), we can shift each sample of the received data to where it was when it was sent, using the function `circshift`. Since the minimum value of the convolution is where the last zero of the received data is, we have to shift the data `-pos + n_zeros` positions to the left so that the zeros are at the beginning of the data.

The amplitude of the reference signal in figure 4.7 isn't really as high as it's represented (in fact, it's just 1), but it has been made bigger only to plot it in a more fitting way and for the reader to understand the concept more easily.

The circular convolution is done with the `cconv` function, where the number of samples has to be specified (in our case it's 768). The outcome of this function is found in figure 4.8, where the minimum point is the `pos` value.

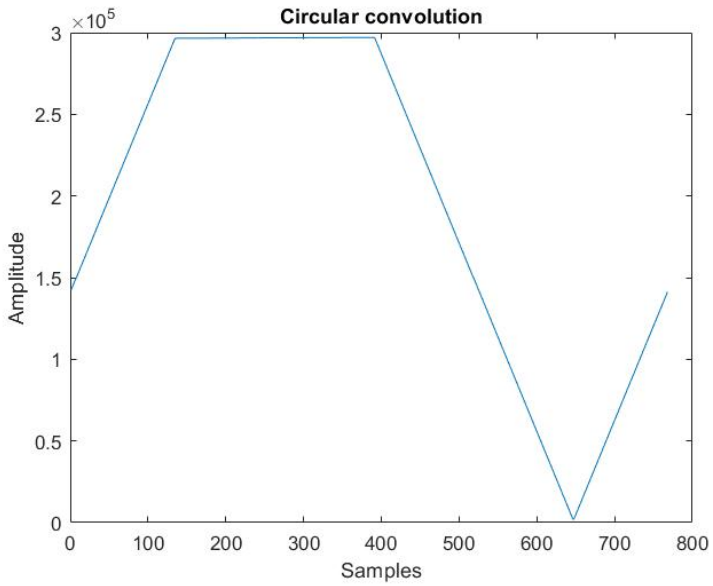

Figure 4.8: Circular convolution of the received data and the reference signal

Finally, the convolution has to be circular because of the nature of the data on which it's applied. The received data is also circular, in the sense that, when we capture it, the data before the zeros (that is the data of the previous chunk of received data) is still plotted. While the simulation is running, the old set of data is gradually substituted for the new set of data, until it's stopped by the user. For this reason, the circular convolution is necessary and a linear convolution wouldn't work well (it would only work if the zeros had "advanced" from its original position, but that's not always the case).

In figures 4.9, 4.10, 4.11 and 4.12, the original received data and the data with the correction of unsorted symbols is shown for TX1, TX2, TX3 and TX4, respectively.

It's worth noticing, though, that if all the transmitters were used at the same time, the correction would be the same for all since they all would be captured at the same time. Hence, when we send data through all the transmitters at the same time, the correction will only have to be done once for the whole received data from the four transmitters.

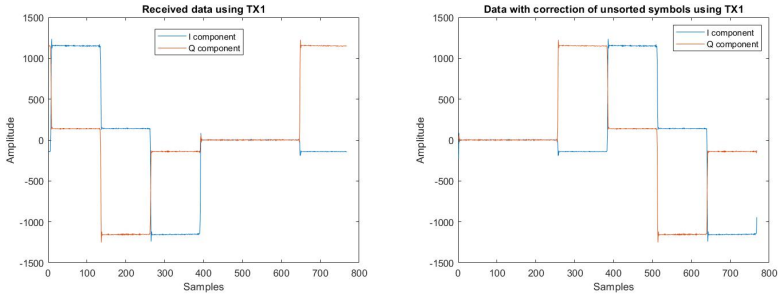

Figure 4.9: Comparison between the received data and the shifted received data using TX1

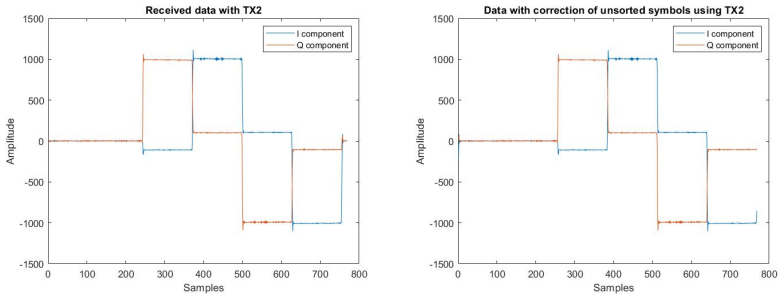

Figure 4.10: Comparison between the received data and the shifted received data using TX2

4.2.3 Correction of phase error in reception

The other main problem we found when trying to synchronise the four transmitters using the IIO Oscilloscope application was that there is a phase error on each of the transmitters (explained in detail in section 3.3.2). Each transmitter, independently, introduces a phase error that affects the received data. Because of this, in order to sync the four transmitters, they must be calibrated in phase. Once the phase error has been calculated, each transmitter will have to be calibrated according to their own phase error and, once each of them has been calibrated, the user will be able to use the four of them at the same time.

To find the phase error each transmission path produces, we calculate the phase each sample has. To find the phase of a particular sample, we use the MATLAB function `angle` and we save the phase of each sample in the `phase_1` variable using a `for` loop, where we also calculate the amplitude of each sample.

```
for (i = 1:1:n_samples)
    phase_1(i) = angle(shift_data(i));
    amplitude(i) = abs(shift_data(i));
end
```

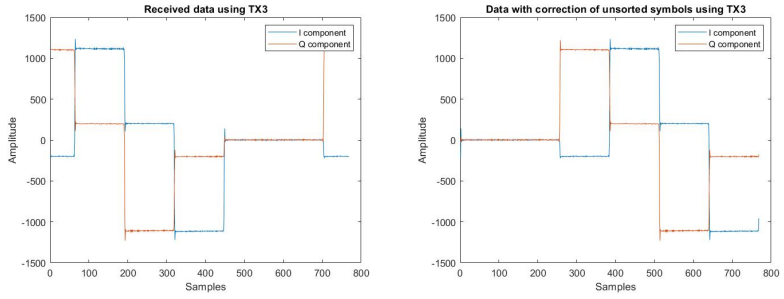

Figure 4.11: Comparison between the received data and the shifted received data using TX3

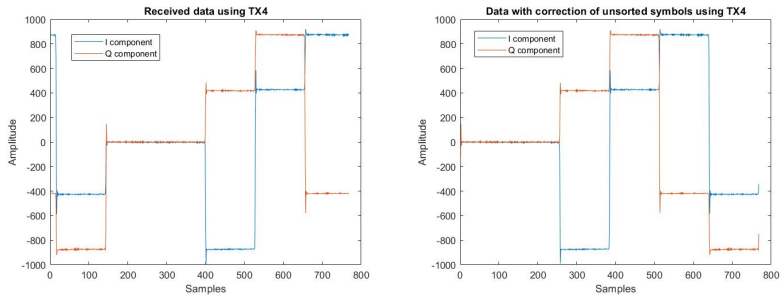

Figure 4.12: Comparison between the received data and the shifted received data using TX4

Then, to calculate the phase error from the theoretical values of the phase of a QPSK, shown in table 3.2, we take the phase values off of the theoretical values. To do so, we calculate the range where each symbol is is to take away its respective theoretical value. Then, we do a mean of the difference without taking into account the initial 256 zeros. Finally, we do a mean of the phase difference of each symbol to have an accurate value of the phase error of all the data.

Knowing that the symbols we sent are in order because we sorted them beforehand, we know where each of the symbols are placed and, hence, we can directly calculate the difference with the theoretical symbol. If we didn't know where the symbols are, though, the phase error could be calculated by first calculating which symbol has been received considering its position in the constellation, or just by dividing the received data with the transmitted data, sample per sample. Since we do know where they are, though, we can use this algorithm to find out the phase error of each transmission path. The order of the sent symbols is: 11,01,00,10, each having 128 samples.

```
range_11 = n_zeros+1:n_zeros+((n_samples-n_zeros)/4); % 257:384
range_01 = n_zeros+((n_samples-n_zeros)/4)+1:n_zeros+((n_samples-n_zeros)/2);
% 385:512
range_00 = n_zeros+((n_samples-n_zeros)/2)+1:n_zeros+3/4*(n_samples-n_zeros);
```

```
% 513:640
range_10 = n_zeros+3/4*(n_samples-n_zeros)+1:n_samples; % 641:768
phase_error(1) = mean(phase_1(range_11) - 5*pi()/4); % 11
phase_error(2) = mean(phase_1(range_01) - 3*pi()/4); % 01
phase_error(3) = mean(phase_1(range_00) - pi()/4); % 00
phase_error(4) = mean(phase_1(range_10) - 7*pi()/4); % 10
amplitude = mean(amplitude(n_zeros:n_samples));
for (j = 1:1:4)
    if (phase_error(j) < -2*pi())
        phase_error(j) = phase_error(j)+2*pi();
    end
end
mean_phase_error = mean(phase_error);
```

In our case, with 256 zeros initially and a total of 768 samples, the range for the '11' symbol is from the sample 257 to 384, for the '01' symbol it's from the 385 to 512 samples, for the '00' symbol from 513 to 640 and for the '10' symbol from the 641 to the 768 samples, as is indicated in the code. Besides the phase error, we also calculate the amplitude in the code above. The amplitude of the data received using each transmitter is of value to us to know if, when we have corrected all the phase errors of the transmitters and we simulate using them all at the same time, the amplitude is actually the sum of the amplitudes received with each transmitter. We calculate it in this piece of code just because it comes in handy and saves time. To calculate it, we just do a mean of the absolute value of each received sample without taking into account the first 256 samples, which are zeros. Hence, we do the mean from `n_zeros` to `n_samples`.

Finally, we have the mean of the phase error stored in the variable named `mean_phase_error` with which we can find the calibration factor for the transmitter that solves its phase error. We achieve this with the following commands:

```
cal_factor = exp(-mean_phase_error*1i);
RX1_data_cal = RX1_data * cal_factor;
```

With these commands, the calibrated received data is stored in the variable `RX1_data_cal`, whose data had been sent using transmitter 1 (TX1). The original received data with no phase correction is shown in figure 4.13 alongside the calibrated received data, using the MATLAB function `scatterplot`.

The effect of calibrating the phase can also be seen in the time domain plot, as is shown in figure 4.14 (on the right), which now clearly resembles the sent time domain plot shown in

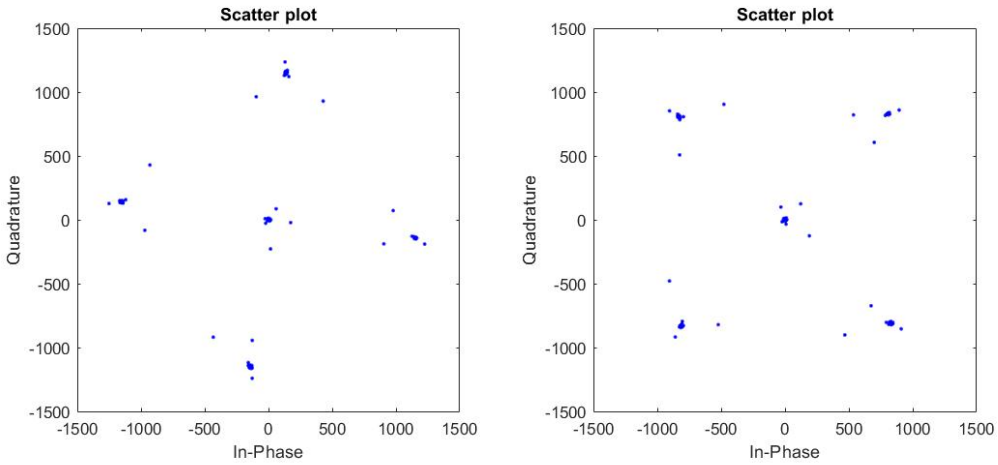

Figure 4.13: Constellation of the received data (left) and the phase-calibrated received data using TX1

figure 3.10. The scatterplots with and without calibration using TX2, TX3 and TX4 are shown

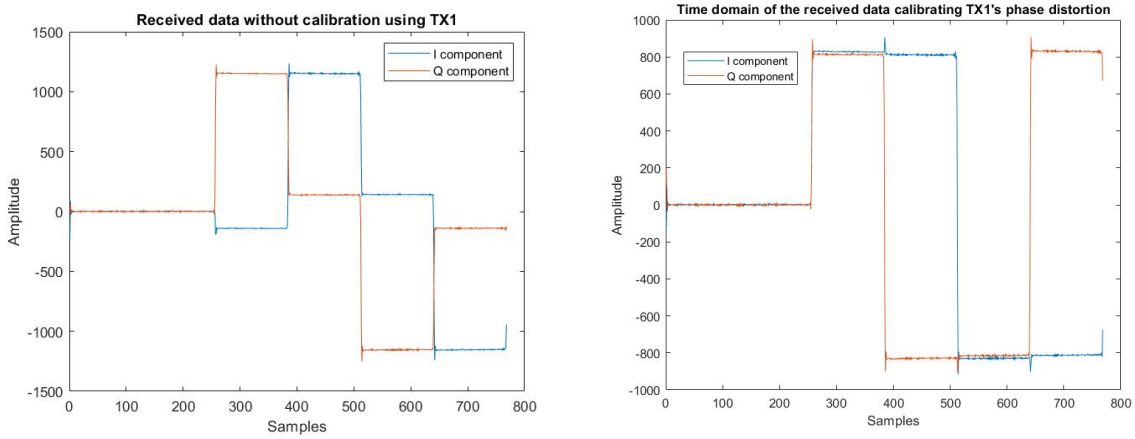

Figure 4.14: Time plot of the received data without (left) and with (right) transmitter calibration

in figures 4.15, 4.16 and 4.17, respectively.

The mean phase error for each transmitter and their calibration factor is found on table 4.2.

This phase error is consistent for every transmitter and doesn't vary much from simulation to simulation. Hence, these same calibration factors can be used for all simulations where the same setup is used and calibrate each transmitter independently to be able to use them all at the same time and enable MIMO capabilities, although sometimes the phase error changes by 180º between simulations. The phase error does change when the setup is changed, though. The calibrated factors in 4.2 are the ones we found when using the setup with the Wilkinson

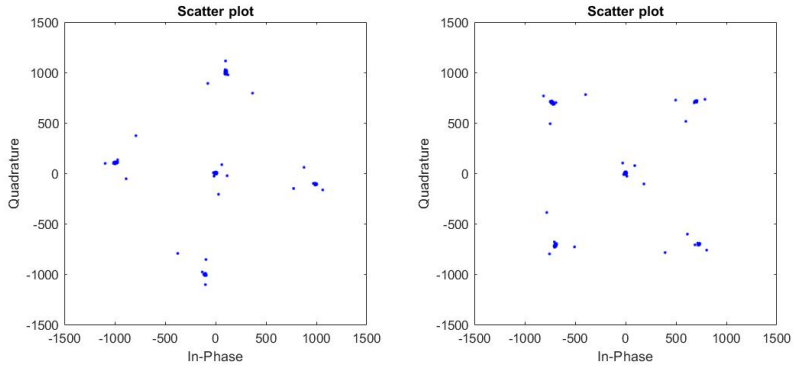

Figure 4.15: Comparison of the constellations with (left) and without (right) phase error using TX2.

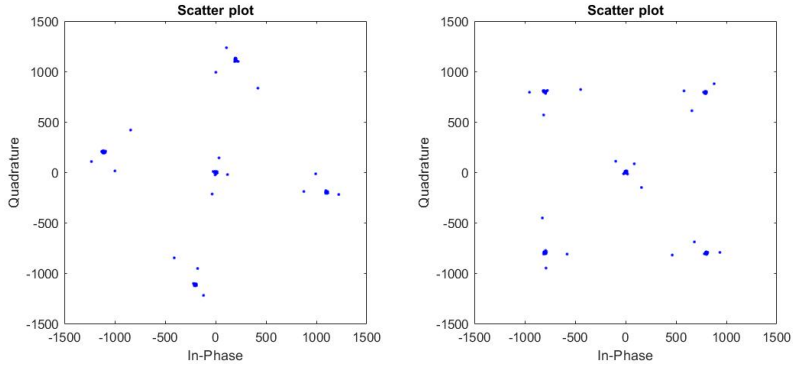

Figure 4.16: Comparison of the constellations with (left) and without (right) phase error using TX3.

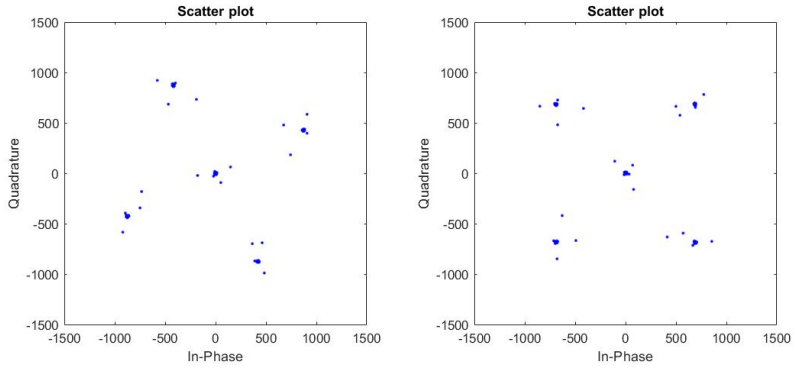

Figure 4.17: Comparison of the constellations with (left) and without (right) phase error using TX4.

power divider, but when we used a setup where the transmitters were directly connected with

TX	Mean Phase Error	Calibration factor
TX1	-53.5639º	0.5939+0.8045j
TX2	-58.044º	0.5293+0.8485j
TX3	-157.2555º	-0.9222+0.3866j
TX4	-178.5317º	-0.329-0.9443j

Table 4.2: Mean phase error and calibration factor for each transmitter.

a coaxial cable to the receiver, the phase varied especially for transmitters three and four. The algorithms worked well with both setups, with the correction of the phase error using the setup where we connected the transmitters directly to the receiver being shown in figures 4.13, 4.15, 4.16 and 4.17.

Once we have all the calibration factors, we can proceed to simulate with the four transmitters at the same time. With the phase calibration, there won't be any destructive interference and the received data should have, approximately, four times the amplitude of a single transmitter (since we'll be using four transmitters). Hence, the calibration will have worked correctly if the received data complies with this statement.

4.2.4 Simulating with synchronised transmitters

As mentioned earlier, when we have the four transmitters synced in phase, we can proceed to simulate the model using all four transmitters at the same time to enable MIMO capabilities for the FMComms5. To do so, we have to multiply the data we sent before the calibration with the calibration factors. The IIO System Object, though, doesn't accept complex data as an input, in the sense that we can't directly multiply the sent data with the calibration factors, because these include complex data. Since it's only a matter of format, though, its solution is rather simple with the implementation of the following code:

```
tx1_data = (ch1i_data + ch1q_data*1i)*cal_tx1;
ch1i_data_cal = real(tx1_data);
ch1q_data_cal = imag(tx1_data);
test_ch1i = timeseries(ch1i_data_cal, time_vector);
test_ch1q = timeseries(ch1q_data_cal, time_vector);
```

The code above is for transmitter 1, but it's the same for the other transmitters, each with its own phase-correcting calibration factor (cal_tx1 being equal to exp(-mean_phase_error*1i)).

First, we tried to simulate each transmitter independently (by setting the other channels to zero using the `zero_matrix` variable) with its calibration factor to check that the phase error is, indeed, corrected, by watching the constellation. Once we found this to be true, we proceded to simulate with all four transmitters. To do so, we just create the timeseries objects with the calibrated data for each channel of each transmitter and we simulate. The resulting constellation of this simulation is shown in figure 4.18

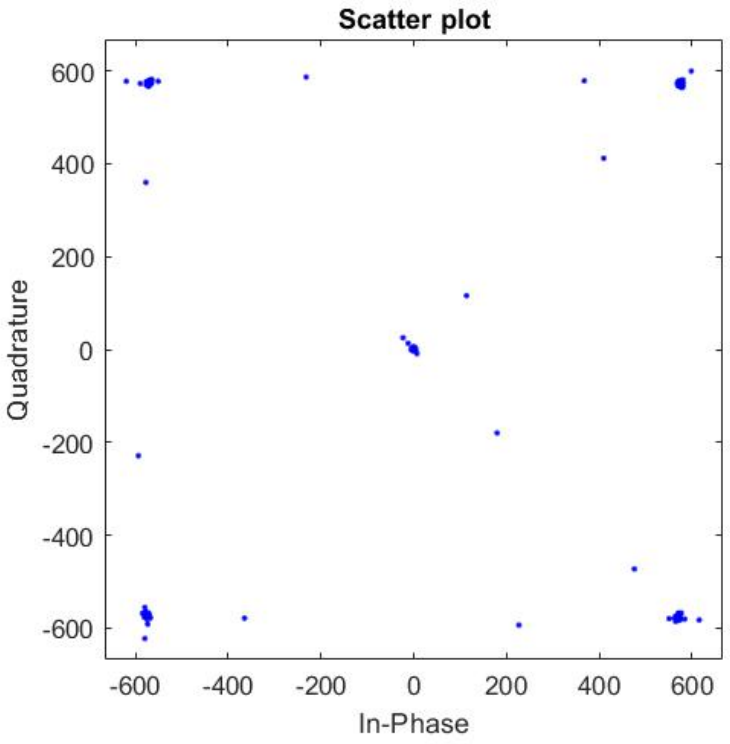

Figure 4.18: Scatterplot of the received data transmitting with all the four synchronised transmitters.

This constellation is, indeed, bigger than the constellations of each independent transmitter. To calculate how much bigger it is, we finally calculate the amplitude of the received data and compare it to the amplitude when using only one transmitter. To calculate the amplitude, we do it in the same way we did it for the independent transmitters:

```
for (i = 1:1:n_samples)
    amplitude(i) = abs(shift_data(i));
end
amplitude_all_TXs = mean(amplitude(n_zeros:n_samples));
```

The resulting amplitude mean value is shown in table 4.3, alongside with the amplitudes found when transmitting with each transmitter alone.

TX	Mean amplitude
TX1	214.9337
TX2	209.0311
TX3	222.4355
TX4	214.0025
All TXs	807.9707

Table 4.3: Mean amplitude values when using each transmitter alone and when using the four synced transmitters at the same time.

From these results, we conclude that the transmitters have synchronised correctly and enable MIMO capabilities since the total amplitude of the received signal when using the four transmitters at the same time is, approximately, 4 times the amplitude of the received data when only using one transmitter. The increment in amplitude provides a higher throughput when transmitting the data because a better modulation could be used and, hence, more symbols can be sent at the same time.

The sum of the mean amplitudes of each transmitter used alone would give a theoretical mean amplitude of 860.403, which is somewhat higher than the result we have, because of the precision of the phase calibration factor, but still gives, roughly, a 94% of the theoretical value, which we consider to be good. Besides the precision of the calibration factor, when transmitting with all four transmitters, the loss in each transmission path can also be a factor to the degrading amplitude, because it may cause interference when adding to the other transmitters' data. These amplitudes vary slightly between simulations, but the performance doesn't vary much.

4.2.5 Implementation with random symbols

Generating the data to be sent from the text file we used in the IIO Oscilloscope is rather uncomfortable and it's way more efficient to generate them with MATLAB, especially if we want to use other modulations too. For this reason, we've generated random symbols to send with various modulations, following ETSI's 136.211 standard [Eur], where the amplitudes for each symbol are defined. The motivation for generating these, besides creating an easier generation of symbols, was to check why the first four sent frames aren't received on the first four frames, where no data is received. By sending random symbols, we can check whether the fifth received frame is the equivalent of the first sent frame or if the first four frames are lost.

We found that there's a delay and that the fifth received frame is, indeed, the first sent frame. We generated random symbols for QPSK, 16QAM, 64QAM and 256QAM modulations, the number of symbols to be generated to be chosen by the user (in this example, 1024 symbols are generated). The modulation is to be chosen with the modulation index.

```
%% Random symbol generation with QPSK/16QAM/64QAM/256QAM modulations
n_samples = 1024;
M = 2; % modulation index
if(M==2) % QPSK, Table 7.1.2-1 136.211 v12.9.0 ETSI
    symbols = [-1/sqrt(2), 1/sqrt(2)];
elseif(M==4) % 16QAM, Table 7.1.3-1 136.211 v12.9.0 ETSI
    r = sqrt(10);
    symbols = [-3/r, -1/r, 1/r, 3/r];
elseif(M==8) % 64QAM, Table 7.1.4-1 136.211 v12.9.0 ETSI
    r = sqrt(42);
    symbols = [-7/r, -5/r, -3/r, -1/r, 1/r, 3/r, 5/r, 7/r];
elseif(M==16) % 256QAM, Table 7.1.5-1 136.211 v12.9.0 ETSI
    r = sqrt(170);
    symbols = [-15/r, -13/r, -11/r, -9/r, -7/r, -5/r, -3/r,-1/r, 1/r,
    3/r, 5/r, 7/r, 9/r, 11/r, 13/r, 15/r];
end
pos_i = randi(length(symbols),n_samples,1);
pos_q = randi(length(symbols),n_samples,1);
data_i = symbols(pos_i)'; data_i(1:256) = 0;
data_q = symbols(pos_q)'; data_q(1:256) = 0;
```

In this example, where the modulation index is 2, QPSK symbols are generated, but the first 256 samples have been set to zero, because the sorting problem and the phase error will have to be corrected when the data is received.
The whole code used to synchronise the transmitters and to generate these random symbols is found in appendix B.

Chapter 5

Conclusions and future work

5.1 Conclusions

Software-defined radios (SDR) have been one of the upcoming technologies in recent years by replacing the traditional telecommunication systems with highly configurable systems whose components can be defined by software while maintining its efficiency and capabilities. The main goal of this project was to analyse the AD-FMComms5-EBZ, a SDR with four transceivers developed by Analog Devices.

During the development of this project the goals we had set initially (explained in section 1.2) have been completed. First, a general research on software-defined radios (SDR) was done to understand their capabilities, applications and general functioning basis. Then, with the documentation provided by Analog Devices, the AD-FMComms5-EBZ board was analysed. Its physical components were studied in the transmission and reception paths and in the clocking and processing systems, while also researching the digital interface of the board that provided the capability of software-defining each component. With it, a full understanding on how SDRs work and the function of each part was achieved, whilst also providing documentation on the practical aspects of the board. By evaluating the digital interface, connectivity with MATLAB and the IIO Oscilloscope application have been reached and used to do some experiments we found interesting. Many problems were found when using the board which required algorithms with MATLAB that solved them, in order to calibrate the board for it to be used as a research tool. With the experiments that have been carried out, all the capabilities that SDRs bring to a telecommunication system have been achieved on the FMComms5. With this, further research has been enabled since the FMComms5 is now fully calibrated to be used in any desired manner. Unfortunately, due to the CoViD-19 outbreak, the experimental part of this project had to be reduced but we ended up doing a more theorical project because of it.

In the end, the research on a robust, configurable and portable system like the FMComms5 SDR that has been done in this project has yielded interesting results that encourage further research on this topic.

5.2 Future research

This project has enabled future research on a wide array of topics since it has made the AD-FMComms5-EBZ software-defined radio usable at a practical level, by calibrating it and solving all the problems it had initially. At first, this project was also meant to test the Directional Modulation but, due to lack of time, and because of difficulties found testing the FMComms5, the project hasn't reached this testing, which is left for future research. Because of the many applications of SDRs, many topics can be researched with it following the research of this project, such as developing layers of a wireless network or testing implementations of algorithms at the physical layer.

Bibliography

[Anaa] Analog Devices, "AD-FMCOMMS2-EBZ Evaluation Board", https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/EVAL-AD-FMCOMMS2.html.

[Anab] Analog Devices, "AD-FMCOMMS3-EBZ Evaluation Board", https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/EVAL-AD-FMCOMMS3-EBZ.html.

[Anac] Analog Devices, "AD-FMCOMMS4-EBZ Evaluation Board", https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/EVAL-AD-FMCOMMS4-EBZ.html.

[Anad] Analog Devices, "AD-FMCOMMS5-EBZ Evaluation Board", https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/eval-ad-fmcomms5-ebz.html.

[Anae] Analog Devices, "AD9361, AD9364 and AD9363", https://wiki.analog.com/resources/eval/user-guides/ad-fmcomms2-ebz/ad9361.

[Anaf] Analog Devices, "AD9361 Product page", https://www.analog.com/en/products/ad9361.html.

[Anag] Analog Devices, *AD9361 Reference Manual, UG-570*, REV-A ed.

[Anah] Analog Devices, "ADCLK846", https://www.analog.com/en/products/adclk846.html.

[But30] S. Butterworth, "On the theory of filter amplifiers", *Wireless Engineers*, Vol. 7, pags. 536–541, 1930.

[Deva] Analog Devices, "AD-FMCOMMS2/3/4/5 Basic IQ Datafiles", https://wiki.analog.com/resources/eval/user-guides/ad-fmcomms2-ebz/software/basic_iq_datafiles.

[Devb] Analog Devices, "ad936x-filter-wizard", https://github.com/analogdevicesinc/ad936x-filter-wizard/releases.

[Devc] Analog Devices, "FMComms5 plugin doesn't support loading filters ", https://github.com/analogdevicesinc/iio-oscilloscope/issues/142.

[Devd] Analog Devices, *FMComms5 schematic*.

[Deve] Analog Devices, "IIO System Object", https://wiki.analog.com/resources/tools-software/linux-software/libiio/clients/matlab_simulink.

[Devf] Analog Devices, "LTE Transmitter and Receiver Example", https://wiki.analog.com/resources/tools-software/linux-software/libiio/clients/lte_example.

[Devg] Analog Devices, "MATLAB bindings for libiio", https://github.com/analogdevicesinc/libiio-matlab.

[Devh] Analog Devices, "MATLAB Filter Design Wizard for AD9361", https://wiki.analog.com/resources/eval/user-guides/ad-fmcomms2-ebz/software/filters.

[Ett] Ettus, "b210", https://www.ettus.com/all-products/ub210-kit/.

[Eur] European Telecommunications Standard Institute, *ETSI TS 136.211: Physical Channels and Modulation*, 12.9.0 ed.

[GNU19] GNU Radio Conference 2019, *gr-iio: Nuances, Advanced Features and New Stuff*, 9 2019.

[Lac95] R. I. Lackey, D. W. Upmal, "Speakeasy: the military software radio", *IEEE Communications Magazine*, Vol. 33, nº 5, pags. 56–61, 1995.

[Lim] Lime Microsystems, "LimeSDR", https://limemicro.com/products/boards/limesdr.

[Man] Mango Communications, "WARP v3", http://mangocomm.com/products/kits/warp-v3-kit/.

[Mata] MathWorks, "MATLAB System objects", https://es.mathworks.com/help/matlab/matlab_prog/what-are-system-objects.html.

[Matb] MathWorks, "Timeseries", https://es.mathworks.com/help/matlab/ref/timeseries.html.

[Mit92] J. Mitola, "Software radios-survey, critical evaluation and future directions", *[Proceedings] NTC-92: National Telesystems Conference*, pags. 13/15–13/23, 1992.

[Nat] National Instruments, "USRP 2922", https://www.ni.com/es-es/support/model.usrp-2922.html.

[Nua] Nuand, "BladeRF", https://www.nuand.com/bladerf-2-0-micro/.

[Nut] Nutaq, "PicoSDR", https://www.nutaq.com/picosdr4x4.

[P.J85] P.Johnson, "New research lab leads to unique radio receiver", Vol. 5, nº 4, pags. 6–7, May 1985.

[Xil] Xilinx, "Zynq ZC706", https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html.

Appendix A

Devices, channels and attributes in libiio environment

This appendix shows all the elements that are configurable by the user, either by changing them directly with Linux commands, using the IIO Oscilloscope or using MATLAB/Simulink. They are shown in the Linux interface by opening a terminal and typing the 'iio_info' command.

IIO context has 1 attributes:
local,kernel: 4.9.0-g0fa1a63
IIO context has 8 devices:

iio:device0: ad7291
9 channels found:

temp0: (input)
3 channel-specific attributes found:
attr 0: scale value: 250
attr 1: raw value: 140
attr 2: mean_raw value: 140

voltage0: (input)
2 channel-specific attributes found:
attr 0: raw value: 1819
attr 1: scale value: 0.610351562

voltage1: (input)
2 channel-specific attributes found:
attr 0: raw value: 3318
attr 1: scale value: 0.610351562

voltage2: (input)
2 channel-specific attributes found:
attr 0: raw value: 3339
attr 1: scale value: 0.610351562

voltage3: (input)
2 channel-specific attributes found:
attr 0: raw value: 3648
attr 1: scale value: 0.610351562

voltage4: (input)
2 channel-specific attributes found:
attr 0: raw value: 1314
attr 1: scale value: 0.610351562

voltage5: (input)
2 channel-specific attributes found:
attr 0: raw value: 1304
attr 1: scale value: 0.610351562

voltage6: (input)
2 channel-specific attributes found:
attr 0: raw value: 1301
attr 1: scale value: 0.610351562

voltage7: (input)
2 channel-specific attributes found:
attr 0: raw value: 1309
attr 1: scale value: 0.610351562

iio:device1: ad9361-phy
11 channels found:

altvoltage0: RX_LO (output)
8 channel-specific attributes found:
attr 0: frequency_available value: [70000000 1 6000000000]
attr 5: frequency value: 2400000000
attr 6: external value: 0
attr 2: powerdown value: 0
attr 4: fastlock_store value: 0
attr 7: fastlock_recall ERROR: Invalid argument (-22)
attr 1: fastlock_save value: 0
178,120,112,208,80,95,122,116,115,113,113,114,248,59,55,123
attr 3: fastlock_load value: 0

altvoltage1: TX_LO (output)
8 channel-specific attributes found:
attr 6: frequency_available value: [46875001 1 6000000000]
attr 1: frequency value: 2400000000
attr 0: external value: 0
attr 4: powerdown value: 0
attr 2: fastlock_store value: 0
attr 3: fastlock_recall ERROR: Invalid argument (-22)

attr 5: fastlock_save value: 0
230,62,181,162,170,228,154,166,162,135,161,174,162,162,227,243
attr 7: fastlock_load value: 0

voltage0: (output)
10 channel-specific attributes found:
attr 7: sampling_frequency value: 32000000
attr 4: sampling_frequency_available value: [2083333 1 61440000]
attr 9: rf_bandwidth value: 18000000
attr 8: rf_bandwidth_available value: [200000 1 40000000]
attr 0: rf_port_select value: A
attr 5: rf_port_select_available value: A B
attr 1: hardwaregain value: -10.000000 dB
attr 3: hardwaregain_available value: [0 250 89750]
attr 6: filter_fir_en value: 0
attr 2: rssi value: 0.00 dB

voltage1: (output)
10 channel-specific attributes found:
attr 7: sampling_frequency value: 32000000
attr 4: sampling_frequency_available value: [2083333 1 61440000]
attr 9: rf_bandwidth value: 18000000
attr 8: rf_bandwidth_available value: [200000 1 40000000]
attr 1: rf_port_select value: A
attr 5: rf_port_select_available value: A B
attr 0: hardwaregain value: -10.000000 dB
attr 2: hardwaregain_available value: [0 250 89750]
attr 6: filter_fir_en value: 0
attr 3: rssi value: 0.00 dB

voltage2: (output)
8 channel-specific attributes found:
attr 5: sampling_frequency value: 32000000
attr 2: sampling_frequency_available value: [2083333 1 61440000]
attr 6: rf_bandwidth_available value: [200000 1 40000000]
attr 7: rf_bandwidth value: 18000000
attr 3: rf_port_select_available value: A B
attr 4: filter_fir_en value: 0
attr 1: scale value: 1.000000
attr 0: raw value: 306

voltage3: (output)
8 channel-specific attributes found:
attr 5: sampling_frequency value: 32000000
attr 2: sampling_frequency_available value: [2083333 1 61440000]
attr 7: rf_bandwidth value: 18000000
attr 6: rf_bandwidth_available value: [200000 1 40000000]
attr 3: rf_port_select_available value: A B

attr 4: filter_fir_en value: 0
attr 0: scale value: 1.000000
attr 1: raw value: 306

voltage0: (input)
15 channel-specific attributes found:
attr 10: sampling_frequency value: 32000000
attr 8: sampling_frequency_available value: [2083333 1 61440000]
attr 6: rf_bandwidth value: 18000000
attr 13: rf_bandwidth_available value: [200000 1 56000000]
attr 1: hardwaregain value: 67.000000 dB
attr 0: hardwaregain_available value: [-3 1 71]
attr 3: rf_port_select value: A_BALANCED
attr 5: rf_port_select_available value: A_BALANCED B_BALANCED C_BALANCED A_N A_P B_N B_P C_N C_P TX_MONITOR1 TX_MONITOR2 TX_MONITOR1_2
attr 4: gain_control_mode value: slow_attack
attr 11: gain_control_mode_available value: manual fast_attack slow_attack hybrid
attr 12: filter_fir_en value: 0
attr 7: rf_dc_offset_tracking_en value: 1
attr 9: quadrature_tracking_en value: 1
attr 14: bb_dc_offset_tracking_en value: 1
attr 2: rssi value: 90.00 dB

voltage1: (input)
15 channel-specific attributes found:
attr 10: sampling_frequency value: 32000000
attr 8: sampling_frequency_available value: [2083333 1 61440000]
attr 6: rf_bandwidth value: 18000000
attr 13: rf_bandwidth_available value: [200000 1 56000000]
attr 3: hardwaregain_available value: [-3 1 71]
attr 2: hardwaregain value: 71.000000 dB
attr 1: rf_port_select value: A_BALANCED
attr 5: rf_port_select_available value: A_BALANCED B_BALANCED C_BALANCED A_N A_P B_N B_P C_N C_P TX_MONITOR1 TX_MONITOR2 TX_MONITOR1_2
attr 4: gain_control_mode value: slow_attack
attr 11: gain_control_mode_available value: manual fast_attack slow_attack hybrid
attr 12: filter_fir_en value: 0
attr 7: rf_dc_offset_tracking_en value: 1
attr 9: quadrature_tracking_en value: 1
attr 14: bb_dc_offset_tracking_en value: 1
attr 0: rssi value: 94.25 dB

voltage2: (input)
13 channel-specific attributes found:
attr 8: sampling_frequency value: 32000000
attr 6: sampling_frequency_available value: [2083333 1 61440000]
attr 4: rf_bandwidth value: 18000000
attr 11: rf_bandwidth_available value: [200000 1 56000000]

attr 3: rf_port_select_available value: A_BALANCED B_BALANCED C_BALANCED A_N A_P B_N B_P C_N C_P TX_MONITOR1 TX_MONITOR2 TX_MONITOR1_2
attr 9: gain_control_mode_available value: manual fast_attack slow_attack hybrid
attr 10: filter_fir_en value: 0
attr 5: rf_dc_offset_tracking_en value: 1
attr 7: quadrature_tracking_en value: 1
attr 12: bb_dc_offset_tracking_en value: 1
attr 0: offset value: 57
attr 1: scale value: 0.305250
attr 2: raw value: 822

temp0: (input)
1 channel-specific attributes found:
attr 0: input value: 38596

out: (input)
1 channel-specific attributes found:
attr 0: voltage_filter_fir_en value: 0
18 device-specific attributes found:
attr 0: dcxo_tune_coarse ERROR: No such device (-19)
attr 11: dcxo_tune_coarse_available value: [0 0 0]
attr 6: dcxo_tune_fine ERROR: No such device (-19)
attr 7: dcxo_tune_fine_available value: [0 0 0]
attr 1: rx_path_rates value: BBPLL:1024000000 ADC:256000000 R2:128000000 R1:64000000 RF:32000000 RXSAMP:32000000
attr 12: tx_path_rates value: BBPLL:1024000000 DAC:256000000 T2:128000000 T1:64000000 TF:32000000 TXSAMP:32000000
attr 2: trx_rate_governor value: nominal
attr 13: trx_rate_governor_available value: nominal highest_osr
attr 17: calib_mode value: auto
attr 3: calib_mode_available value: auto manual manual_tx_quad tx_quad rf_dc_offs rssi_gain_step
attr 14: xo_correction value: 40000000
attr 4: xo_correction_available value: [39992000 1 40008000]
attr 5: gain_table_config value: ¡gaintable AD9361 type=FULL dest=3
start=1300000000 end=4000000000¿
-3, 0x00, 0x00, 0x20
-3, 0x00, 0x00, 0x00
-3, 0x00, 0x00, 0x00
-2, 0x00, 0x01, 0x00
-1, 0x00, 0x02, 0x00
0, 0x00, 0x03, 0x00
1, 0x00, 0x04, 0x00
2, 0x00, 0x05, 0x00
3, 0x01, 0x03, 0x20
4, 0x01, 0x04, 0x00
5, 0x01, 0x05, 0x00
6, 0x01, 0x06, 0x00
7, 0x01, 0x07, 0x00

8, 0x01, 0x08, 0x00
9, 0x01, 0x09, 0x00
10, 0x01, 0x0A, 0x00
11, 0x01, 0x0B, 0x00
12, 0x01, 0x0C, 0x00
13, 0x01, 0x0D, 0x00
14, 0x01, 0x0E, 0x00
15, 0x02, 0x09, 0x20
16, 0x02, 0x0A, 0x00
17, 0x02, 0x0B, 0x00
18, 0x02, 0x0C, 0x00
19, 0x02, 0x0D, 0x00
20, 0x02, 0x0E, 0x00
21, 0x02, 0x0F, 0x00
22, 0x02, 0x10, 0x00
23, 0x02, 0x2B, 0x20
24, 0x02, 0x2C, 0x00
25, 0x04, 0x27, 0x20
26, 0x04, 0x28, 0x00
27, 0x04, 0x29, 0x00
28, 0x04, 0x2A, 0x00
29, 0x04, 0x2B, 0x00
30, 0x24, 0x21, 0x20
31, 0x24, 0x22, 0x00
32, 0x44, 0x20, 0x20
33, 0x44, 0x21, 0x00
34, 0x44, 0x22, 0x00
35, 0x44, 0x23, 0x00
36, 0x44, 0x24, 0x00
37, 0x44, 0x25, 0x00
38, 0x44, 0x26, 0x00
39, 0x44, 0x27, 0x00
40, 0x44, 0x28, 0x00
attr 15: ensm_mode value: fdd
attr 8: ensm_mode_available value: sleep wait alert fdd pinctrl pinctrl_fdd_indep
attr 9: multichip_sync ERROR: Input/output error (-5)
attr 10: rssi_gain_step_error value: lna_error: 0 0 0 0 mixer_error: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
gain_step_calib_reg_val: 0 0 0 0 0
attr 16: filter_fir_config value: FIR Rx: 0,0 Tx: 0,0

iio:device2: ad9361-phy-B
11 channels found:

altvoltage0: RX_LO (output)
8 channel-specific attributes found:
attr 0: frequency_available value: [70000000 1 6000000000]
attr 5: frequency value: 2400000000
attr 6: external value: 0

attr 2: powerdown value: 0
attr 4: fastlock_store value: 0
attr 7: fastlock_recall ERROR: Invalid argument (-22)
attr 1: fastlock_save value: 0
83,78,83,83,83,83,83,83,83,87,83,6,83,83,83,83
attr 3: fastlock_load value: 0

altvoltage1: TX_LO (output)
8 channel-specific attributes found:
attr 6: frequency_available value: [46875001 1 6000000000]
attr 1: frequency value: 2400000000
attr 0: external value: 0
attr 4: powerdown value: 0
attr 2: fastlock_store value: 0
attr 3: fastlock_recall ERROR: Invalid argument (-22)
attr 5: fastlock_save value: 0
251,243,81,179,251,251,250,255,179,211,144,250,251,255,145,255
attr 7: fastlock_load value: 0

voltage0: (output)
10 channel-specific attributes found:
attr 7: sampling_frequency value: 32000000
attr 4: sampling_frequency_available value: [2083333 1 61440000]
attr 9: rf_bandwidth value: 18000000
attr 8: rf_bandwidth_available value: [200000 1 40000000]
attr 0: rf_port_select value: A
attr 5: rf_port_select_available value: A B
attr 1: hardwaregain value: -10.000000 dB
attr 3: hardwaregain_available value: [0 250 89750]
attr 6: filter_fir_en value: 0
attr 2: rssi value: 0.00 dB

voltage1: (output)
10 channel-specific attributes found:
attr 7: sampling_frequency value: 32000000
attr 4: sampling_frequency_available value: [2083333 1 61440000]
attr 9: rf_bandwidth value: 18000000
attr 8: rf_bandwidth_available value: [200000 1 40000000]
attr 1: rf_port_select value: A
attr 5: rf_port_select_available value: A B
attr 0: hardwaregain value: -10.000000 dB
attr 2: hardwaregain_available value: [0 250 89750]
attr 6: filter_fir_en value: 0
attr 3: rssi value: 0.00 dB

voltage2: (output)
8 channel-specific attributes found:
attr 5: sampling_frequency value: 32000000

attr 2: sampling_frequency_available value: [2083333 1 61440000]
attr 7: rf_bandwidth value: 18000000
attr 6: rf_bandwidth_available value: [200000 1 40000000]
attr 3: rf_port_select_available value: A B
attr 4: filter_fir_en value: 0
attr 1: scale value: 1.000000
attr 0: raw value: 306

voltage3: (output)
8 channel-specific attributes found:
attr 5: sampling_frequency value: 32000000
attr 2: sampling_frequency_available value: [2083333 1 61440000]
attr 7: rf_bandwidth value: 18000000
attr 6: rf_bandwidth_available value: [200000 1 40000000]
attr 3: rf_port_select_available value: A B
attr 4: filter_fir_en value: 0
attr 0: scale value: 1.000000
attr 1: raw value: 306

voltage0: (input)
15 channel-specific attributes found:
attr 10: sampling_frequency value: 32000000
attr 8: sampling_frequency_available value: [2083333 1 61440000]
attr 6: rf_bandwidth value: 18000000
attr 13: rf_bandwidth_available value: [200000 1 56000000]
attr 1: hardwaregain value: 68.000000 dB
attr 0: hardwaregain_available value: [-3 1 71]
attr 3: rf_port_select value: A_BALANCED
attr 5: rf_port_select_available value: A_BALANCED B_BALANCED C_BALANCED A_N A_P B_N B_P C_N C_P TX_MONITOR1 TX_MONITOR2 TX_MONITOR1_2
attr 4: gain_control_mode value: slow_attack
attr 11: gain_control_mode_available value: manual fast_attack slow_attack hybrid
attr 12: filter_fir_en value: 0
attr 2: rssi value: 91.00 dB
attr 7: rf_dc_offset_tracking_en value: 1
attr 9: quadrature_tracking_en value: 1
attr 14: bb_dc_offset_tracking_en value: 1

voltage1: (input)
15 channel-specific attributes found:
attr 10: sampling_frequency value: 32000000
attr 8: sampling_frequency_available value: [2083333 1 61440000]
attr 6: rf_bandwidth value: 18000000
attr 13: rf_bandwidth_available value: [200000 1 56000000]
attr 2: hardwaregain value: 71.000000 dB
attr 3: hardwaregain_available value: [-3 1 71]
attr 1: rf_port_select value: A_BALANCED
attr 5: rf_port_select_available value: A_BALANCED B_BALANCED C_BALANCED A_N A_P

B_N B_P C_N C_P TX_MONITOR1 TX_MONITOR2 TX_MONITOR1_2
attr 4: gain_control_mode value: slow_attack
attr 11: gain_control_mode_available value: manual fast_attack slow_attack hybrid
attr 12: filter_fir_en value: 0
attr 0: rssi value: 93.25 dB
attr 7: rf_dc_offset_tracking_en value: 1
attr 9: quadrature_tracking_en value: 1
attr 14: bb_dc_offset_tracking_en value: 1

voltage2: (input)
13 channel-specific attributes found:
attr 8: sampling_frequency value: 32000000
attr 6: sampling_frequency_available value: [2083333 1 61440000]
attr 4: rf_bandwidth value: 18000000
attr 11: rf_bandwidth_available value: [200000 1 56000000]
attr 3: rf_port_select_available value: A_BALANCED B_BALANCED C_BALANCED A_N A_P B_N B_P C_N C_P TX_MONITOR1 TX_MONITOR2 TX_MONITOR1_2
attr 9: gain_control_mode_available value: manual fast_attack slow_attack hybrid
attr 10: filter_fir_en value: 0
attr 5: rf_dc_offset_tracking_en value: 1
attr 7: quadrature_tracking_en value: 1
attr 12: bb_dc_offset_tracking_en value: 1
attr 0: offset value: 57
attr 1: scale value: 0.305250
attr 2: raw value: 751

temp0: (input)
1 channel-specific attributes found:
attr 0: input value: 37719

out: (input)
1 channel-specific attributes found:
attr 0: voltage_filter_fir_en value: 0
18 device-specific attributes found:
attr 0: dcxo_tune_coarse ERROR: No such device (-19)
attr 11: dcxo_tune_coarse_available value: [0 0 0]
attr 6: dcxo_tune_fine ERROR: No such device (-19)
attr 7: dcxo_tune_fine_available value: [0 0 0]
attr 1: rx_path_rates value: BBPLL:1024000000 ADC:256000000 R2:128000000 R1:64000000 RF:32000000 RXSAMP:32000000
attr 12: tx_path_rates value: BBPLL:1024000000 DAC:256000000 T2:128000000 T1:64000000 TF:32000000 TXSAMP:32000000
attr 2: trx_rate_governor value: nominal
attr 13: trx_rate_governor_available value: nominal highest_osr
attr 16: filter_fir_config value: FIR Rx: 0,0 Tx: 0,0
attr 17: calib_mode value: auto
attr 3: calib_mode_available value: auto manual manual_tx_quad tx_quad
rf_dc_offs rssi_gain_step

attr 14: xo_correction value: 40000000
attr 4: xo_correction_available value: [39992000 1 40008000]
attr 5: gain_table_config value: ¡gaintable AD9361 type=FULL dest=3
start=1300000000 end=4000000000¿
-3, 0x00, 0x00, 0x20
-3, 0x00, 0x00, 0x00

-3, 0x00, 0x00, 0x00
-2, 0x00, 0x01, 0x00

-1, 0x00, 0x02, 0x00
0, 0x00, 0x03, 0x00

1, 0x00, 0x04, 0x00
2, 0x00, 0x05, 0x00
3, 0x01, 0x03, 0x20
4, 0x01, 0x04, 0x00
5, 0x01, 0x05, 0x00
6, 0x01, 0x06, 0x00
7, 0x01, 0x07, 0x00
8, 0x01, 0x08, 0x00
9, 0x01, 0x09, 0x00
10, 0x01, 0x0A, 0x00

11, 0x01, 0x0B, 0x00
12, 0x01, 0x0C, 0x00
13, 0x01, 0x0D, 0x00
14, 0x01, 0x0E, 0x00
15, 0x02, 0x09, 0x20
16, 0x02, 0x0A, 0x00
17, 0x02, 0x0B, 0x00
18, 0x02, 0x0C, 0x00
19, 0x02, 0x0D, 0x00

20, 0x02, 0x0E, 0x00
21, 0x02, 0x0F, 0x00
22, 0x02, 0x10, 0x00
23, 0x02, 0x2B, 0x20
24, 0x02, 0x2C, 0x00
25, 0x04, 0x27, 0x20
26, 0x04, 0x28, 0x00
27, 0x04, 0x29, 0x00
28, 0x04, 0x2A, 0x00
29, 0x04, 0x2B, 0x00
30, 0x24, 0x21, 0x20
31, 0x24, 0x22, 0x00
32, 0x44, 0x20, 0x20
33, 0x44, 0x21, 0x00

34, 0x44, 0x22, 0x00
35, 0x44, 0x23, 0x00
36, 0x44, 0x24, 0x00
37, 0x44, 0x25, 0x00
38, 0x44, 0x26, 0x00
39, 0x44, 0x27, 0x00
40, 0x44, 0x28, 0x00
attr 15: ensm_mode value: fdd
attr 8: ensm_mode_available value: sleep wait alert fdd pinctrl pinctrl_fdd_indep
attr 9: multichip_sync ERROR: Input/output error (-5)
attr 10: rssi_gain_step_error value: lna_error: 0 0 0 0 mixer_error: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
gain_step_calib_reg_val: 0 0 0 0 0

iio:device3: xadc
9 channels found:

temp0: (input)
3 channel-specific attributes found:
attr 0: scale value: 123.040771484
attr 1: offset value: -2219
attr 2: raw value: 2494

voltage1: vccaux (input)
2 channel-specific attributes found:
attr 0: scale value: 0.732421875
attr 1: raw value: 2447

voltage2: vccbram (input)
2 channel-specific attributes found:
attr 0: scale value: 0.732421875
attr 1: raw value: 1359

voltage3: vccpint (input)
2 channel-specific attributes found:
attr 0: scale value: 0.732421875
attr 1: raw value: 1353

voltage4: vccpaux (input)
2 channel-specific attributes found:
attr 0: scale value: 0.732421875
attr 1: raw value: 2452

voltage5: vccoddr (input)
2 channel-specific attributes found:
attr 0: scale value: 0.732421875
attr 1: raw value: 2033

voltage6: vrefp (input)
2 channel-specific attributes found:
attr 0: raw value: 1697
attr 1: scale value: 0.732421875

voltage7: vrefn (input)
2 channel-specific attributes found:
attr 0: scale value: 0.732421875
attr 1: raw value: -3
1 device-specific attributes found:
attr 0: sampling_frequency value: 961538

iio:device4: cf-ad9361-dds-core-lpc (buffer capable)
16 channels found:

voltage0: (output, index: 0, format: le:S16/16¿¿0)
3 channel-specific attributes found:
attr 0: calibscale value: 1.000000
attr 1: calibphase value: 0.000000
attr 2: sampling_frequency value: 32000000

voltage1: (output, index: 0, format: le:S16/16¿¿0)
3 channel-specific attributes found:
attr 0: calibphase value: 0.000000
attr 1: calibscale value: 1.000000
attr 2: sampling_frequency value: 32000000

voltage2: (output, index: 0, format: le:S16/16¿¿0)
3 channel-specific attributes found:
attr 0: calibphase value: 0.000000
attr 1: calibscale value: 1.000000
attr 2: sampling_frequency value: 32000000

voltage3: (output, index: 0, format: le:S16/16¿¿0)
3 channel-specific attributes found:
attr 0: calibphase value: 0.000000
attr 1: calibscale value: 1.000000
attr 2: sampling_frequency value: 32000000

voltage4: (output, index: 0, format: le:S16/16¿¿0)
1 channel-specific attributes found:
attr 0: sampling_frequency value: 32000000

voltage5: (output, index: 0, format: le:S16/16¿¿0)
1 channel-specific attributes found:
attr 0: sampling_frequency value: 32000000

voltage6: (output, index: 0, format: le:S16/16¿¿0)

1 channel-specific attributes found:
attr 0: sampling_frequency value: 32000000

voltage7: (output, index: 0, format: le:S16/16¿¿0)
1 channel-specific attributes found:
attr 0: sampling_frequency value: 32000000

altvoltage0: TX1_I_F1 (output)
5 channel-specific attributes found:
attr 0: phase value: 90000
attr 1: scale value: 0.251160
attr 2: frequency value: 4999588
attr 3: raw value: 1
attr 4: sampling_frequency value: 32000000

altvoltage1: TX1_I_F2 (output)
5 channel-specific attributes found:
attr 0: phase value: 90000
attr 1: scale value: 0.000000
attr 2: raw value: 1
attr 3: frequency value: 4999588
attr 4: sampling_frequency value: 32000000

altvoltage2: TX1_Q_F1 (output)
5 channel-specific attributes found:
attr 0: raw value: 1
attr 1: phase value: 0
attr 2: frequency value: 4999588
attr 3: scale value: 0.251160
attr 4: sampling_frequency value: 32000000

altvoltage3: TX1_Q_F2 (output)
5 channel-specific attributes found:
attr 0: raw value: 1
attr 1: phase value: 0
attr 2: frequency value: 4999588
attr 3: scale value: 0.000000
attr 4: sampling_frequency value: 32000000

altvoltage4: TX2_I_F1 (output)
5 channel-specific attributes found:
attr 0: frequency value: 999526
attr 1: phase value: 90000
attr 2: raw value: 1
attr 3: scale value: 0.251160
attr 4: sampling_frequency value: 32000000

altvoltage5: TX2_I_F2 (output)

5 channel-specific attributes found:
attr 0: raw value: 1
attr 1: frequency value: 999526
attr 2: phase value: 90000
attr 3: scale value: 0.000000
attr 4: sampling_frequency value: 32000000

altvoltage6: TX2_Q_F1 (output)
5 channel-specific attributes found:
attr 0: phase value: 0
attr 1: scale value: 0.251160
attr 2: frequency value: 999526
attr 3: raw value: 1
attr 4: sampling_frequency value: 32000000

altvoltage7: TX2_Q_F2 (output)
5 channel-specific attributes found:
attr 0: raw value: 1
attr 1: phase value: 0
attr 2: scale value: 0.000000
attr 3: frequency value: 999526
attr 4: sampling_frequency value: 32000000
2 buffer-specific attributes found:
attr 0: watermark value: 2048
attr 1: data_available value: 0

iio:device5: cf-ad9361-dds-core-B
12 channels found:

voltage0: (output)
3 channel-specific attributes found:
attr 0: calibscale value: 1.000000
attr 1: calibphase value: 0.000000
attr 2: sampling_frequency value: 32000000

voltage1: (output)
3 channel-specific attributes found:
attr 0: calibphase value: 0.000000
attr 1: calibscale value: 1.000000
attr 2: sampling_frequency value: 32000000

voltage2: (output)
3 channel-specific attributes found:
attr 0: calibphase value: 0.000000
attr 1: calibscale value: 1.000000
attr 2: sampling_frequency value: 32000000

voltage3: (output)

3 channel-specific attributes found:
attr 0: calibphase value: 0.000000
attr 1: calibscale value: 1.000000
attr 2: sampling_frequency value: 32000000

altvoltage0: TX1_I_F1 (output)
5 channel-specific attributes found:
attr 0: phase value: 90000
attr 1: scale value: 0.251160
attr 2: frequency value: 4999588
attr 3: raw value: 1
attr 4: sampling_frequency value: 32000000

altvoltage1: TX1_I_F2 (output)
5 channel-specific attributes found:
attr 0: phase value: 90000
attr 1: scale value: 0.000000
attr 2: raw value: 1
attr 3: frequency value: 4999588
attr 4: sampling_frequency value: 32000000

altvoltage2: TX1_Q_F1 (output)
5 channel-specific attributes found:
attr 0: raw value: 1
attr 1: phase value: 0
attr 2: frequency value: 4999588
attr 3: scale value: 0.251160
attr 4: sampling_frequency value: 32000000

altvoltage3: TX1_Q_F2 (output)
5 channel-specific attributes found:
attr 0: raw value: 1
attr 1: phase value: 0
attr 2: frequency value: 4999588
attr 3: scale value: 0.000000
attr 4: sampling_frequency value: 32000000

altvoltage4: TX2_I_F1 (output)
5 channel-specific attributes found:
attr 0: frequency value: 999526
attr 1: phase value: 90000
attr 2: raw value: 1
attr 3: scale value: 0.251160
attr 4: sampling_frequency value: 32000000

altvoltage5: TX2_I_F2 (output)
5 channel-specific attributes found:
attr 0: raw value: 1

attr 1: frequency value: 999526
attr 2: phase value: 90000
attr 3: scale value: 0.000000
attr 4: sampling_frequency value: 32000000

altvoltage6: TX2_Q_F1 (output)
5 channel-specific attributes found:
attr 0: phase value: 0
attr 1: scale value: 0.251160
attr 2: frequency value: 999526
attr 3: raw value: 1
attr 4: sampling_frequency value: 32000000

altvoltage7: TX2_Q_F2 (output)
5 channel-specific attributes found:
attr 0: raw value: 1
attr 1: phase value: 0
attr 2: scale value: 0.000000
attr 3: frequency value: 999526
attr 4: sampling_frequency value: 32000000

iio:device6: cf-ad9361-A (buffer capable)
8 channels found:

voltage0: (input, index: 0, format: le:S12/16¿¿0)
5 channel-specific attributes found:
attr 0: calibphase value: 0.000000
attr 1: calibbias value: 0
attr 2: calibscale value: 1.000000
attr 3: samples_pps ERROR: No such device (-19)
attr 4: sampling_frequency value: 32000000

voltage1: (input, index: 1, format: le:S12/16¿¿0)
5 channel-specific attributes found:
attr 0: calibbias value: 0
attr 1: calibphase value: 0.000000
attr 2: calibscale value: 1.000000
attr 3: samples_pps ERROR: No such device (-19)
attr 4: sampling_frequency value: 32000000

voltage2: (input, index: 2, format: le:S12/16¿¿0)
5 channel-specific attributes found:
attr 0: calibscale value: 1.000000
attr 1: calibbias value: 0
attr 2: calibphase value: 0.000000
attr 3: samples_pps ERROR: No such device (-19)
attr 4: sampling_frequency value: 32000000

voltage3: (input, index: 3, format: le:S12/16¿¿0)
5 channel-specific attributes found:
attr 0: calibphase value: 0.000000
attr 1: calibscale value: 1.000000
attr 2: calibbias value: 0
attr 3: samples_pps ERROR: No such device (-19)
attr 4: sampling_frequency value: 32000000

voltage4: (input, index: 4, format: le:S12/16¿¿0)
2 channel-specific attributes found:
attr 0: samples_pps ERROR: No such device (-19)
attr 1: sampling_frequency value: 32000000

voltage5: (input, index: 5, format: le:S12/16¿¿0)
2 channel-specific attributes found:
attr 0: samples_pps ERROR: No such device (-19)
attr 1: sampling_frequency value: 32000000

voltage6: (input, index: 6, format: le:S12/16¿¿0)
2 channel-specific attributes found:
attr 0: samples_pps ERROR: No such device (-19)
attr 1: sampling_frequency value: 32000000

voltage7: (input, index: 7, format: le:S12/16¿¿0)
2 channel-specific attributes found:
attr 0: samples_pps ERROR: No such device (-19)
attr 1: sampling_frequency value: 32000000

2 buffer-specific attributes found:
attr 0: watermark value: 2048
attr 1: data_available value: 0

iio:device7: cf-ad9361-B
4 channels found:

voltage0: (input)
5 channel-specific attributes found:
attr 0: calibphase value: 0.000000
attr 1: calibbias value: 0
attr 2: calibscale value: 1.000000
attr 3: samples_pps ERROR: No such device (-19)
attr 4: sampling_frequency value: 32000000

voltage1: (input)
5 channel-specific attributes found:
attr 0: calibbias value: 0
attr 1: calibphase value: 0.000000
attr 2: calibscale value: 1.000000

attr 3: samples_pps ERROR: No such device (-19)
attr 4: sampling_frequency value: 32000000

voltage2: (input)
5 channel-specific attributes found:
attr 0: calibscale value: 1.000000
attr 1: calibbias value: 0
attr 2: calibphase value: 0.000000
attr 3: samples_pps ERROR: No such device (-19)
attr 4: sampling_frequency value: 32000000

voltage3: (input)
5 channel-specific attributes found:
attr 0: calibphase value: 0.000000
attr 1: calibscale value: 1.000000
attr 2: calibbias value: 0
attr 3: samples_pps ERROR: No such device (-19)
attr 4: sampling_frequency value: 32000000

Appendix B

Entire MATLAB code to synchronise the four transmitters

main.m

```
clear
test_ch=load('test_allch.dat')/362/sqrt(2);
fs=30.72e6;Ts = 1/fs;
n_frames = 15;
n_samples = length(test_ch);
% 768 inputchannel = n_samples;
outputchannel = n_samples;

% Inicialització variables:

matriu_zeros = repmat(zeros(n_samples,1),[1 1 n_frames]);
vector_temps = [0:n_frames-1]*n_samples*Ts;

% test_chxy init
test_ch1i = timeseries(matriu_zeros, vector_temps);
test_ch1q = timeseries(matriu_zeros, vector_temps);
test_ch2i = timeseries(matriu_zeros, vector_temps);
test_ch2q = timeseries(matriu_zeros, vector_temps);
test_ch3i = timeseries(matriu_zeros, vector_temps);
test_ch3q = timeseries(matriu_zeros, vector_temps);
test_ch4i = timeseries(matriu_zeros, vector_temps);
test_ch4q = timeseries(matriu_zeros, vector_temps);

%% Simular primer Tx
% Utilitzant dades de test_allch.dat
dades_ch1i = repmat(test_ch(:,1),[1 1 n_frames]);
dades_ch1q = repmat(test_ch(:,2),[1 1 n_frames]);

test_ch1i = timeseries(dades_ch1i, vector_temps);
test_ch1q = timeseries(dades_ch1q, vector_temps);
```

```
sim('qpsk4txrx',Ts*n_samples*n_frames);

RX1_data= RX1.data(:,:,5);

% Calcula cconv, circshift i error de fase mitjà
circ_conv
error_fase

cal_1 = exp(-mean_phase_error*1i); % Factor de cal·libració de la fase
RX1_data_cal = shift_data*cal_1;
amplitude_TX1 = amplitude;

tx1_data = (dades_ch1i + dades_ch1q*1i)*exp(-mean_phase_error*1i);
dades_ch1i_cal = real(tx1_data);
dades_ch1q_cal = imag(tx1_data);
test_ch1i = timeseries(dades_ch1i_cal, vector_temps);
test_ch1q = timeseries(dades_ch1q_cal, vector_temps);

sim('qpsk4txrx',Ts*n_samples*n_frames);
RX1_data_cal_tx1= RX1.data(:,:,5); % Aquí no hi ha error de fase

%% Simular segon Tx
test_ch1i = timeseries(matriu_zeros, vector_temps);
test_ch1q = timeseries(matriu_zeros, vector_temps);
dades_ch2i = repmat(test_ch(:,3),[1 1 n_frames]);
dades_ch2q = repmat(test_ch(:,4),[1 1 n_frames]);
test_ch2i = timeseries(dades_ch2i, vector_temps);
test_ch2q = timeseries(dades_ch2q, vector_temps);

sim('qpsk4txrx',Ts*n_samples*n_frames);

RX1_data= RX1.data(:,:,5);

circ_conv
error_fase

cal_2 = exp(-mean_phase_error*1i);
RX1_data_cal = RX1_data*cal_2;
amplitude_TX2 = amplitude;

tx2_data = (dades_ch2i + dades_ch2q*1i) * exp(-mean_phase_error*1i);
dades_ch2i_cal = real(tx2_data);
dades_ch2q_cal = imag(tx2_data);
test_ch2i = timeseries(dades_ch2i_cal, vector_temps);
test_ch2q = timeseries(dades_ch2q_cal, vector_temps);

sim('qpsk4txrx',Ts*n_samples*n_frames);
RX1_data_cal_tx2= RX1.data(:,:,5); % Aquí no hi ha error de fase
```

```
%% Simular tercer Tx
test_ch2i = timeseries(matriu_zeros, vector_temps);
test_ch2q = timeseries(matriu_zeros, vector_temps);
dades_ch3i = repmat(test_ch(:,5),[1 1 n_frames]);
dades_ch3q = repmat(test_ch(:,6),[1 1 n_frames]);
test_ch3i = timeseries(dades_ch3i, vector_temps);
test_ch3q = timeseries(dades_ch3q, vector_temps);

sim('qpsk4txrx',Ts*n_samples*n_frames);

RX1_data= RX1.data(:,:,5);
%scatterplot(RX1_data);

circ_conv
error_fase

cal_3 = exp(-mean_phase_error*1i);
RX1_data_cal = RX1_data*cal_3;
amplitude_TX3 = amplitude;

tx3_data = (dades_ch3i + dades_ch3q*1i) * exp(-mean_phase_error*1i);
dades_ch3i_cal = real(tx3_data);
dades_ch3q_cal = imag(tx3_data);
test_ch3i = timeseries(dades_ch3i_cal, vector_temps);
test_ch3q = timeseries(dades_ch3q_cal, vector_temps);

sim('qpsk4txrx',Ts*n_samples*n_frames);
RX1_data_cal_tx3= RX1.data(:,:,5); % Aquí no hi ha error de fase

%% Simular quart Tx
test_ch3i = timeseries(matriu_zeros, vector_temps);
test_ch3q = timeseries(matriu_zeros, vector_temps);
dades_ch4i = repmat(test_ch(:,7),[1 1 n_frames]);
dades_ch4q = repmat(test_ch(:,8),[1 1 n_frames]);
test_ch4i = timeseries(dades_ch4i, vector_temps);
test_ch4q = timeseries(dades_ch4q, vector_temps);

sim('qpsk4txrx',Ts*n_samples*n_frames);
RX1_data= RX1.data(:,:,5);

circ_conv
error_fase

cal_4 = exp(-mean_phase_error*1i);
RX1_data_cal = RX1_data*cal_4;
amplitude_TX4 = amplitude;
```

```
tx4_data = (dades_ch4i + dades_ch4q*1i) * exp(-mean_phase_error*1i);
dades_ch4i_cal = real(tx4_data);
dades_ch4q_cal = imag(tx4_data);
test_ch4i = timeseries(dades_ch4i_cal, vector_temps);
test_ch4q = timeseries(dades_ch4q_cal, vector_temps);

sim('qpsk4txrx',Ts*n_samples*n_frames);
RX1_data_cal_tx4= RX1.data(:,:,5); % Aquí no hi ha error de fase

cal = [cal_1 cal_2 cal_3 cal_4];

%% Simular amb tots els TXs calibrats

test_ch1i = timeseries(dades_ch1i_cal, vector_temps);
test_ch1q = timeseries(dades_ch1q_cal, vector_temps);
test_ch2i = timeseries(dades_ch2i_cal, vector_temps);
test_ch2q = timeseries(dades_ch2q_cal, vector_temps);
test_ch3i = timeseries(dades_ch3i_cal, vector_temps);
test_ch3q = timeseries(dades_ch3q_cal, vector_temps);
test_ch4i = timeseries(dades_ch4i_cal, vector_temps);
test_ch4q = timeseries(dades_ch4q_cal, vector_temps);

sim('qpsk4txrx',Ts*n_samples*n_frames);
RX1_data = RX1.data(:,:,5);

%cconv
circ_conv

for (i = 1:1:n_samples)
   amplitude(i) = abs(shift_data(i));
end
amplitude_all_TXs = mean(amplitude(n_zeros:n_samples));

% amplitude_all_TXs hauria de ser la suma de les amplituds de tots
% els transmissors
```

error_fase.m

```
% Càlcul de l'error de fase mitjà per una QPSK

for (i = 1:1:n_samples)
   phase_1(i) = angle(shift_data(i));
   amplitude(i) = abs(shift_data(i));
end

rang_11 = n_zeros+1:n_zeros+((n_samples-n_zeros)/4); % 256:384
rang_01 = n_zeros+((n_samples-n_zeros)/4)+1:n_zeros+((n_samples-n_zeros)/2);
% 385:512
```

```
rang_00 = n_zeros+((n_samples-n_zeros)/2)+1:n_zeros+3/4*(n_samples-n_zeros);
% 513:640
rang_10 = n_zeros+3/4*(n_samples-n_zeros)+1:n_samples; % 640:768

phase_error(1) = mean(phase_1(rang_11) - 5*pi()/4); % 11
phase_error(2) = mean(phase_1(rang_01) - 3*pi()/4); % 01
phase_error(3) = mean(phase_1(rang_00) - pi()/4); % 00
phase_error(4) = mean(phase_1(rang_10) - 7*pi()/4); % 10
amplitude = mean(amplitude(n_zeros:n_samples));

for (j = 1:1:4)
    if (phase_error(j) < -2*pi())
       phase_error(j) = phase_error(j)+2*pi();
    end
end

mean_phase_error = mean(phase_error);
```

circ_conv.m

```
% Calcular cconv i circshift

n_zeros = 256;
ref_conv = zeros(n_samples,1);
ref_conv(1:n_zeros) = 1;
conv_data = cconv(abs(RX1_data), ref_conv, n_samples);
[min_data, pos] = min(conv_data);
shift_data = circshift(RX1_data, -pos+n_zeros);
```

symbol_generation.m

```
%% Random symbol generation with QPSK/16QAM/64QAM/256QAM modulations
n_samples = 1024;

M = 2; % modulation index

if(M==2) % QPSK, Table 7.1.2-1 136.211 v12.9.0 ETSI
    symbols = [-1/sqrt(2), 1/sqrt(2)];

elseif(M==4) % 16QAM, Table 7.1.3-1 136.211 v12.9.0 ETSI
    r = sqrt(10);
    symbols = [-3/r, -1/r, 1/r, 3/r];

elseif(M==8) % 64QAM, Table 7.1.4-1 136.211 v12.9.0 ETSI
    r = sqrt(42);
    symbols = [-7/r, -5/r, -3/r, -1/r, 1/r, 3/r, 5/r, 7/r];
```

```
elseif(M==16) % 256QAM, Table 7.1.5-1 136.211 v12.9.0 ETSI
    r = sqrt(170);
    symbols = [-15/r, -13/r, -11/r, -9/r, -7/r, -5/r, -3/r, -1/r,
1/r, 3/r, 5/r, 7/r, 9/r, 11/r, 13/r, 15/r];

end

pos_i = randi(length(symbols),n_samples,1);
pos_q = randi(length(symbols),n_samples,1);
data_i = symbols(pos_i)'; data_i(1:256) = 0;
data_q = symbols(pos_q)'; data_q(1:256) = 0;
```