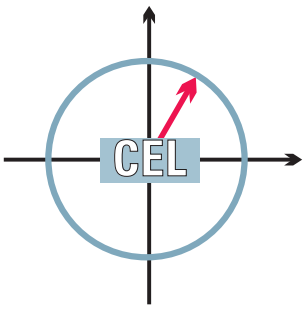
Karlsruhe Institute of Technology
Communications Engineering Lab
Prof. Dr.rer.nat. Friedrich Jondral



Implementation of a Cognitive Radio Handshake Protocol

Master's Thesis

Robert Monje Estrem

Hauptreferent	:	Prof. Dr.rer.nat. Friedrich Jondral
Betreuer	:	Dipl.-Ing. Martin Braun

Beginn	:	06.08.2009
Abgabe	:	08.03.2010

Abstract

In this project, as the title suggest, is presented a handshake protocol for Cognitive Radio. We talk about Cognitive Radio when in a wireless communication, the set-up is able to look at some particulars factors and then, as a result of this, it can decide if the communication would be better if some parameters of the connection were changed. Parameters such as the modulation of the signal, the frequency center, the bandwidth, etc.

This readjust of the parameters will be done automatically and the user will not even notice it. So, the interchange of information will not be interrupted.

To achieve this, the device will be developed as software device and not as hardware. This is called Software-Defined Radio. Cognitive Radio is, in fact, an extension of the Software-Defined Radio.

From all the factors that can be taken into account, the radio frequency spectrum stands out for its bad efficiency. While some ranges of the spectrum can be overloaded, other are rather empty.

To solve this, a Cognitive Radio system has to look at the spectrum and analyse which would be a better frequency to start the communication. When the system is able to do this, it also means that the system must be able to recognize which is this frequency.

What is presented in this project is a simple system of one transmitter and one receiver, which must recognize themselves in the radio frequency spectrum. This is exactly the handshake protocol.

In a real-life example, the transmitter would decide first which is the most appropriate frequency to stablish the connection. After this, the receiver is the one that will have to find which is this frequency. Here is where the handshake protocol starts and when it is over, both devices are able to start the communication.

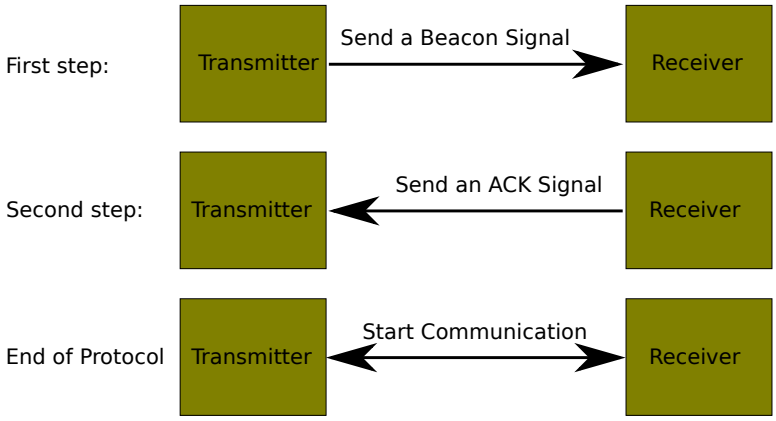


Figure 1: Protocol

The protocol will be the one shown in the figure 1.

For the implementation of the code, it has been used the GNU Radio software, which was designed on purpose for Software-Defined Radio, then it makes this tool very useful for the aim of this project.

The GNU Radio works in blocks developed with C++. Each block has its signal processing function, such as a Fast Fourier Transformation (FFT) or a modulation process.

To connect this with the radio frequency spectrum is used the USRP. The USRP is a device which its main function is to transmit or receive the information from the RF and convert it for the right processing in the GNU Radio. So it is composed by a DAC (Digital-to-Analog Converter), a ADC (Analog-to-Digital Converter) and a device the transmit and receive this information.

In the figure 2 can be seen how the GNU Radio is connected to the USRP. As can be observed one USRP can work as a transmitter and a receiver at the same time.

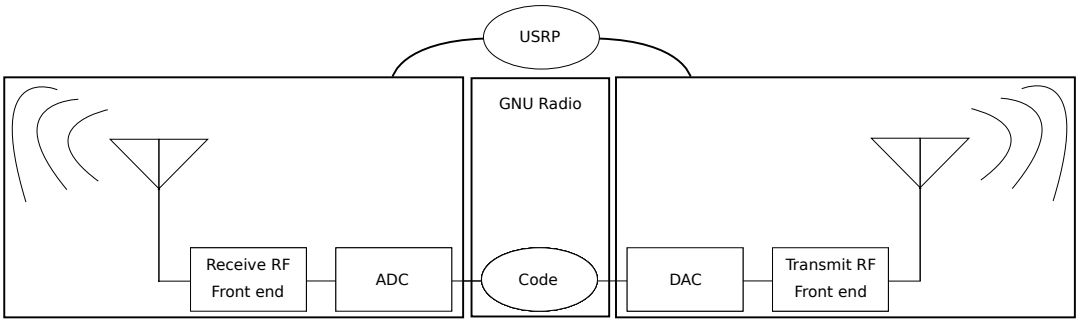


Figure 2: USRP

The transmitter and the receiver will, both, need to be used as a transmitter and

a receiver. The transmitter will transmit the beacon signal and receive the ACK signal while the receiver will receive the beacon signal and transmit the ACK signal.

First of all, both devices must agree in a sequence for the beacon signal and another one for the ACK signal. This sequence will be the one that permits to know them if the signal received is the one they are waiting for.

Once the system is ready to start, the transmitter will send the beacon signal at the frequency it has previously chosen. The transmitter will send the sequence in packets. These packets will be sent in GMSK modulation.

Is used GMSK modulation because, it provides a good spectrum efficiency in wireless transmission systems, so when interchanging the information between the transmitter and receiver it will be able to send more data than others modulation. Another reason is that it has constant envelope which will permit to transmit the signal at higher power and so, to be detected for the other device easily. The relatively narrow bandwidth and the coherent detection capability are others good characteristics of the GMSK modulation.

Then, using the possibilities of GNU Radio, the sequence will be sent packet by packet and modulated in GMSK as it has been explained before. The receiver, waiting for the signal, will first detect any signal that comes to it.

To detect the signal is used a energy detector as is shown in the figure 3

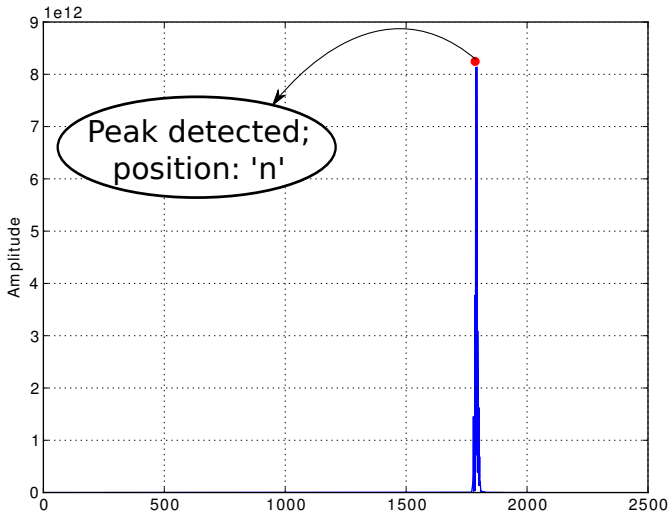


Figure 3: Energy detector: peak detected

But, this protocol needs to get the frequency with a higher precision. So, to detect the frequency with good precision the transmitter change its decimation in order to get focused in the frequency point, getting at the end to the frequency with a tolerance of just 5 - 10 kHz.

To make this focus useful it will be also necessary to re-tune the USRP to the frequency detected. This has to be done every time that is changing the decimation.

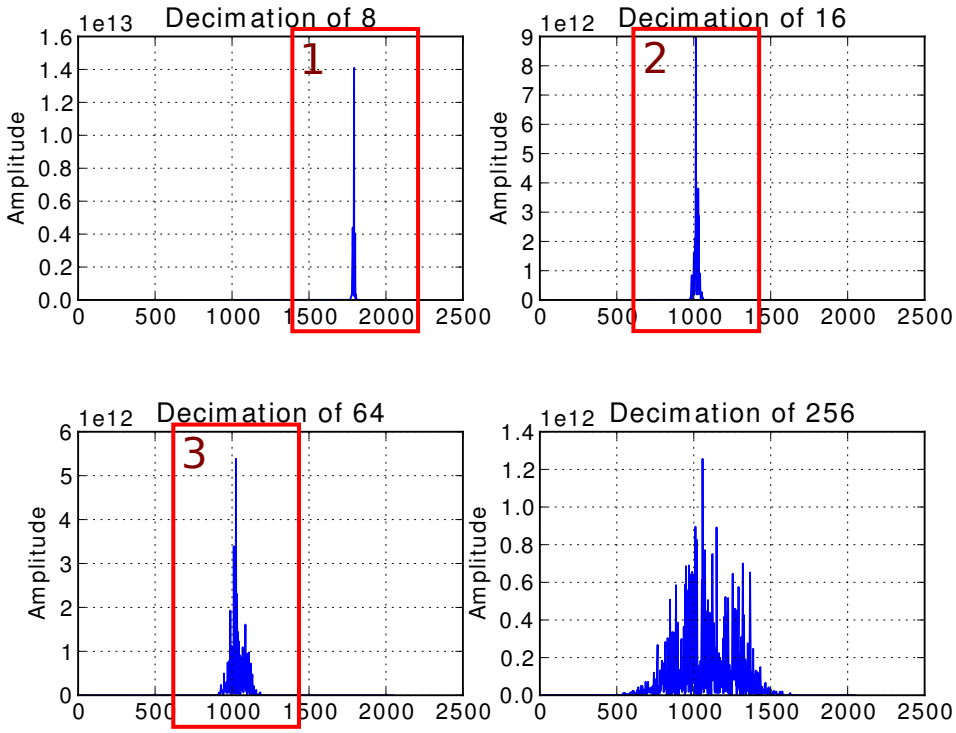


Figure 4: Zoom to the frequency detected

As is shown in the figure 4, the last plot of the signal shows clearly the envelope of a GMSK, and its center frequency is centered at the middle of the FFT size. Which is good as the frequency should be around the frequency detected in the previous detection.

The change of decimation is done 4 times because passing from a big range of scanning to a little one just in one step, is not a good idea as in the first step the resolution is the order of 5 kHz and sometimes could result that the range of the new decimation is so small that the frequency is really out of the range.

Once this signal is detected the receiver will have to check if the sequence of the signal is the right one. To make this it just need to compare the packets received with the original sequence, takin into account the it will probably be delayed as shown in the figure 5.3.

If the result of this is that the sequence is the correct one, the receiver will send the ACK signal to the transmitter. With this, the receiver will be ready to start the transmission of the information.

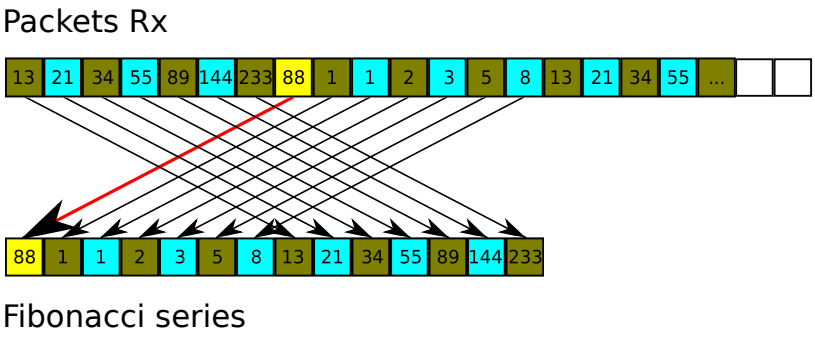


Figure 5: Checking of the packets

Back in the transmitter, the protocol will be the same, receive the ACK signal and demodulate the packets to compare them to the original sequence of the ACK signal.

If this is correct, the transmitter is ready for the data transmission too.

In the project can be found the whole description of how the Tx and the Rx are implemented with the GNU Radio blocks and also the code of how is the signal detected.

There are also some section which talk about the possible problems of the protocol. All of them with the own solution.

Contents

List of Figures

Abbreviations

ACK	Acknowledgement
ADC	Analog-to-Digital Converter
CR	Cognitive Radio
DAC	Digital to Analogue Converter
DBPSK	Differential Binary Phase Shift Keying
DDC	Digital Down Converter
DQPSK	Differential Quadrature Phase Shift Keying
DUC	Digital Up Converter
FFT	Fast Fourier Transform
FPGA	Field-Programmable Gate Array
GMSK	Gaussian Minimum Shift Keying
GSM	Global System for Mobile Communications
RF	Radio Frequency
Rx	Receive
SDR	Software-Defined Radio
Tx	Transmit
USRP	Universal Software Radio Peripheral

Acknowledgements

I would like to thank Dipl.-Ing. Martin Braun, for the dedication and the support given during this time in the department.

And also thanks to all the friends who have shared with me this experience abroad, the friends in Reus, Barcelona and Karlsruhe and my family for their support.

Chapter 1

Introduction

The world of information evolves really fast. Anyone who does not inform itself about news technologies and its innovations or renovations will be out of the field in a very short time. These last decades the communication topics have been changing in a revolutionary way.

Internet as the main new technology, and as a great invention. Internet gives to the people a high-speed communication to any kind of information. In other words, since then, people was able to be in communication with all the world.

On the other hand, wireless technology has been also evolving. Due to mobility, which is one of the most important characteristics in the modern society. Wireless technology gives to the people the ability to get information while they are moving.

This wireless technology and its actual use are growing up. This makes an increase in mobile services and so an increase in the demand for mobile technology. The mobile cellular is now a lifestyle, not just in a business way but also avoiding the problem of time and distance.

Since this happened, there exists plenty investigations in wireless communications, and a lot of them focused in developing intelligent radios. A good example is when somebody is calling with his mobile phone and as it is moving, the signal from the terminal used to stablish the communication gets too far. The phone will readjust this by switching the connection to another terminal.

This example is eye-catching. Another one could be the design of reconfigurable receivers the recognize the modulation of the signal automatically. And that last one gives another view to the developing of the devices.

As said before, the more use in wireless communication the more demand of new services. And these news services forces the technology to innovate. This evolution is going so fast that what now could be defined in advanced technology quickly will become common to everybody. And this has happened these last years in every new technology that is invented.

Getting back to the mobility, this issue is not just with cellular phones, it can be applied to many systems. And not just with a switch from one terminal to another one but in with characteristics. And nowadays, these systems are required to readjust their parameters constantly and automatically.

To achieve this state, where the system, can manage itself in purpose to keep the communication, the devices that compose the system, they must be able to know about the state of the connection, and not just the connection but information about how to improve or at least to maintain it.

In wireless communications it is known that the RF spectrum is bad shared. This fact leads to some kind of new job that the system will have to do. The search of new frequencies to realize the communication.

In this project is attacked this issue. The project presents a system which is able to start the communication in a random frequency. For this, previously, the system would have decided which is the best frequency to stablish the connection. For the implementation of the system is used Cognitive Radio.

Chapter 2

Cognitive Radio

The definition of Cognitive Radio can be expressed in several ways. To give a general one, it could be defined as 'the ability of a network to change its parameters, either in transmission or reception, to get a communication in a more efficient way'

The practical meaning of the Cognitive Radio is to change the communication network in order to not overload it. To give an example, imagine that you are listening to a radio station (on a specific frequency), then other transmitters interfere with your reception. In a non-cognitive radio case, the receiver would not make any action, so you would just have to wait until it gets out of the interference. But, with the Cognitive Radio case it would respond by switching automatically to an open backup frequency that carries your station's broadcast. If this example is taken as a cell-phone user, which is trying to do an emergency call; the call may be cut off and you would have to re-call again. The Cognitive Radio, in this particular example will really help as you would not have to re-call again; and in case the emergency call is really serious, it would even save a life.

2.1 Introduction

Cognitive Radio first appearance was in an article wrote by Joseph Mitola III and Gerald Q. Maguire, Jr in 1999. After this, Mitola described this advance in wireless communications with the following words:

"The point in which wireless personal digital assistants (PDAs) and the related networks are sufficiently computationally intelligent about radio resources and related computer-to-computer communications to detect user communications needs as a function of use context, and to provide radio resources and wireless services most appropriate to those needs."[MM99]

Software-Defined Radio

The idea of Cognitive Radio came from the Software-Defined Radio (SDR). A good explanation of the SDR systems, how Cognitive Radio comes from these systems and its evolution can be found in [Jon05]. To describe SDR, first has to be noticed that originally, the radio, understood as any device which receive or transmit signals through wireless connection in purpose to transfer information, it was based just in hardware. It means that it just could be modified physically. Which this, can be translated into a bigger cost in the production and a low flexibility. In SDR, instead, it's used devices working under the command of software, in which case it won't need to replace machinery to readjust the system.

Figure 2.1: Software-Defined Radio

2.2 Types

There are plenty parameters that can be taken into account to take the decision of reconfiguring the set up of the system. Depending on which parameters are being used, Cognitive Radio can be divided in, mainly, two sorts:

- **Full Cognitive Radio** (sometimes known as 'Mitola Radio') which is when every parameter observable is taken into account.
- **Spectrum Sensing Cognitive Radio**, in this case, as its names suggests, the software just pay attention to the radio frequency spectrum.

Another distinction can be done; this division will be done depending on the parts of the spectrum available.

- **Licensed Band Cognitive Radio** is the kind of radio that is allowed to use bands assigned to licensed users, besides the unlicensed ones.
- **Unlicensed Band Cognitive Radio** can only make use of the unlicensed bands of the spectrum.

2.3 Frequency Spectrum

Operating frequency, power output, antenna orientation and beam-width, modulation, and transmitter bandwidth are a few of the parameters that the Cognitive Radio system can adjust automatically in order to solve all kind of problems that could appear in a communication between receiver and transmitter. That is why, at first, Cognitive Radio was supposed to be a Software-Defined Radio extension. However, from all these variables, the one that has the less efficiency in the way which is used is the RF spectrum. In fact, the major investigation work in Cognitive Radio is centred in Spectrum Sensing Cognitive Radio [CMB04].

The Federal Communications Commission in the United States, and its counterparts around the world, allocate the radio spectrum in swaths of frequency of varying widths [SW04]. The different bands covers AM radio VHF television, cell phones, and so on. But not every channel in every band is used the whole time. Some statistics independent from the FCC says that in some points during the day, the use of the spectrum does not exceed 30% [MMBF+02]. In a case like this, it would be really ridiculous to get a bad communication because of an interference while in other frequencies there is nobody transmitting. Here is where Cognitive Radio would help with its automatically parameters changes [Sav06].

From now on, with the problem detected, we need to get the solution. And the Cognitive Radio has it. That's why the principals functions are:

- **Spectrum detection:** of course, if we are looking for a more effective distribution in the RF spectrum, the first to do, is to find out the location of the less occupied ranges. This can be done through different methods such as:
  - Interchanging information between users to know about the situation of the spectrum
  - Interferences detection, so that will give information about the using in this range
  - The ability of detecting if other main signals are in a specific spectrum
- **Spectrum management:** use the spectral bandwidth which is more suitable with the Quality of Service (QoS) required for the user over all the available ones. Here two parts can be distinguished:
  - Spectrum analysis; identify the characteristics of every available band to know which are the advantages and the drawbacks of using it (i.e. delays, bit error...)
  - Spectrum decision; due to the first part, it's time to compare and contrast every one of them to choose the best option

After these two main points, what a good Cognitive Radio must also achieve is the ability of changing the frequency automatically and giving no problems to the user. Another good point to get for would be the fair distribution in the spectrum taking into account all the users [Hay05].

Chapter 3

Tools

In order to make a good contribution within the cognitive radio world, it is necessary to get some suitable tools to proceed into the work. As has been said before, cognitive radio is a kind of software-defined radio, then, one of the tools that needs to be chosen is the software to use.

3.1 GNU Radio

GNU Radio is a software that was developed by Eric Blossom on purpose to be used in SDR systems. One of the points about it that makes GNU Radio work is that is an open software and it's released under the GNU General Public License. The GNU General Public License (GNU GPL or simply GPL) is a used free software license. It helps to the free software definition, and preserve its own development [Blo01].

And not only GNU Radio is free but all the equipment it might be needed on the work; for example, Python, C++ and Linux. Nearly all the GNU Radio applications are written in Python, which is a relatively easy to manage and quite useful high-level programming language. All the functions and a detailed explanation of how Python works can be found in [Doc]. So, it means that although it could seem, as it is a really huge and useful software, GNU Radio has not big hardware requirements but rather the opposite, with usual computers can work in a very worthy (although USRP it's needed). Another good reason related to the money one is that it does not need, in the most condition, expensive RF test machine.

With GNU Radio can be achieved a good flexibility in both software and hardware. In software, as it's an SDR, can be reconfigured for many other modulation methods and,

obviously, just changing the algorithms can be an improve in the quality of the system. In the hardware side, as it will be shown afterwards with the USRP, can select both Rx or Tx methods and retune the frequency [ZKY06].

The architecture of GNU Radio, as is shown in figure 3.1, consists of blocks. These are made in C++.

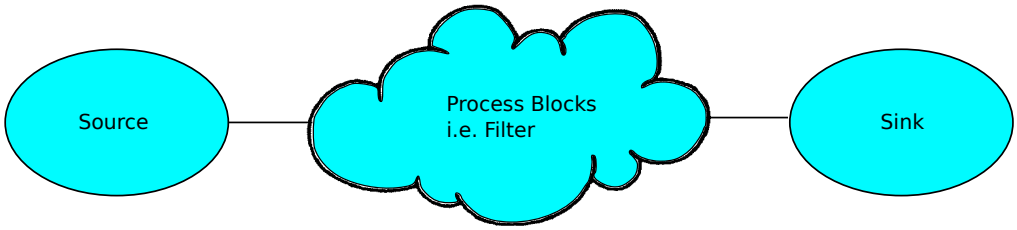


Figure 3.1: Architecture of the GNU Radio

The vertices are signal processing blocks and the edges represent the data flow between them. Every flow graph must have at least one source and one sink to work. With this simple way to use it, it looks at first glance that the reconfigurability can be done with just connecting different blocks to another instead of using multiple expensive radios [Pro]. GNU Radio is a powerful signal processing software. There are plenty of blocks to use, and what is more, create new signal processing blocks it is not a tough job. Each block works independently, and as can be seen, GNU Radio is the perfect substitute for the hardware.

With the software it's easier to achieve the purpose. And here is where GNU Radio gets its own goal, which is to give to the people the ability to work easily with the frequency spectrum.

For example, in figure 3.2, is shown a simple program which takes the radio frequency spectrum, then pass it through a low pass filter (LPF) and finally save the result in a file.

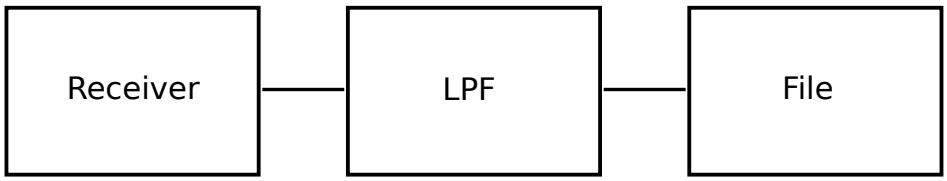


Figure 3.2: Low Pass Filter

In this case, the source is the receiver of the spectrum and the sink is the file where is saved the information.

But, in the GNU Radio's world, not everything is free. The only cost it need to be paid is the knowledge one. If something's wrong with this software is that there is a

large knowledge involved. In the software environment it's needed to work with Python and C++ and their libraries. The Linux and its supports packages and the GNU Radio architecture. And as it's working in communications, it's also needed to posses digital communications, wireless communications and DSP knowledge. As it might be used some FPGA and Assembly language.

3.2 USRP

In SDR, as it is used the software instead hardware, it will be also necessary to use digital information instead of the analogue one. So, it is going to be necessary a digital to analogue converter (DAC). The device that is used in GNU Radio is the Universal Software Radio Peripheral (USRP).

The USRP is developed by a team led by Matt Ettus [LLCa]. Although it has an open design, and because of this, everybody can work on it.

The USRP is connected to the computer through USB port and it's composed basically by the daughterboard, which is an implementation of RF front ends, and the USRP itself, which can be called as motherboard.

In the figure 3.3 is shown the receive path of the USRP and in the figure 3.4 the transmit path.

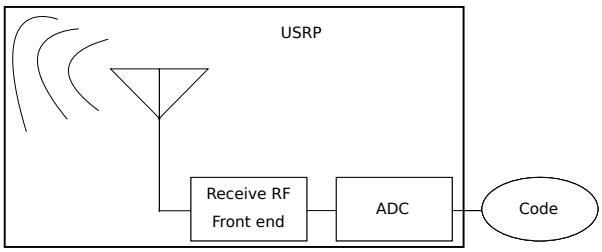


Figure 3.3: Receive path

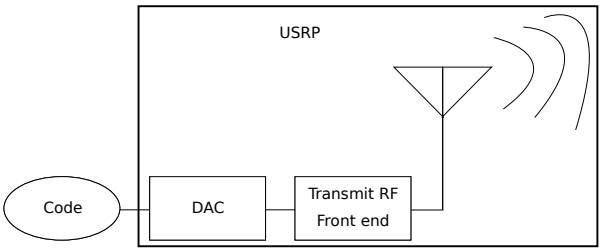


Figure 3.4: Transmit path

There are plenty different daughterboards, each one for different frequency bands. The range these daughterboards can afford is from DC to approximately 5.9 GHz. Depending on which daughterboard is used, its characteristics change a little, besides, of course, the frequency band [Ham08].

The USRP can be used such as a Tx or a Rx, so it means that is a DAC and a ADC at the same time. Another good point of the USRP is its price, which in a relative way, for all the advantages that are given with it, is really affordable; not more than 1000$

Inside the motherboard as shown in the figure 3.5, can be found 3 main parts; the USB controller, the FPGA and the DAC or the ADC [Blo01; Pro; ZKY06].

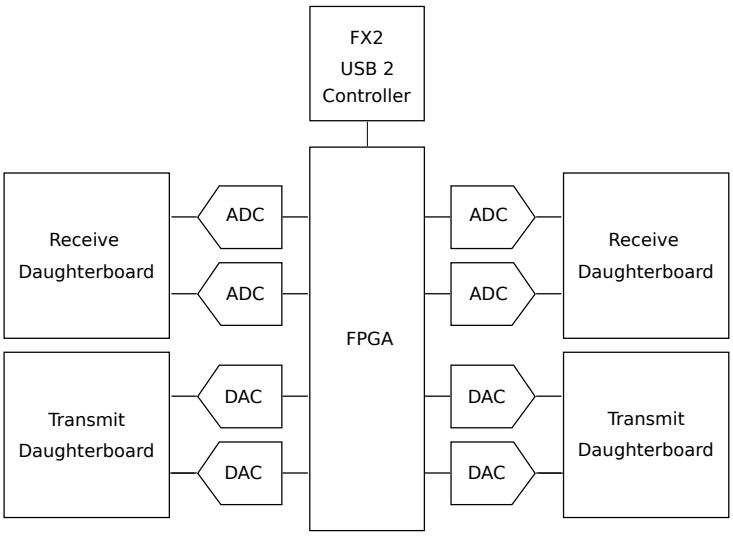


Figure 3.5: Architecture of a USRP

- **USB**
  - Support USB 2.0, but not good for USB 1.x
  - Support 32 MBps across the USB
  - Samples sent are in 16-bit signed integers in IQ format (16-bit I and 16-bit Q complex data)
- **FPGA**
  - Includes digital down converters (DDC) implemented with cascaded integrator-comb (CIC) filters. A DDC converts the signal from the IF band to the base-band.
  - Includes digital up converters (DUC), which are in the AD9862 CODEC chips, not in the FPGA
  - Decimation and interpolation

- **DAC/ADC**
  - **DAC** four 14-bit 128 Mega-samples per second, 83 dB SFDR
  - **ADC** four 12-bit 64 Mega-samples per second, 85 dB SFDR

In 2008 it also has been invented USRP2, which, although it seems, because of the name, that it is a substitute of the original USRP, it is not. USRP2 has others features and the FPGA is not compatible [LLCb]. Although USRP is not the only device that can be used with GNU Radio, nearly all the people who works on it use it.

Investigation projects

The USRP has been used as [Wik]:

- An APCO25 compatible Transmitter/Receiver and Decoder
- RFID reader
- Testing equipment
- A cellular GSM base station
- A GPS receiver
- An FM radio receiver
- An FM radio transmitter
- A digital television decoder
- Passive radar
- Synthetic aperture radar
- An amateur radio
- A teaching aid
- Digital Audio Broadcasting (DAB/DAB+/DMB) transmitter

Chapter 4

Goals

As explained before, the cognitive radio helps in the communication process. Due to that it can look at the parameters of the connection and evaluate if there is a better way to keep up with the communication.

4.1 Introduction

As is known, the radio frequency spectrum is one of the actual problems in telecommunications due to it's not well used. The point is not that the division of the spectrum is wrong, the main problem is that each division is not managed as good as could be.

This project tries to help a little into these kind of communication systems. For example, when a system is done in order to transmit in a unique frequency and by the time that communication wants to start, this frequency is occupied for another connection, or let's just say that there are some interferences. Normally, the system would wait until it gets free or maybe start the communication but with a non-usual high bit error probability.

What if in another frequency, in that moment, there is no transmission, or it's less overload. Thanks to cognitive radio, a system can look at its parameters and decide which frequency could be better for the connection. Of course, cognitive radio is always aware if the frequency range is a licensed band or not and if it is possible to use it.

The situation can be solved if the transmitter and the receiver can change the frequency of the communication. If the system could retune automatically its antenna into a frequency that the system itself thinks it's more appropriate because it seems to be free in that moment or, at least, seems to have more stability, the users of the system could start the connection without even taking care of the problem.

4.2 Handshake Protocol

In this project it is presented a particular handshake protocol between a transmitter and a receiver. In [BEJ09] is explained why this protocol is used in CR and is detailed how is done the terminal synchronisation.

Previously, the transmitter should have found which is the appropriate frequency for the system connection. Once the transmitter knows the frequency that is going to use, it sends a sequence of packets at this frequency. The information that is sent must be a sequence known by both, transmitter and receiver.

Then, the receiver, gets the signal and checks if the sequence is the correct one. When this happens, the receiver will know the frequency that the transmitter has chosen. After knowing the frequency, the receiver will send an acknowledge signal (ACK) to the transmitter, in the same frequency.

When the transmitter, which now is converted in a receiver, gets the ACK then the connection is ready to start at the frequency that the transmitter wants.

- **TX:**
  - Send the Beacon Signal
  - Wait for ACK Signal
- **RX:**
  - Scan the Radio Frequency Spectrum
  - Get the Signal and so the Frequency at which is Received
  - Check if the Packets are the expected Sequence
  - Send an ACK at the same Frequency

Those are the main steps that the transmitter and the receiver will have to do to get the protocol done.

4.2.1 Transmitter and Receiver

As the transmitter will not know if the receiver is waiting for the beacon signal, and vice versa, the receiver will not know when the transmitter will start to send the packets; both devices will have to wait indefinitely for this to happen.

On the Tx side, the transmitter just have to start the connection, whenever the user decides. But it also has to wait for the ACK from the receiver. So, in this case, there will be two states; send the beacon signal and wait for the ACK. Once the ACK signal has been received then the information exchange can start.

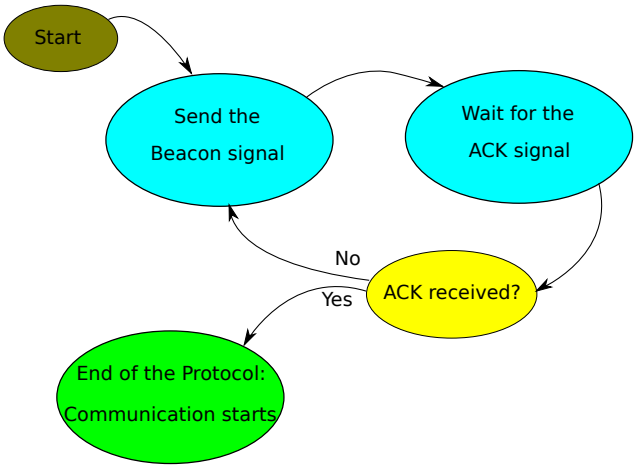


Figure 4.1: Transmitter States Diagram

On the Rx side has not the problem of change its state from receiver to transmitter all the time, as it has only to wait for the beacon signal, and then, when it detects it, just send back the ACK signal. The problem in here is that there might be some other signals which the receiver will detect and check if these are the packets he is waiting for. In the case that they are not the right ones, the receiver goes to the waiting status again until it gets the correct signal. When this beacon signal is found, then the receiver exits the waiting status and sends the ACK, which it means that its job is over.

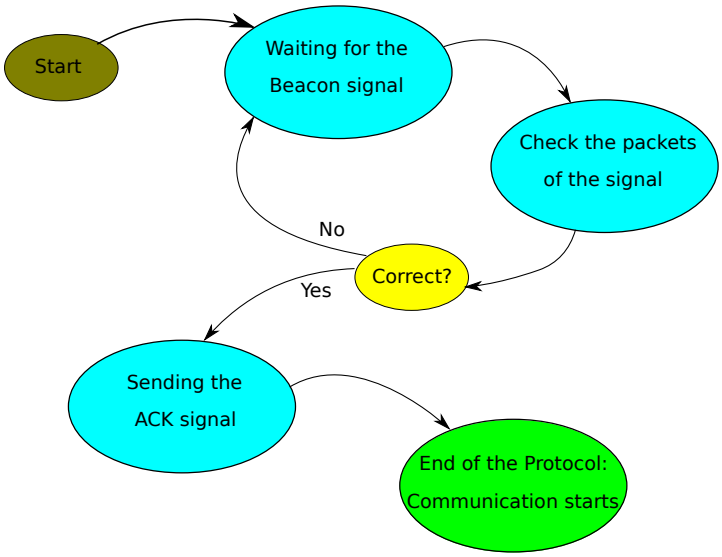


Figure 4.2: Receptor States Diagram

4.2.2 Synchronization

Another point to get into account is the time. On one side, there is the transmitter which is sending the beacon signal and then, waiting for the ACK signal On the other, the receiver is scanning the radio frequency spectrum.

In a perfect world, the time diagram of the protocol would be like is shown in the figure 4.3.

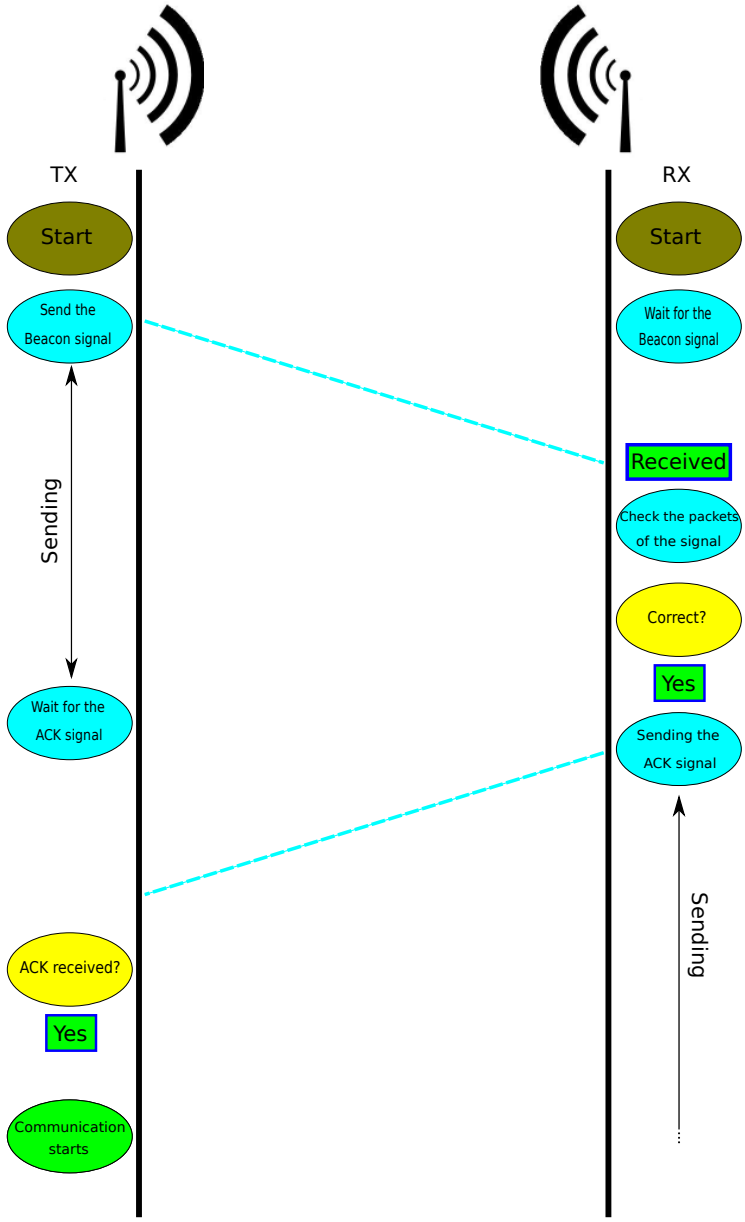


Figure 4.3: Time Diagram

But what would the transmitter do if it gets no answer from the receiver. And this is not just in case it has no time to get the ACK signal but, for example, when the receiver device is not started.

The transmitter will repeat its job until something is received.

In the figure 4.4, the ACK signal is received at the second time.

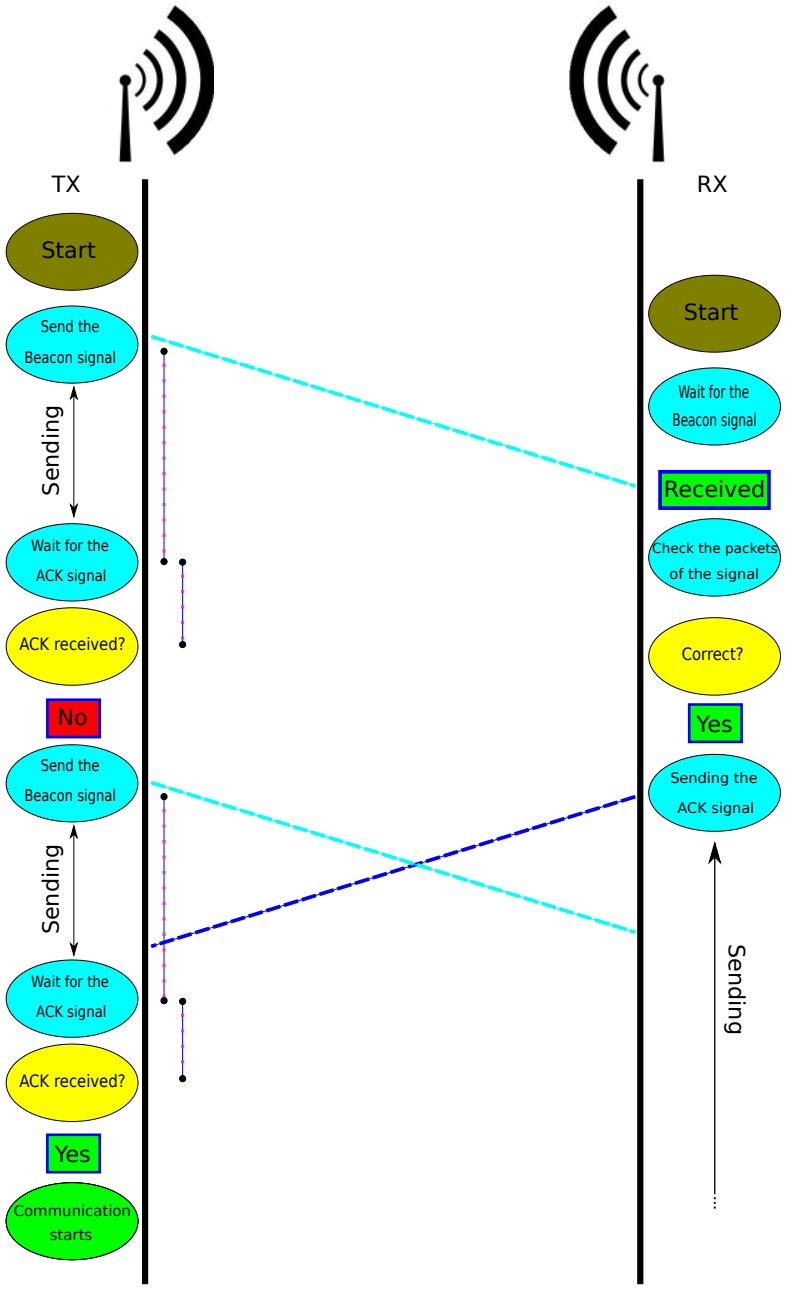


Figure 4.4: Time Diagram: when no ACK from the receiver

Another problem could be if the receiver is started, but it has been done with some delay. So, there is the possibility that the receiver, during the scanning, receives the signal correctly, but then, when it goes to check if the packets are the correct ones, the transmitter turns to be in the waiting state.

As shown in the figure 4.5, it takes 2 times to the receiver to get the signal and check the packets.

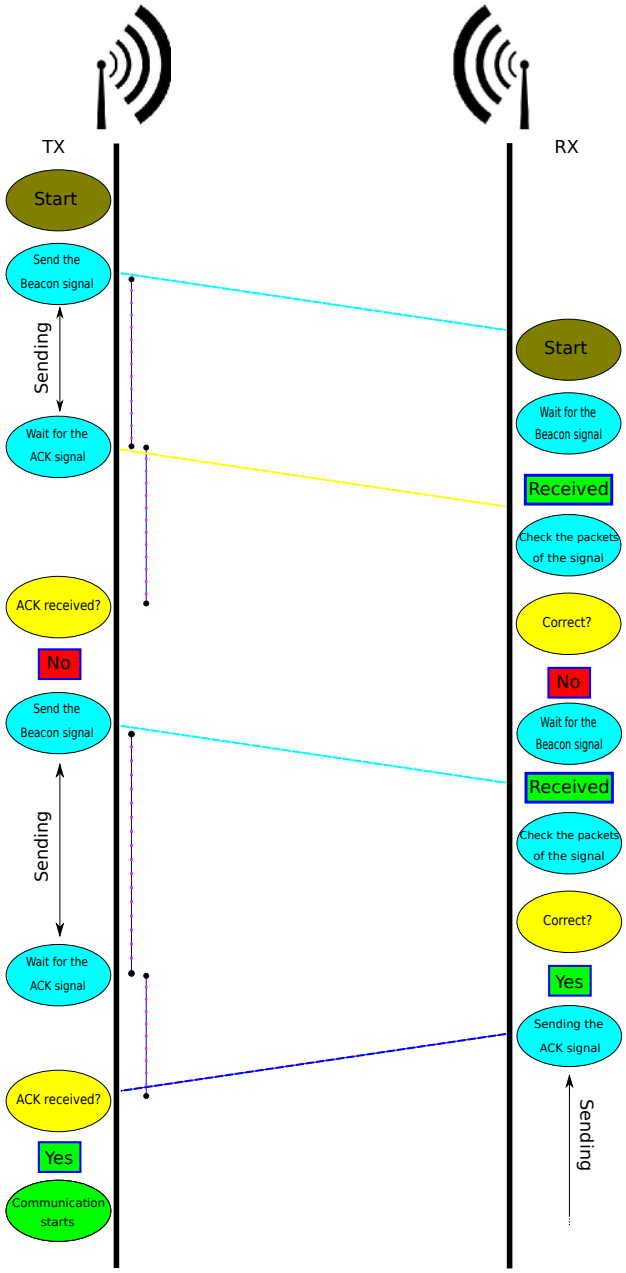


Figure 4.5: Time Diagram: when the receiver has no time to check the packets

But the real problem in synchronization, is when the time of the *Send the Beacon Signal* is not long enough for the receiver to get the signal, get the frequency of it and check the packets. Then, there is a faint possibility that the receiver never gets out of the receiving state.

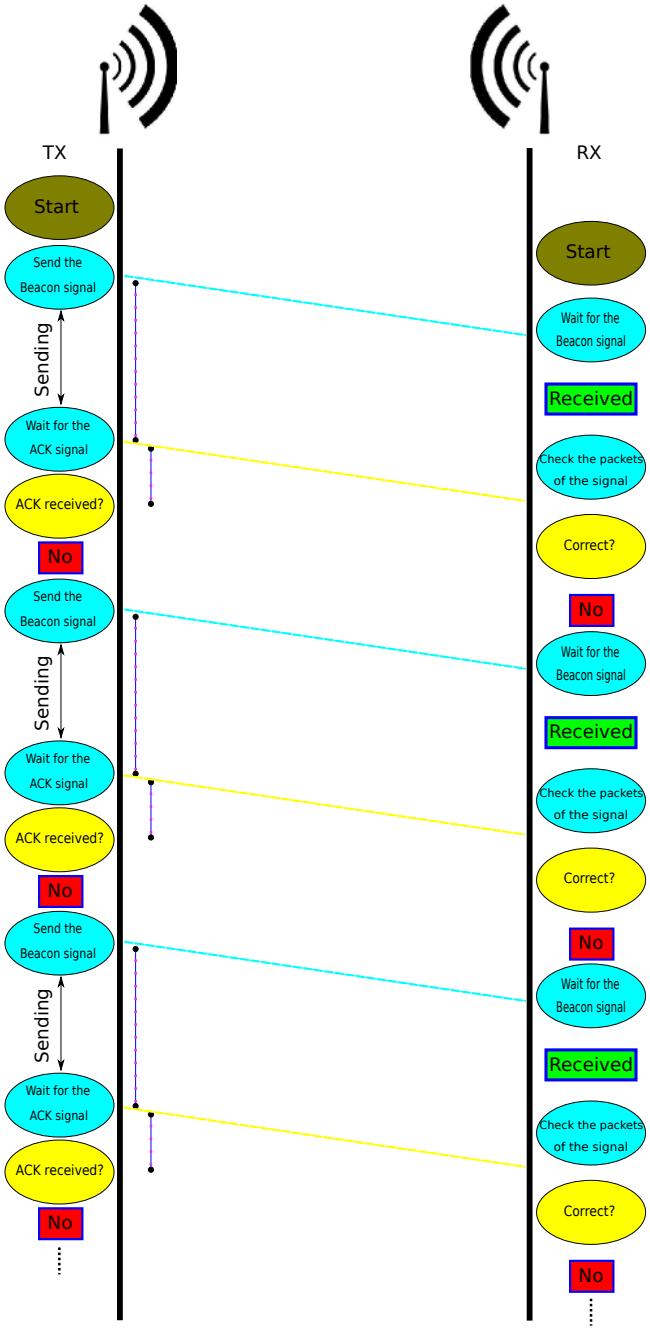


Figure 4.6: Time Diagram: worst case

Chapter 5

Implementation

The implementation of the handshake protocol is the main chapter in this project. After getting all the information from Cognitive Radio, GNU Radio, Python, USRP, etc... and learning the way to use it, is time to start everything up.

5.1 Introduction

In this section are described the steps that have been done to get the final code for the protocol. The aim of the protocol is that both devices recognize themselves in the radio frequency spectrum. One, the transmitter, will decide which frequency wants to transmit. On the other hand, there is the receiver, which will have to find the frequency that the transmitter has chosen. When that last one has found the frequency, the system will be ready to transmit the information.

First of all, is needed to know the basic structure of the system. Once achieved, it will be easier to decide the code. Which will be translate into GNU Radio blocks, USRP configuration and Python algorithms [OCE07].

5.2 Structure

In the system that is going to be implemented, there are transmitter and receiver. That's why are needed two different codes. Although most part of the code can be used for both of them, the structure of the Tx and the Rx will be a little different.

5.2.1 Transmitter

The structure of the transmitter is based in two parts. The Tx of the beacon signal and the Rx of the ACK signal from the receiver. For the first part, is needed a specific sequence which will be transmitted repeatedly until the ACK signal is received.

As said previously, the sequence must be known by both devices. Once transmitted this sequence, the transmitter will start the waiting state, hoping to get the ACK. Since both signals will be in the same frequency and one is sent and the other is received, it seems to be better not to use the same sequence for both, beacon and ACK signals.

After the waiting state, there might be two ways. The first one, if the transmitter has not received the signal. Which can be that the receiver has not sent the ACK or that the transmitter did not have time enough to check it, maybe because it has received the ACK when it was running out the waiting time. If it is like this, then, the device will return to its first state and will send again the beacon signal. This will be repeated as many times as needed.

The other way means that the transmitter has received the ACK. In this case, the protocol is over. Which means that the system is ready to start the transmission of information at the frequency decided. Then, the Cognitive Radio would have done its job properly.

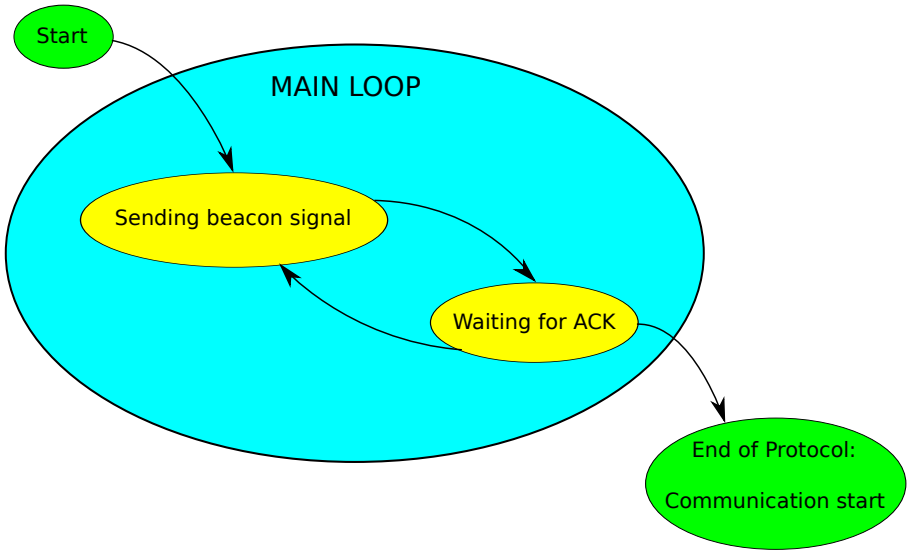


Figure 5.1: Transmitter Structure

5.2.2 Receiver

On the receiver side, it can be distinguished also two parts. The Rx of the beacon signal and the Tx of the ACK signal.

On the receiving part, it also can be divided in two pieces. The first one is going to search any signal that comes into the range of the radio frequency spectrum where the device is aware of. This part of the code will be repeated infinitely until something comes in.

After getting a signal, as it does not mean that it is going to be the right signal, is needed a checking process to know if the signal received is the expected beacon signal. To do this, is needed to go packet by packet and then, when all of them are saved, check if it is the correct sequence or not.

If the checking state gives a negative answer, the program will go back to the first step, where the device is looking again for any signal to comes in.

However, if the answer is a positive one, it will mean that the receiver has got the beacon signal and therefore, the device already know the frequency of the signal and now this state is done.

Knowing the frequency, now the receiver becomes a transmitter. The receiver will send the ACK signal to the same frequency which it first received the beacon signal. This will be sent taking into account the time that takes the transmitter to send the beacon signal and receive the ACK; for it to be able to get the ACK signal.

Once this is over, the connection is ready to be started.

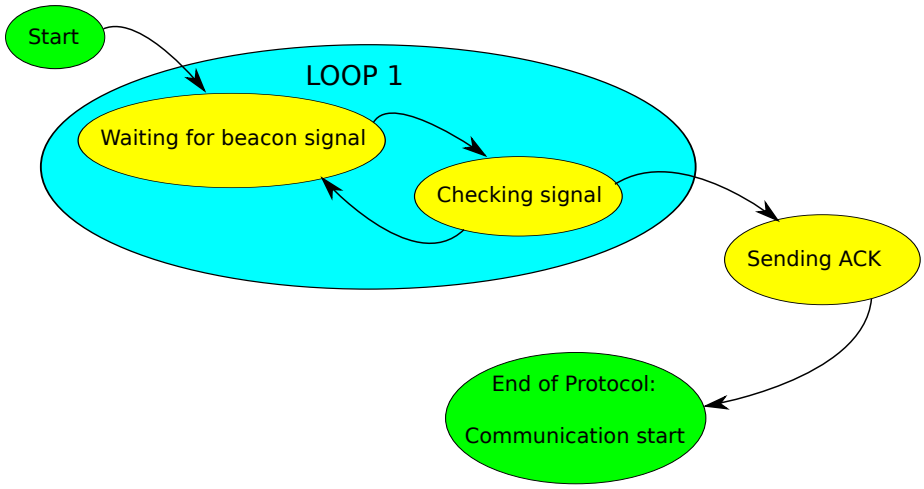


Figure 5.2: Receiver Structure

5.3 Signal Properties

Before getting to the specific code for the protocol, is needed to know much more about the signal in order to write a good program.

About the signal, is really important to take a good modulation. According to the protocol used, first it looks for any signal and then it checks the packets of this signal.

As an example, GSM systems uses GMSK, which became famous and very used since then [WMJ98]. GMSK provides a straightforward, spectrally efficient modulation for the wireless communications.

This good spectral efficiency will permit the system, when it is ready to start the communication, to transmit more data in less time than what it would send using other modulations.

Another good property of GMSK is its linearity and the constant envelope it has [AB99]. If the envelope is constant it gives the possibility to transmit the signal in a high power close to the saturation level. If the modulation used had non-constant envelope it would require a linear power amplification which would consume a big part of the primary power, generate heat and is a relatively expensive device. GMSK comes from a MSK modulation, which is not a linear modulation. But, due to the linearity in phase that the Gaussian low pass filter (Gaussian LPF) gives to the modulation, and the suppression of the side-lobes that MSK generates, GMSK is an appropriate modulation for wireless transmission.

Another good points of using GMSK is its relatively narrow bandwidth, which is also thanks to the G-LPF, and it coherent detection capability.

For an extensive explanation of how GMSK works with wireless systems and why its properties are advantages can be found in [MH81].

Other alternative modulations could be DBPSK or DQPSK which come from BPSK and QPSK, respectively. But in both alternatives, GMSK still have better use of the bandwidth (narrower bandwidth), and so, more power. Another point, for example, is that most mobile products uses Class C power amplifiers, and as they are non-linear GMSK is better in front of QPSK. In front of the DBPSK the GMSK gets to reduce the bandwidth in exchange for increasing inter-symbol interference in the system. While DBPSK gets lower error probability but with more bandwidth [NP08].

Because of all this interdependence is impossible to decide which modulation is better in front of the other.

Once known the information to start coding, it is time to divide the two programs. One will be for the Tx and the other one for the Rx, as also will be needed two devices for the system, which also means that will be necessary two USRP's. However, both devices, both USRP's and both programs will act as both transmitter and receiver.

5.4 Transmitter

The first thing needed for the transmitter is to define the sequence. Any sequence is good to be a beacon signal. Even just a number would be enough for it to work out.

5.4.1 Signal

However, in this project is used a particular sequence of 14 numbers. Actually, the sequence is the beginning of the *Fibonacci series*. The decision of choosing this sequence is just that this sequence, as many others, does not repeat the numbers and so there is less confusion if it has to be taken a number as a reference. So, any other sequence that complies this requirement would be good enough.

The Fibonacci numbers of the series are:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, \ldots$$

Although in this particular case, the 0 has been replaced by number 88, just because it is a number easily recognizable. And in case of errors, it would be easier to detect if it had happened something with the packets.

This first number will be the reference to compare the packets received with the sequence. It means; when the packet containing number 88 is received, the packets received will have a position to start comparing the numbers. Although all the packets numbers will be saved into auxiliary variables, it will not start checking until it gets the reference packet

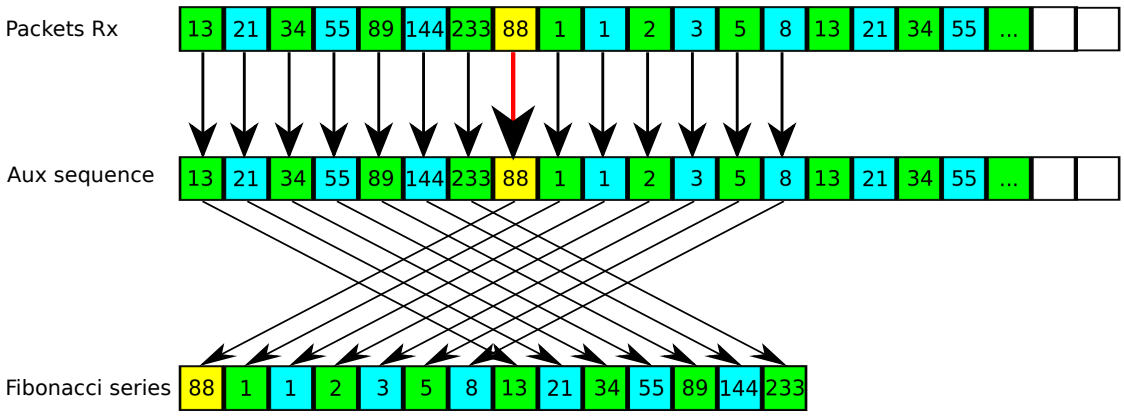


Figure 5.3: Sequence Check Diagram

This sequence must be sent repeatedly until the state time is over. The state time is also a parameter to take into account as the speed of the protocol will be directly affected by this timing and others. The longer it is, the more will have to wait the next state. And as well, the shorter it is, the less time will have the receiver to get it.

In theory, the time which will get the best efficiency is the time that takes the receiver to get the signal and to check the packets. But, as there might be some problems from the outside, such as noise or interferences, and it can provoke some errors in the transmission; the time given to make the whole process will be longer.

Once the sequence is chosen, it has to be modulated into GMSK packets. In GNU Radio, already exists a block which modulate the signal and send it as packets.

This block works as a source, and the data to be, first modulated and finally sent, will be given from the Python domain. When creating the block there are a few parameters to set for the transmission of the signal. The most important is to choose the modulation, which in this case will be GMSK.

In the Python side, will be necessary just to call the function *send_packet()* introducing on it the data for it to be sent. The data will be the number of the sequence, so, for sending the whole sequence (14 numbers) the function *send_packet()* must be used 14 times, which will generate 14 packets.

5.4.2 Tx state

Now, to send the beacon signal, just need the program to be started.

The program will first ask in which frequency the user want to send the signal. In a possible future, this frequency will be given by the own software, when it can gets all the information of the channel, so can decide which is the best frequency to place the connection.

Then, the main loop will be started. Here will be required a variable which will be set to '0' from the beginning and it will only change to '1' when the ACK sequence is checked, so the program will get out of the loop, and the protocol will be finished. In the same possible future, when this protocol is finished, the interchange of information would start here.

The first thing to do in the main loop is to set-up the USRP. In this set-up stage will be defined a lot of parameters. The frequency, which has been given by the user previously. The gain of the signal, the interpolation (or decimation in Rx case), the bit rate and several other features which are pre-defined although they can be redefined if necessary.

Although in the setting of the USRP are needed a lot of lines and parameters to define, in the end, the USRP will just work as a sink. So, this state will just consist in two blocks. The modulator block, and the USRP as a sink.

As explained before, it will send the sequence using the *send_packet()* function. So it can be used as many times as required.

The set of the time that the program will send the beacon signal is done by the number of times that the *send_packet()* function is called. This time can be readjusted depending on the bytes sent and if is necessary to send the sequence repeatedly many times. For the final packet to send it is necessary to call the *send_packet()* but this time with the End Of Line (EOF) set to '1'.

Once this last packet is sent, the program will be able ton continue and go to the Rx state.

Below, in the figure 5.4, there is the diagram of the basic program and the blocks used for the transmission.

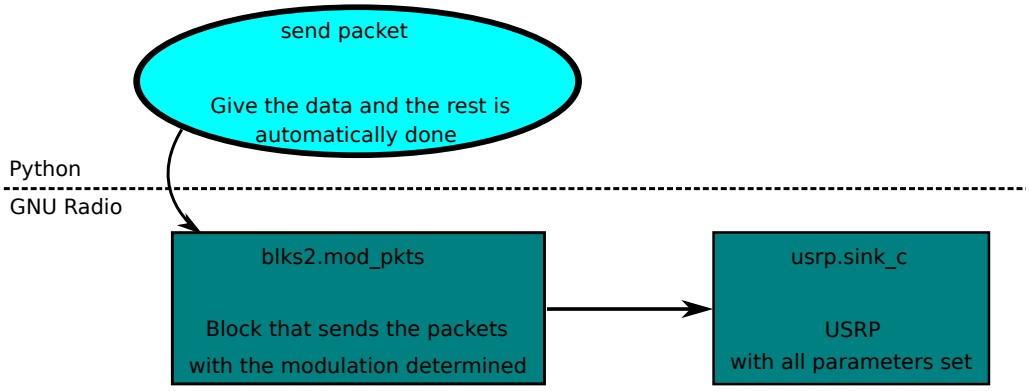


Figure 5.4: GNU Radio blocks for the Tx

5.4.3 Rx state

When the Tx process is already done, now the program will have to start the set-up for being a Rx.

As it is supposed, the reception shall be done in the same frequency as the transmission. So, the first thing to do here is to set-up the USRP. But this time, the USRP will be set as the source block.

For receiving the packets will be required the same kind of blocks as in the transmission.

Is normal that if exists a *modulation block*, there also will be a *demodulation block*. This block works almost in the same way that the first one.

This time is needed a variable where will be placed all the data received, and every time a new packet is demodulated it will be saved into an auxiliary variable in order to get all the packets and finally recreate the sequence.

The difference in this case is that, as the Rx does not know when the packets will come, to set the time it will be used another system. Python itself provides a library of *time* and in it has plenty of functions.

The function used this time is called *sleep()*. This function will 'sleep' the program in the Python domain, so the GNU Radio blocks will be still working and, of course, able to get all the packets that came in. Thanks to this function, this time will be easier to control the time that the program is stocked in this state.

Once the time set is over, the program will have in the auxiliary variable all the packets received and is ready to check if these packets are the ones from the sequence.

Here in the figure 5.5 is shown the flow graph used for the demodulation process.

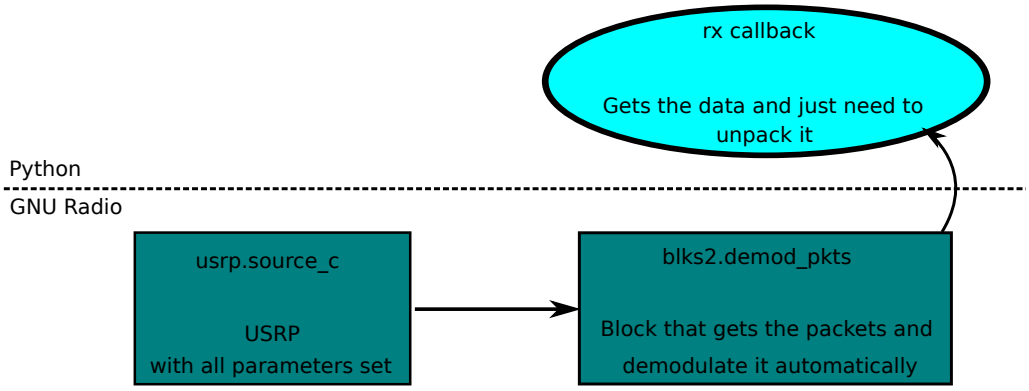


Figure 5.5: GNU Radio blocks for the Rx

This time, as said before, is received another sequence. This sequence will be also known by both devices. This time is chosen a sequence which can not be confused or mixed with the first one.

$$99, 101, 101, 102, 103, 105, 108, 113, 121, 134, 155, 189$$

The checking process will be the same for the Fibonacci series, taking into reference the first number of the sequence. If the checking process confirm that the packets received match the sequence it will mean that the protocol is finished. So, here is the time to set the variable of the main loop to '1', so the program can exit the loop.

However, if the result of the checking process is negative; this global variable will not be set to stop the main loop. Then, the program will go again to the beginning, where it sends the beacon signal.

5.5 Receiver

In the receiver program, the first thing needed is to set up the USRP. This time, as the receiver still does not know the frequency of the signal, obviously, as it is the aim of this project, is needed to set the USRP in order to achieve the maximum possible radio frequency spectrum.

5.5.1 USRP

This time, unlike the previous one, it will be not the only time that the USRP is set. To get the maximum possible range spectrum, the decimation will have to be as low as possible. Which in the USRP used will be 8 (the decimation of the USRP can goes from 8 to 256, two by two).

Depending on the daughterboard used, the frequency range will be around the 400 MHz, 900 MHz, 2.4 GHz, etc... As the sample rate of the USRP is fixed (64 Mega-samples per second in Rx and 128 Mega-samples per second in Tx), the range of the frequency spectrum that can be gotten is easily calculated.

Is just needed to divide the sample rate for the decimation factor. In the case of a decimation factor of 8, the range of the radio frequency spectrum that cover will be 8 MHz

Taking, as an example, the 400 MHz daughterboard, the range will be from 396 MHz to 404 MHz.

Is really important to set-up the USRP in order to get the packets from the first USRP. So, the parameters for the good reception of the packets will must be the same as the USRP of the transmitter. The bit rate, the decimation and other parameters are linked to the set-up of the transmitter.

5.5.2 Rx state

The reception state consists in two sub-states, the one where it is looking for any signal and the second when there is something received, for checking the packets.

In this case, the main loop will just contain the Rx state, as it will only get to the Tx state when it has received and checked the beacon signal. Then, it will remain in the transmission stage and will not get back to the reception stage.

Once it has entered the main loop and set the USRP it will directly go to the searching state.

5.5.2.1 Searching signal

This probably is, the most important an elaborate part of the whole protocol. Here, the program wants to get the beacon signal. The USRP, as a source block, will provide all the RF spectrum signals, and the program will have to find the frequency of the signals. For this, will be necessary a transformation from time to frequency. In GNU Radio already exists a block, called *FFT*, where the input vector is transformed from time domain to frequency domain.

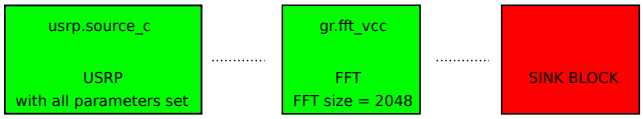


Figure 5.6: Tx blocks; first step

This particular block will be the most important one in this stage, and the one that will determine the composition of the whole flow graph.

As said before, the FFT block needs a vector as an input; and the source block (the USRP) outputs stream. Fortunately, in GNU Radio, exists a block which converts stream of items into a stream of blocks, or in other words, to a vector.

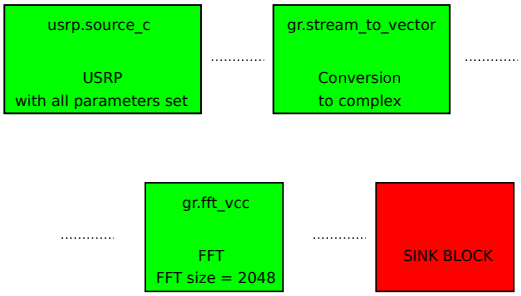


Figure 5.7: Tx blocks; second step

However, it is not enough. As the FFT block will only give the result with a finite input vector. But thanks again to the GNU Radio blocks, there is also another block, called *head*, which takes the first 'n' items of a signal/vector and pass it through the output, after this, the signal is over. Now, the program can take the signal, make it finite, convert it to a vector, and finally pass it to frequency domain.

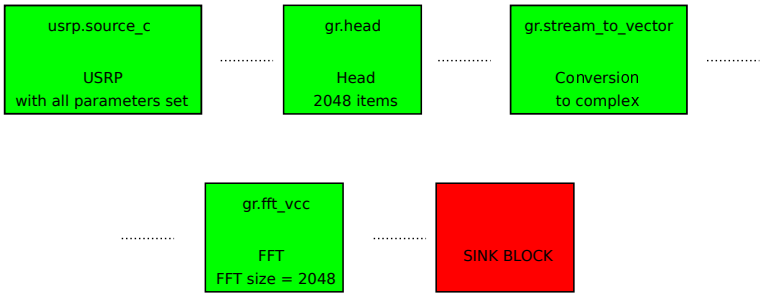


Figure 5.8: Tx blocks; third step

As the FFT block will output the domain conversion of the signal, the sink will receive complex items. And as it is needed to get envelope of the signal, the program will use

another block, called *complex_ to_ mag_ squared*, which will give the envelope squared of the signal.

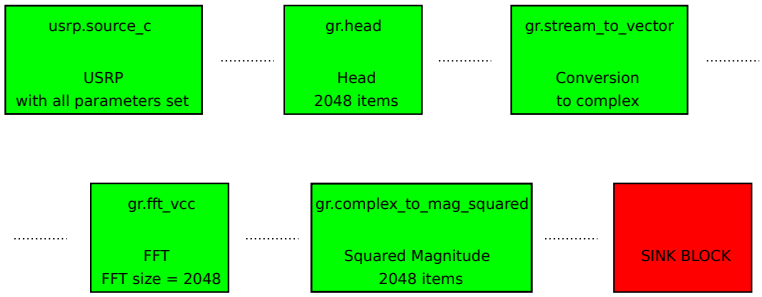


Figure 5.9: Tx blocks; fourth step

Now the program is ready to see if there is a relevant signal taking a look at the frequency spectrum. To find any signal in the frequency, when looking in the spectrum there is a pick, this is the signal. To get this peak, there are plenty of possibilities. For example, GNU Radio itself, has several blocks that detect it, and output different vectors depending on the configuration of the block.

However, in this project it has been decided to take the vector to the Python domain to work there with it. To pass the information from GNU Radio to Python it has been used a block called *message sink*. Which take the input and convert it into a message and save it into a queue, that will be easily accessible from Python

Once this is done, the flow graph is already done, and it looks like in the following figure:

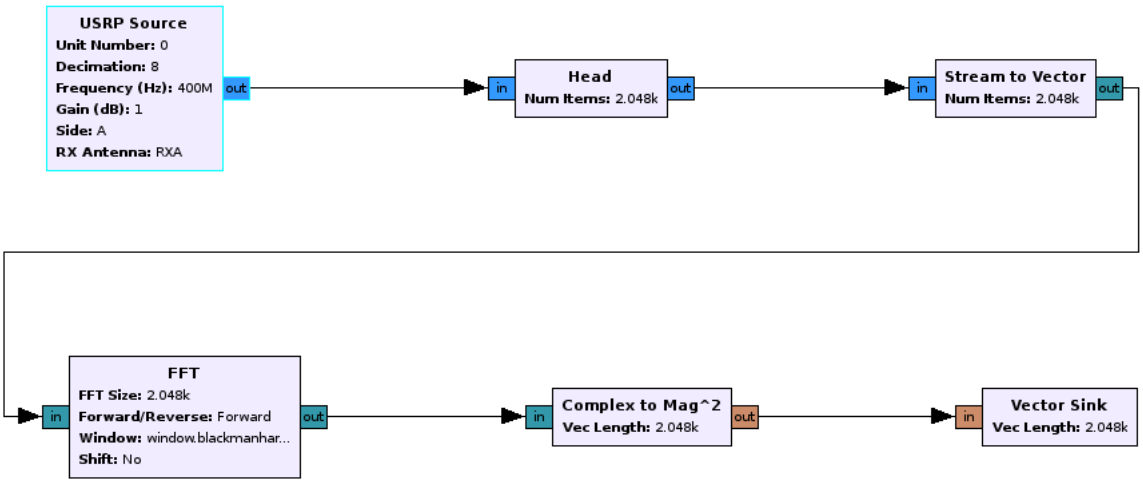


Figure 5.10: Tx blocks; final step

With this flow graph finished, it is time to process the information and search the beacon signal. When the information is received in the python domain, a function will search the peak of the signal. which is quite easy as it is just the *max()* function.

At this point the program will need a reference level. All the vectors will have a maximum point, however, this does not mean that there is a signal. That is why, before getting to the next step, this point will have to pass a threshold. If this sample does not overpass the threshold, the program will do again the reception and it will try to get another time any signal.

This is repeated until the maximum point get over the threshold. It will mean that it has received something different from just noise.

Knowing the highest point in the vector, the position of that point in the vector will be the frequency. To get which is the frequency is needed a conversion from the number of the FFT to the real frequency.

Due to the Digital Down Converter (DDC) of the USRP, the first sample of the FFT output (sample number '0') which is the DC, so 0 Hz, is referred to the central frequency of the USRP. Continuing in the example of 400 MHz, the first sample will be understood as the 400 MHz sample.

To calculate which frequencies correspond to which sample, it is needed the sample rate (which is calculated before with the decimation) and the FFT size. The FFT size will be the same as the size of the vector. In this program it has taken a size of 2048 samples.

So, to know the frequency correspondence is needed a simple conversion.

$$pos = \text{position of the sample that has the peak of the vector}$$

$$f_s = \text{sample rate}$$

$$N = \text{size of the FFT}$$

$$f_D = pos * f_s/N$$

Then, if the sample belongs to the first half of the vector the real frequency is:

$$f_f = f_c + f_D$$

However, if the sample belong to the second half, the frequency is:

$$f_f = f_c - f_D$$

With this, the frequency will be known and theoretically, this is ready to check the packets in the next stage.

The point is that the precision of the detection is not good enough due to the resolution, which in the previous example is 4 kHz. Which means that just 5 samples wrong (out of 2048) are 20 kHz of misfit. As the block of the receiving packets has a tolerance around 10-15 kHz, will be need a really good precision in the detection. To solve this, there are different possibilities.

What is needed is to get the result of the formula *position * sample rate / FFT size* as low as possible. Sample rate and FFT size are the parameters able to be changed.

The FFT size is 2048 so it seems to be big enough, and if it turns to get more size, maybe it will make the program run too much slower. On the other hand, it is possible to change the sample rate by increasing the decimation.

When the decimation is increased, the range of the spectrum received is smaller and then, the resolution is bigger. The protocol has been programmed to do 4 different decimations, although with just 3, or even 2 would be enough.

This is going to be like make a zoom into the frequency detected in each time.

The figure 5.11 shows this 4 different vectors, gotten from the 4 different decimations. In this particular example, the frequency of the USRP was set at 900 MHz and the signal was at 903 MHz.

In the picture, the plots have been changed in order to get the center frequency in the middle of the vector and not in the sample number '0'.

As can be seen, the first time, the frequency detected remains at 902972656 Hz, which is a little far from the 903 MHz, but as can be seen in the next ones, the number is getting closer, getting to the last one, which is at just 489 Hz away from the really center frequency.

This process will increase the detection time just little and gives to the next stage a really accurate result so the packets receive will not have problems to get them. The only problem in this time increasing, is that there is the remote chance that the transmitter stop sending when the program is looking for the 3rd time, so it will not get a signal in the 4th time, which means that it will have to back again to the beginning of the waiting time. Obviously, setting again the decimation to 8, just for the case that it was not the correct signal.

Finally, when all these steps are done, and if there is a result at the 4th time that the program is looking for the signal. This frequency will be the one that is passed to the next stage.

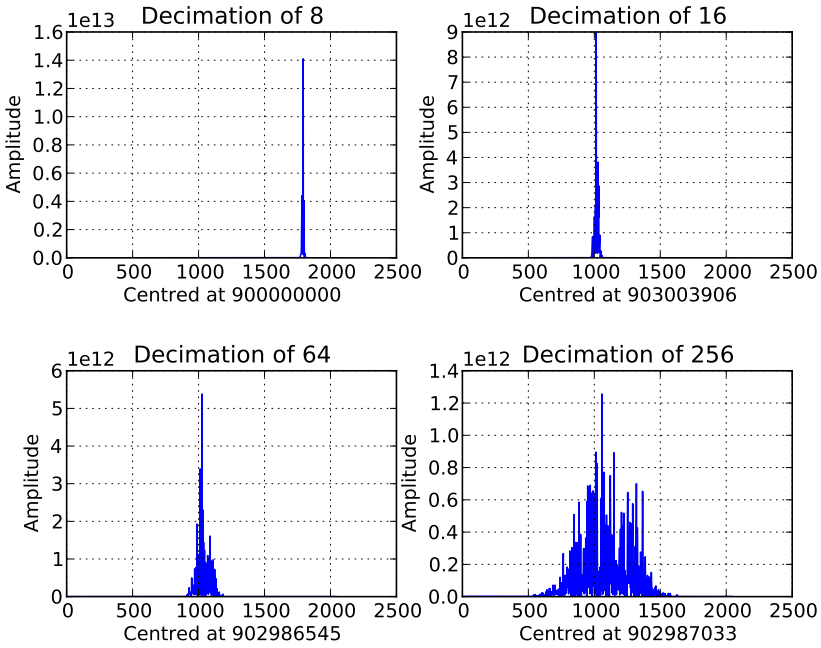


Figure 5.11: FFT plot with 4 different decimations

5.5.2.2 Checking packets

The very first of all, is setting the USRP to its parameters. The important parameters will be, basically, the decimation and the center frequency. Because the bit rate and the others should be in the right configuration. The decimation will be set to 256 and the center frequency to the frequency received from the previous state.

Once the USRP is set up, the packets will be checked the same way that in the transmitter. This time, with the Fibonacci series.

Depending on the result of the check, the protocol will go to one or another point. If the result is negative, the protocol will remain in the main loop for, at least, one more time.

Although the check stage says that the packets received are not the correct ones, there is the possibility that the transmitter has stopped transmitting the signal in that very moment. But, as the receiver can not know this until it has checked all the sequence, the protocol will make its job and will go back to the beginning of the program.

On the other hand, if the checking reveals that the sequence is the Fibonacci series, the main loop will be over and the check state will be over.

The protocol will get out of the main loop and will go to the Tx state. The last job of this state is to pass the frequency center to the next step for it to send the ACK signal.

5.5.3 Tx state

Once the whole Rx state is finished, the only thing to do for the receiver is to send the ACK signal, and then, the handshake protocol will give way to the transmission of information.

For sending the ACK signal, first of all, is needed to set-up the USRP. As explained before, the USRP configuration must match with the configuration from the transmitter.

After the set up of the USRP, which will be the sink block, in GNU Radio domain, is time to create the packets. For this, it will be used the same system that in the transmitter. First they will be modulated and then they will be transmitted by calling the *send_packet* function.

Unlike the transmitter, this time, the ACK signal will be repeated infinitely. This is done as in the protocol ends at the moment that the ACK signal is sent.

In a possible future application, the ACK signal will be sent for a while. The enough time for the transmitter (now converted into receiver) to receive and check it.

Chapter 6

Conclusions

The goal of this project was to connect two wireless nodes. This two nodes, were, supposedly waiting for transmit information. Having the property of Cognitive Radio, they can decide which is the best frequency to stablish the connection.

One of the devices decides the frequency where the communications will be, then, the other one has to connect it without knowing, a priori, which is the frequency chosen by the first device.

The protocol of this project is able to achieve the connection between this two nodes. The range of action of the protocol is 8 MHz.

This two nodes can be two nodes in a network or they can also be two independent users trying to connect themselves. This flexibility provides several applications where this protocol can be used.

In this project the system presented was composed by just two devices, one called transmitter and the other one called receiver. The first one was called transmitter because that one is the device to decide the connection frequency and because it is the one to send the beacon signal.

Keeping this nomenclature, in the system can be added many receivers as wanted because all of them are able to listen to the RF spectrum and check the signal. The code would be adapted in the transmitter program which would not start the communication until all the receivers sent their ACK signal.

With a little bit more of complexity, more transmitter could be added in the system if the program is adapted for all of them to listen for Beacon signals and at the same time, for transmitting it.

6.1 Future Work

The protocol can be adapted into several different ways. Adapted to many different Cognitive Radio systems and of course, the protocol can be improved.

Improved, for example, in more precision to get the frequency. Can also be improved in the timing. Which means to make the protocol faster.

Finally, mention that this protocol can give some ideas on how detect signals using energy detectors or, as well, how to detect signals using other kinds of process. Maybe how to synchronise the beacon signal to improve the efficiency of the receiving time.

Bibliography

[AB99] Ambreen Ali and Felicia Berlanga. Linear vs. Constant Envelope Modulation Schemes in Wireless Communication Systems. In *Introduction to Wireless Communication systems*, December 1999.

[BEJ09] Martin Braun, Jens Elsner, and Friedrich Jondral. Signal Detection in Cognitive Radios with Smashed Filtering. In *Proceedings of the IEEE Vehicular Technology Conference*, April 2009.

[Blo01] Eric Blossom. GNU Radio: Tools for Exploring the Radio Frequency Spectrum. *Linux journal*, 122:76–81, 2001.

[CMB04] D. Cabric, S.M. Mishra, and R.W. Brodersen. Implementation issues in spectrum sensing for cognitive radios. *Conference Record of the Thirty-Eighth Asilomar Conference on Signals, Systems and Computers, 2004*, 1(1):772–776, November 2004.

[Doc] Python Documentation. http://docs.python.org.

[Ham08] Firas Abba Hamza. The USRP under 1.5X Magnifying Lens, June 2008. GNU Radio Project.

[Hay05] S. Haykin. Cognitive Radio: Brain-Empowered Wireless Communications. *IEEE Journal on Selected Areas in Communications*, 23(2):201–220, February 2005.

[Jon05] Friedrich K. Jondral. Software Defined Radio - Basics and Evolution to Cognitive Radio. *EURASIP Journal on Wireless Communications and Networking*, 8(3):275–283, 2005.

[LLCa] Ettus LLC. http://www.ettus.com.

[LLCb] Ettus LLC. http://www.ettus.com/downloads/ettus_ds_usrp2_v2.pdf.

[MH81] K. Murota and K. Hirade. GMSK Modulation for Digital Mobile Radio Telephony. *IEEE Transactions on Communications*, 29(7):1044–1050, July 1981.

[MM99] Joseph Mitola and Gerald Q. Maguire. Cognitive radio: making software radios more personal. *Personal Communications, IEEE*, 6(4):13–18, August 1999.

[MMBF+02] Chair Michael Marcus, Jim Burtle, Bruce Franca, Ahmed Lahjouji, and Neal McNeil. Spectrum Policy Task Force Report. Technical report, Federal Communications Commission, November 2002.

[NP08] Juan Mario Torres Nova and Hernán Paz Penagos. Studying and comparing spectrum efficiency and error probability in GMSK and DBPSK modulation schemes. *Ingeniería e Investigación*, 28(3):75–80, December 2008.

[OCE07] Timothy J. O'Shea, T. Charles Clancy, and Hani J. Ebeid. Practical Signal Detection and Classification in GNU Radio. In *Proceeding of the SDR 07 Technical Conference and Product Exposition*, 2007.

[Pro] GNU Radio Project. http://gnuradio.org.

[Sav06] Neil Savage. Cognitive Radio. http://www.technologyreview.com, April 2006.

[SW04] Gregory Staple and Kevin Werbach. The end of Spectrum Scarcity. http://spectrum.ieee.org/telecom/wireless/the-end-of-spectrum-scarcity, March 2004.

[Wik] Wikipedia. http://en.wikipedia.org/wiki/Universal Software Radio Peripheral. Uses References.

[WMJ98] A. Wiesle, R. Machauer, and F. Jondral. Comparison of GMSK and linear approximated GMSK for use in Software Radio. *Proceedings of the 5th international Symposium on Spread Spectrum Techniques & Applications ISSSTA '98*, pages 557–560, September 1998.

[ZKY06] Chen Zhifeng and Chen Ke-Yu. GNU Radio. http://www.wu.ece.ufl.edu/projects/softwareRadio/documents/Project Report James Chen.pdf, December 2006.

Appendix A

Transmitter Protocol Code

Each and every line of the code has a little explanation of what is its aim.

Start Transmitter

The main part of the ocde starts asking the frequency chosen and getting into the main loop composed by the 'sending beacon signal stage' and the 'waiting for ACK signal stage'.

```
if __name__ == '__main__':
    try:
        x = 0
        #Variable of the main loop
        print '\033[1;44mStart\033[1;m'
        print '\033[1;36mEnter frequency\033[1;m'
        f = int(raw_input())
        #Ask the user for the frequency he wants to transmit

        while x == 0:
        #start the main loop
            main(f)
            #start the stage 'send beacon signal'
            x = main2(f)
            #start the stage 'wait for ACK signal'
            #and waiting for the result
```

```
            if x == 1:
            #ACK detected
                print '\033[1;32mTHE_END\033[1;m'
            else:
            #ACK no detected, we don't get out of the loop
                print '\033[1;31mNO_ACK_detected\033[1;m'

    except KeyboardInterrupt:
        pass
```

A.1 Send Packets

The code of how the packets are send and which is the size used to send are written below.

```
    nbytes = int(1e6 * options.megabytes)
    n = 0
    #variables for the time of the sending

    pkt_size = int(options.size)

    fib = [88,1,1,2,3,5,8,13,21,34,55,89,144,233]
    #sequence of 14 numbers
    i = 0
    #counter

    print '\033[1;34mSending_Beacon_signal_at\033[1;m', frequency

    while n < nbytes/6:

        data = (pkt_size - 2) * chr(fib[i] & 0xff)
        #the packet will have one size
        #but the important part is the header
        #which has the number of the fibonacci series

        payload = struct.pack('!H', fib[i] & 0xffff) + data
        #pack the payload and ready to be sent

        send_pkt(payload)
        #send the packet

        i = i + 1
        #increase the counter
```

```
        if i == 14:
            i = 0
        #check if the counter has get to the end of the sequence
        #in order to repeat it

        n += len(payload)
        if options.discontinuous and pktno % 5 == 4:
            time.sleep(1)
        pktno += 1

    send_pkt(eof=True)
    #send the last packet, so the program will stop sending
    #and get to the next step; waiting for ACK signal
```

Check ACK

For the checking of the ACK signal, look at Appendix B.

The checking process of the beacon signal by the receiver has the same code that the check process of the ACK signal.

The only difference is that the sequence to check is another one.

Appendix B

Receiver Protocol Code

B.1 Get Signal

The 'get signal' code is divided in GNU Radio domain and Python domain.

Get Signal; GNU Radio part

The flow graph of the 'get signal' wants to get the signal into the frequency domain and translate it to the Python part to work with it.

```
class get_signal(gr.top_block):
    def __init__(self, u, Number):
        #u == the usrp
        #Number == FFT size, decided previously

        gr.top_block.__init__(self)

        num = Number
        head_c = gr.head(gr.sizeof_gr_complex, num)
        #block that get the first 'num' items

        nitems_per_block = Number
        str_vec = gr.stream_to_vector(gr.sizeof_gr_complex,
                nitems_per_block)
```

```
#block that converts stream (input) to a vector (output)
#in this case, in complex samples

fft_size = Number
wind = [1]*fft_size
#in this case there is no particular windows chosen
shift = False
action = True #True for FFT - False for iFFT
fft  = gr.fft_vcc(fft_size, action, wind, shift)
#FFT block
#converts the signal to frequency domain

vlen1 = Number
squared = gr.complex_to_mag_squared(vlen1)
#get the magnitude squared of the signal

self.mq = gr.msg_queue()
len = Number*4
sinkmq = gr.message_sink(len, self.mq, False)
#sink block
#will be accessed from the python domain
#to work with the samples

self.connect(u, head_c, str_vec, fft, squared, sinkmq)
#connect all the blocks
```

Get Signal; Python part

This part of the code is for the detection of the signal. In the code are the four different decimations done for the zoom to the frequency detected

```
N = 2048
#FFT size

sample_rate = 64000000/decim
#64 Mega-samples (from the USRP)
#decim == 8
#sample rate == 8 MHz

check = 0
#main loop variable
```

```
    counter1 = 0
    #counter for the first look at the samples
    #with decimation paramater set at 8

    m1 = 0
    #first look loop variable

    f_final = 0
    f_final2 = 0
    f_final3 = 0
    f_final4 = 0
    #the frequency gotten from the result, first set at 0 Hz
    #this is needed just to initializate it

    aux = sample_rate/2
    #this will give the range of the spectrum

    print '\033[1;47mThe range is from %4d
                to %4d\033[1;m' % (f-aux, f+aux)
    print '\033[1;44mStart\033[1;m'
    #The range where the program can look of the spectrum
    #centered in the frequency of the USRP (f)

    while check == 0:
    #main loop started

        print '\033[1;34mWaiting for Beacon signal\033[1;m'

        while m1 == 0:
        #first little loop

            counter1 = counter1 + 1
            #increase first counter
            #this is auxiliary for knowing
            #how many times it has been stocked here

            tb2 = get_signal(tb.u, N)
            tb2.run()
            #set and start the get signal flow graph

            d1 = tb2.mq.delete_head()
            long = '2048f'
            info1 = unpack (long, d1.to_string())
```

```
        #get the information from the flowgraph

        threshold = 5*1e12
        #set a threshold

        x = info1
        #copy the information to work with it

        if max(x)>threshold:
        #see if there is any signal

            position = search_maximum(x)
            #search the position of the peak

            if position < len(info1)/2:
                m1 = sample_rate*position/N
                f_final = m1 + f
            else:
                m1 = sample_rate*(len(info1) - position)/N
                f_final = f - m1
            #get the frequency of the peak

    #WE HAVE THE 'F '... NOW WE NEED RESOLUTION

    decim2 = 16
    tb.u.set_decim(decim2)
    tb.u.set_center_freq(f_final)
    #set a new decimation
    #set the new center frequency

    m2 = 0
    #second look loop variable

    counter2 = 0
    #counter to not get stocked in the second look
    #and be able to get to the first one if there is nothing
    #for getting the maximum possible range

    sample_rate2 = 64000000/decim2
    f2 = f_final
    #new sample rate and new center frequency

    while m2==0:
```

```
        if counter2 > 5:
            m2 = 1
        #if in 5 times it has not detect a signal
        #it goes back to the first little loop

        counter2 = counter2 + 1
        tb2 = get_signal(tb.u, N)
        tb2.run()
        #set and start the get signal flow graph

        d2 = tb2.mq.delete_head()
        long = '2048f'
        info2 = unpack (long, d2.to_string())
        #get the information from the flowgraph

        threshold = 5*1e12
        x2 = info2
        #set a threshold
        #copy the information to work with it

        if max(x2)>threshold:
        #see if there is any signal

            position2 = search_maximum(x2)
            #search the position of the peak

            if position2 < len(info2)/2:
                m2 = sample_rate2*position2/N
                f_final2 = m2+f2
            else:
                m2 = sample_rate2*(len(info2) - position2)/N
                f_final2 = f2-m2
            #get the frequency of the peak

if m2 == 1:
    m3 = 1
#if it has passed the loop but there was no signal found
#it won't get in the 3rd little loop

else:
    m3 = 0
    counter3 = 0
```

```
        f3 = f_final2
        decim3 = 64
        sample_rate3 = 64000000/decim3
        tb.u.set_decim(decim3)
        tb.u.set_center_freq(f_final2)
    #in case it has passed the loop and there is a signal
    #the USRP is re-set

    while m3==0:
        if counter3 > 5:
            m3 = 1
        counter3 = counter3 + 1
        #if in 5 times it has not detect a signal
        #it goes back to the first little loop

        tb2 = get_signal(tb.u, N)
        tb2.run()
        #set and start the get signal flow graph

        d3 = tb2.mq.delete_head()
        long = '2048f'
        info3 = unpack (long, d3.to_string())
        #get the information from the flowgraph

        threshold = 5*1e12
        x3 = info3
        #set a threshold
        #copy the information to work with it

        if max(x3)>threshold:
        #see if there is any signal

            position3 = search_maximum(x3)
            #search the position of the peak

            if position3 < len(info3)/2:
                m3 = sample_rate3*position3/N
                f_final3 = m3+f3
            else:
                m3 = sample_rate3*(len(info3) - position3)/N
                f_final3 = f3-m3
            #get the frequency of the peak
```

```
if m3 == 1:
    m4 = 1
#if it has passed the loop but there was no signal found
#it won't get in the 4th little loop

else:
    m4 = 0
    decim4 = 256
    tb.u.set_decim(decim4)
    tb.u.set_center_freq(f_final3)
    sample_rate4 = 64000000/decim4
    f4 = f_final3
    counter4 = 0
#in case it has passed the loop and there is a signal
#the USRP is re-set

while m4==0:
    if counter4 > 5:
        m4 = 1
    counter4 = counter4 + 1
    #if in 5 times it has not detect a signal
    #it goes back to the first little loop

    tb2 = get_signal(tb.u, N)
    tb2.run()
    #set and start the get signal flow graph

    d4 = tb2.mq.delete_head()
    long = '2048f'
    info4 = unpack (long, d4.to_string())
    #get the information from the flowgraph

    threshold = 5*1e12
    x4 = info4
    #set a threshold
    #copy the information to work with it

    if max(x4)>threshold:
    #see if there is any signal

        position4 = search_maximum(x4)
        #search the position of the peak
```

```
            if position4 < len(info4)/2:
                m4 = sample_rate4*position4/N
                f_final4 = m4+f4
            else:
                m4 = sample_rate4*(len(info4) - position4)/N
                f_final4 = f4-m4
            #get the frequency of the peak


        print '\033[1;32mFound a signal at \033[1;m', f_final4
        #in case there is a signal found
        #it will print the frequency
        #if it has get here, but it has not found a signal
        #it will print a '0'

        if m4 == 1:
            m1 = 0
            tb.u.set_decim(8)
            tb.u.set_center_freq(f)
            f_final = 0
            f_final2 = 0
            f_final3 = 0
            f_final4 = 0
        #if it hasn't found a signal in all of the 2nd, 3rd or 4th
        #it will go back to the first little loop
        #with all the parameters re-set
        else:
            tb.u.set_center_freq(f_final4)

            tb.start()          # start flow graph
            time.sleep(5)
            tb.stop()
            tb.wait()
        #in case it has found something
        #it will start the checking state
        #it will give 5 seconds to look for the packets

            if correct == 1:
                check = 1
            #if the check is correct
            #it will put the variable of the main loop to 1
            #so it will get out of the loop
            #and will go to the sending ACK state
```

```
        else:
            print '\033[1;38mNot the Beacon signal\033[1;m'
            f_final = 0
            f_final2 = 0
            f_final3 = 0
            f_final4 = 0
            tb.u.set_decim(8)
            tb.u.set_center_freq(f)
            m1 = 0
        #if the check state is negative
        #the parameters are re-set
        #as well the USRP
        #and it will go back to the beginning
```

Get Signal; Auxiliary Function

This function returns the position in the FFT vector where the peak is placed

```
def search_maximum(samples):
    maximum = max(samples)
    for i in range(len(samples)):
        if sample[i] == maximum:
            position = i
    return position
    #return the position in the vector where the peak is placed
```

B.2 Check Signal

This function receives the packets in the checking state. It saves every packet to an auxiliary sequence to be able to check it when it is complete

```
    def rx_callback(ok, payload):
    #it gets the packets from the demodulation block

        global beacon_found, correct

        (pktno,) = struct.unpack('!H', payload[0:2])
        #it only looks at the value of the header
        #(the number of the sequence)
```

```
        correct = fib_work(pktno,beacon_found)
        #it save the value and says
        #if the sequence is checked or note

        beacon_found = beacon_found + 1
        if beacon_found == 14:
            beacon_found = 0
        #this is for positioning the value in the sequence
        #to be able to compare it later
```

Check Signal; Auxiliary Function

This is the actual function that saves the packets into an auxiliary variable and returns a '1' when the sequence is checked

```
def fib_work(value, pos):
    global fibseq, no_print
    #fibseq is the vector where will be placed
    #the packets received in the Rx

    auxseq = range(len(fibseq))
    #this vector is auxiliary and not global

    fib = [88,1,1,2,3,5,8,13,21,34,55,89,144,233]
    #fibonacci series of 14 numbers

    for i in range(len(fibseq)):
        auxseq[i] = fibseq[i]
        if i == pos:
            auxseq[i] = value
    #here the value is placed
    #in the proper place

    fibseq = auxseq

    auxseq2 = range(len(fibseq))
    #this is a second auxiliary sequence

    if fibseq == fib:
        if no_print == 0:
            print '\033[1;42mSequence Checked\033[1;m'
            no_print = 1
```

```
        return 1
    #compare the whole sequence
    #it can happens that, by casuality
    #the packets received start with the first one (88)

    aux = 0
    #auxiliary variable

    for i in range(len(fibseq)):
        if fibseq[i]==88:
            aux = i
    #check if the packet number 88 is received
    #and if it is, get the position

    for i in range(len(fibseq)):
        if i + aux > 13:
            auxseq2[i] = fibseq[i+aux-14]
        else:
            auxseq2[i] = fibseq[i+aux]
    #takes the sequence received at that moment
    #and copies to the second auxiliary sequence
    #with the reference (88) delay


    if auxseq2 == fib:
        if no_print == 0:
            print '\033[1;42mSequence Checked\033[1;m'
            no_print = 1
        return 1
    #compare the sequence of packets received
    #with the fibonacci series sequence
    #with the reference (88) delay

    #if the result is positive, it returns '1' (positive)

    #if it gets here, it means that the sequence
    #has not been checked yet

    return 0
    #so it returns '0' (negative)
```

Send ACK Signal

For the sending of the ACK signal, look at Appendix A.

The sending process of the beacon signal by the transmitter has the same code that the sending process of the ACK signal.

The only difference is that the sequence to send is another one.