Universidade de Aveiro
2019

Departamento de Eletrónica,
Telecomunicações e Informática

Diogo Manuel
Policarpo Batista

Leitor SDR para sensores passivos backscatter

SDR reader for passive backscatter sensors

Universidade de Aveiro Departamento de Eletrónica,
Telecomunicações e Informática
2019

**Diogo Manuel
Policarpo Batista**

Leitor SDR para sensores passivos backscatter

SDR reader for passive backscatter sensors

“*For it is impossible for anyone to begin to learn that which he thinks he already knows.*”

— Epictetus

Universidade de Aveiro Departamento de Eletrónica,
2019 Telecomunicações e Informática

Diogo Manuel
Policarpo Batista

Leitor SDR para sensores passivos backscatter

SDR reader for passive backscatter sensors

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Eletrónica e Telecomunicações, realizada sob a orientação científica do Doutor Arnaldo Silva Rodrigues de Oliveira, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Nuno Miguel Gonçalves Borges de Carvalho, Professor Catedrático do Departamento de Eletrónica e Telecomunicações da Universidade de Aveiro.

Ao meu irmão, que me ensinou muito mais do que se apercebe

o júri / the jury

presidente / president	Prof. Doutor Pedro Miguel da Silva Cabral professor auxiliar do Departamento de Eletrónica e Telecomunicações da Universidade de Aveiro
vogais / examiners committee	Prof. Doutor Arnaldo Silva Rodrigues de Oliveira professor associado do Departamento de Eletrónica e Telecomunicações da Universidade de Aveiro
	Prof. Doutor Rafael Ferreira da Silva Caldeirinha professor coordenador do Departamento de Engenharia Eletcrotécnica da Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Leiria

agradecimentos / acknowledgements

Agradeço aos meus pais e irmão, gigantes nos ombros dos quais assento, pela linha de apoio constante, pelos anos de crescimento ao vosso lado e por todos os obstaculos que me ajudaram a conquistar. Ao Rùbêñ Filipe, por ter sido uma peça tão essencial, pela partilha de momentos, pelo feedback pontual, pela ajuda a automatizar a produção de *iron* e por todas as conversas. À Joana Reis, que me ensinou novas maneiras de crescer. Ao João Pantaleão, por todas as *memes*. Ao Fábio Henriques, a pessoa com a maior habilidade social que já conheci, por lá ter estado na altura mais dificil da minha vida. Ao Tiago Costa, em quem eu me revejo fortemente e que me aturou muito mais do que devia. Ao Emanuel Valente, que, através do que me mostrou, me colocou desde muito cedo num melhor caminho. Ao Pedro Marques, porque quis ajudar, porque tem um grande sentido de humor e simplesmente porque sim. Ao Daniel Canedo, o meu socialista favorito, que nunca te falte *skill*. Ao Professor Doutor Arnaldo Oliveira, pela sua expertise a cada passo deste projecto, dos melhores professores que já tive. Ao Professor Doutor Nuno Borges de Carvalho, pelo esclarecimento de dúvidas quando o processo encravava. Ao Felisberto e ao Ricardo, pela inteligência toda que empurrou esta tese para a frente, e pelo impacto que eu acredito que irão deixar no mundo. Ao Ribeiro, BG, Gapo, Christian, João, Vitó, Botelho e Raul pelos momentos que passámos a mandar dardos e a jogar Minecraft. Ao Mostardinha, pelo excelente apoio técnico, e por só me ter arranjado soluções quando eu só lhe arranjava problemas. À Dona Elisabete, pelo café que me pagou e por estar sempre bem disposta e pronta a animar a malta. Ao CSU, em especial à Dra. Patrícia Marinho, que incutiu em mim cinquenta novos mundos e me facilitou de maneira tremenda na adaptação a este. Ao Instituto de Telecomunicações, pela excelente infraestrutura, equipamento laboratorial e *staff*. À Universidade de Aveiro que tão bem me tratou nesta fase da minha vida, e da qual me despeço na melhor das notas. E à cidade de Aveiro, pois vir para cá foi o melhor investimento que já fiz.

Palavras Chave

IoT, backscattering, wireless sensor networks, reader, receiver, digital signal processing, SDR, Python.

Resumo

À medida que o conceito Internet of Things se aproxima cada vez mais da realidade, são também identificadas novas lacunas que necessitam de ser preenchidas. São necessários novos avanços na tecnologia de leitores, de maneira a que os mesmos fiquem mais capazes, baratos e adaptáveis. Os leitores convencionais situam-se presentemente, a nível de custo, na ordem dos dos milhares de euros. É importante baixar esta barreira de entrada, pois quanto menor esta for, mais atraentes serão as soluções IoT e maior será a velocidade de crescimento desta rede global. Melhorias de desempenho e preço em *single board computers* comercialmente disponíveis como o Raspberry Pi, e em receptores RF como os RTL-SDR abriram as portas a uma nova classe de leitores ultra-low-cost. Esta dissertação procurará explorar uma nova possibilidade de recepção e desmodulação sinal RF oriundo de *tags* passivos utilizando as tecnologias já mencionadas. A arquitetura do protótipo final foi desenhada e implementada de maneira a que o front end RF baseado numa *dongle* RTL-SDR, receba o sinal ASK oriundo da *tag*, e o acondicione e discretize/quantize, para então ser passado através de USB para o Raspberry Pi, onde o mesmo será processado sob a forma de filtragem, decimação, desmodulação, validação e integração em rede. Este processo é primeiro introduzido do ponto de vista arquitetural e depois, mais detalhadamente, é feita uma descrição da implementação. Serão também apresentados resultados de teste com variações de distância entre antennas e grau de decimação do sinal. O protótipo foi testado em condições de laboratório e de campo, os resultados obtidos são promissores. São adicionalmente deixados algumas sugestões de tópicos de trabalho futuro, juntamente com um manual de utilizador e um pacote de ferramentas uteis a eventuais futuros desenvolvimentos.

Keywords IoT, backscattering, wireless sensor networks, reader, receiver, digital signal processing, SDR, Python.

Abstract

As the Internet of Things concept moves closer to reality, some technological gaps come to light and require addressing. Advancements in reader technology are needed in order to achieve higher degrees of capability and adaptability at lower costs. Conventional readers are currently priced in the range of thousands of euros. It's crucial to lower this entry barrier, because the lower it gets, the more attractive IoT solutions will become and the faster the global network of sensors will grow. Improvements in the performance and cost of single board computers such as the Raspberry Pi and also of RF front-ends such as the RTL-SDR have opened the way to a new class of ultra-low-cost readers. This dissertation aims to explore a new possibility of reception and demodulation of an RF signal transmitted from a passive sensor using the aforementioned technologies. The final prototype's architecture was designed and implemented to facilitate the reception, conditioning and discretization/quantization of the ASK modulated RF signal by the RTL-SDR based front-end, so that it can then be sent via USB into the Raspberry Pi where its information will be processed through steps such as filtering, decimating, demodulating, validating and integrating into the network. This process is first introduced under the architectural point of view and then, under a more detailed approach, in a description of the implementation. Testing results are presented with variations of antenna distance and decimation ratio. The prototype was tested in laboratory and field environments and the results are promising. Also included are some potential future development suggestions and a package of development tools.

Contents

List of Figures

List of Tables

Glossary

ADC	Analog to Digital Converter
AGC	Auto-Gain Controller
ASK	Amplitude Shift Keying
COFDM	Coded Orthogonal Frequency Division Multiplexing
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DC	Direct Current
FIFO	First-In, First-Out
FTP	File Transfer Protocol
GPIO	General Purpose Input/Output
ID	Identification
IF	Intermediate Frequency
IQ	In-phase and Quadrature
IoT	Internet of Things
LED	Light Emitting Diode
PBZ	Pass By Zero
PC	Personal Computer

RF	Radio Frequency
RPI	Raspberry PI
RTL	Register-Transfer Level
SDR	Software Defined Radio
SMA	SubMiniature version A
SPBZ	Smart Pass By Zero
SSH	Secure Shell
TMC	Transition Match Comparator
USB	Universal Serial Bus
USD	United States Dollars
USRP	Universal Software Radio Peripheral
VSG	Vector Signal Generator
WPT	Wireless Power Transfer

Chapter 1

Introduction

The Internet of Things (IoT) consists on the inclusion of electronic components in everyday objects, making them able to provide information into or be managed remotely by a global network. These tags can be retransmitting previously stored information, like a product Identification (ID) or a virtual license plate, but it can also be real-time harvested information, such as temperature or humidity of its environment.

The logistical advantages created through the deployment of such a large scale automated product identification and contact-less information exchange are highly beneficial for a number of different industries. Aerospace engineering [1], agriculture, food monitoring [2], bioimplants [3] and worker safety protocols [4] are some of the practical applications of this technology.

Figure 1.1 illustrates the IoT concept.

Figure 1.1: The internet of Things (icons from [5]).

The architecture of a tag network can be as simple as a tag and a reader attached to a processing station, like the one seen in Figure 1.2. This is expandable, multiple tags and readers can be connected to the same network facilitating the harvest of multiple kinds information from different sources.

Each tag can be read by multiple readers simultaneously. One reader can interact with potentially thousands of different tags, each fulfilling a specific role such as managing vacancy levels, item types, and atmospheric conditions inside a warehouse. If there is a malfunction in either a single tag or reader, each can be replaced individually without inducing a massive reconfiguration ripple into the system. All of this labour, which would otherwise be costly if executed manually, is now automated into a configuration with much lower upfront and maintenance costs.

Even though the potential benefits of an ever expanding source of actionable custom tailored information are extremely appealing, this venture brings its own challenges [6] to the table, in particular it requires a massive deployment of billions of sensors, acting as the groundwork for a global interconnection system. Casting such a wide hardware net causes, in turn, the emergence of added barriers. Interconnectivity [7], need for privacy [8] and energetic autonomy are a few examples of focus points within the scientific community.

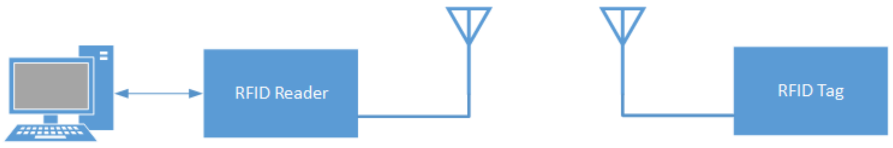


Figure 1.2: Basic RFID system

With the growth in network size, an ever increasing amount of sensors will be deployed simultaneously, each of them performing energy draining activities such as sensing, processing or transmitting. This energy must be sourced from somewhere, and battery-based approaches are unfavoured here. Efficiency can be increased and consumption lowered, but batteries eventually deplete, and depending on the network architecture and number of nodes, replacement can be infeasible or outright impossible.

As a potential solution to the energy concerns, alternate ways of power supply have been explored, namely Wireless Power Transfer (WPT). By having a central Radio Frequency (RF) source from which the nodes can draw power, batteries can be avoided altogether. This has significant logistical, economical and environmental benefits. Disadvantages are also present, WPT is inefficient at higher energy requirements, and even more so as the distance between source and sink increases. Additionally, even in battery based approaches higher power requirements mean shorter tag lifespans. For both cases a strong argument exists towards efficient power management.

In order to decrease tag energy consumption, modulated backscattering methods have been implemented [9] [10]. These rely on encoding information in an incoming carrier wave by pattern switching between different reflection coefficients. Doing so transports the sensed data out of the sensor and into a listening reader without resorting to any active transmission components on the sensor side.

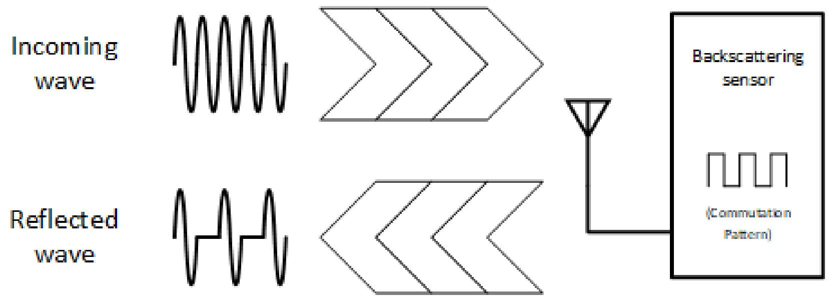


Figure 1.3: Visualization of the process by which the signal is modulated.

Due to the low-power consumption orientation of these tags, other aspects have also been optimized in that direction. Namely the communications protocol, it consists of small packets, with simple validation methods and a low bitrate.

Regarding existing readers for this scenario, there are some market options. The Ettus Universal Software Radio Peripheral (USRP) N210 motherboard [11] + daughterboard [12] combo which is currently priced at 2.493€ or the Alien ALR-9680 [13] which comes with a price tag of 999 United States Dollars (USD). None of these could be considered a low-cost solution, and this market opportunity represents an additional incentive.

Additionally, in 2017, a research team from Instituto de Telecomunicações published developments on a semi-passive sensor [9], able to backscatter information created in real-time. This event would give the final push necessary to jump-start this project.

1.1 Motivation

The main goal of this dissertation is to conceptualize and build a reader prototype. This system would need to be able attain the following goals:

- **Backscattered ASK demodulation ability**
  The reader must primarily be able to receive and extract information from an incoming RF ASK backscattered signal.
- **Adaptability to different communication protocols**

  This prototype can serve as a basis for solving logistical problems across a large heterogeneous group of industries, each of them imposing their own custom requirements and limitations. It is beneficial that the reader can adapt as much as possible to different environments.
- **Be low-cost**

Over the past few years technological developments have lowered the price-tag on SDR based RF transceivers, thus facilitating a new approach to this problem, the low-cost mindset.

As development progressed through its stages, facing different challenges and suffering adaptive demands, these three were maintained as the fundamental guiding principles.

1.2 Project Time-line

During early development stages a proof-of-concept methodology was adopted. MATLAB [14] was used as the initial project grounds, it was an easy-to-implement path to gauge achievable reliability levels. At this stage, special focus was given to the maximum attainable distances between reader and sensor antennas in order to maximize the number of potential applications.

A few months into development, and after the concept was proven via laptop/MATLAB implementation to perform at appropriate levels, a shift in direction was introduced. A new and lower cost platform was adopted, the Raspberry PI [15]. This in turn prompted the change from MATLAB to Python [16]. Existing code was translated and new working setups were designed.

After a new round of development and testing, a new barrier was identified, it consisted of a low data throughput due to chosen low-cost platform. A second directional change happened at a later time, focus shifted from improving range into optimizing code interpretation speed. This is the reason why three different data recovery algorithms are implemented, each spawned from different necessities at different stages in the project, and each with its usages and drawbacks.

A ton of avenues are still left open to explore, care was taken in order to facilitate as much as possible the work of future developers, suggestions on this area were left on section 6.2.

1.3 Document structure

This document is sliced into the following chapters.

- Introduction
  This chapter.
- Basic Concepts
  Contextualization of several concepts mentioned throughout the document.

- Reader architecture
  Abstract description of the project. Data pipeline architecture laid out in "black-box" like segments.
- Reader implementation
  Detailed description of the critical path implementation. Overall implementation schemes, technologies used, thread architectures, compromises made and reasoning behind them. Strengths, drawbacks and ideal usage scenarios.
- Hardware Setup and Results
  List of Hardware required to reproduce measurements. Measured results
- Conclusion
  Conclusions drawn from the project.
- Annex - User's Guide
  Usage guide aimed at the end user.
- Annex - Developers' Tools
  Inclusion of all non-critical additions in the project. Filter and signal simulation tools. Useful for future developments.

Chapter 2

Basic Concepts

2.1 Introduction

In order to facilitate the transmission of knowledge, the best approach is often to begin by the most abstract, simpler concepts. This helps introduce the thesis' context on a smoother, easier to absorb way, and also aids on placing author and reader on the same page. On this chapter, we will approach some key fundamental ideas which will be built upon later throughout this document.

2.2 Amplitude shift keying

ASK is a digital modulation technique that encodes information into a carrier wave by modifying its amplitude over time. This type of modulation offers simplicity of implementation at the expense of susceptibility to noise. This type of modulation usually encodes one bit into two possible states of the modulated wave, corresponding two different constants being multiplied by the carrier's amplitude. The biggest amplitude state is commonly referred as the 'HIGH' state, whilst the smaller one is the 'LOW' state. These states can be corresponded to '0' and '1' bit values in the demodulation process, thus facilitating the transfer of information via RF wave over a medium.

On the top of Figure 2.1, we can see the main input of this process. The signal on the bottom represents the output.

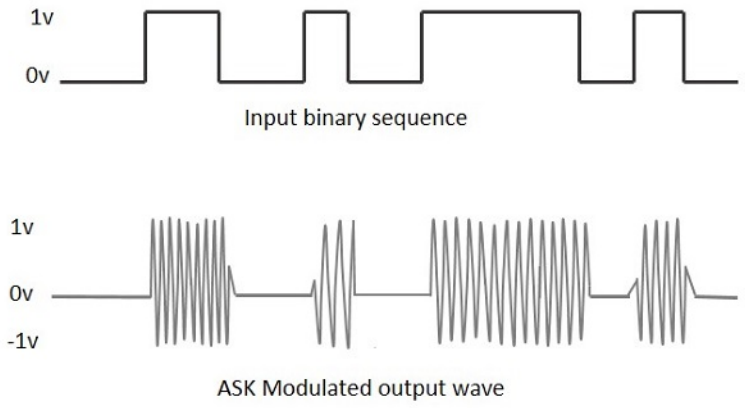


Figure 2.1: Binary sequence converted into ASK (image from [17]).

The techniques used to modulate or demodulate information using ASK are described in more detail in the following subsections.

2.2.1 Modulation

During the modulation process, a carrier wave is multiplied with a digital modulating signal, resulting in the modulated wave. The method is illustrated on Figure 2.2.

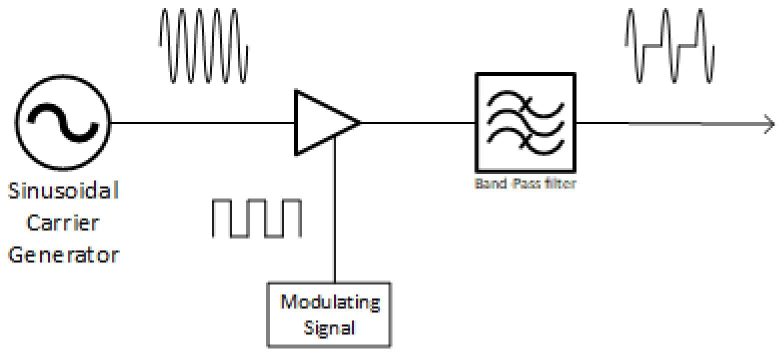


Figure 2.2: ASK modulation of a binary modulating signal

Starting with an analogue sinusoidal wave, designated as the carrier, we shift the amplitude ($A_c * m_s$) to different thresholds according to the sequence of values in a discrete signal (m_s). The angular frequency (ω) and phase (ϕ) remain untouched.

The process happens according to the following formula.

$$x(t) = A_c * m_s * cos(\omega t + \phi) \tag{2.1}$$

Furthermore, ASK can vary on modulation index, also occasionally designated as depth level. This is value can be directly obtained from HIGH/LOW states' relative amplitude levels.

$$M_{index} = \frac{V_H - V_L}{V_H + V_L} \tag{2.2}$$

The ratios between carrier, discrete input signal, and produced modulated signal, for a case where the modulation index equals 0.5 can be seen on Figure 2.3.

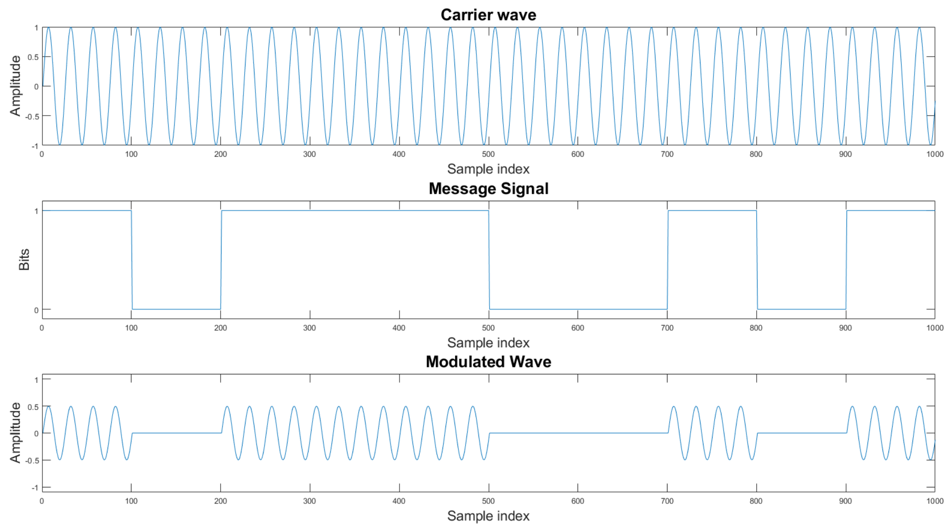


Figure 2.3: Different signals involved in the modulation process. Depth level equals 0.5 in this example.

The resulting analogue signal $x(t)$ now contains in itself all the information in m_s, while giving the benefit of choosing the frequency band at which transmission occurs. At the receiving end, the inverse of this method must be applied in order to recover m_s, as described in the next section.

2.2.2 Demodulation

In order to recover the original message from a modulated ASK signal, three distinct processes are required. To start, the signal must be submitted to full-wave rectifier. Secondly, a filter must be applied in order to remove the carrier component of the modulated signal. And finally, the filter output must be run through a comparator/data recovery algorithm in order to extract the message bits, previously designated m_s.

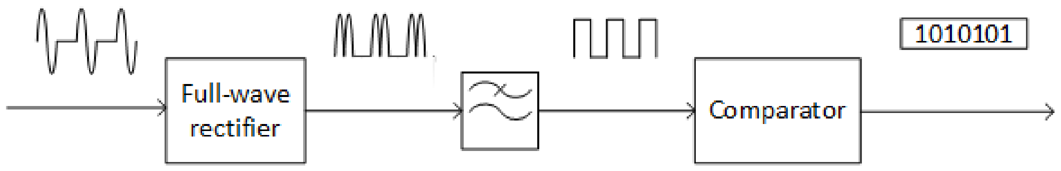


Figure 2.4: ASK demodulation process

In typical cases the next step is the parsing of information within the bit, this includes error detection, an example of such a method is described on subsection 4.7.3.

2.3 Modulated backscattering

In chapter 1, the concept of a communications system with a tag and reader has already been introduced. Let's now analyse in closer detail the tag's function and inner workings. From the whole system's point of view, it acts as a source of relevant information, this information can be directly sourced from an inbuilt sensor, or deduced by the tag's presence or absence on a key location. But for any of this to be possible a method of sending information from tag to reader is required.

Firstly, however, it's important to identify different classes of tags according to power source. There are three main classes, but only the last two are relevant in the context of this thesis.

- Active
  This class of tags sources energy from an wired external power source.
- Semi-passive
  This class of tags sources energy from an inbuilt battery.
- Passive
  This class of tags sources energy from WPT methods, or harvests it from the environment.

In the last two cases, especially on the last, energy consumption management is a relevant concern. Sourcing power from a battery means it will eventually need replacement, and, in the case of fully passive sensors, WPT and ambient power harvesting both become less appealing as we increase potency requirement levels. Recognizing the need to transport information from tag to reader, but at the same time knowing that these energy concerns discourage the inclusion of an active transmitter component on the tag, leaves us in a position to favour a modulated backscattering approach.

2.3.1 Inner workings

In the context of a backscatter system, a tag is a piece of hardware that includes an antenna, a power source/energy harvester, a microprocessor and, optionally, a sensor. However, from an RF circuit point of view, and to an external observer, such as the reader, all these details can be abstracted into a load. This load is intentionally variable via internal switching between different circuit paths, managed by a microcontroller. This means that, by extension, so is the tag's reflection coefficient. By controlling the pattern in which this coefficient changes over time, the tag can effectively force amplitude pattern modifications into the reflected component of the incoming carrier wave, essentially imposing information into it via an amplitude based modulation. Relevant information generated in the tag's sensing components can be encoded in this reflected signal, thereby creating a system capable of transmitting information without consuming power on the transmission itself, such as the one in Figure 1.3.

If we consider Z_A and Z_L to be line and load impedances, respectively, we can calculate the reflection coefficient (Γ) using the following formula:

$$\Gamma = \frac{Z_L - Z_A}{Z_L + Z_A} \tag{2.3}$$

Because the reflection coefficient is the ratio between incident ($\mathrm{V_i}$) and reflected ($\mathrm{V_r}$) voltages, we can also apply:

$$V_r = V_i * \Gamma \tag{2.4}$$

In practical terms, the carrier wave is received by the sensor's antenna, and then directed through a transistor into one of multiple possible paths. These paths will reflect the wave back through the same antenna, however with different reflection coefficients. For example, one of them will attempt to fully reflect the signal ($\Gamma = 1$) whilst another will try to absorb rather than reflect it ($\Gamma = 0$). By switching between these two states it becomes possible to encode into the resulting reflected signal an ASK modulated message, with an ideal depth of 100%. From the reader's point of view, an ASK modulated signal is being received, with a frequency equal to the original transmitter's carrier wave deviated by the backscattering sensor's symbol rate. Other modulation schemes such as [18] have been implemented.

This method allows for information transmission between tag and reader without an active transmitting component built into the sensor, which comes with the advantage of lower power consumption, making WPT a more feasible implementation.

2.4 Software defined radio

SDR is the designation given to a platform which relies on software or flexible hardware and its software based reconfigurability to process a radio signal. It has the big software-backed advantage of being able to quickly adapt into changing circumstances. A system like this is easier to maintain, as an update can be rolled out through the network even from the other side of the globe. It can be kept up to date in a much easier fashion as regulations, communication protocols, frequencies and modulation schemes change, without being bound to the higher inertia of the more hardware based counterparts as small updates do not require a complete redesign and redeployment of the system.

Until recently, a common SDR setup would be a USRP motherboard [11] attached to a daughterboard [12] working as a transceiver [19], connected in turn to a Personal Computer (PC) (often running GNURadio[20]). This changed with the introduction of low-cost DVT-B USB Dongle [21], pictured in Figure 2.5. This hardware revolution combines a tuner such as the Rafael Micro R820T or the Elonics E4000 (now discontinued) with the Realtek RTL2832U chip. This combination allows for signal harvesting and conversion to baseband and then into discrete digital samples for a price of 20€. Current publicly available software libraries [22] allow the user to bypass the inbuilt demodulation methods and use debug mode included in the dongle, turning it into an IQ sampler with a configurable sampling rate of up to 3.2MS/s.

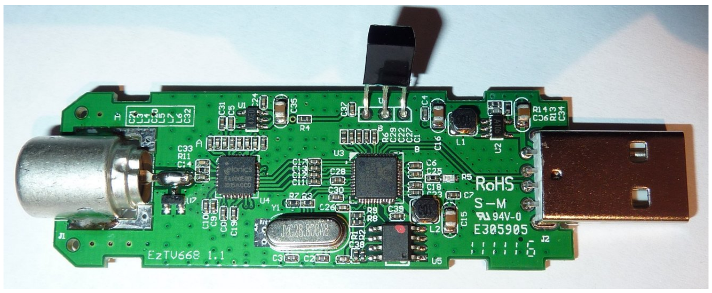


Figure 2.5: Caseless DVT-B USB dongle [23]

In Figure 2.6 we can see diagram of this device.

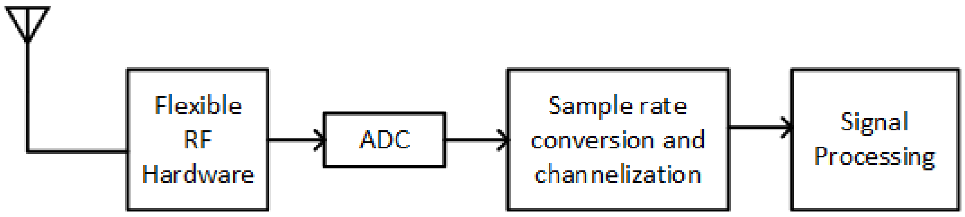


Figure 2.6: High level overview of the typical SDR transceiver.

The RF front-end in this project consists of a similar concept, but without transmitting capabilities. SDR based receivers aim towards converting the front-end collected incoming RF signal into the digital domain as quickly and efficiently as possible. In order to do so the signal must first be conditioned via a an approach similar to a super-heterodyne receiver. This happens through the application of a bandpass filter with cut-off frequencies matching the desired tuning, followed by an amplifier and a up/down-converter to bring the incoming RF signal to into a constant Intermediate Frequency (IF). At this stage, narrower filtering and amplification occur in order to prepare the signal for the following ADC. In Figure 2.7 we can see an overview of this channel.

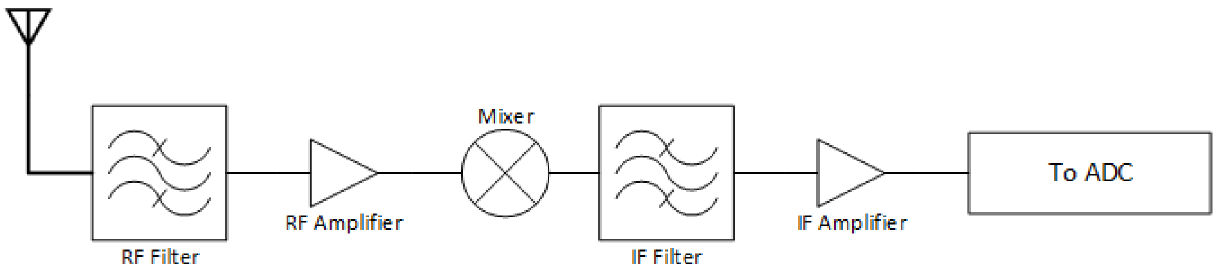


Figure 2.7: Diagram of the R820T tuner used in the receiver.

After the signal as been shifted into an adequate intermediate frequency, it is then fed into an ADC and digitally shifted into baseband via a digital down-converter, as illustrated in Figure 2.8.

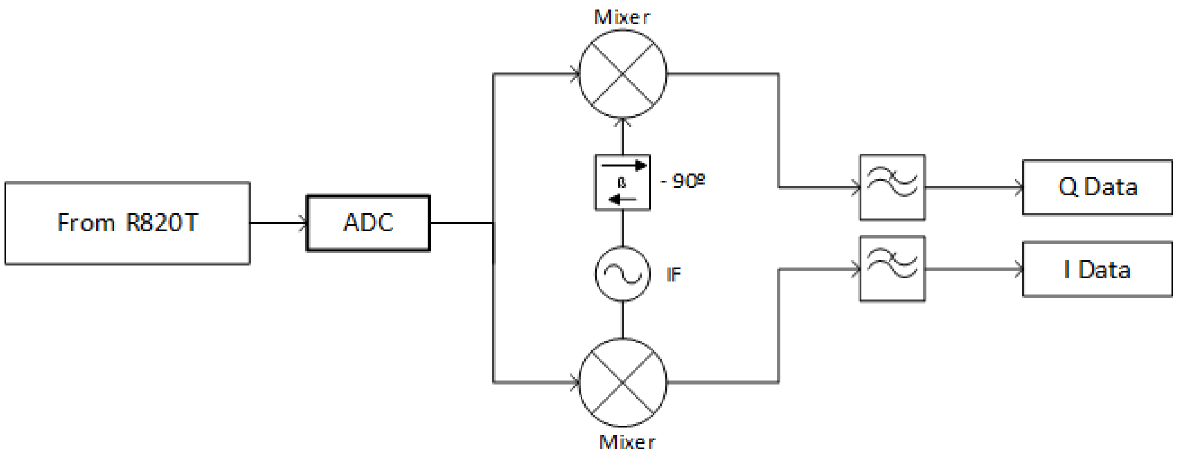


Figure 2.8: Diagram of the RTL2832U chip used in the receiver.

This system outputs IQ samples, which in turn would require to be further processed. An efficient solution to this need is a single board computer, such as the Raspberry PI (RPI). This combination (RPI/RTL-SDR) has been tried before in situations like spectrum analysis [24], spectrum management [25] and disaster relief [26]. This is in part due to its extremely low monetary entry barrier, both of its parts are cheap (50€ total), commercially available, and already possesses freeware options for operation [27] [22]. These are also reasons as to why it was adopted for this project.

Now that these important concepts have been introduced, let's introduce the architectural design in chapter 3.

Chapter 3

Reader architecture

3.0.1 Introduction

In this chapter, an abstract description of the whole system architecture will be exposed, firstly as a whole and then each individual step, with added detail.

The system consists of a pipelined processing chain with different modules, each with a specific function and place in the whole. For now these modules will be approached as a black box, however a detailed description regarding each implementation's inner workings can be found in chapter 4.

Let's begin by defining inputs and desired outcomes. As stated on chapter 1 the, goal of this project is to work as a reader for a passive backscattering tag. This means the RF waves reflected towards the reader will serve as the input, or the source, of this project. This RF signal will carry packets of information according to a previously agreed upon protocol, as explained in section 2.3 and subsection 4.7.1. The desired output, or sink, will be act of digitally storing in memory the payloads extracted from validated incoming packets.

A set sequence of actions is required in order to process de radio waves digitally and extract the payloads from incoming ASK signal. In Figure 3.1, we can see a simplified pipeline of the whole process.

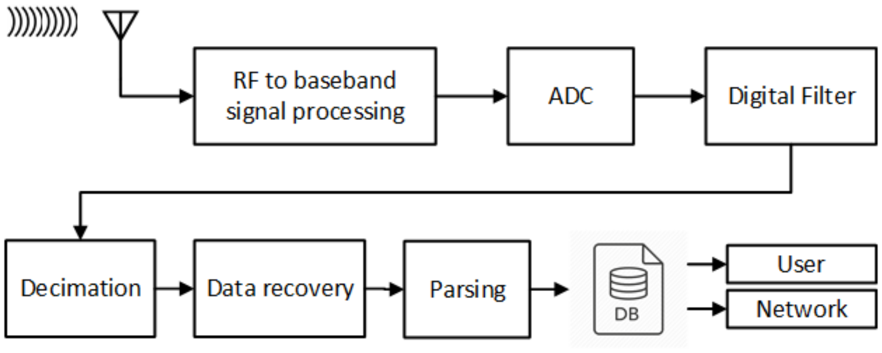


Figure 3.1: Simplified data pipeline (icon used from [5]).

3.1 RF to baseband signal processing

A representation of the RF to baseband downconversion step can be seen in Figure 3.2.

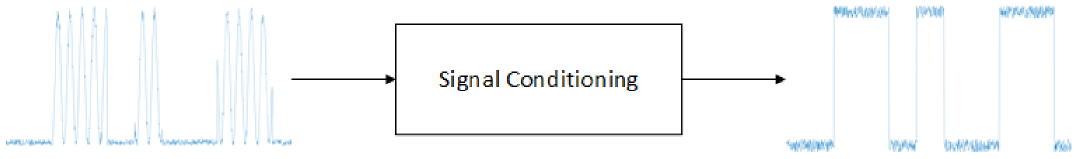


Figure 3.2: RF to baseband downconversion.

It is not feasible to directly feed the signal from antenna into the ADC, doing so would leave the latter sampling from the entire spectrum, and most likely saturating its analogue inputs. Some signal conditioning is necessary, such as a frequency filter, an amplifier and a mixer to down convert a select frequency band into workable levels, this process separates signal from the carrier wave. It was introduced in section 2.4 and illustrated in Figure 2.7 and Figure 2.8.

3.2 Analogue to digital converter

A representation of the ADC step can be seen in Figure 3.3.

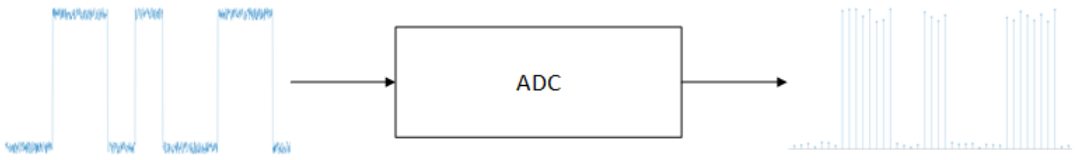


Figure 3.3: ADC stage.

After conditioning, the signal is then fed through an ADC. The output of this stage will be a frame of discretized and quantized real values.

3.3 Digital filter

A representation of the digital filtering step can be seen in Figure 3.4.

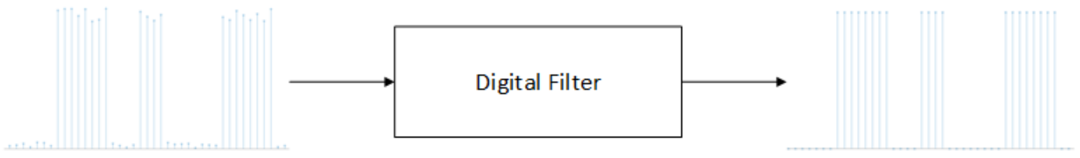


Figure 3.4: Digital filter stage

A second filter is applied, this time on the digital domain. This filter has a much narrower passband, it serves as a mean to remove unwanted noise leftover from the previous while also muting any Direct Current (DC) component in the signal. The reasoning for using two separate filters is further explained in section 4.4.

3.4 Decimation

A representation of the decimation step can be seen in Figure 3.5.

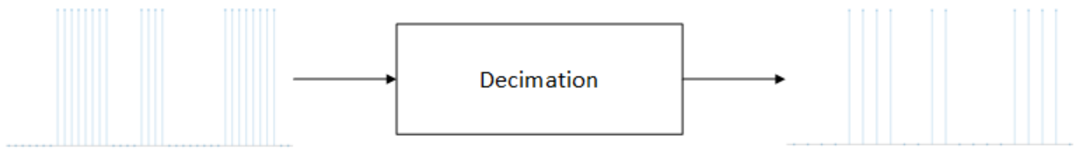


Figure 3.5: Decimation stage.

Depending on the setup and configuration, in some cases oversampling might be occurring, as explained in section 4.5. This might, during the following stage, cause unnecessary Central Processing Unit (CPU) drain and, in more severe cases, the loss of signal as described in subsection 4.2.3. For this reason, and in order to provide an additional degree of freedom to the end user, this optional decimation stage is included in the architecture of this project. Its ratio is configurable and can be set to 1:1, effectively disabling it. In most cases the ratio should be set to the highest possible value that does not compromise the quality of results. Depending on user configurations, the output of this stage is identical to the previous one or a subset of it.

3.5 Data recovery

A representation of the data recovery step can be seen in Figure 3.6.

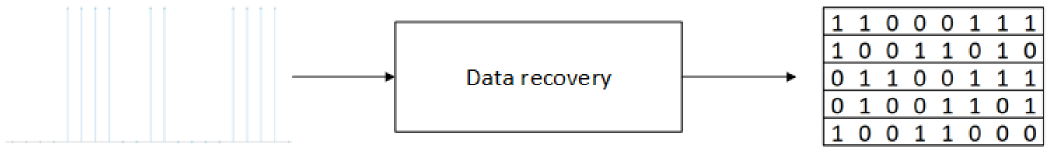


Figure 3.6: Data recovery stage.

Having prepared the samples via filtering and decimation, it is now time for the most CPU intensive aspect of the project, data recovery. Given that the adopted modulation scheme for this project contains one bit per symbol, the decision is binary. Selected input samples are compared against a condition (a quantization threshold), and depending on the result a value of ‘0’ or ‘1’ is obtained.

3.6 Parsing

A representation of the parsing step can be seen in Figure 3.7.

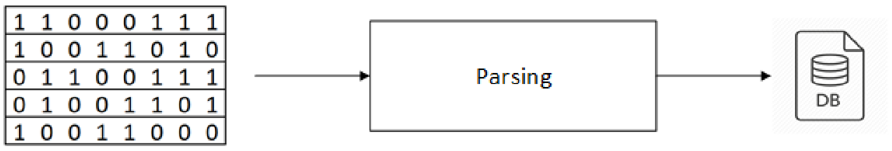


Figure 3.7: Parsing stage (icon used from [5]).

After recovering the bits, it is necessary to parse the bitstream in order to detect frames and decode them. A communications protocol has been agreed upon with the developers of the sensor. As explained in subsection 4.7.1, expected within each packet are fields such as a preamble, a payload and stop bits. It is up to this module to identify packets, validate them through implemented error detection methods and finally extract the payloads from validated packets and store them in a database to be used later in an application specific manner.

The project is now architecturally complete, inputs and outputs have been defined both at a global level but also in each of the modules. On the next chapter, an implementation based approach will be exposed, detailing at a more technical level the chosen development avenues and accepted trade-offs.

Chapter 4

Reader implementation

4.1 Introduction

In this chapter, the central implementation aspect of this thesis will be introduced. First the software package will be introduced, followed by a description of the data flow and resource management methods. After that a more detailed description of the pipeline will be detailed, starting with the signal inputs and proceeding all the way to parsing and validation.

4.2 Software

ID.all (pronounced *ideal*) is the name given to the entire software package developed as a management tool for a platform of communications capable of listening to a backscattering sensor. Powered by Python [16] and developed with the Raspberry Pi [15] platform in mind, this script is able to, through an antenna, receive RF ASK modulated signals from the air, demodulate and parse them into network-ready database entries.

Python was adopted for this project due being an already established coding language with a multitude of available libraries online. The Raspberry Pi was adopted for cost and easiness of development reasons.

This software makes use of third party libraries such as `pyrtlsdr` [22], `scipy` [28], and `RPi.GPIO` [29] but also custom modules developed in the scope of this project which will be explained further in this chapter.

In Figure 4.1 we can see a simplified design of this project.

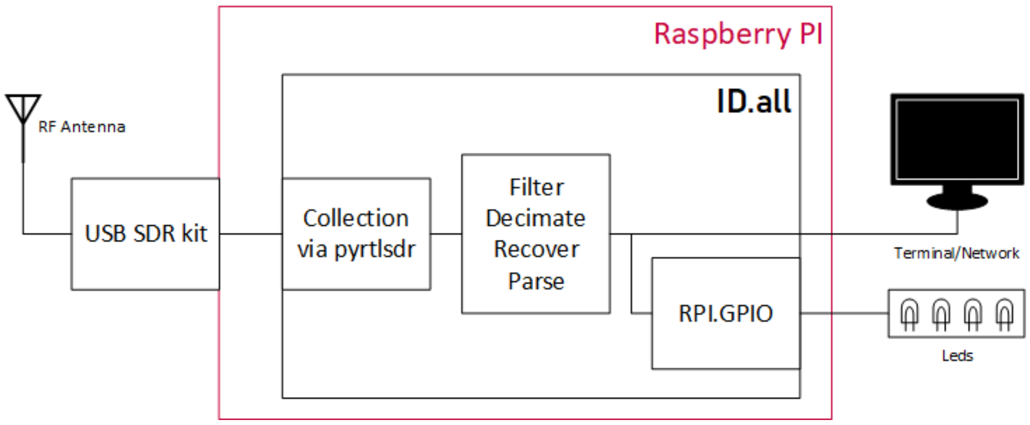


Figure 4.1: Simplified signal processing pipeline.

4.2.1 General structure

A modular approach was taken to the different blocks of software that compose this project. The goal behind this decision was to have a product that was easy to maintain and build upon without sacrificing flexibility or overall result quality.

A main module, titled *IDall.py* is called upon by the user via terminal, it acts as the main function. It has two major roles, the first one being defining the sequence of signal processing steps described in chapter 3, each of these steps is performed by invoking it's relevant library and providing it with its necessary input parameters. Its second major function is thread management, this is explained further in subsection 4.2.3.

On Figure 4.2 we can see a diagram of ID.all's internal dependencies.

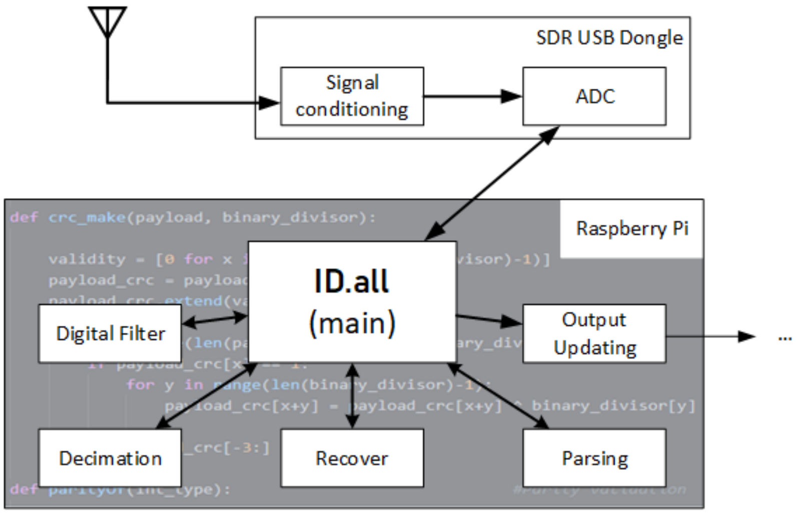


Figure 4.2: ID.all's general architecture.

By adopting this implementation design, instead of serial one in which each successive step is being called by the previous module, eventual required editions to the code become easier, as new modules can be added or existing ones edited without requiring changes or compromising the overall structure.

This chapter will focus on explaining in detail the essential software parts, the critical path. Even though a few other non-essential libraries, also developed by the author in the scope of this project, are included in this package, they will only be mentioned in Appendix A. These extra libraries preform functions primarily aimed at increasing the productivity of future developers.

4.2.2 Data flow

The pipeline of data goes through many different stages, and throughout the process is handled by two different types of threads. A producer and a consumer. The point of contact between these two threads is a FIFO. This FIFO is filled by the producer thread and emptied by the consumer thread. An operating scheme can be see in Figure 4.3

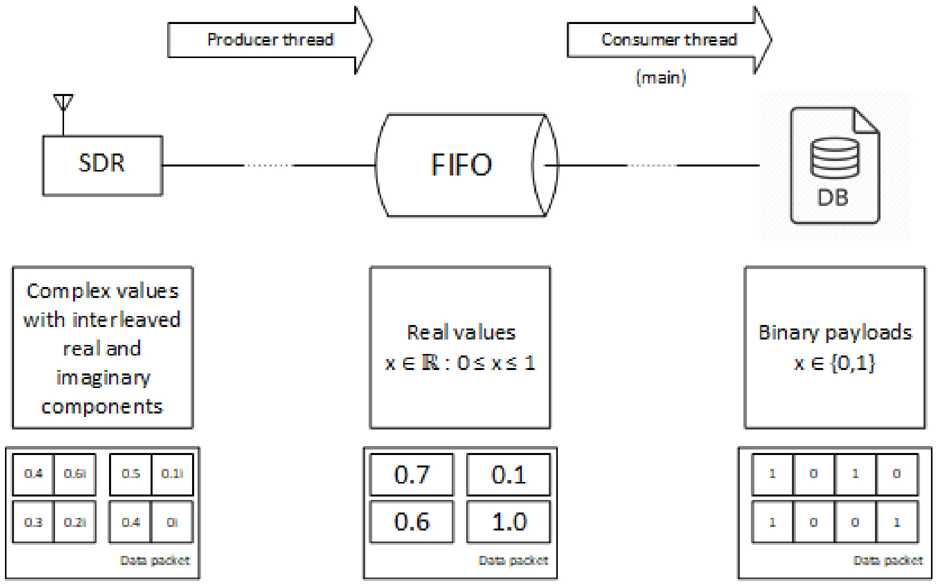


Figure 4.3: ID.all's data flow architecture. The data travels through steps in packets, whose sizes are configurable by the user. Ratios between packet sizes in this picture are not representative.

As further detailed in subsection 4.2.3, each regular iteration of the script's main loop deposits one item into the FIFO, but also pulls another from it, thus ensuring its occupancy remains at a steady level of 1 to 2 items throughout interpretation, preventing bottlenecks or overflows.

With such stable and low FIFO occupancies, why even use one at all in detriment of simpler data storing mechanisms? The answer is scalability. Even though the software itself is only currently planned to handle one SDR kit and one sensor at once, this can potentially change in the future. Future changes might bring increases in throughput and even imbalances between data production and consumption, thus it was decided as the best course of action to leave this growth door open.

4.2.3 Thread structure

Let's first approach the resource allocation aspect of thread management.

As explained in the previous section, during script interpretation two different threads work in tandem to ultimately deliver to the database the payloads contained in incoming RF ASK backscattered signal. The first thread, the producer, interacts with an RTL-SDR USB dongle via the `pyrtlsdr` software library to convert RF signal into a discrete and quantized digital list of samples, which are then stored into a FIFO. Because currently the dongle works exclusively in burst-mode, a new instance of this thread is created for each burst of data. These bursts can also be called frames. In Figure 4.4 we can see a state machine representation pertaining this thread.

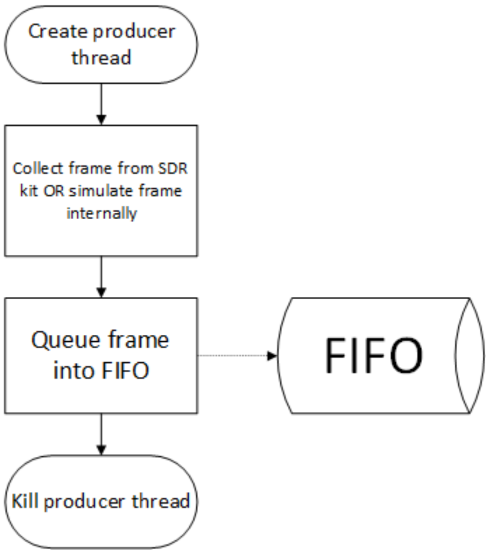


Figure 4.4: ID.all's producer thread state machine.

The second thread, the consumer/main, removes items from the FIFO and, after a demodulation/validation process, obtains a binary payload which it stores into memory. This output can then be displayed in a terminal or passed along into the network, at the user's discretion. This thread also manages the creation of producer threads. We can observe its state machine in Figure 4.5.

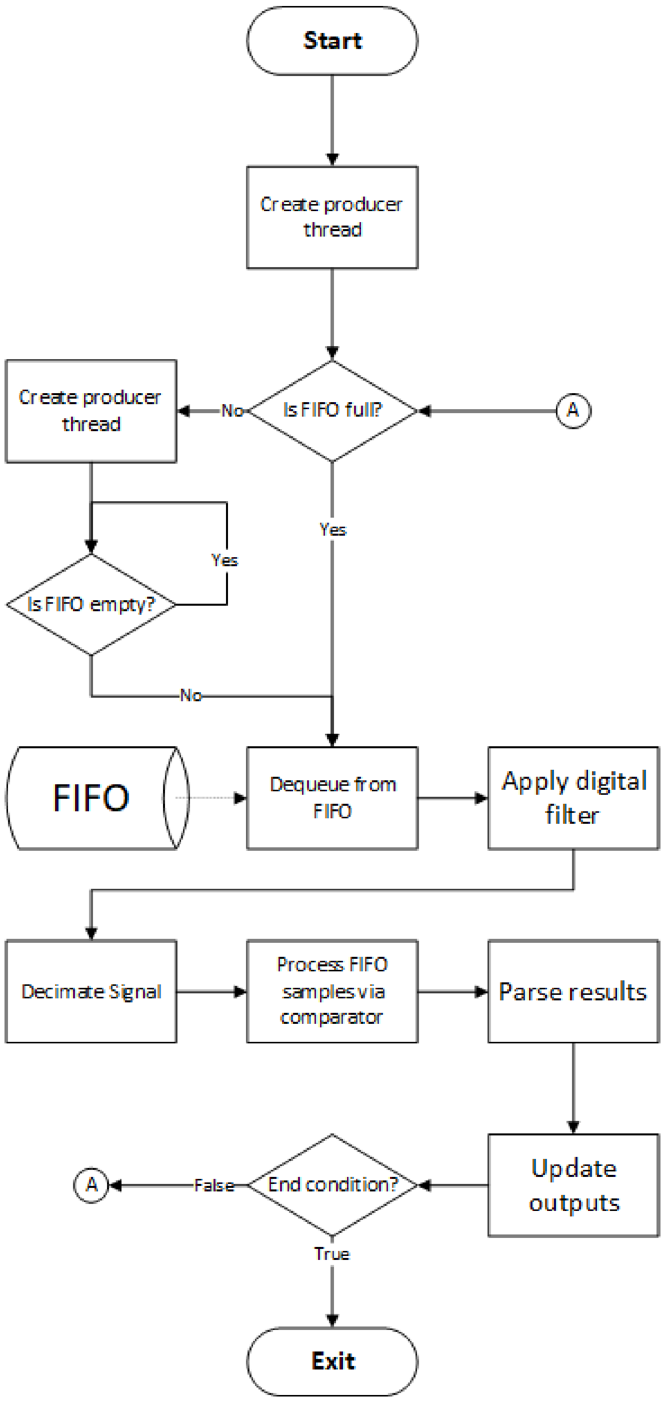


Figure 4.5: ID.all's main thread state machine.

FIFO pre-initializing Multi-threading was implemented in order parallelize some of the required steps and thus reduce the critical path's time duration. The 'Create Collector Thread' step at the start of execution appears unnecessary, but is not so. The initial number of items in the FIFO is 0, this happens because producer threads are created once during each consumer thread iteration, and only place one item into the FIFO per call, and because the main thread consumes an equal amount itself per loop cycle. Without this initial call, FIFO occupancy would almost always be 0, except for a few infinitesimal moments after the conclusion of each producer thread, where it would equal 1. Execution between the main thread would always be halted in the ‘is FIFO empty?’ decision until collection was finished, effectively reverting the process back from parallel into serial. In Figure 4.6 we can see a time-line of side-by-side thread executions exposing this problem.

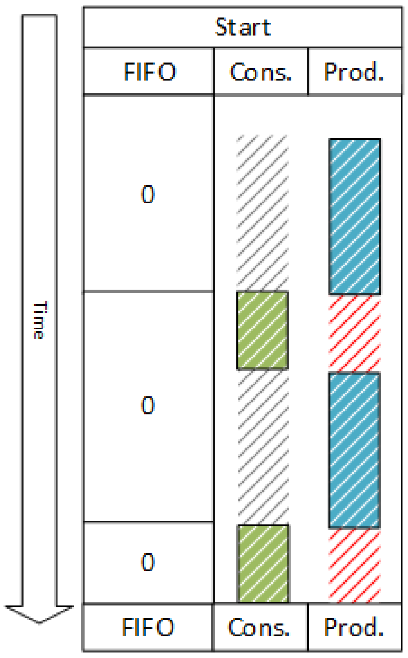


Figure 4.6: Side-by-side thread diagram, without FIFO pre-initialization. Green areas represent signal processing intervals while blue areas represent signal collection. Red areas represent signal not collected due to poor software optimizations and should be avoided.

As the script begins interpretation, it attempts to both collect and process signal. But since no signal was collected yet, the FIFO is empty and there is nothing to process, so execution remains halted at the ‘is FIFO empty?’ step of Figure 4.5 until the producer thread concludes its job and deposits data on the FIFO. The data is then immediately consumed by the produced thread and as such the FIFO occupancy only attains the value of 1 for an infinitesimal moment.

However, if we include the mentioned pre-initialization step, the situation changes, and becomes what we can see on Figure 4.7.

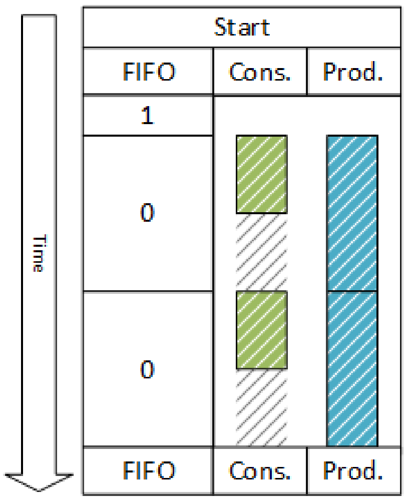


Figure 4.7: Side-by-side thread diagram, with FIFO pre-initialization. Green areas represent signal processing intervals while blue areas are equivalent for signal collection. Notice the absence of signal loss due to poor optimizations.

With this implementation the data consuming main thread always has at least one item in the FIFO to process, with each new producer thread producing data to be consumed in the next consumer loop iteration, and correspondingly, each of these iterations consuming data that was produced by a thread initialized during the previous cycle. Because this is the currently working state of the program, from now on, when explaining other aspects, it should always be assumed a pre-initialization was done.

Producing vs consuming data - time interval ratios This thread forking approach raises other details that benefit from an explanation. Producer threads have an explicit lifetime, they are started, request data from the USB dongle, deposit this data into a FIFO and then are terminated. This chain of events takes a certain interval of time which is more or less constant. Conversely, the main thread is operating in a loop, with each iteration also having its own approximate temporal duration. Due to the fact that the software took a modular approach, time intervals may vary depending on user choices and settings. In some cases different modules execute the same function but with varying degrees of speed optimization, it's up to the user to select the one most suited to his needs (see subsection 4.6.1 for more information).

Two possibilities arise here, either the producer threads' lifetime is longer than one iteration of the main loop, or it is shorter.
For cases where the consumer concludes its iteration faster than the producer, we have a situation where data is being consumed faster than it can be produced. An indefinite loop was purposefully introduced in the script (and can be seen in picture 4.5) with the exit condition '*FIFO not empty*' in order to throttle data consumption down to

production levels and manage this bottleneck, as otherwise there would be no data to consume and the thread would simply be continuously polling an empty FIFO for data, wasting system resources in the process. For cases where the producer terminates before a new main loop iteration, we have a situation where data is being produced faster than it can be consumed. Because a new producer thread will only be created after the consumer is done processing its current batch of information, a 'standby' time during which there are no producer threads active is inevitable. Although this prevents FIFO overflow it also represents a data throughput decline via loss of opportunity to collect signal and is to be avoided if possible. This 'standby' time is not to be confused with the deafness period described later in subsection 5.3.1 even tough both have the same practical additive outcome, which is sampling downtime.

In Figure 4.8 we can see a comparison between both of those cases.

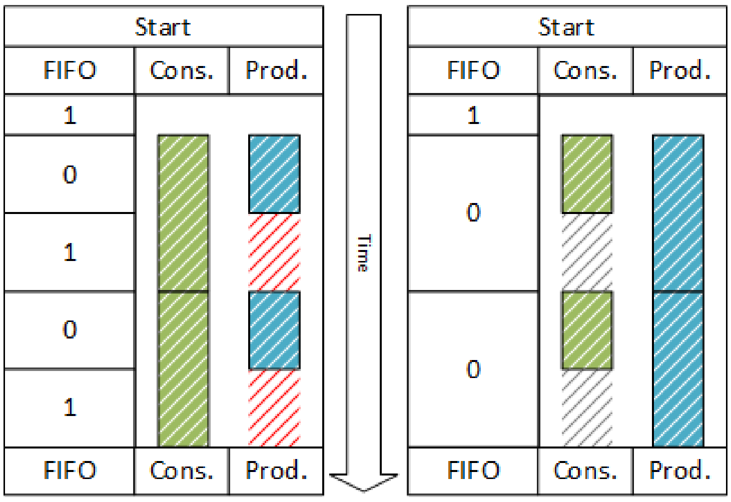


Figure 4.8: Side-by-side thread diagram, on the left is the depicted a situation where signal processing takes longer than signal collection, and vice-versa on the right. Green areas represent signal processing intervals while blue areas are equivalent for signal collection. The column on the left in each table represents FIFO occupancy level.

A stand-by on signal collection is considered a more serious issue than an equivalent standby period on signal processing. This is because while the first represents an additional loss of information, assuming the transmitter is continuously emitting, the latter is actually a growth avenue. By only taking a fraction of the time to process the signal, we can host several producer threads in parallel, each handling a different RF front-end, and have all the data produced by these threads being processed during a different time-slot of the consumer thread.

Currently, two of the three comparators included in this thesis, Pass By Zero (PBZ) (subsection 4.6.3) and FAST (subsection 4.6.5) are able to process signal faster or as fast as it can be generated.

4.3 Signal input

Even though under regular circumstances the incoming signal will be provided via the work of the producer thread and obey the specifications described in the previous section, it's actual origin will be one of two possibilities. The first one is the harvesting of RF waves from the air, this is the standard option and the one described in this section. A second one was added as a feature for debugging purposes, the software also has the ability of generating/simulating the signal internally, more can be read about this additional feature on section A.1.

4.3.1 Signal reception

Using the publicly available, `librtlsdr` [30] and `pyrtlsdr` [22], which are respectively, a C language package developed to turn the RTL2832U [31] into an SDR kit and its Python language wrapper, the Raspberry Pi connects to the kit and harvest samples of electromagnetic radiation levels from the environment, thus fulfilling the architectural requirements for signal conditioning and analogue to digital conversion mentioned in chapter 3.
Several input parameters are necessary in order to calibrate the readings [32] [22], amongst them, the user will have to define a tuning frequency and a sampling rate.

Once script interpretation begins, a producer thread is created and tasked with querying the IQ sampler embedded in the SDR kit for an array of complex information with a number of entries equalling twice the amount of samples harvested, consisting of their interleaved real and imaginary components. Since we are using ASK, a modulation that does not care about phase, we convert these entries into their absolute value and discard the originals, thus halving list size. The result is a parametrization of the ambient RF power levels at the previously specified tuning frequency, as seen in 4.10. The signal power in Watts is unknown to the user, and cannot be easily extrapolated due to an un-configurable Auto-Gain Controller (AGC) within the RTL2832U, this does not cause issues as, firstly, the process is abstracted, requiring only the differentiation between two distinct power levels, and secondly, more focus is placed on what happens from this point onwards, instead of looking back.

Additionally, this chosen RF front-end also applies a low pass filter with a cut-off frequency equal to the sampling rate set by the user. As far as we know, there isn't publicly available documentation nor can it be deactivated.

On Figure 4.9, we can see a complex plane plot of the samples as they are received from the RTL2832U, before the calculation of their absolute values.

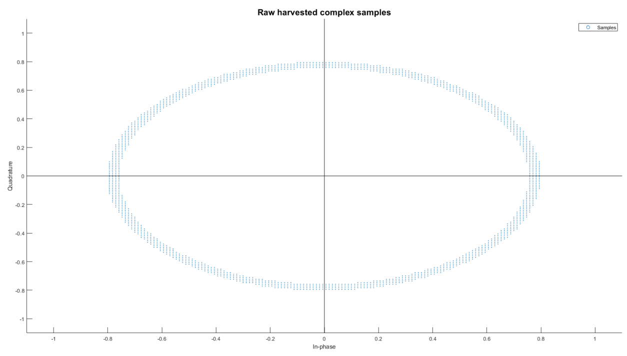


Figure 4.9: Asynchronous communication between sensor and reader. The two different ASK amplitude levels are too close to be visible in this plot. IQ sample collection result.

After we calculate their absolute values and remove the DC component we get an array such as the one plotted on Figure 4.10.

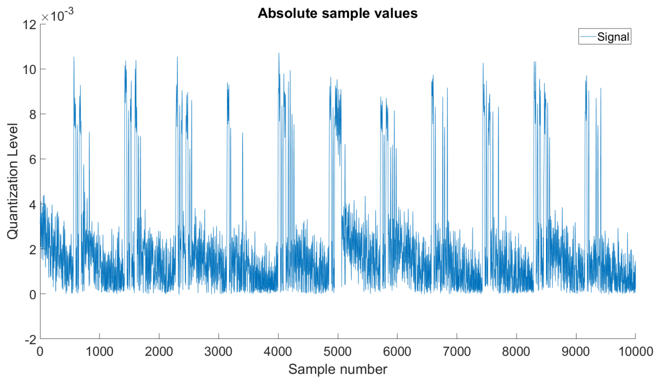


Figure 4.10: Absolute values of samples after removal of DC component. 11 complete packets are visible in this figure. All of them inter-spaced by a silence period.

If we take a closer look into one single packet from picture 4.10, we obtain Figure 4.11.

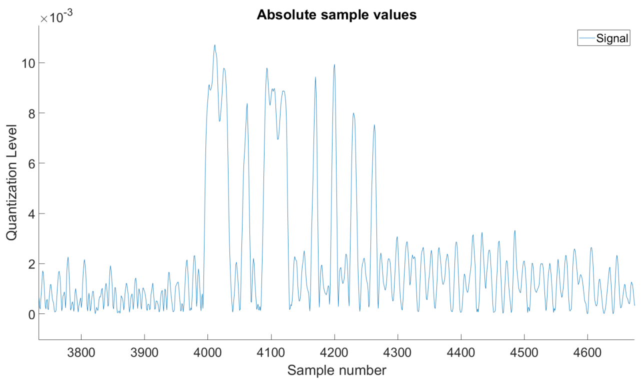


Figure 4.11: One packet, preceded and followed by silence and noise.

Limitations Using this RF front-end kit comes with its own drawbacks. They come in the form of tuning frequency, sampling method and rate limitations. The tuner bundled with the adopted hardware, Rafael Micro R820T, has a tuning range of 24 - 1766 MHz. As for the sampler, it can record samples at a rate from 226kHz up to 3.2 MHz, although some unreliability exists towards the higher end of this interval. It's specifications state that the highest sampling rate at which sample loss does not occur is 2.56 MHz [23].

Additionally, due to the quantization intervals provided by the ADC, the result of this process in proper working conditions is an array of values, either a real or complex part of a sample, both between 0 and 1. We can observe in figure 4.9 a scatter plot of the raw complex samples, note that due to the format's nature, and as pictured in Figure 4.12, it is possible to collect samples with a higher absolute value than 1, this however means that ADC saturation is occurring, information is potentially being lost and the software gain or antennas need to be adjusted.

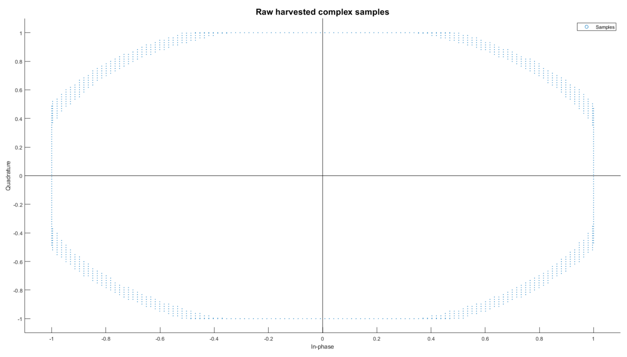


Figure 4.12: Asynchronous IQ sample collection result. Notice the saturation around the horizontal and vertical edges due to inadequate antenna/software gains.

4.4 Digital bandpass filter

After interfacing with the RF front end and collecting the results of its discretization and quantization processes, the next step is to filter any unwanted information out of the signal, as described in this section.

In order to reduce noise and facilitate the comparator's work downstream, a band-pass filter is applied to the signal. This is implemented by importing and using the `signal` module from `Scipy` [28], a Python library. This was implemented even though the SDR dongle already provides a low pass filter, for two reasons. Firstly, in some scenarios the sampling rate will be considerably higher than the symbol rate. The SDR kit internally calibrates its cut-off frequency to the same value as the user provided sampling rate. It also forces a minimum value of 226kHz on the sampling rate, as mentioned in subsection 4.3.1. This leaves, in some cases, with sub 226kHz signal symbol rates, an undesirable pass-band region between actual signal symbol rate, and internal SDR filter cut-off which would allow noise to contaminate other steps further down the pipeline. Secondly, sometimes lower frequency components, such as DC, might be undesirable, and the previous low pass filter alone would be ineffective at removing them.

For this step, a Butterworth filter is designed with user selected cut-off frequencies, these are set by default at 1 and 3700Hz, this is because for most testing scenarios the symbol rate of the signal equalled 3600Hz. The lower cut-off frequency exists in order to remove any DC and extremely low frequency components from the signal while the higher is set in order to remove all the harmful noise above the desired symbol rate.

As an example, in 4.13 we can see the simulation of an order 2 Butterworth bandpass filter with cut-off frequencies of 5 and 3650 Hz.

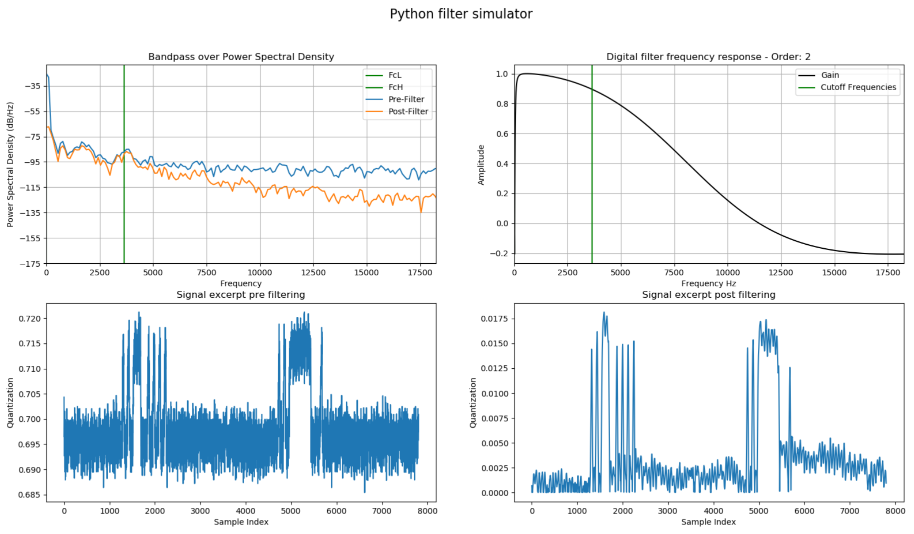


Figure 4.13: Filter projection plotted with the simulation tool described in subsection 4.7.1

A secondary effect of applying this filter is that, due to the removal of all the harmonic components of the desired signal located above the higher cut-off frequency, the signal's shape itself changes, adopting smoother, slower, more rounded transition edges (as demonstrated in 4.34). This might have effects down the line, specifically during threshold setting and synchronization attempts. Increasing the higher cut-off frequency might reduce these effects at the expense of increasing the amount of noise post-filtering. Care must be taken while raising the order of the filter, as it might cause gain considerable gain variability in the passband region. For the pictured example, second order was the maximum the code allowed before compromising filter results.

4.5 Signal decimation

Due to hardware restrictions on both sides, the RF front-end and tag, substantial oversampling occurs. Assuming the minimum possible sampling rate allowed by the RTL2832U ADC sampler (226kHz) is paired with the highest possible symbol rate supported by the backscattering tag used during testing rounds (3650Hz) the expected amount of samples per symbol is:

$$\frac{226000}{3650} \approx 61.92 \tag{4.1}$$

This is considerably higher than required for signal demodulation. If the data recovery algorithm is not properly optimized for speed, complications might occur due to the forced oversampling of the signal. In order to prevent this, decimation is applied after the filter and immediately before the comparator. Decimation order is a configurable setting and must equal an integer greater or equal than 1.

By applying a decimation order of 10 on the given example, the number of samples per symbol can be brought down from 61 to roughly 6, as seen on the stem plot in 4.14.

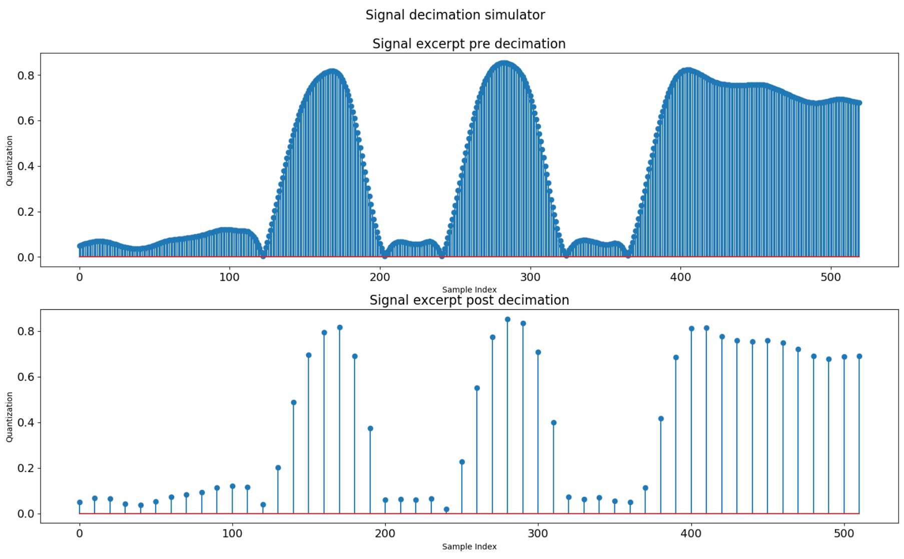


Figure 4.14: Signal decimation example, order of 1:10

By doing this, the amount of samples passed on, and therefore the CPU time required to process them, gets reduced while still maintaining enough information in the signal not to compromise the next steps.

4.6 Data recovery

In order to explore different possibilities, in terms of result accuracy and processing performance, several different options for this stage were projected and implemented. Each of them preforms the same task, but in a different way, and aiming to optimize different aspects.

Four different comparators were developed, they were named TMC,the PBZ, Smart Pass By Zero (SPBZ) and FAST. Three of them were implemented, SPBZ was eventually discarded. During initial development stages, viable distance between antennas was seen as the defining quality metric, which led to the development of TMC. It excelled at interpreting weak signals substantially contaminated by noise, at the expense of increased necessary processing power. This necessity would eventually clash with a development environment based around a Raspberry Pi platform, which has limited processing power.

As development progressed, and because the chosen platform, avoiding high CPU load scenarios became a more concerning issue. Eventually, it turned into a strong enough pressure to justify a different, lighter approach, and thus the second iteration, titled PBZ, materialized. It applied simpler temporal synchronization algorithms to achieve similar results much faster. And because the approach was simpler in nature, it also brought about vulnerabilities not present in TMC. Noise was a major hindrance to synchronization attempts implemented in this new comparator. Attempting to iron them out branched a new, eventually unsuccessful approach, which was named SPBZ.

Feeling that these improved implementations still weren't taking full advantage of adopted small packet sizes, an even faster and lighter method was developed. Titled FAST, due to its main focus on execution speed (not an acronym), this last implementation manages to be the fastest one at the expense of expected loss of synchronization on longer packets.

The data recovery module is essential and the most complex piece in the entire architecture. Handled by the main thread, it performs the signal pattern recognition part of the demodulation process, taking as an input the filtered decimated samples and, after applying an ASK demodulation method, returning a list of binary values which are considerably easier for the program to manipulate. It has been customized to work not only with the burst mode requirement of the used SDR kit but also with the silence periods between packets transmitted by the sensor.

The following paragraphs detail a list of guiding principles behind the inner workings of each algorithm.

TMC - Transition Match Comparator This comparator starts by analysing signal variance in order to locate packet timings, and after doing so discards samples outside a packet neighbourhood. It takes the user configurations such as sample and symbol rates to estimate distances between symbol boundaries and uses this information to build a 'ghost grid' of symbol transition timings. After this, it attempts to achieve timing synchronization by matching this grid to the signal based on the degree of equivalence between the ´ghost' and the real transitions. Once this is achieved, a threshold is applied and the resulting bitstream passed along to the next module.

PBZ - Pass By Zero Based on the assumption that transitions between the two symbols force the quantization levels to pass by an intermediate constant pre-established value, this comparator uses the timings on those intersections to establish synchronization. It then applies a threshold to obtain the bitstream.

SPBZ - Smart Pass by Zero Currently non-functional. Identical to PBZ, with added noise management methods.

FAST When the signal's quantization levels raise above a certain threshold after a prolonged period of silence, this comparator assumes a packet reception is taking place. It then selects a few relevant samples in the packet beginning and attempts to match them with the expected preamble. If it succeeds, it re-samples a following pre-set interval of samples at adequate times, compares the result against the same threshold and outputs a bitstream.

Three of these four are currently usable and choosing the most adequate is up to the final user. Some guiding points are provided on the next subsection.

4.6.1 Choosing an algorithm

A summarized list of data recovery algorithms and adequate contexts can be see in Table 4.1 and Table 4.2, respectively.

Algorithm	Advantages	Drawbacks
TMC	Best noise immunity Harmonics filtering immunity	Very CPU intensive Signal variance dependent
PBZ	Lightweight CPU-wise	Biggest noise vulnerability Vulnerable to harmonic filtering
SPBZ	-	Extremely CPU intensive
FAST	Very lightweight CPU-wise	De-synchronization on longer packets Noise vulnerability

Table 4.1: Comparison between developed algorithms

Algorithm	Adequate context
TMC	Powerful CPU Noisy environment
PBZ	Long packet sizes Low noise
SPBZ	-
FAST	Short packet sizes Low noise

Table 4.2: Ideal context for each algorithm.

Having now introduced an initial description, advantages, drawbacks and ideal contexts for each algorithm, let's analyse and describe them in detail.

4.6.2 Transition Match Comparator

As we step into the data recovery part of this thesis it is important to establish to the reader a visual representation of how the software identifies areas of the signal where information can be extracted from. This subsection will explain things in a more visual way, with extra attention to detail. The remaining algorithms will be explained in a more succinct manner with references to this subsection when relevant, so as to prevent redundancies.

General diagram

In Figure 4.15 we can see a diagram describing the general process flow of this algorithm.

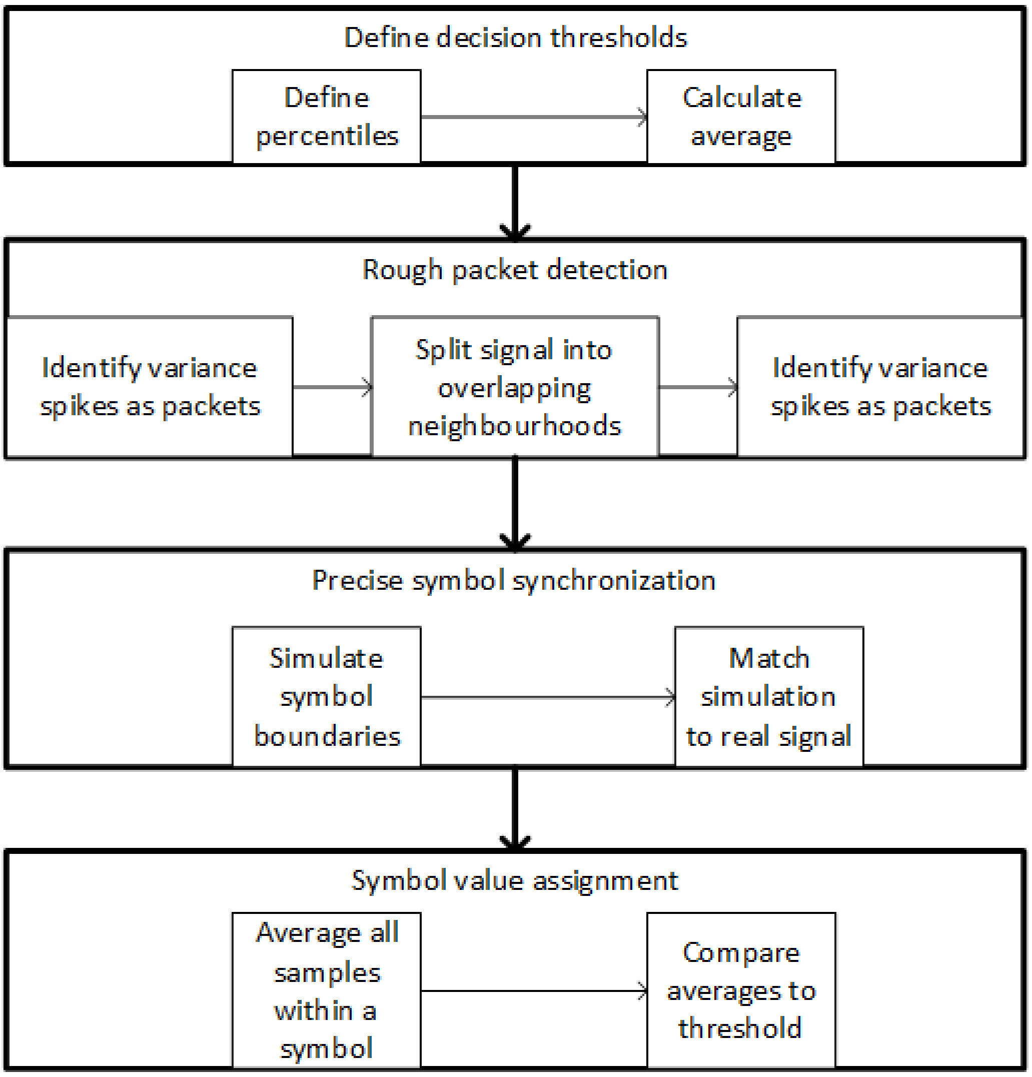

Define decision thresholds
Define percentiles
Calculate average
Rough packet detection
Identify variance spikes as packets
Split signal into overlapping neighbourhoods
Identify variance spikes as packets
Precise symbol synchronization
Simulate symbol boundaries
Match simulation to real signal
Symbol value assignment
Average all samples within a symbol
Compare averages to threshold

Figure 4.15: Transition Match Comparison process.

Detailed explanation

Defining decision thresholds ASK modulation is relatively simple to decipher from a logic standpoint. Assuming ideal conditions, two states and 100% depth, there either is or isn't a carrier wave present, with each of these states corresponding to a different binary value. This is depicted in Figure 4.16

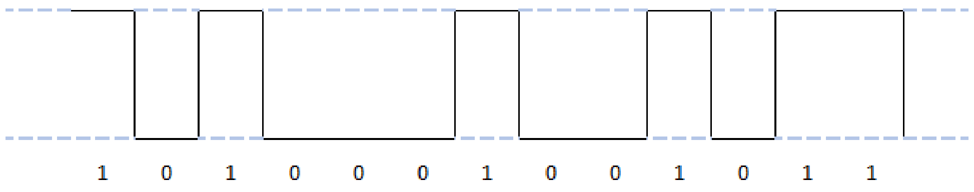


Figure 4.16: Example packet with denoted bit levels.

From the receiver's point of view this would hypothetically be a simple process, RF power would continuously be switching between two possible levels while being detected as inputs via the antenna, and a binary sequence of digits could be directly derived from the readings. However, when background noise is taken into account, the possible number of inputs can no longer be considered discrete. It isn't practical to check if equality exists between each input and an expected value, even more so because in practice there would be no exact matches. A better approach now would be to define a threshold to separate between two different binary outcomes, as exemplified in Figure 4.17.

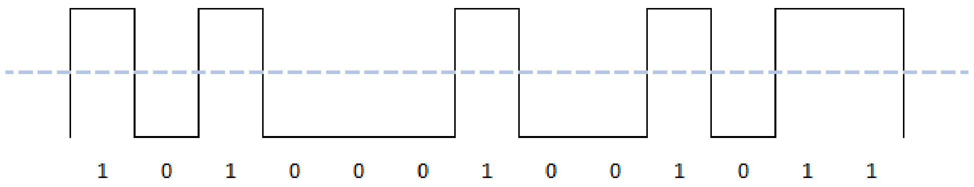


Figure 4.17: Example packet with denoted threshold.

Further issues are now raised, at what level do we define this threshold? What would happen if the relative position of the tag and reader changed between readings? What if different antennas are used? Or the temperature/weather changes? All of these factors can have an impact on signal strength and completely throw off track the

threshold calibration. It is imperative that the demodulator can maintain the ability to dynamically adjust itself in real time, and as such, rigid, hard-coded threshold levels are an unwise option here.
The adopted solution was the implementation of percentiles and geometrical centres. There is a certain nuance on this approach, one could simply declare that the threshold equals 50% percentile, or in other words, the average value of all the samples in a given frame. This approach would be suboptimal as major problems would arise later on due to the fact that imbalances in the ratio of received 0's and 1's would skew the average (and by extension, the threshold) away from its optimal position. On the other side, if we simply take the amplitude geometrical centre value of all the quantization levels we are welcoming into the process a huge vulnerability to noise bursts.

Figure 4.18 demonstrates graphically the vulnerabilities in the first approach.

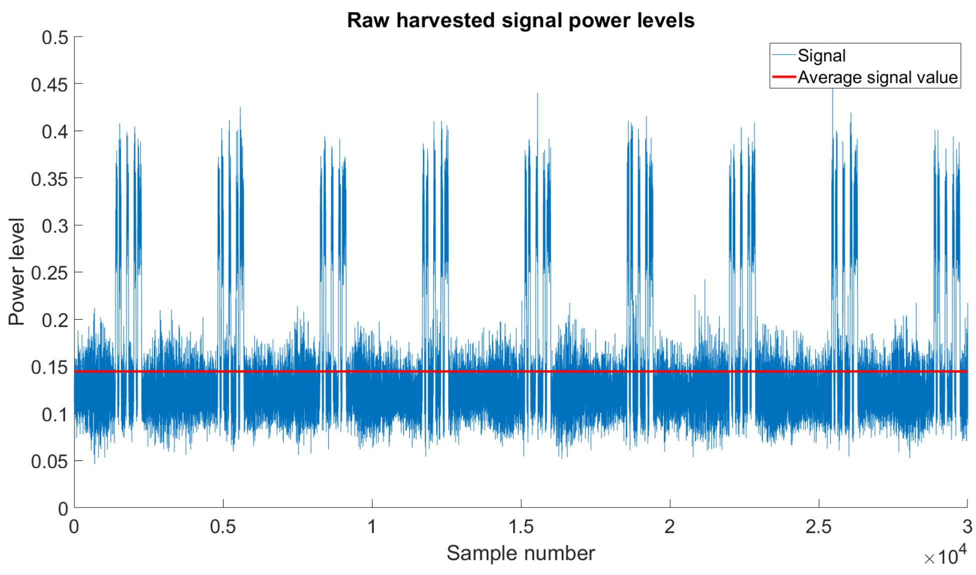


Figure 4.18: Signal excerpt containing 9 packets and its average value.

It is important to note in the above plot that due to the imbalance between packet and silence durations in the signal, the total average value can dip below even the noise level, making it an unfavourable avenue for setting thresholds.

On the other hand if we base our decision threshold on the average value of upper and lower amplitude limits of the signal, noise becomes a considerable obstacle, Figure 4.19 simulates a scenario where this method is used, but the signal reception suffers a noise burst.

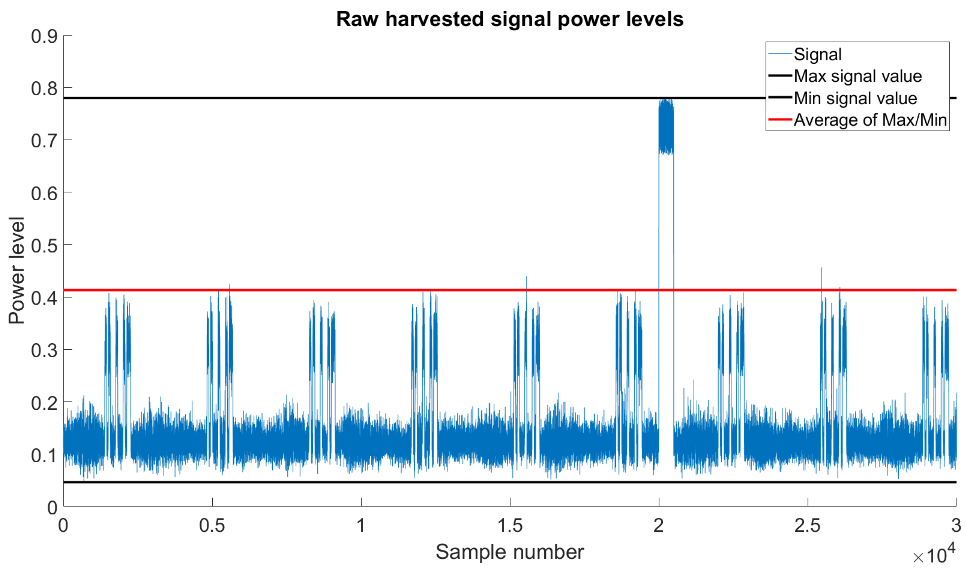


Figure 4.19: Signal excerpt with deliberate noise introduction.

As we can see, going purely by the geometrical centre of the signal isn't a good approach either, as any brief deviation from the current power levels would corrupt the whole frame of samples.

If, however, we implement a method that makes use of both these tools simultaneously, we can work around these issues. The percentile numbers themselves represent the compromise between vulnerability to bit ratio imbalances and to the presence of noise bursts and must be adjusted to the signal one is hoping to demodulate. They were determined a good fit though a process of trial and error and are currently hard coded to the value described in Equation 4.2.

$$[P_1, P_2] = [N, 100 - N],\ N = 3 \tag{4.2}$$

On Figure 4.20 we can se the visual representation of this approach.

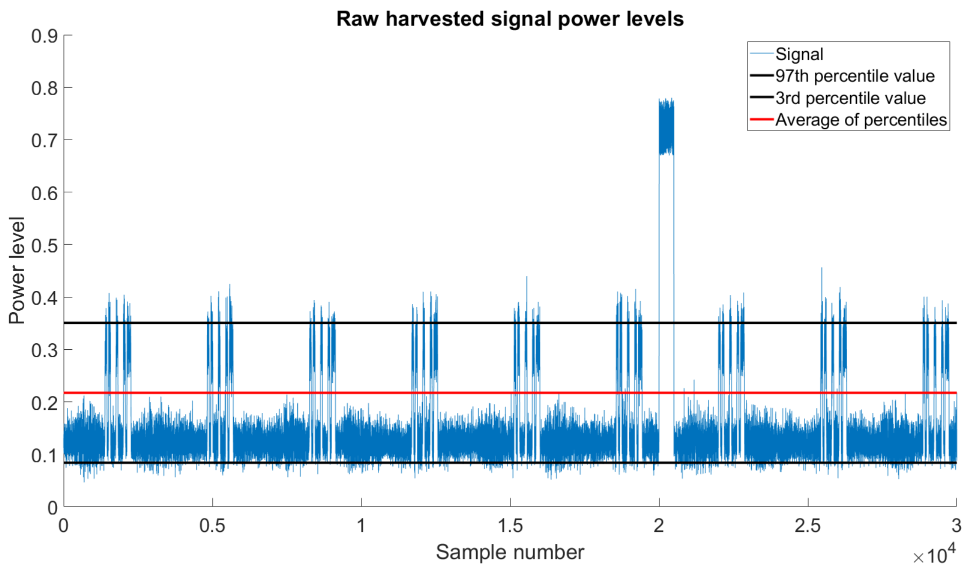
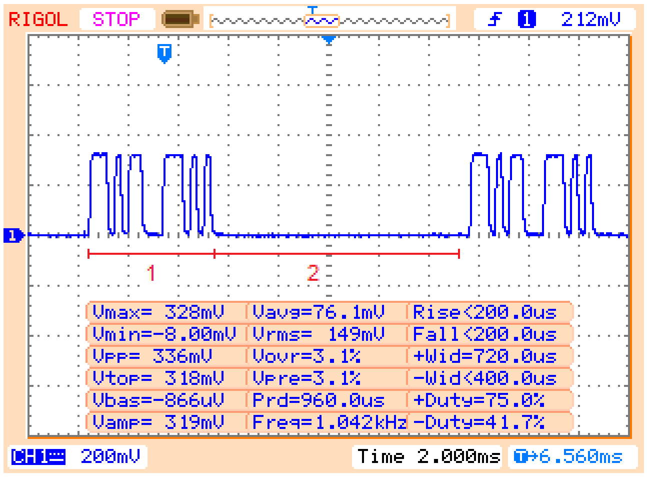


Figure 4.20: Signal excerpt with deliberate noise. Visual representation of final implemented technique.

Having reached an algorithm able to robustly set decision thresholds adequate to an ASK demodulation, the focus shifts to the next step, timing synchronization.

Timing synchronization It is important to mention here the development and testing of this software relied heavily on a specific semi-passive sensor [9]. Due to energy constraints, this piece of hardware transmits intermittently, which means that during a significant portion of the time we spend listening to the channel, there is no message being transmitted at all, just continuous periods of silence. This can be visualized in Figure 4.21.

To achieve a proper temporal synchronization it becomes then necessary to, first, differentiate between periods of transmission and silence. We can achieve this by taking successive slices of the signal and measuring the variance of all the samples values contained within. Knowing that during a silence period the power levels will remain more or less contained inside the background noise interval it becomes possible to infer that spikes in the variance of a certain neighbourhood coincide temporally with an incoming packet. This is the idea behind the synchronization method introduced in the next two paragraphs.



```
Vmax= 328mV      Vavg=76.1mV      Rise<200.0us
Vmin=-8.00mV     Vrms= 149mV      Fall<200.0us
Vpp= 336mV       Vovr=3.1%        +Wid=720.0us
Vtop= 318mV      Vpre=3.1%        -Wid<400.0us
Vbas=-866uV      Prd=960.0us      +Duty=75.0%
Vamp= 319mV      Freq=1.042kHz    -Duty=41.7%
```

Figure 4.21: Signal excerpt measured from the sensor directly via oscilloscope. Tag 1 represents a backscattered transmission. While tag 2 is a silence period.

Defining rough packet boundaries This routine begins by taking all the samples in an interval $[n-a, n+a]$ and calculating their variance value, which is then stored in position n of an array. The process is then repeated in a cycle for all the different possible a sized intervals and, afterwards, the variance array is compared against its own average value. This process allows us to, through identification of spikes in variance, locate the packets in the signal. Additionally it allows the isolation of each packet from the global samples array and its future processing separately. This is a big advantage as it represents a considerable CPU performance boost.

If we define $s[n]$ as the discrete function that contains in sequence all the samples harvested in a frame, we can obtain $z[n]$, a variance value of a $2a$ sized neighbourhood of $s[n]$.

$$z[n] = \sigma(s[n-a, n+a]), a < n < |s| - a \tag{4.3}$$

And now we can approximate all the packet boundaries' array indexes, meaning the beginning OR end of a packet, with:

$$W_F := \{n \in \mathbb{N} : z[n] \approx \bar{z}\} \tag{4.4}$$

The neighbourhood size (a) should be large enough to build some immunity to noise bursts but not so large as to drain valuable processing time.

A visual representation of the algorithm written above can be seen in Figure 4.22. In this example, the variance value has been multiplied by a factor of 10 in order to facilitate visualization.

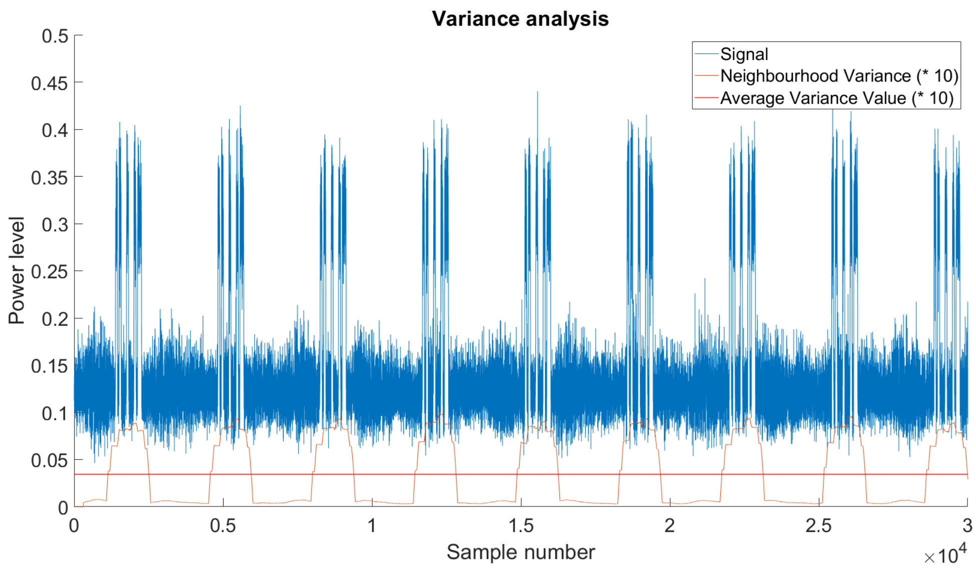


Figure 4.22: Signal excerpt with added variance plot.

In the above example, the value of a is given by Equation 4.5.

$$a = (SamplingRate/SymbolRate) * 5 \tag{4.5}$$

The sampling rate considered is post-decimation, this provides a 10 symbols worth of samples as a neighbourhood inside which to calculate variance.

The intersections between neighbourhood variance level and its global average value are considered as rough approximations to packet boundaries. If we plot vertical lines in all these intersections it becomes easier to visualize the approximation made by the script to these boundaries. Figure 4.23 is an example of this.

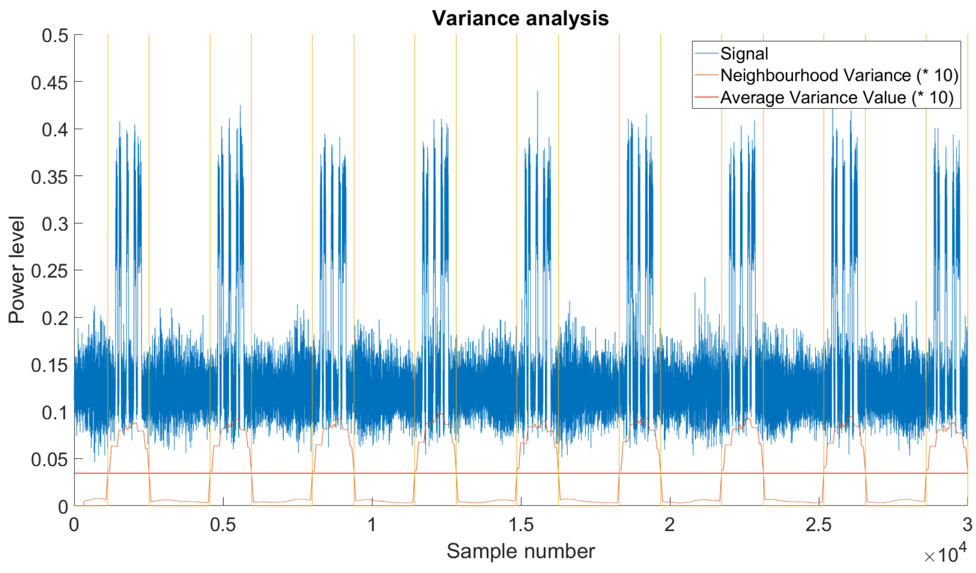


Figure 4.23: Signal excerpt with added variance plot.

It is not strictly necessary at this stage that a defined packet boundary coincides exactly with its real beginning or end, care was taken to have them be a bit before if a beginning, or a bit after if an end. This discrepancy will be compensated for further into the script execution and is left for now as tolerance room to make sure no packet is accidentally bisected.

Limitations This method is not without its disadvantages. Firstly, in order to infer synchronization from variance spikes one has to guarantee beforehand that none of the packets have too many consecutive identical bits which might cause a drop in variance in an undesirable place. The maximum number itself might be regulated by changing the value of a in Equation 4.3 With a higher value we are calculating the variance over a larger interval and therefore increasing the amount of consecutive identical bits necessary to force an error into the demodulation process. This, however, consumes more CPU time when calculating $z[n]$ in equation 4.4 . Another way to prevent this issue is to include in the most vulnerable areas of the message protocol a symbol that is always the negation of its previous/following symbol. This will force some variance at the expense of increasing packet size. Furthermore, the calculation of variance, even with low neighbourhood interval sizes (a) is still a very time consuming aspect of the process. In order to ameliorate this, an extra decimation exclusive this step was applied to the samples within neighbourhood interval [n-a, n+a]. This is because consecutive samples will in the majority of cases have very similar values thus making somewhat redundant the processing of all of them for variance calculation purposes.

Once we have established that a certain consecutive subgroup of samples contains a packet of modulated information, the next step is to determine symbol boundaries within that same packet.

Defining symbol boundaries In this context, synchronization is the process of identifying the moment at which one symbol ends and another begins, this event will from now on be referred to as a symbol boundary.

During the course of this paragraph, two types of symbol boundaries will be referred.

- **Real-world boundaries**
  The actual real-world moment at which the transmitting tag switched from one symbol to another during it's transmission, due to discretization induced errors by the ADC, this exact value is impossible to obtain, we can only estimate it.
- **Simulated boundaries**
  A 'ghost' boundary. The current best guess the comparator is making in regards to the timing of a real-world boundary.

By the end of this process, the timing difference between these two events should be reduced to the possible minimum.

After, concluding the previous step, we will already have a rough approximation regarding where to look for information, it becomes now necessary to temporally synchronize the received signal. No data can be extracted unless we know where inside the message one symbol ends and another begins. Achieving this is somewhat resource intensive and a big processing time sink. Figure 4.24 illustrates the symbol boundaries we are attempting to locate during this step, in an ideal noiseless scenario. The black vertical signal transitions represent the real-world boundaries, while the light blue dotted lines represent the simulated ones. In this example synchronization would already have been achieved as the two match.

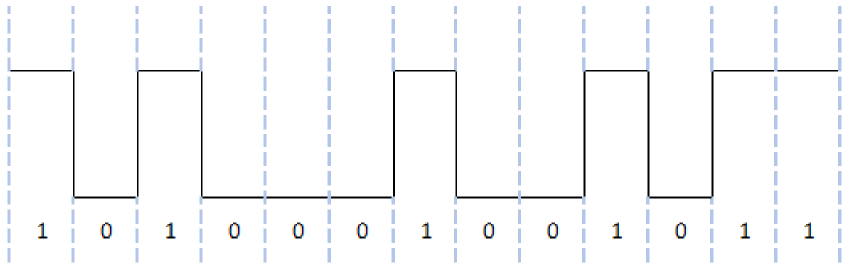


Figure 4.24: Example packet with denoted symbol boundaries.

It should be noted that for cases where the signal consecutively maintains identical symbols, there are no sharp amplitude changes. For explanation purposes we will still consider those events as real-world boundaries.

Four premises are assumed and built upon for this method, they are:

- The communication channel is asynchronous
- The sampling frequency is constant (F_s)
- The symbol rate is constant. (S_r)
- $F_s \gg S_r$

Given that we have no shared clock with the emitter upon which to locate the symbol boundaries, we must use the sequence of samples themselves as a way to estimate and eventually synchronize. Knowing that all the real-world boundaries are separated by a, known, constant (k) number of samples is helpful, so lets start by finding that value. Equation 4.6 gives us the value of k, meaning the number of samples during a symbol period.

$$k = \frac{F_s}{S_r} \tag{4.6}$$

By knowing that the number of samples in every symbol is identical, we can infer that there is always at least one symbol boundary in any k sized consecutive sample interval, defined by Equation 4.7.

$$[n, n + (k - 1)] \text{ , } with \; [n + k \leq total \; sample \; number] \tag{4.7}$$

In Figure 4.25, we take a closer look at the one of the packets identified earlier by the script during the stage depicted in 4.23.

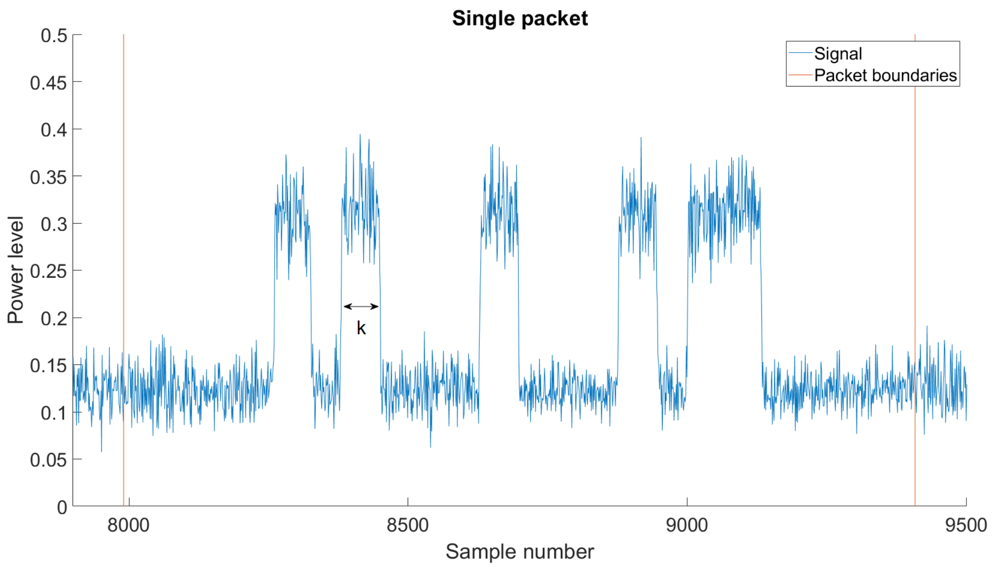


Figure 4.25: Example packet.

Lets define the left packet boundary as sample 0, and the right one as sample L so we can state this is an L sized interval.

Due to the continuous nature of the signal, plus the fact that sampling is a discrete process, we must accept that none of our samples will be an exact match of the real-world temporal boundary event, the strategy here is to find which sample is the best approximation. The next logical step becomes arbitrarily selecting a k-sized interval, and from there building a grid of simulated boundaries.

In Figure 4.25, we can see a rough estimate for the beginning and end of packet represented by the vertical brown lines.

If we apply the earlier conclusion regarding bit boundary locations from Equation 4.7 to the arbitrarily chosen interval $[0, (k - 1)]$ we can conclude that there is at least one symbol boundary inside the interval highlighted in Figure 4.26.

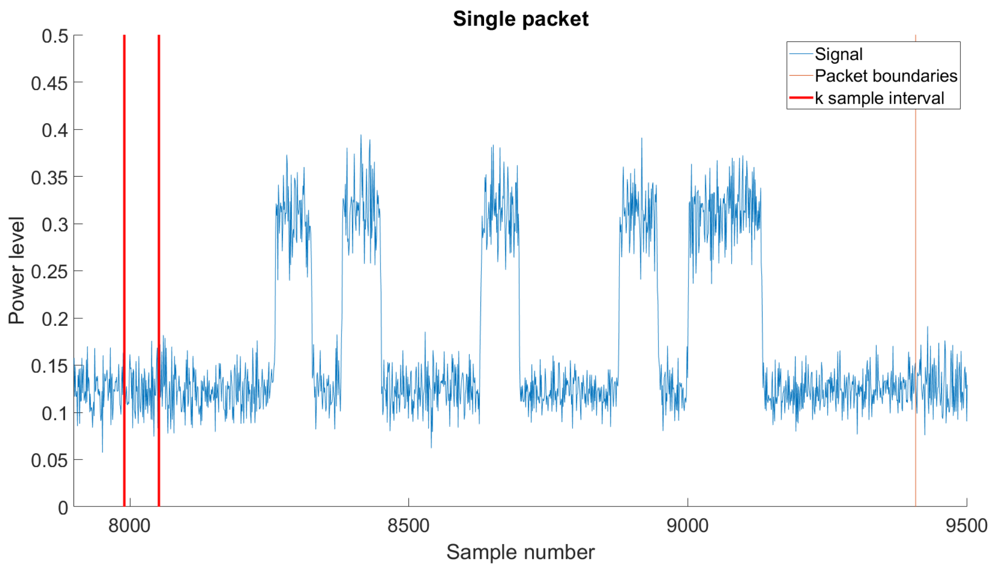


Figure 4.26: Interval $[0, (k-1)]$

Starting at sample 0, the comparator will simulate boundaries at every k^{th} sample until we reach sample L, thus we obtain the group of all simulated boundaries for this packet, designated B. B is given by Equation 4.8. Let's assume $G = 0$, for now.

$$B = \left\{ m * k + G : m \in \mathbb{N} \wedge m < \frac{L}{k} \right\} \tag{4.8}$$

After doing this, we also indirectly obtain a group of intervals, visible in Figure 4.27. These intervals correspond to the groups of all samples separated by the same two consecutive members of B, and because they are k sized, we know that during each of them there is at least one real-world boundary. Plus, knowing that real-world boundaries are equidistant from each other (due to a constant symbol rate) also tells us they will all be at the same distance from the closest left and right simulated boundaries. If we treat these distances as an offset, it becomes now a matter of adding the correct offset to the entire simulated boundary grid so that it aligns with the real-world boundaries.

We do not know however, the value of this offset. In order to determine it, we test for all possible values (given by Equation 4.9), measure the alignment quality index for each case and then select the highest quality index as the best match.

$$G \in [0, k-1] \wedge G \in \mathbb{N} \tag{4.9}$$

Spacing

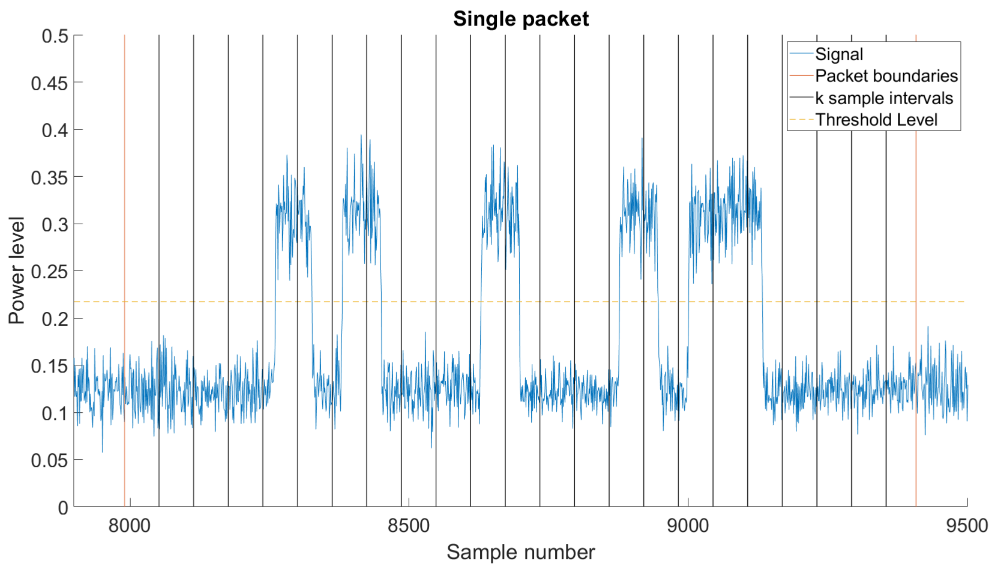


Figure 4.27: Consecutive k sized intervals.

Because each bar of the simulated boundary grid is k samples apart, there are k potential grid variations, or in other words, k non-redundant offsets.

We can measure how good of an alignment the real-world symbol boundaries are to a given simulated black grid variation by applying Equation 4.11 after Equation 4.10. In order to do this it is necessary to include the previously calculated threshold value (T) demonstrated in figure 4.20.

$$Q_{index} = \left| \sum_{x=n}^{n+k} s[x] - T \right| \tag{4.10}$$

$$A_{quality} = \sum Q_{index} \tag{4.11}$$

After calculating the value of $A_{quality}$, we increment G by one (or we can define a different step size and use it) and repeat the process until $G \geq k$.

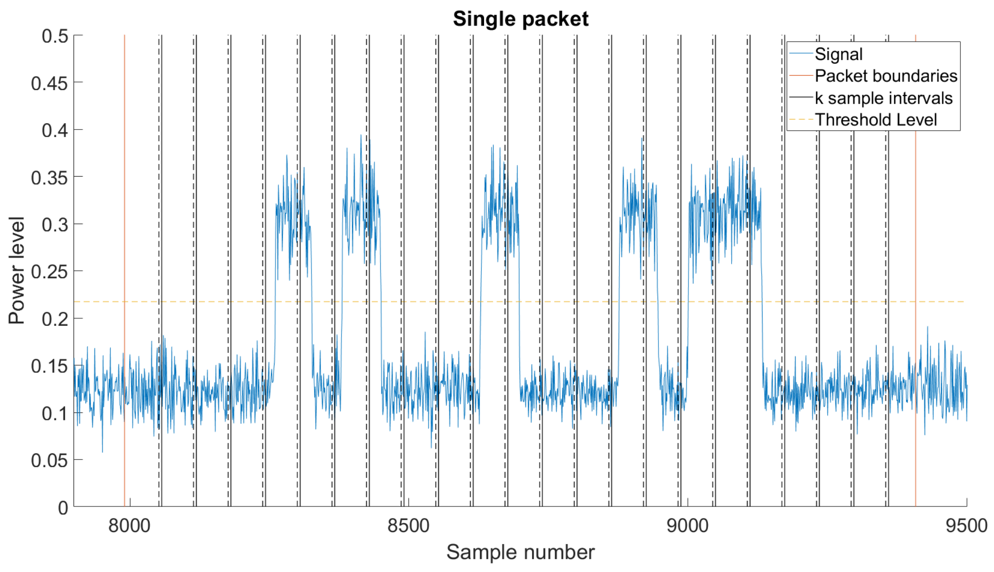


Figure 4.28: Consecutive k size intervals shifted 5 samples to the right for easier visualization

Overall, the process is repeated $\frac{k}{StepSize}$ times, which will result in a list of all the $A_{quality}$ of all the tested values of G. The highest value of this list represents the simulated 'ghost' grid that best matches the real symbol boundaries. We take the index of that value, add it as an offset to sample 0 and build the rest of the grid from there. The final result is pictured in Figure 4.29.

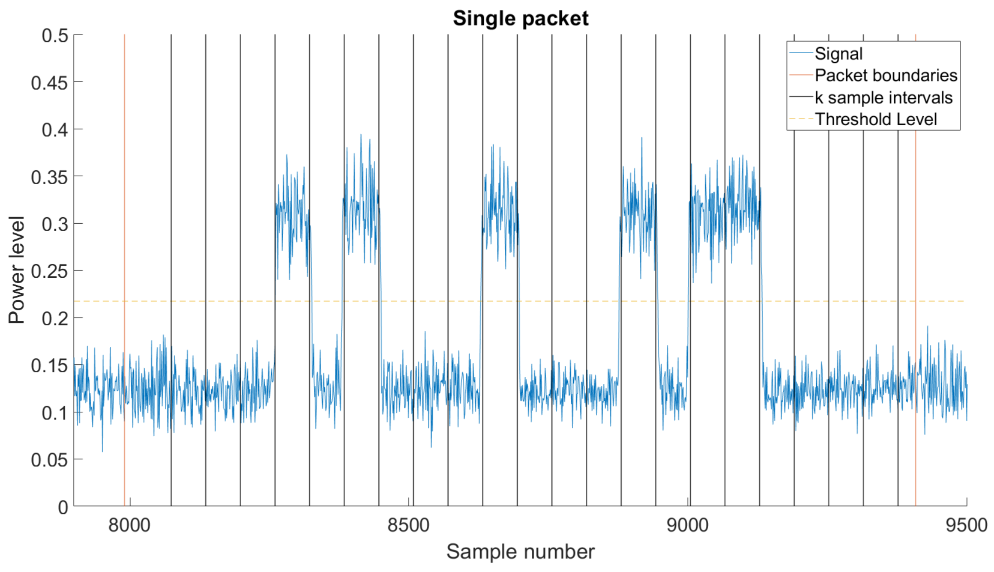


Figure 4.29: Best alignment quality between simulated and real symbol boundaries.

Now that the optimal simulated symbol boundary placement has been found, lets also define it via Equation 4.12. This is still taking into account the reference for sample

0 given earlier. Each information packet identified in the previous step will have its own origin reference.

$$S_{boundary}[i] = k * i + G \tag{4.12}$$

This concludes the temporal synchronization process. As all the relevant symbol timing information is now gathered, we can begin to start differentiating the binary values.

Symbol value assignment Now that both temporal synchronization and decision thresholds have been defined, it becomes possible to begin extracting binary values from the received signal. The average values (S_{AV}) for all the samples corresponding to each symbol (delimited by the boundaries we previously established) are calculated, and this value is then compared to the threshold level (T), the result is obtained according to Equation 4.13.

$$S_{AV}[i] = \bar{s}[S_{boundary}[i] : S_{boundary}[i+1]] \tag{4.13}$$

Which visually translates to Figure 4.30.

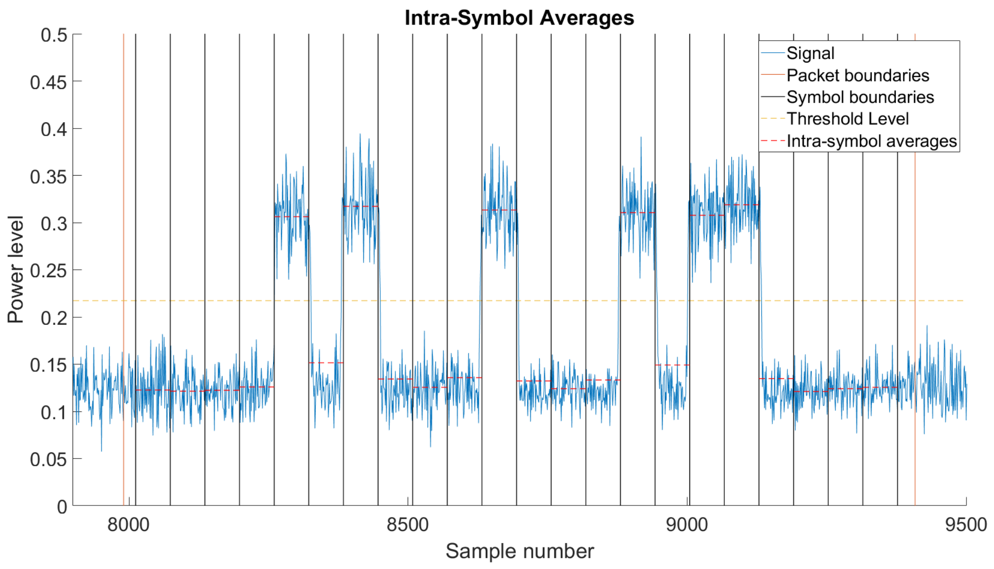


Figure 4.30: Average value of all the k sized subgroups.

After this step, a binary bitstream (C) can be obtained via application of Equation 4.14.

$$C(i) = \begin{cases} 0, & \text{if } S_{AV}[i] \leq T \\ 1, & \text{if } S_{AV}[i] > T \end{cases} \tag{4.14}$$

Resulting in what we can see in Figure 4.31.

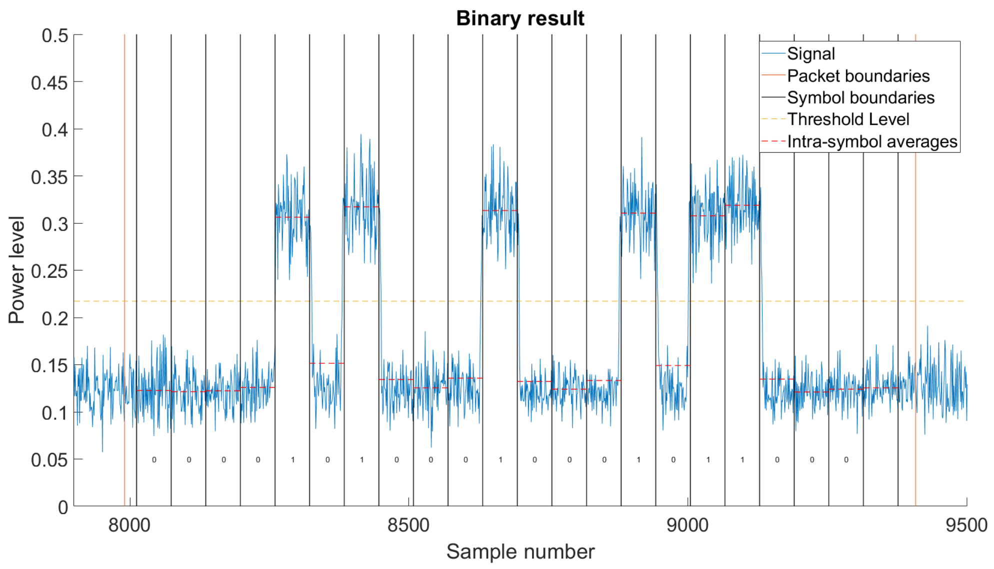


Figure 4.31: Result of the horizontal threshold comparation process.

The result of all the applied methodology so far is an array of binary values visible at the bottom part of Figure 4.31. The same process will be applied to all the received packets (but not to the periods of silence in-between, for optimization reasons). The resulting bit array in this specific packet, and in a majority of cases includes two blocks of zeroes at the ending that must be trimmed in order to retrieve the payload. This is done in the next step, the parsing, detailed in section 4.7. However, before that subject is approached, it's important to elaborate on a few alternative methods of signal processing regarding comparison, they will be explained in the following subsections.

4.6.3 Pass By Zero

Pass by Zero is a processing method that attempts to use the timings at which the signal equals a constant pre-set value to extract clock information.

This method outshines its counterpart, the TMC algorithm, in its simplicity. Whilst the previously described comparator was built with antenna range maximization in mind, for this approach a more CPU efficient route was adopted.

General diagram

In Figure 4.32 we can see a diagram describing the general process flow of this algorithm.

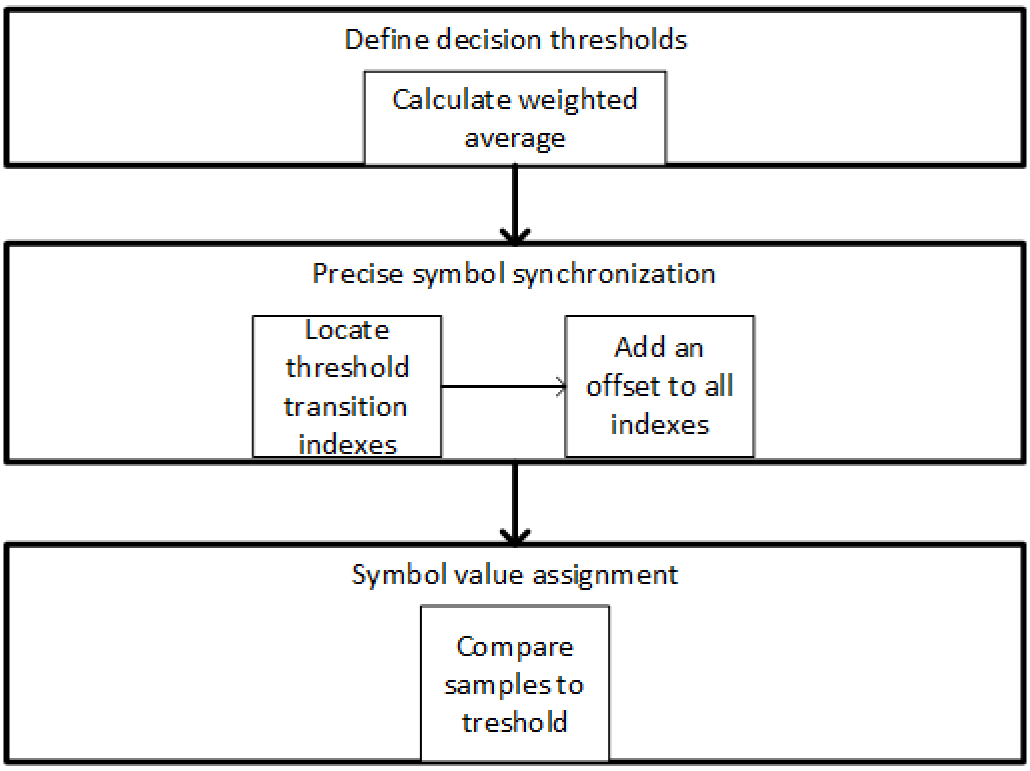

Define decision thresholds
Calculate weighted average
Precise symbol synchronization
Locate threshold transition indexes
Add an offset to all indexes
Symbol value assignment
Compare samples to treshold

Figure 4.32: Pass By Zero process.

Detailed explanation

Defining decision thresholds In order to favour a faster execution time time, this comparator defines the threshold (T) simply by measuring the global maximums(G_{max}) and minimums (G_{min}) of all quantized values in each frame. It then applies the following formula:

$$T = (G_{max} - G_{min}) * R + G_{min} \tag{4.15}$$

Where R is a configurable ratio, its value in figure 4.33 was 0.35.

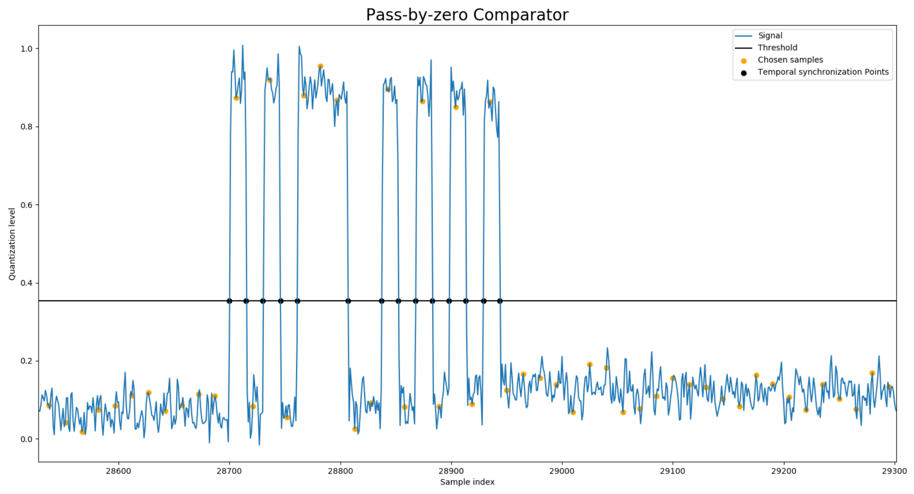


Figure 4.33: Pass By Zero comparative process applied to a slice of internally generated signal

Timing synchronization Not all symbol boundaries include a sharp quantization increase/decrease. For example, in scenarios with several consecutive identical symbols in a row quantization levels would remain similar. However, all deliberate sharp transitions will occur in very close proximity to a symbol boundary. If we piggyback on these sharp transition timings to find one boundary, we can use this knowledge to extrapolate timings for the remaining ones. In figure 4.33 we can see a visual representation of this process.

Despite the fact that signal is received from the RF frontend in frames and not continuously, it is still interpreted from left to right. Because the horizontal black line represents the threshold separating different symbol amplitudes, we can assume that in normal operating conditions every time the blue signal line crosses the horizontal black line, we are in relative close proximity to a real-world symbol transition. By estimating the first synchronization point ($S_{p_{[0]}}$) as a close match to a real symbol boundary and moving to the right we can start re-sampling the signal at relevant points, ideally chosen mid-symbol. This is done according to Equation 4.16.

$$k = S_{p_{[n]}} + \frac{S_R}{2 * S_r} + \frac{S_R}{S_r} * a : k < S_{p_{[n+1]}} \wedge a \in \mathbb{N} \tag{4.16}$$

This will give us the index offsets for the re-sampling process, which when applied to the signal function return the desired samples. Unlike the previous algorithm, PBZ uses single samples and not group averages as a basis for the decision process described in the next step.

Symbol value assignment After re-sampling is done, the extracted sample values are compared against the threshold. Being then attributed a value of either '0' or '1' according to equation 4.14.

Limitations In real-world applications, the added external factor of slower (i.e. non-instantaneous) transition times may introduce issues. The interval of time between start of transition and threshold-signal intersection will cause a deviation between real-world and estimated symbol boundary timing. This delay becomes a considerable source of error as the ratio between transition time and signal period gets closer to 1. A visual representation of this flaw can be seen in figure 4.34.

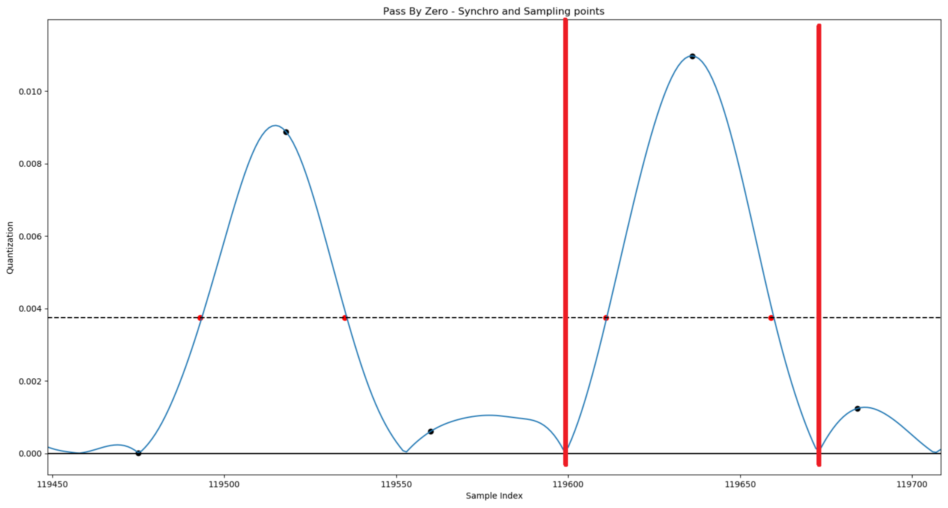


Figure 4.34: Synchronization issue arising from non-instantaneous transitions, partly caused by the digital filter. The red line represents the symbol boundary, the ideal point at which the comparator should synchronize with the signal. The red points represent the actual synchronization moments.

Other synchronization issues arise in specific oncoming packet configurations. This approach relies on semi-frequent symbol changes in order to re-synchronize the clock. For cases when the transmitter attempts to send a configuration of several identical symbols in a row, timing errors might accumulate during the relatively larger period between synchronizations leading to errors such as a miscount in the number of received symbols. This can be counteracted by changes in the communications protocol, padding the packets with periodical forced transitions, for example with a rule such as every sixth bit always being the negation of its previous one. Another example could be switching to a Manchester or Return to Zero approach.

Another flaw in this comparator, one that eventually propelled the development of SPBZ is the fact that sharp noise bursts might force the signal across the threshold, thereby forcing synchronization at the wrong moment, effectively de-synchronizing. This is pictured on Figure 4.35

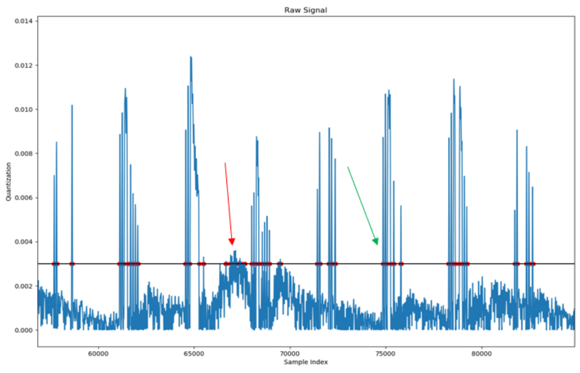


Figure 4.35: Not all threshold transitions are intentional (green), some are product of noise (red).

On subsection 4.6.4, a workaround attempt to this last limitation is detailed.

4.6.4 SPBZ - Smart Pass By Zero

The Smart Pass by Zero algorithm began development as an exploration of a solution to the problem depicted on 4.35. As exposed earlier on section 4.6.3. Having an algorithm that relies on transitions over a certain threshold to synchronize can leave us exposed to ambient noise bursts. How does it differentiate between intended and noise-based threshold transitions? Attempting to answer this question was ultimately what branched this comparator from PBZ.

A solution was eventually built and it did achieve its purpose, but not with good reliability and efficiency. Because of this, it was later abandoned. Reading this subsection is, therefore, not required in order to get a full understanding of the current state of the program. It exists merely to document already explored avenues of development. And because it builds upon the previous one, its structure is somewhat similar. Usual aspects like defining decision thresholds, temporal synchronization methods and symbol value attributions won't be detailed here and should be assumed as identical to subsection subsection 4.6.3. Let's begin then at the so-far unsolved flaw described on the aforementioned section and depicted on 4.35. Three main methods at filtering out the undesired transitions were adopted, they relied on the following premises:

1. **Synchronization issues arising from slow transition times must first be minimized**

   As we can see in figure 4.34 the PBZ comparator was using intersections of the signal with a pre-defined threshold as a guidance for synchronization. However because due to real-world limitations, transitions are non-instantaneous, an error was introduced which could jeopardize future developments in this area. In order to improve upon this method, the error source must be solved/minimized.
2. **Packets must arrive at a consistent rhythm**

   By assuring the time interval between transmitted packets remains constant, over time a consistent pattern of packets and periods of silence can be observed on the transmission line. We are able to used that pattern, on the receiving end, to learn how to predict during which windows of time transitions should be expected and considered, and during which we they should be ignored because they are certainly unintended.
3. **The symbol rate is constant**

   Because the signal period does not change, and because transitions are located in the borders between two symbols, it can be inferred that deliberate transitions will be approximately located in intervals of k samples (equation 4.6) or multiples of that number, for cases with consecutive identical symbols. This allows us to designate hotspots inside which the probability of a transition being intended increases, and conversely it allows us to tag as unintended any transitions which deviate from this pattern.

Now that three objectives have been established, alongside their reasoning, we can proceed to approach them in order.

To guarantee premise 1, some shifting of the synchronization points was necessary depicted in Figure 4.33. Until now, synchronizations would happen too late in rising edge cases and too early in falling edges. In order to rectify this issue the following approach was taken:

1. Calculate the signal's integral.
2. For each transition:
   - Designate as point A the current value for the transition.
   - Designate as point B the closest relevant point at which the integral equals zero. Use the closest previous point for rising edges and the closest following point for falling edges.
3. Designate as the new transition value the geometrical middle of points A and B.

In figures 4.36 and 4.37 we can see an excerpt the signal, encompassing a few symbols, depicting this shifting:

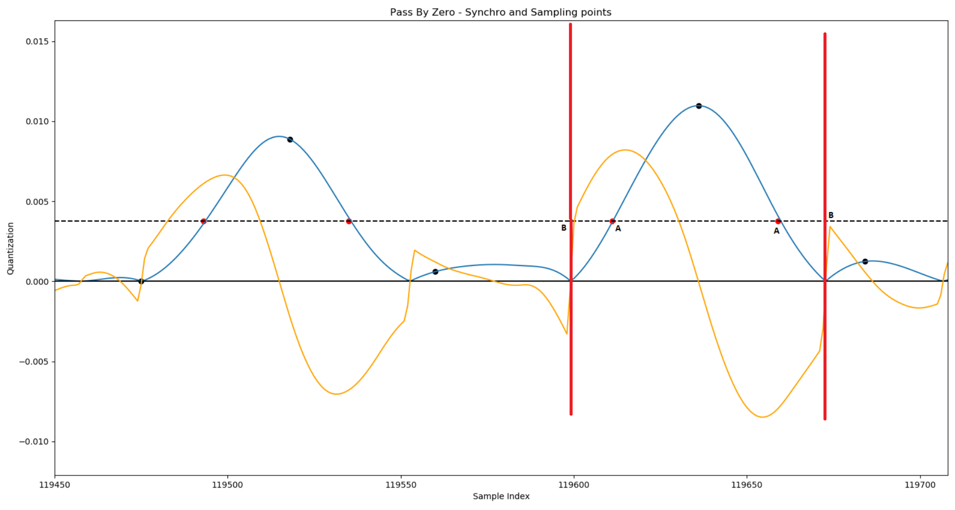


Figure 4.36: Points of interest before synchronization shifting. The yellow function is the blue signal's integral. The vertical red lines are located at points where the integral equals zero. The black dotted line represents the threshold described in the PBZ filter subsection. The red dots represent points at which the PBZ comparator synchronized with the signal. The blue dots represent the sampling points.

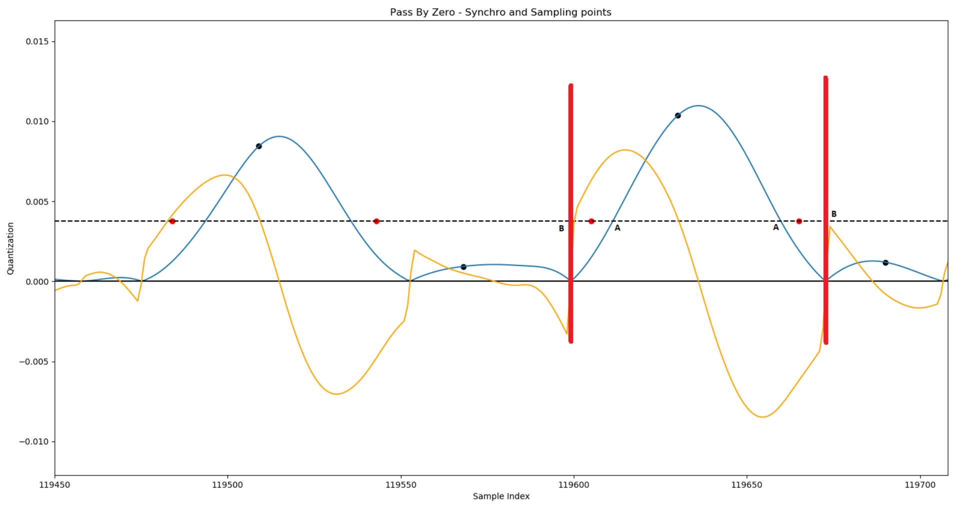


Figure 4.37: Points of interest after synchronization shifting. The yellow function is the blue signal's integral. The vertical red lines are located at points where the integral equals zero. The black dotted line represents the threshold described in the PBZ filter subsection. The red dots represent points where the PBZS comparator will synchronize with the signal. The blue dots represent the sampling points.

By applying this method to all the signal-threshold transitions, and therefore shifting all synchronization points closer to their real values we have made possible the improvements built upon this premise. Some error will still persist, as the system is fundamentally asynchronous, but now at manageable levels.

In order to apply premise 2. We must attempt to discern the pattern caused by intermittent packet transmission. In order to do this two steps are taken.

Firstly an arbitrary transition is selected, with its position in the horizontal axis deemed as the origin. All transitions located within a certain neighbourhood of the origin are have their respective relative positions stored (this being difference between their actual horizontal axis position and the position of the current origin). This value was named as the 'deltas' of a transition. The process is then repeated but with a different transition as the origin until every single one has been selected. All the relative positions can then be grouped into bins and displayed as an histogram. In figure 4.38 we can visualize a representation of this process.

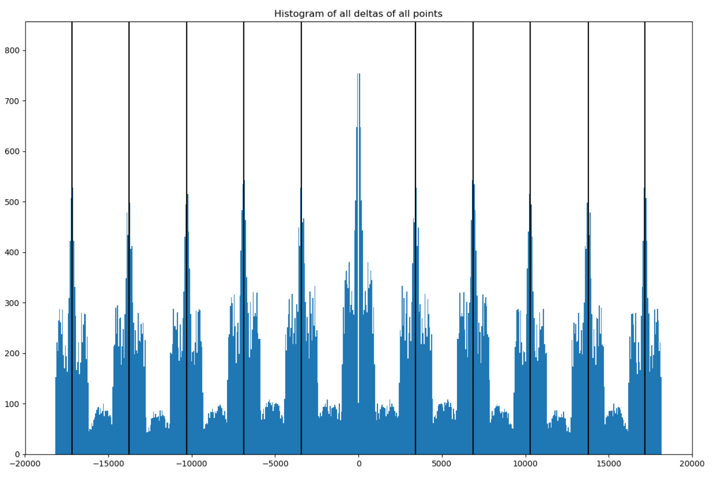


Figure 4.38: Histogram depicting the frequency of intervals between packets. Vertical black lines represent identical parts of previous/following packets.

Having found the rhythm at which transitions occur, it can then be used to differentiate intended from unintended ones.

In figure 4.39 we can see the comparison between Figure 4.38 and two other plots, one of them containing the deltas of a previously selected intended transition, and another from an unintended one.

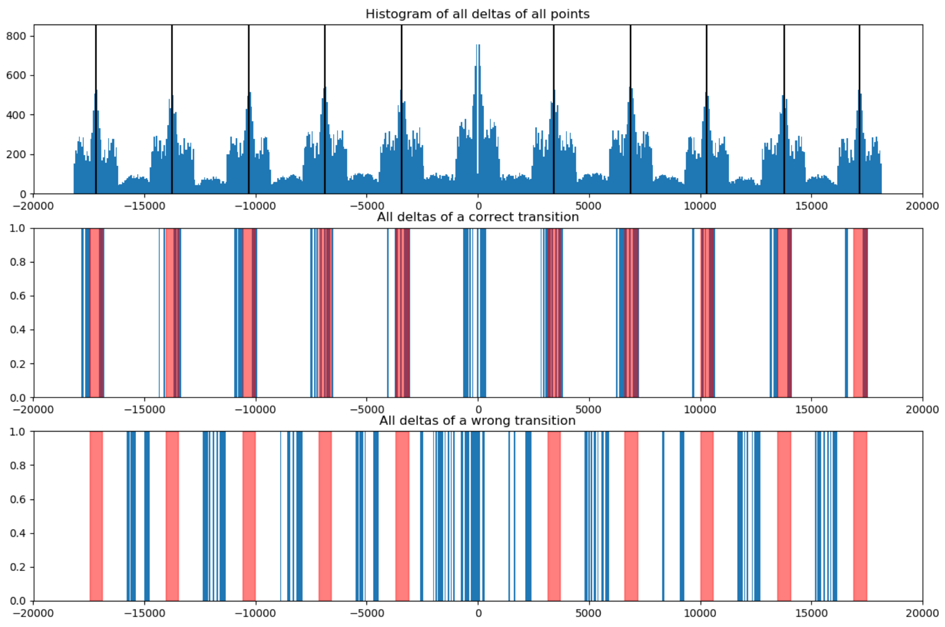


Figure 4.39: Comparison between figure 4.38, deltas of an intended transition and a noise-caused one.

The highlighted red areas represent areas of high expected delta density for cases when current origin is an intended transition. Another way to describe would be that if for any given intended transition one can look forward or backwards temporally in the signal and expect to find other intended transitions at constant intervals of time. However, for cases when an unintended transition is the current origin, the pattern observed will not match the initial histogram of patterns between packets.

This allows us to filter out the transitions that do not match the pattern, if for any given transition we attribute a score system based on the amount of deltas within the red zone we will obtain the result seen in figure 4.40.

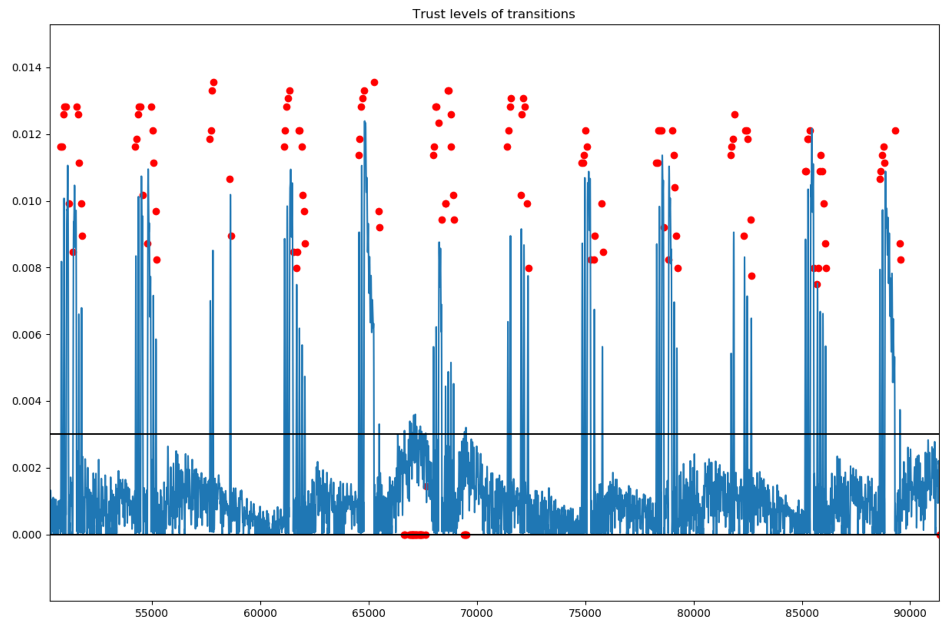


Figure 4.40: Comparison of relative scores in-between threshold transitions. Each red dot represents one transition, its vertical positioning being proportional to the likelihood it is an intended one.

All that's left now is to apply a score threshold and discard all the transitions who fail it. Thus concluding implementation of premise 2.

It is now possible to establish 'transmission' and 'silence' areas in the signal (without resorting to CPU intensive methods used before, like variance monitoring), this means that all transitions occurring during a 'silence' are assumed to be caused by undesirable factors, such as noise, and won't be taken into account at the later synchronization stage.

But part of the problem still persists, what about the unintended transitions happening during a 'transmission' area? This problem is tackled with the implementation of premise 3. Because the number of samples in each symbol (also designed as k in equation 4.6) is constant. By knowing a single transition, designated as the origin, we can predict an interval for the location of each of the remaining ones within the same packet. And if that prediction matches reality, in theory we are likely looking at an intended transition. On the other hand if we make predictions based on an origin and no actual transitions match them, that origin was likely an noise-caused transition. This is a variation of the previous method but only applied at an individual packet level instead of in-between packets. Figure 4.41 describes this process.

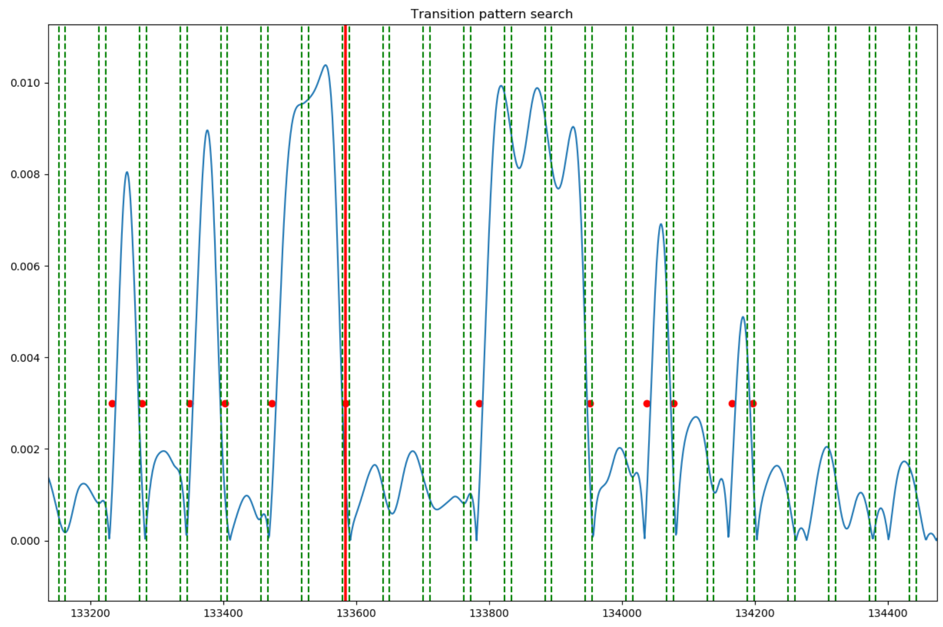


Figure 4.41: Implementation of premise 3. Red line is this example's origin. Green lines represent intervals centred in multiples of k distance from the origin. By accepting the transition marked with the red line as the origin we should be finding other transitions inside the intervals delimited by the vertical green lines. An accumulation of two different sources of error makes it not always be the case.

Limitations Here a major problem arises. Because there is an error between the real value of k and its integer estimate used by the program. And because that error accumulates as we get further from the origin, it eventually becomes impossible to sync-up with other intentional transitions. It is not productive either to make the intervals wider, as that would weaken the method substantially by also potentially incorrectly identifying unintentional transitions as correct. This is amplified by the fact that, as described earlier and depicted on figures 4.36 and 4.37, synchronization points are an approximation of their real value.

Two signal areas have been established before in Figure 4.21, 'transmission' and 'silence'. Intended transitions will by definition only occur during predictable intervals, always on a 'transmission'. Unintended transmissions however, can happen during both occasions. If we split them according to this characteristic we can also verify that this comparator has a different degree of success on identifying a noise-based transition based on when it happens. Unintended transitions occurring during a silence period are identified more easily, but those are also the ones that represent the smallest impact on the program's ability to preform correctly. Even if they cause an out-of-phase synchronization during the silence period, it would eventually be re-synced at the first intentional transition of the next packet, making all the effort of identifying them unnecessary.

On the other hand, for those that occur during a packet, on a 'transmission' area, have a dire impact on the comparators ability to correctly define boundaries between symbols and thus re-sample at the appropriate times. Identifying these unwanted transitions proves a much more demanding task for the developed algorithm.

Furthermore, the selected approach relied on factors which would impose significant restrictions at a later point. For example, if timing between packets was assumed to always be identical, it would be much harder to implement multiple transmitter compatibility later.

Overcoming these new obstacles would require adding more lines of code. However, by this point, the comparator was already tremendously CPU heavy. The calculations required at different steps, such as integration, synchronization point shifting and signal analysis with infinitesimal step sizes were taking a tremendous toll and bringing execution times up to at least one order of magnitude higher than its other counterparts.

A context is created where the PBZS was not very good at executing its task. And was particularly ineffective at the most crucial details. In addition, its execution was consuming prohibitive amounts of processing resources which made it not worth the implementation even if any other problems were disregarded. This was the breaking point that instigated development into new directions. Instead of investing more time into the fine-tuning of this comparator, it was opted to mitigate the amount of noise in the signal, attempting to prevent the issue of noise-caused transitions in the first place.

Development was shifted away from this avenue before implementation was fully finalized, and as such, this comparator is not mentioned in any other area of this thesis.

4.6.5 FAST

In Figure 4.42 we can see a diagram describing the general process flow of this algorithm.

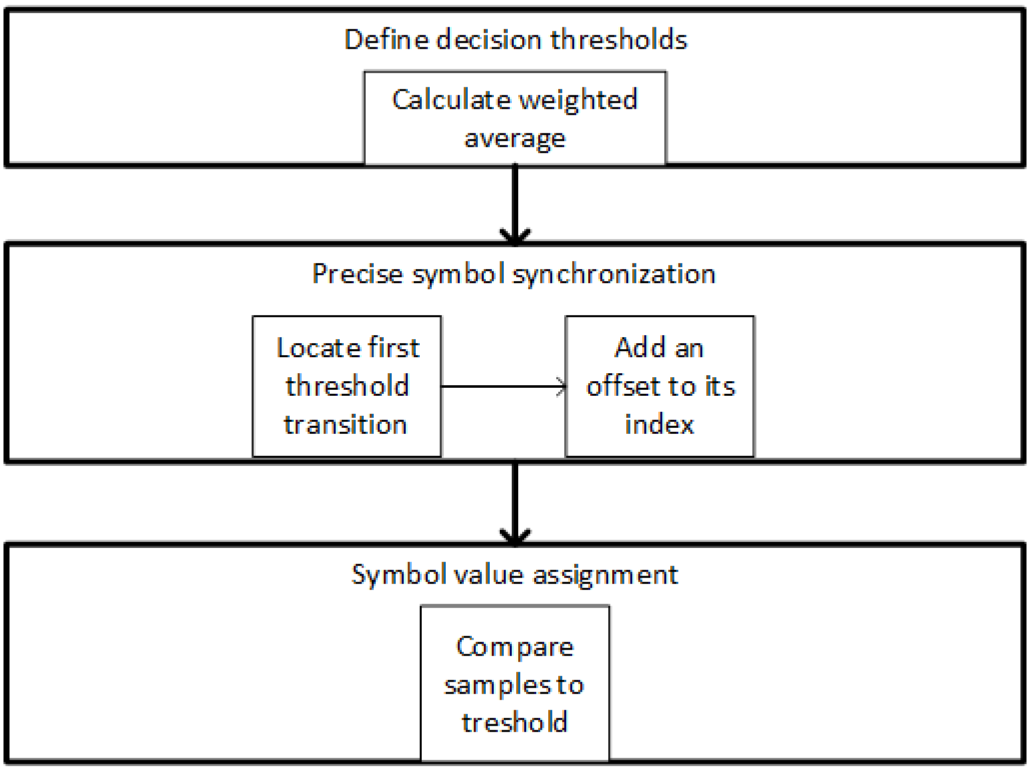

Define decision thresholds
Calculate weighted average
Precise symbol synchronization
Locate first threshold transition
Add an offset to its index
Symbol value assignment
Compare samples to treshold

Figure 4.42: FAST process.

Detailed explanation

This fourth comparative method is similar to PBZ, with its main difference being the incorporation information parsing methods to accelerate execution speed even further.

FAST was designed because PBZ's execution speed was not sufficiently fast. Under some circumstances data was still being harvested faster than it could be processed, forcing a signal harvesting 'stand by' state that impacted data throughput levels as described in subsection 4.2.3. This motivated the creation of FAST.

As we can see in Figure 4.43, this comparator is similar to PBZ, its major difference being fewer timing synchronization points.

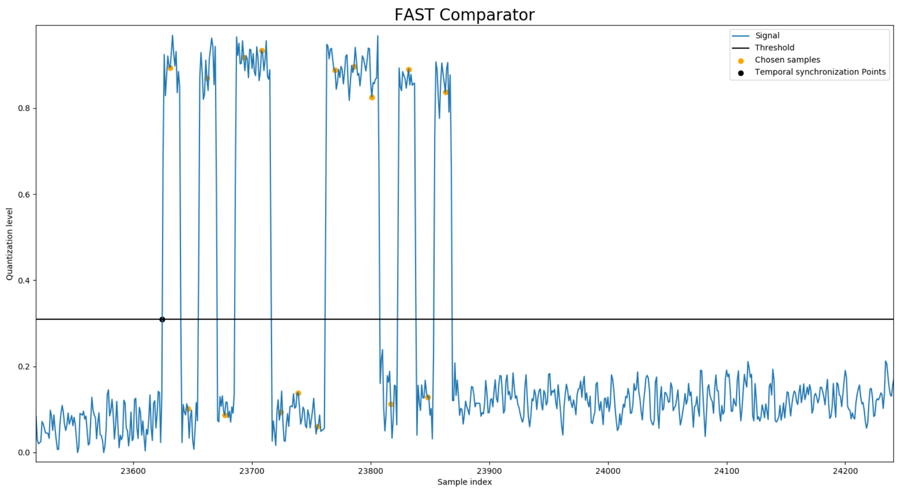


Figure 4.43: FAST comparative process applied to a slice of internally generated signal

Defining decision thresholds The threshold definition process is identical to the one described in the PBZ comparator sub-section

Temporal synchronization As mentioned before, a similar methodology to PBZ is implemented here, albeit with sparser synchronization points. In this algorithm, they only occur at the beginning of each packet. Such beginnings are identified by a rising transition after a long period of permanence in the lower level. This means that the time interval between packets must always be equal or greater to a known, pre-set, value. After packet start identification occurs, samples will start being taken at regular intervals according to Equation 4.17.

$$k = S_{p_{[n]}} + \frac{S_R}{2 * S_r} + \frac{S_R}{S_r} * a : 0 < a < P_s \tag{4.17}$$

Where P_s is expected packet size. This means that advantage is being taken from the fact that packet sizes are known, constant, and relatively small. As an added feature, advantage is taken from the fact that received packets begin with a preamble that is both constant and known. For a size N preamble sampling will be stopped in any instances where the first N samples of any packet do not match the preamble, being resumed again at the beginning of next packet. This has an impact on the output of this comparator. Where others will produce a bitstream with large amounts of consecutive zeroes (silence) separating the occasional sequence of 1's and 0's (transmission), in this comparator, because of the re-sampling method, the silences will not be present in the output. This happens as a side-effect of speed optimization and is inconsequential.

Symbol value attribution The symbol value attribution process is identical to its equivalent described in the PBZ comparator sub-section

Limitations Using this comparator represents a major boon in terms of speed in contrast to the previous two. As a trade-off, several limitations are placed upon the possible adopted communications protocol. Due to the once-per-packet time synchronization points, it is expected for temporal adjustment to fall off if packet size is increased, imposing a theoretical upper packet size limit. Another disadvantage is that because of the method used to differentiate the beginning of packets from regular transitions, a silence period must be maintained between packet transmissions, with a minimum recommended duration of $Symbol\,period * (Packet\ size + 1)$. Any sufficient noise forcing a transition during that period will cause the algorithm not to correctly identify the preamble later. And lastly, for setups where the signal level for silence periods also equals one of your symbols (in this case silence equals 0), it becomes infeasible to begin any of your packets with that same symbol.

4.7 Data parsing

We have now, after filtering, decimating and data recovering, abstracted the signal into a stream of bits. In this step we will attempt to retrieve any relevant information and validate it. Let's first make a structural introduction on the communications protocol currently adopted.

4.7.1 Custom Packet Protocol

ID.all exists in order to provide means to receive the data encapsulated within a signal backscattered by a tag. It is then captured by the readers antenna before being filtered, compared and eventually parsed. Parsing is only possible because a protocol has been previously established and the reader knows how the incoming information is structured. Measures were taken in order to facilitate the customization of this protocol by the end user, and as such, adaptation into other protocols such as Zigbee[33], is easy to implement. The standard option is a custom protocol specifically structured with this project in mind. This structure depicted in figure 4.44 and consists of (in order):

- **Preamble**

  With the primary purpose of providing frame-sync, it's size and content can be customized by the user. The default preamble is [1,0,1,0].
- **Payload**

  Data storing field. Customizable size with a default of 8.

- **Cyclic Redundancy Check (CRC)**

  Used in packet validation. Detailed explanation in subsection 4.7.3, default divisor is [1,0,1,0].

- **Stop bits**

  Signals the end of frame. Optional, given the (currently) constant packet size, default value is [1,0].

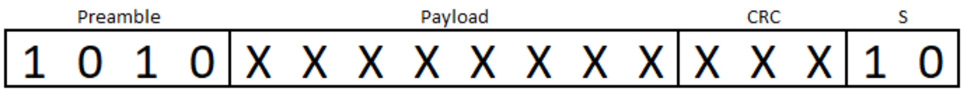


Figure 4.44: Adopted signal protocol structure.

All of these packet fields, along with other signal properties such as symbol rate, can be edited by the user. For information on how to achieve this, please refer to Appendix B.

4.7.2 Identifying packets

In regular working conditions, the output of the previous stage will be a bitstream such as the one pictured in Figure 4.45. It will be preceded and succeeded by unnecessary irrelevant bits, and the first step of the parser will be to discard them.

. . . 0 0 0 1 0 1 0 1 1 0 0 0 1 1 1 0 1 0 1 0 0 0 0 . . .

Figure 4.45: Bit sequence resulting from the output of the previous step.

In order to do so, the program will iterate sequentially through all bits in the list and attempt to find a sequence that matches the preamble, in our example this means [1 0 1 0]. After doing so it'll also know where to find all the other fields from their relative position to the preamble itself. This means that now all the overhead bits can be trimmed away.

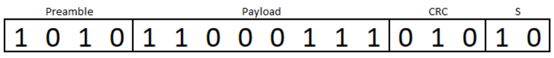


Figure 4.46: Packet after removal of unnecessary data.

Having now an array of bits, demodulated from the signal wave, the focus shifts into how to interpret the sequence and extract desirable information. As we also know the protocol's structure, it becomes trivial to find the remaining fields. The bit array is then sliced on the relevant index offsets. This also prevents the accidental flagging of preambles inadvertently inside payloads, as the parser analyses the bit list sequentially and will correctly disregard preambles for areas already identified as non-preambles, such as the payload or CRC field.

After tagging each packet field, they are then passed on to the next stage, the validation.

4.7.3 Error detection / Packet validation

Noise is always a given when working with communication systems, it's important to be sure the information received matches what was transmitted. A CRC is a code that aims at identifying the presence of errors in a given transmission, it makes no attempt at recovering information when discrepancies are found, typically in those cases the flagged information is simply tagged as unreliable and/or discarded. Because this reader is aimed at interfacing with sensors that can transmit the same message several times in a second, recovering from errors is not very rewarding. As it becomes more efficient to simply detect them and discard affected packets. CRC becomes a good choice here, as it is merely used for detection and is also easy to implement, while not being very resource intensive on either side of transmission, nor having a large message overhead.

It consists of a polynomial division between two blocks of data, carried out by series of alternate xor's and right shifts. Before transmission, the blocks of data are divided by previously agreed upon value, the divisor, and the remainder of this division is included with the data packet being sent, as a CRC field. After transmission, the same operation is repeated on the receiving end, and because the divisor is constant and known to both sides, the remainders will match in circumstances where no errors happened during transmission. This event is called a validation. For cases where noise has corrupted one or more bits during transmission, the results of these operations will almost certainly not be identical, making the packet fail validation.

The division process is illustrated on Figure 4.47.

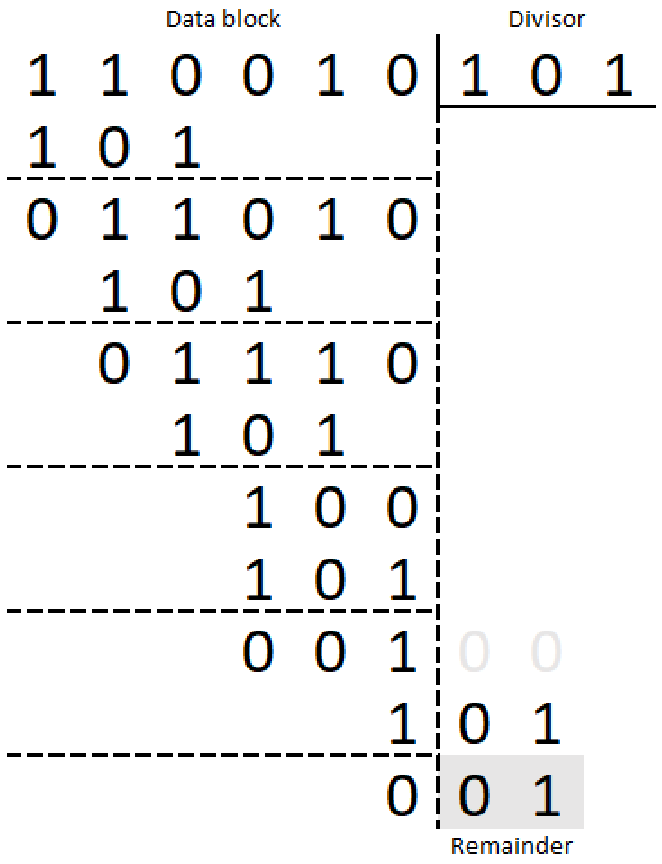


Figure 4.47: Cyclic Redundancy Check

According to [34, p.494],the probability of a false positive happening in this process is seen in Equation 4.18.

$$P_{FP}(\epsilon) \leq 2^{-(n-k)}[1-(1-\epsilon)^n] \tag{4.18}$$

Where ϵ is the probability of data recovery error in a symbol, k represents the payload size, and n represents $k + CRCfieldsize$. This sets an upper bound at a level we can define in Equation 4.19

$$P_{FP} \leq 2^{-(n-k)} \tag{4.19}$$

4.7.4 Validated information storage

After each successful validation the respective payload is added into a list. Each list corresponds to the validated payloads of a given frame. When the program is done parsing a frame it will save its respective list in a `.npy` [28] file within the sub-folder `./outfiles/`. The creation time-stamp for this file will also be used as its name. No more than 5 files will ever be in this sub-folder at any given time, the program will delete oldest file before creating a new one if the file number exceeds 4. This is done to prevent memory bloating.

After storage, the contents can be freely accessed by another agent with network access and read permissions, enabling network integration.

Having concluded the demodulation process, we have also contextualized the methods behind the results presented in the next chapter.

Chapter 5

Hardware Setup and Results

In this chapter, a list of hardware used throughout the testing phase will be introduced, followed by a diagram with pictures on how to assemble it. In the second half, tables and graphs of testing results will be presented, before a conclusion with results interpretation.

5.1 Hardware Setup

The following sections detail a list of required materials and final design for the setups used to measure the values presented on section 5.3.

5.1.1 Hardware list

The following sub-sub-sections detail all the hardware used during tests.

RTL-SDR USB dongle

This dongle provides the interfaces between antenna and Raspberry PI, acting as an RF frontend. Incorporates an R820T2 tuner chip and the RTL2832U Coded Orthogonal Frequency Division Multiplexing (COFDM) demodulator. Forces restrictions upon the setup, tuning and sampling wise. Tuning frequency is restricted to the 24 - 1766 MHz interval, whilst sampling frequency must be in the 0.226 - 3.2 MHz range.

RTL-SDR.COM
QUICKSTART SETUP GUIDE: RTL-SDR.COM/QSG
DVB-T+DAB+FM+SDR
RTL2832U R820T2 TCXO+BIAS T+HF
V3

Figure 5.1: An example of an SDR USB dongle, image retrieved from amazon.com.

868 MHz rectangular patch antenna

Three rectangular patch antennas, designed for usage in the 868MHz spectrum zone were used, one of them can be see in Figure 5.2.

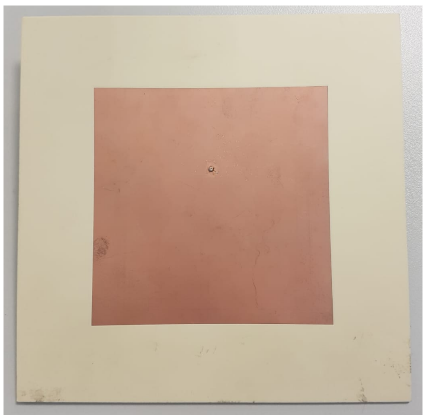


Figure 5.2: 868 MHz rectangular patch antenna

Raspberry PI

A Raspberry PI Kit, as seen in Figure 5.3 was used as the central hosting platform, included in the kit are a Raspberry Pi 3 Model B+, a case, a micro USB power supply and a microSD memory card. The chosen operating system was Raspbian 9.

Figure 5.3: Raspberry Pi 3 model B+ (image from [15]).

Semi-Passive backscattering sensor

The tag [9] used during the testing phase. Restrictions included a maximum symbol rate of 3600 Hz on the transmission and non-instantaneous transition times. In Figure 5.4 and Figure 5.5 we can see both of its sides.

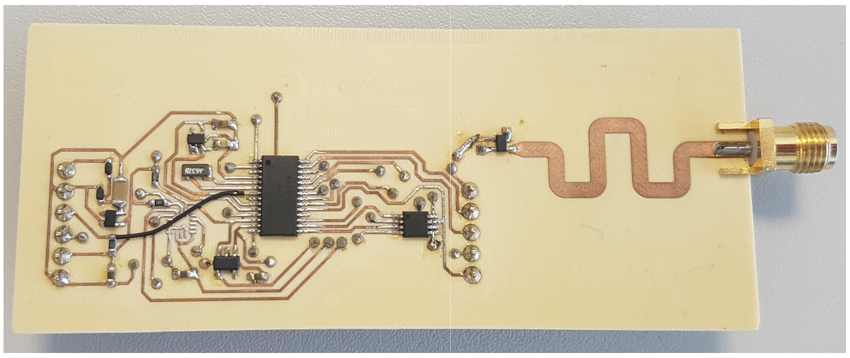


Figure 5.4: Adopted sensor, front.

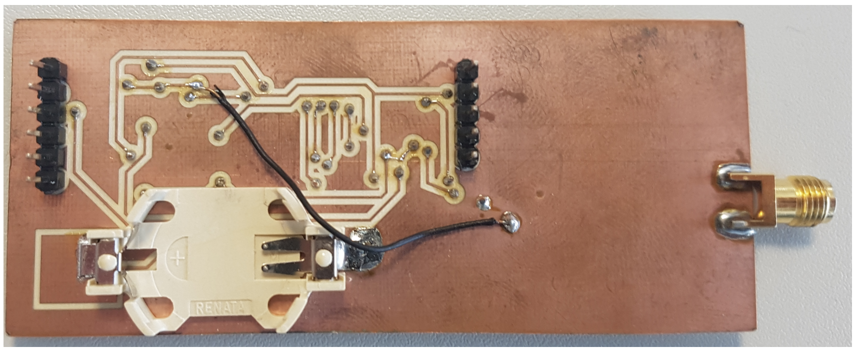
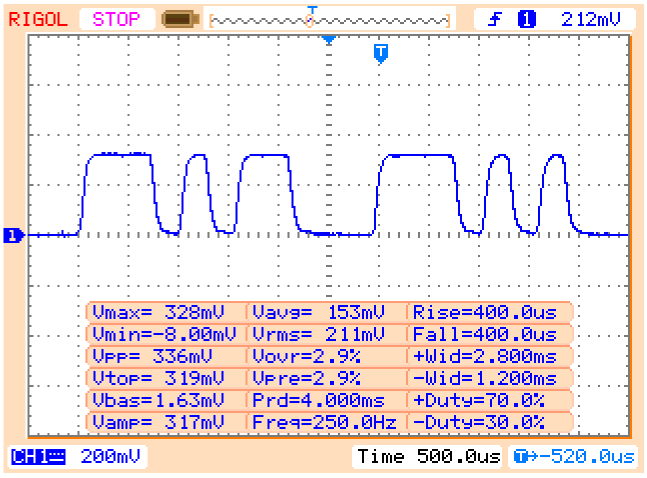


Figure 5.5: Adopted sensor, back.

An example of its backscattering commutation pattern can be seen in Figure 5.6.


```
RIGOL  STOP                                   1  212mV

Vmax= 328mV    Vavg= 153mV    Rise=400.0us
Vmin=-8.00mV   Vrms= 211mV    Fall=400.0us
Vpp= 336mV     Vovr=2.9%      +Wid=2.800ms
Vtop= 319mV    Vpre=2.9%      -Wid=1.200ms
Vbas=1.63mV    Prd=4.000ms    +Duty=70.0%
Vamp= 317mV    Freq=250.0Hz   -Duty=30.0%

CH1  200mV                    Time 500.0us  T→-520.0us
```

Figure 5.6: One packet, measured directly from the sensor with the help of a digital oscilloscope. The communications protocol is explained in detail in subsection 4.7.1.

Keyboard, USB mouse and screen

These items are required in order to interact with the Raspberry PI, a Wi-Fi based Secure Shell (SSH) connection can be used as an alternative.

Rohde & Schwarz SMJ100A VSG

The signal generator show in Figure 5.7 was used to provide a carrier wave to the tag, which was backscattered into the reader.

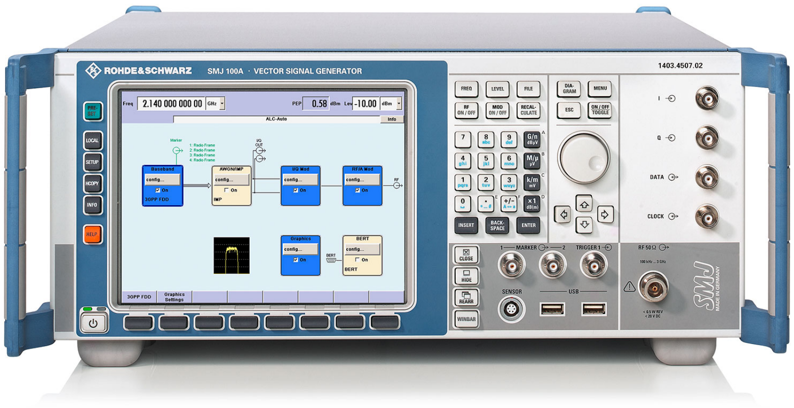


Figure 5.7: SMJ100A VSG (image from [35]).

Spectrum analyser (Optional)

Used for debugging purposes.

SMA cables

SubMiniature version A (SMA) cables, such as the ones pictured in Figure 5.8 are required to connect between the antennas and VSG/tag/RPI.

Figure 5.8: SMA cables (image from [36]).

5.2 Assembled setups

Two different setups were used during the testing phase. Setup 1 is intended as a "vacuum" testing platform, and a support measure to debug by potential issues with setup 2 via isolation. Because setup 2 also includes a tag developed by another party, every time a problem occurred, testing would be switched to setup 1 as an attempt to troubleshoot and identify if the source of the problem resided reader side.

Setup is meant as a close representation of final implementation of the system. It attempts to simulate practical, real-world, applications of this technology.

5.2.1 Setup 1

On Figure 5.9 we can see a schematic for setup 1. On Figure 5.10 we can see the entire setup from the reader's and transmitter's perspective. In Figure 5.11 and Figure 5.12 we have frontal views of the reader and transmitter, respectively.

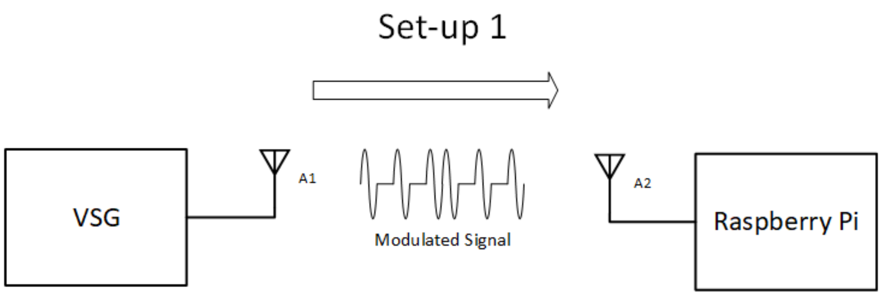


Figure 5.9: Schematic for setup 1

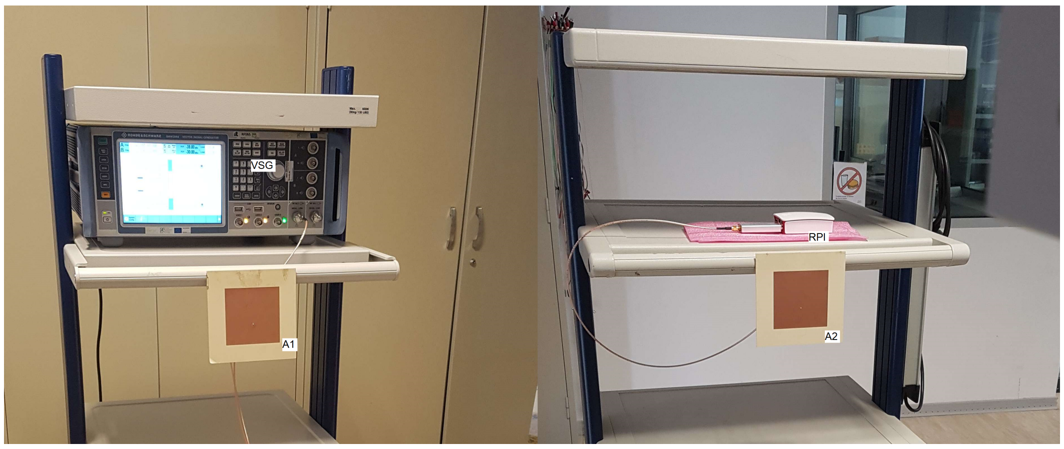


Figure 5.10: Setup 1

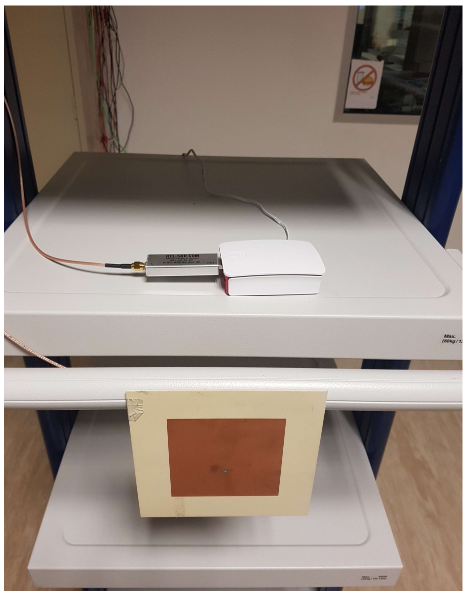


Figure 5.11: Setup 1

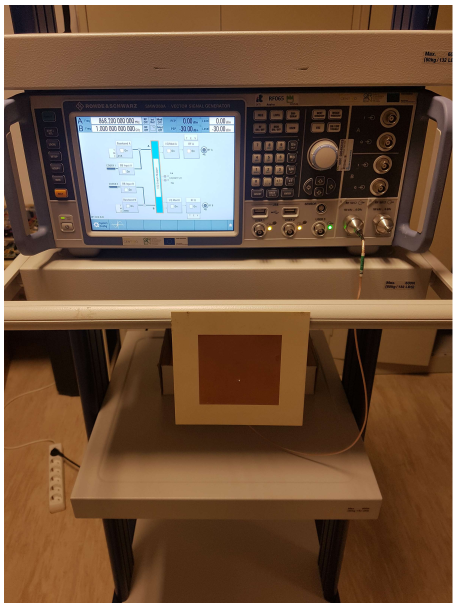


Figure 5.12: Setup 1

5.2.2 Setup 2

In Figure 5.13 we can see a schematic for setup 2. In Figure 5.14 we can see the entire setup from the reader's and VSG's perspective. In Figure 5.11 and Figure 5.12 we have frontal views of the tag and generator/reader, respectively.

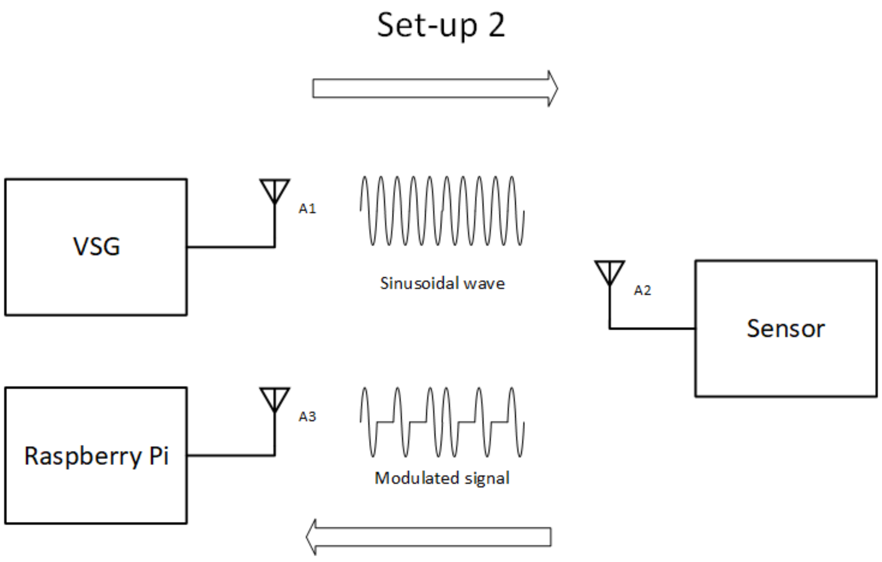


Figure 5.13: Schematic for setup 2

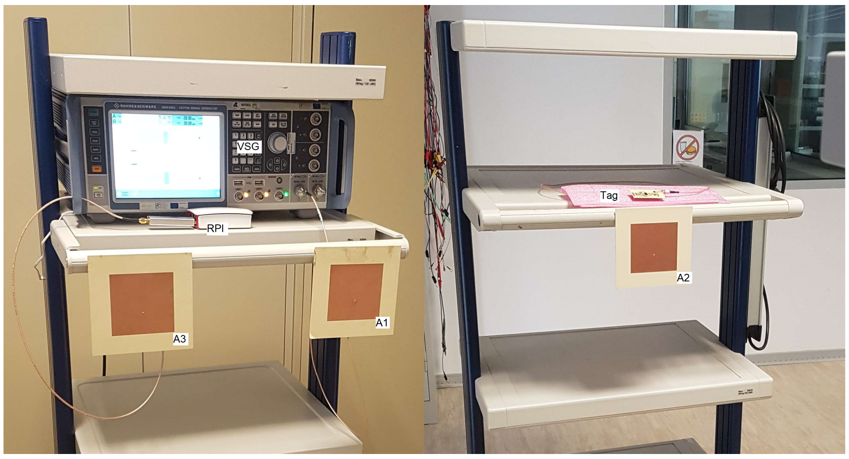


Figure 5.14: Setup 2

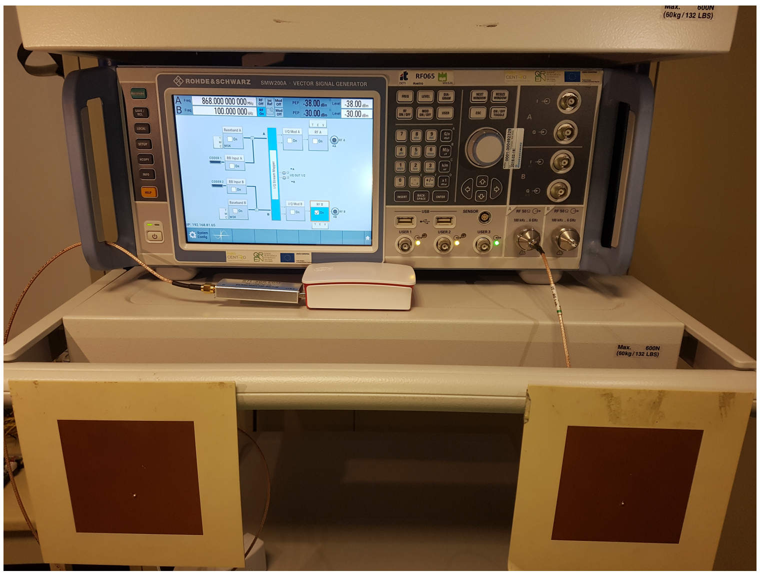


Figure 5.15: Setup 2 - Side 1

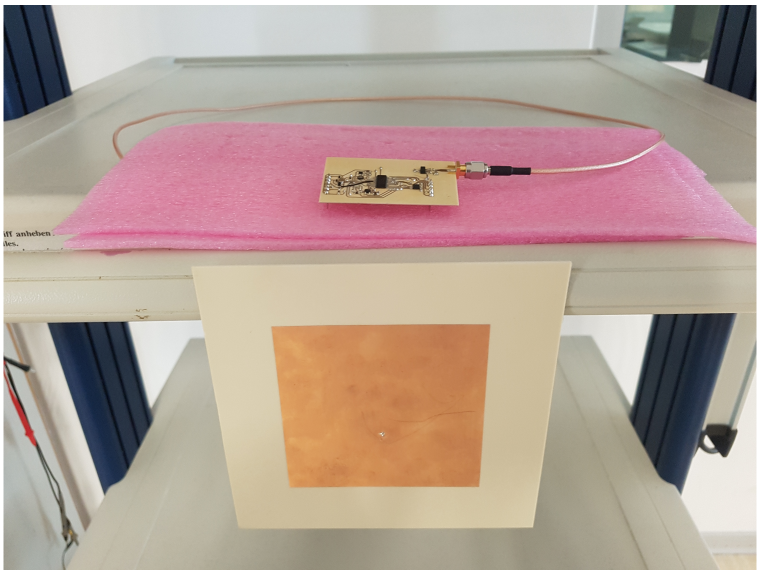


Figure 5.16: Setup 2 - Side 2

5.3 Results

Let's now first contextualize and then present the results.

5.3.1 Packet loss

Before taking a dive into the raw data provided by the experimental testing phase, it is important to contextualize the different limitations of the setup and how they can influence the testing results.

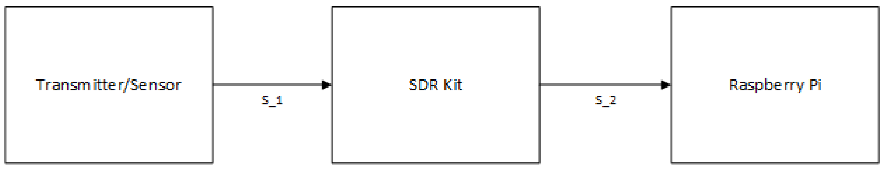


Figure 5.17: SDR-PI data pipeline

Let's begin with the SDR USB dongle and it's interfacing method with Python. The software library[22] used to integrate this piece of hardware into the project provides a burst-mode method of communication in order to transmit the sampling information collected by the dongle, each burst being denominated as a frame. It is important to keep in mind that there is no guarantee that last sample of a frame was collected immediately before the first sample of the following frame. Depending on the setup, an unspecified amount of time might have passed in-between collections. And although this is not an issue for currently intended practical applications, caution must be taken when adapting this work into other projects.

Picture 5.18 is a visual simulation of the described problem, green area corresponds periods where samples are being taken, while red area represents instants where the receiver is not actively sampling the airwaves ('deafness' periods). During these instants the software is executing other tasks, such as storing harvested information in an internal memory or setting up a new harvesting action. The ratios between areas were arbitrarily chosen and are not representative of this project.

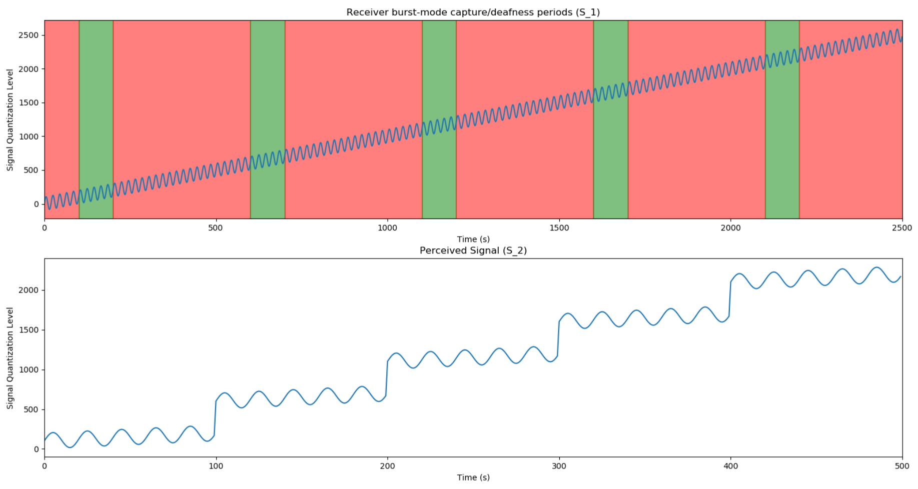


Figure 5.18: Contrast between hypothetical signal transmitted into the receiver (top) and perceived signal by the data recovery algorithm (bottom). Each green slice represents a frame.

Another way to look at this is to interpret the SDR kit working akin to a camera shutter, in that it only allows the signal to pass during specified intervals of time. The RPI, and by extension, the demodulator, only receives a series of concatenated slices of the original signal (green areas). The intended use for this setup is to demodulate an ASK signal encoded according to the protocol packets described in subsection 4.7.1. Figure 5.19 represents the difference between raw signal and the information that reaches the RPI. For demonstration purposes all packets are represented entirely by 1's, with 0's being the channel silence period between packets. Any ratios in the figure are not representative.

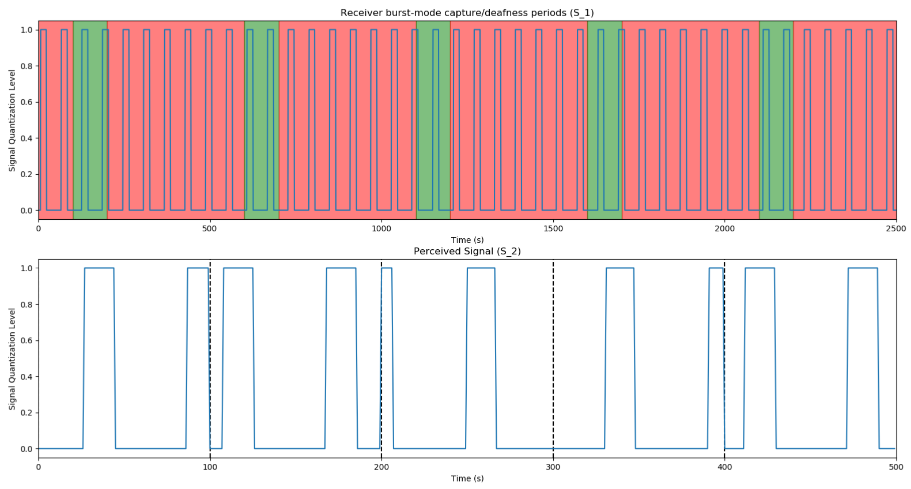


Figure 5.19: Contrast between signal backscattered by the sensor (top) and perceived signal by the comparator (bottom). One green slice represents a 'frame' of samples.

Because the signal within red areas is never sampled by the reader, it never proceeds down the pipeline in order to be fed into the RPI and eventually validated. Because these packets are lost at the reader, it becomes important to differentiate different kinds of packet loss, according to the part of the system in which the packets are lost.

It is also important to mention that because this deafness period was not an obstruction towards achieving the project goals, it has only been measured at a superficial level.

Three types of packet loss present here are:

SDR packet loss

This refers to the ratio between packets never sampled by the SDR kit and total packets, it can be roughly estimated with Equation 5.1.

$$PL_{SDR} = \frac{total\ sampling\ time}{\delta t} \tag{5.1}$$

Or more precisely, as seen in Equation 5.2, by calculation the ratio between number of packets in signals S_1 and S_2 (Figure 5.17), calculating this value requires some degree of knowledge regarding the tag/transmitter. Namely, if the tag is transmitting constantly, and what's its transmission rate.

$$PL_{SDR} = \frac{pckt\ S\_2}{pckt\ S\_1} \tag{5.2}$$

RPI packet loss

This refers to the ratio between packets that occurred during sampling times and thus reached the RPI, and those which were not successfully validated by the software demodulator. There are several reasons as to why this can happen, noise during the sampling might send a sample into the wrong side of the threshold, synchronization problems might occur or even imperfections with the data recovery algorithm or validation method might cause the loss of packets.

This statistic was treated as the most relevant of the three as it had a closer relation to the overall quality level of the developed software. Development was heavily directed in order to minimize this value. In order to measure it, the amount of validations is divided by the number of expected packets. Expected packet number (EPN) for a given time interval is calculated according to the following formula.

$$EPN = (\frac{SR}{PS + SBP * SR}) * \Delta t \tag{5.3}$$

Where PS is the packet size in symbols, SR is the symbol rate, SBP the interval of time between packets during which the channel is silent, in seconds, and Δt being the interval of time, also in seconds, for which we are calculating the EPN

This ensures that losses due to SDR sampling methods are not weighted while other potential disturbances like noise in the channel still are. Reliable 0% RPI packet loss rates will still not be feasible due to bisection of packets either at the beginning or end of burst-mode frames, which prevents their validation. The method used to validate is described in further detail on subsection 4.7.3. A potential workaround to this issue has been identified but not explored, it is detailed further under chapter 6 section 6.2

Cumulative packet loss

The product of both previous items

$$PL = PL_{SDR} * PL_{RPI} \tag{5.4}$$

5.3.2 Nomenclature

The following section contains the measurement results done either in a lab environment or outdoors. Different comparators were used, and for each one at least two different kind of test were made, distance and decimation factor.

Some designations should be properly clarified before proceeding. As previously explained, data is produced and processed in bursts of samples. During each test, three bursts of samples were collected.

The first table in each category refers to directly measured values, such as the absolute number of packets validated, the interval of time required by the processing unit to complete demodulation, the signal power measured at the receiving antenna. The second table refers to data extrapolated from the results shown in the first one. Things like average number of validated packets per frame, average time required to demodulate and RPI packet loss.

The following list serves as a reference for the result tables:

- **packet#**
  Number of packets successfully validated during collection. Each collection corresponds to a burst of data collected by the RF frontend. Two packets and the silence period between them can be seen in figure Figure 5.6. Results are numbered 1 trough 3, so packet2 would correspond to the number of packets successfully validated during the second repetition of testing.
- **runtime#**
  This value corresponds to the interval of time in seconds required by SoundGen to process the signal contained in each collection. Results are numbered 1 trough 3, so runtime2 would correspond to the interval of time in seconds required by SoundGen to process the signal contained in the second repetition of testing.
- **dBm rcv**
  This value corresponds to the signal power measured at the reader's antenna. Expressed in dBm.
- **d**
  Distance in meters between the transmitting and receiving antennas. A1/A2 for setup 1 and A2/A3 for setup 2.
- **avg p**
  The average of values packet1, packet2 and packet3 for the respective measurement.
- **percent**
  The percent of packets correctly validated by respective to all packets harvested. This value is calculated by subtracting the RPI packet loss value from 1.
- **pcket loss**
  Corresponds to the RPI packet loss. Calculated according to equation 5.3.
- **avg t**
  The average of values runtime1, runtime2 and runtime3 for the respective measurement. Expressed in seconds.
- **pps - packets per second**
  The average number of packets validated per unit of time during code interpretation. Calculated based of the values of *avgp* and *avgt*. Expressed in Hertz.

- **decim**
  Decimation order. Decimation occurs between signal filtering and comparison, as shown in Figure 4.5.
- **L Ratio**
  Upper boundary for the ratio between sampling window duration and sampling turnaround time. This turnaround is the time interval between the start of two consecutive sampling windows. Sampling window duration is represented by the green areas on Figure 5.19.

5.3.3 False Positives

The error detection mechanism implemented does not guarantee 100% accuracy. According to Equation 4.19 and knowing that during testing our payload size was 8 and CRC remainder size was 3, we can calculate an upper bound of 12.5% for false positive validations. This value corresponds to the worst case scenario of a 50% bit error rate.

5.4 Lab results

The following sub-sections present the raw data obtained via testing both of the setups. Testing variables were distance between tag and reader antennas and decimation order. All of these tests were subjected to uncontrollable influences which might have affected the results. Some happened in an indoors environment, with signal reflections between walls and different setups from other researchers also emitting signal to the air in different spectrum areas. Others were outdoors which subjected the antennas to displacement by wind and uneven terrain.

5.4.1 TMC

For the following measurements setup 2 was used with a signal generator emitting at a power of 5dBm. The sensor was calibrated with a symbol rate of 3650 Hz and a packet transmission frequency of 65.3 Hz. Three frames of samples were collected for each tested distance. The sampling rate was 226kHz and each frame consisted of 327680 samples or 1.45 seconds worth of signal. The theoretical maximum for validated packets under these conditions is 95.

Distance

No signal decimation was applied in the following test.

d	packet1	runtime1	packet2	runtime2	packet3	runtime3	dBm rcv
1	93	9.656	93	9.566	93	9.559	-70.3
1.5	88	9.274	84	9.371	85	9.363	
2	74	9.587	84	9.393	85	9.394	-75.2
2.5	80	9.49	76	9.45	82	9.359	
3	86	9.475	86	9.437	91	9.208	-77.6
3.5	22	10.649	43	9.388	42	10.01	
4	0	8.87	0	5.55	0	4.66	-78.3

Table 5.1: Number of packets validated using the TMC comparator relative to antenna distance.

d	avg p	percent	pcket loss	avg t	pps	L Ratio
1	93.00	0.98	0.02	9.59	9.69	0.15
1.5	85.67	0.90	0.10	9.34	9.18	0.16
2	81.00	0.85	0.15	9.46	8.56	0.15
2.5	79.33	0.84	0.16	9.43	8.41	0.15
3	87.67	0.93	0.07	9.37	9.35	0.15
3.5	35.67	0.38	0.62	10.02	3.56	0.14
4	0.00	0.00	1.00	6.36	0.00	0.23

Table 5.2: Packet validation data using the TMC comparator relative to antenna distance.

Decimation

Receiver and sensor antennas were 2 meters apart in the following test. SDR kit sampling rate was fixed at 226kHz, this value is before decimation.

decim	packet1	runtime1	packet2	runtime2	packet3	runtime3	dB rcv
01:02	87	6.017	90	6.035	82	6.035	-75.2
01:03	84	4.986	85	4.946	86	5.021	-75.2
01:05	65	3.94	54	3.84	60	3.877	-75.2
01:10	17	3.059	14	2.971	17	2.888	-75.2
01:15	7	2.761	10	2.804	11	2.806	-75.2
01:20	3	2.61	4	2.644	2	2.533	-75.2
01:30	4	2.498	2	2.489	5	2.486	-75.2

Table 5.3: Number of packets validated using the TMC comparator relative to decimation order.

decim	avg p	percent	pcket loss	avg t	pps	L Ratio
01:02	86.33	0.91	0.09	6.03	14.32	0.24
01:03	85.00	0.90	0.10	4.98	17.05	0.29
01:05	59.67	0.63	0.37	3.89	15.36	0.37
01:10	16.00	0.17	0.83	2.97	5.38	0.49
01:15	9.33	0.10	0.90	2.79	3.34	0.52
01:20	3.00	0.03	0.97	2.60	1.16	0.56
01:30	3.67	0.04	0.96	2.49	1.47	0.58

Table 5.4: Packet validation data using the TMC comparator relative to decimation order.

5.4.2 PBZ

For the following measurements setup 2 was used with a signal generator emitting at a power of 5dBm. The sensor was calibrated with a symbol rate of 3650 Hz and a packet transmission frequency of 65.3 Hz. Three frames of samples were collected for each tested distance. The sampling rate was 226kHz and each frame consisted of 327680 samples or 1.45 seconds worth of signal. The theoretical maximum for validated packets under these conditions is 95.

Distance

d	packet1	runtime1	packet2	runtime2	packet3	runtime3	dB rcv
1	87	3.589	94	3.587	94	3.576	-70.3
1.5	78	3.55	79	3.554	85	3.547	
2	90	3.566	83	3.571	88	3.552	-75.2
2.5	63	3.575	58	3.589	65	3.559	
3	94	3.547	92	3.551	89	3.548	-77.6
3.5	0	3.79	0	3.71	0	3.720	
4	0	3.562	0	3.56	0	3.572	-78.3

Table 5.5: Number of packets validated using the PBZ comparator relative to antenna distance.

d	avg p	percent	pcket loss	avg t	pps	L Ratio
1	91.7	96.73	03.3	3.584	25.58	40
1.5	80.7	85.12	14.9	3.550	22.72	41
2	87.0	91.81	8.2	3.563	24.42	41
2.5	62.0	65.42	34.6	3.574	17.35	41
3	91.7	96.73	03.3	3.548	25.83	41
3.5	0.0	0.00	100.0	3.740	0.00	39
4	0.0	0.00	100.0	3.564	0.00	41

Table 5.6: Packet validation data using the PBZ comparator relative to antenna distance.

Decimation

Receiver and sensor antennas were 2 meters apart in the following test. Sampling rate was fixed at 226kHz, this value is before decimation.

decim	packet1	runtime1	packet2	runtime2	packet3	runtime3	dB rcv
01:02	80	2.981	79	2.967	80	2.981	-75.2
01:03	82	2.777	76	2.773	79	2.776	-75.2
01:05	83	2.603	83	2.598	85	2.604	-75.2
01:10	28	2.469	38	2.469	23	2.467	-75.2
01:15	32	2.408	35	2.407	37	2.427	-75.2
01:20	57	2.401	45	2.395	56	2.398	-75.2
01:30	8	2.372	10	2.371	0	2.371	-75.2

Table 5.7: Number of packets validated using the PBZ comparator relative to decimation order.

decim	avg p	percent	pcket loss	avg t	pps	L Ratio
01:02	79.7	84.07	15.9	2.976	26.77	49
01:03	79.0	83.36	16.6	2.775	28.47	52
01:05	83.7	88.29	11.7	2.601	32.16	56
01:10	29.7	31.31	68.7	2.468	12.02	59
01:15	34.7	36.58	63.4	2.414	14.36	60
01:20	52.7	55.58	44.4	2.398	21.96	60
01:30	06.0	06.33	93.7	2.371	02.53	61

Table 5.8: Packet validation data using the PBZ comparator relative to decimation order.

5.4.3 FAST

For the following measurements setup 2 was used with a signal generator emitting at a power of 5dBm. The sensor was calibrated with a symbol rate of 3650 Hz and a packet transmission frequency of 65.3 Hz. Three frames of samples were collected for each tested distance. The sampling rate was 226kHz and each frame consisted of 327680 samples or 1.45 seconds worth of signal. The theoretical maximum for validated packets under these conditions is 95.

Distance

d	packet1	runtime1	packet2	runtime2	packet3	runtime3
1	74	1.458	73	1.457	77	1.442
1.5	70	1.428	73	1.472	73	1.43
2	75	1.445	68	1.445	74	1.453
2.5	55	1.339	60	1.424	54	1.496
3	43	1.187	38	1.304	18	1.887
3.5	5	1.49	4	1.494	1	0.1499
4	0	1.45	0	1.51	0	1.439

Table 5.9: Number of packets validated using the FAST comparator relative to antenna distance.

d	avg p	percent	pcket loss	avg t	pps	L Ratio
1	74.7	91.33	08.7	1.452	51.41	100
1.5	72.0	88.07	11.9	1.443	49.88	100
2	72.3	88.48	11.5	1.447	49.97	100
2.5	56.3	68.91	31.1	1.419	39.68	100
3	33.0	40.37	59.6	1.459	22.61	99
3.5	03.3	04.08	95.9	1.044	03.19	100
4	00.0	00.00	100.0	1.466	0.00	99

Table 5.10: Packet validation data using the FAST comparator relative to antenna distance.

5.5 Lab results - Graphs

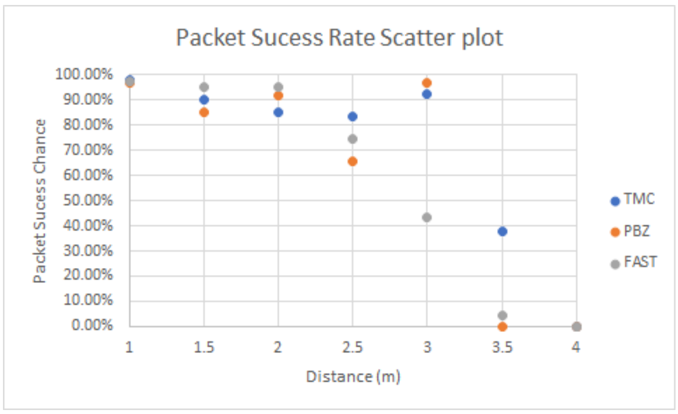


Figure 5.20: Success rate over different antenna distances and comparators

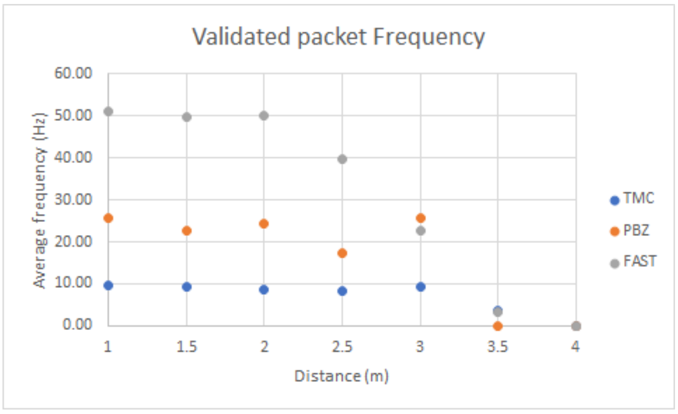


Figure 5.21: Packet throughput over different antenna distances and comparators

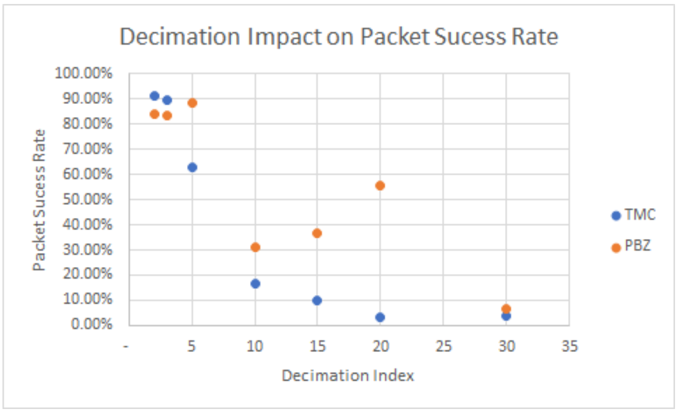


Figure 5.22: Success rate over different decimation orders and comparators

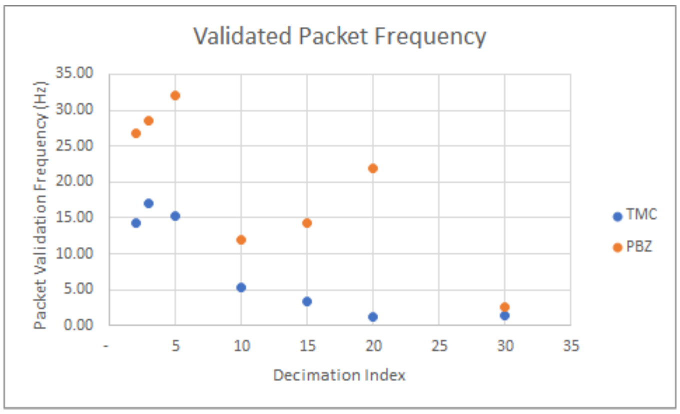


Figure 5.23: Packet throughput over different decimation orders and comparators

5.6 Field results

Field results were measured on an open slightly uneven grass terrain, at least 15 meters away from any reflective vertical surfaces such as walls.

5.6.1 FAST

Direct - Setup 1

For the following measurements setup 1 was used with a signal generator emitting at a power of 5dBm. The generator was calibrated to emit an ASK modulated signal with a carrier wave of 868MHz, a symbol rate of 3650 Hz and a packet transmission frequency of 50 Hz. Three frames of samples were collected for each tested distance. The sampling rate was 226kHz and each frame consisted of 327680 samples or 1.45 seconds worth of signal. The theoretical maximum for validated packets under these conditions is 72.

d	packet1	runtime1	packet2	runtime2	packet3	runtime3
05	71	1.397	71	1.449	70	1.445
10	71	1.451	71	1.442	71	1.457
15	71	1.447	71	1.451	71	1.451
20	71	1.447	71	1.456	71	1.452
25	71	1.452	71	1.439	71	1.452
30	71	1.451	71	1.442	71	1.441
35	68	1.453	69	1.438	66	1.467
40	27	1.458	27	1.438	31	1.433

Table 5.11: Field results regarding number of packets validated using the FAST comparator relative to antenna distance.

d	avg p	percent	pcket loss	avg t	pps
05	70.7	98.15	01.9	1.430	49.41
10	71.0	98.61	01.4	1.450	48.97
15	71.0	98.15	01.4	1.449	48.98
20	71.0	98.15	01.4	1.451	48.91
25	71.0	98.15	01.4	1.447	49.04
30	71.0	98.15	01.4	1.444	49.15
35	67.7	93.98	06.0	1.452	46.58
40	28.3	39.35	66.6	1.443	19.64

Table 5.12: Field results regarding packet validation data using the FAST comparator relative to antenna distance.

Setup 2

For the following measurements setup 2 was used with a signal generator emitting at a power of 5dBm. The tag was calibrated with a symbol rate of 3650 Hz and a packet transmission frequency of 65.3 Hz. Three frames of samples were collected for each tested distance. The sampling rate was 226kHz and each frame consisted of 327680 samples or 1.45 seconds worth of signal. The theoretical maximum for validated packets under these conditions is 94.

d	packet1	runtime1	packet2	runtime2	packet3	runtime3	dB rcv
1	72	1.447	71	1.449	74	1.440	-70.7
1.5	66	1.450	65	1.453	65	1.451	-74.8
2	72	1.438	70	1.457	68	1.437	-74.3
2.5	64	1.554	63	1.926	59	1.485	-75.3
3	63	1.487	65	1.442	65	1.456	-75.9
3.5	37	1.464	33	1.425	31	1.456	-75.6
4	10	1.441	07	1.468	07	1.402	-76.2
4.5	00	1.448	00	1.448	00	1.454	-75.3

Table 5.13: Field results regarding number of packets validated using the FAST comparator relative to antenna distance.

d	avg p	percent	pcket loss	avg t	pps
1	72.3	76.95	23.0	1.445	50.05
1.5	65.3	69.50	30.5	1.451	45.02
2	70.0	74.47	25.5	1.444	48.48
2.5	62.0	65.96	34.0	1.655	37.46
3	64.3	68.44	31.6	1.461	44.01
3.5	33.7	35.82	64.2	1.448	23.25
4	08.0	08.51	91.5	1.437	05.57
4.5	00.0	00.00	100.0	01.45	00.00

Table 5.14: Field results regarding packet validation data using the FAST comparator relative to antenna distance.

5.7 Result analysis

Some results can be distilled after analysis of the tables and plots in the previous chapter. In this section we will attempt to extract information from the results.

The most direct interpretation we can infer is that the reader is receiving signal and demodulation is successfully being concluded. This is true for all three comparison methods.

Furthermore, regarding these different methods, additional details deserve clarification. Let's start by looking at each comparator individually.

5.7.1 TMC

Starting with the direct results obtained by the project using the TMC option. As seen in Table 5.1, results are good with almost every transmitted packet being validated given a 1 meter distance between antennas, with validation rates initially decreasing slightly with an increase in distance and then abruptly when a range of roughly 3.25m is reached. We can also see the average runtime for the entire process from signal receiving to validation takes on average around 9.5 seconds for each 1.45 seconds of signal, except in the highest range where it drops to 6. This drop is due to the incapability for the system to distinguish any packets at all from the background noise led to a uniform variance level, which in turn caused the comparator to skip to the next step entirely, saving resources and moving to the next batch of data quicker.

On Table 5.2 we can see that TMC consistently maintains around 10% packet loss values up to distances of 3, and 60% up to 3.5 meters, the best result in all comparators in terms of range. This is due to the fact it was implemented with this goal in mind, it includes a more thorough information processing method, designed to be able to operate in situations with a lower signal to noise ratio. However, due to this, it also validates on average the lowest amount of packets per second. Despite being successful at validating the information in each receiving window, it takes a vastly superior amount of time to process that information, 3 times as much if compared to PBZ and up to 6 times if compare with FAST. The direct consequence of this is that these windows become temporally sparser. And if in the same time interval much less potential packets are collected, fewer validations will also take place.

Regarding signal decimation pre data recovery, we can see by Table 5.3 and Table 5.4 that it has a big impact on frame processing times. Decimation rations can be increased at least until 1:3 without a noticeable drop in result quality, further increases do decrease calculation times at the expense of also decreasing its ability to validate packets, bringing the overall amount of packets validated per time unit lower.

The results point at this comparator being the best choice for ranges between 3 and 3.5 meters, if CPU load is a concern decimation can be calibrated at up to a ratio of 1:3 without impacting result quality.

5.7.2 PBZ

The PBZ algorithm started out as an attempt to lower runtime intervals measured in TMC tests. It was a way to explore if moving focus away from antenna distance as the critical factor, but without disregarding it completely, would yield better global results. During the first iterations of this recovery method, there was no second digital filter implemented. This was due to the already heavy strain imposed on the CPU by its TMC counterpart. This meant that a higher amount of noise was present in the signal, and while this was no issue for TMC due to its higher resilience to noise, it became an issue here. Fortunately, the amount of processing resources freed by switching to a more CPU-efficient alternative also meant that there was now room to add additional signal processing stages to the pipeline, such as the digital bandpass filter described in section 4.4. So even though this algorithm was designed to maximize computing efficiency, it also made possible the inclusion of tools which indirectly increased signal to noise ratio and thus antenna distance.

Regarding this speed optimization, the results are directly noticeable, as displayed by the average runtime of 3.5 seconds for equivalent amounts of signal, as can be seen in Table 5.5 and Table 5.6. Lowering the runtime to roughly a third of the value previously measured in the TMC comparator also achieves three other benefits. Firstly, not spending so much time doing calculations means collecting signal more often, and as such the L ratio goes up. Secondly, because the L ratio goes up, so does the maximum potential amount of packets validated in a fixed time interval. This increase is also substantial, to the tune of a 2.5 factor, as we can verify by comparing the `pps` column of Table 5.2 and Table 5.6. Lastly, in time-critical implementations with ranges up to 3 meters, lower runtimes also mean lower reaction times by the system to environment changes.

Regarding distances, there is a slight drop in performance. The system is no longer able to detect tags that are 3.5 meters from the reader's antenna, as opposed to TMC.

The results presented here alongside the implementation methods described in subsection 4.6.3 suggest this comparator is ideal for applications with distances below 3 meters, imposing a few communication protocol restrictions.

5.7.3 FAST

FAST existed as a way to capitalize on the short packet sizes used by the tag under testing. Altough PBZ was already relatively fast, it still was not taking advantage of this potential optimization.

Due to the once per packet synchronization method implemented by FAST, no infinitesimal increment sweeping of the samples is necessary. The algorithm synchronizes only once in the packet beginning. As a result of this, as we can see in Table 5.9 and Table 5.10 the interval of time necessary to process information is approximately half as long as PBZ and six times smaller than TMC.

No decimation tests were conducted as processing time was already similar to collection time, leading to no useful optimization in that front.

5.7.4 Additional observations

During outdoor testing, regarding setup 1, we can see in tables Table 5.11 and Table 5.12 that unlike any other scenario, removing the tag and testing with a generator simulating its signal causes the resulting values to be much more stable over a much wider range of distances, this implies that the previously measured variability in the results is caused by the tag and not by a fault in the reader or the environment.

Another important aspect is that for the same comparator, FAST, higher distances were achieved during outdoor testing, validating the fact that laboratory environments are more prone to RF interference.

In the next chapter we will attempt to contextualize these results with the project objectives listed in chapter 1 during the next chapter.

Chapter 6

Conclusion

In the final chapter on this thesis, let us summarily review the entire document's chain and draw conclusions.

6.1 Final Remarks

With each passing year, IoT takes shape as less of a concept and more of a reality. In order to keep up new technologies have a strong development incentive. Battery-less backscattering sensing tags are one of these examples, and by extension, a reader able to interface with them. Low cost is a very welcome addition as it removes entry barriers to implementation, and as such it was added to the list of goals.

This project began as a proof-of-concept challenge regarding the viability of reliable information transfer between a sensor and the network. This has been achieved. Distances of up to 4 meters have been validated. Very low packet losses (under 5%) have also been verified over short distances. This reader is able to receive ASK signal from a backscattering source and also from any other transmitter, in the case of direct transmission, distances of up to 35 meters have been measured. Additionally, decimating capabilities have also been implemented in order to optimize execution speeds. As itemized before in chapter 1, the specific objectives were three, lets re-approach them now individually.

6.1.1 Signal reception and demodulation

As we can infer by the results in section 5.3, the reader preforms these functions. Performance levels vary depending on range, environment and decimation ratios. It is safe to conclude that validations can be achieved to a satisfactory degree (±85%) in ranges up to 2.5 meters.

6.1.2 Low-cost

The prototype is priced at 50€, consisting of a RPI costing 31€ and an RTL-SDR priced at 19€. It currently beats the cheapest alternative named in chapter 1 by a factor of 20.

6.1.3 Adaptability

Due to the inherent advantages software based implementation, adapting to changing requirements is simple. Furthermore, three distinct data-recovery methods are already provided, each with its own ideal use-cases. At the communications protocol level, freedom is also given to the user, as the protocol itself can be calibrated to suit specific needs of a given implementation. A guide on how to do this is available to the end user and can be found in Appendix B.

6.2 Future Work

In order to maximize the usefulness and potential applications of this prototype, it would be beneficial to minimize the limitation described in subsection 5.3.1 regarding the 'deafness' periods in-between each burst-mode collection. One potential solution to this problem resides in the `rtlsdraio` module included in the `pyrtlsdr` [22] library.

Other vectors vectors of improvement still persist, such as:

- **Forward Error Correction** Recuperating corrupted information instead of simply discarding it might add value and open the door to other potential implementations.
- **Testing with several simultaneous tags or readers** Pushing in the expandability direction will help us identify potential challenges to coverage limits.
- **Development of a more sophisticated network integration method** A more elaborate process than a file dump could improve the desirability of this project. IoT Gateways are an example on how to bridge the gap between this thesis and the cloud.

Appendices

Appendix A

Developers' Tools

During development, several tools were created which do not have a specific place in the main data pipeline. They exist for debugging, ease of development of testing purposes and will be described in this chapter.

A.1 Internal signal generator

As an alternative to external signal reception, the program also has the ability to generate simulated signal internally. This is useful as a debugging tool and also as a test-bed for feature implementation. No user interface exists at the time and in order for the user to tweak signal generation configurations a direct code edition is necessary.

The internal generator was made with the aim of reproducing as faithfully as possible stretches of signal that were commonly harvested from the real world setup illustrated in subsection 4.3.1, this includes the intentional addition of white noise. Settings that can be tweaked by the user include:

- Signal amplitude
- Signal frequency
- Packet protocol structure
  - Preamble size/content
  - Payload size/content
  - CRC divisor size/content
- Temporal spacing between packets
- Noise amount/distribution

An introduction on how to tune these settings can be found on the User's Guide attached to this thesis. In figure A.1 we can see an excerpt of internally generated signal, with and without deliberately added noise.

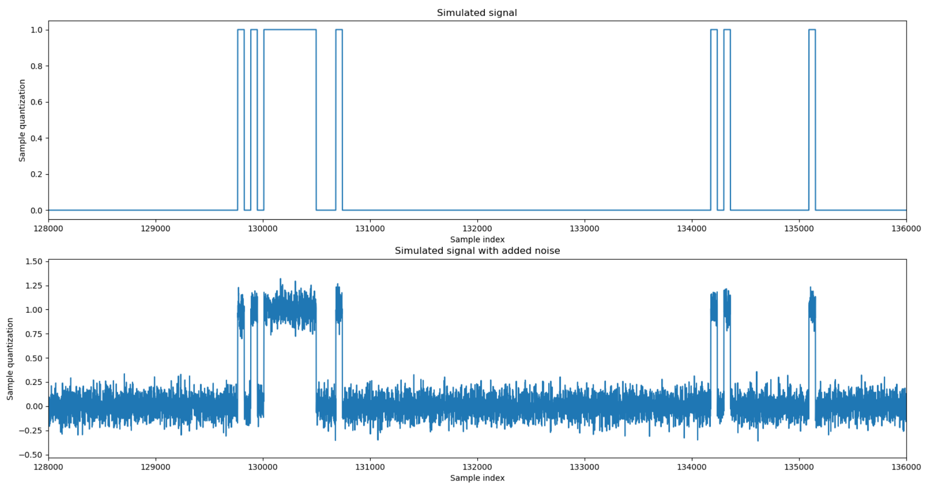


Figure A.1: Internal signal generation with optionally added noise.

A.2 Filter simulation tool

A filter simulation tool is included in the project, titled *filterSim.py*. It plots several filter characteristics on the same chart in order to facilitate design and adjustment by the user.

It features 4 sub-plots:

- Power Spectral Density plot
- Filter frequency response plot
- Signal Excerpt pre-filtering
- Signal Excerpt post-filtering

The user interface is visual only, changes require code edition and re-running the script. This tool outputs a plot, an example can be seen on Figure A.2.

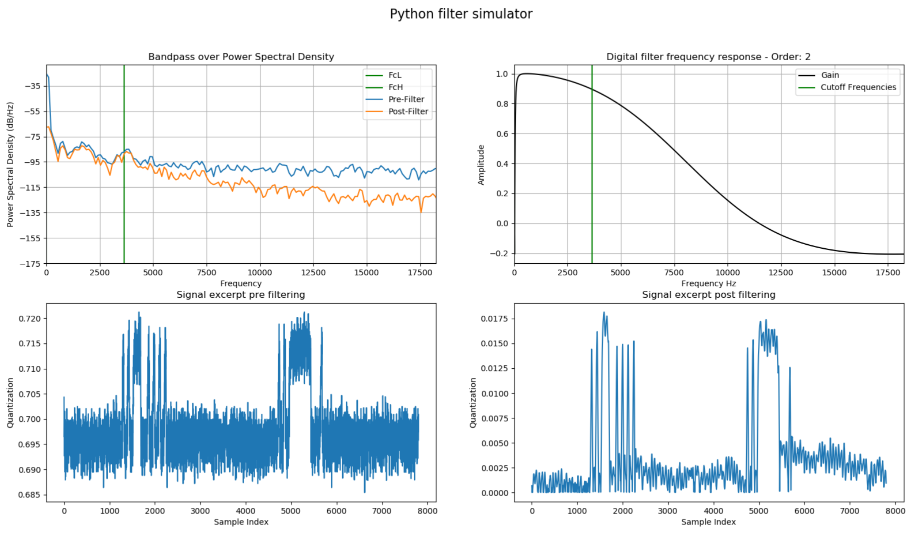


Figure A.2: Filter simulation tool

A.3 Debugging Tools

One of the most important aspects of software development is its debugging tools depth. In order to facilitate this project's development, and also to keep the door open for any future contributors, a set of debugging mechanisms were also implemented. They execute heavy lifting tasks that would be too burdensome and not entirely essential for default mode execution but provide helpful insight in diagnosing possible misdirections which may arise or be identified. These tasks include signal slicing, graph creation and runtime interval measurements.

Each data recovery function comes with inbuilt information feedback mechanisms, these are not meant to be activated during default usage but only in irregular situations to diagnose issues. Using them will slow down and at certain points even halt the normal flow of the program. Distinct debug variables must be turned `True` or `False` in the code. They are assigned in the very first few lines of the its respective file. Called `debug`, `debug1` and `debug2`, their default value on a fresh installation comes as `False`.

A.3.1 During script interpretation

TMC

debug

Activates the plotting mode, two different plots will be generated per frame, Figure A.3 and Figure A.4. In the first one we are able to visualize all the received samples, their neighbourhood variance levels and respective variance threshold level, in a similar fashion to figure 4.22 on page 41. On the second plot a visualisation of the final symbol synchronization will be displayed, akin to figure 4.29 on page 47. The program execution will halt while any of the plots remains on the screen, to proceed the user must close the plot and press enter.

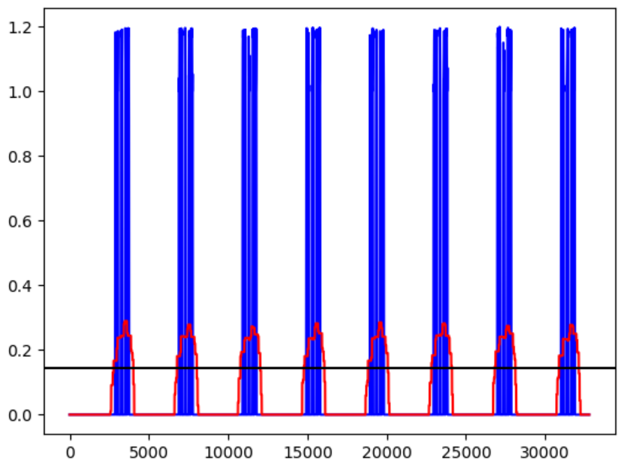


Figure A.3: Debug variance plot.

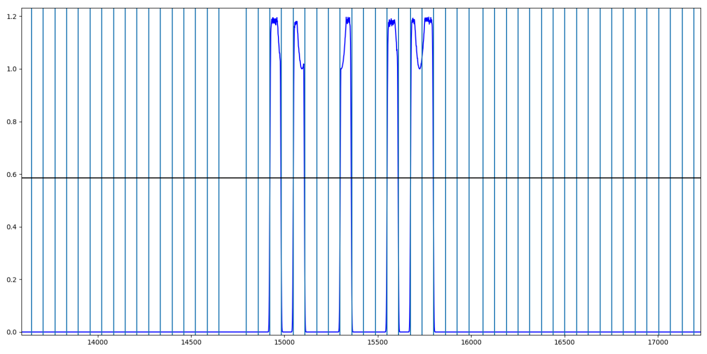
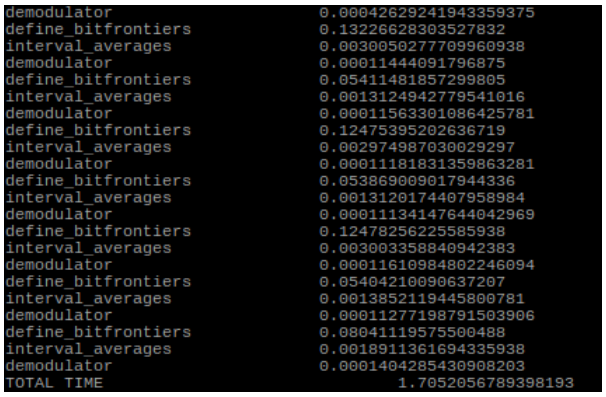


Figure A.4: Debug synchronization plot.

debug1

Activates the partial time-stamp mode. A stamp of each function's execution duration, in seconds, will be displayed on the terminal, as seen in Figure A.5. An additional stamp with the total runtime will be printed just before the library concludes executing and returns to the main. This is useful to understand which blocks of code are consuming the biggest slices of processor time.



```
demodulator                         0.00042629241943359375
define_bitfrontiers                 0.13226628303527832
interval_averages                   0.0030050277709960938
demodulator                         0.00011444091796875
define_bitfrontiers                 0.05411481857299805
interval_averages                   0.0013124942779541016
demodulator                         0.00011563301086425781
define_bitfrontiers                 0.12475395202636719
interval_averages                   0.002974987030029297
demodulator                         0.00011181831359863281
define_bitfrontiers                 0.053869009017944336
interval_averages                   0.0013120174407958984
demodulator                         0.00011134147644042969
define_bitfrontiers                 0.12478256225585938
interval_averages                   0.003003358840942383
demodulator                         0.00011610984802246094
define_bitfrontiers                 0.05404210090637207
interval_averages                   0.0013852119445800781
demodulator                         0.00011277198791503906
define_bitfrontiers                 0.08041119575500488
interval_averages                   0.0018911361694335938
demodulator                         0.0001404285430908203
TOTAL TIME                                  1.7052056789398193
```

Figure A.5: Debug time-stamps.

debug2

Activates the global time-stamp mode. Prints on terminal the total runtime of each data recovery method call. Equivalent to the previous option but prints only the last line, without the step-by-step stamps.

A.3.2 PBZ and FAST

debug

Generates a graph similar to the one seen on Figure 4.33, but encompassing the whole signal instead of simply a packet. Information in the plot contains horizontal threshold level, synchronization points and also sampling points.

A.3.3 After script interpretation

Another built in tool useful to debug your code are the various logs produced by the program. This is only possible if the user set the `debug` variable to `True` when initializing the program. The program will overwrite over these files without asking if they already exist from a previous execution. All of these files will come in `.npy` format, they are:

outfile_samples.npy

Contains a list with all the samples harvested since program start.

outfile_signal.npy

Contains a list with all the symbols demodulated since program start.

oufile_SPB.npy

Contains a single value, which equals k, as seen in equation 4.6 on page 43

All of these files can be opened and plotted in MATLAB. A MATLAB script capable of doing so, titled 'Graphmaker.mat' can be found in the repository.

A.4 Repository / Documentation

A repository of this software is maintained online via GitLab [37]. A User's guide is also provided in Appendix B, for setup and usage instructions, it serves as the official documentation.

A.5 MATLAB version

The original implementation of this project was done in MATLAB. Its repository can be found in GitHub[38], it includes a smaller amount of features, no support or documentation is provided for this release.

Appendix B

User's Guide

Welcome to ID.all's documentation!

What is ID.all:

SoundGen is a flexible burst mode ASK signal demodulator and frame processor, created with commercial DVB-T USB dongles in mind. For a larger contextualization consult the dissertation this document is annexed to.

This software couples an easy setup with the advantage of a flexible tuning mechanism included in the SDR kit. It includes the possibility of network integration. The script has been tested only for scenarios where an emitter is transmitting ASK encoded frames in bursts, spaced by periods of silence, with constant distinct RF power levels for different symbols. This user guide is oriented towards the final user, if you are a developer who wishes to contribute please read the master thesis associated with this project. Furthermore, all the details in this document assume you have a working with a clone of the master SD card.

Example:

Demodulating a signal with a 5kHz symbol rate riding on a 95MHz carrier wave is as simple as:

```
python3 SoundGen.py -f 95000000 -sym 5000
```

The code repository can be found on Gitlab [37].

B.1 Required material

For an alternative illustrated list with on the required material please read section 5.3 on the annexed dissertation.

1. Rasperry Pi[15] (with Raspbian[27] installed)
2. SDR-RTL kit [23] acting as your RF front-end.
3. Three antennas designed for the same carrier frequency

4. A backscattering sensor, or any sort of modulated signal source

B.2 How to setup

The following is a step-by-step guide on how to setup the required hardware.

1. Install all the dependencies. This step is only necessary if you have not been provided with a cloned microSD card.

- PyLab
- RtlSdr
- argparse
- numpy
- matplotlib

2. Connect the RF front-end to your RPI via USB. Connect one antenna to the front-end remember to use an adequate antenna to the carrier wave you intend to receive.

Run rtl_test on the command line to check if the kit was properly detected and no warnings or errors were returned.

3. Make sure you are on the project's root folder. It should be located at `./SoundGen_Python/SoundGen_Python/`.

4. You are good to go! :)

How to use

`python3 SoundGen.py -h` on the terminal will give you an updated list of all the arguments you can pass into the script. They are:

Arg	Argument description	type
-f *	Center frequency tuning, should equal the carrier frequency	int
-sym	Symbol rate of expected ASK signal. Defaults to 3650.	int
-s	Sampling rate. Defaults to 226kHz. Acceptable range is 226kHz - 3.2MHz.	int
-ff	Size of the internal FIFO. Default value is 5, must be 5 or greater.	int
-sf	Frame size in samples. Defaults to 327 680. Must be a multiple of 512.	int
-g	Software gain. Defaults to 15. Acceptable range is 0 - 50.	int
-it	Number of main loop cycles before program exits. Meaningless if -i True.	int
-i	Infinite mode, program runs indefinitely if True. Defaults to True.	bool
-db	Debug mode, dumps debugging info into three distinct outfiles.	bool
-cp	Which comparator to use, -DEEP -PBZ, or -FAST. Defaults to -FAST.	bool
-gen	Generate signal internally (True) or use an external input. Default is False.	bool
-h	Ignore all other arguments, print this table and exit.	None

Table B.1: Implemented input arguments. Asterisk means mandatory.

Proper format is:

```
python3 SoundGen.py [arg] [value] [arg] [value] ... [arg] [value]
```

For example, in order to capture a signal with a 10k symbol rate on a 868MHz carrier wave, with debugging mode on and a gain of 10dB, I would type into the terminal:

```
python3 SoundGen.py -f 868000000 -sym 10000 -db True -g 10
```

B.3 Frame processing

Not only will this package demodulate ASK frames, it will also process them according to the following packet protocol.

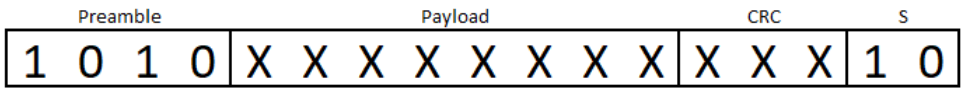


Figure B.1: Adopted signal protocol structure.

Every time a preamble is detected and the subsequent packet passes the CRC validation, its payload field will be added to a list of all the data fields successfully harvested from the current frame.

At the end of program execution (or between cycles if infinite mode is turned on), all the payload fields in that list will be dumped into a .npy file in the "*./outputs/*" folder. The user can run the following script in order to read the contents of this folder.

```
python3 readGen.npy
```

No more than 5 .npy files will ever be in that folder at any given time, if the number exceeds the limit then the oldest one will be overwritten in order to prevent memory bloating.

Each file's name will equal the time-stamp of its creation.

B.3.1 Message protocol changes

The protocol can be changed by editing the file `SoundGen.py`. In order to do so, open the file in your favourite text editor, do a text search for `‘Packet characteristics’` and edit the line immediately after. By default the line it should look like the following:

```
Packet = classGen.Packet([1,0,1,0], 8, [1,0,1,0], [1])
```

Each field is identified by comments in the code and separated by commas. They are four in total.

1. Preamble - [1,0,1,0] by default
2. Payload size - 8 by default
3. CRC divisor - [1,0,1,0] by default
4. Stop bits - [1] by default

B.4 Network integration

Currently, the easiest way to provide network integration is to remotely access the RPI (see Remote Connection) and harvest the .npy files in "./outputs/" in real time.

B.5 Debugging mode

WARNING: ENABLING INFINITE MODE AND DEBUGGING MODE IN THE SAME EXECUTION IS NOT ADVISABLE.

When the argument -db is set to ’True’, the program will produce three aditional .npy files in parallel to it’s normal "./outputs". These files will be placed in the script’s root folder. Any debugging files from previous executions of the script still present in the destination folder will be overwritten. They contain the following information:

- outfile_samples.npy A list with all the samples collected during the programs execution. This file can get very big very fast. After a few minutes of continuous execution this will become a major resource hog, it will slowdown the script considerably.
- outfile_signal.npy A list with the demodulation result of the entire program execution. It will exclusively consist of 1’s and 0’s. If multiple frames were processed the results will be concatenated into a single list.
- outfile_SPB.npy A numeric value which equals the ratio: SamplingRate/SymbolRate

These will provide you with helpful graphs and timestamps that might help with diagnosing issues or identifying optimization routes.

B.6 Infinite mode

Toggle this on if you wish for the program to run in an infinite loop, indefinitely.

B.7 GPIO

Some General Purpose Input/Output (GPIO) pins are configured for Light Emitting Diode (LED) control, in order to provide the user with additional real-time feedback.

This method works only in rigid packet mode. During normal working of the program, output signals will be produced from the RPI. These signals are intended to be used to control leds. Five different LEDs will be supported. Four of them [1-4] will be indicative of the success rate in receiving expected packets, calculated via number of received packets versus time interval. A fifth LED [5] will serve as a heartbeat display.

All resistances in Figure B.2 should equal 1kΩ.

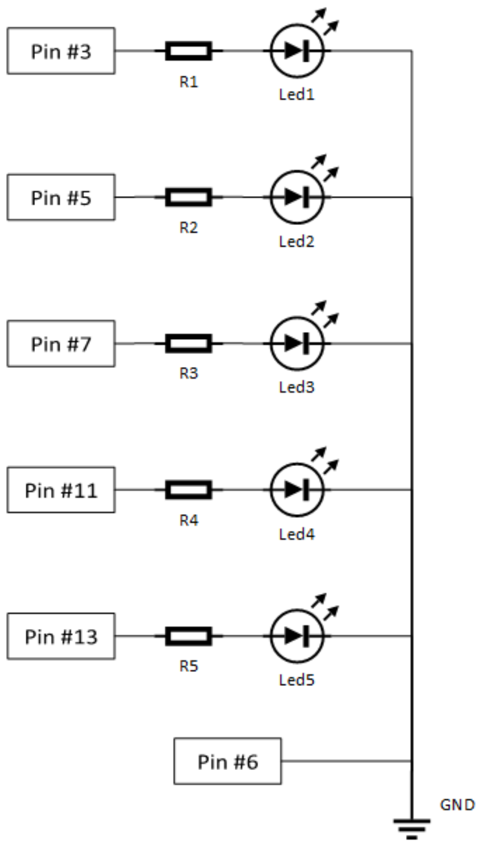


Figure B.2: Adopted signal protocol structure.

The conversion rate between packet loss and number of active leds can be seen in Table **??**.

Number of LEDs turned on	Packet loss
0	100 - 75%
1	75 - 50%
2	50 -25%
3	25 - 10%
4	<10%

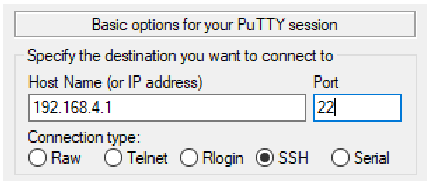
B.8 Remote connection

The main outputs produced by the program are printed via terminal, and therefore are fully compatible with remote SSH connections. File Transfer Protocol (FTP) is also possible in order to access all the outfiles. The Raspberry PI will by default create a hotspot called RPI2. The password to connect is password123.

Relevant software: PuTTY (SSH)[39] and FileZilla (FTP)[40]

By default the address the RPI will assign itself will be 192.168.4.1, this can be customized. Here's [41] a good RPI wifi access point tutorial.

Once connected to the RPI hotspot:


Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address): 192.168.4.1

Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Figure B.3: PuTTy.

```
Username: pi
Password: raspberry
```

Code 1: Default RPI user credentials.

Note: These are the default login credentials for Raspbian, and also the configured ones in the environment included in the microSD card image annexed to this thesis. This fact combined with SSH being enabled represents a security concern. The default credentials should be changed by the user, alternatively SSH can be disabled.

B.9 Autonomous mode

On the RPI's root directory a shell executable file can be found (*startup.sh*), it is configured to run on start-up, this file can be edited to make the RPI automatically start demodulating upon power on without any necessary user input. In order to activate this mode the user should edit the file with:

```
sudo nano startup.sh
```

Code 2

And uncomment the second line of code, adding input arguments as necessary. (See How to use) When all the desired changes are done. Save and close the before rebooting the RPI with:

```
sudo restart
```

Code 3

In order to deactivate the autonomous mode, the same line of *startup.sh* must be commented again.

References

[1] M. Razfar, J. Castro, L. Labonte, R. Rezaei, F. Ghabrial, P. Shankar, E. Besnard, and A. Abedi, "Wireless network design and analysis for real time control of launch vehicles", in *IEEE International Conference on Wireless for Space and Extreme Environments*, Nov. 2013, pp. 1–2. DOI: 10.1109/WiSEE.2013.6737574.

[2] R. Badia-Melis, L. Ruiz-Garcia, J. Garcia-Hierro, and J. I. R. Villalba, "Refrigerated fruit storage monitoring combining two different wireless sensing technologies: Rfid and wsn", *Sensors*, vol. 15, no. 3, pp. 4781–4795, 2015, ISSN: 1424-8220. DOI: 10.3390/s150304781. [Online]. Available: http://www.mdpi.com/1424-8220/15/3/4781.

[3] M. Kafi Kangi, M. Maymandi-Nejad, and M. Nasserian, "A fully digital ask demodulator with digital calibration for bioimplantable devices", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, pp. 1–1, Aug. 2014. DOI: 10.1109/TVLSI.2014.2343946.

[4] E. Pievanelli, A. Plesca, R. Stefanelli, and D. Trinchero, "Dynamic wireless sensor networks for real time safeguard of workers exposed to physical agents in constructions sites", in *2013 IEEE Topical Conference on Wireless Sensors and Sensor Networks (WiSNet)*, Jan. 2013, pp. 55–57. DOI: 10.1109/WiSNet.2013.6488632.

[5] *Icons8, online catalogue of icons*, https://icons8.com/icons/set/planet-earth, Available as of: 2019-03-19.

[6] R. H. Weber, "Internet of things – need for a new legal environment", *Computer Law and Security Review*, vol. 25, pp. 522–527, 6 2009, ISSN: 0267-3649. DOI: 10.1016/j.clsr.2009.09.002.

[7] H.-D. Ma, "Internet of things: Objectives and scientific challenges", *Journal of Computer Science and Technology*, vol. 26, pp. 919–924, 6 2011, ISSN: 1000-9000,1860-4749. DOI: 10.1007/s11390-011-1189-5.

[8] R. H. Weber, "Internet of things – new security and privacy challenges", *Computer Law and Security Review*, vol. 26, pp. 23–30, 1 2010, ISSN: 0267-3649. DOI: 10.1016/j.clsr.2009.11.008.

[9] F. Pereira, R. Correia, and N. B. Carvalho, "Passive sensors for long duration internet of things networks", *Sensors*, vol. 17, no. 10, 2017, ISSN: 1424-8220. DOI: 10.3390/s17102268. [Online]. Available: http://www.mdpi.com/1424-8220/17/10/2268.

[10] O. B. Akan, M. T. Isik, and B. Baykal, "Wireless passive sensor networks", *IEEE Communications Magazine*, vol. 47, no. 8, pp. 92–99, Aug. 2009, ISSN: 0163-6804. DOI: 10.1109/MCOM.2009.5181898.

[11] *Ettus usrp motherboard*, http://www.ettus.com/all-products/un210-kit/, Accessed: 2019-04-29.

[12] *Ettus usrp daugtherboard*, https://www.ettus.com/all-products/sbx/, Accessed: 2019-04-29.

[13] *Alien alr 9680*, `https://www.atlasrfidstore.com/alien-alr-9680-rfid-reader-4-port/`, Accessed: 2019-04-29.

[14] *Matlab is a multi-paradigm numerical computing environment and proprietary programming language developed by mathworks.* `https://www.mathworks.com/products/matlab.html`, Accessed: 2019-05-11.

[15] *The raspberry pi is a credit card–sized computer.* `https://www.raspberrypi.org/`, Accessed: 2019-04-05.

[16] *Python software foundation. python language reference, version 2.7.* `https://www.python.org/`, Accessed: 2019-04-21.

[17] *Tutorialspoint - simply easy learning*, `https://www.tutorialspoint.com/digital_communication/digital_communication_amplitude_shift_keying.htm`, Available as of: 2019-04-17.

[18] L. Zhang, H. Zhang, Z. Shen, M. He, X. Hao, E. Gong, L. Ye, and H. Liao, "Quadrature amplitude modulated backscatter for 2.4ghz self-powered chips", in *2018 China Semiconductor Technology International Conference (CSTIC)*, Mar. 2018, pp. 1–3. DOI: `10.1109/CSTIC.2018.8369322`.

[19] A. L. G. Reis, A. F. B. Selva, K. G. Lenzi, S. E. Barbin, and L. G. P. Meloni, "Software defined radio on digital communications: A new teaching tool", in *WAMICON 2012 IEEE Wireless Microwave Technology Conference*, Apr. 2012, pp. 1–8. DOI: `10.1109/WAMICON.2012.6208436`.

[20] *Gnuradio - the free and open software radio ecosystem*, `https://www.gnuradio.org/`, Accessed: 2019-02-25.

[21] K. Vachhani and R. A. Mallari, "Experimental study on wide band fm receiver using gnuradio and rtl-sdr", in *2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, Aug. 2015, pp. 1810–1814. DOI: `10.1109/ICACCI.2015.7275878`.

[22] *Pyrtlsdr - a python wrapper for librtlsdr*, `https://github.com/roger-/pyrtlsdr`, Accessed: 2018-09-14.

[23] *Register-transfer level, software defined radio usb kits*, `https://osmocom.org/projects/rtl-sdr/wiki/Rtl-sdr`, Accessed: 2018-09-5.

[24] D. Ball, N. Naik, and P. Jenkins, "Lightweight and cost-effective spectrum analyser based on software defined radio and raspberry pi", in *2017 European Modelling Symposium (EMS)*, Nov. 2017, pp. 260–266. DOI: `10.1109/EMS.2017.51`.

[25] ——, "Spectrum alerting system based on software defined radio and raspberry pi", in *2017 Sensor Signal Processing for Defence Conference (SSPD)*, Dec. 2017, pp. 1–5. DOI: `10.1109/SSPD.2017.8233266`.

[26] V. Tomar Vijendra Singh; Bhatia, "Low cost and power software defined radio using raspberry pi for disaster effected regions", *Procedia Computer Science*, vol. 58, pp. 401–407, 2015, ISSN: 1877-0509. DOI: `10.1016/j.procs.2015.08.047`. [Online]. Available: `http://gen.lib.rus.ec/scimag/10.1016%2Fj.procs.2015.08.047`.

[27] *Raspbian - a raspberry pi oriented operating system.* `https://www.raspberrypi.org/downloads/raspbian/`, Accessed: 2019-02-25.

[28] E. Jones, T. Oliphant, P. Peterson, *et al.*, *SciPy: Open source scientific tools for Python*, [Online; accessed 05-03-2019], 2001. [Online]. Available: `http://www.scipy.org/`.

[29] *Rpi.gpio 0.6.3 - a module to control raspberry pi gpio channels*, `https://pypi.org/project/RPi.GPIO/`, Accessed: 2019-04-21.

[30] *Librtlsdr - software to turn the rtl2832u into an sdr dongle*, `https://github.com/librtlsdr`, Accessed: 2018-09-14.

[31] *Rtl2832u, a dvb-t cofdm demodulator*, https : / / www . realtek . com / en / products / communications-network-ics/item/rtl2832u, Accessed: 2019-01-04.

[32] *Matlab comm.sdrrtlreceiver system object documentation - receive data from rtl-sdr device*, https://www.mathworks.com/help/supportpkg/rtlsdrradio/ug/comm.sdrrtlreceiver-system-object.html, Accessed: 2019-01-04.

[33] T. Elarabi, V. Deep, and C. K. Rai, "Design and simulation of state-of-art zigbee transmitter for iot wireless devices", in *2015 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, Dec. 2015, pp. 297–300. DOI: 10.1109/ISSPIT.2015.7394347.

[34] S. G. Wilson, *Digital Modulation and Coding.* Prentice Hall, 1996.

[35] *Rohde and schwarz gmbh and co kg is an international electronics group specialized in the fields of electronic test equipment.* https://www.rohde-schwarz.com/, Accessed: 2019-04-05.

[36] https://www.thorlabs.com/, Accessed: 2019-04-05.

[37] *Id.all - online repository*, https://gitlab.com/diogobatista/soundgen_python, Available as of: 2019-03-06.

[38] *Id.all - online repository for an early matlab release of this project.* https://github.com/Espigao25/SoundGen, Available as of: 2019-04-09.

[39] *Putty is an ssh and telnet client, developed originally by simon tatham for the windows platform.* https://www.putty.org/, Accessed: 2019-04-05.

[40] *Filezilla - the free ftp solution for both client and server. filezilla is open source software distributed free of charge.* https://filezilla-project.org/, Accessed: 2019-04-05.

[41] *Setting up a raspberry pi as an access point in a standalone network (nat)*, https://www.raspberrypi.org/documentation/configuration/wireless/access-point.md, Accessed: 2019-04-05.