SOFTWARE DEFINED RADIO PLATFORM

FOR

RFID PHY/MAC EXPERIMENTATION

BY

David Marmoy, Kenneth F. Fisker

Master's Thesis

Supervisors: Qi Zhang, John Rohde

Thesis on Communication Technology

Aarhus University, Institute of Engineering

March 2015

This Thesis is printed with Computer Modern 10pt
Layout and typography by the author using LaTeX
All simulated graphs designed by authors with pyplot or MATLAB™
Remaining figures designed by authors using Lucidchart.

Abstract

Many RFID protocol researchers today base their research on the 2010 *gen2reader* software from Buettner and Wetherall [7]. This software provides a partial network stack implementation of the *EPC/RFID UHF Air Interface Protocol* [20], which makes it an attractive method to save time on setup and quickly get started with the research. However, as this thesis shows, the gen2reader software has not been sufficiently tested itself and in fact contains several serious flaws that could affect the validity of measurements. In addition, the design and implementation of the software hinders extendability and modifiability of the gen2reader, and makes it difficult to reproduce results. This thesis first analyses the gen2reader software to identify these flaws, then eliminates them through a redesign that also makes the software much easier to extend and modify. The redesigned software, which we call the *MacReader* also clearly distinguishes the MAC and PHY layers. We validate the MacReader through PHY layer tests, which lead to several interesting results. We find that it is possible to affect the RFID *tag response delay* through manipulation of the RFID *interrogator* command preamble. We also identify a characteristic of the USRP N210 *software-defined radio*, that enables the MacReader to *passively* power the tag. Thereby it does not have to wait for the tag response falling edge, but instead can begin processing the response already from the first rising edge. We also profile the transmission power of the USRP N210's daughterboard, and experiment with different methods for shielding the signal between the interrogator RX and TX antennas. Finally we identify a flaw in the WISP programmable RFID tag implementation that results in erroneous encoding of the tag responses. The value of this work is threefold; the exposal of the weaknesses of an established and accepted research platform, the proposal and implementation of a platform that addresses these weaknesses, and the performance of experiments that validate the platform and leads to several useful observations regarding the PHY layer and hardware behavior.

Table of Contents

List of Figures

List of Tables

Introduction

The technology used to identify an object is only useful if its price is considerably lower than the value of the object that it identifies. When *Radio Frequency Identification* (RFID) was first envisioned in the 1930's, it was used in the identification of warplanes, to distinguish friend from foe[18]. In those early stages, the price, size and power requirements were magnitudes larger than today. As technology has advanced, price, size and power requirements have decreased to a degree where RFID identification is commonly used with individual consumer goods such as clothing items or small electronic devices. It is also used in industrial settings for asset management and warehouse inventory. These modern RFID devices, known as *tags*, are so compact that visually they are hard to distinguish from the sticker typically used to attach them to the *object-of-interest*. The true value of RFID tags over regular barcode identification is the *non-dependence on clear line-of-sight* between the reader hardware, known as the *interrogator* and the tag. Since the RFID technology is based on radio waves, the read range is instead dependent on the signal power and propagation. This is even truer for *passive RFID*, since the **passive RFID tags have no independent power source and instead feed off the power emitted from the interrogator**. This means that the read range is dictated by the ability of the tag to process this power efficiently, as well as the transmitted signal power and the propagation of the signal in the operating environment. One of the ways passive RFID tags achieve this is by not actively transmitting the response, but rather by modulating the backscatter profile of their antennas. This form of communication, known as backscatter communication, along with the harsh power restrictions, define the protocols, by among other things, dictating two distinct communication links. Today the energy efficiency of the technology has advanced to such a degree that the power supplied by interrogators is sufficient to support sensor-equipped RFID tags and tags with larger than usual processing capabilities. This vastly increases the applicability of RFID tags to a much wider range of scenarios. Sensor-equipped tags can now be embedded into structures and supply relevant sensor data for years without the need for replacement or recharging. This has opened up for new areas of research like *backscatter sensor networks* and *computational RFID*.

With the increasing capabilities and expanding usage scenarios, RFID is still a field with rich opportunity for research. However, since commercial RFID tags and interrogators are typically proprietary devices with little possibility for manipulation or improvement by researchers, it is an ongoing challenge for researchers to find equipment for real-world experiments, that satisfies both the performance requirements and allows them to modify the control software.

Software-defined radio (SDR) is a technology that presents an enormous opportunity for RFID researchers. Software defined radios make it possible to take what is usually handled in the hardware and move it to the software. Typical operations to mention are: filtering, modulation/demodulation and encoding/decoding of signals. However, SDR's

also put a higher responsibility on researchers to implement their own complete network stack. In 2006 Smith et al. [41] published an article presenting the *Wireless Identification and Sensing Platform* (WISP), a reprogrammable RFID tag, which conceptually can be thought of as a backscatter SDR. The open source firmware on the WISP attempts to comply with the current standard for passive RFID communication, the *EPC/RFID UHF Air Interface Protocol*, giving researchers the handles they need to experiment on the tag-side. On the interrogator-side, there have been many attempts to construct an SDR-based research platform during the past decade [3] [42] [30], but none of them have been adopted by the wider community. In 2010 Buettner and Wetherall [7] published an article presenting a partially implemented interrogator based on the *EPC/RFID UHF Air Interface Protocol*. The program, known as the *gen2reader*, was subsequently made available to the community for download and has been widely used by RFID researchers since then [49] [14] [45].

Both the WISP project and the gen2reader are steps in the right direction, but although the WISP project is keeping momentum[1], driven by researchers at the University of Washington, the gen2reader project has not been updated since 2010. Nonetheless, it is clear from the literature that many researchers are using the gen2reader as the basis for their research. Whereas the gen2reader project is used for testing both RFID interrogators and tags, there has been a surprising lack of source criticism when it comes to the tool itself. When we first got our hands on the project source code it was immediately clear that it was not made for widespread distribution, and definitely not to be used as a trusted scientific research tool. Further analysis has only confirmed this initial impression.

The purpose of this thesis is to rectify the flaws of the gen2reader and provide the scientific community with a trustable research platform. In the process we hope to remind researchers of the need to evaluate tools with a critical eye prior to adoption. We provide an in-depth analysis of the gen2reader software and the needs of such a tool for RFID research, and based on the results we design and implement the MacReader, a redesigned and improved gen2reader that features dynamic reconfiguration and proper decoupling between the PHY and MAC layers, besides correcting several serious flaws in the gen2reader and updating the tool stack to contemporary technologies. We validate the design decisions behind the MacReader with tests and analysis. We also perform PHY layer measurements, using the MacReader, in order to provide a proof-of-concept of the tool's capabilities. In the following sections we will describe the thesis goals, the contents of this document and the main contributions that we have made.

Thesis goals

Through this thesis we aim to further the progress that has been made regarding SDR-based RFID research. The end-goal is to provide the research community with a plug-and-play platform that, once the necessary hardware has been obtained, can be ready for use in hours. In addition to being fast and easy to set up, the platform implementation and architecture should be based on good software design principles, such as decoupling and separation of concerns, so that changes are self-contained and can be made with predictable results. The sub-goals are to:

- Facilitate ease of deployment by upgrading to a modern technology stack and good

[1] Version 5.0 was released Q4 2014

documentation

- Facilitate ease of modification through good software design
- Facilitate reproducibility of results by documenting and testing design decisions and configuration parameters
- Facilitate extendability and improvement of the platform through good software design
- Demonstrate the capabilities of the platform through tests.

Thesis contents

The rest of the thesis is split into the following 6 chapters.

1. Thesis background This chapter first provides an in-depth explanation of the terms and concepts used in the thesis. Then we establish the gen2reader as the current state-of-the-art SDR-based RFID research tool. Finally, we analyse the gen2reader for flaws and other possibilities for improvement.

2. The MacReader This chapter explains the design of the MacReader software and how it improves upon the gen2reader. We also describe the system limitations, and identify possibilities for future research.

3. Measurements and results This chapter contains descriptions of all the tests we have performed.

4. Lessons learned This chapter contains a selection of the lessons we learned while working on this thesis. We have selected those lessons that are most likely to save others time and frustration.

5. Related works This chapter relates our results and findings to the literature.

6. Conclusion This chapter concludes the thesis with a summary and a discussion of the achieved results.

Appendixes Due to the large amount of material produced during this project, we have only included the most relevant into the thesis itself. The rest can be found in the appendixes. We have also made installation and setup guides for the systems we have used, since the process of figuring out those represents months of work.

Thesis contributions

In this thesis we

- Identify and document the flaws and limitations of the gen2reader, a widely used SDR-based RFID research platform.
- Design and implement an SDR-based RFID research platform that facilitates:
  - Dynamic reconfigurability (Cognitive Radio)
  - MAC/PHY layer separation
  - Adding new functionality or changing functionality without modifying core code
  - Channel measurements
  - Ease-of-use/setup
- Profile the WBX 50-2200 MHz daughterboard's transmission power
- Test several methods for antenna shielding.
- Identify and document an encoding flaw in the WISP 4.1
- Identify and document the relationship between the tag response time and the length of the RTcal symbol

Additional reading

The following is a list of resources recommended for anyone who wishes to learn more about RFID and related technologies or to continue the work started in this thesis. We have compiled this list with the intention of providing someone new to the field with the most efficient way to get up-to-speed.

- *The RF in RFID: UHF RFID in Practice*(2012). This book provides a thorough foundation to understand every aspect of RFID technologies. It is especially useful to understand the hardware used for RFID interrogators and tags.
- *EPC/RFID UHF Air Interface Protocol*(2013) This is the current standard specifying the passive RFID protocol. It is well-written and contains everything you need to know in order to implement the protocol yourself.
- `www.wirelessinnovation.org` [26] This is a non-profit organization born from the original *SDR Forum*. They work for the advancement of radio technologies, and especially software-defined radio, cognitive radio and dynamic spectrum access. Their site contains all the resources needed to get a fundamental understanding about software-defined radio.

Nomenclature

Table 1
THE TERMINOLOGY USED IN THIS THESIS

Acronym	Description
ACK	Acknowledgement (protocol message)
API	Application Programming Interface
ASK	Amplitude Shift Keying (modulation)
BLF	Backscatter Link Frequency
CPLD	Complex Programmable Logic Device
CRC	Cyclic Redundancy Check
CUDA	Compute Unified Device Architecture
CW	Continuous Wave
DHCP	Dynamic Host Configuration Protocol (server)
DR	Divide Ratio
EIRP	Effective Isotropic Radiated Power
EPC	Electronic Product Code
ERP	Effective Radiated Power
FIR	Finite Impulse Response (filter)
FPGA	Field-Programmable Gate Array
FSA	Framed Slotted ALOHA
GDB	GNU Project Debugger
GPP	General-Purpose Processor
GPU	Graphix Processing Unit
GRC	GNU Radio Companion
GUI	Graphical User Interface
HDL	Hardware Description Language
HW	Hardware
I/O	Input/Output
IC	Integrated Circuit
IDE	Integrated Development Environment
IP	Internet Protocol (address)
KLE	Kilo Logic Elements
LTS	Long Term Support (Ubuntu release version)
M	Miller (encoding)
MCU	Micro Control Unit
MM	Mueller and Muller (clock recovery)

Continued on next page

Table 1 – *Continued from previous page*

Acronym	Description
MAC	Media Access Control (layer)
NAK	Negative-Acknowledgement (protocol message)
PCB	Printed Circuit Board
PFB	Polyphase FilterBanks (clock recovery)
PHY	Physical (layer)
PIE	Pulse Interval Encoding
Q	Slot count parameter
RF	Radio Frequency
RFID	Radio Frequency IDentification
RN16	16-bit random number (protocol message)
ROC	Rate Of Change
RSSI	Receved Signal Strength Indicator
RX	Receive(r)
SDK	Software Development Kit
SDR	Software-Defined Radio
SIMD	Single Instruction Multiple Data
SINR	Signal-to-Interference and Noise Ratio
SMA	SubMiniature version A (cable/connector)
SNR	Signal-to-Noise Ratio
SW	Software
T1	Tag→Interrogator link timing
T2	Interrogator→Tag link timing
Tari	Type A reference interval
TX	Transmit(ter)
UHF	Ultra-High Frequency
UHD	USRP Hardware Driver
USRP	Universal Software Radio Peripheral
Volk	Vector Optimized Library of Kernels
WISP	Wireless Identification and Sensing Platform
WYSIWYG	What You See Is What You Get (text editor)
ZC	Zero-Crossing (clock recovery)

Chapter 1

Thesis Background

This chapter provides a basis to understand the concepts that will be covered in the remaining chapters. In this chapter we also establish the current state-of-the-art with regards to SDR-based RFID research through a literature review. Finally we analyse the identified state-of-the-art system and find areas with opportunity for improvement.

We cover the following concepts in depth:

Passive RFID

- Background
- Backscatter communication
- The EPC/RFID UHF Air Interface Protocol standard
  - Commands
  - Inventory
  - Miller encoding
  - Pulse-Interval Encoding (PIE)
  - Amplitude-Shift Key (ASK) modulation

Software-defined radio

- Background
- GNU Radio
  - Blocks

1.1. Passive RFID

Passive RFID is the most common form of RFID, and most people are likely to have encountered at least a few passive RFID tags. Passive RFID tags are used to identify clothes and other consumer products in shops, books in libraries and luggage in airports. Passive RFID is so widely used because the tags are very small and cheap to produce, due

to low hardware requirements, such as no battery or other external power sources, and very small, low power integrated circuits (IC). An important benefit of these factors is that tags can be deployed and never need maintenance until the day that components fail. Figure 1.1 shows an example of what passive RFID tags look like. The main identifying component in passive RFID tags is their antennas, which come in many shapes and sizes depending on the use. The term *passive*, refers to the communication form used with passive RFID tags, which is called *backscatter communication*.

Figure 1.1: An example of a passive RFID tag, in this case the Alien ALN-9640 Squiggle Inlay.

1.1.1 Backscatter communication

Backscatter communication refers to a communication form where devices communicate by varying the antenna input impedance and thereby modulate the reflection of a *continuous wave* (CW) transmitted from a high-power radio [18]. For passive RFID, the continuous wave also functions as the tags' power source. All their power comes from the broadcast signal from the interrogator, which means that they can only ever passively respond to a request, never actively initiate contact. When a interrogator wishes to interrogate with a tag, it must first *charge* the tag in order to be heard, otherwise the tag will not have the power to process the transmitted command. After transmitting a high-power signal for a sufficient amount of time, the interrogator transmits its command, and then must resume the broadcast of the high-power signal which will provide the foundation for the backscattered response.

Backscatter communication has several implications to the propagation of the signal. First of all, the interrogator broadcast power has to be much higher than it needs to be for other technologies communicating at comparable distances. This is because the interrogator signal is going on a return trip to the tag and back, not just one way. Figure 1.2 illustrates the range of the communication. Because the interrogator signal has to travel both ways, all the sources of signal propagation that usually affect wireless communication, affect backscatter communication twice. "*The signal to noise ratio (SNR) for typical backscatter communication decays with the square of distance for the forward link and to the fourth power of distance for the backscatter link.*"[50]. In addition, tags only have around 30% efficiency converting the received signal into useable power, further reducing the strength of the reflected signal [18]. Because such a small fraction of the signal survives the return trip between interrogator and tag, the exchange is very susceptible to noise and interference.

RFID *interrogators* communicate with tags using a modulated *Ultra-High Frequency* (UHF) carrier wave. The specific carrier frequency depends on regional regulation. Table 1.1 shows the regulations for Europe and the United States (US).

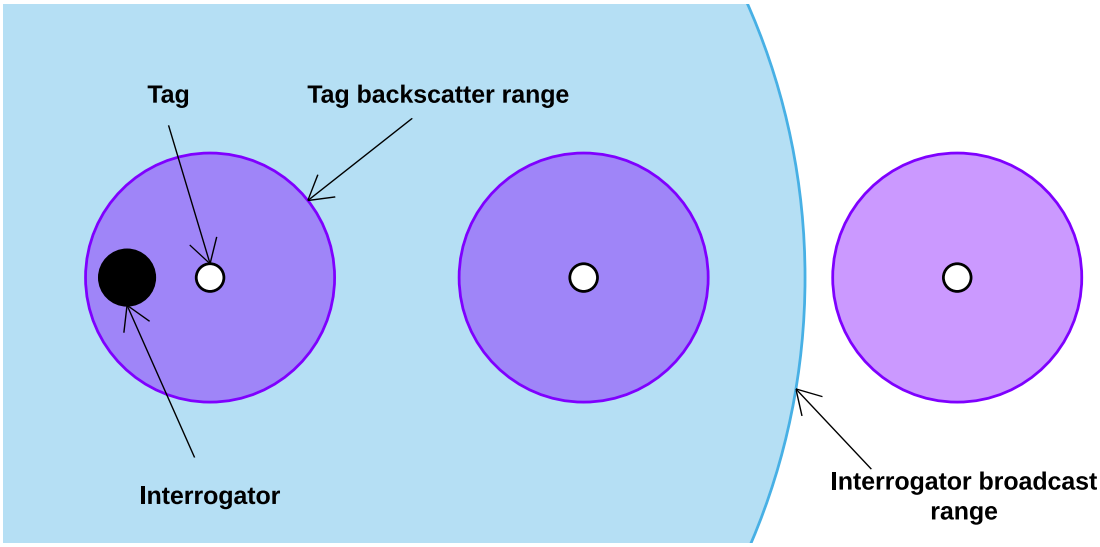


Figure 1.2: Illustration of the three possible physical communication scenarios. From left to right, the first tag is within the broadcast range of the interrogator and the interrogator is within the backscatter range of the tag, so bidirectional communication is possible. The second tag is within range of the interrogator, but the interrogator is outside the range of the tag, so the tag will receive the interrogator command, but the interrogator will not receive the response. No form of communication is possible with the last tag, since it is outside the range of the interrogator broadcast.

Regulatory agency	Frequency range	Max. power
ETSI(Europe)	865-868 and 915-921[1]	2W and 4W Effective Radiated Power (ERP)
FCC(US)	902-928	4W Effective Isotropic Radiated Power (EIRP)

Table 1.1: Regional RFID regulations [19] [25]

1.2. The EPC/RFID UHF Air Interface Protocol standard

Since 2004, the EPC/RFID UHF Air Interface Protocol standard, from hereon known simply as *the EPC standard*, has defined passive RFID communication. It has been refined several times until it was last updated in 2013.

1.2.1 Terminology

In the EPC standard, the high-powered sink node is called a reader or an *interrogator*. This thesis will use the term interrogator to avoid confusing it with the main component of the gen2reader system covered in section 1.7. The interrogator broadcasts *commands* to the tags which backscatter *responses*. The interrogator-to-tag link is modulated on the *carrier waveform*, and the tag-to-interrogator link on the *subcarrier waveform*. From hereon, the subcarrier frequency will be referred to as *the Backscatter Link Frequency* (BLF).

1.2.2 Commands

In the EPC standard, the medium access is controlled through interrogator commands. The three categories of commands are *select*, *access* and *inventory* as shown in table 1.2. This thesis will focus exclusively on the inventory commands.

Select	Inventory	Access
Select	Query	Req_RN
	QueryAdjust	Read
	QueryRep	Write
	ACK	Kill
	NAK	Lock

Table 1.2: Mandatory commands of the EPC standard. This thesis will focus exclusively on the inventory commands, marked with red for convenience.

Select commands allow the interrogator to restrict communication to a subset of the tag population, based on tag characteristics specified in the command.

Access commands are used by the interrogator to access the tag memory, i.e either *read* from or *write* to the tag. Additionally there is a subset of access commands concerned with security such as the *kill* and *lock* commands.

Inventory commands, as the name suggests are used in the inventory of the tag population. This is the part that people intuitively associate with the term RFID. The only inventory command that is somewhat complex is the *query* command. The query command initiates the inventory round and is responsible for specifying a number of PHY/MAC settings for the duration of inventory round. It consists of the 9 fields shown in figure 1.3.

	Command	DR	M	TRext	Sel	Session	Target	Q	CRC
# of bits	4	1	2	1	2	2	1	4	5
description	1000	0: DR=8 1: DR=64/3	00: M=1 01: M=2 10: M=4 11: M=8	0: No pilot tone 1: Use pilot tone	00: All 01: All 10: ~**SL** 11: **SL**	00: S0 01: S1 10: S2 11: S3	0: *A* 1: *B*	0–15	CRC-5

Figure 1.3: The table shows the fields that compose a Query command [20].

- *The command field* is the start of all commands and identifies the command itself. For the query command it is *1000*.
- *The DR field* specifies the *divide ratio* used in the calculation of the BLF.
- The M field specifies the *Miller encoding* to be used in the tag response. Miller encoding is covered in section 1.2.4.
- *The TRext field* specifies whether to prepend the response preamble with a pilot tone.

- *The Sel field* specifies which subset of the tags should respond to the query. The select commands are used for the division of the tag population into subsets.
- *The session field* is used when multiple interrogators inventory a tag population. The interrogators must know about each other and coordinate between them so that they do not use the same session.
- *The target field* is used across inventory rounds to ensure that an interrogator inventories the complete population. The interrogator specifies either target A or B until no more tags with that target respond, after which the interrogator may switch target and begin another inventory.
- *The value of the Q field* provides the basis for the MAC scheme used in an inventory round. The mechanisms behind Q, and the MAC scheme, will be explained in the next paragraph.
- *The CRC field* provides a mechanism for error-detection at the tag.

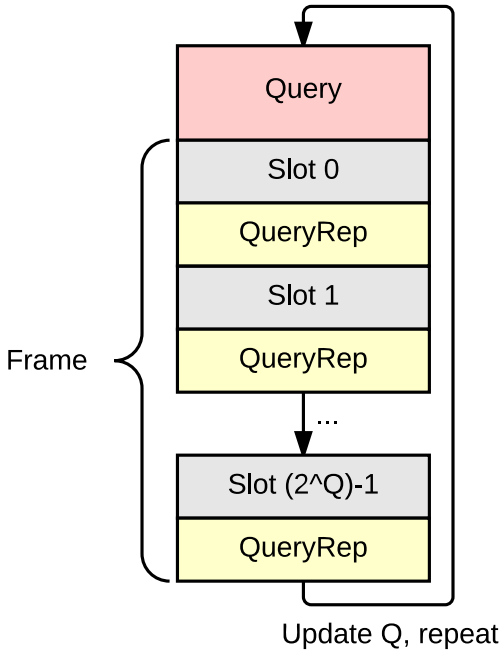


Figure 1.4: The EPC standard slotted ALOHA frame. Slots are delimited by QueryRep commands and the number of slots determined by Q.

1.2.3 The inventory round

The EPC standard uses *Framed Slotted ALOHA* (FSA) for medium access control. As mentioned in the previous paragraph, the interrogator specifies a Q-value with the query command which is used to calculate the number of slots to divide the FSA access frame into. Figure 1.4 shows the relation between Q and the number of slots. FSA is a random

access scheme where the tags generate a random number seeded from Q to determine which slot they should respond in. There is no coordination between the tags since they cannot independently transmit and since the slot selection algorithm is random, three things can happen in any given slot, as illustrated in figure 1.5.

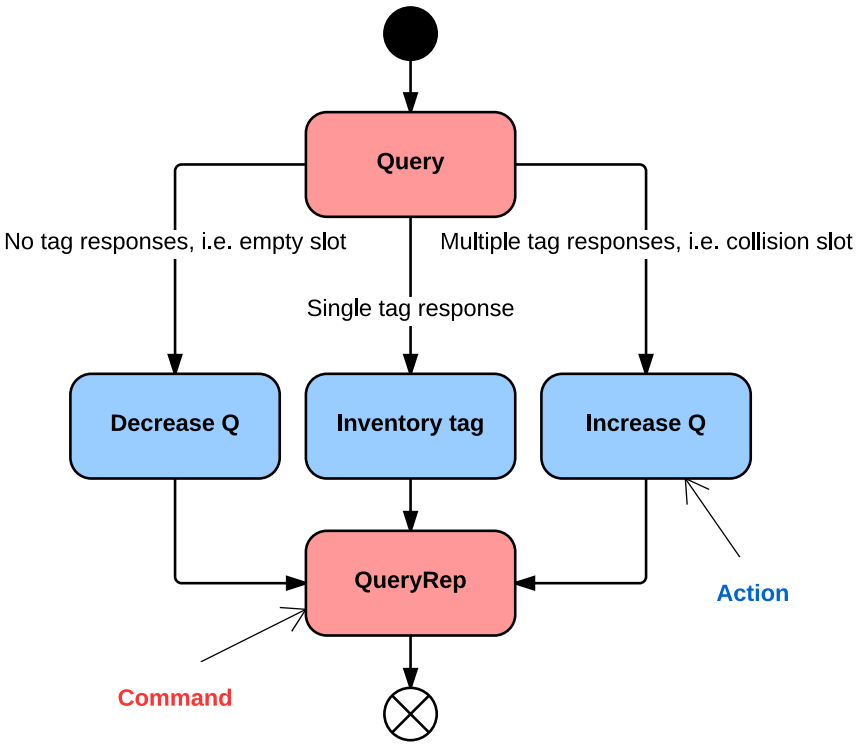


Figure 1.5: The state diagram illustrates how the interrogator adapts the number of slots to the tag population density. Note that this is a simplification, ignoring the possibility of updating Q with the QueryAdjust command, and only showing a single slot

- no tags respond
- a single tag responds
- multiple tags respond

It is the responsibility of the interrogator to detect collisions and mitigate their effect. In order to minimize the probability of collisions, the number of slots within a frame is chosen based on a best estimate of the size of the population. The optimization goal is to maximize the number of slots with a single tag response and minimizing the number of slots with zero or multiple tag responses. The FSA medium access is temporally divided, but the length of time of the frame or even each slot is not known in advance. That is because a full interrogator-tag inventory exchange may range from zero exchanges if the slot was empty, to two exchanges through the tag handshake and identification phase. Because the length of the slot cannot be known in advance, the slots are delimited by the QueryRep command, which informs the tags of the beginning of the next slot. When a

tag receives a QueryRep command it decrements its slot count. This is repeated until the slot count reaches zero at which point the tag responds to the interrogator. Figure 1.6 illustrates the tag state machine, beginning from the slot selection and ending with the initiation of the tag-interrogator handshake.

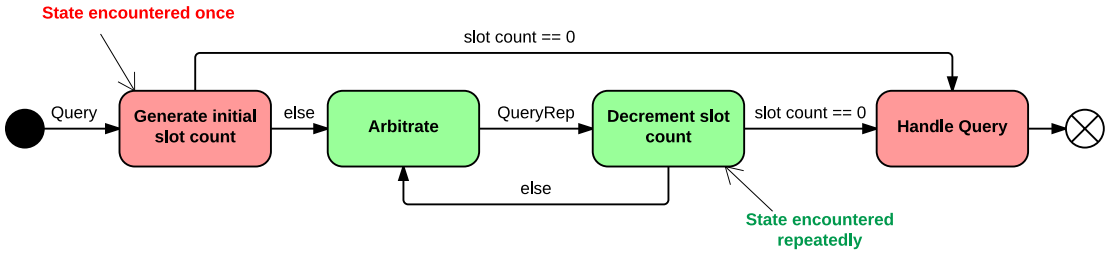


Figure 1.6: The figure illustrates how the tags select their slots.

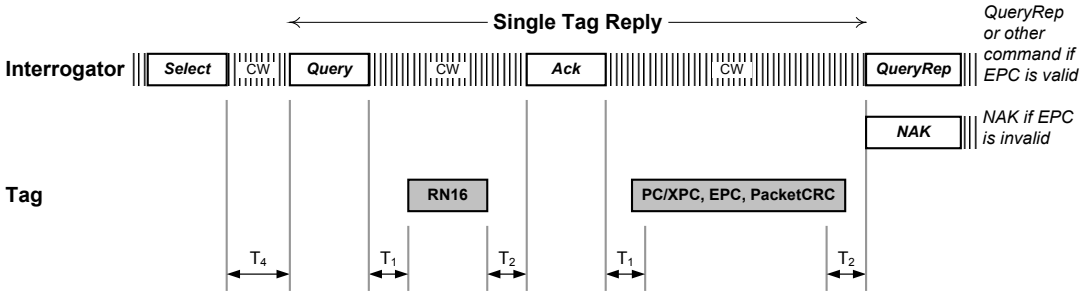


Figure 1.7: The figure illustrates a succesful inventory of a single tag [20].

When a tag responds to a query command, it backscatters a random 16-bit number, called RN16, which begins the handshaking mechanism referred to above. The interrogator responds to a received RN16 with an ACK command, which echoes the received RN16 and finishes the handshake. This exchange is illustrated in figure 1.7. If the interrogator has correctly received an RN16 response and sent an Ack command in return, the tag responds by backscattering its *Electronic Product Code* (EPC). The interrogator verifies the validity of the EPC using CRC, and if the EPC is valid, it advances to the next slot by transmitting a QueryRep. If the EPC is not valid, the interrogator transmits a NAK command and advances to the next slot.

1.2.4 Miller encoding

The tag responses are *Miller-encoded*, which means that the binary response is converted into data-1 and data-0 symbols like the ones shown in figure 1.8. The EPC standard specifies 4 variants of Miller encoding, defined by the number of clock cycles M that make a data symbol, either 8, 4, 2 or 1, with the last being referred to as FM0 encoding. In this thesis we will exclusively use Miller-4, since that is the encoding supported by the WISP tags. A Miller data-1 symbol is simply M square-wave subcarrier cycles with a phase-inversion in the middle of the symbol resulting in what we will refer to as a *Miller long-pulse*. Data-0 symbols are also M square-wave subcarrier cycles, but without phase-inversion. However, when two consecutive data-0 symbols occur, there is a phase-inversion

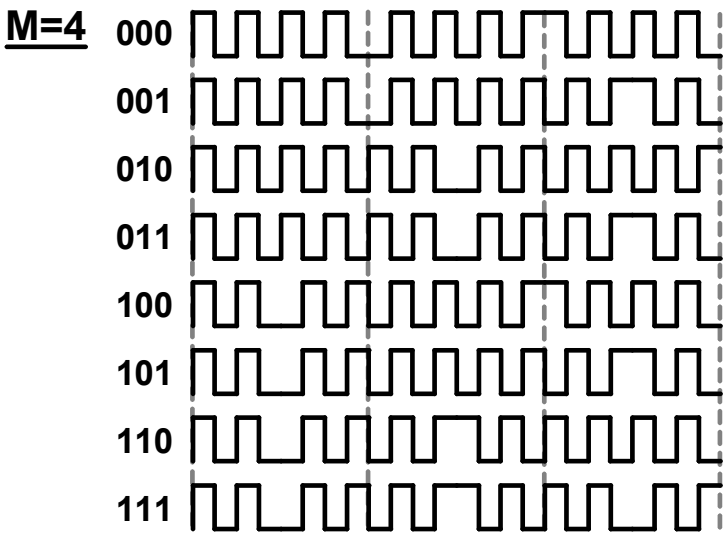


Figure 1.8: Illustration of how 3-bit sequences translate into Miller-4 symbols [20]. The number *4* refers to the number of clock cycles per symbol

on the border between them. We recommend taking a moment with figure 1.8 to get familiarized with the data-symbol signatures. Note how both data-1 and data-0 symbols have an equal number of high pulses and low pulses. Also note that the sign ($\pm$) of a data symbol is invertible and depends on the previous symbol. Finally note that it is possible to detect errors in a Miller-encoded signal by validating the placement of the phase-inversions. For example:

- No two phase-shifts can be closer than M cycles as in a binary *000*-sequence,
- No two phase-inversions can be further away than $2 * M$ as in a binary *101*-sequence
- If a long-pulse is low, then the next long-pulse must be high and vice versa, which also means that there must be an even number of cycles between phase-inversions.

The encoding process itself consists of feeding the binary message through a finite state machine of valid next-states, as shown in figure 1.9, which will convert the message into a baseband waveform.

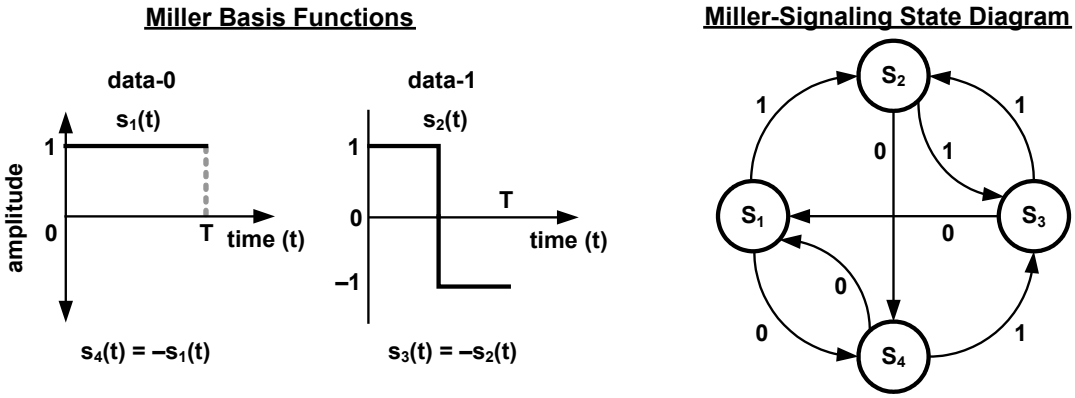


Figure 1.9: The Miller basis functions and generator state diagram [20]

That waveform is then multiplied by a square-wave signal corresponding to the BLF. The complete process is illustrated in figure 1.10. One of the benefits of Miller encoding

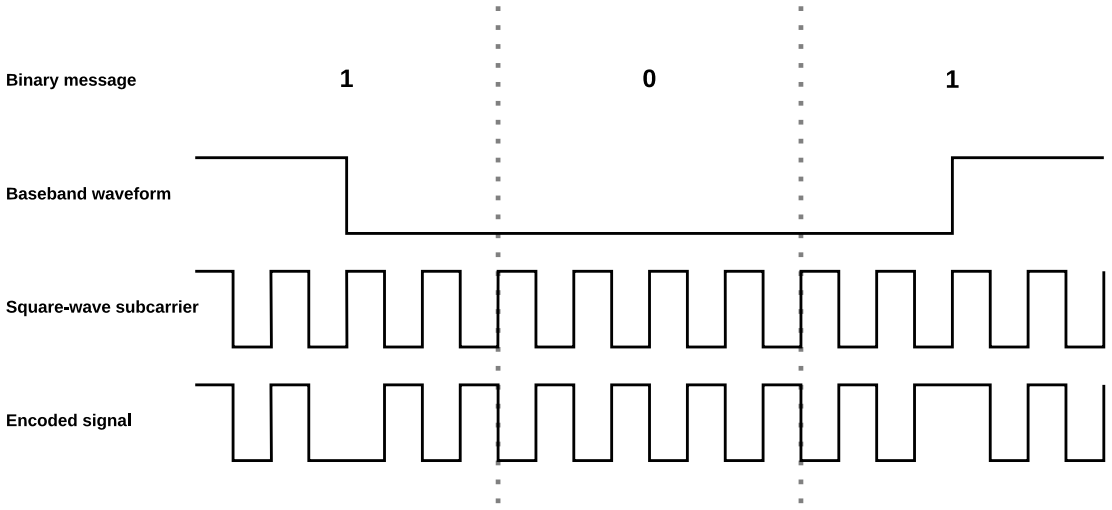


Figure 1.10: An example of Miller encoding, where a binary sequence of *101* is encoded using Miller-4, i.e. with 4 subcarrier cycles per symbol. The figure shows the initial message, the baseband waveform after encoding, the square-wave subcarrier at 4 cycles per symbol, and the final encoded signal after multiplying the baseband waveform by the subcarrier.

is that the encoded signal contains information about the clock that was used to encode it.

Headers The tag responses are prepended with a header that depends on two parameters: the Miller encoding, and the TRext field in the query command. The TRext field determines whether the preamble should be prepended with a pilot tone. The pilot tone is just 12 symbol lengths of unencoded subcarrier cycles. The Miller encoding determines how the data-1 and data-0 symbols in the preamble look, as can be seen in figure 1.11.

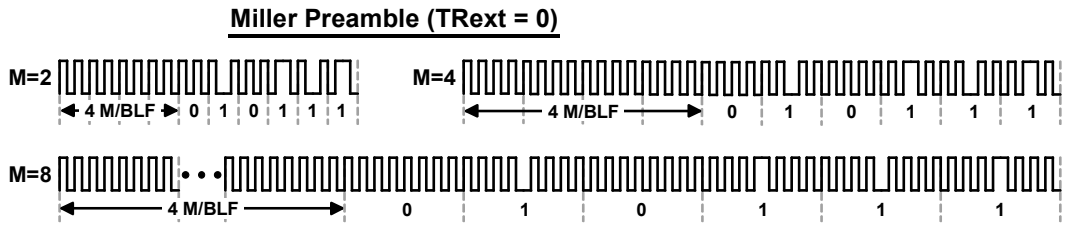


Figure 1.11: Miller preambles [20]. Note that in addition to the 4 symbol lengths of unencoded subcarrier cycles that can be seen in the figure, it is possible to have 10 more symbols lengths if the TRext field is set to 1.

1.2.5 Pulse Interval Encoding (PIE)

The interrogator commands are PIE encoded, which means that the binary command is converted into data-1 and data-0 symbols like the ones shown in figure 1.12. PIE

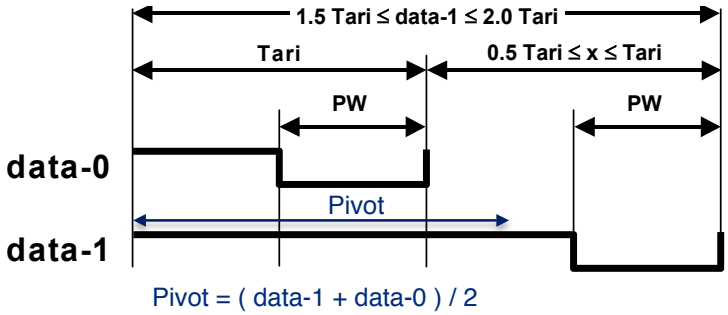


Figure 1.12: PIE symbols [20]. Note that we have added annotations to illustrate the pivot length as calculated by the tags.

symbols are different from Miller symbols in several ways. First of all, the data-1 symbol is asymmetric, since it does not have a mean of 1/2. Secondly, they are not tied to the carrier cycle, but rather to a reference value called *Tari* (Type A Reference Interval). Tari defines the length of a data-0 symbol, which in the case of the EPC standard is in the range of 6.25 µs to 25 µs. The length of a data-1 symbol is defined to be in the range of 1.5 Tari to 2.0 Tari.

Headers Interrogator commands are prepended by headers, which contain information about the length of the data symbols in the form of an RTcal symbol. The length of this symbol is equal to the combined length of a data-1 and a data-0 symbol, and the tags use it to calculate a pivot length, which is shown in figure 1.12 This pivot length is compared to the length of each subsequent symbol, with symbols shorter than the pivot converted to 0 and symbols longer than the pivot converted to 1. Interrogator headers can take two forms: either a preamble or a framesync. Figure 1.13 shows the two header types, and how the preamble is really just a framesync with a TRcal symbol appended.

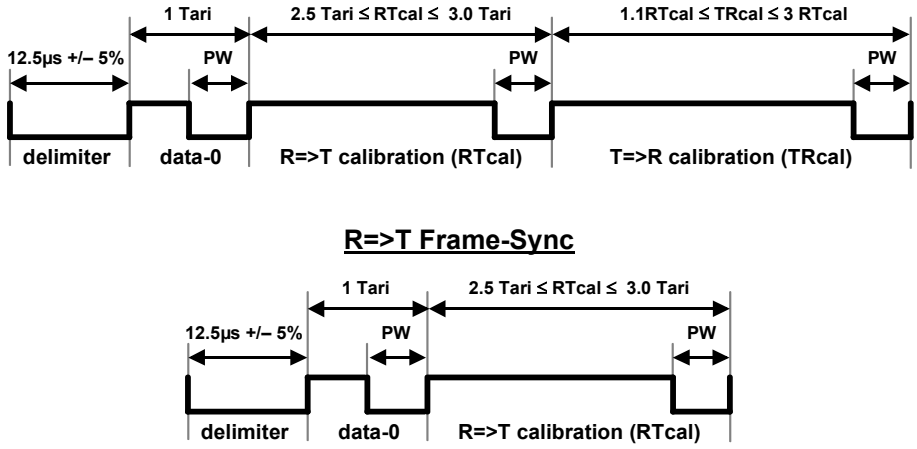


Figure 1.13: Interrogator→tag headers [20]. Note that the preamble consists of a framesync with a TRcal symbol appended.

1.2.6 Amplitude-Shift Keying (modulation)

The EPC standard specifies *Amplitude-Shift Keying* (ASK) as the method for modulation. ASK modulation multiplies the information signal with the carrier signal, thereby giving the signal high amplitude when the information signal is high, and low amplitude when the information signal is low.

1.3. Software-defined radio

Software-defined radio (SDR) is defined as "*Radio in which some or all of the physical layer functions are software defined* [23]". Software-defined radios are intended to increase flexibility compared to radios implemented fully in hardware. The underlying idea is to view the physical layer as a chain of modules, which the signal must pass through, terminating at the antenna. In software-defined radios the hardware modules in this chain are replaced by software modules as much as possible. This concept is illustrated in figure 1.14 Hardware implementations are optimized for performance, and in exchange sacrifice the ability to modify the specific implementation. There are scenarios however, such as research, where the ability to change the implementation weighs heavier than the raw performance. Table 1.3 shows the benefits and penalties of using SDR-based tools for research [24].

Benefits	Penalties
Easy to change implementation details	Lower performance than hardware-implementations
Short time from idea-to-implementation	Tests not performed on real-world system
Low cost over time	
Digital implementations easy to share	

Table 1.3: Benefits and penalties of using SDR-based tools for research

With technological advances the performance sacrifices will decrease and/or become financial sacrifices, but at the moment it is still not possible to fully conform to modern communication protocols.

Most commercial software-defined radios include a *field-programmable gate array* (FPGA). It is possible to achieve higher performance by moving code to the FPGA, but the disadvantage of this approach, which is outside the scope of the article, is that programming an FPGA requires additional resources and skills which will further restrict the userbase due to lack of skills and resources.

The results from an SDR-based research tool are not directly comparable to results from the equivalent hardware-implementation due to the performance sacrifice, however software-defined radios are an excellent starting point for the initial tests of a hypothesis before moving to hardware, and can also be used to gauge the relative effects of a change.

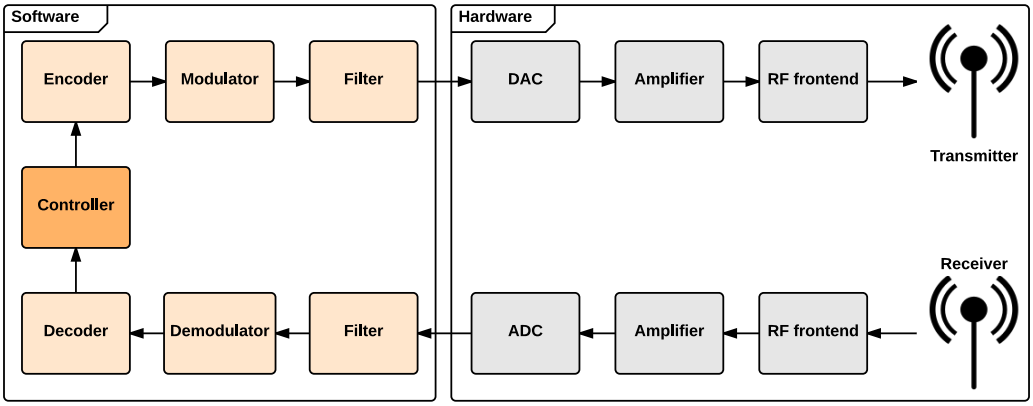


Figure 1.14: An example signal chain of an SDR. The controller block is responsible for the communication layers above the PHY layer.

1.3.1 USRP N210

The USRP N210 is a software-defined radio made for research and prototyping purposes. It is developed by Ettus Research and has an Xilinx Spartan 3A-DSP 3400 FPGA, which can be programmed with custom fabric. The FPGA has the potential to process up to 100 million samples per second in both the transmit and receive direction. The USRP N210 can stream up to 50 million samples per second to the host machine. The USRP N210 can be connected to the host machine thought the Gigabit Ethernet interface. The operation range of the USRP is 0Hz to 6GHz.[38]

The USRP N210 is flexible on the hardware for the transmit and receive chain, because it uses daughterboards for transmitting and receiving radio signals. This gives the user of the USRP N210 a lot of options regarding the hardware, such as choosing daughterboards with a large operating range, or a high transmission power.

The driver for the USRP N210 is the *Universal Hardware Driver* (UHD), which is the official hardware driver from Ettus Research[37]. This driver is also included in the OsmoSDR driver, which is a common software Application Programming Interface (API) for radio hardware from osmocomSDR[17]. The UHD makes it possible to access the USRP N210 from GNU Radio, through the *UHD Source* and *UHD Sink* blocks.

1.3.2 WISP 4.1

Based on the principles of software-defined radio, the WISP 4.1 devices are software-defined RFID tags. They are designed and built by researchers at the University of Washington, and were first introduced in the 2006 article "A Wirelessly-Powered Platform for Sensing and Computation." Unlike standard commercial tags, the WISP is programmable, which makes it very flexible for the use of different applications and ideal for research. The WISP tag has sensors on it, which enables it to sense the surrounding temperature and detect its own motion, however data sensing is not part of the EPC standard. The WISP is run by a Micro Control Unit (MCU), which has the program the WISP will execute, when it is powered up.

1.3.3 Micro Control Unit (MCU) of the WISP 4.1

The MCU on the WISP is the *Texas Instruments MSP430F2132*. It supports a very low level of voltage range from 1.8V to 3.6V and consumes very little energy. In *active mode* it uses 250μA at a clock frequency of 1MHz at 2.2V. In *standby mode* it uses 0.7μA. It supports a clock frequency up to 16MHz.[27]

The MSP430F2132 can run C/C++ programs, which can be flashed on to its memory. When it powers up it starts running the *main()* function inside the program. The MSP430F2132 has a lot of configurable components, such as a timer, a watchdog, general purpose I/O ports, RAM and much more. To configure these components, it is necessary to access the registers of the MSP430F2132. These registers have physical addresses, which you can write to and thereby for example start a timer inside the MSP430F2132, or set an I/O port to be an output port. The electrical components connected to the MSP430F2132 adds functionality to MSP430F2132 by the physical aspect they have, such as the antennas, that reacts to signal waves. The WISP 4.1 firmware is a C program, which can be flashed into the MSP430F2132 on the WISP 4.1DL tag. This program follows the EPC standard for communication from an RFID tag to an RFID interrogator (see appendix H).

1.4. GNU Radio

GNU Radio is a software toolkit and framework for the development of SDR applications. GNU Radio integrates with several drivers for SDR hardware, but the most important, and the de facto standard at the time of writing, is the UHD driver. The UHD driver was originally developed by Ettus Research for the USRP family of devices, but has since been adopted by other hardware manufacturers, and adapted by the community to integrate with even more SDR hardware. GNU Radio was originally built for PHY layer implementations, but new features in recent versions have improved the possibility of implementing MAC layer functionality as well. The PHY layer focus is still apparent in the software architecture of the framework, which is based on the *pipes and filters* architectural design pattern [9], illustrated in figure 1.15. This pattern is used to decompose

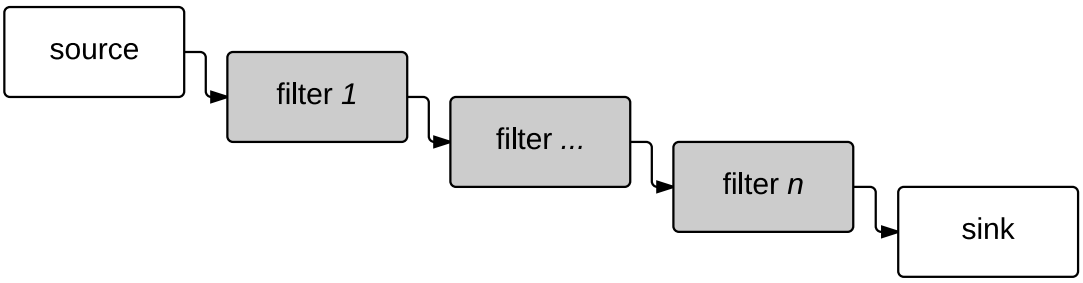


Figure 1.15: Illustration of the pipes and filters architectural design pattern.

the processing of a signal into multiple reusable components (filters) connected through well-defined interfaces (pipes). It is a very simple pattern that can be implemented in many ways. In GNU Radio the filters are referred to as blocks, which prevents ambiguity, since GNU Radio blocks can perform other tasks than just filtering. The GNU Radio

framework defines the pipe interface through the *Block class*, and enforce this interface by making all other blocks inherit from it.

1.4.1 GNU Radio blocks

For simplicity's sake, we define a block as a component that consists of a set of configuration parameters, an input and an output. Blocks without inputs are called source blocks, while blocks without output are called sink blocks. Sources and sinks are blocks that integrate with external I/O-interfaces like an SDR driver, the file-system or the graphical user interface (GUI). This concept is illustrated in figure 1.16.

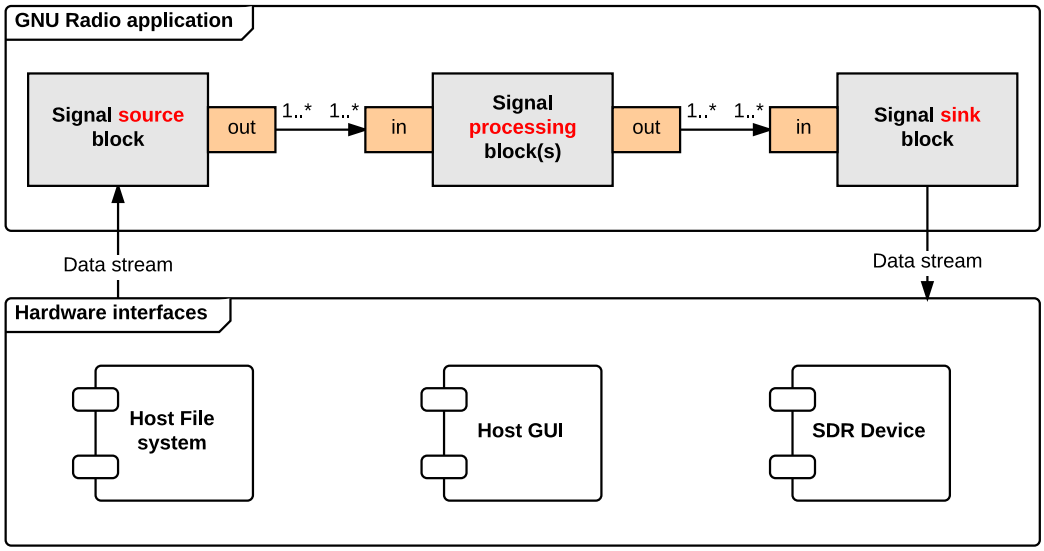


Figure 1.16: The basic structure of a GNU Radio application and how it connects to I/O interfaces.

1.4.2 Development environment

Swig The GNU Radio API is defined for both Python and C++, with Python being the recommended language for fast prototyping and C++ recommended for performance. GNU Radio supports mixing the two languages inside the same GNU Radio application through the use of a tool called Swig, which is used to define wrapper interfaces that can be called by the other language. The Swig project is being developed independently from GNU Radio.

GNU Radio Companion GNU Radio comes with a GUI tool called GNU Radio Companion (GRC). This tool allows users to connect and configure blocks into an SDR application using drag-and-drop as long as those blocks have their interfaces declared in an xml-file.

GNU Radio Modtool Another very useful tool for GNU Radio development is a command-line program by the name GNU Radio Modtool. It allows developers to generate the skeleton for a custom GNU Radio project, referred to as *out-of-tree modules*. This provides an easy way to get started with GNU Radio development by generating the

necessary block files, including Swig wrapper files, the main program file, initial, basic xml-declarations for GNU Radio Companion, and a makefile to build the project.

1.5. Determining state of the art

Through a literature review we have evaluated the different approaches towards SDR-based RFID research testbeds from 2007 to 2014. The complete list can be found in appendix A. We distinguish between custom platforms and commercially available platforms for both hardware and software. Some researchers implement their own custom hardware *and* software platforms, but most choose a combination of custom software built on commercial or open-source frameworks and commercial hardware. We note that after 2010 all of the reviewed research is based on the USRP hardware, which coincides with the year that the two researchers Buettner and Weatherall made their USRP-based *gen2reader* open-source.

We also note that some platforms seem to be better suited for MAC research based on their publicised use. Out of the 5 different non-USRP-based platforms only 1 has been used for MAC layer research, whereas in the case of USRP-based platforms the numbers are 16 out of 18.

On the software side, there is a clear preference for GNU Radio. After 2010 there are only 2 out of 14 platforms not based on GNU Radio, one of which is based on Matlab and the other based on custom C-code. Out of the platforms based on GNU Radio, the majority are based on the gen2reader in one form or another. Some researchers have adapted the code to be used with other USRP devices, and some have extended or improved the functionality of the code base.

We therefore conclude that the gen2reader platform is as close to what can be considered current state-of-the art as it gets. However, as we will document in section 1.7, the reason that the gen2reader software is so widely used, is not that it is a good research tool.

1.6. Software-defined Radio hardware review

This section contains a review of the 5 most viable SDR hardware platforms. The original list of 62 SDR devices [48], was reduced based on non-flexible requirements like frequency-range, lack of support for TX, driver-support, insufficient power output levels, etc. Just the frequency range requirement reduced the list to 13 devices. With the list reduced to 5 candidates it became apparent that these were almost the same devices on the list of hardware with GNU Radio drivers [35]. Table 1.4 compares the 5 candidates in terms of features. These devices were:

- HackRF One
- bladeRF
- UmTRX v2.2
- USRP1
- USRP N210 (Described in section 1.3.1)

HackRF One [34] HackRF One is an open source SDR hardware platform developed by Michael Ossmann. GNU Radio integration is handled through the open source gr-osmoSDR driver [17] developed for the OsmoSDR. The gr-osmoSDR driver incorporates libraries to support the HackRF One, as well as the bladeRF, USRP devices, the UmTRX devices and some of the devices that were eliminated above. Unlike the rest of the devices in this shortlist, the HackRF One does not use an FPGA, but rather a Complex Programmable Logic Device (CPLD). The transmission power varies depending on the used frequency:

- 10 MHz to 2150 MHz: 5 dBm to 15 dBm.
- 2150 MHz to 2750 MHz: 13 dBm to 15 dBm.
- 2750 MHz to 4000 MHz: 0 dBm to 5 dBm.
- 2750 MHz to 4000 MHz: 0 dBm to 5 dBm.
- 4000 MHz to 6000 MHz: -10 dBm to 0 dBm

bladeRF [32] The bladeRF is an open SDR hardware platform developed by Nuand. It comes in two versions, the x40 which is the basic version at 40000 Logic Elements (40KLE) and the x115 at 115000 Logic Elements (115KLE).

UmTRX [21] The UmTRX is an open SDR hardware platform developed by Fairwaves, designed to be able to work as a GSM basestation, but flexible enough to be used for other purposes.

USRP1 [39] The USRP is an SDR hardware platform developed by Ettus Research. It requires a daugtherboard to support transmission and reception. It supports up to 2 daugtherboards. The USRP Hardware Driver (UHD) [37] is the offical driver for the USRP1.

1.7. Analysis of gen2reader

In the previous section we established the gen2reader SDR software as the current state-of-the-art tool for RFID research. In this section we will evaluate the system to show the need for improvement. The following subsections take different perspectives on the gen2reader in order to explain the flaws that prevent it from fulfilling its role as a research tool.

The gen2reader is built on the GNU Radio framework and consists of 5 custom blocks (components):

- *command gate*: responsible for guarding the rest of the receive chain from processing irrelevant parts of the incoming signal.
- *center*: digitizes the signal and centers it around zero.
- *clock recovery*: converts pulses to bits based on a clock recovery mechanism known as zero-crossing.

Device	**HackRF One**	**bladeRF**	**UmTRX**	**USRP1** RFX900	**USRP N210** WBX 50-2200 MHz
Freq. range	10MHz to 6GHz	300MHz to 3.8GHz	300MHz to 3.8GHz	DC to 6GHz	DC to 6GHz
Max. power	5dBm (2150MHz) to 15dBm (10MHz)	6 dBm	16.98dBm (1.8GHz) to 20dBm (900MHz)	23 dBm	20 dBm
Price	€339	x40: €374 x115: €579	€1336	€672	€1632
Bus	USB 2.0	USB 3.0	gigabit Ethernet	USB 2.0	gigabit Ethernet
Hardware driver	grosmosdr	grosmosdr	grosmosdr	grosmosdr/UHD	grosmosdr/UHD
IC	CPLD: LPC4320FBD144	FPGA: Cyclone 4 (40KLE/115KLE)	FPGA: Xilinx Spartan-6	FPGA: Altera Cyclone	FPGA: Xilinx Spartan 3A-DSP 3400

Table 1.4: Summary comparison of the viable SDR devices

- *tag decoder*: decodes the received response.
- *reader*: constructs the interrogator commands, encodes the commands and transmits them.

For a detailed explanation of the gen2reader source code, refer to appendix D.

1.7.1 Documentation

There is no formal documentation of the gen2reader software. The available material consists of a few comments in the source code itself and the articles that have been written about it, which take a top-down approach without going into detail. The result is that most of the design decisions of the gen2reader software are not cited, nor supported by measurements. Examples of such decisions are the length of the CW before the Query command, the sampling rate used in the receive chain, using cross-correlation for tag response decoding, the SNR algorithm implementation, Q algorithm implementation, clock synchronization algorithm and many others. An example of the consequences of such decisions is the fact that tag responses are considered valid as long as the preamble and the final symbol were received correctly. Essentially, the software cannot be trusted in its current state as a research tool, since neither the measurements themselves nor the basis for their calculations have been scientifically validated.

1.7.2 Technology stack

The gen2reader was built in 2010 on a contemporary technology stack consisting of, at best, Ubuntu 10.04 and GNU Radio 3.3.X[2]. GNU Radio has matured greatly since then, both with added features, but also with changes to the API that make the gen2reader software incompatible with GNU Radio version 3.7.X and above. In order to use the gen2reader software on a current-day technology stack, it therefore needs to be upgraded.

1.7.3 Installation and setup

Closely related to the previous two areas, the gen2reader software comes without any documentation or instructions regarding installation and setup. This would be less of an issue if the technology stack was not so outdated, but it has been a big challenge to identify the correct combination of Ubuntu version, GNU Radio version, UHD version and package dependency versions that would allow the gen2reader to run. In addition to that, there are no instructions with neither the gen2reader nor GNU Radio for which development tools to use for out-of-tree modules. This will prove challenging for anyone who is not an experienced C++ developer and will get more challenging as time passes.

1.7.4 Useability

One of the features of GNU Radio is the GNU Radio Companion, which allows users to configure and connect blocks into a flow-graph. GNU Radio Companion can then generate a python program from the flow-graph, allowing users to work with GNU Radio programs without the need for writing, building and compiling the source code themselves. The

[2]Since there is no mention anywhere of the technology stack version numbers used, this is our educated guess based on the versions available in early 2010.

gen2reader software does not integrate with GNU Radio Companion, which would make the software easier to use and get to know.

1.7.5 The source code quality

Being the first to develop a functioning RFID reader based on the USRP1 and GNU Radio must have been enormously challenging and we have great respect for the dedication that Buettner and Wetherall [7] put into the project. However, results aside, there is no coming around that the quality of the code itself is very low. Quality in the context of source code is not directly tied to the quality of the software application, although the two often correlate. A program can have a badly written source code, but still do the job it was made to to. When we talk about code quality, we refer to concepts like complexity, readability and software design principles like cohesion, separation of concerns and loose coupling. These concepts and the violations of the gen2reader will be explained below.

1.7.5.1 Complexity and readability

The source code of the Command gate, the Clock Synchronizer and the Reader is unnecessarily hard to understand. There are several reasons for this. The first is the excessive scope of most of the variables used. If you look at the Reader member functions for example, most of them take zero arguments. This is because the variables that they use are class member variables or even defined in a global, cross-block header. This is not a problem for constants, but when a method assigns a new value to a variable, the use of global variables makes it very hard to trace that assignment to its source. Even more importantly, it creates an implicit relationship between the order in which methods must be called, such as the one between *gen_ query_ cmd()* and *send_ query()*. That relationship should be made explicit by having *send_ query()* accept the output of *gen_ query_ cmd()* as input. The second reason is the low level of abstraction that characterizes the code. The extract below can be found 23 different places in the Reader block and simply putting it in a method called add_msg_to_out_queue would improve the code tremendously.

```
gr_message_sptr cw_msg1 = gr_make_message(0,
        sizeof(float), 0,
                (usrp_pkt_size) * sizeof(float));
memcpy(cw_msg1->msg(), cw_buffer,
        (usrp_pkt_size) * sizeof(float));
out_q->insert_tail(cw_msg1);
```

1.7.5.2 Separation of concerns

The concept of separation of concerns refers to separating pieces of code with clearly differing responsibilities. Doing so disentangles the code, making it easier to read, and easier to make changes when requirements change for one area of responsibility. As an example, the reader block is responsible for not only the MAC layer state machine and the construction of the commands, but also the encoding and framing. These should be split into separate blocks, which would make it possible to experiment with different command encodings without modifying the Reader block. A consequence of having it all in a single block is that the commands are copied between 6 containers from initial creation until

being passed to the USRP, and out of these, 3 are global[3].

1.7.5.3 Cohesion

This refers to grouping parts of the code that have closely related responsibilities, together. The Miller decoding is an example of how this is broken in the gen2reader. It is split into three parts. The decoding itself is placed in the decoder block, the decoder configuration parameters are placed in the *global_vars.h* header, whereas the dynamic configuration of the decoder based on the configuration parameters happens in the Reader block.

1.7.5.4 Coupling

In essence, coupling refers to the degree of interdependence between otherwise distinct modules[4] of code. When a module needs to know more about another module than the interface it exposes, the two modules are tightly coupled. To take an example mentioned in another context above, the *send_query()* method is tightly coupled to the *gen_query_cmd()* method since it will break unless *gen_query_cmd()* has been called first. Another example mentioned above is accessing shared data, since changing the state of the shared data effectively changes the state of any module of code that acesses it. There is also a tight coupling in several blocks between the configuration and the block implementations. An example is the clock synchronizer, where the number of samples per pulse[5] is configured to 4, although this number is really dependent on both the specific sampling rate of the USRP source block, and the selected BLF. The same manual configuration can be found in the center block where the moving average window length is set to 128, but should be dependent on the same to values. Finally there is tight coupling between the block implementations and the specific BLF used, which means that not only must you change the configuration to change the BLF, you must also change the source code of the blocks to ensure that the receive chain functions properly. Coupling not only decreases the readability of the code, it also makes the code much harder to extend or modify, since any modification may have far-reaching and unforeseen consequences. One of the worst examples of coupling in the gen2reader is the *global_reader_state* object in the *global_vars.h* header, which tightly couples the state machines in the reader block, the command gate block and the decoder block.

1.7.6 The clock synchronizer

The clock synchronizer block uses a synchronization algorithm known as zero-crossing, which is implemented as an iterative method that tunes an approximation of the tag response cycle length and uses that to synchronize the input stream pulses to uniform lengths.

However, the tolerances specified in the standard are 22%, and the initial approximation is 4 samples per pulse, which gives a valid range of 3.12:4.88. The sampling rate has a resolution of 1, which means that we are working with integers and the only integer within the tolerances is 4 itself. Therefore algorithm 1 used in the clock synchronizer has no effect in the clock synchronizer. This means that the block is completely inflexible to

[3] q_bits, d_query_cmd, query_msg, out_q, tx_msg and out

[4] Components, classes, methods or any section of code clearly distinct from other sections

[5] A pulse is essentially an upsampled 1 or 0

Algorithm 1 The pulse width approximation update algorithm

1: **if** $measurement > approximation * (1 - tolerance) \&\& measurement < approximation * (1 + tolerance)$ **then**
2: $approximation = approximation * tuning_factor + measurement * (1 - tuning_factor)$
3: **end if**

clock skew. The other problem with the gen2reader implementation is that the center block also has a resolution of 1 and therefore is likely to skew the signal much more than the tag itself. By our measurements, the maximum single pulse measurement is 6 samples, and the maximum double pulse measurement is 10 samples. Since the gen2reader synchronizer converts measurements to single or double pulses based on the calculation below, it will potentially convert a single pulse to 2 pulses and a double pulse to 3 pulses.

```
int num_pulses = (int) floor((measurement / 4) + 0.5);
```

The main takeaway from this is that **the gen2reader clock synchronizer misinterprets clock skew as if it comes from the tag, when it is much more likely to come from the system itself**.

1.7.7 Assumptions

The gen2reader takes some liberties when it comes to interpreting channel conditions. As described in the previous section, tag response errors are likely to have been introduced by the gen2reader itself, but the system treats any error as a symptom of poor channel conditions, and goes so far as to assume that errors in a response with a correctly identified preamble, must have been the result of a collision between multiple tags. It also assumes that a slot without a correctly identified response preamble must be empty. These assumptions are not supported anywhere, but are still used to calculate an estimation of the tag population size.

Configuration The *global_vars.h* header is a mess of violations of the principles of cohesion, loose coupling and separation of concerns. It has three distinct responsibilities: message passing between blocks, global state and compile-time configuration. This violates the principle of separation of concerns. The compile-time configuration violates the principle of cohesion, since only some of it can be found in the header, with the rest spread out among the blocks. Many of the configuration parameters are based on the EPC standard and should provide a flexible interface to test different configurations. However, many of the configuration parameters in the standard are interconnected, i.e. coupled, and this coupling is not apparent from the code. Therefore, as it is now, if a researcher changes a parameter for a test, they may break the standard and not know it. An example is that the software downloaded directly from the repository is configured with an invalid RTcal symbol length. It is an error that is not immediately apparent, since the tags can recover from it, but it does affect the timing properties of the link and therefore may skew results.

1.7.8 Summary

To summarize the evaluation of the gen2reader software as a research tool, we have identified the following areas in need of improvement:

Trust The design decisions have not been documented, nor have the foundation for those decisions been cited. The limitations of the software, which affect the validity of the results, have not been documented. Any upgrade to the gen2reader should document the design, properly test and validate the whole system and disclose any flaws and limitations.

Modifiability and extendability The lack of documentation and the complexity of the software makes it very hard to understand the code well enough to make changes, and to foresee and understand the effect of those changes. Any upgrade to the gen2reader should follow good programming practices to provide a robust and easily modifiable research tool.

Validation of configuration parameters The system should validate the configuration parameters to ensure that configuration errors are corrected.

Configurability The block implementations should support the widest possible range of configurations within the EPC standard. In addition it should support dynamic reconfiguration on runtime.

Technology stack In order to fully take advantage of technological advancements, the technology stack should be updated to the newest available.

Ease of use as a research tool It takes a lot of time and effort to get the gen2reader software running on your system, and even more to get to a point where it is possible to modify or extend the software. This will prevent many from taking advantage of the many possibilities of an EPC standard interrogator.

Chapter 2

The MacReader

2.1. Introduction

The shortcomings of the gen2reader software identified in chapter 1 provide a starting point for the design of the upgraded software. Another way to find inspiration for an improved interrogator implementation is to analyse unused GNU Radio features. We have identified the following four features of interest:

- Volk (Vector Optimized Library of Kernels)
- Message passing
- Stream tagging
- Precise transceiver scheduling

Volk is a GNU Radio optimization tool for vector operations. It consists of two parts, one of which is a processor profiling tool. The profiler matches vector operations with the optimal SIMD (Single Instruction Multiple Data) instruction set supported by the processor and stores the results in a configuration file. The other part of volk is the library, which works like an abstraction layer over SIMD-optimized implementations of vector operations. When a GNU Radio developer uses a generic volk vector operation, the library ensures that the optimal SIMD implementation is used based on the configuration file from the profiler. In GNU Radio all inputs and outputs are essentially vectors, so there is rich opportunity to apply volk to the system code.

Message passing. GNU Radio features asynchronous message passing based on the publish-subscribe paradigm. It is the only built-in data exchange method that supports sending data upstream, which is a necessity when it comes to a feature like *dynamic reconfiguration*, where the PHY layer receive chain needs to be reconfigured from the MAC layer.

Stream tagging is another data exchange paradigm, based on the concept of metadata. Samples in the signal stream can be tagged with metadata and passed downstream[1] along

[1]Not to be confused with the downlink channel. In the interrogator receive chain the direction in which the data moves is referred to as downstream, true to the fluid analogy. In the communication between the

with the signal to the next blocks. Such a feature would be very useful for analysis of channel quality, allowing the receive chain to keep the MAC layer informed about SNR, timing, and other channel data.

Precise transceiver scheduling. The UHD driver uses stream tags for precise scheduling of transmissions, and also tags the rx signal with timing information. These features would be useful when implementing a standards-compliant RFID reader, to ensure that the link timing requirements are satisfied.

2.2. Design and improvements

Based on the findings decribed in section 1.7 and 2.1, we have designed the MacReader. The MacReader eliminates the shortcomings of the gen2reader, and makes substantial improvements to the usefulness of the system as a research tools.

2.2.1 Architecture overview

The MacReader consists of a two distinct layers, the MAC layer and the PHY layer. The interrogator commands originate at the MAC layer, and are passed *downstream* through the PHY layer *transmit chain* to the USRP sink. The tag responses are received at the USRP source and passed *downstream* through the PHY layer *receive chain* to the MAC layer. Figure 2.1 gives an overview of this architecture.

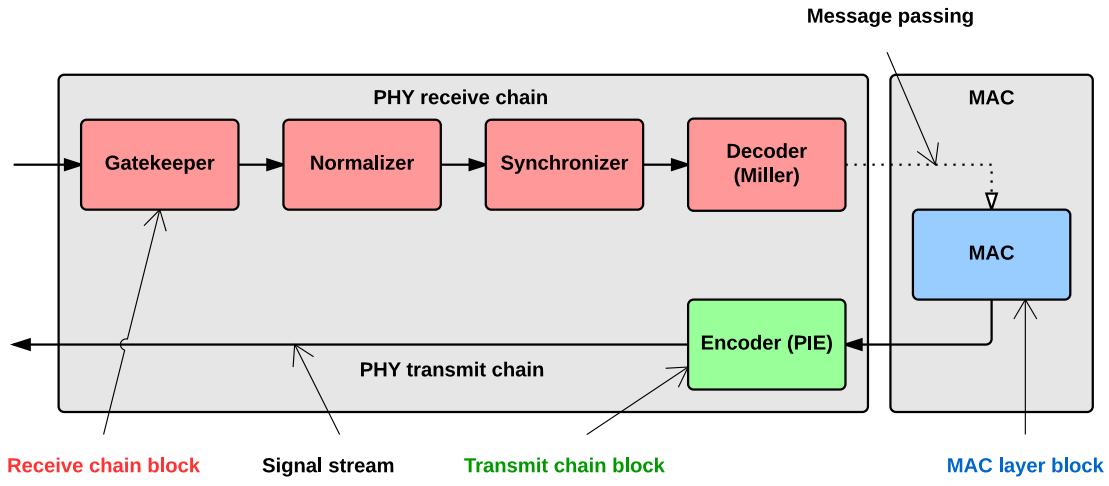


Figure 2.1: The distribution of the custom blocks across the PHY and MAC layer. The solid arrows illustrate the signal stream, whereas the dotted arrow illustrates message passing

interrogator and the tag, the interrogator→tag link is the downlink channel, whereas the tag→interrogator link is the uplink channel

2.2.2 Data exchange paradigms

The MacReader uses three different paradigms for inter-block data exchange. Table 2.1 gives a good overview of these paradigms and their uses in the MacReader. The signal samples themselves are transferred as a *stream*, through the GNU Radio stream buffers. The *channel metadata* is attached to relevant samples using *stream tags*. Both of these data exchange methods are downstream only, which is why asynchronous *message passing* is used for pushing *configuration parameters* from the MAC layer to the rest of the system. Figure 2.2 shows the various configuration parameters used and which blocks consume them. By only using built-in GNU Radio data exchange paradigms, it is possible to intercept data exchanges for debugging or logging without making changes in the system code. Examples of such interception methods are tag debugger blocks, file sink blocks and signal plotting sinks. Referring back to figure 2.1, we also use message passing between the decoder and the MAC block, although the only data passed from the decoder is signal and channel metadata. This is a compromise made necessary because the MAC block must be implemented as a GNU Radio *source block* to initiate the first transmission, and GNU Radio source blocks cannot have a stream input port.

Paradigm	Direction	Data
Streaming	Downstream	Signal
Stream tags	Downstream	Channel metadata
Message passing	Downstream/upstream	Configuration parameters

Table 2.1: The three GNU Radio data exchange paradigms and how they are used in the project

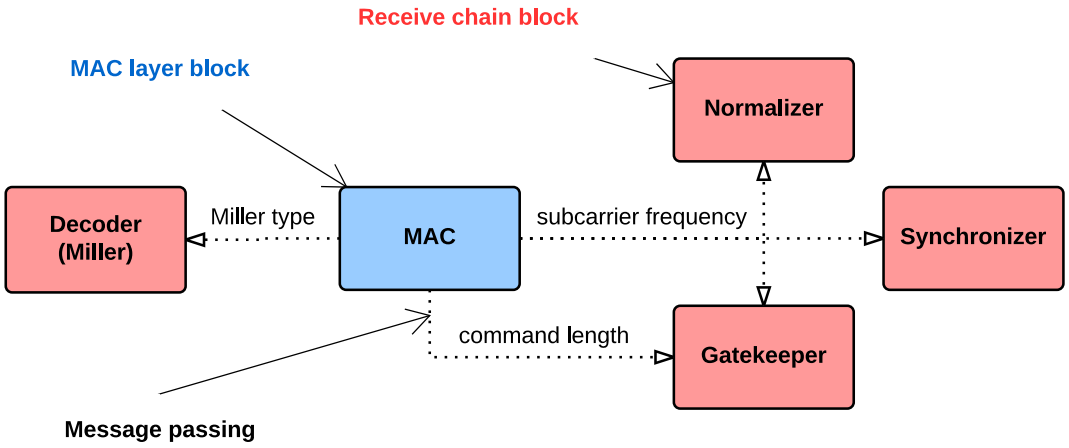


Figure 2.2: Overview of the configuration parameters each block receives through message passing

2.2.3 Static configuration

In order to reduce the complexity of the gen2reader's static configuration we have made the following improvements. First, we have added validation of parameters with interdependencies, such as those between the RTcal and the TRcal symbols. Next we made the configuration more dynamic by moving select parameters to the MAC layer and distributing configuration updates from the MAC layerto the various PHY blocks when

relevant. Third, we have applied the principle of cohesion and moved those parts of the configuration that did not belong, to the relevant blocks. Finally we have gathered the remaining static system configuration parameters in a single header, so it is no longer necessary to search for their location in the system source code. The result is a single source for static configuration, and a flexible framework for dynamic reconfiguration.

2.2.4 The MAC layer

The MAC layer is implemented in a single block with three responsibilities.

1. The primary responsibility is to maintain and update the interrogator state. The state machine implementation is decoupled from the rest of the MAC layer, to let researchers easily exchange it for other state machine implementations when experimenting. The default state machine, as illustrated in figure 2.3, is a simplified inventory round. We have also implemented a state machine which simply loops over the query command.

2. The second responsibility of the MAC layer is to construct the initial bit-representations of the interrogator commands based on the static configuration. Several fields in the query command depend on the initial static configuration and any later dynamic changes to it.

3. Finally, the last responsibility of the MAC layer is to push the relevant configuration parameters to the rest of the system, both for initial configuration and for dynamic reconfiguration. The specific command parameters that the MAC block pushes can be seen by revisiting figure 2.2.

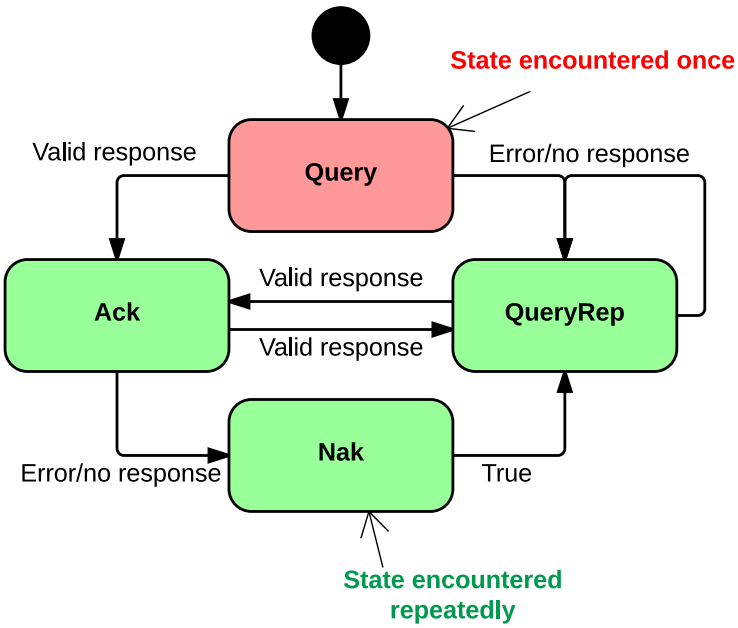


Figure 2.3: C1G2 standard *inventory round* as implemented in the MAC block. The state diagram omits any reconfiguration that may happen between states.

2.2.5 The PHY layer

The PHY layer consists of both custom blocks and standard GNU Radio blocks configured for use in the system. Without going into detail, the two most important standard blocks are the USRP sink and source blocks, which are built on the UHD driver and function as the link between the application software and the USRP hardware. In the receive chain the signal is passed through a decimating FIR filter before it is converted from a complex signal to real magnitude. In the transmit chain, the signal is converted from float to a complex signal for the USRP sink. Figure 2.4 provides an overview of the standard GNU Radio blocks used and where they fit in.

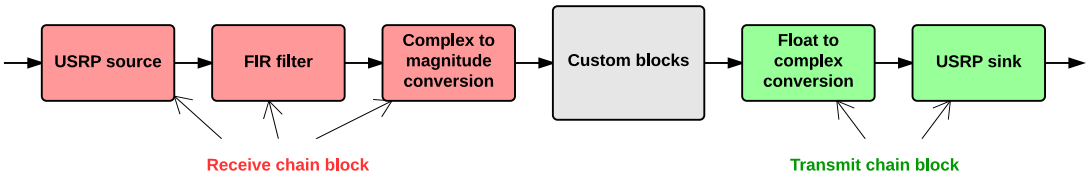


Figure 2.4: The standard GNU Radio blocks used in the system.

2.2.5.1 The Encoder

The *encoder* uses pulse-interval encoding (PIE) to encode the command, before it prepends a header. The encoder is also responsible for the transmission of the CW. The most important improvement of the encoder has been the reduction of the CW length from more than 1000 mikrosecond, down to a single mikrosecond. The inspiration for this came from the observation that when transmitting a burst, the received signal did not abruptly fall at the end of the burst. Instead it slowly declined over an extended period of time.

We then found the same behaviour even when continuously transmitting, as can be seen in figure 2.5. Based on these two observations, we decided to test whether it was possible to reduce the length of the CW without negatively affecting the communication. The results show that the length of the transmitted CW makes no difference as long as it is larger than zero and shorter than 10 milliseconds. To understand the significance of this finding, it is necessary to understand that the application is time-decoupled from the physical world through a set of buffers, where the USRP stores the signal until the application can consume it. When a GNU Radio application transmits a signal, the receive chain is not scheduled until after the transmission has concluded, and therefore it is optimal to finish transmitting as soon as possible. By shortening the CW, the receive chain can start consuming the input stream before the tag has responded, and thereby process the response while it is being received, instead of waiting until after. These are unique circumstances that only affect systems like an RFID interrogator, where an interrogator must transmit a CW and receive the backscattered reflection of that CW at the same time.

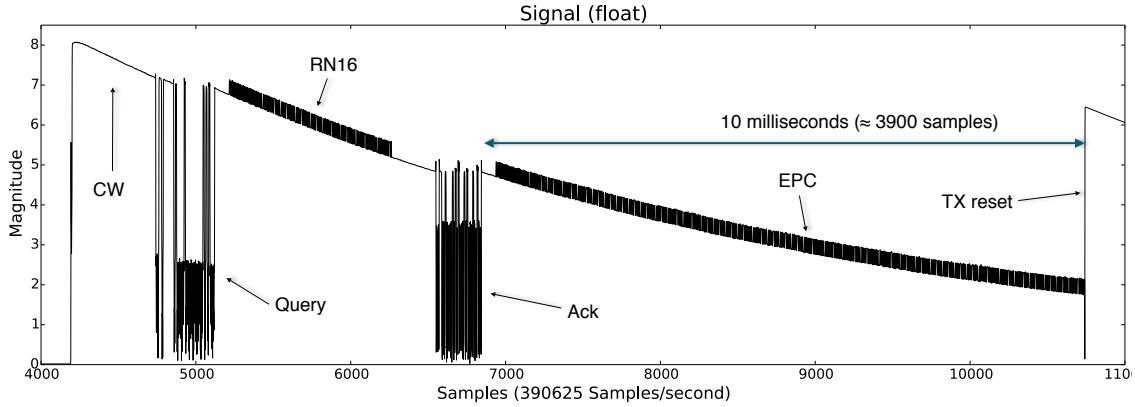
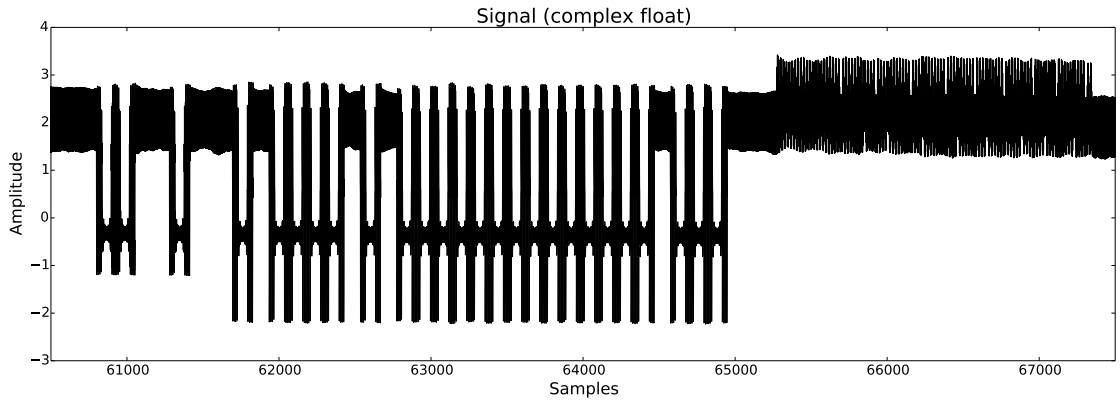
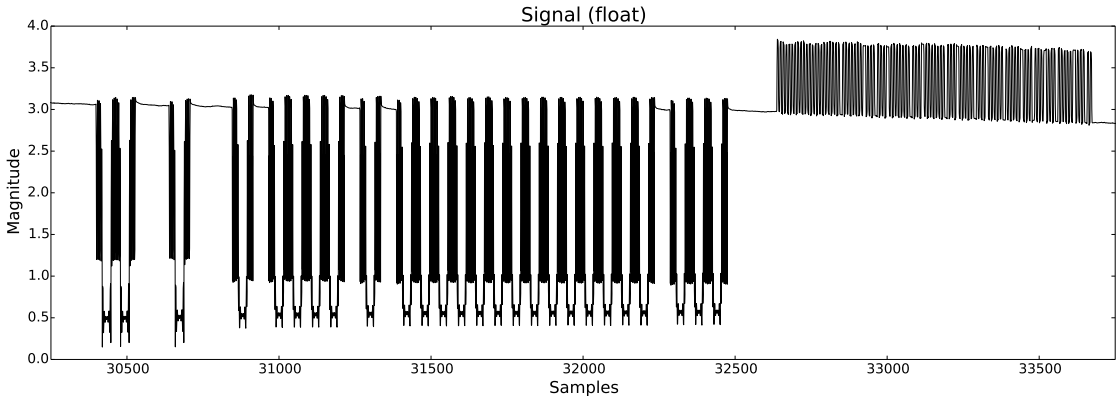


Figure 2.5: This plot shows the steady decrease over time in signal magnitude. The signal shown consists of a query command followed by an RN16 response, the interrogator ACK command and finally the tag EPC response. The plot also illustrates that 10 milliseconds after the end of the last active interrogator transmission, the signal resets to near maximum magnitude.



(a) Complex signal

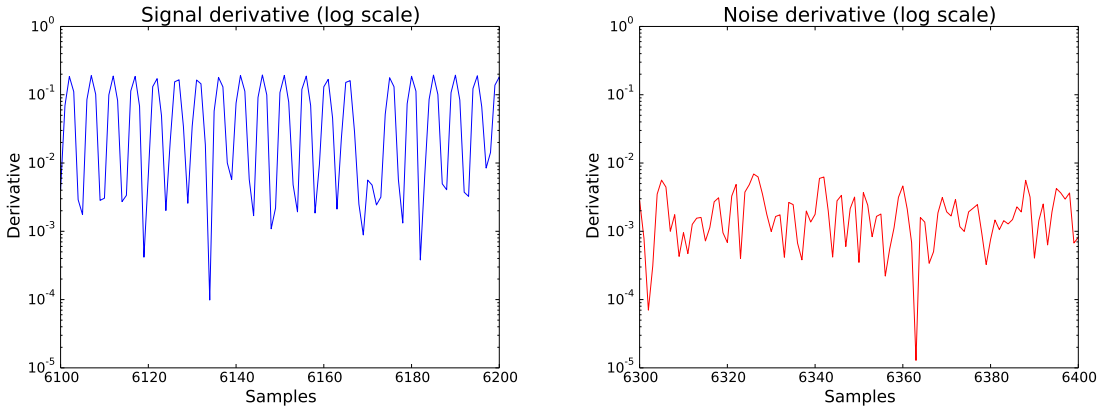


(b) Magnitude signal

Figure 2.6: The signals both consist of an interrogator query followed by a tag RN16. The plots illustrate that edge detection is easier when working with magnitude.

2.2.5.2 The Gatekeeper

The first custom block in the receive chain is the *gatekeeper*. This block analyses the incoming signal in order to only allow the tag response through to the downstream blocks. Thereby the gatekeeper limits the amount of signal processing and the scheduler overhead related to managing the signal stream. The gatekeeper is built on a state machine that uses edge detection to identify the first rising edge and the last falling edge of the tag responses. The block also calculates relevant channel metadata such as link timing, SNR and response length, which is then attached to the signal stream for processing at the MAC layer. We have made an important change by converting the signal from complex to magnitude before it reaches the gatekeeper rather than doing it after as in the gen2reader.



(a) Signal derivative

(b) Noise derivative

Figure 2.7: The signal and noise derivatives plotted on a log-scale, showing the clear difference between the sharpness of the signal edges and the sharpness of the noise edges.

Figure 2.6 shows the difference between the input signal to the old gate and the input signal to the new gate. Converting to magnitude before the Gatekeeper makes it easier to work with the signal, and has allowed us to use edge-detection to determine the exact link timing values of the system. In order to improve the accuracy of the edge-detection, we have performed tests to profile the noise edges and the signal edges. Figure 2.7 illustrates the edge sharpness through the derivatives of the signal and the noise. As the figure shows, the noise edges are nearly an order of magnitude smaller than the signal edges, and by taking advantage of this the system recognizes response edges with an accuracy of 100%. Figure 2.8 illustrates the gatekeeper state machine and shows how it recognizes not only tag responses, but also interrogator commands. The *Wait for reset* state prevents the gatekeeper from continuously consuming input, which would otherwise delay the transmission of the next command.

2.2.5.3 The Normalizer

The *normalizer* prepares the signal for the *synchronizer* by converting the analog signal to a digital signal centered around zero. It uses algorithm 2 for the conversion.

We have made two important improvements to the normalizer. First, we have replaced the manual calculation of the mean by the equivalent *volk* method. This improves the mean calculation processing time by 75% [2] at the subset size of 128 samples that is used

[2] The code used for mean calculation was extracted from the rest of the code for the tests

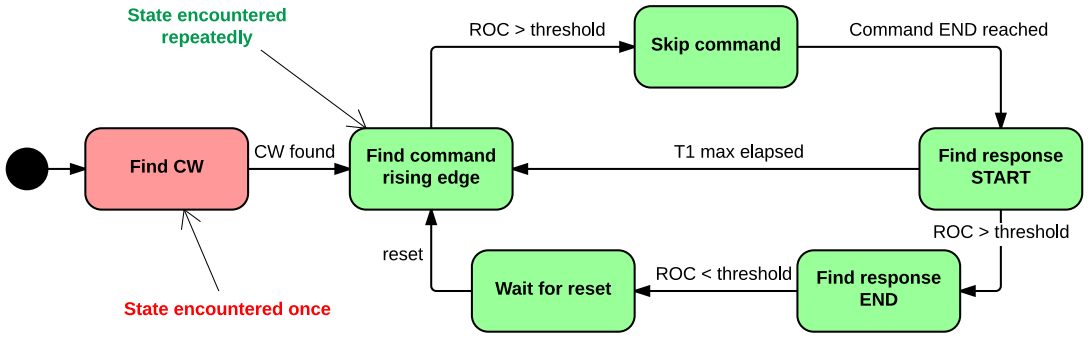


Figure 2.8: The states and transitions of the gatekeeper state machine. The red *Find CW* state is only passed once, after which the state machine will switch exclusively between the green states.

Algorithm 2 The normalizer algorithm

```
for all inputsamples do
    if inputsample > mean then
        outputsample ← 1
    else
        outputsample ← 0
    end if
end for
```

in the gen2reader. The results for other subset sizes can be seen in figure 2.9, which plots a performance comparison between the volk algorithm and the gen2reader algorithm with respect to subset size.

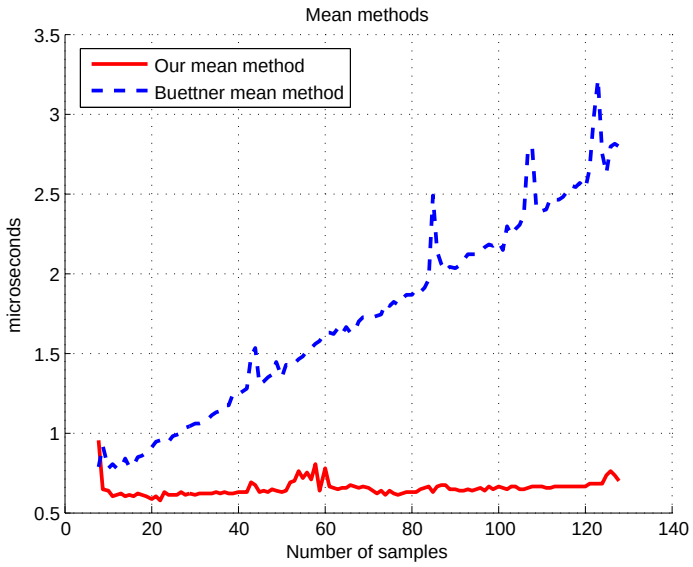


Figure 2.9: Comparison of the mean algorithm processing time using volk versus not using volk.

In addition to that, we have calulcated that the optimal subset size is much lower than the 128 samples used in the gen2reader. To find the optimal subset size, there are two

factors that need to be balanced: processing load and noise robustness. First of all, a larger subset size leads to higher processing load, but as figure 2.9 shows, reducing the subset size has little impact on the processing time when using Volk. With regards to noise robustness, we have found that at low Signal-to-Noise Ratio (SNR), the noise may affect the mean more than the signal, but that reducing the size of the observed window minimizes that effect. Figure 2.10 illustrates this finding. Based on the above we want to

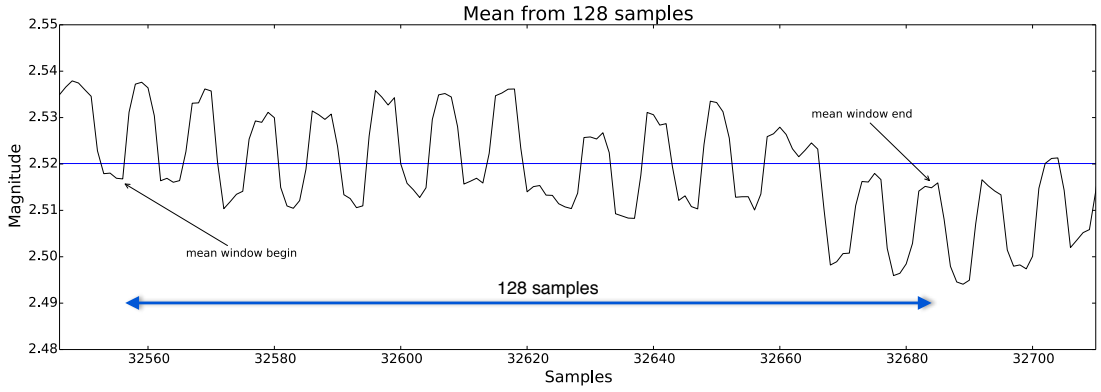


Figure 2.10: The plot shows how a high pulse may be interpreted as a low pulse by the normalizer, because the mean is compared to the sample at the end of the mean window and because the window is so long.

minimize the mean subset size, but in order to find the minimum acceptable size, we need to look at the Miller encoding. That is because Miller symbols are symmetric, meaning that they consist of an equal number of high and low pulses, which makes the nominal mean equal to $1/2$.

Figure 2.11 illustrates how the mean at different subset sizes deviates from the nominal mean, supported by the data in table 2.2.

Pulses	max(mean)	max(deviation from mean)
1	1	1/2
2	1	1/2
3	2/3	1/6
4	3/4	1/4
5	3/5	1/10
6	4/6	1/6
7	4/7	1/14

Table 2.2: The worst case mean and deviation from mean, calculated for subset sizes ranging from 1 to 7 pulses. See figure 2.11 for visual support

From the calculations in table 2.2 we found that the maximum deviation from the nominal mean can be calculated as

$$\text{deviation}_{max}(n) = \begin{cases} 1/n, & \text{if n is even.} \\ 1/(2 \cdot n), & \text{if n is non-even.} \end{cases} \tag{2.1}$$

where n is the subset size in pulses. We also calculated the resolution of the receive chain to be $1/5$ samples per pulse, which means that **any accuracy better than 1/5 is lost**

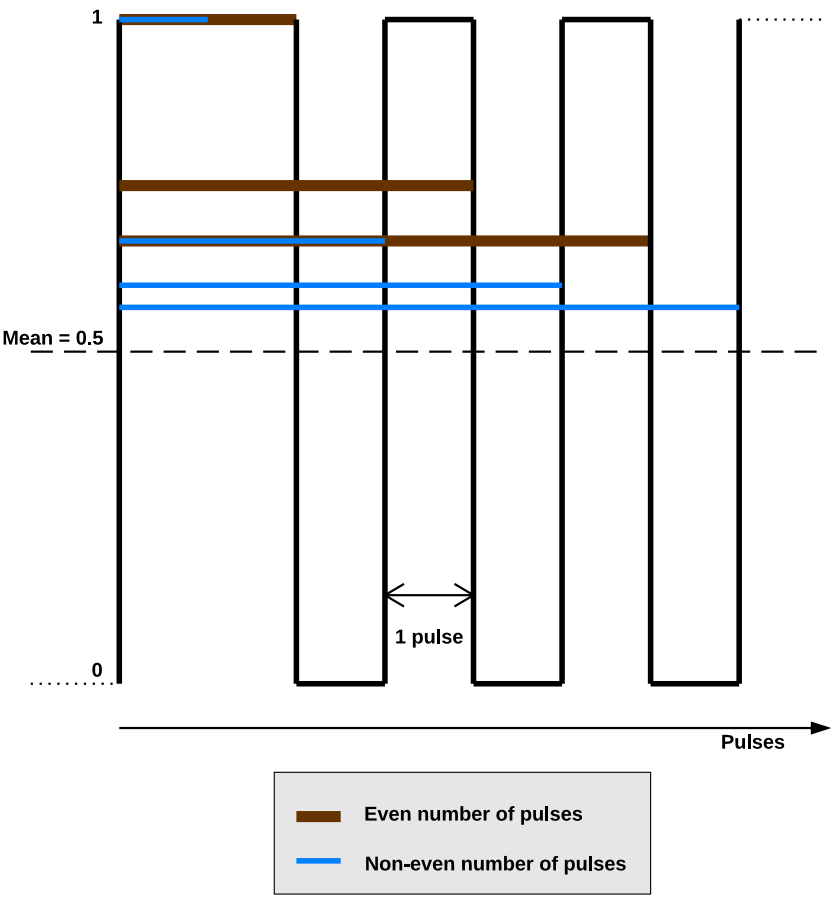


Figure 2.11: The figure shows an extract of 7 pulses from a Miller symbol sequence. The colored lines illustrate the mean calculated at different subset sizes, from 1 pulse to 7 pulses and how much they deviate from the nominal mean at 0.5. Note how an uneven number of pulses gives a much smaller deviation compared to an even number of pulses

to the sampling resolution. Given this information, the optimal subset size can be calculated as

$$1/(2 \cdot n) \leq 1/5 \Rightarrow n \leq 2/5 \tag{2.2}$$

We round that up to the smallest non-even integer which gives a subset size of 3 pulses or 3/2 subcarrier cycles. To compare that with the subset size of 128 samples used in the gen2reader, we convert from cycles to samples for the configured sampling rate and BLF:

$$2000000 \text{ samples/s} \cdot \frac{\frac{3}{2} \text{ cycles}}{256000 \text{ cycles/s}} \approx 11.7 \text{ samples} \tag{2.3}$$

To reiterate, **the gen2reader uses a subset size of 128 samples to calculate the sample offset relative to the signal mean, but according to our calculations that is far more than necessary, and also more fragile towards noise at low SNR levels**.

2.2.5.4 The Synchronizer

The synchronizer decimates and synchronizes the tag response signal to 2 samples per subcarrier cycle. Although we have made improvements to the normalizer block, the sampling rate still dictates a sampling resolution that may lead the normalizer to add clock skew while digitizing the signal. To overcome this challenge at the synchronizer we found inspiration from the way the tags synchronize the interrogator command signals. They measure the length of the RTcal symbol in the command preamble, which informs the tag of the combined length of a data 1 and a data 0 symbol. The tag then calculates a pivot, which lies right in the middle and simply converts any symbol that is longer than the pivot to a data 1 and shorter than the pivot to a data 0. We calculate a similar pivot based on the expected BLF. From the BLF we calculate the length of a single subcarrier cycle, and combine that with the knowledge that Miller encoding is constructed from long and short pulses with lengths of respectively 1 cycle and 1/2 cycle. The pivot length can then be calculated as

$$\text{pivot} = \frac{1 \text{ cycle} + \frac{1}{2} \text{ cycle}}{2} = \frac{3}{4} \text{ cycle} \tag{2.4}$$

Our method of clock synchronization is more robust than the gen2reader's zero-crossing implementation, because it neither defines a maximum allowed symbol length nor a minimum, making the tolerances of our method more forgiving to clock skew.

2.2.5.5 The Miller decoder

The final block in the receive chain is the Miller decoder. Like the decoder in the gen2reader, it uses cross-correlation to recognize Miller data symbols and decode them. A Miller encoded tag response consists of a preamble, a payload and a single Miller-encoded 1, referred to as a dummy-bit. When the decoding has finished, the block extracts all channel metadata from the stream, packs it all with the payload and publishes it in a message to the MAC block.

2.2.6 About timing and robustness

In the original vision for the system, we wanted to use precise scheduling of interrogator transmissions to comply with the link timing requirements of the standard. However, tests showed that the the USRP has hard timing requirements and therefore drop late packets without notice. Due to the relatively high latency of SDR's, we found all of our packets being dropped. We realised that this was expected behaviour for any standards-compliant interrogator, but that the purpose of a research system is not to be standards-compliant. Instead, the purpose of a research system is to collect the best data possible. This proved to be a major paradigm shift in the interrogator design, which lead to another realization, that the gen2reader receive chain does not distinguish between tag response failures that come from the physical channel and failures that come from the interrogator implementation itself. Analysis and tests of the code showed that there are many points where signal processing in a block introduces errors. Examples of this have been covered above, but here we reiterate some of them:

- A sampling rate that gives a resolution of $\approx 1/5$ subcarrier pulse
- A normalizer that introduces clock skew

- A synchronizer that pushes ghost bits to the decoder
- A decoder that fails if the preamble has more than 2 bit-errors

The new interrogator implementation has been designed to avoid these weaknesses. The relative improvements are inversely proportional to the SNR or in other words, the improvements become bigger as the signal becomes worse.

2.2.7 Summary of improvements

This summary will tie the improvements we have made to the system together with the weaknesses of the gen2reader which we identified in 1.7 .

Trust We have documented the complete system, and supported the design decisions made with references and measurements.

Modifiability and extendability We have reduced the complexity of the code-base drastically. We have also decoupled the blocks through the use of the various GNU Radio data exchange paradigms.

Validation of configuration parameters The system now validates the compile-time configuration parameters against each other and the specifications of the EPC standard.

Configurability The MAC layer now implements dynamic reconfiguration through the use of channel metadata and message passing. In addition, block configuration parameters have been simplified and made dynamic, so that static configuration happens on an application basis, not a per-block basis.

Data collection We have added link timing calculations, proper SNR calculations, and an extendable framework for feeding the MAC layer with channel metadata.

Technology stack The technology stack has been updated to Ubuntu 14.04 LTS and GNU Radio 3.7.5.

Useability We have integrated the blocks with GNU Radio Companion, allowing researchers to work with the finished blocks in a graphical user interface. Additionally we have written a pyplot script that plots output from GNU Radio file sink blocks. With this tool, researchers can zoom and scroll through the complete signal, and generate PDF files from their plots. The pyplot script is extremely useful for debugging and for generating scalable figures for latex. See appendix N.

Setup and installation With the update to GNU Radio 3.7+, it is possible to perform the complete installation of GNU Radio, UHD and dependencies through the use of a single script, that also has the option to specify the exact version needed. See appendix C.

Development platform We have documented how to import a complete GNU Radio out-of-tree module into Eclipse. This gives researchers all the power that comes with the Eclipse IDE, most important of which is the graphical debugger. See appendix I.

2.3. Limitations

Latency The system latency lies in the range from 300 microseconds and up. The interrogator→tag link timing limit *T2* is 20/BLF seconds, which gives a range of 31.25 up to 500 microseconds for the frequency range 640 kHz down to 40kHz. Due to high latency variation, even at a T2 limit of 500 microseconds, there is a high probability of exceeding that limit. When link timing is not in the scope for a test it is therefore necessary to filter out exchanges where the T2 limit is exceeded in order to get relevant results. This is made easy by the MacReader, which accurately measures the exact link timing values for both tag→interrogator and interrogator→tag communication exchanges.

Sample resolution The transmit chain has a sampling rate of 1e6 samples per second, translating to a sampling resolution of 1 microsecond. The EPC standard specifies timing values at a resolution up to 1/100 microseconds. The most relevant example is the TRcal symbol length for 256 kHz, which is specified to 83.3 microseconds. In the MacReader that is rounded to 83 microseconds, making the calculation of the BLF:

$$\frac{divide\ ratio}{TRcal} = BLF \Rightarrow \frac{\frac{64}{3}}{83} = 257028Hz \tag{2.5}$$

Due to the tolerant design of the receive chain this should have no practical effect, but it is something to be aware of. The MacReader is designed for a $(sampling\ rate)/BLF$ ratio of approximately 20 at the USRP source, and 10 after decimation at the FIR filter. The specific frequency is ruled by a requirement from the USRP that $100MHz/(sampling\ rate)$ is an even number. This leads to a receive chain resolution of approximately 5 samples per subcarrier cycle[3], which, as described previously, has implications with regards to the accuracy of the normalizer and the synchronizer. Again, the MacReader is designed to be tolerant to these inaccuracies, but users should understand this aspect.

Functionality removed from the gen2reader The gen2reader had a few features that have been removed from the MacReader. One of those features is the tag estimation algorithm that was based on unsupported assumptions about the relationship between response errors and slot state. This algorithm can easily be reintroduced, and with the current ability to consistently and accurately detect responses it should now be possible to trust the output. We have also removed the Read and Req_RN commands from the system since Buettner never finished implementing them and they were outside the scope of this work. Due to scope we also removed the *interrogator command decoder* which was used to decode commands from commercial interrogators. This should also be easy to port at a later point, although we recommend to switch from the mean-based detection of the current version to derivative-based detection.

2.4. Future work

Excess input samples Initial observations hint that a considerable part of both the application latency and latency variation comes from excess input samples to the gatekeeper.

[3] At the USRP sampling rates used for the MacReader, 781250 for 40 kHz BLF and 5e6 for 256 kHz BLF, the receive chain resolution is ≈ 4.88 samples per subcarrier cycle

Our tests show that the average excess is ≈ 134 samples, which translates to between 53.6 and 343 microseconds at the sampling rates used in the MacReader. Therefore it would provide valuable information towards optimizing the application, to map the relationship between the performance, the sampling rate and the input buffer size requirements of the gatekeeper. Our experience shows that increasing the sampling rate will reduce the effect of excess input samles, but at the same time increase the load on the receive chain. Additionally, reducing the input buffer size requirements should reduce the average number of excess input samples, but increase the GNU Radio scheduler overhead. Besides providing data for the optimization of the application, it is also a source of latency and latency variation that to the best of our knowledge has not been covered in the literature.

Host system latency The original deployment instructions for the gen2reader include executing the application with the Linux *nice* command. This is a command that sets the processor priority of the application, however on our system it made no observable difference. On the other hand, another Linux command, the *ionice* command, which sets the I/O-priority of the application shows a noticeable difference, not unexpectedly more so when file-sink blocks are connected in the flow-graph. Finally, hardware advances since the time of the gen2reader have meant that multicore processors are much more common. Indeed, the host for the MacReader has 8 cores. The *chrt* command is used to delegate processor cores to a single application, which could potentially reduce the *context switching overhead* and thereby the latency and latency variation on a system. To the best of our knowledge noone has tested the effect of the *ionice* and *chrt* commands for an SDR system, and there is a need to test whether the *nice* command has an effect on the MacReader.

Abandoning blocks to avoid the GNU Radio scheduler overhead Although GNU Radio provides a comprehensive and structured framework for SDR applications, these things come with the tradeoff of scheduler overhead. There is no reason why the application could not bypass GNU Radio blocks and be designed as a regular component-based C++ application integrating directly with the UHD API. It may sound complicated, but it is actually rather easy to cannibalize GNU Radio block internals into regular C++ classes, and the whole streaming concept is not really necessary once passing the gatekeeper. After that, the tag responses could be treated as single chunks of data. This would eliminate a big source of latency, but at the expense of making the project less approachable. As proof of concept we have implemented a functioning application where all the blocks except the MAC layer are converted to regular C++ classes and initialized and used directly from the MAC block.

Making a proper Miller decoder The Miller decoding strategy used by both the gen2Reader and the MacReader is decoding through cross-correlation. However, as covered in appendix K, Miller encoding has memory, which means that the previous state determines the set of valid next-states. Therefore, by implementing the decoder as a state machine, it is possible to build-in a certain level of error-detection and probabilistic error-correction. It would be interesting to test such an implementation.

Using the characteristics of the signal derivative for synchronization Late in the process we realized that the signal derivative provides a better foundation to digitize

the analog signal than the mean. This is because the mean is sensitive to noise as documented in section 2.2.5.3. The signal derivative however identifies the sharpest edges of the signal regardless of noise. This is important because the sharpest edges are the most accurate delimiters of the signal pulses and therefore this method should improve the whole downstream receive chain.

Using a monostatic antenna Both the gen2reader and the MacReader are based on bistatic antennas, but with the industry moving more and more towards monostatic antennas, it is relevant to research the challenges and possibilities of implementing a monostatic SDR-based RFID interrogator.

Chapter 3

Measurements and results

This chapter contains the experiements we have performed during this thesis. They are grouped into 5 categories:

1. Algorithmic tests
2. GNU Radio framework tests
3. Protocol tests
4. Hardware tests
5. Analysis of gen2reader attributes

The chapter starts by providing the information needed in order to reproduce our results, and then goes through each category and test, explaining the background and results.

3.1. Measurement setup

3.1.1 Hardware

We have performed tests on several distinct systems, which required very different equipment and tools and therefore this section will describe the equipment setup for each system scenario separately. These scenarios are:

1. gen2eader: The gen2reader software is from 2010 and cannot run on the newest GNU Radio framework. Therefore it is necessary to use a separate technology stack for tests.
2. MacReader: The MacReader is based on the newest available software from 2014.
3. Impinj Speedway R220: This commercial interrogator was used when we needed to compare functionality or confirm correct operation of the MacReader and the WISP.
4. WISP tag: The reprogrammable WISP tag has been used to confirm correct operation of the MacReader, through run-time debugging of the tag firmware.

5. Algorithms: Some parts of the interrogator source code is framework ambivalent, so these have been tested separately.
6. Shield: Testing of the shielding methods between the transmitter and receiver antennas on the USRP radio.
7. Output Power: Testing the output power from the daughterboard.

We used the following hardware for the measurements:

- Software-defined Radio (interrogator): USRP N210
- USRP Daughterboard: WBX 50-2200 MHz RX/TX
- Antennas (interrogator): Two monopole antennas tuned to 1/4 wavelength of 867.5 MHz, soldered to an SMA connector.
- Host machine A (Scenarios: *gen2reader*, *MacReader*):
  - Manufacturer: FUJITSU
  - Model: Esprimo P920 E85+ - i7
  - Processor: Intel RCoreTMi7-4790 CPU @ 3.60GHz
  - Memory: 16 GB DDR3 SDRAM
  - Storage: 256 GB SSD
  - Ethernet: Gigabit
- Host machine B (Scenarios: *Impinj Speedway R220*, *WISP tag*, *Algorithms*, *Shield* and *Output Power*):
  - Manufacturer: FUJITSU
  - Model: E751
  - Processor: Intel RCoreTMi5-2520M CPU @ 2.50GHz
  - Memory: 6 GB DDR3 SDRAM
  - Storage: 256 GB
  - Ethernet: Gigabit
- Commercial interrogator: Impinj Speedway R220
- Commercial antenna: Caen RFID WANTENNAX005
- Tags:
  - AD-827 (Manufacturer: Avery Dennison)
  - ALN-9629 Square (Manufacturer: Alien)
  - EURUHFT(4931/5202)(Manufacturer: Eureka)
  - WISP 4.1
- MSP430 USB-Debug-Interface MSP-FET430UIF
- WISP 4.1 Programmer(An adapter board)

3.1.2 Software

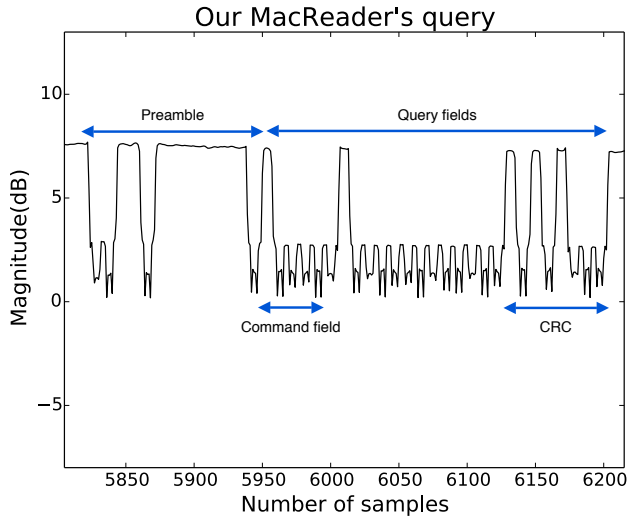
We used the following software for the measurements:

- MacReader:
  - Ubuntu 14.04
  - GNU Radio 3.7.5
  - UHD 3.8.0
  - Eclipse Mars 4.5.0 M3(The C++ compiler must be configured to use 2011 C++ syntax)
- gen2reader:
  - Ubuntu 12.04
  - GNU Radio 3.6.5
  - UHD 3.5.4
  - Eclipse Mars 4.5.0 M3
- Impinj Speedway R220:
  - Windows 7
  - MultiReader for Speedway Gen2 RFID 6.6.10.240
  - Visual studio 2010
- WISP 4.1:
  - Windows 7
  - IAR Embedded Workbench IDE 5.40.3

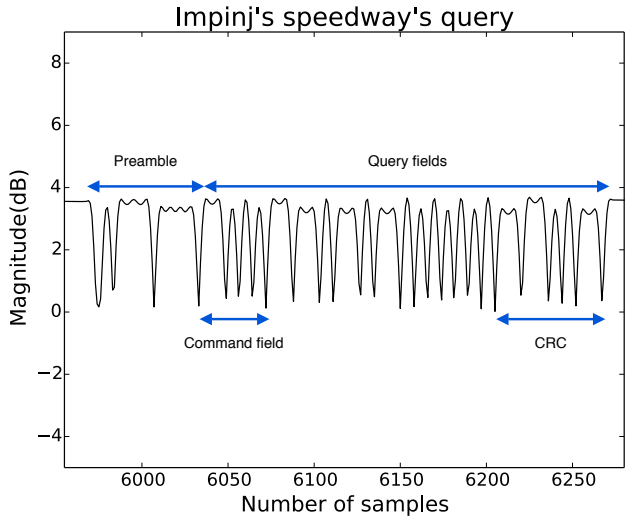
3.2. Verifying interrogator query command

Because a query command is so important for the communication between an interrogator and an RFID tag, we first needed to verify that the structure of the query command sent from the MacReader was correct. We tested the structure of our query command by transmitting a query with the MacReader and using a *file sink* to capture the transmitted signal. We repeated the test with the Impinj Speedway R220 interrogator and plotted both the signals with *pyplot*, to compare them.

Figure 3.1 shows the structure of the two queries. It is possible to see that both queries use PIE encoding (See section 1.2.5). The preambles are different because the two interrogators use different RTcal and TRcal lengths, which corresponds to different data symbol lengths and a different BLF. Both queries have the mandatory command field *1 0 0 0* (See figure 1.3), but use different configurations, which is why the commands are not identical. The reason that the MacReader data-0 pulses seem to have lower magnitude relative to the data-1 pulses is simply because of the low sampling rate.



(a) MacReader query



(b) Impinj Speedway R220 query

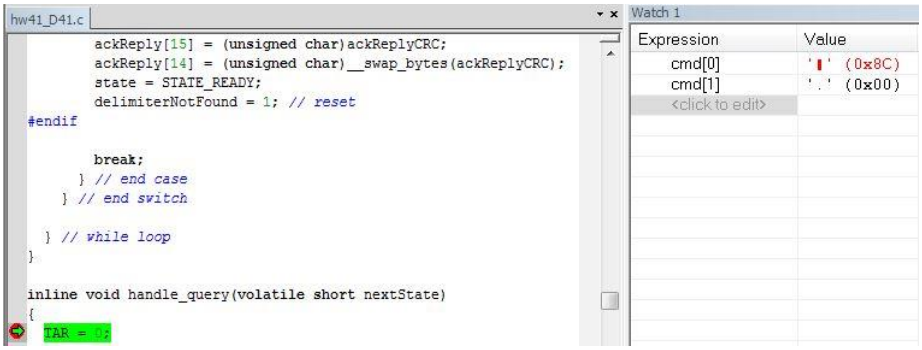
Figure 3.1: Queries sent from MacReader and the Impinj Speedway R220

3.3. Verifying query command received by WISP

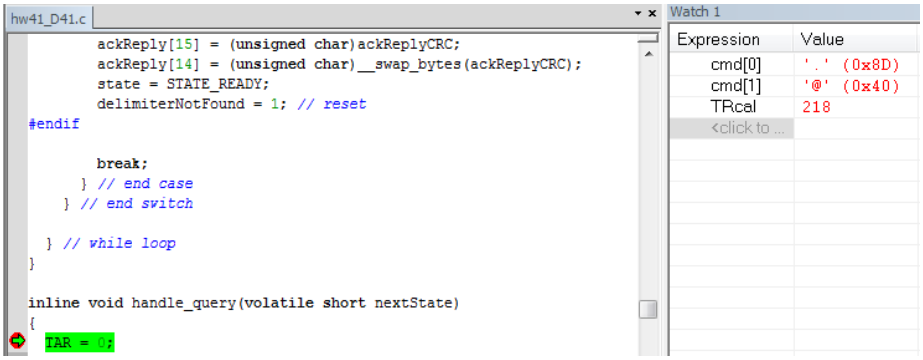
The WISP tag logic is implemented as a state machine (see appendix H.3), which starts in the *ready* state. When the tag receives a query command, it goes into the *reply* state and replies with a RN16 response. When receiving a new command it goes into a new state depending upon the command. The query command is the start of the inventory round and it tells the response frequency the tag should use (The WISP always use 256 kHz) and the length of a data-0 and data-1 in the commands of the inventory round.

3.3.1 The values of the query

It is important that a RFID tag receives a query command from a interrogator correctly, otherwise the communication between them is broken. To ensure that our query command from the MacReader was received correctly, we wanted to test what values the WISP tag received. We tested this by having a WISP tag connected to a host machine and debugging its program with a breakpoint set inside the *handle_ query()* interrupt and adding watches to the *cmd* array variable for indexes 1 and 2, which is where the received values are stored[1]. We then transmit a query to the WISP tag from first the MacReader and then from the Impinj Speedway R220.**(During the test we were able to identify the values for the following fields in the query command: "Command", "DR", "M", "TRext", "Sel", "Session" and "Target". This was done by repeatedly sending queries with their field values different from each other)**



(a) MacReader query received by WISP



(b) Impinj Speedway R220 query received by WISP

Figure 3.2: Queries received at WISP

Figure 3.2 shows that the received values for the MacReader query are *0x8C* and *0x00* and for the Impinj Speedway R220, *0x8D* and *0x40*. These values are in hex format which is converted into binary to facilitate the comparison to the query structure (see figure 1.3). In binary form, the values are *1000 1100, 0000 0000* and *1000 1101, 0100 0000*. The first 4 bits are the command field. Both queries have *1000* as values here, which shows that they are query commands. The fifth bit is for the *DR*. The sixth and seventh are for *miller*

[1]For a full description of the WISP software setup, see appendix H.2

encoding (M). The eighth bit is the *TRext* field. The ninth and tenth are for the *Sel* field. The eleventh and twelfth are for the *Session* and the thirteenth is for the *Target*. With the debug information from the WISP we verified that the tag received the query fields correctly.

3.4. Algorithmic tests

3.4.1 Volk performance

The gen2reader uses a custom implementation to calculate the mean over a set of input samples. Given the introduction of the Volk tool in newer versions of GNU Radio, it is relevant to test the potential performance effect of replacing that custom implementation with the equivalent Volk method. The Volk library, as mentioned in section 2.1 is a library, optimized for vector operations, which makes it a good candidate for improving the performance of the mean method, since the signal stream essentially a vector. The relevant Volk library method is called *volk_32f_stddev_and_mean_32f_x2* and as the name implies, also calculates the standard deviation, something which might potentially increase the processing time.

3.4.1.1 Test description

We tested these mean methods by generating vectors of samples of random values in the range from 0.0 to 1.0 with a resolution of 0.1, meaning that a sample could have a value of 0.0,0.1,0.2...1.0. We varied the vector sizes from 8 to 128 samples. Each randomly generated vector was fed to each of the mean methods, so they had the same conditions. We written the test code in a separate C++ project, in order to eliminate irrelevant overhead, and use the C++ clock() method to get the local clock time before and after each loop. To normalize the results, each test was run 1000 times and the average time spend by the mean methods was used as the results. The algorithm used for testing the mean methods is shown in algorithm 3.

Algorithm 3 The test setup for a mean method

1: **for** $SampleSize := SampleSize_{min}$ **do** $SampleSize = SampleSize_{max}$
2: $Take\ TimeStamp_{start}$
3: **loop** $NumberOfRun$
4: Use **Mean Method**
5: **end loop**
6: $Take\ TimeStamp_{end}$
7: $AverageTime = TimeStamp_{end} - TimeStamp_{start}$
8: $AverageTime = AverageTime/NumberOfRun$
9: **end for**.

3.4.1.2 Results

Figure 3.3 clearly shows that the Volk method is the optimal choice for nearly all vector sizes, except $sample_{size} = 8$, where the gen2reader algorithm took 0.782μs and the Volk took 0.951μs. However the most relevant comparison is at $sample_{size} = 128$, since

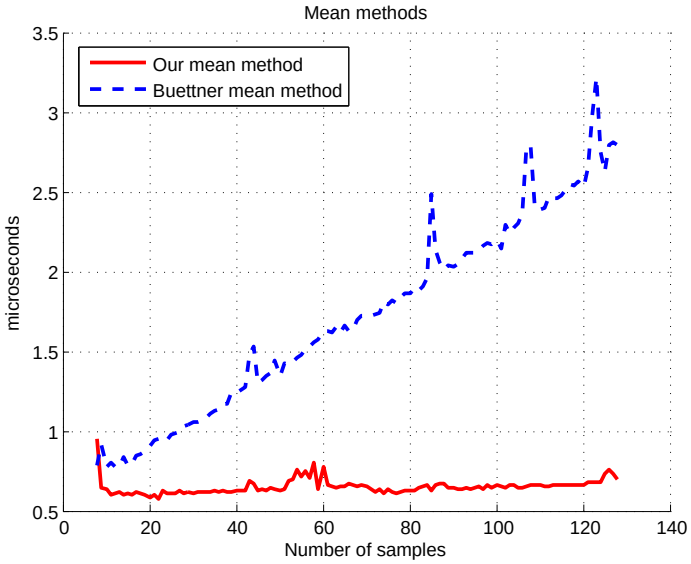


Figure 3.3: Comparison of using Volk versus a the custom gen2reader algorithm for calculation of vector mean.

that is the vector size used in the gen2reader. For $sample_{size} = 128$, the gen2reader took 2.792μs and Volk took 75.3% less at 0.69μs. For the ideal vector size of ≈12 samples as calculated in section 2.2.5.3, the Volk method reduces the processing time by 11% or 0.16μ. That means that with the reduced vector size in the MacReader there is very little to be gained by switching to the Volk mean method in terms of performance, but we will still recommend to use it since it also reduces the complexity of the code, and it calculates the standard deviation as a by-product, something which might be useful to have in some future usage scenarios for the MacReader.

3.5. GNU Radio framework tests

3.5.1 Message passing overhead

Passing messages in GNU radio has some advantages over streaming data between the blocks:

1. It is the only method were you can pass information upstream.

2. It is in a packet form, therefore you know the boundaries of the data.

Because it is closely tied to the GNU Radio scheduler, the only guarantee with regards to the time of reception is that a message will be delivered before the receiving block is next scheduled. Before introducing a new message passing paradigm into the system we wanted to understand the performance implications it would have, since it was already hard for the system to satisfy the Interrogator→Tag link timing requirements of the EPC standard.

3.5.1.1 Test description

To setup the tests we implemented 4 identical GNU Radio blocks whose only responsibility was to receive and forward messages. We also implemented a fifth block with the responsibility of measuring the time and calculating the average time spent. The blocks passed the message between them based on a *Round-Robin* setup, conceptually connecting the blocks in a circle, as illustrated in figure 3.4.

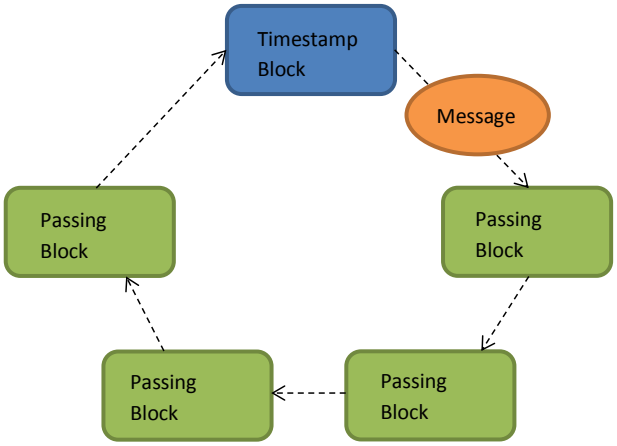


Figure 3.4: The *round-robin* scheme used in the message passing test

The test program starts by sending a message to the timestamp block. The first time the timestamp block recevies a message it will take a timestamp just before forwarding the message. After that it will simply forward the message until reaching a count of 100000, at which point it takes another timestamp and calculates the average time spent. We perform this test for two different scenarios:

1. We unpack and pack the message, before sending it to the next block. This is to test the real-world overhead of using message passing.

2. We send the message without unpacking and packing it. This provides the baseline.

The algorithms 4 and 5 shows the block logic.

3.5.1.2 Results

Figure 3.5 shows that message passing is expensive when compared to the link timing restrictions. The average time for the unpacking/packing message scenario was 26.2933μs, which is 84.13% of the available time for $T2 = 31.25\mu$s and 5,25% for $T2 = 500\mu$s. The scenario where the blocks only forwarded the messages and did not process them, showed an average time of 23.5044μs, which is 11.86% less than the scenario where the blocks unpacked and repacked the messages, which indicates, that it is the message passing itself and not the data handling that is expensive. However, the latency variation was very high, with results between 17μs and 30μs, which is why we averaged over 100000 runs. This points to inefficient scheduling by the GNU Radio scheduler, something which could be further researched at a later time. The bottom line is that message passing adds a large

Algorithm 4 Algorithm for the timestamp block

```
1: Receive message
2: Unpack message                                   ▷ Only done in 1. scenario
3: if first_run then
4:     Take timeStamp_begin
5: end if
6: if current_run == number_of_runs then
7:     Take timeStamp_end
8:     total_time = timeStamp_end − timeStamp_begin
9:     mean_time = (total_time/number_of_runs)/number_of_blocks
10: end if
11: current_run++
12: Pack message                                    ▷ Only done in 1. scenario
13: Send message
```

Algorithm 5 Algorithm for the message-passing blocks

```
1: Receive message
2: Unpack message                                   ▷ Only done in 1. scenario
3: Pack message                                     ▷ Only done in 1. scenario
4: Send message
```

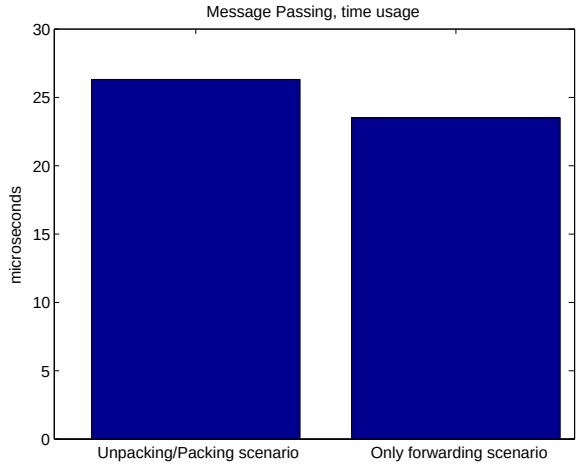


Figure 3.5: The average time spent per message passed for both scenarios.

overhead to a GNU Radio application, which means that it should only be used when absolutely necessary. Unfortunately it is the only built-in method for upstream data exchange, which means that there are cases where it is unavoidable. We have incorporated these results into the MacReader by making one message queue per configuration parameter, and thereby ensuring that only the relevant message handlers are triggered whenever there is a dynamic configuration change. Originally we had made a configuration object which was distributed to all blocks whenever a configuration change happened. This required only a single message queue, but had unacceptable overhead as evidenced by these tests.

3.6. Protocol tests

3.6.1 RTcal misconfiguration

The EPC standard specifies that RFID tags should use the RTcal symbol to calculate a pivot as explained in section 1.2.5, which is then used to distinguish between data-0 and data-1 symbols in the command received from an interrogator. Because the RTcal length was misconfigured in the version of the gen2reader that we originally got was misconfigured, we decided to test exactly how sensitive the RFID tags are to a misconfigured RTcal symbol.

The RTcal symbol length is equal to the sum of the lengths of the data-0 symbol and the data-1 symbol. Our data-0=24 μs, data-1=48 μs and $RTcal = 24 + 48 = 72$ μs and the pivot is half of the RTcal symbol $pivot = 72/2 = 36$ μs.

3.6.1.1 Test description

In order to test the effect of a misconfigured RTcal symbol, we adjust the RTcal length and test whether the transmitted query command elicits a tag response. We expected edge cases for the misconfiguration around the points where the pivot was equal to the length of respectively a data-1 symbol and a data-0 symbol, or in other words when the RTcal length is respectively double the length of a data-1 symbol, or double the length of a data-0 symbol. The test was performed with the Eureka EURUHFT(4931/5202) tag.

3.6.1.2 Results

Table 3.1 shows the RTcal lengths at which we always received a response, rarely received a response and never received a response. These three scenarios represent the lower limits of the tolerance towards a misconfigured RTcal. The same test was performed for the upper limits of the tolerance towards a misconfigured RTcal length. Those results are shown in table 3.2.

RTcal close to data0	Got RN16 answer
(2*data0 + 7) = 55μs	always
(2*data0 + 6) = 54μs	rarely
(2*data0 + 5) = 53μs	never

Table 3.1: This table shows when Eureka EURUHFT tag responded with an RN16 for a RTcal configuration close to the double length of the data-0 symbol.(rarely means 1 out of 10 times came with a RN16 response)

RTcal close to data1	Got RN16 answer
(2*data1 - 14) = 82μs	always
(2*data1 - 13) = 83μs	rarely
(2*data1 - 12) = 84μs	never

Table 3.2: This table shows when Eureka EURUHFT tag responded with an RN16 for a RTcal configuration close to the double length of the data-1 symbol.(rarely means 1 out of 10 times came with a RN16 response)

It turns out, that the Eureka EURUHFT tag is more tolerant towards a an RTcal that is too long than one that is too short. The upper limit to receive consistent responses was an RTcal of 82μs, which gives a pivot of $82/2 = 41\mu$s. The lower limit was 55μs, which gives a pivot of $55/2 = 27.5\mu$s. This gives pivot offsets of respectively 8.5μs and 4μs, showing that the tag is aproximately twice as tolerant towards longer RTcal values than towards shorter RTcal values. The results of this test lead directly to the next test, since we observed that the RTcal length had a significant effect on the Tag→Interrogator link timing.

3.6.2 RTcal impact on T1

During our test of the RTcal misconfiguration, we observed that the Eureka EURUHFT tag's response time to the query command was affected by the length of the RTcal symbol. Bounded by the RTcal symbol boundaries identified in the previous test, table 3.3 shows the measured response link timing.

RTcal	**T1**
54μs	56.4μs
60μs	62μs
66μs	68.4μs
72μs	74μs
77μs	79.2μs
83μs	85.2μs

Table 3.3: The Eureka EURUHFT tag's response time to a Query command at different RTcal lengths

We decided to test if the effect of the RTcal symbol length is the same for different tags, so we repeated the test on an Avery Dennison tag. In figure 3.6 we have plotted the least squares best fit of each test to show that there is a clear linear relationship between the RTcal symbol length and the Tag→Interrogator link timing.

We found that the Avery Dennison tag showed a similar linear relationship between the RTcal symbol length and the Tag→Interrogator link timing, but with a different absolute offset. From these measurements it is clear, that RTcal misconfiguration has the same effect on different tags. We were able to manipulate the response time between 56.4μs and 85.2μs for the Eureka EURUHFT tag and between 52μs and 81.2μs for the Avery Dennison tag. It may be worth it to test the applicability of manipulating the RTcal symbol length to achieve shorter tag population inventory times.

3.6.3 Tag power-up CW length

An RFID tag needs to be powered up before it can hear the commands sent from a reader. This is done by sending a CW to the RFID tag at the beginning of each inventory round. The gen2reader code uses a tag power-up CW length of 1430μs, and refers to the EPC standard to support this choice, but we have not been able to find any references to this number in the standard. We are interested in finding the shortest amount of the *settling time*, where a RFID tag response to a Query command. Figure 3.7 shows the tag power-up CW transmitted before the query command.

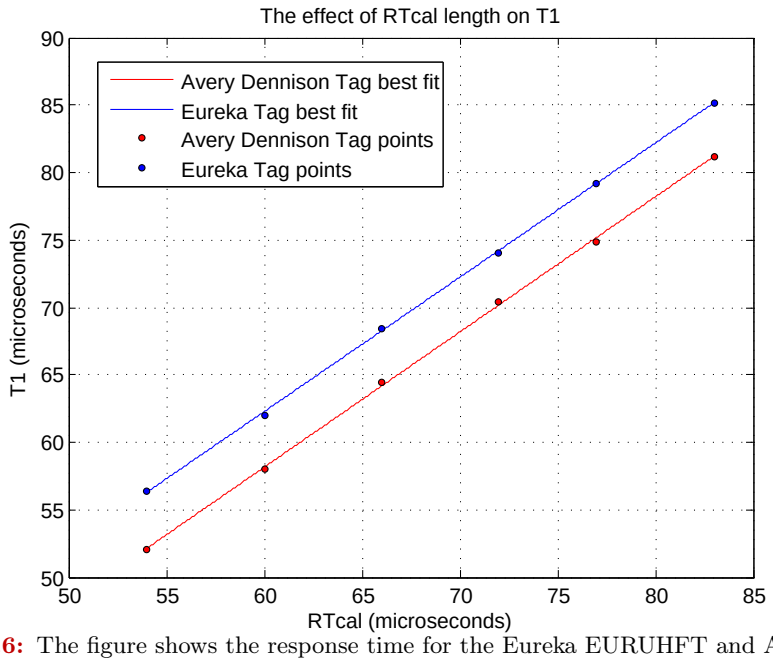


Figure 3.6: The figure shows the response time for the Eureka EURUHFT and Avery Dennison tags for different RTcals and the least square best fit of the measurements

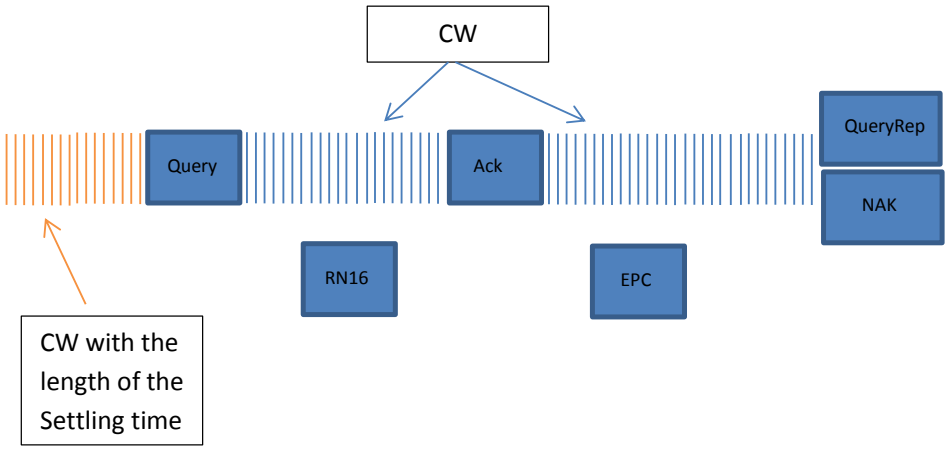


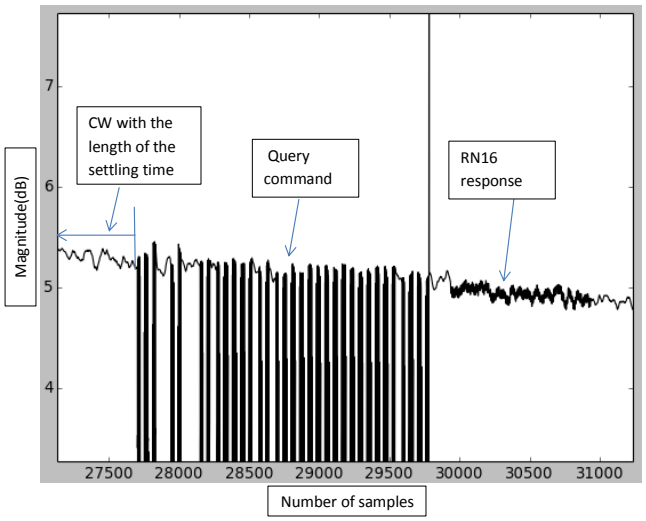
Figure 3.7: The tag power-up CW before the query

3.6.3.1 Test description

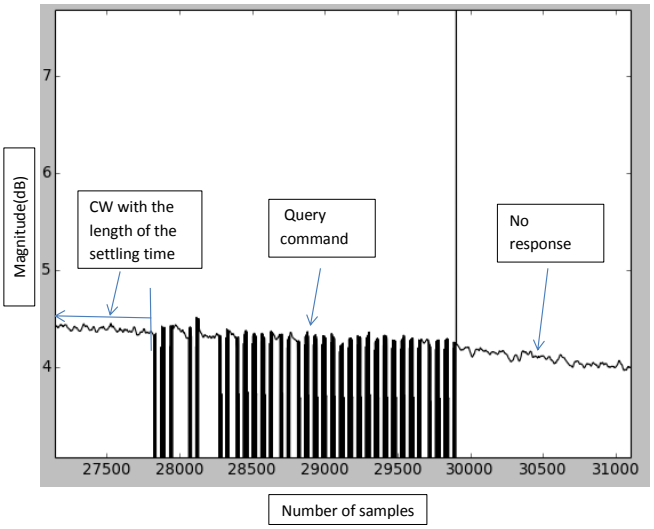
To find the shortest possible tag power-up CW, we started by setting the length in our MacReader to 1430 μ, as used in the gen2reader, and decreased it until the tag no longer reponded. We used the Avery Dennison tag for these measurements.

3.6.3.2 Results

We stopped getting answers from the Avery Dennison tag at a CW length of 855μs, meaning that the shortest CW length with a response from the tag was *CW length=856μs*. That is a difference of 574μs compared to the value used in the gen2reader. This can be seen in figure 3.8.



(a) The Avery Dennison tag responds with an RN16 when the initial CW length is 856 μs



(b) The Avery Dennison tag does not respond to the Query command when the initial CW length is 855μs

Figure 3.8: Tag responses for different CW lengths

3.6.4 Response time of the MacReader

The response time *T2* of a interrogator is a critical point of the RFID system, if the interrogator takes too long to answer the RFID tag, then the RFID tag will ignore the command from the interrogator. It is properly the must difficult aspect of the interrogator, when a RFID tag sends an response to the interrogator, then the interrogator has to capture the signal from the tag, decode the response, interpret the response, change the state of the interrogator, construct the next command, encode the command and sent it all within the allowed response time. *T2* is the ideal reference point for telling, if the

interrogator is fast enough in its handling of data in a RFID system.

3.6.4.1 Testing the T2 of the MacReader

We wanted to test how fast the MacReader responded to a RFID tag, to see if the MacReader was able to answer a tag within the time limit *T2* of the EPC standard. To test the response delay we set up the MacReader to run an inventory round and placed the Eureka EURUHFT tag between the antennas of the USRP with a distance of 0.3cm to each of the antennas. We added a *file sink* to the receive chain which stored the signal in a file. The signal was then plotted with *pyplot*. The frequency for the tag was set to 40 kHz, which leads to a *T2* limit of maximum 500μs. We ran the MacReader with the Linux commands *nice –20* and *ionice -c1*. *nice* is a scheduling priority command and *-20* is an argument for most favourable scheduling. This gave the MacReader the highest possible scheduling priority when it was executed. *ionice* is an I/O-scheduling priority command and *c1* sets the system to *real-time* scheduling on the I/O. Since Ubuntu is not a real-time operating system, this means that the operating system will only attempt to approximate real-time performance.

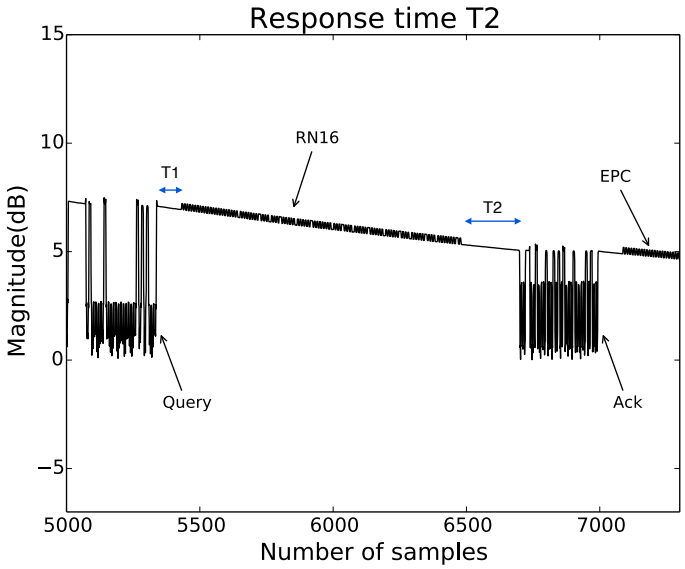


Figure 3.9: The *T2* response time of the MacReader (223 samples at a sampling rate of 390625).

Figure 3.9 shows the response time *T2* of the MacReader for the RN16 response from the Eureka EURUHFT tag. The length of *T2* is 223 samples and the sampling rate is 390625 samples per seconds. This means that the MacReader used

$$\frac{T2_{length}}{sampling\ rate} = \frac{223s}{390625S/s} = 570,88\mu s \tag{3.1}$$

This shows that the MacReader is actually 70.88μs, or 14% too slow, but the tag still responded with an *Ack*, which means that the Eureka EURUHFT tag has a higher tolerance for response time delay.

3.6.5 RN16 response from the WISP tag

During the development of the MacReader, we used the WISP 4.1 tag to test different aspects of the behaviour, such as the data that the WISP tag received from the query command and the RN16 that the interrogator received from the WISP tag. We noticed that the WISP RN16 response had two consecutive long pulses, which should not occur in Miller encoding. We decided to verify that this was indeed an error in the encoding and not just a misunderstanding.

3.6.5.1 Test description

To test if the WISP tag respects Miller encoding we did the following:[2]

1. We send a query command from the interrogator to the WISP tag[3].
2. We store the response captured by the interrogator in a file and plot the signal.

3.6.5.2 Results

The RN16 response we got the from the WISP tag is shown in figure 3.10**(We don't show the x-axis and y-axis for figure 3.10, 3.11, 3.12 and 3.13 because otherwise the structure of the signal in these figures would be too compressed for visualisation)**.

Figure 3.10: The RN16 response from the WISP tag (The x-axis is 'Number of samples' and the y-axis is 'Magnitude(dB)')

To test if the data in the response is valid, we will first have to remove the preamble and the dummy bit from it. The preamble from the response is shown in figure 3.11.

Figure 3.11: The preamble of the response from the WISP tag(The x-axis is 'Number of samples' and the y-axis is 'Magnitude(dB)')

The dummy bit from the response is shown in figure 3.12.

Figure 3.12: The dummy bit of the response from the WISP tag(The x-axis is 'Number of samples' and the y-axis is 'Magnitude(dB)')

The payload in the response is shown in figure 3.13.

The raw bits from this signal are the following:
(The 0s are actually -1s, but for readability they are shown as 0s)
0101010110101010010101011010101001010101101010100 1 01010101001101
0101010011010100110101001101010011010100110 1010101001101010101 0
There are 127 samples here, which is 1 sample less, than we expected, because with M4

[2]To see the full description of the RN16 response test, look in appendix K

[3]This was done with TRext=0, with the rest of the parameters being unimportant, since the WISP ignores them and uses its own settings for the response

Figure 3.13: The figure shows the payload of the response from the WISP tag (The x-axis is 'Number of samples' and the y-axis is 'Magnitude(dB)')

encoding there are 8 samples per bit and there are 16 bits in a RN16. $(8 * 16 = 128)$

To check if these samples are structured right, we revert this RN16 back to its basic form by dividing it with the square wave for M=4:

$$BasicRN16 = RN16/Squarewave$$

(This is done sample for sample in the signals)

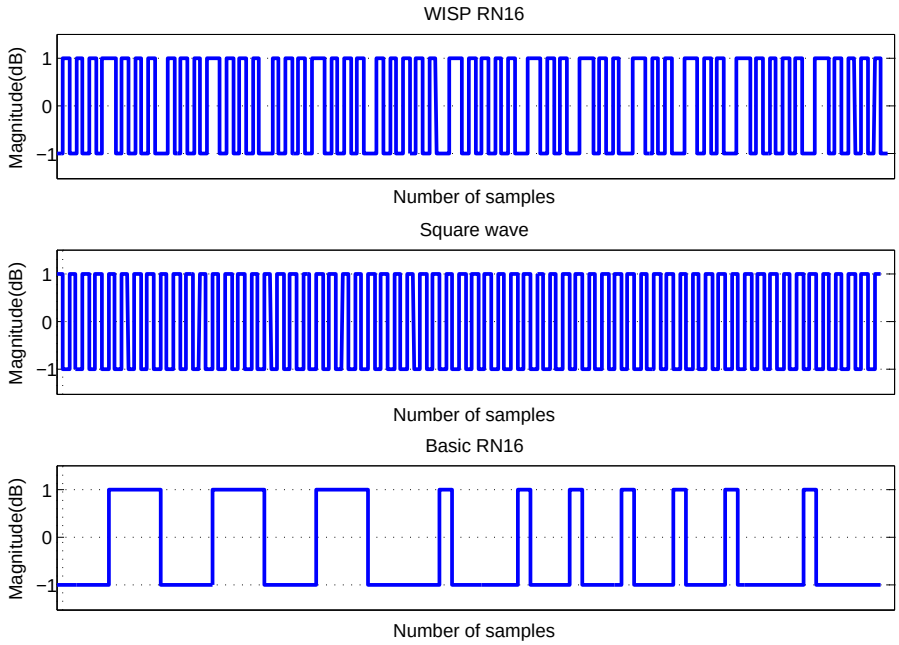


Figure 3.14: The figure shows the RN16 response from the WISP tag, the square wave and the basis RN16

Figure 3.14 shows that the basic RN16 has the following states (from left to right): S4→S1→S4→S1→S4→S1→S4(See section 1.2.4)

After that the basic RN16 does not follow the Miller state diagram any more, because the signal structure in figure 3.15 is not a valid state.

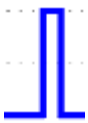


Figure 3.15: The figure shows an invalid state in the basis RN16 response from the WISP tag

Figure 3.15 shows an invalid state from the basic RN16, because there is no data symbol that inverts its phase twice in the same time period. Data-1 symbols invert their phase once in the middle of the time period and data-0 symbols invert the phase between

two consecutive data-0 symbols. This shows that the WISP tag response with an invalid structure.

3.7. Hardware tests

3.7.1 Transmit power of the daughterboard WBX 50-2200 MHz Rx/Tx

The RFID tags get their power from the signal transmitted by the interrogator, so it is vital to know how strong the transmitted signal needs to be in order to power up the tag. Since we were experiencing difficulty communicating with the tags, we wanted to find out whether the reason could be that the transmitted power from the USRP N210 is too low. The only reference point we have for the gen2reader is from the article "A flexible software radio transceiver for UHF RFID experimentation" by Buettner and Wetherall [5], where they state that the interrogator used had an output power of 75 mW.

3.7.1.1 Gain

GNU radio allows you to manipulate the transmission power through a *gain* setting in the USRP sink block. This gain has a range from 0 to 25, where a higher gain means a higher transmission power.

Gain	dBm	mW
25	19.27	84.527884516
24	19	79.432823472
20	15.2	33.113112148
18	11.7	14.791083882
14	4.79	3.0130060242
10	-2.1	0.61659500186
9	-3.7	0.4265795188
8	-5.8	0.26302679919
5	-7.7	0.16982436525
3	-7.7	0.16982436525
1	-7.9	0.16218100974
0	-7.97	0.15958791472

Table 3.4: This table shows the measured output power for the radio with the daughterboard WBX 50-2200 MHz Rx/Tx, at various gain settings.

3.7.1.2 Measuring the output power

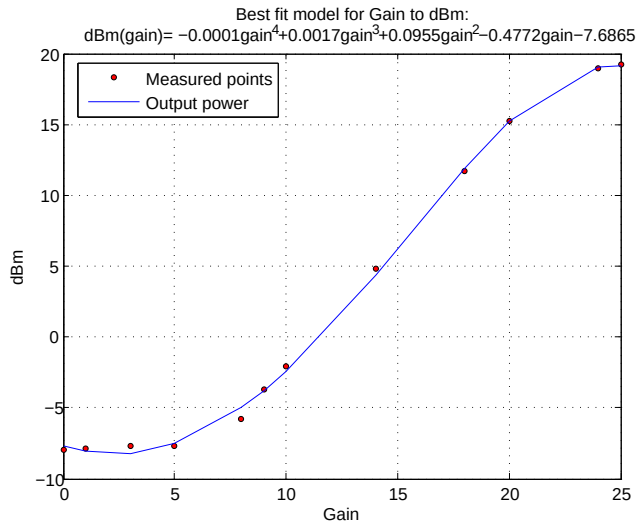
To measure the output power for the daughterboard, one needs a power meter or a spectrum analyser. We used an FSQ26 spectrum analyser to measure the output power.

To protect the FSQ26 spectrum analyser, an attenuator was connected to it, which weakened the signal from the USRP by 10 dB. The cable used for the connection between the FSQ26 spectrum analyser and the USRP also weakened the signal by 0.5 dB. The

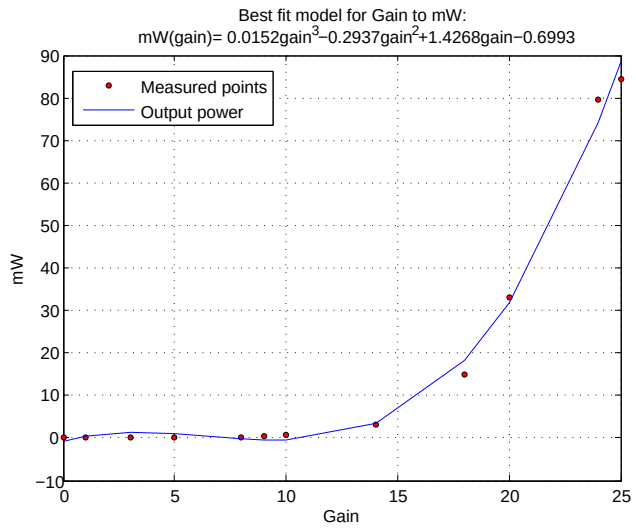
measurements were done with a frequency tuned to 868 MHz[4]

3.7.1.3 Output power results

With all the attenuation effects taken into account, table 3.4 shows the results of the measurements. Based on the 12 measurements in table 3.4, we used least-squares best fit to calculate the relationship between the gain and the transmission power, which is shown in figure 3.16. The resulting equations are not necessarily accurate representations of this relationship, but do provide a close approximation.



(a) Gain To dBm model



(b) Gain To mW model

Figure 3.16: Best fit models for gain to power

The maximum output power for the transmission from the daughterboard was 84.52 mW at a gain of 25, which is larger than the 75 mW mentioned by Buettner. This shows

[4]To see the full description of the output power test setup, look in appendix L

that the daughterboard is capable of producing a sufficient amount of power.

3.7.2 TX/RX shielding

Dobkin [18] explains how one of the weaknesses of bistatic RFID interrogators is leakage between the transmitter and the receiver antennas, since the transmitted signal from the interrogator is so much stronger than the response from the tag. We wanted to test whether we could reduce the self-interference between the antennas through shielding.[5]

3.7.2.1 Shield method

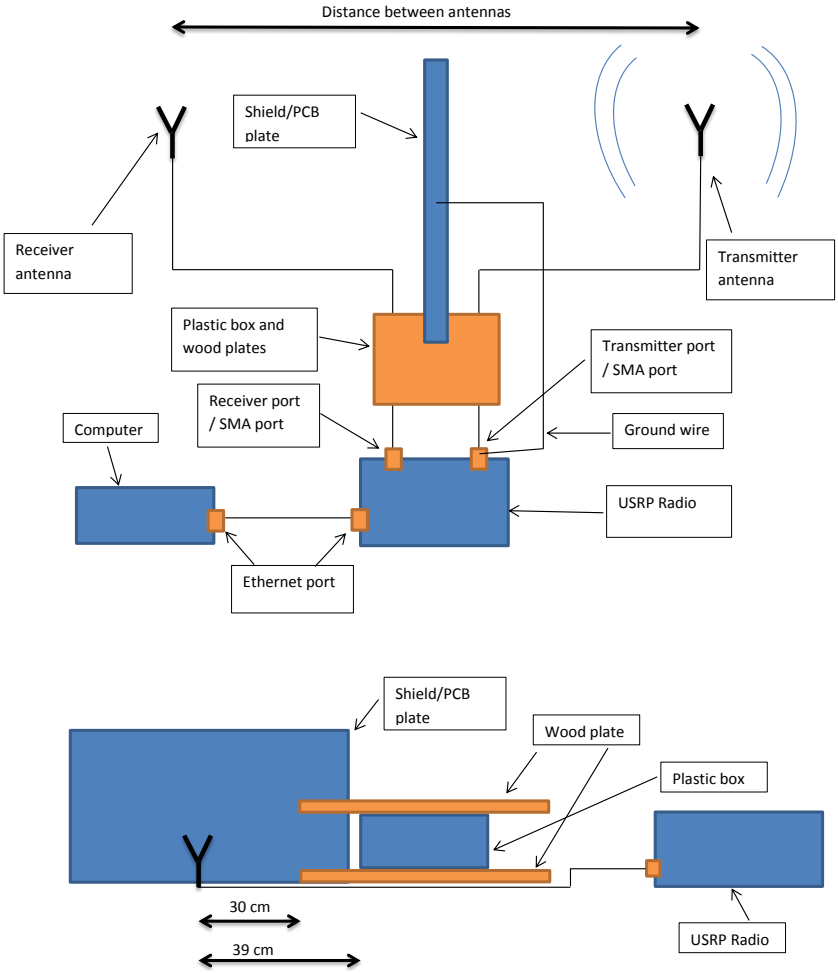


Figure 3.17: The shield placement between the antennas of the USRP

Our first method to attenuate the signal from the transmitter antenna is to place a shield between the transmitter antenna and the receiver antenna, which is illustrated in figure 3.17.

In the first test the antennas are placed in a vertical position, where they are placed parallel to each other, which is shown in figure 3.18. The overview of the setup is shown in figure 3.19.

[5] To see the full description of the shield test setup, look in appendix M

Figure 3.18: The antennas placed vertical to each other.

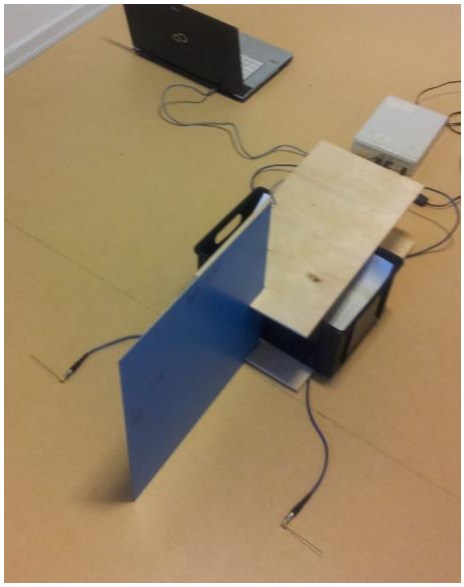


Figure 3.19: The shield test setup

The shield is a Printed Circuit Board (PCB) plate and it is grounded to the ground from the transmitter antenna, as shown in figure 3.20. This way the signal should be attenuated in the direction from the transmitter antenna to the receiver antenna.

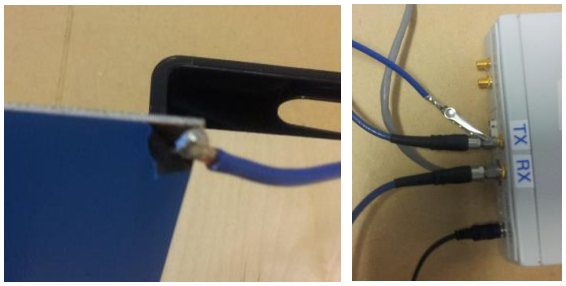


Figure 3.20: The PCB plate is soldered to a wire, which is connected to the ground of the transmit antenna

3.7.2.2 Horizontal method

Our second method to attenuate the signal, is to position the antennas in a horizontal position with their bases facing each other, which is shown in figure 3.21. This way the signal from the transmitter antenna will be weakened in the direction of the receiver antenna.

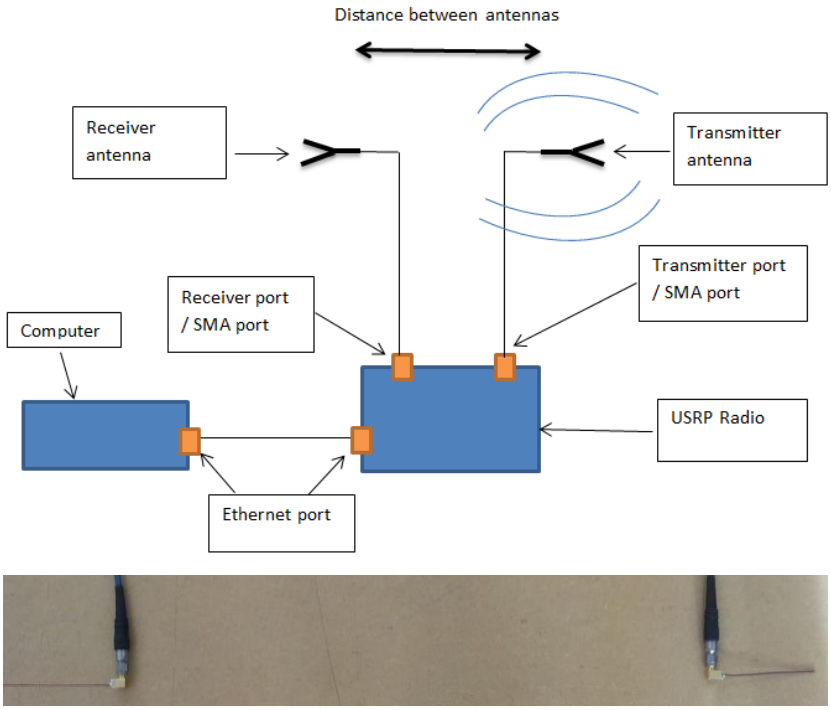


Figure 3.21: The antennas placed in a horizontal position with their bases facing each other.

3.7.2.3 Grounded antennas method

Our last method to attenuate the signal, is to ground the antennas, by soldering a PCB plate to the base of the antenna. There are 3 test scenarios for this method:

1. Only the receiver antenna is grounded, and placed in a horizontal position with a edge of the PCB plate touching the surface and the base is facing the transmitter antenna. The transmitter antenna is placed in a vertical position, as shown in figure 3.22.

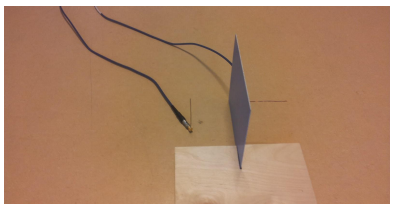


Figure 3.22: The receiver antenna is grounded, while the transmitter antenna is not.

2. Both antennas are grounded. The receiver antenna is placed in a horizontal position with a edge of the PCB plate touching the surface and the base is facing the transmitter antenna. The transmitter antenna is placed in a vertical position with the base of PCB plate touching the surface, this is illustrated in figure 3.23.

3. Both antennas are grounded, they are placed in a horizontal position with a edge of the PCB plates touching the surface and with their bases facing each other, as shown in figure 3.24.

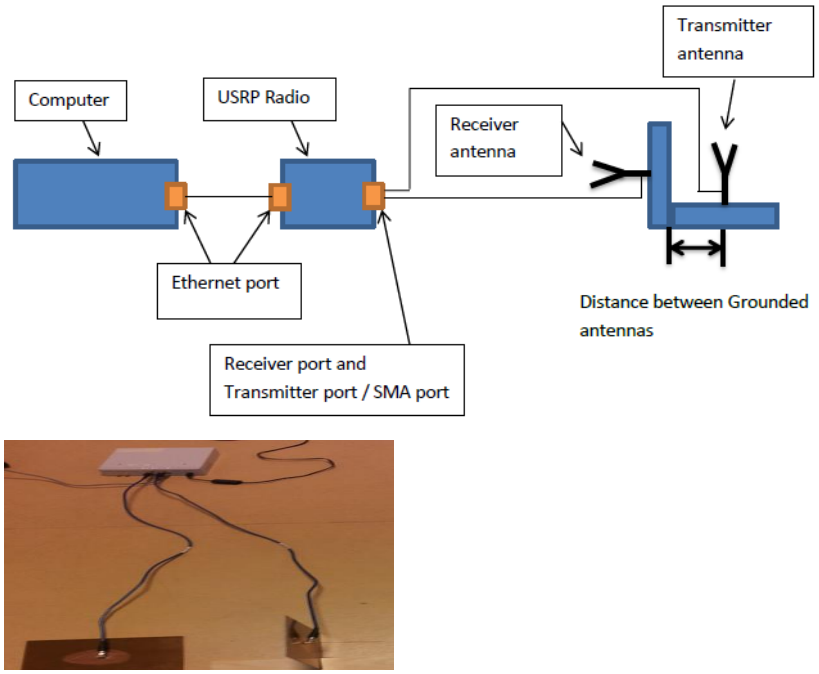


Figure 3.23: The antennas grounded to a PCB plate each.

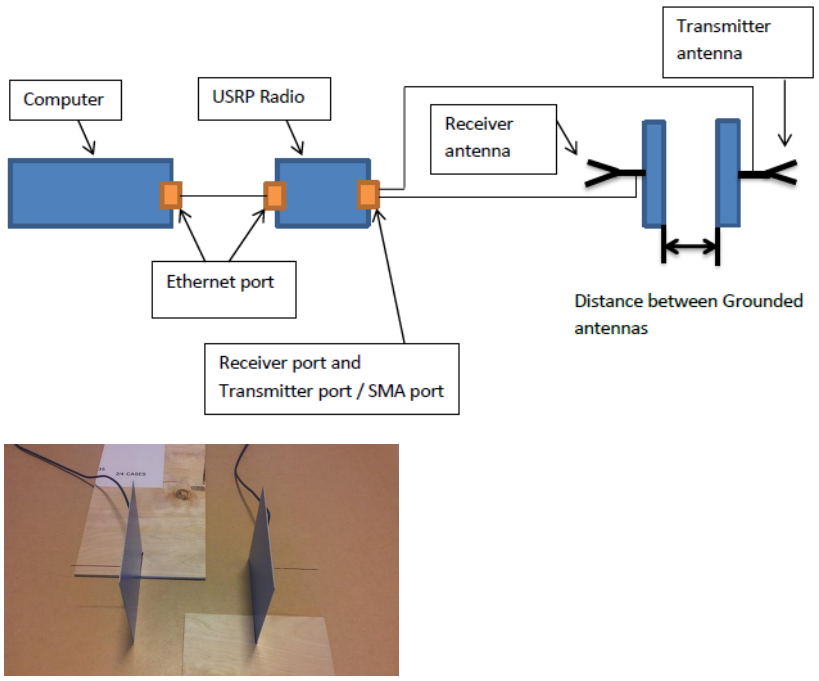


Figure 3.24: Both the antennas are grounded with their bases facing each other

3.7.2.4 Shield testing results

The results for the attenuation effect of the shield method, horizontal method and grounded antennas method can be seen in figure 3.25 and figure 3.26. The top graph in figure 3.25 shows the attenuation effect on the received signal by placing a shield between the two antennas. The bottom graph in figure 3.25 shows the attenuation effect on the

received signal by placing the antennas in the horizontal position and also placing a shield between the two antennas.

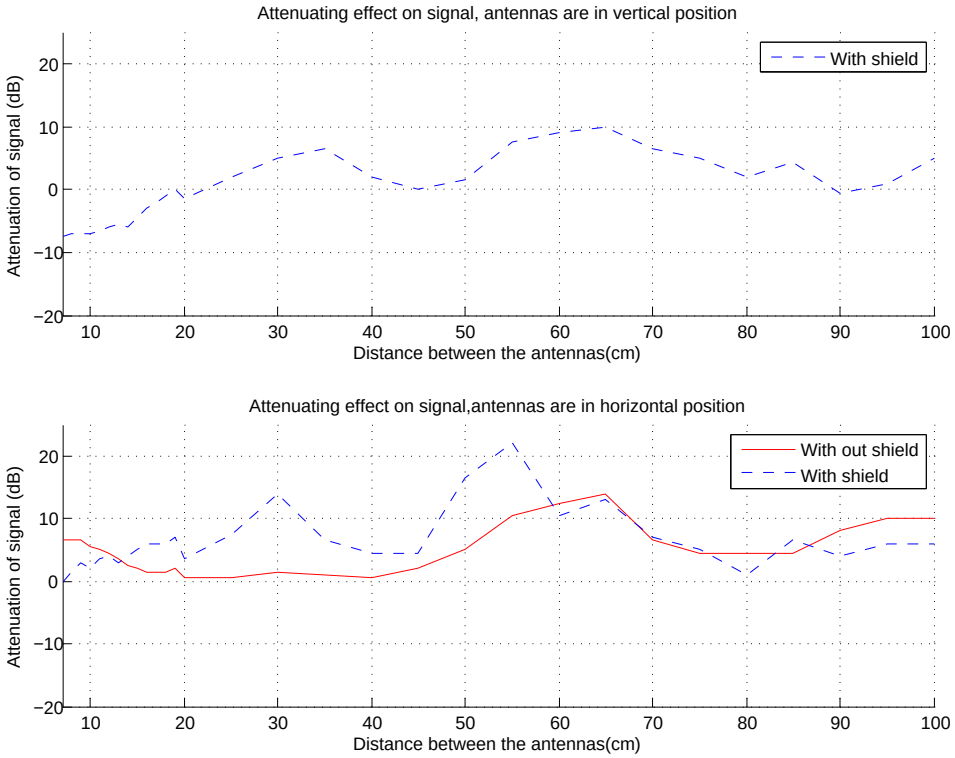


Figure 3.25: Signal attenuation with a shield for both vertical and horizontal position

The top graph in figure 3.26 shows the attenuation effect on the received signal by placing the receiver antenna in a horizontal position and the transmitter antenna in a vertical position, this is done where both antennas are grounded and also where only the receiver antenna is grounded. The bottom graph in figure 3.26 shows the attenuation effect on the received signal by placing the antennas in a horizontal position where both antennas are grounded.

These measurements used the vertical position with no shield as the reference point, so when a graph shows an attenuation of 5 dB, it means that the signal for the given attenuation method, at the given distance, is 5 dB weaker compared to the vertical position with no shield at the same distance and if it shows an attenuation of -5 dB, then the signal is 5 dB stronger.

The attenuation methods prove very difficult to use, because the results varied a lot. The test with the antennas in vertical position and with use of the shield, showed that within a distance of 20 cm, the signal was amplified and after the 20 cm distance the signal weakened between 0-10 dB for the various distance with no clear relationship between the attenuation effect and the distance.

The test with the antennas in horizontal position, was the only test where the signal did not get amplified, but the attenuation effect still varied a lot for the various distances. The shield did not provide any clear result here either. At a distance of 55 cm the shield weakened the signal with 11 dB more than without the use of it, but at a distance of less than 12 cm, the signal was weaker without the use of the shield than with it.

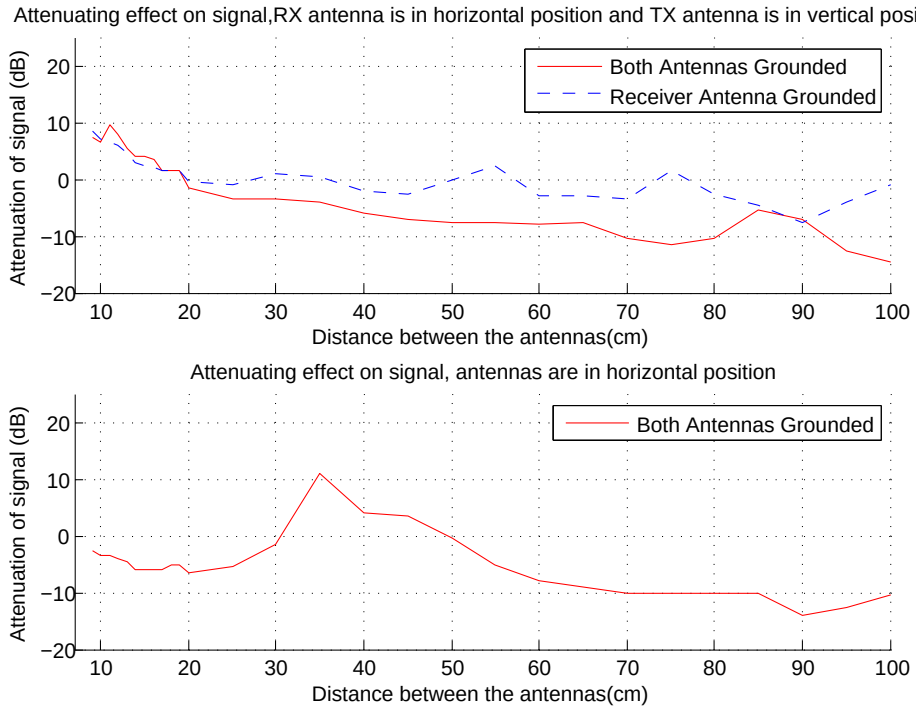


Figure 3.26: The signal attenuation with grounded antennas

The test with the receiver antenna in a horizontal position and the transmitter antenna in a vertical position, proved even more inconclusive. At a distance larger than 20 cm the signal was amplified between 0-10 dB. The amplification was larger when both the antennas were grounded compared to when only the receiver antenna was grounded.

The test with both the antennas grounded and their bases facing each other also came with inconclusive results. At a distance of 35 cm the signal was weaken by 10 dB, but here the attenuation effect spiked and if the antennas were closer to each other or moved away from each other, the attenuation effect would decrease.

The tests in this section have been restricted by the available tools:

1. We would have liked to perform the tests in a sound room, where the walls could have absorbed the signals. The room, where the tests were performed, was a standard office setting with no control over the signals, when they reflected from the surface of the surrounding environment. It is very likely that this had an influence on the reading, since the reflection of the signals could have been in the received signals.

2. We would have liked to do the measurements in dBm and not dB, since that is a absolute value for describing the signal strength and not a relative values that dB is. Unfortunately this information is not available via GNU Radio[36] and the only thing one can measure is the amplitude or magnitude of the signal. To measure the received signal in dBm one would have to place a power meter or spectrum analyser between the receiver antenna and the radio.

3. We would also have liked to test how strong the signal should be for enabling the RFID tags, since none of the tag providers have provided that information about

the tags. To get that information one would have to place a voltmeter between the antennas, to check when the voltage is high enough to enable the micro controller on the RFID tag and a power meter or spectrum analyser to get the signal strength.

3.8. Analysis of gen2reader attributes

3.8.1 Clock skew

The gen2reader clock synchronizer logic can be explained with the following pseudo-code:

1. Update the clock rate (i.e. samples per pulse), but only if the sample count is within the skew tolerances. This is based on the previous clock rate, the number of samples since the last zero crossing and some tuning variables

2. Calculate how many pulses there were since the last zero crossing. This is based on the number of samples per pulse

3. Add 1 bit per pulse.

The problem with this approach is that it is only tolerant to the levels of clock skew specified in the EPC standard. The tags themselves can generally be trusted to comply with the standard, but the signal that reaches the synchronization block has had additional clock skew introduced by the center block.

The nominal number of samples per pulse hardcoded in the gen2reader is 4. If the number of samples per pulse is larger than 4.88 or less than 3.12, the clock synchronizer will ignore the pulse and not use it to update the BLF estimation. However, since the clock syncronizer is incapable of counting a non-integer number of samples between zero-crossings, the check is equivalent to checking whether the number of samples is different from 4. This means that **the gen2reader will only update the clock skew when there is no clock skew, or in other words, that the BLF estimation is never updated**.

Therefore in the following we can safely assume a nominal number of samples per pulse to always be 4. In order to convert a pulse into a number of bits, the gen2reader clock synchronizer takes the number of pulses between zero-crossings and divides by the nominal number of samples per pulse, i.e. 4. This calculation looks like the following:

```
int num_pulses = (int) floor(
        (samples_between_zcs / samples_per_pulse)
                + 0.5);
```

The problem is that by our count, a valid pulse coming out of the center block can be anywhere from 4 to 6 samples and a valid double pulse can be anywhere from 8 to 10 samples. This means that at 6 samples, a single pulse will be converted to 2 pulses, and at 10 samples a double pulse will be converted into 3 pulses. It is not as fragile as it may sound, because this only occurs when there is noise. However, we want to be robust to noise. In addition, it becomes even more fragile when considering that the centering block calculates the mean over a window of 128 samples, offsetting the mean because of three things:

- The mean of sample i is calculated based on the previous 128 samples, not the 64 samples before and the 64 samples after.
- The signal has a natural drop-off due to the hardware (may not have been the case on the USRP1), which means that the mean calculated for sample i is based on a signal that has been falling for 128 samples. This makes the mean higher than it should be, and therefore produces more zeros than ones.
- If there is noise in the signal, the signal magnitude gets offset for periods much shorter than 128 samples, which means that those whole periods will appear to be either below or above the average, as illustrated in figure 2.10 in section 2.2.5.3.

Chapter 4

Lessons learned

This chapter is included in order to save others some of the hard work and headaches that we have experienced during this project. The subjects covered by no means include all the challenges that we have experienced, only those where the time spent figuring something out the first time is out of proportion with the time needed to repeat it later.

4.1. GNU Radio development

Use the GNU Radio source code The people behind GNU Radio readily acknowledge that the documentation is spotty, but that is often the nature of open source projects. Throughout our project we have experienced missing and incomplete documentation, which makes development difficult. However, **since GNU Radio is open source, it is easy to open up the source code for any block implementation and check how others have done it**. We recommend to download the complete source code and use it when in doubt about how to interpret the documentation, or when there simply is not any. There is only source code for C++ implementations available, but examples tend to be written in Python.

Use the GNU Radio mailing list When there is no documentation and the source code is not helping, another good source of documentation/help is the GNU Radio mailing list at *discuss-gnuradio@gnu.org*. Only use the mailing list after reading a guide for how to use mailing lists. And be aware that **it is not enough to send an email to the mailing list address. You must be registered on the mailing list, otherwise your emails will not arrive**.

No guaranteed number of block input samples The GNU Radio block interface defines a variable called *ninput_items_required*. The value of this variable defines the number of input items that a block requires from the upstream block. For too long we interpreted that to mean that a block would get the number of input items specified with *ninput_items_required*, however, ***ninput_items_required* only specifies the minimum number of items required**. Therefore **a block developer should never assume that they will get the same number of input items every time**, or even that it is close to the number specified with *ninput_items_required*.

Do not trust stream tag propagation Stream tags are still a recent addition to GNU Radio and not all standard blocks support it yet. Therefore, if you use a standard block, use the *tag_debug* block to verify that it propagates the stream tags, since there is no other indication of missing stream tags. If the standard block does not propagate stream tags, it can be solved by making a custom block based on the standard block source code and implementing tag propagation manually. Stream tag propagation is part of the GNU Radio blocks interface, but the standard implementation only works for blocks that have a linear relationship between the number of input items and the number of output items. If your block does not have that, it will drop the tags without notice unless you manually implement the tag propagation. Even this will fail if you do not manually disable the automatic tag propagation. This bears repeating: **to get tag propagation to work in a custom block it is necessary to implement your own tag propagation AND actively turn off the automatic tag propagation, which otherwise will interfere**.

The GNU Radio scheduler and debugging Debugging a GNU Radio application is difficult, because each block may conceptually represent its own program, but the execution of that program is interrupted by the execution of all the other blocks. Therefore identifying when and where the application crashes can be difficult. It is made harder because the GNU Radio scheduler will only schedule blocks that request more input items, which means that it is possible to have an error in a block that prevents it from being scheduled. An example of this is the MAC block, which must produce the first command before it can consume the first response. However, the GNU Radio scheduler will not schedule the MAC block under those circumstances. It took a while to understand this relationship and why the program did not run, but it is the reason that the MAC block is implemented as a source block, which does not have these restrictions, and receives the tag response through message passing.

No signal power metrics There is no way from within GNU Radio to measure the strength of a transmitted or received signal. To measure signal strength one must use hardware like a power meter or a spectrum analyser. The only thing GNU Radio can measure about a signal is the amplitude or magnitude and both are given in the relative unit decibel (dB).

4.2. Host network configuration

Subnet When configuring the host system for use with the USRP N210, it is also necessary to setup the network interface. The USRP N210 is set up with a static Internet Protocol (IP) address, which needs to be configured on the system. The USRP N210 IP address need to be on the same subnet as the host machine, otherwise it will not answer to packets sent from the host machine, because it thinks that the packets are not meant for it. Ettus Research only had a small note on this topic, so it was difficult for us to find this out, because there is no indications on this type of problem, one can only see that the USRP N210 does not respond to the packets.

Disable network manager Another issue when working with the static IP address is that the Ubuntu Network Manager will break the connection with the device every time it updates with the Dynamic Host Configuration Protocol(DHCP) server, which in our

experience is once every few minutes. In order to overcome this, it is necessary to disable the Network Manager for the specific device, as described in appendix C.

4.3. C++ development

When we first started this project, our experience with C++ was purely academic. Therefore we went through several phases before we were able to import GNU Radio out-of-tree modules into Eclipse. We started out with the Vim command-line interface editor, the GNU Project Debugger (GDB) command line debugger and using command line tools to compile and link the application. These tools are not easy to use, and definitely not for those with little C++ development experience. Through a good amount of research we were able to import the project into Eclipse. The necessary steps are described in appendix I.

4.4. UHD/USRP documentation

The UHD driver documentation is very limited. As an example, the USRP produces console output to indicate errors, but only a few of these are documented. As mentioned in section 4.1, the concept of stream tags is still relatively new. The UHD driver supports stream tags for precise scheduling, and has specific console output to indicate stream tag errors, but these are not documented. We were able to find some meeting notes from a conference where they discuss what possible console output could be used for these errors, and the errors that we experienced were among those suggestions. Another place to find documentation for the UHD console output is by looking directly in the source code for the *USRP_sink* and *USRP_source* blocks.

4.5. Physical conditions

Antennas It is possible to use homemade monopole antennas, but being linearly polarized they require that the orientation of the tag antennas and the interrogator antennas match.

Tag placement With our setup, it is necessary to keep the tags and antennas close to each other. To get the strongest, clearest signal, place the transmitter within a few centimeter of the receiver and place the tag within a few millimeter of the receiver, directly between the antennas. To emulate a noisy environment, place the tag within a few millimeter of the transmitter, on the far side from the receiver.

4.6. WISP

Power The signal radiated from the monopole antennas is not strong enough to power the WISP, but we have found that keeping the WISP connected to a host machine throught

the debugger hardware will provide the power needed for the tag to respond. The setup is described in appendix H.

Encoding errors The WISP uses invalid encoding, which cannot be decoded through cross-correlation. The proof for this is described in section 3.6.5 and appendix K. However even with the encoding errors, the Impinj SpeedWay R220 is somehow still able to decode the tag responses.

Communication with Impinj Speedway R220 In order to read the WISP tag with the Impinj Speedway R220, it is necessary to configure the interrogator to use Miller 4 encoding. Appendix F describes the hardware and software setup of the Impinj Speedway R220 and how to configure it for communication with the WISP.

WISP link timing The WISP timing values are very sensitive to the specific configuration and code implementation on the tag. Therefore WISP link timing measurements are not represeantative of behaviour on commercial tags.

Flawed random number generator The WISP random number generator is anything but random. If one understands how it works, one can control the slot selection from the interrogator, because the algorithm is depended upon the received command. The random number generator takes the first 7 bits from the command and shifts them bit wise. There is a note from the developer in the WISP code, that this was done to preserve power.

No validation of ACK The WISP firmware does not check whether it received the correct ACK. Therefore it is possible for the interrogator to send back an already prepared ACK without processing the RN16 first.

4.7. USRP calibration

In the beginning of the project process, we were not able to communicate with the WISP tags at all. We found out that the UHD comes with a calibration tool, so we ran it and repeated our experiments, but the interrogator still did not receive a reply from the WISP tags. We had originally configured the calibration tools according to the documentation accessed through the calibration tool *help* command, but found additional documentation on the Ettus website which stated that the antennas should be removed for calibration. After removing the antennas and rerunning the calibrations, we were able to get replies from the WISP tags at 1 mm distance from the transmitter, but only if the tag was placed directly between the transmitter and the receiver, and parallel to the antennas.

4.8. Tuning of monopole antennas

When we originally constructed the monopole antennas, we cut them slightly longer than a quarter wavelength with the intention to cut them shorter during tuning. We were then advised that tuning would only make an insignificant difference to the performance.

However, since we had cut the antennas so much longer at 11 cm, when we only needed them to be 8.6 cm, the difference was not insignificant. When we were made aware of the misunderstanding, we tuned the antennas which increased the readable distance to the WISP tag from ≈1 millimeter to ≈5 millimeter.

4.9. Visualization of the signal

We have found that using the Python library *pyplot* is a much more powerful way to visualize the signals from GNU Radio. Pyplot can read the raw data from GNU Radio *file sinks* and plot them with a few simple lines of script. In the resulting plotted figure it is possible to zoom to any size you want, and scroll through the signal as well. This has been an invaluable tool to understand and debug the transmitted signals, and to generate the plots used in this thesis. Pyplot allows you to save plots in PDF, which translate into scalable figures in Latex. Appendix N contains one of the scripts we have written, and a screenshot of the plot window for illustration.

4.10. Keep local copies of all external material

When we started working on the thesis, the gen2reader source code was hosted on the *cgran* repository for GNU Radio out-of-tree modules. We initially cloned only a single copy to work on, not suspecting that *cgran* would be shut down shortly after. When we went back for a clean copy there was no way to get it. It was not a major setback, since another researcher, Yuanqing Zheng had made a slightly modified version for the USRP N210 available on github [51]. However, then we became aware that De Donno et al. [14] claimed to have improved the gen2reader clock synchronizer and decoder and made the source code available in the same repository. We needed to know what these improvements consisted of, so we contacted the site administrator, who luckily gave us access to download the code we needed, although it was close to the thesis delivery deadline. It taught us to never rely on the availability of online material, which is also why a physical copy of the complete source code of both the gen2reader and our MacReader accompanies this document.

4.11. LaTeX

Although this lesson is not directly related to the thesis subject, we mention it here because it might save others from headache. When we first started, neither of us had any experience with Latex, and we had every intention of using Microsoft (MS) Word to write the thesis. It quickly turned out that although a *What You See Is What You Get* (WYSIWYG) editor like MS Word automates a big part of the formatting, it does so at the expense of consistency and lack of control, and in a large document like a Master thesis, that is a problem. We had been warned that Latex has a steep learning curve, and to a certain degree it has, but it has made up for it in consistency. It is better to spend a little extra time and fix something once and for all than to having to fix the same thing again and again, never getting it right. Much can be written about the features

and benefits of Latex, but that is for another time and place. Suffice to say that without Latex, we would have spent much more time on formatting and bibliography, and the result would still have been unsatisfying.

Chapter 5

Related works

In this chapter we will relate the contributions and findings in this thesis to the literature from 2005 to the present. We have chosen this period because it represents the time since the first release of the EPC standard, and includes both the release of the WISP and the gen2reader software.

5.1. Platform choices

In section 1.5 we gave an overview of the distribution between custom made research platforms and ready-made research platforms in the literature. In this section we further support the claim that the gen2reader represents the current state-of-the-art in SDR-based RFID research tools. Tables 5.1 and 5.2 show **the change that happened around 2010, causing a shift away from custom SDR platforms**. The data is based on literature review from the period 2007-2014.

2007-2010	**Custom HW**	**Commercial HW**
Custom SW framework	4 articles 2 research groups	0 articles
Free/proprietary SW framework	1 article	4 articles 3 research groups

Table 5.1: The general division between custom and commercial SDR platforms in literature 2007-2010. Table does not show non-SDR-based platforms. See appendix A for supporting data

2011-2014	**Custom HW**	**Commercial HW**
Custom SW framework	0 articles	1 article
Free/proprietary SW framework	0 articles	13 articles 8 research groups

Table 5.2: The general division between custom and commercial SDR platforms in literature 2011-2014. Table does not show non-SDR-based platforms. See appendix A for supporting data

The trend-shift from custom SDR-platforms to ready-made SDR-platforms coincides with the release of the gen2reader and indeed out of the 8 research groups using a combination of commercial HW and commercial/open-source SW, only one group does not use the gen2reader as a basis [28]. Their research consists of the implementation of a backscatter sensor network, where the sink node exclusively listens and the tags are fed power from an RF signal generator, which explains why the gen2reader would not be applicable. The dominance of the gen2reader software in recent research means that it has had and will continue to have a large impact on the new research being done. This supports the necessity to thoroughly test, evaluate and improve the software.

5.2. Effects of the gen2reader flaws

This section uses examples from the literature to illustrate the effects of using the flawed gen2reader implementation. The gen2reader software is going on 5 years, however out of all the research based on it, none have mentioned the age of the technology stack. Only De Donno et al. [14] comment on the quality of the program and claim to have improved the gen2reader receive chain "*with major upgrades at the synchronization and decoding stages* [14]". They made the source code available online, which made it possible to analyse it. The code has the same lack of structure and documentation as the gen2reader, but based on our experience with forementioned, we found that De Donno et al. [14] have made improvements that adress three of the flaws that we have identified in the gen2reader PHY layer. First, they use a similar approach to ours, calculating a pivot value at 3/4 cycle, for the synchronizer to distinguish between long and short pulses, thereby increasing the tolerance of the synchronizer. Secondly, they address the low sampling resolution at the synchronizer, by calculating an estimation of the BLF based on the average length of 32 pulses, rather than 1 pulse in the gen2reader. Because they only use half the sampling rate, this results in a factor 16 improvement in resolution. Intuitively the increased tolerance introduced through the first improvement should make the second improvement irrelevant, but this would need to be tested. They had also made changes to the decoder. In the gen2reader a perfect correlation with a Miller data-1 symbol would translate to a 1, and any non-perfect correlation to a data-1 would translate to a 0. In De Donno et al. [14]'s gen2listener, they correlate with both a data-1 symbol **and** a data-0 symbol, and whichever has the higher score 'wins'. Theoretically that should result in added robustness to errors. This approach is analogous to the pivot in the synchronizer, with an implicit pivot between the data-1 correlation score and the data-0 correlation score. **The improvements made by De Donno et al. [14] validate our findings and improvements** in two ways. First, the fact that their improvements correct some of the flaws that we identified in the gen2reader backs up our analysis. Secondly, the fact that no other researchers mention the quality of the gen2reader receive chain, and that De Donno et al. [14] did not find the flaws in the normalizer and the rest of the code proves that the flaws in the gen2reader are impacting current research efforts and results. In 2012 De Donno et al. [16] research collision detection using their improved gen2reader. They consider a slot to be a collision slot if the response has failure in either the preamble or the payload, and the SNR is lower than a specific threshold. This is only possible because they have improved the recieve chain, since in the unmodified gen2reader response failures are more common and would therefore result in more false positives. In 2013 De Donno, Ricciato, and Tarricone [12] take another approach to im-

proving the gen2reader clock synchronizer, by completely replacing it with standard GNU Radio clock synchronization blocks. They test three clock recovery methods, zero crossing (ZC), Mueller and Muller (MM), and Polyphase filterbanks (PFB). Figure 5.1 shows the success ratio at different SNR levels for the respective methods. The ZC implementation is a clear loser, however, remember that De Donno, Ricciato, and Tarricone [12] show in [16] that they have not identified the problems in the center block. Therefore **it is unfair to attribute the difference to the zero-crossing method, since at least the Polyphase filterbanks method does not require the signal to be centered first, and an unknown part of the performance difference could be attributed to the flawed center block implementation**. De Donno, Ricciato, and Tarricone [12] have not made the source code from this research available and have not responded to our inquiries, so we cannot verify the details of the implementation.

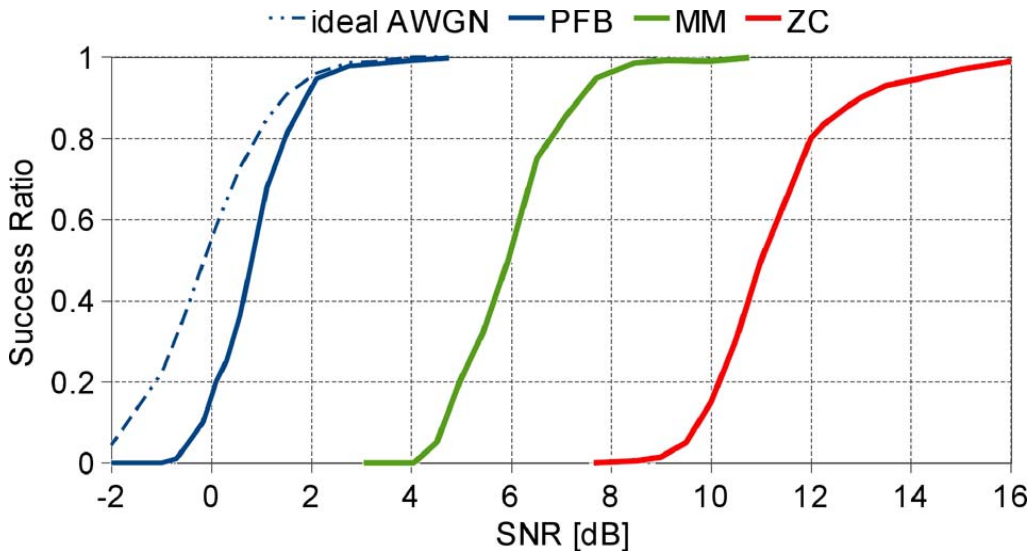


Figure 5.1: The success ratio at different SNR levels for respectively gen2reader zero-crossing (ZC), Mueller and Muller (MM) and Polyphase filterbanks (PFB) clock recovery methods [12]. Also compared to the theoretical optimum for an Additive White Gaussian Noise (AWGN) channel

5.3. The need for dynamic reconfiguration

This section uses examples from the literature to illustrate the relevance of dynamic reconfigurability in RFID research. Dynamic reconfiguration nearly synonymous with *cognitive radio*. Cognitive radios are capable of adapting the system parameters in order to optimize the communication quality. In table 5.3 Fette [22] provides an overview of the *meters* and *knobs*, respectively observable parameters and writeable parameters, that are relevant to a cognitive radio.

The *meters* in the table marked with a star are those that can be collected already in the MacReader. The rest will need additional hardware to measure, but the existing capabilities provide sufficient metrics to make a cognitive RFID interrogator. In 2012 Zhang, Gummeson, and Ganesan [50] designed BLINK, a "*high throughput link layer for backscatter communication*". The need comes from backscatter sensor tags, since these produce

Layer	Meters	Knobs
NET	**Packet delay* **Packet jitter*	Packet size Packet rate
MAC	**Cyclic Redundancy Check (CRC)* Automatic Repeat Request (ARQ) **Frame error rate* **Data rate*	**Source coding* **Channel coding rate and type* **Frame size and type* Interleaving details Channel/**slot*/code *allocation* Duplexing Multiple access Encryption
PHY	**Bit Error Rate (BER)* **Signal-to-noise ratio (SNR)* Signal-to-interference and noise ratio (SINR) **Received signal strength indicator (RSSI)* Pathloss Fading statistics Doppler spread Delay spread Multipath profile Angle of arrival (AOA) Noise power Interference power Peak-to-average power ratio Error vector magnitude Spectral efficiency	**Transmitter power* Spreading type Spreading code **Modulation type* Modulation index Bandwidth Pulse shaping **Symbol rate* **Carrier frequency* Dynamic range Equalization Antenna beam shape

Table 5.3: Examples of observable parameters (meters) and writeable parameters (knobs) of cognitive radio [22]. Note that we have marked the parameters directly applicable to the MacReader with a star

more data than traditional tags. Zhang, Gummeson, and Ganesan [50] design a dynamically reconfigurable link layer architecture which uses RSSI and packet lossto determine whether a tag is mobile, relative to the interrogator, or static. If the tag is mobile, the interrogator dynamically adapts the bitrate, i.e. the Miller encoding, to fit the channel characteristics and switches channel if necessary. Note that Zhang, Gummeson, and Ganesan [50] only designed BLINK, but did not implement it. They did however test the effects of different configurations on different scenarios, confirming that **the ability to dynamically adapt to changing channel conditions will increase the overall throughput**. This is illustrated in figure 5.2 which shows the "*empirically measured optimal goodput mode and their corresponding RSSI and lossrate across several reader-tag distances, placements and times of day*" [50].

In 2014, Liu et al. [29] design an *Adaptive COntinuous Scanning* (ACOS) scheme for scenarios where a single interrogator must scan a tag population spanning a larger area than the interrogator range. They determine that the main challenge of such a sce-

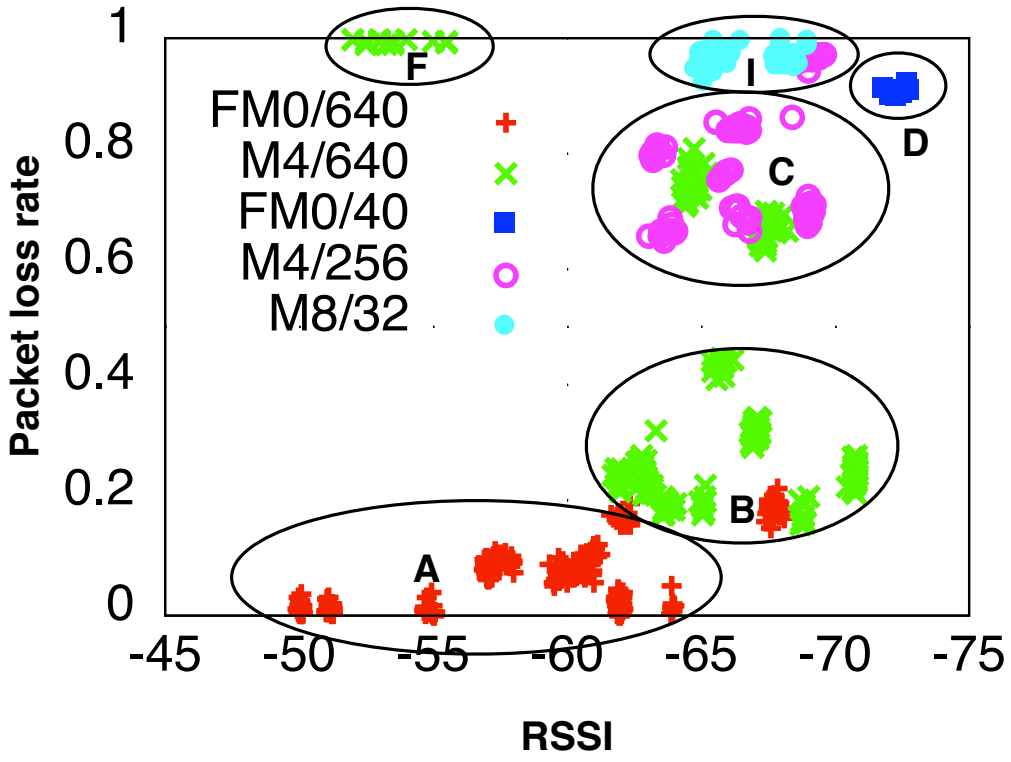


Figure 5.2: Empirical measured link metric map. It shows the optimal bitrate under different RSSI and packetloss. [50]

nario is to avoid redundant reads of overlapping tag populations in consequtive inventory rounds. In order to overcome this, they split the inventory into two phases, an estimation phase and an execution phase. Based on the data collected in the estimation phase **they dynamically select the execution strategy that minimizes the number of redundantly read tags**.

5.4. Latency

In this section we use examples from the literature to illustrate the need for further research into the performance and latency of an SDR-based RFID research platform. Schmid, Sekkat, and Srivastava [40](2007) research the sources of latency on a USRP1 and GNU Radio-based SDR platform. They conclude that there are three primary sources of latency in SDR-based radio systems that need to be reduced for SDR's to be viable for research of wireless short-range protocols. These sources are: bus latency, host processor speed and application implementation. The implication is that the main contribution towards latency from the USRP hardware itself is the choice of bus technology. The USRP1 uses USB 2.0, and the authors suggest that it would be preferable to use a faster bus. They also suggest moving application code with high delay impact, such as filters, to the SDR hardware. Nychis et al. [33](2009) analyse and identify a minimum set of MAC functions that "*must be implemented close to the radio in a high-latency SDR architecture*

to enable high performance and efficient MAC implementations". They implement this principle in a USRP-based setup and test both delay and jitter. They conclude that many MAC protocols cannot be implemented on the host due to the measured delay and jitter. However, the EPC protocol link timing requirements are more forgiving than other MAC protocol specifications towards these limitations. The EPC standard defines the maximum interrogator-to-tag link latency as T2. The length of T2 is from "*the end of the last (dummy) bit of the Tag reply to the first falling edge of the Interrogator transmission*" and must be $20.0T_{pri}$, which translates to 20 subcarrier cycles. The EPC standard defines the subcarrier frequency range to lie between 40 kHz and 640 kHz. In order to convert subcarrier cycles to seconds we use the equations:

$$\text{frequency}(Hz) = \frac{\text{cycles}}{\text{second}} \tag{5.1}$$

$$\frac{\text{seconds}}{\text{cycle}} = \frac{1}{\text{frequency}} \tag{5.2}$$

This gives:

$$RTT_{min_{low}} = \frac{1}{640000} \cdot 20\text{cycles} = 3.125 \cdot 10^{-5}\text{seconds} \tag{5.3}$$

$$RTT_{min_{high}} = \frac{1}{40000} \cdot 20\text{cycles} = 50 \cdot 10^{-5}\text{seconds} \tag{5.4}$$

Which means that $RTT_{min} = \{31.25 : 500\}\mu$s, depending on subcarrier frequency.

Nychis et al. [33] calculate a round trip time between GNU radio and the USRP FPGA to maximum 9000 µs, minimum 289 µs. The majority of the latency variation is introduced between the user and the kernel, which they hypothesise is due to processor scheduling. These numbers show that regardless of the speed of the SDR, the host is the primary source of latency. Their numbers show that it is possible to satisfy the EPC standard link timing requirement of maximum 500 µs at least some of the time, which matches our experience.

The articles were published in respectively 2007 and 2009, so the performance of host hardware can be expected to have improved compared to then. The following paragraphs will cover some of the potential improvements that are worth testing.

Verma and Yu [44](2013) suggest 6 methods for improving the performance of a host system.

Turning off CPU throttling The i7 Core processor on our host has Intel Speedstep option, which is responsible for down-throttling. This can be turned off. The i7 processor also comes with Turbo Boost, which is responsible for up-throttling. This can also be turned off, which may be preferable, since it affects the number of cores that the system can utilize.

Increasing kernel pipe buffer size This refers to the pipe buffer between the kernel and GNU Radio. The authors claim that if the buffer size is too small, it may constitute a bottleneck in the system. The buffer size can be increased or the buffering can be turned off.

Thread priority In linux it is possible to set the priority of a thread, as well as specify the scheduler profile to use. This is important to avoid the issues experienced by Nychis et al. [33], to keep the latency and especially the latency variation low.

Moving code to C The authors propose to move code from MatLab to c, with a similar rationale to how Nychis et al. [33] proposed to move the GNU radio c++ code to HDL on the FPGA. The programming languages of GNU Radio are either Python or C++. C++ is not significantly slower than C, if at all, and with Python libraries like NumPy and SciPy can be used for optimization that brings it close to C efficiency.

Parallelization This is the possibility of taking advantage of the multi-core architecture of modern processors by specifically implementing code with a parallel paradigm in mind. This area may have potential, but also requires thorough programming knowledge. What the authors do not mention however, is that Linux also comes with tools that allow the segregation of processses to separate processor cores. It is worth investigating whether this could lead to improved performance.

GPU programming The authors propose to move parts of a program to the Graphical Processing Unit (GPU). The software framework used in the article is Matlab, and the suggestion to move code makes more sense in that context since MatLab has built-in functions for that. For GNU Radio you could use Compute Unified Device Architecture (CUDA), an NVIDIA Application Programming Interface (API) for GPU's, however the effort required to do so might be better spent moving those same parts to the FPGA.

5.5. Antennas

This section reviews the literature in order to shine a light on some of the possible causes to the issues that we experienced relating to the physical communication. Dobkin [18](2012) explains how directional antennas can be used to increase the power received in the direction of interest. The monopole antennas we have used although not completely isotropic, are a far cry from directional and therefore a lot of the radiated power is wasted. Dobkin [18] explains how using a directional antenna with a power increase of x will increase the range of an antenna by the square root of x. He also explains the importance of matching the polarization of interrogator and tag antennas. Tag antennas are usually dipole and therefore linearly polarized. With linearly polarized interrogator antennas, the "*forward-link-limited read range will be found to be proportional to the cosine of the misalignment angle*" [18] between the interrogator and the tag antennas. Therefore it is important to keep the same alignment between antennas when using linearly polarized antennas. Another observation he makes is that the absolute orientation of a linearly polarized interrogator antenna should be determined based on an evaluation of the environment. A horizontal orientation for example will increase the reflection of the signal from the floor. Circularly polarized interrogator antennas on the other hand do not require the interrogator and tag antenna orientations to be aligned, because the polarization changes over time in a circular fashion. The disadvantage of circular polarization is that at any given point in time the polarization is linear, meaning that the alignment between a circularly polarized antenna and a linearly polarized antenna will increase and decrease in a sinusoidal manner. The result is that although tags of any alingment will be read, the power that they will receive is bounded because the polarization will be misaligned the majority of the time. Finally, Dobkin [18] explains that the most popular RFID interrogator antenna is the patch antenna because it achieves good directivity and is circularly polarized.

In “Antennas and propagation in UHF RFID systems”(2008), Nikitin and Rao [31]attempt to characterize the behaviour and performance of the RFID interrogator and tag signals. They state that for a circularly polarized interrogator and a linearly polarized tag, the "*best achievable polarization efficiency is 0.5, which translates into 70% of maximum possible tag range* [31]".

[14] successfully read tags at transmitter-to-tag distances up to 360 cm, and with tag-to-receiver distances above 12.60 m with a maximum output power of 200 mW. At a maximum output power of 100 mW, the cutoff points for our system are approximately 5 cm for transmitter-to-tag distance and 15 cm for tag-to-receiver distance.

Chapter 6

Conclusion

In this thesis we first established the open source SDR-application, known as the gen2reader, as the current state-of-the-art research tool for SDR-based RFID protocol research. We based this finding on the research platforms used in published works from the past decade. Our findings show that the RFID research community readily adopted this new platform when it was released and that it is still the basis of most new SDR-based RFID protocol research. Through analysis of the gen2reader source code we identified a wide range of problems, from scientifically unsupported design decisions, over unmaintainable code, to serious flaws that would affect the validity of research results if not found and corrected. These problems are undocumented in the research community, which leads us to believe that researchers are unaware of them and are possibly not critical enough in their approach towards the tool. Only one research team have mentioned improving the gen2reader, but use the term *improvement* rather than *correction*, which would be more fitting, and warn other researchers of the issues. Based on analysis of the source code from this team, they have found and corrected some flaws, but missed several others. In order to provide the research community with a trustable research tool, we have redesigned and reimplemented the gen2reader into the MacReader application. This application is based on well-documented and well-tested design decisions, as well as an architecture built for modifiability, extendability and dynamic reconfigurability. By our estimations the tool should cut the setup time to a few hours, where it took us many weeks to find the correct combination of outdated technology stack versions and host configuration for the gen2reader software. In addition, the time needed to making changes to the software has been cut considerably because the software has been designed based on the principles of cohesion and loose coupling, meaning that local changes only require knowledge of local code, where in the gen2reader, local changes required deep knowledge of the whole project source code due to the implicit connections between all parts of the software. Finally, we have also cut development time and complexity by providing a guide to import the MacReader out-of-tree module into Eclipse, giving access to a graphical debugger and many other development tools. We have proven the useability and correctness of the MacReader through PHY layer experiments and identified several interesting characteristics of the EPC standard protocol, such as a linear relationship between the RTcal symbol length and the tag response delay (T1), and the true tolerances of RTcal symbol length misconfiguration. We identified a characteristic of the WBX 50-2200 MHz daughterboard that meant that we could reduce interrogator response delay (T2) by reducing the length of the continuous wave (CW) transmitted after each command to 1 single sample. We

have also tested and documented the effects of shielding on the monopole antennas used for this project and mapped the relationship between the USRP gain setting and the output power of the WBX 50-2200 MHz daughterboard. We have also found that a considerable part of the interrogator response delay (T2) can be attributed to GNU Radio input buffer overshoot, which has not been mentioned in any other latency research and is a very interesting possiblity for future optimization.

Besides the possibilities for future work covered in section 2.4, there is plenty of opportunity for future work with the MacReader research tool. In section 1.6 we identified a shortlist of SDR hardware that, on paper, should support the features needed for RFID protocol research, including native integration with GNU Radio. Some of these are considerably cheaper than the USRP N210+daughterboard combination, so testing whether these tools are valid alternatives in praxis could potentially make the whole solution cheaper and thereby an even more attractive platform. To our best knowledge, no research has been done on the different sources of latency within GNU Radio, only on a GNU Radio application as a whole. It would be a very valuable contribution to identify and profile the different sources of latency in GNU Radio and to identify methods for mitigation. Another opportunity for future research is into host latency and methods for latency reduction. The latest research article for host latency optimization we have found is Schmid, Sekkat, and Srivastava [40] from 2007, and technology has changed much since then, especially with the advent of multi-core processors. In 2013 Verma and Yu [44] suggest a number of potential methods for host latency reduction in an SDR-based system, but they are only suggestions, with no test results for support.

Appendices

Appendix A

Overview of SDR-based platforms used for RFID research

The table contains an overview of the hardware (HW), software (SW) and tag platforms used for SDR-based research. The data is based on a literature review of literature from the period 2007 to 2014.

Author(s)	Platform	Year	Protocol layer(s)
Schmid, Sekkat, and Srivastava [40]	**SDR**: USRP **SW**: GNU Radio **Tag**: None	2007	MAC
Buettner and Wetherall [8]	**SDR**: USRP **SW**: GNU Radio **Tag**: Alien 9460-02	2008	PHY MAC
Angerer et al. [3]	**SDR**:ARC board **SW**: C++ and MatLab **Tag**: not specified	2008	PHY MAC
Tung and Jones [42]	**SDR**: Xilinx Spartan III 400 FPGA, an Actel Fusion AFS90 flash-based FPGA, and a Xilinx Cool- runner II CPLD **SW**: Custom **Tag**: not specified	2008	PHY
Angerer and Langwieser [1]	**SDR**: Custom **SW**: Custom **Tag**: not specified	2009	PHY
Nikitin and Rao [30]	**SDR**: National Instruments LabVIEW-controlled PXI RF **SW**: Labview **Tag**: Avery Dennison AD-222, AD-224, AD-223 and Alien ALN-9640	2009	PHY

Buettner and Wetherall [7]	**SDR**: USRP2 **SW**: GNU Radio **Tag**: Alien 9460-02	2010	PHY MAC
De Donno, Ricciato, and Catarinucci [11]	**SDR**: USRP **SW**: GNU Radio **Tag**: not specified	2010	PHY MAC
Angerer, Langwieser, and Rupp [2]	**SDR**: ARC Board **SW**: C++/MatLab **Tag**: not specified	2010	PHY
Catarinucci et al. [10]	**SDR**: USRP **SW**: GNU Radio **Tag**: Alien 9640	2011	PHY
Buettner and Wetherall [6]	**SDR**: USRP **SW**: GNU Radio **Tag**: WISP	2011	PHY MAC
De Donno, Tarricone, and Catarinucci [13]	**SDR**: USRP **SW**: GNU Radio **Tag**: WISP	2012	MAC
De Donno et al. [15]	**SDR**: USRP **SW**: GNU Radio **Tag**: WISP	2012	MAC
Wang et al. [46]	**SDR**: USRP **SW**: GNU Radio **Tag**: UMASS MOO	2012	MAC
Briand, Albert, and Gurjao [4]	**SDR**: USRP1 **SW**: GNU Radio **Tag**: USRP1	2012	PHY MAC
Valenta and Durgin [43]	**SDR**: USRP n200 **SW**: Custom C **Tag**: Custom	2012	PHY MAC
De Donno et al. [16]	**SDR**: USRP **SW**: GNU Radio **Tag**: WISP	2012	PHY MAC
Kimionis, Bletsas, and Sahalos [28]	**SDR**: USRP **SW**: MatLab **Tag**: Custom	2012	PHY MAC
Zheng and Li [52]	**SDR**: USRP n210 **SW**: GNU Radio **Tag**: WISP	2013	MAC
Wang et al. [47]	**SDR**: USRP **SW**: GNU Radio **Tag**: MBTA Charlie subway card, Alien Squiggle, UMASS Moo	2013	MAC
Wang and Katabi [45]	**SDR**: USRP n210 **SW**: GNU Radio **Tag**: Alien Squiggle	2013	PHY

De Donno, Ricciato, and Tarricone [12]	**SDR**: USRP **SW**: GNU Radio **Tag**: Impinj Thinpropeller	2013	PHY MAC
Zhang and Ganesan [49]	**SDR**: USRP **SW**: GNU Radio **Tag**: UMASS MOO	2014	MAC

Appendix B

gen2reader Software Installation Guide

Notes: Where version numbers are specified, make sure to install that exact version. Where values are marked with "<" or ">" like <example> replace the entire thing, including the "<" and the ">" with whatever it is referring to.

1. Install Ubuntu 12.04.5 32bit
   a) Download
      http://releases.ubuntu.com/12.04/ubuntu-12.04.5-desktop-i386.iso
   b) Burn image to CD/DVD or create a bootable USB drive
   c) Boot from selected media and install
2. Install dependencies
   a) apt-get update
   b) apt-get install libboost-all-dev swig libcppunit-dev git gawk automake
3. Setup network:
   a) Run ifconfig and note down the HWaddr of the eth0 interface
   b) To avoid having the networkmanager disconnect the eth0 interface because it cannot find a dhcp server, add the following to /etc/NetworkManager/NetworkManager.conf
      [keyfile]
      unmanaged-devices=mac:<HWaddr>
   c) Open /etc/network/interfaces and change eth0 to:
      auto eth0
      iface eth0 inet static
      address 192.168.1.1
      netmask 255.255.255.0
   d) It may be necessary to restart the eth0 interface. (Ignore any warnings/error messages):
      ifdown eth0
      ifup eth0

4. Install UHD version 003.005.004-62:
   a) Download `http://files.ettus.com/binaries/uhd_stable/uhd_stable_2013-11-13/uhd_003.005.004-62-stable_Ubuntu-12.04-i686.deb`
   b) Double click file
5. Identify the ip address of the USRP (necessary for next step)

   The ip address can be found with

   `uhd_find_devices`

   or set with

   `./<uhd-install-path>/lib/uhd/utils/usrp2_recovery.py--ifc=<Networkinterface> --new-ip=<newipaddress>`

   The ip address should conform to the netmask specified in step 3, i.e match the eth0 interface address on the first 3 'digits' (192.168.1.xxx)

   After setting the ip address of the USRP it is necessary to restart the USRP to make the change permanent.
6. Update USRP n210 firmware
   a) Download firmware

      There are 2 ways to get the firmware files
      - The latest version can be downloaded through the `uhd_images_downloader` command. This will put the images in: `<uhd-install-path>/share/uhd/images/`
      - The version that is confirmed to work, directly from `http://files.ettus.com/binaries/maint_images/uhd-images_003.007.002-release.zip`. For simplicity, extract this to <uhd-install-path>/share/uhd/images/

      There are 2 files: one for the firmware and one for the FPGA
   b) Transfer the images to the USRP

      ```
      python usrp_n2xx_net_burner.py \
      --addr="<usrp-IP-address>" \
      --fw="<uhd-install-path>/\
      share/uhd/images/usrp_n210_fw.bin" \
      --fpga="<uhd-install-path>/share/uhd/\
      images/usrp_n210_<revision number>_fpga.bin" \
      "--dont-check-rev"
      ```

      See step 5 for the USRP ip address

      See the back of the USRP for the revision number

      The currently installed firmware version can be found with:

      `uhd_usrp_probe`
   c) Restart usrp
7. Install GNURadio version 3.6.2-38:
   a) Download `http://files.ettus.com/binaries/gnuradio/gnuradio-stable_2012-12-07/gnuradio_3.6.2-38_Ubuntu-12.04-i686.deb`

b) Double-click file

8. Copy the `gruel_common.i` file to directory /usr/include/gnuradio/swig

```
cp /usr/include/gruel/swig/gruel_common.i \
/usr/include/gnuradio/swig
```

9. Install usrp2reader:

   a) Download https://github.com/yqzheng/usrp2reader/archive/master.zip

   b) Extract somewhere

   c) Run bootstrap in the extracted directory (../usrp2reader-master/rfid/): ./bootstrap

   d) Run: ./configure && make && make install

   e) To avoid "ImportError: libgnuradio-rfid.so.0: cannot open shared object file: No such file or directory", run ldconfig

10. Start the program

    python <usrp2reader-install-path>/`gen2_reader`

11. Configure system for performance

    a) Give uhd permission to set real-time threading priority (if you get a warning when you run the program)

       i. Identify a group that the user is a member of by running
          groups

       ii. Input into /etc/security/limits.conf
          `@<group-name>  - rtprio  99`

       iii. Log out and log back in

    b) Set the daughterboard TX and RX serial number (any number works)

       `./<uhd-install-path>/utilities/usrp_burn_db_eeprom--unit=TX--ser=<serial>`

       `./<uhd-install-path>/utilities/usrp_burn_db_eeprom--unit=RX--ser=<serial>`

    c) Calibrate the transmitter and receiver

    d) Run GNURadio volk optimizations (takes a very long time on 32bit OS)

       `volk_profile`

Appendix C

MacReader Software Installation Guide

1. Install Ubuntu 14.04 LTS
2. Install GNURadio via the build-gnuradio script.

   ```
   $ wget http://www.sbrac.org/files/build-gnuradio
   $ chmod a+x ./build-gnuradio
   $ ./build-gnuradio
   ```

   The last command can take an hour, maybe longer. *Do not use sudo, but run the last command as a sudoer.*
3. Setup network
   a) Ubuntu's network-manager service will interfere with the usrp ethernet connection unless instructed to ignore it. To do that we first need to get the hardware address of the eth0 interface:

      ```
      $ ifconfig | grep 'eth0.*HWaddr'
      ```

      This will return something like below:

      ```
      eth0 Link encap:Ethernet HWaddr 90:1b:0e:3d:ab:c9
      ```

   b) Make a note of the *HWaddr*.
   c) Open */etc/NetworkManager/NetworkManager.conf* and add the following, but substitute <HWaddr> with your specific HWaddr (*not* the one from the example above):

      ```
      [keyfile]
      unmanaged-devices=mac:<HWaddr>
      ```

   d) Save the file.
   e) Next, open */etc/network/interfaces* and change eth0 to:

      ```
      auto eth0
      iface eth0 inet static
          address 192.168.1.1
          netmask 255.255.255.0
      ```

This will assign a static ip-address to the eth0 network interface. Any ip-address assigned to the USRP itself must be in the same subnet, i.e. only differ on the last number.

f) Save the file.

g) It may be necessary to restart the eth0 interface. (Ignore any warnings/error messages)

```
$ ifdown eth0
$ ifup eth0
```

Appendix D

Detailed documentation of gen2reader

D.1. Introduction

The RFID Tools library is an SDR-based RFID interrogator implementation for USRP1, published by Buettner in 2010 and modified for use with USRP2 hardware in 2013, by Yuanqing Zheng. It is written in a combination of C++ and Python, in order to combine the performance of C++ with Python's ease-of-use. The gen2reader application is written in Python, but it is only responsible for initializing variables and objects and wiring the GNURadio components together. The actual protocol implementation is found in the C++ classes. In order to be able to use the C++ classes from the Python application, GNURadio uses a tool called Swig to map the C++ library to Python interfaces. Both the Python program and the C++ library uses the GNURadio framework and toolkit to interface with the SDR hardware. Since the RFID Tools library is built upon the GNU-Radio framework, there are a few GNURadio concepts that it is essential to understand before diving into the library source code.

D.1.1 GNURadio

GNURadio is a software development toolkit for developing software-defined radio applications. It consists of an API for developing signal processing components, in GNU-Radio terminology referred to as 'blocks', a set of radio hardware drivers, and a graphical user interface called GNURadio Companion (GRC). GRC is intended to allow developers to construct SDR applications as flow-graphs constructed from individual blocks chained together with the output from one block feeding the input from another. GNURadio comes with many pre-made blocks and it is possible to build your own custom blocks if the existing ones are insufficient. A GNURadio application can be written either directly as a python script, or constructed in GRC, which then generates the python script from the flow-graph. Besides the many signal processing blocks, GNURadio also provides sink blocks and source blocks, that allow you to read a signal stream from, or send it to, SDR hardware or the file system. To properly define the term block, we could describe it as a software component that consists of a set of configuration parameters, a signal input stream and/or output stream. In GRC, the input and output stream interfaces are referred to as ports.

This document contains documentation of Buettner's custom GNURadio blocks in the order they appear in the flow-graph:

- command gate
- center
- clock recovery
- tag decoder
- reader

Every GNURadio block must implement the 'general_work' method. This method is inherited from the gr_block class, and maps the port concept through an input argument and an output argument. The configuration parameters are mapped through the block constructor. Finally, most of the RFID Tools blocks also implement another gr_block method called 'forecast'. This method is used for ensuring that a block can supply a large enough output stream when one is requested, by telling the block that functions as input, approximately how much data it will need to satisfy an output request.

The gen2reader application The gen2_reader.py file contains the interrogator program. It is structured according to the typical pattern of a GNURadio program:

- Variable declarations
- Block declarations
- Connecting the blocks into a graph
- Running the program

The only deviation from this pattern is the addition of a log method, which extracts all the log messages in memory from the Reader block when the program is terminated, and prints them to the console and to a file.

D.1.1.1 Command Gate

This block guards the rest of the processing chain from wasting system resources processing irrelevant signals. It achieves this by analysing the input stream and performing pattern recognition, trying to recognize a tag response. The Reader block will reset the Command Gate block when a tag response can be expected, which means that the Command Gate reduces the probability of overhearing other tag-reader exchanges. The Reader block also informs the Command Gate if the transmitted command was NAK, in which case no response can be expected. Finally, the Reader block tells the Command Gate how many samples the tag response is expected to consist of, which is how the Command Gate determines how long the gate should be open for. Basically, the command gate looks for the the reader command by counting the number of low pulses in the command. It recognizes a low pulse by comparing the magnitude of each input sample with a threshold value.

D.1.1.2 Center

Description This block converts the incoming signal to an array of values of either 2 or -2, depending on whether the corresponding value in the incoming array is higher or lower than the average over 32 samples.

D.1.1.3 Clock Recovery

Description This block is needed because the received signal may suffer from the effects of clock skewing, which causes a received signal to be stretched or constricted at different points in the time-domain compared to the original signal. The clock recovery algorithm first identifies all zero-crossings in the signal, i.e where the signal switches from 0 to 1 or 1 to 0. It then restores the original signal by replacing short sequences of bits by a single bit of the same value, and replacing longer sequences of bits by two bits of the same value. The reason that short and long sequences are treated differently comes from the use of Miller encoding on the signal, where a repeated bit indicates a one, and switches between 0 and 1 to indicate zero.

D.1.1.4 Tag Decoder

Description This block performs Miller decoding of the received command. The decoding is performed through correlating the received signal in sequences with a Miller encoded 1-vector. If the correlation score is higher than 0.6, the sequence is interpreted as a one, and if it is lower, it is interpreted as a zero. Buettner takes the absolute value of the correlation score because a Miller encoded 1 can be represented either by the sequence 10100101 or its inverse, and if he did not take the absolute value, the correlation score for the inverse signal would be negative.

D.1.1.5 Reader

Description The most important block in the gen2reader is the Reader block. This block is in charge of processing tag responses and constructing the reader commands. It contains the interrogator state machine which determines the reader-side MAC layer behaviour. The reader block only fully implements 4 interrogator commands: Query, QueryRep, ACK and NAK. There is only a single command missing to have all the inventory commands, and that is the QueryAdjust command. In addition to the Inventory commands the block consists of two Access commands: Req_RN and Read. These however are incomplete and have been switched off by Buettner himself. Therefore they will not be covered in this documentation.

D.1.2 How the state machine is implemented

The state machine is implemented as a loop with conditionals, where the current state of the reader is determined by the last command to be transmitted. The loop is not apparent in the code, but is instead imposed by the gnuradio block design, which periodically calls the general_work() method.

D.1.3 Implementation details

The Reader block source code can be divided into the following categories:

Constructor The constructor constructs the framesync, preamble and commands in advance, except the ACK command which depends on the RN16 tag response.

Gnuradio block methods These are general_work and forecast, whose general purpose has been described in section nn. In the reader block, general_work also contains the state machine. Command generation These methods have a method signature like gen_<command-name>_cmd. These methods construct their corresponding messages.

Command message wrapping These methods have a method signature like send_<command-name>. These methods do not send the commands, contrary to what their names would lead you to believe. Instead they wrap the commands in the gr_message type, add the continuous wave before and after the command and add the whole thing to the out-queue.

Command gate coordination These are set_num_samples_to_ungate and ctrl_q. The first method predicts the number of samples to ungate based on the expected next response from the tag. The calculated value is stored in a global variable that the command gate block has access to. The second method ...

D.1.3.1 Helper methods

update_q Updates Q, the number of slots specified by a Query

check_crc Verifies the CRC of the input array.

send_another_query Returns true if another Query command should be sent, based on detected collisions in the slots and whether the reader has been configured for multiple cycles.

Log methods These methods are used for storing and retrieving log messages. Log messages are stored in memory to save time.

D.1.4 Constructing a command

In order to understand how commands are constructed, it is necessary to look at how they are specified in the EPC standard. All commands consist of a sequence of one or more fields. The first field specifies the command, and the rest are used as arguments to specify different aspects of the communication, depending on the purpose of the command. Out of the four commands that are implemented in the RFID Tools library, the Query command is the most complex.

The Query command consists of 9 fields as can be seen in Table xx. We can divide these fields up depending on when the data is available. We distinguish between Constant fields, which are always the same, configuration fields, which are determined as part of the system solution design, and dynamic fields, which depends on the immediate communication context. The EPC specification does not make this distinction, and the concrete distinctions made here are only valid for this specific implementation. That is because several fields that are determined by system configuration in the RFID Tools library, would be considered dynamic fields in a system that implements the complete specification.

The Query Command fields are:

- Constant fields:
  - Command field

  - DR: Data rate
  - M: Miller encoding (WISP tags require M=4)
  - TRext: TODO
  - Session: The Session field is used when multiple readers concurrently Inventory tag populations, to coordinate the tag responses to each reader. Each reader specifies a different session. This means that in a single reader environment the Session field can be treated as a constant.

- Dynamic fields

  - Sel (Select) The Select field is used when a tag population has been divided into subsets, and can be set to include all tags, include a subset, or exclude a subset. In the RFID Tools project, the Select commands have not been implemented, so the Select field is a constant.
  - Target The Target field is based on the inventoried flag, which is stored on the tag. This flag is used to split an Inventory session into multiple time periods. When the Session field is set to 00, the specification dictates that the inventoried flag is reset between each power-up of the tag, which in effect means that it is turned off. Therefore, in the RFID Tools implementation, the Target field is a constant.
  - Q Q specifies the number of slots for tag responses. When the size of a tag population is known this can be set to a constant, but when the tag population size is unknown, a cardinality estimation algorithm is used to calculate the optimal number of slots for the estimated tag population size. Depending on the purpose of the research performed this field can be a constant, or it can be dynamically set at runtime. The RFID Tools implementation contains a method, update_q, which is used for dynamically setting Q if cardinality estimation is turned on in the configuration.
  - CRC This field contains the CRC hash of all the other fields of the command, and must therefore be dynamically calculated for each Query command sent.

The other three commands implemented in the RFID Tools project are QueryRep, ACK and NAK. The NAK command is constant, and so is the QueryRep command, although it contains a Session field, which would make it dynamic in a multireader setting. The last command, ACK, consists of a Command field, which is constant, and a field containing the RN16 received from the tag.

Constructing a command in the RFID Tools library consists of: Constructing an array of bits, although the specific data type used is float. This array consists of the concrete values for each field that constitute the command. The command array is stretched according to the specified sample rate, which consists of replacing each occurrence of a one with a sequence of ones, and each zero with a sequence of zeros. The framesync or preamble is prepended to the message. The signal is padded with a continuous wave, after and/or before. The final message is passed on to the next block.

D.1.4.1 Updating the slot-count

The reader block implements a method called update_q, which can be directly mapped to the algorithm illustrated in the following figure from the EPC standard. Q_fp is a float

representation of Q which is rounded to an integer value after each Query round. Delta is a tuning parameter with a value between 0.1 and 0.5. In the RFID Tools library delta is set to 0.5. Q_fp is a configuration variable and is set to the estimated number of tags in the population. The default value in the RFID Tools library is 2.

D.1.5 The EPCGlobal Gen2 Class1 timing requirements and the RFID Tools library

The EPC Physical layer specification mandates a set of timing values for both the physical layer and the link (MAC) layer. In this section we will cover the relevant timing values from the specification and relate them to the specific timing values used in the RFID Tools library.

D.1.5.1 Physical layer timing

Reference timing interval All timing values in the EPC standard are defined either directly or indirectly by a reference timing interval called Tari. The EPC standard restricts Tari to the range 6.25Î$\frac{1}{4}$s to 25Î$\frac{1}{4}$s. In the RFID Tools library Tari is set to 24 us.

Data symbols EPC mandates the use of Pulse interval encoding (PIE), which means that transmitted 1's and 0's distinguished by the time between low amplitude periods. Transmitted 0's are referred to as data-0, and 1's are referred to as data-1.

- data-0: The length of a data-0 symbol is equal to Tari, and consists of a shorter period of high amplitude followed by a period of low amplitude referred to as Pulse-Width (PW).
- data-1: The lenght of a data-1 symbol is between 1.5 Tari and 2.0 Tari, and consists of a longer period of high amplitude followed by PW.
- PW: The length of PW is restricted to the range MAX(0.265*Tari, 2):0.525*Tari

In the RFID Tools library the length of the symbols are:

- data-0 = Tari = 24 us
- data-1 = 48 us
- PW = 12 us

Framesync and preamble Each command that the reader transmits must be preceded by either a preamble or a framesync. The timing values of the framesync and preamble symbols are important because the tag uses these to determine the communication deadlines and the backscatter link frequency, i.e the subcarrier frequency on which to reply to the interrogator.

The R=>T Frame-Sync consists of a $12.5 \pm 5\% us$ delimiter of low amplitude, a data-0 and an R=>T calibration symbol (RTcal) with a length in the range $2.5 * Tari : 3.0 * Tari$.

The R=>T Preamble consists of an R=>T Frame-Sync and a T=>R calibration symbol (TRcal) with a length in the range $1.1 * RTcal : 3.0 * RTcal$

In the RFID Tools library the length of the symbols are:

- delimiter = 12 us

- $RTcal = 60us + PW = 72us$
- $TRcal = 75us + PW = 87us$

TRcal implications The T=>R calibration time in combination with the divide ratio (DR) that is specified in the Query command, is used by the tag to calculate the backscatter link frequency (BLF).

$$BLF = 1/T\_pri = DR/TRcal \tag{D.1}$$

When we calculated this for the gen2reader library settings it became clear that something was not adding up. We calculated the BLF to be:

$$BLF = 8/(87 * 10^{-}6) = 91954Hz \tag{D.2}$$

This does not correspond to the BLF of 256 kHz used by the wisp tags, but it turns out that the wisp tags ignore the interrogator instructions and have the 256 kHz BLF hardcoded in the firmware.

RTcal implications The RTcal time is used by the tag to distinguish data-0 symbols from data-1 symbols. The length of the RTcal symbol must be the sum of the length of a data-0 and a data-1. The values in the RFID Tools library adhere to this requirement. When the tag receives the RTcal symbol either in the preamble or the framesync, it calculates a pivot which is half the length of the RTcal symbol, and afterwards interprets all symbols received that are longer than the pivot to be data-1 symbols and all symbols that are shorter than the pivot as data-0 symbols.

D.1.5.2 Link timing

The link timing refers to the timing of the reader-to-tag and tag-to-reader communication flow. There are two timing values that are especially relevant to the RFID Tools library and wisp communication: the delay between a reader command and a tag response, and the delay between a tag response and a reader command. The reason for this is that a standards adherent tag will treat a reader communication as failed if the reader takes longer to reply than mandated in the specification, and vice versa for the reader. In this appendix we will refer to the delay between a reader transmission and a tag reply as Tag Reply Delay (TRD) and the delay between a tag reply and a reader transmission as Reader Reply Delay (RRD).

Tag Reply Delay The timing restriction for TRD is referred to as T1 and can be found in Table oo. Tpri refers to the length of one BLF cycle, i.e 1/BLF, and FrT refers to the frequency tolerance, which for 256 kHz BLF is ±10%. For the RFID Tools library this means that TRD is restricted to be between:
$Tpri = 1/256000 = 3.90625us$
$TRD\_min = MAX(72us, 10 * (Tpri)) * (1 - 0.1) - 2us = MAX(72us,\ 39us)) * (0.9) - 2us = 72us * 0.9 - 2us = 62.8us\ TRD\_max = 72us * 1.1 + 2us = 81.2us$

Reader Reply Delay The timing restriction for RRD is referred to as T2 and can be found in Table oo. For the RFID Tools library this means that TRD is restricted to be between: $RRD\_min = 3 * Tpri = 11.71875us\ RRD\_max = 20 * Tpri = 78.125us$

Other link timing values There are several other timing values in the specification, but they relate to interrogator commands that are not implemented in the RFID Tools library, and are therefore not covered in this document.

Appendix E

Working with the gen2reader source code

E.1. Modifying the program

1. Download or otherwise acquire the gen2reader source code
2. Locate the source code of the gen reader. The source is located in the following directory: <unzipped directory>/rfid/lib/
3. Make the desired changes to the *.cc and *.h files.
4. Rebuild the library
   a) In the terminal, navigate to <unzipped directory>/rfid/
   b) Run

      ```
      $ sudo make & make install
      ```

      This builds and installs the application

E.2. Debugging the source code

To debug the code that runs behind the gen_reader.py one can use the GDB debugger. GDB comes as part of the Ubuntu distribution.

E.2.1 Start the GBD debugger

Run the following command in the terminal:

```
$ gdb python<Version> \newline
```

or

```
$ gdb <path to python>
```

This will start the python environment in the GDB debugger

E.2.2 Set a breakpoint in the program

Run the following command in the terminal:

```
$ break <class>::<method>
```

or

```
$ break <filename without extension>:<line number>
```

You will get the following message:

```
Function "<class>::<method>" not defined.
Make breakpoint pending on future
shared library load? (y or [n])
```

```
Type 'y' and press enter.
```

When the program reaches your breakpoint you will get the following message:

```
Breakpoint 1, <class>::<method> (this=<address of
the object(not sure)>) at <class>.cc:<line number>
```

E.2.3 Run the program

```
$ run <unzipped directory>/gen2\_reader.py
```

E.2.4 Print the values of variables in the GDB debugger

Run the following command in the terminal:

```
print <variable name>
```

E.3. Example

1. Open the rfid_reader_f.h file with a text editor. Inside the public scope of the rfid_reader_f class, add the following line:

   ```
   void myprint();
   ```

2. Open the rfid_reader_f.cc file and add the following lines (remember to include iostream):

   ```
   void rfid\_reader\_f::myprint()
   {
           std::cout<<"Hello World";
   }
   ```

3. Save the changes to the rfid_reader_f.h and rfid_reader_f.cc.

4. Call the *myprint()* method inside the the *start_cycle()* method by doing the following:

   a) Find rfid_reader_f::start_cycle() in the rfid_reader_f.cc file).

b) Add the following line:

```
myprint();
```

5. Now it is time to build the source code again. Navigate to:

   ```
   <unzipped directory>/rfid/
   ```

6. Run the following commands in the terminal:

   ```
   $ sudo make & make install
   ```

7. Open the GDB debugger in the terminal with the python environment, but replace the 2.7 with your version of python.

   ```
   $ gdb python2.7
   ```

8. Add your breakpoint to the program and confirm with 'y'

   ```
   $ break rfid_reader_f::myprint
   ```

9. Run the program

   ```
   $ run <unzipped directory>/gen2\_reader.py
   ```

   Now you should see something like the following line:

   ```
   Breakpoint 1, rfid\_reader\_f::myprint
   (this=0x9054a50) at rfid\_reader\_f.cc:378
   ```

   This tells you, that the program has stopped at your breakpoint and in turn, that you have successfully added a myprint method to the rfid_reader_f class.

Appendix F

Impinj Speedway R220 setup and communication with the WISP

The Impinj Speedway R220 is a commercial RFID interrogator. It comes with several software tools such as the 'MultiReader for Speedway', a program that can identify tags by reading the EPC values, and the 'Octane SDK', which is a Software Development Kit (SDK). The Impinj can only read the WISP 4.1 tag when the *Reader Mode* is set to 'Dense Reader (M=4)'.

F.1. Impinj Speedway R220 hardware setup

1. Connect the Impinj Speedway R220 to the PC with an ethernet cable.

Figure F.1: The Impinj Speedway R220 connected via ethernet to the host

2. **Make sure that the cable is in the 'ETHERNET' port, and NOT the 'CONSOLE' port.**

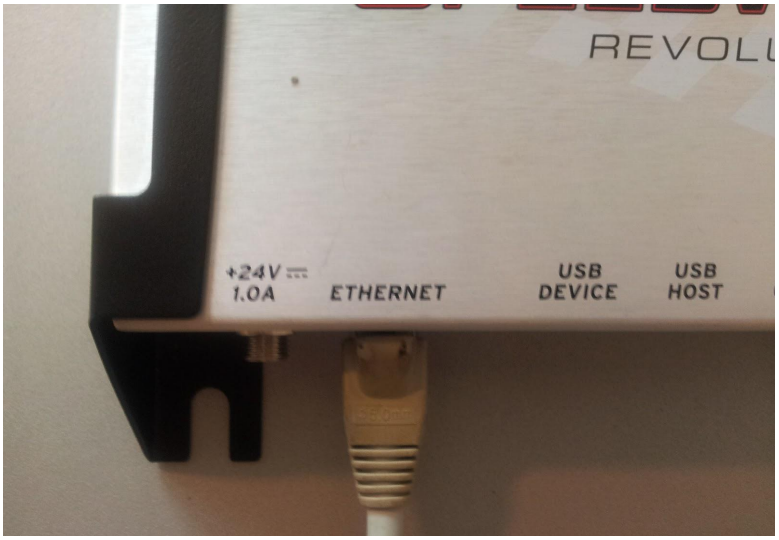


Figure F.2: The ethernet cable correctly connected to the Impinj Speedway R220's ethernet port.

3. Connect the antenna to one of the "ANT" ports (Here it is connected to ANT 1)

Figure F.3: The Caen RFID WANTENNAX005 antenna connected to Impinj Speedway R220's ANT1 port.

4. Connect the power adapter to the '+24 V' port

Figure F.4: The power adapter connected to Impinj Speedway R220's '+24 V' port.

F.2. MultiReader Setup

1. Download the MultiReader from `https://support.impinj.com/hc/en-us/articles/202755258-Impinj-MultiReader-Software`

2. Go to Settings→Configure ...

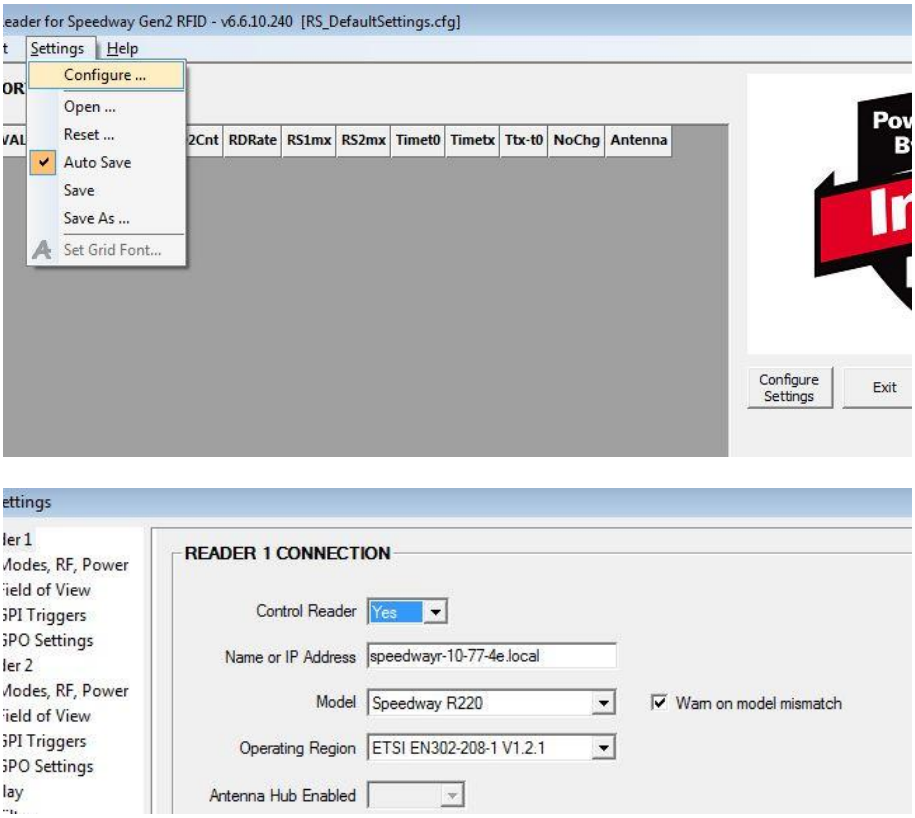


Figure F.5: This figure shows the configure window in the MultiReader program.

a) Type in the 'Name or IP address' The 'Name or IP address' is speedwayr-xx-xx-xx.local, where the xx-xx-xx values are the last 6 values to the right of 'MAC 00:(See figure:F.6)' (Here the xx-xx-xx are 10-77-4e)

Figure F.6: The 'MAC 00' of the speedway, here the xx-xx-xx is 10-77-4e.

b) Select 'Speedway R220' as the model of the speedway. (Click 'Warn on model mismatch' it will tell you, if you have selected the wrong model)

c) Select your Operating Region. (In Europe it is ETSI EN302-208-1 V1.2.1)

d) Click 'Apply' to connect to the speedway reader.

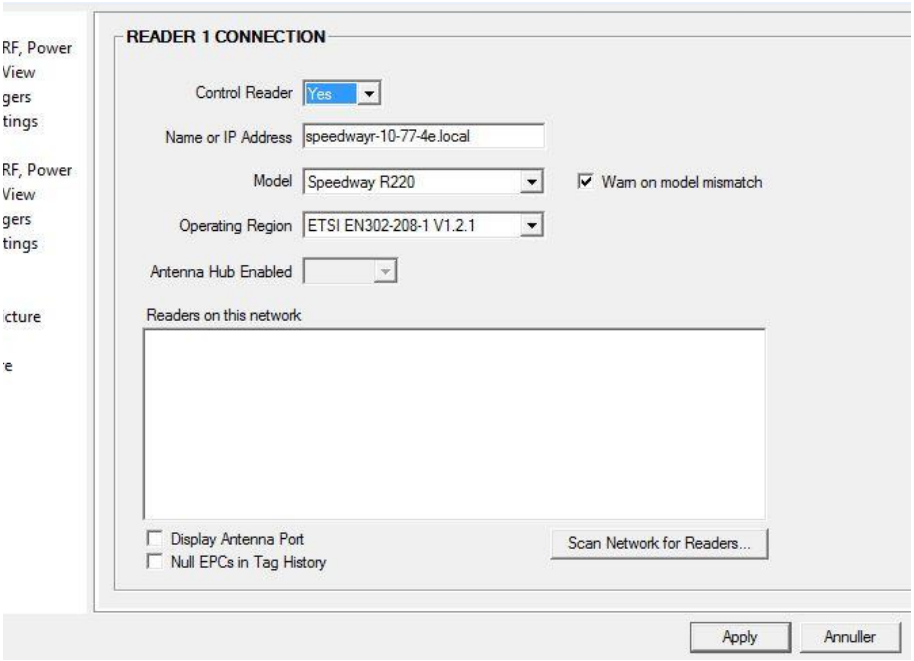


Figure F.7: The connection setup in the MultiReader program.

3. To run the program click 'START Inventory Run'

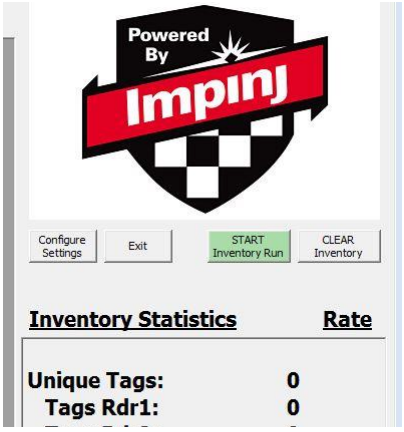


Figure F.8: The start button in the MultiReader program.

When the Impinj Speedway identifies a tag, it shows the EPC value of the tag.

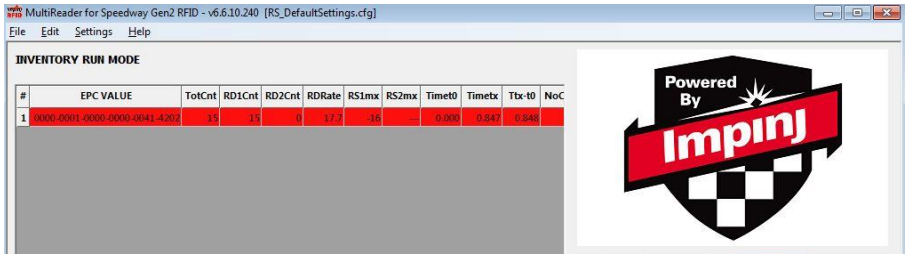


Figure F.9: The EPC value of a tag that has been identified by the Impinj speedway.

F.3. Ping the Impinj Speedway R220

Another way to check, if the Impinj Speedway R220 is connected to your PC, is to ping it.

1. Open a terminal(Like run 'cmd.exe')
2. Type the following: **ping speedwayr-xx-xx-xx.local** (the xx-xx-xx values are the last 6 values to the right of "MAC 00:" label, see figure F.6)

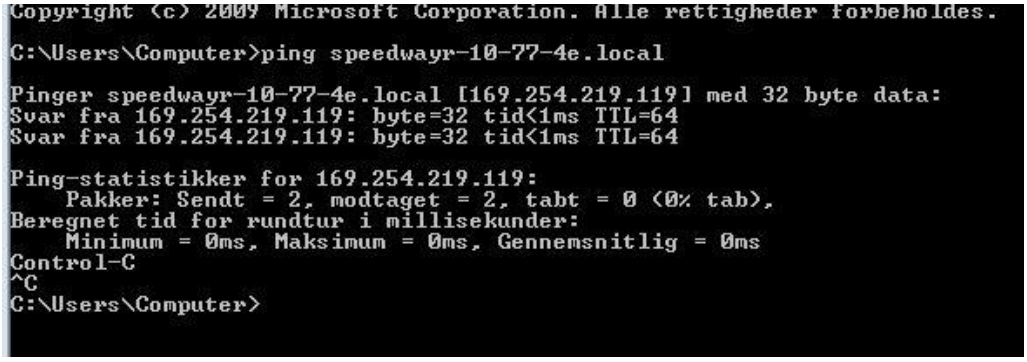


Figure F.10: This figure shows the Impinj Speedway R220 being pinged.

F.4. Setup Impinj Speedway R220 for WISP tag communication

The WISP 4.1 tag uses Miller encoding with M4 sub carrier modulation. In order to communicate with the WISP 4.1 tag, the Speedway needs to be configured to use M4.

1. Open *Settings* and select 'Modes, RF, Power' under *Reader 1*. Set the 'Reader Mode' to 'Dense Reader (M=4) 1'

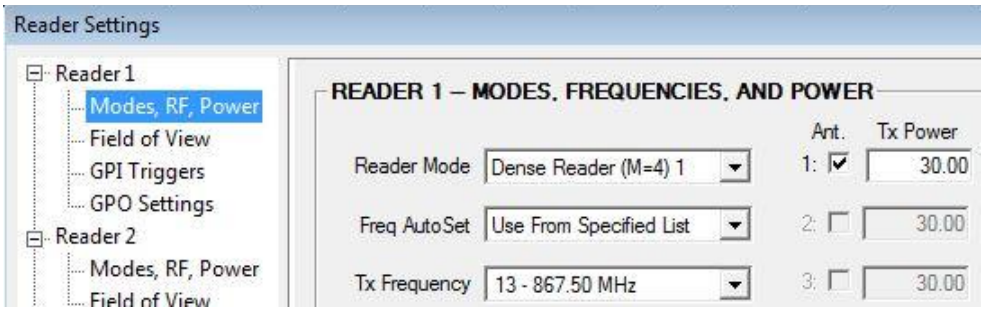


Figure F.11: This figure shows the Impinj Speedway R220's 'Reader Mode' being set to 'M4'.

F.5. Octane SDK

It is possible to program the Impinj Speedway R220 with the *Octane SDK*, which supports .NET and Java. The Octane SDK is only for application level programming and the low level communication protocols are not accessible from within this SDK. We have tried to read the EPC from a WISP 4.1 tag using a C# program based on the Octane SDK.

F.5.1 C# with Octane SDK setup

1. Download Octane SDK from `https://support.impinj.com/hc/en-us/articles/202755268-Octane-SDK`

2. Open the Solution in Visual Studio

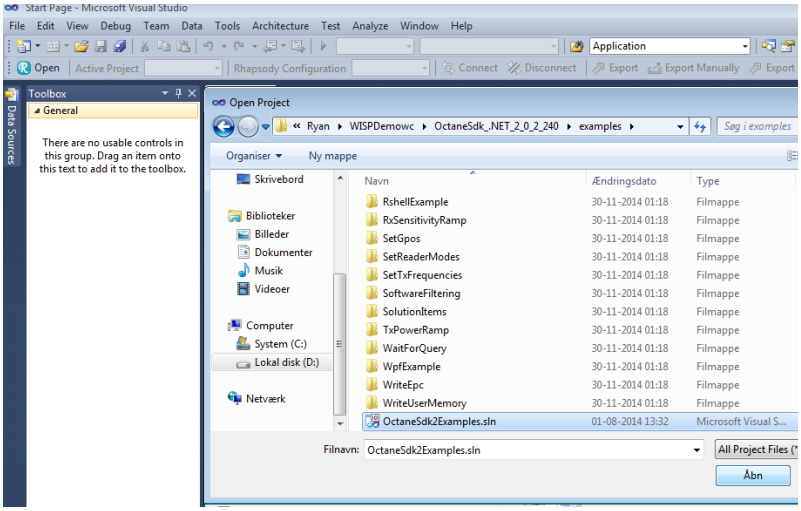


Figure F.12: How to open Octane SDK in Visual Studio.

3. Open the 'WpfExample'

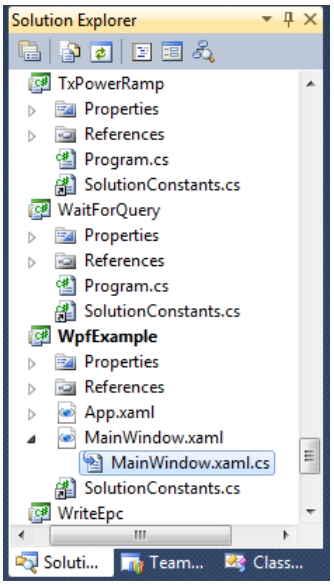


Figure F.13: This figure shows the Octane WpfExample program

4. Open the 'SolutionConstants.cs' in the project.

   a) Change the ReaderHostname to:
   public const string ReaderHostname = "SpeedwayR-xx-xx-xx.local";
   (where xx-xx-xx are the 6 last values to right of "MAC 00:" label on the speedway)

SolutionConstants.cs | MainWindow.xaml.cs | MainWindow.xaml

OctaneSdkExamples.SolutionConstants

```
1  //////////////////////////////////////////////////////////////////////////
2  //
3  //    Solution Constants
4  //
5  //////////////////////////////////////////////////////////////////////////
6
7  namespace OctaneSdkExamples
8  {
9      public static class SolutionConstants
10     {
11         public const string ReaderHostname = "SpeedwayR-10-77-4e.local"
12     }
13 }
14
```

Figure F.14: This figure shows the hostname in the WPF example, which is used to connect the host machine to the Impinj speedway reader.

b) In the 'public MainWindow()' somewhere after this code:
Settings settings = reader.QueryDefaultSettings(); (Around line 40 in the code)
Insert the following code: **settings.ReaderMode = ReaderMode.DenseReaderM4;**

MainWindow.xaml.cs* | MainWindow.xaml | SolutionConstants.cs

OctaneSdkExamples.MainWindow

```
19     /// </summary>
20     public partial class MainWindow : Window
21     {
22         // Create an instance of the ImpinjReader class.
23         private ImpinjReader reader = new ImpinjReader();
24
25         public MainWindow()
26         {
27             InitializeComponent();
28
29             try
30             {
31                 // Connect to the reader.
32                 // Change the ReaderHostname constant in SolutionConst
33                 // to the IP address or hostname of your reader.
34                 reader.Connect(SolutionConstants.ReaderHostname);
35
36                 // Get the default settings
37                 // We'll use these as a starting point
38                 // and then modify the settings we're
39                 // interested in.
40                 Settings settings = reader.QueryDefaultSettings();
41
42                 settings.ReaderMode = ReaderMode.DenseReaderM4;
```

Figure F.15: This figure shows the ReaderMode being set to M4 in the WPF example.

settings.ReaderMode = ReaderMode.DenseReaderM4; Set the encoder to M4, which is the encoder the WISP tag uses.

Appendix G

Tag states of the EPC standard

The EPC standard for the RFID tag specifies, that a tag should be a state machine, which have different states in a program, and will interact differently with the commands sent from the Interrogator/reader depending on with state the program is in. The states specified for the RFID tag are the following:

G.1. Ready state

This is the first state of the tag, view it as a holding state, when a tag is energized it goes into this state, where it is listening for a Query command from a reader, where the session and sel parameters matches its current flag values. When the tag receives a Query with matching session and sel parameter it draws a Q-bit from the its RNG and loads this value into its slot counter, if this value is nonzero, the tag goes into a arbitrate state, if not, it goes into a reply state. A tag can be participating in an inventory round or is killed and loses its power, if that is the case, then it won't go to the ready state, when it is re energized. The session and sel are parameters used for handling a reader population and a tag population, the RNG and slot counter are also used for handling a tag population, all this is explained later.

G.2. Arbitrate state

View this state also as a holding state but unlike the ready state, this is a holding state for a tag that is participating in the current inventory round, where its slot counter holds a nonzero value. In the arbitrate state, the tag is listening for a QueryRep command with a matching session parameter from the reader, when this happens the tag decrements its slot counter. The tag goes to the reply state when the slot counter reaches the value of 0000h A tag that returns to the arbitrate with a slot counter value of 0000h will decrement the slot counter to 7FFFh when it receives the next QueryRep, it will remain in the arbitrate state until the slot counter reaches 0000h again.

G.3. Reply state

When a tag goes into the reply state, it backscatters an RN16 to the reader, then the tag listens for a ACK command from a reader with the value from the RN16, when the tag receives a valid ACK command with the RN16, it backscatters its EPC to the reader and goes to the acknowledged state. If a tag doesn't receive a valid ACK within a time T2(max) it returns to the arbitrate state.

The RN16 is a random 16 bits value. T2 (max) is a time value within the range of 3.0 Tpri to 20.0 Tpri. Tpri is the link time from the tag to the reader. Tpri equal to 1 divided by BLF

G.4. Acknowledged state

When a tag is in the acknowledged state, it may go to any state except killed state depending on the received command from a reader(look in figure xxx). If a tag receives a valid ACK with the correct RN16, it will backscatter the EPC again. A tag that does not receive a valid command within the time T2(max) returns to the arbitrate state.

G.5. Open state

A tag goes to the open state from the acknowledged state, when it receives a `Req_RN` command from a reader and the tag has a nonzero access password, it also backscatters a new RN16, which is referred to as a handle, that the reader will use in subsequent commands and the tag will use in subsequent replies. When a tag is in the open state, it may execute some access commands like the Read, Write and Kill commands, which are not allowed in the Acknowledged state. A tag in the open may go to any state except acknowledged state depending upon the command it receives. If a tag in the open state receives a valid ACK command with the correct handle, it will re-backscatter the EPC.

G.6. Secured state

A tag goes to the secured state from the acknowledged state, when it receives a `Req_RN` command from a reader and the tag has a zero access password, it also backscatters a new RN16, which is referred to as a handle, that the reader will use in subsequent commands and the tag will use in subsequent replies. When a tag is in the open state it may go to the secured state following a successful Access command sequence or Interrogator authentication, maintaining the same handle that was previously backscattered, when it went into the open state. A tag in the secured state may execute all access commands. A tag in the secured state may go to any state except acknowledged state depending upon the command it receives. If a tag in the secured state receives a valid ACK command with the correct handle, it will re-backscatter the EPC.

G.7. Killed state

When A tag is in the open state or secured state may goto the killed state when receiving a successful password-based Kill-command sequnce with a correct nonzero kill password and handle. Kill permanently disables a tag, when a tag goes to the killed state, it notifies the reader, that the kill command was successful and will not respond to any command from a reader thereafter, a killed tag remains in the killed state under all circumstances. Killing a tag is irreversible.

Appendix H

WISP 4.1 setup and firmware structur

H.1. Physical setup

The WISP Tag uses the Texas Instruments MSP430F2132 MCU, which is the controller you will be programming, when you are programming the WISP. You will need the following equipment:

1. Texas Instruments MSP430 USB-Debug-Interface
2. JTAG cable
3. USB cable (one side type A and the other side type B)
4. WISP 4.1 Programmer
5. WISP Tag 4.1 DL

Figure H.1: The MSP430 USB-Debug-Interface is in the top middle, the JTAG cable is in the bottom middle, the USB cable is in the right, the WISP programmer is in the bottom left and the WISP tag is in the top left

Figure H.2: The USB cable connects the MSP430 USB-Debug-Interface with the computer. The JTAG cable connects the MSP430 USB-Debug-Interface with the WISP 4.1 Programmer and the WISP 4.1 Programmer is also connected with the WISP tag

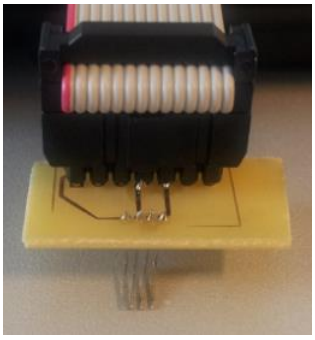


Figure H.3: Because there is more than one way to connect the JTAG cable with the WISP 4.1 Programmer, it is important to get the position of the connection right(**Take notes of the red coloured line!**)

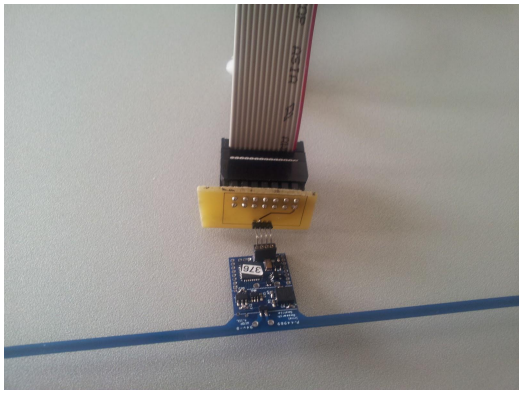


Figure H.4: The WISP 4.1 Programmer and the JTAG cable should have the same position as shown in the picture above, use the red colored line on the JTAG cable as a reference. The WISP TAG should also be connected to the WISP 4.1 Programmer with the same position as shown in the picture above

Luckily the rest of the connects can only be connected in one way, so one can't go wrong with them.

H.2. Software IDE for the WISP tag

The software tool used to program the WISP is the "IAR Embedded Workbench for MSP430"(v5.40 or lower is recommended because of compatibility issue with newer versions), which can communicate with the MSP430 USB-Debug-Interface, it can be found here:
`http://iar-embedded-workbench-kickstart-for-msp.software.informer.com/download/`

After you have installed the IAR Embedded Workbench for MSP430, download the WISP firmware for the WISP DL 4.1:
`https://github.com/wisp/dlwisp41/tree/v1.0` (stable version)

Now one have to open the workspace of the WISP firmware:

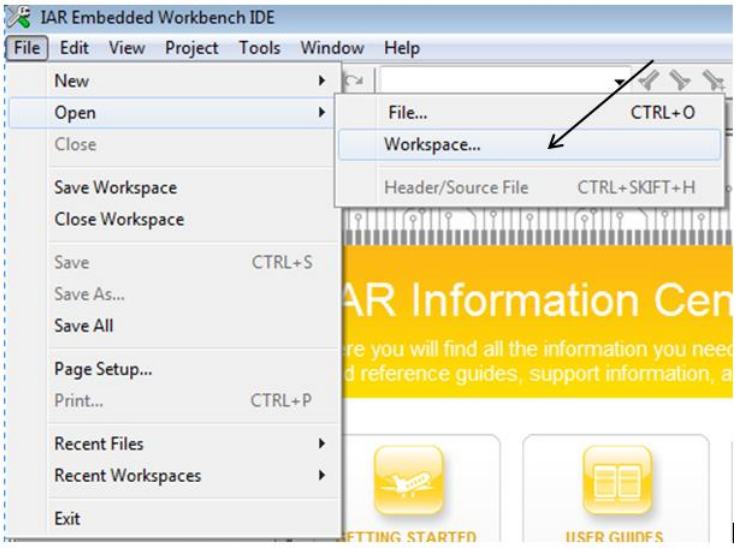


Figure H.5: The figure shows how one opens a workspace in the IAR IDE, which is used as the WISP IDE

Locate the ”dlwisp41.eww” file and open it:

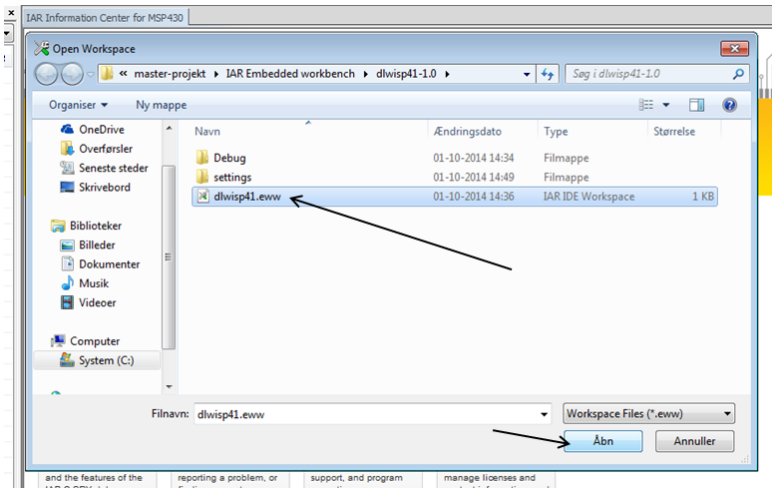


Figure H.6: The figure shows how one opens the wisp project in the IAR IDE

Now one has to set up some options inside the project, right-click the project and select ”options...”:

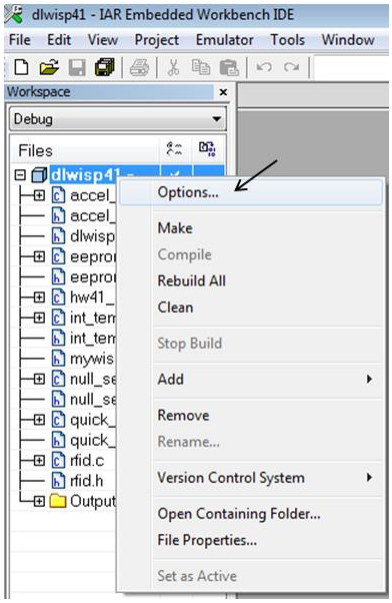


Figure H.7: The figure shows how one opens the options in a wisp project

First one has to set the ”Device” to match the MCU on the WISP 4.1 tag (It is ”MSP430F2132”):

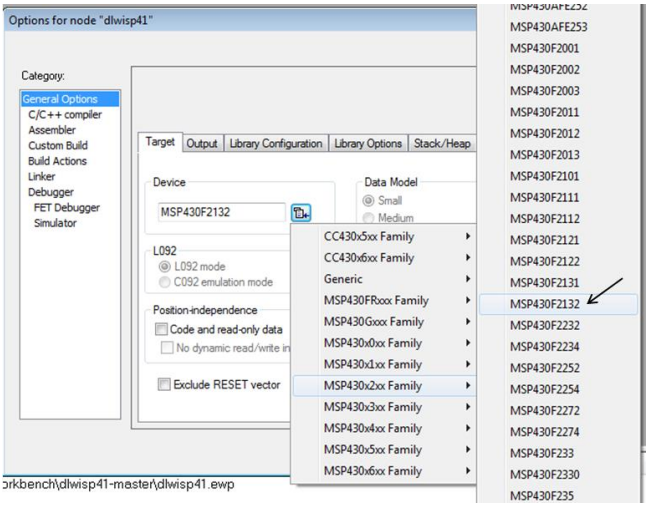


Figure H.8: The figure shows how one selects the ”MSP430F2132” in a wisp project

Next under ”C/C++ compiler” make sure that ”C++” is selected under the ”language 1” tab:

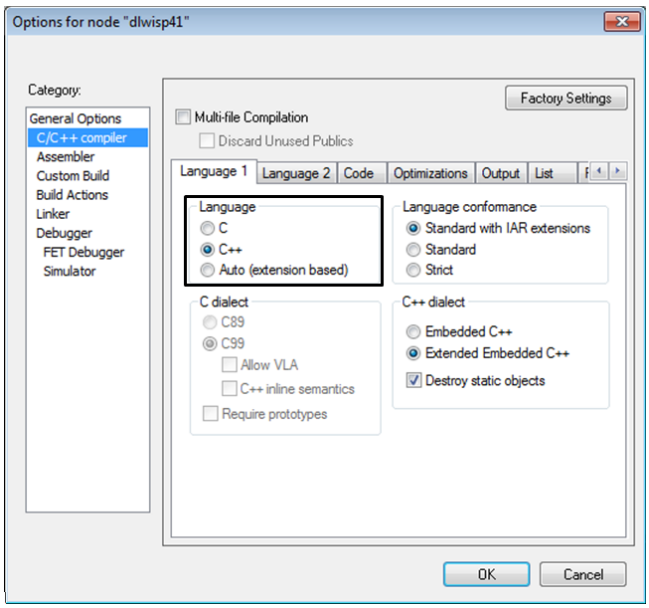


Figure H.9: The figure shows how one selects the language in a wisp project

Enable the regvar settings for registers 4 and 5 under the ”Code” tab:

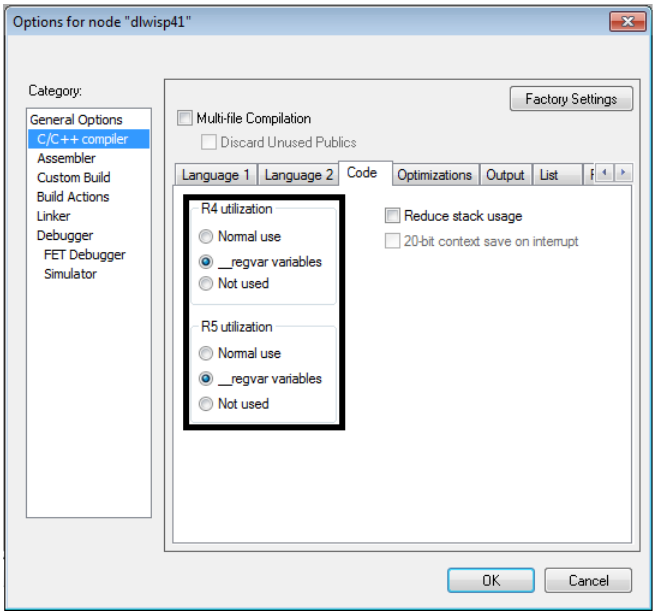


Figure H.10: The figure shows how one selects the regvar for R4 and R5 utilizations in a wisp project

Next under Debugger setup set ”Driver” to ”FET Debugger”:

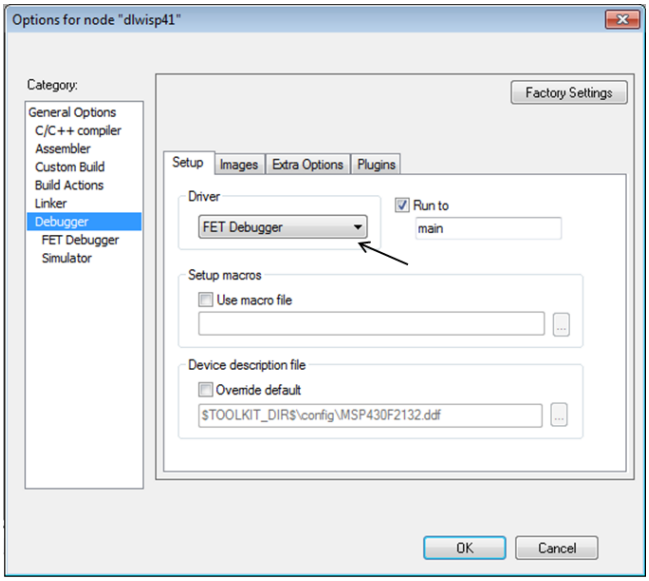


Figure H.11: The figure shows how one selects the FET debugger in a wisp project

Lastly set Connection to ”Texas Instrument USB-IF” and ”Automatic”, the ”Target VCC (in Volt)” should be set to around 2.5V(recommended)(**NOT lower than 1.8 V or higher than 3.6 V**,because that is the voltage supply for the MCU on the WISP tag):

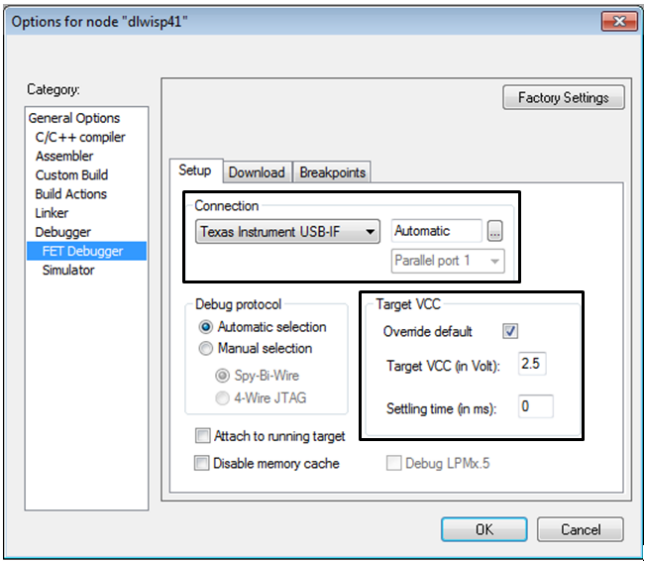


Figure H.12: The figure shows how one selects the voltage in a wisp project

Now you is ready to compile and download the firmware onto the WISP tag. Open the "`hw41_D41.c`" file and press **CTRL+D** to download and debug the WISP program:

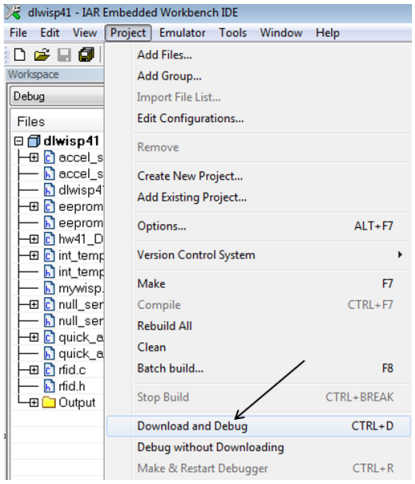


Figure H.13: The figure shows how one downloads WISP project on to the WISP tag and debug it

You should get something like this:

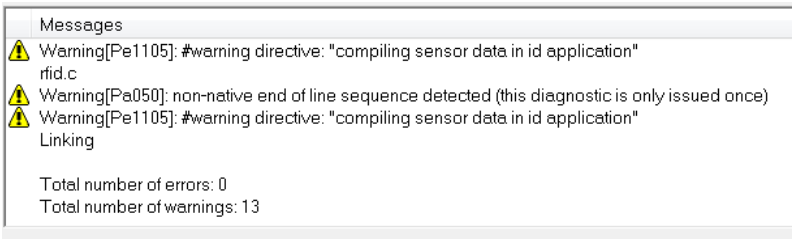


Figure H.14: The figure shows the results in the IAR IDE, when the WISP program is compiled

IF YOU GET SOMETHING LIKE THIS:

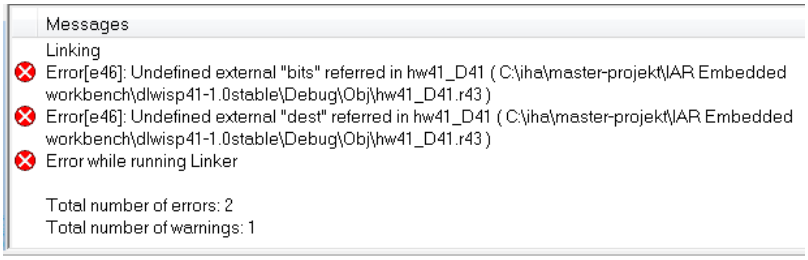


Figure H.15: The figure shows the results in the IAR IDE,if the WISP program is **NOT** compiled

You need to remove the ”extern” in the definition of both ”bits” and ”dest”:

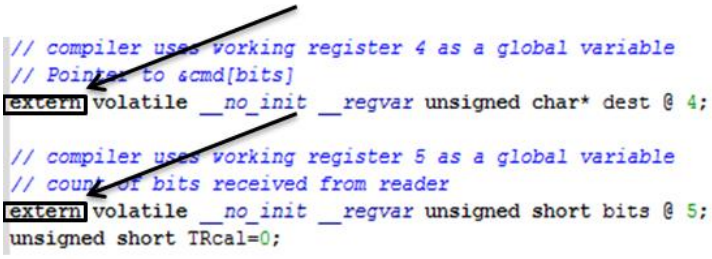


Figure H.16: The figure shows the ”extern” for ”bits” and ”dest” in the WISP program

Like this:

```
// compiler uses working register 4 as a global variable
// Pointer to &cmd[bits]
volatile __no_init __regvar unsigned char* dest @ 4;

// compiler uses working register 5 as a global variable
// count of bits received from reader
volatile __no_init __regvar unsigned short bits @ 5;
unsigned short TRcal=0;
```

Figure H.17: The figure shows the ”extern” for ”bits” and ”dest” **removed** in the WISP program

H.2.1 Debugging the WISP

You can add a breakpoints by double-clicking the grey area to the left of the line you want a breakpoint in:

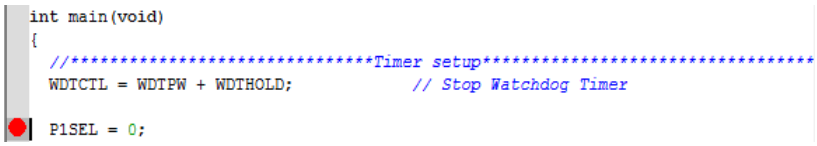


Figure H.18: The figure shows a breakpoint in the WISP program

Then one starts the debugger by pressing **CTRL+D**, there are the following commands for the debugger:

Command	Keys
Go	F5
Step over	F10
Step into	F11
Step out	Shift+F11
Stop debugging	Shift+CTRL+D

Table H.1: This table shows the commands available for the debugger

You can check to value of a variable by holding the mouse cursor over it:

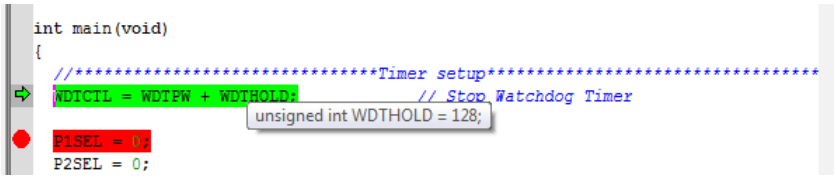


Figure H.19: The figure shows the value of a variable

You can only use 2 breakpoints, the MCU can't handle more and will disable the rests of the breakpoints.

H.3. Structure of the WISP 4.1 firmware(stable version)

The WISP 4.1 firmware is a C++ program, which that can be flashed into the "MSP430F2132" on the WISP 4.1DL tag, this program follows the EPC standard for communication from a RFID tag to a reader. The firmware consist of some global variables, which is used to configure the behavior of the WISP program, for example there are some applications mode in the firmware:

```
// Step 1: pick an application
// simple hardcoded query-ack
```

```
#define SIMPLE_QUERY_ACK                1
// return sampled sensor data as epc. best for range.
#define SENSOR_DATA_IN_ID               0
// support read commands. returns one word of counter data
#define SIMPLE_READ_COMMAND             0
// return sampled sensor data in a read command.
// returns three words of accel data
#define SENSOR_DATA_IN_READ_COMMAND     0
```

To select the application mode one wants the WISP to run, one set the variable for the desired mode to '1' and the rest to '0'. In the code above, the application mode is set to "simple hardcode query-ack", which is used to send the EPC value of the WISP to the reader.

Also inside the global variables the EPC value for the WISP is set:

```
// Step 4: set EPC and TID identifiers (optional)
#define WISP_ID 0x00, 2
#define EPC   0xDE, 0xAD, 0xBE, 0xEF, 0x00, 0xDE, 0xAD,
0xBE, 0xEF, WISP_VERSION, WISP_ID
#define TID_DESIGNER_ID_AND_MODEL_NUMBER  0xFF, 0xF0, 0x01
```

In the code above the EPC is set to "DE,AD,BE,EF,00,DE,AD,BE,EF,41,00". The `WISP_VERSION` is defined under a section, where the version of the WISP hardware is define look in the code below:

```
// Step 2: pick a reader and wisp hardware
// make sure this syncs with project target
#define BLUE_WISP                       0x41
#define WISP_VERSION                    BLUE_WISP
#define IMPINJ_READER                   1
```

The firmware is started under the "main()" function:

```
int main(void)
{
...
}
```

The main function is structured in the following way:

1. Setup the hardware on the MCU.
2. Go into a switch case, which is used as a state machine.

The state machine does 3 things:

1. Listens for commands from the reader.
2. Sends replies to the reader.
3. Switch the state of the WISP.

H.3.1 The flow of commands between the WISP and the reader

The flow of commands between the WISP and the reader, depends upon the configuration of both the WISP and the reader, does one want to read the ”EPC” of the from the WISP, read the temperature, accelerometer? What one wants, will affect which commands are sent. The default configuration is set to a simple Query-ACK, which is there one wants the EPC from the WISP. To get the EPC from the WISP the following commands are sent:

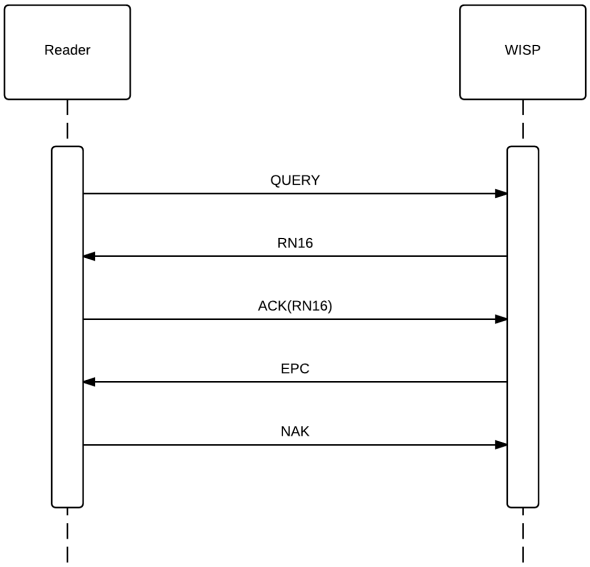


Figure H.20: The figure shows the flow of commands between a reader and a RFID tag for getting the EPC value from the tag

With the simple Query-ACK application mode, the WISP has the following states:

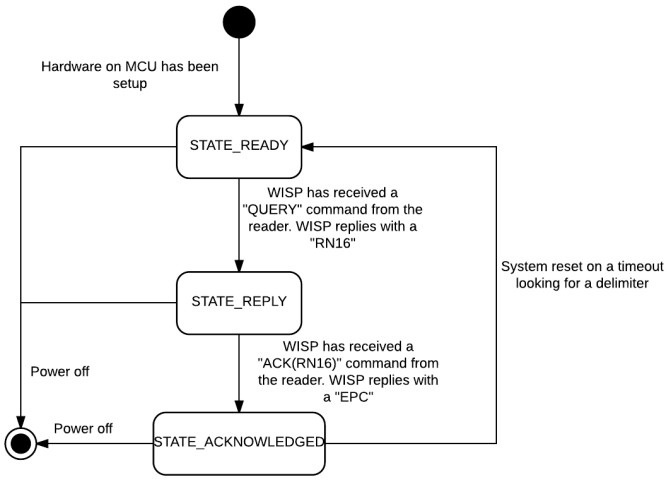


Figure H.21: The figure shows a state diagram of the states of the WISP for the simple Query-ACK application mode

Here are all the defined states in the WISP firmware:

```
#define STATE_READY                0
#define STATE_ARBITRATE            1
#define STATE_REPLY                2
#define STATE_ACKNOWLEDGED         3
#define STATE_OPEN                 4
#define STATE_SECURED              5
#define STATE_KILLED               6
#define STATE_READ_SENSOR          7
```

H.3.2 Send and receive data on the WISP

The WISP uses a function called **sendToReader(volatile unsigned char *data, unsigned char numOfBits)**, which takes an array of unsigned chars as the data to be sent to the reader, and an unsigned char which tells the number of bits to be sent. How exactly the data are sent from the MCU, is difficult to tell, because one needs to understand the circuit, which the MCU is connected to, and how registers works on the MCU.

The receiving part is unfortunately also difficult to understand, what we have found out, is that there is an interrupt on port 1, which is used to find the delimiter, which is the start of a message, there is also an interrupt on a time controller inside the MCU, which is periodically called, it moves the stored data from the registers onto a ”cmd” buffer, this buffer is used in the state machine to check, which command has been received from the reader.

Appendix I

Setting up Eclipse for out-of-tree module editing and debugging

There are two settings that must be configured in order to use eclipse with a GNURadio out-of-tree module: include paths and linking paths.

1. Ensure that any header referenced in your project source code is installed on the system and that its location is in the *include path* of Eclipse.

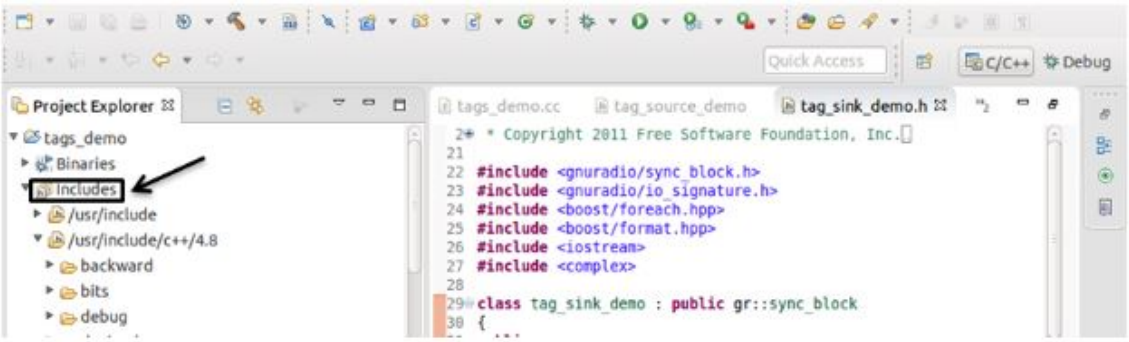


Figure I.1: The figure shows the *Includes* folder, where all the included headers of the form *#include <header>* must be located. Headers of the form *#include "header"* are local and do not need to be in the *Includes* folder.

2. Ensure that any library used in your project is installed on the system and that its location is in the *linking path* of Eclipse. To reach the dialog in figure I.2:
   a) Right-click the project folder and select *Properties*.
   b) Select *C/C++ Build -> Settings*
   c) In the *Tool Settings* tab, select *Libraries* under the *<used compiler> Linker*
3. Locate the library files on your system. These have the form *lib<name>.so*.
4. Add the paths to the library files in the *Library search path(-L)* field, by pressing the + button and pasting the path into the text field. Repeat for each path.
5. Add the libraries themselves in the *Libraries (-I)* field, by pressing the + button and pasting the *name*-part of the *lib<name>.so*.

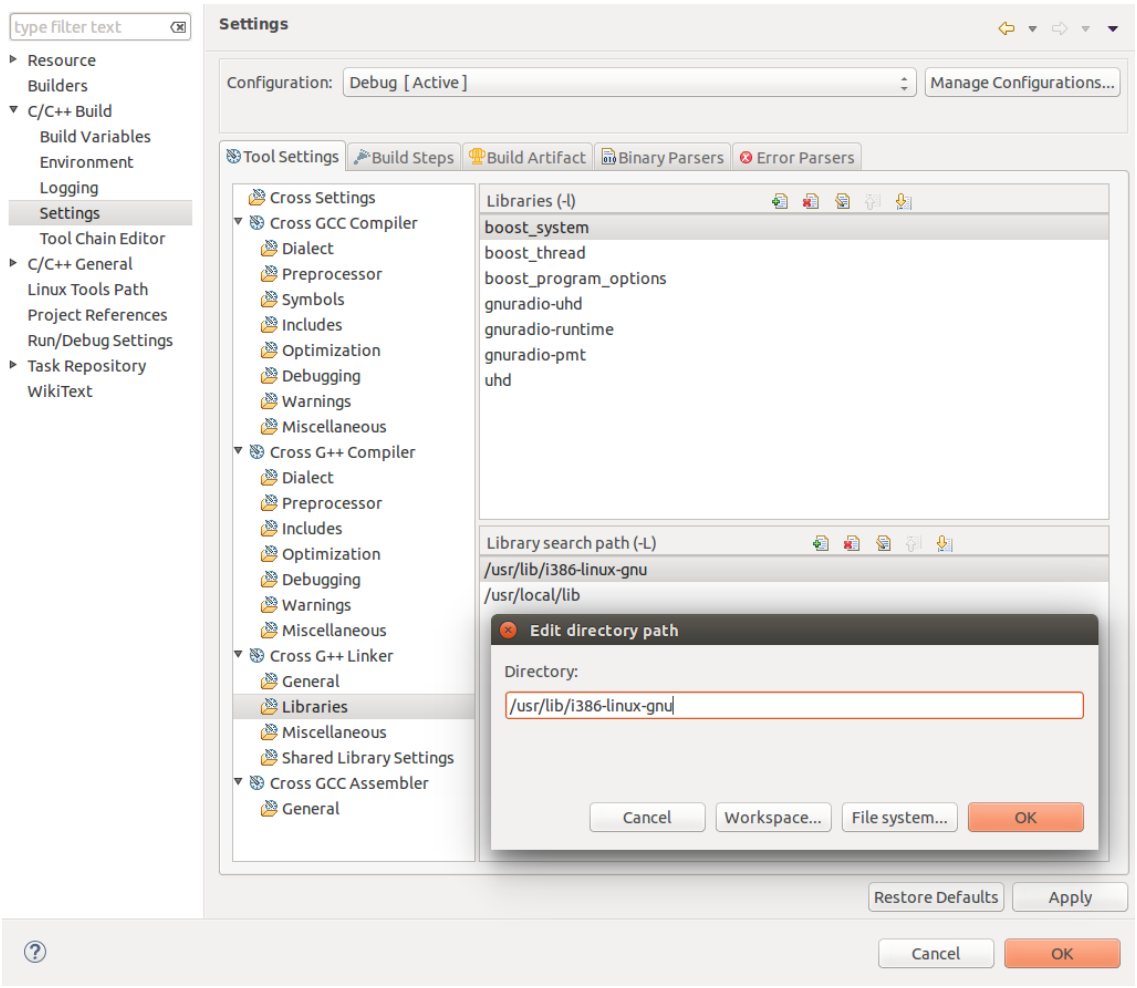


Figure I.2: The figure shows the build settings dialog, where the library linking is configured.

Appendix J

Making out-of-tree modules for GNU Radio

GNURadio includes standard blocks that cover many, but not all use cases. When blocks are needed beyond what GNURadio offers by default, it includes a tool called gr_modtool that allows users to set up a project and implement their own blocks. GNURadio refers to such projects as out-of-tree modules.

In order to use the gr_modtool it is assumed that GNURadio is already installed and configured correctly. It is not sufficient to install GNURadio from the package manager, since this does not include the development tools. Instead, use the installation instructions in these appendixes.

1. The first step is to navigate to the intended location of the new module source. The gr_modtool will create the module directory and place the module skeleton inside.

2. The next step is to use the gr_modtool newmod command:

   ```
   $ gr_modtool newmod name-of-module
   ```

   Replace *name-of-module* with the intended name. gr_modtool will prepend the chosen name with *gr-* for the module directory.

3. From within the module directory, add a block:

   ```
   $ gr_modtool add -t block-type block-name
   GNU Radio module name identified: module-name
   Language: C++
   Block/code identifier: block-name
   Enter valid argument list, including default arguments:
   Add Python QA code? [Y/n]
   Add C++ QA code? [y/N]
   Adding file 'block-name_impl.h'...
   Adding file 'block-name_impl.cc'...
   Adding file 'block-name.h'...
   Editing swig/module-name_swig.i...
   Editing python/CMakeLists.txt...
   Adding file 'module-name_block-name.xml'...
   Editing grc/CMakeLists.txt...
   ```

- Replace *block-type* with the type of block needed. It is important to understand the difference between the block types, since it affects which class your block inherits from and which methods you are required to override. If you need to change the block type at a later time it is not as simple as just changing the parent class.
- Replace *block-name* with the intended name of the block.
- When prompted for the argument list, you need to write the constructor arguments exactly as they will appear in your block constructor definition. This string will be copied directly into the code, including any errors.
- You can choose to add QA files if you want. This includes unit-test files and an integration test file.

4. Replace the block template code. The *gr_modtool add* command adds the block source files to the *<module-directory>/lib* directory. Some of the code is template code that needs to be replaced.

5. Implement block

6. Add gnuradio companion block declaration, so you can drag-and-drop blocks in the gnuradio companion GUI.

   ```
   $ gr_modtool makexml name-of-block
   ```

   This will generate an xml block definition in *<module-directory>/grc*. For anything except very simple blocks this xml will most likely have errors, but it is a good starting point. Open gnuradio companion and try to drag the block into the graph to debug errors in the xml.

7. Before the block can be used, it must be built, compiled and installed.

   a) Navigate to the module directory.

   b) Make a new directory called build and navigate into it.

   c) Run cmake

      ```
      $ ../cmake
      ```

      If there are any errors it is likely that there are some GNURadio development dependencies that you have not installed.

   d) Compile

      ```
      $ make
      ```

      You will probably have errors here. These are standard compiler errors telling you what is wrong with your code and where. Fix these errors and try again.

   e) Install

      ```
      $ sudo make install
      ```

8. Now you should be able to use your block. Open gnuradio companion, create a flow-graph, generate the python program and run it.

Appendix K

Encoding errors from WISP 4.1

K.1. Tag communication

The Wisp tag communicates with the radio by using backscatter modulation, where the WISP tag switches the reflection coefficient of its antennas between two states corresponding to the data being sent. The Modulation the wisp tag uses is ASK (Amplify shift key), the encoding is Miller with the Subcarrier Sequences being M=4, the data rate is 256 kHz. The Wisp tag doesn't follow the EPC standard, since it is actually the radio, that decides the BLF(Backscatter link frequency) by the ratio between the DR (divide ratio) and the TRcal (T→R calibration symbol), which the WISP tag gets from the preamble sent with a query from the radio, the encoding isn't decided by the tag according the EPC standard, but by the radio, where inside the query command, there is a M section, which tells the tag the type of encoding it should use, the DR is giving to the tag inside a DR section of the query.

K.2. Miller encoding

The basis of the Miller encoding is the following:

1. Each symbol has a T time period.
2. The phase is inverted between two data-0 symbols in sequence.
3. The phase is inverted in the middle of a T time period for a data-1 symbol

The Miller encoding also has a subcarrier sequence. This is where the basic miller encoding is modulated by multiplying it with a squared wave, which has M times the symbol rate, this means that the squared wave has M cycles per symbol over the time period T. There are the following M values 2, 4 and 8, which means that the squared wave can have 2, 4 or 8 cycles per symbol. Here are the possible phases for a sequence of three data symbols with the different subcarrier sequence: (Note at all these sequences start with a high phase, if the start of the sequence is a low phase, it is still here just with an inverted phase order)

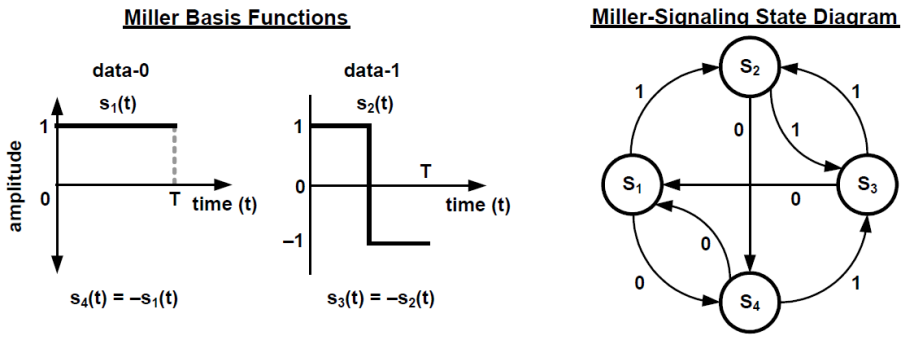


Figure K.1: The figure illustrates the basic state for data-0 and data-1 symbols, it also show the state diagram for the possible flow of the states

To mark the end of a respond from a tag, a dummy bit is placed in the end of the responds. The dummy bit is a data-1 symbol. Here are the possible dummy bits for the different subcarrier sequences:

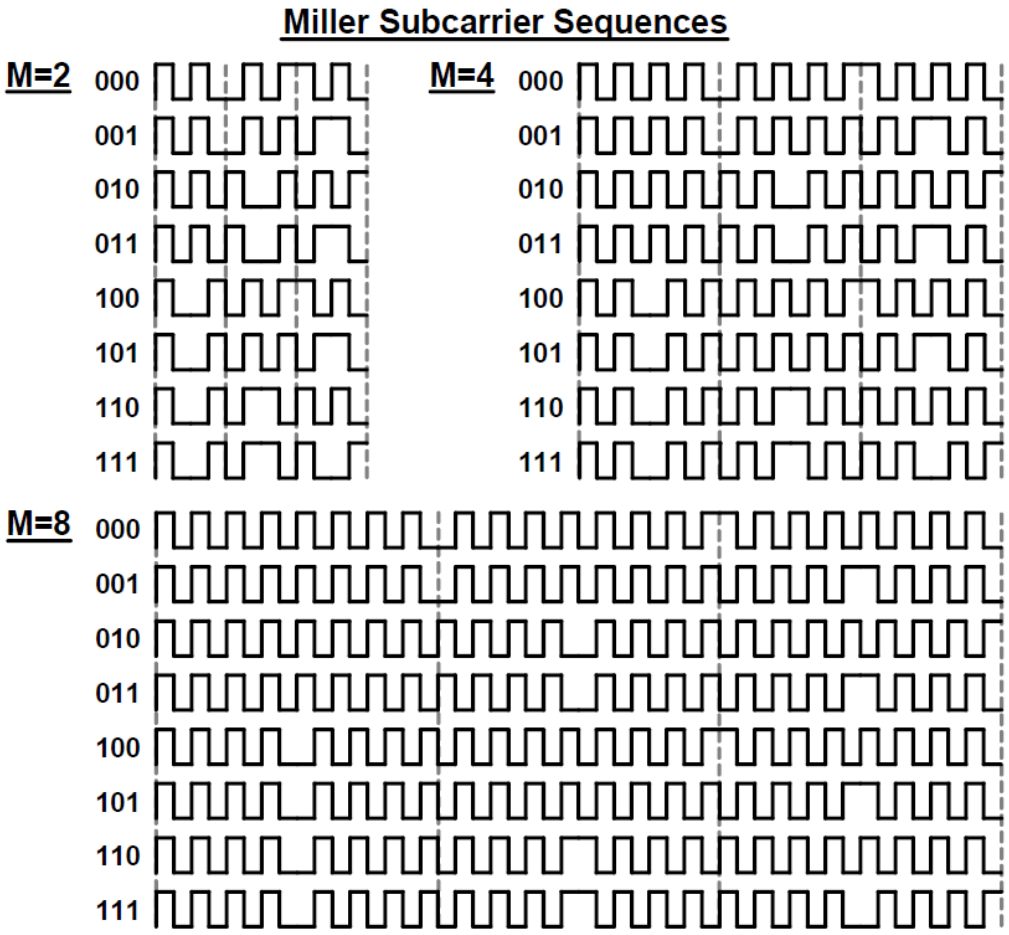


Figure K.2: The figure illustrates the possible phase state for a three data symbol sequence

To mark the beginning of a respond from a tag, a Miller preamble is placed in front of the responds. The preamble consists of two parts:

1. Data-0 symbols covering a certain time period, this period is depended upon three things: M, TRext which is a value in the query command, which tells the tag whether it should extend its preamble or not and the BLF used for the responds.
2. A data sequence of the following data symbols: 010111

K.3. Test RN16 response from WISP tag

To test if the WISP tag respects Miller encoding with its RN16 response to the radio. We did the following:

1. We send a query command from the radio to wisp tag (This was done with TRext=0, and the rest of the parameters are unimportant, since the wisp ignores them and uses its own settings for the response).

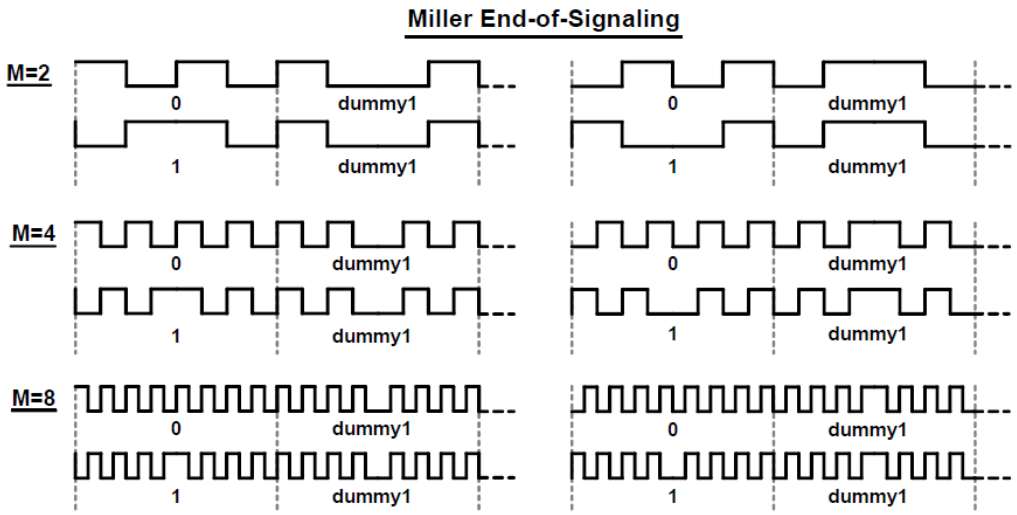


Figure K.3: The figure illustrates the possible phase state for the dummy bit, marking the end of tag a response

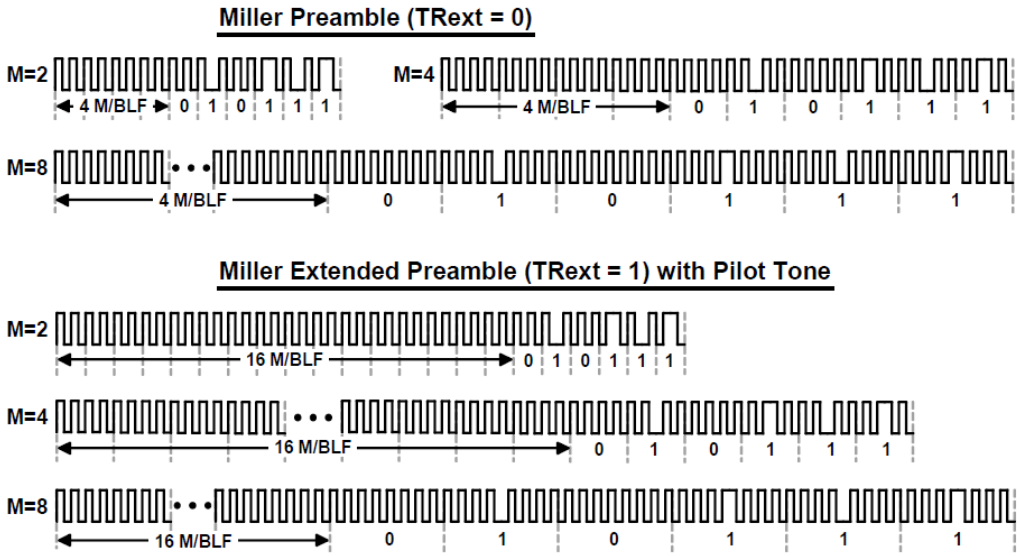


Figure K.4: The figure illustrates the possible phase state for the preamble, marking the start of tag a response

2. We stored the response captured by the USRP in a file and draw it on a graph.

K.3.1 Results of the RN16

We got the following RN16 response from the WISP tag:

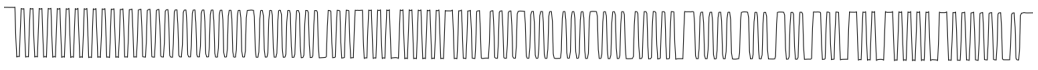


Figure K.5: The figure shows the RN16 response from the WISP tag

To test if the data in the response is valid, we will first have to remove the preamble and the dummy bit from it. Here you can see the preamble from the response:

Figure K.6: The figure shows the preamble of the response from the WISP tag

Here you can see the dummy bit from the response:

Figure K.7: The figure shows the dummy bit of the response from the WISP tag

Here you can see the payload in the response:

Figure K.8: The figure shows the payload of the response from the WISP tag

The raw bits from this signal are:
(The 0s are actually -1s, but for readability they are shown as 0s)
0101010110101010010101011010101001010101101010100 1 01010101001101
0101010011010100110101001101010011010100110 10101010011010101010

There are 127 samples here, which is 1 sample less, then we expected. Because with M4 encoding there are 8 samples per bit and there are 16 bits in a RN16. (8*16=128).

To check if these samples are structured right, we invert this RN16 back to its basic form by dividing it with the square wave for M=4:

Basic RN16 = RN16/Square wave

This is done sample for sample in the signals:

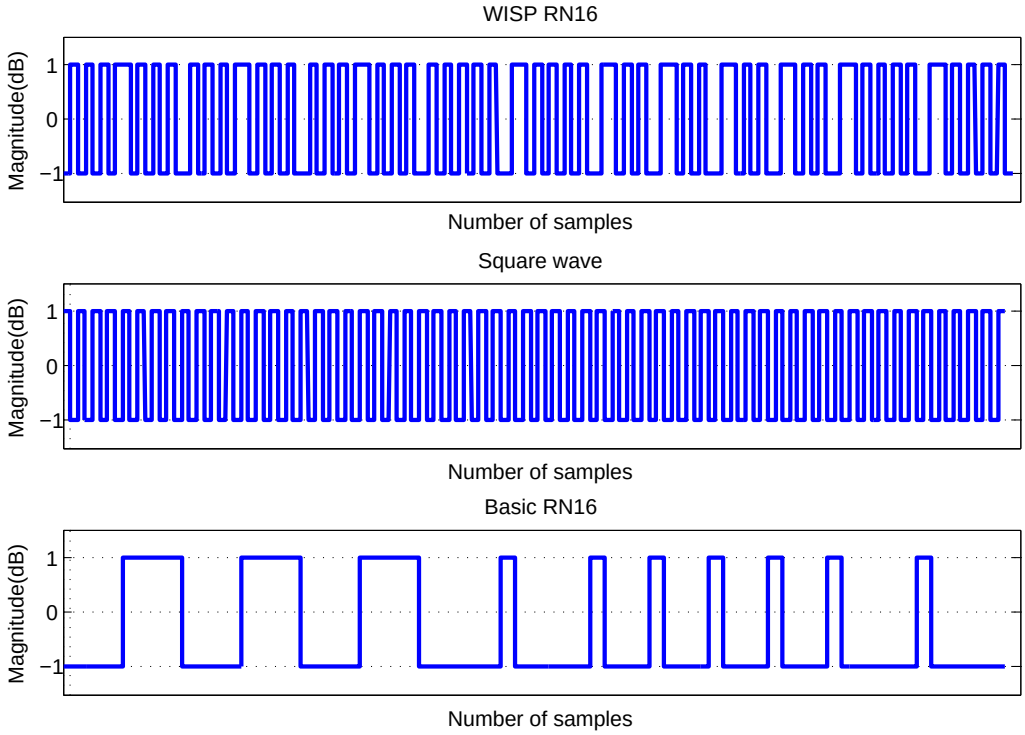


Figure K.9: The figure shows the RN16, the square ware and the basis RN16 response from the WISP tag

The basic RN16 has the following states (from left to right):
S4→S1→S4→S1→S4→S1→S4

After that the basic RN16 doesn't follow the Miller state diagram anymore, because the signal structure below is not a valid state:

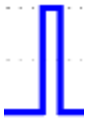


Figure K.10: The figure shows a invalid state in the basis RN16 response from the WISP tag

There is no data symbol, which inverts their phase twice in the same time period, data-1 symbols inverts the phase once in the middle of the time period and data-0 symbols inverts the phase between the time periods.

So the WISP tag response with an invalid structure and it is also too short for the RN16.

Appendix L

Transmission Power for the daughterboard WBX 50-2200 MHz Rx/Tx

L.0.2 Powering the tags

The RFID tags gets their power from the signal transmitted by the reader, therefore it is a vital thing to know, how strong the transmitted signal must be in order to power up a RFID tag, unfortunately there is no documentation on the used tags in the measurements, telling how strong a signal must be in order to power up a tag**(put reference)**. The only reference point we have is from on of buettner's articles**(put reference)**, where he stated, that he used an output power of 75 mW, when he was communication with RFID tags.

L.0.3 Gain

GNU radio offers very little information about the transmission power of the radio, there is only one setting inside of GNU radio framework, that allows you to have some control over the transmission power, this setting is the 'Gain' setting inside the USRP sink block, it has a range from 0 to 25, where the higher the value, the higher the transmission power.

L.0.4 Measuring the output power

To measure the output power for the daughterboard, one needs a power meter or a spectrum analyser. We used a FSQ26 spectrum analyser to measure the output power.

To protect the FSQ26 spectrum analyser an attenuator was connected to it, which weakened the signal from the USRP with 10 dbm. The cable uses for the connection between the FSQ26 spectrum analyser and the USRP also weakened the signal with 0.5 dbm. The measurements were done with a frequency tuned to 868 MHz

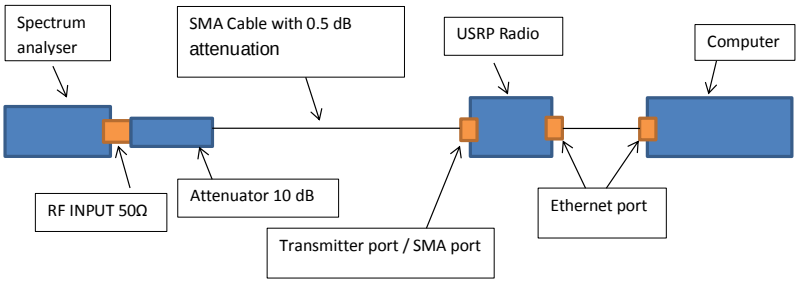


Figure L.1: The figure shows the setup for measuring the output power on the daughterboard WBX 50-2200 MHz Rx/Tx

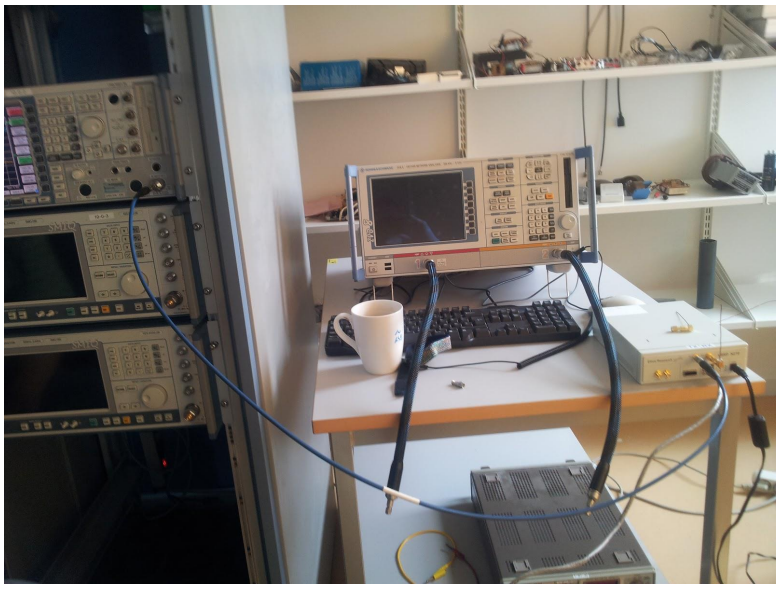


Figure L.2: The figure shows the overview setup for measuring the output power. Note it is cable with the white label that's the SMA cable used for the experiment

Figure L.3: The figure shows the attenuator used to protect the FSQ26 spectrum analyser

L.0.5 Calculating the output Power

The reading shown in figure L.4 shows a signal of -5.71 dBm, this reading can't be used directly because of the attenuation effects in the setup:

1. The attenuator weakens the signal with 10 dB.
2. The SMA cable weakens the signal with 0.5 dB.

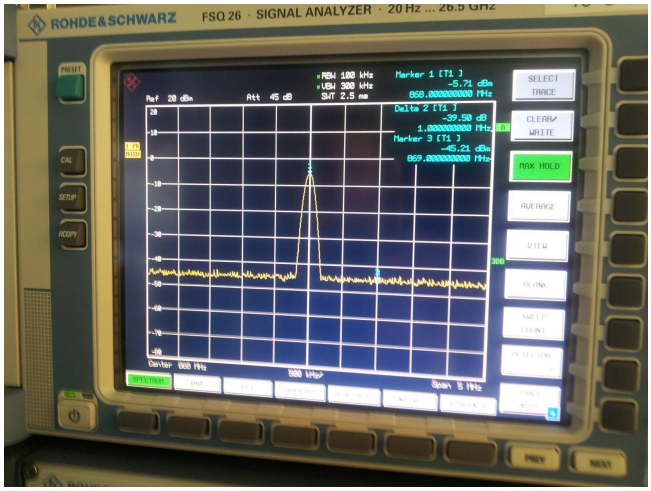


Figure L.4: The figure shows the FSQ26 spectrum analyser's display reading a signal with the strength of -5.71 dBm at the frequency 868 mHz

So we need to add these attenuations to the reading for getting the real output power from the daughterboard:

$$outputPower = measuredPower + attenuator + cable$$
$$outputPower = -5.71dBm + 10dB + 0.5dB$$
$$outputPower = 4.79dBm$$

This reading was done with a Gain=14.

We also converted the readings from dBm to mW, since this was the quantity buettner used, when he described the signal strength in his work**(put reference)**.

We used the following formula to convert the readings:

$$P_{mW} = 1mW * 10(\frac{P_{dBm}}{10})$$

For 4.79 dBm:

$$P_{mW} = 1mW * 10(\frac{4.79dBm}{10})$$

$$P_{mW} = 3.0130060242mW$$

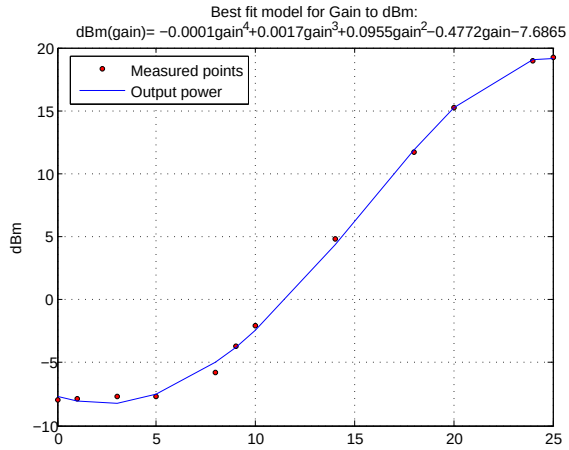
L.0.6 Output power results

With all the attenuation effects taking into account, the following measurements were observed:

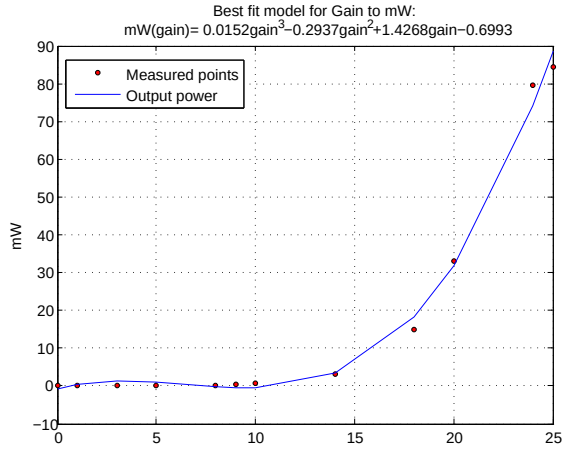
Gain	dBm	mW
25	19.27	84.527884516
24	19	79.432823472
20	15.2	33.113112148
18	11.7	14.791083882
14	4.79	3.0130060242
10	-2.1	0.61659500186
9	-3.7	0.4265795188
8	-5.8	0.26302679919
5	-7.7	0.16982436525
3	-7.7	0.16982436525
1	-7.9	0.16218100974
0	-7.97	0.15958791472

Table L.1: This table shows the measured output power for the radio with the daughterboard WBX 50-2200 MHz Rx/Tx, at a given gain setting inside the USRP sink Block.

With the 12 measurements, we tried to make some formulas, so we could predict the transmission power based on the Gain setting used in GNU radio, we did this with the use of the 'Least-squares best fit' theory for Gain to dBm and Gain to mW, there is no guarantee that the formulas are completely correct, but they can give an qualified idea above the relationship between the Gain and the output power with the use of the daughterboard WBX 50-2200 MHz Rx/Tx.



(a) Gain To dBm model



(b) Gain To mW model

Figure L.5: Best fit models for Gain to decibel-milliwatts and Gain to milliwatt

The maximum output power for the transmission from the daughterboard was 84.52 mW, with buettner's 75 mW for the communication with the tags, we know that with a gain≥24, we have a signal strength just as strong as the one he used.

Appendix M

Shielding the signal between the antennas

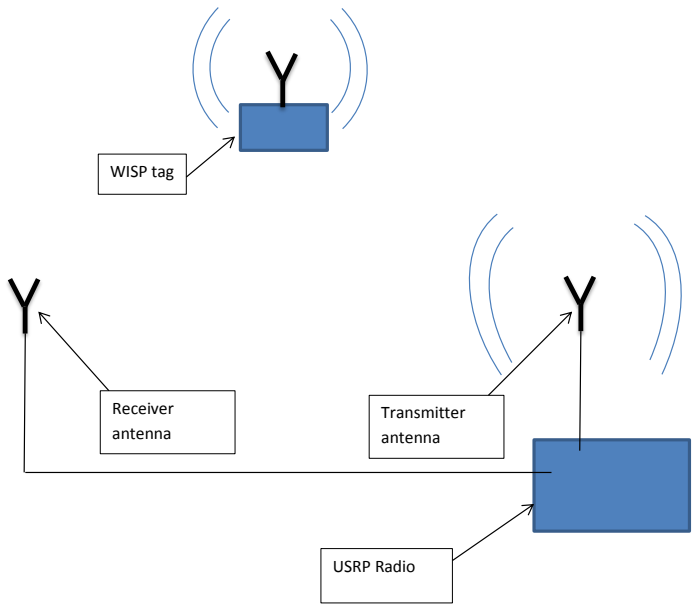


Figure M.1: The figure shows the communication scenario between a tag and the antennas on a USRP radio

The transmitter antenna needs to send an strong signal to power up the RFID tag, the RFID tag backscatters this signal, when it sends a reply. The receiver antenna will receive both the signal from the transmitter antenna and the backscattered signal from the RFID tag, therefore the signal from the transmitter antenna needs to be attenuated, so the receiver antenna can hear the signal from the RFID tag, else the RFID tags signal might be drowned by the signal from the transmitter antenna.

M.0.7 Shield method

Our first method to attenuate the signal from the transmitter antenna, is to place a shield between the transmitter antenna and the receiver antenna, the shield is a PCB plate and it is grounded to the ground from the transmitter antenna. This way the signal should be attenuated in the direction from the transmitter antenna to the receiver antenna. The antennas are in a vertical position, where they are placed parallel to each other.

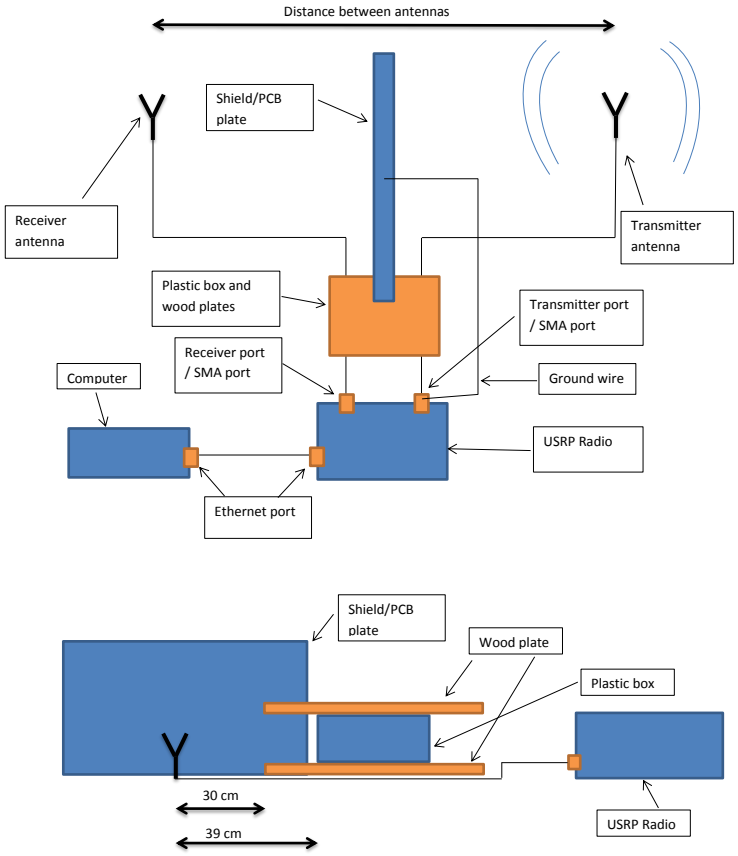


Figure M.2: The figure shows the shield placed between the antennas on a USRP radio setup

M.0.8 Horizontal method

Our second method to attenuate the signal, is to position the antennas in a horizontal position with their bases facing each other, this way the signal from the transmitter antenna will be weakened in the direction of the receiver antenna.

M.0.9 Ground antennas method

Our last method to attenuate the signal, is to grounding the antennas, a PCB plate is soldered to the base of antenna. There are 3 test scenarios for this method:

1. Both antennas are grounded. The receiver antenna is placed in a horizontal position with a edge of the PCB plate touching the surface and the base is facing the transmitter antenna. The transmitter antenna is placed in a vertical position with the base of PCB plate touching the surface.

2. Only the receiver antenna is grounded, it is placed in a horizontal position with a edge of the PCB plate touching the surface and the base is facing the transmitter antenna. The transmitter antenna is placed in a vertical position.

3. Both antennas are grounded, they are placed is a horizontal position with a edge of the PCB plates touching the surface and with their bases facing each other.

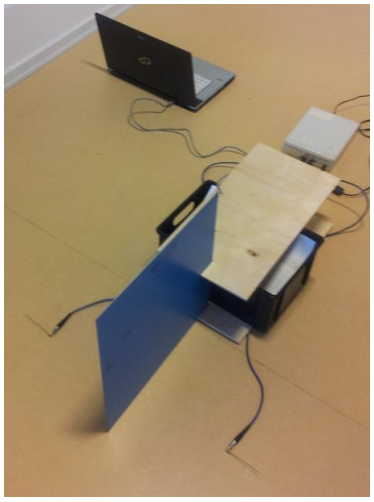


Figure M.3: The figure shows the setup of the shield test

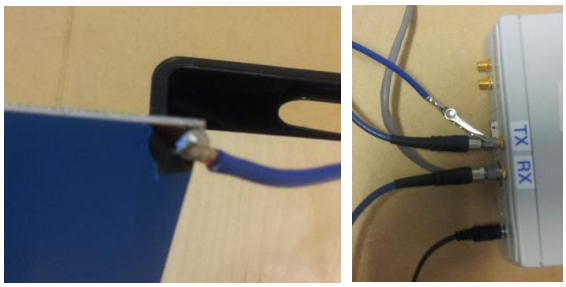


Figure M.4: The figure shows the PCB plate is soldered to a wire, which is connected to the ground of the transmit antenna

Figure M.5: The figure shows the antennas placed vertical to each other.

M.0.9.1 Used material for experiment

The length of receiver antenna is 8.1 cm for the receiver antenna and 7.5 cm for the transmitter antenna, the antennas are made of copper.

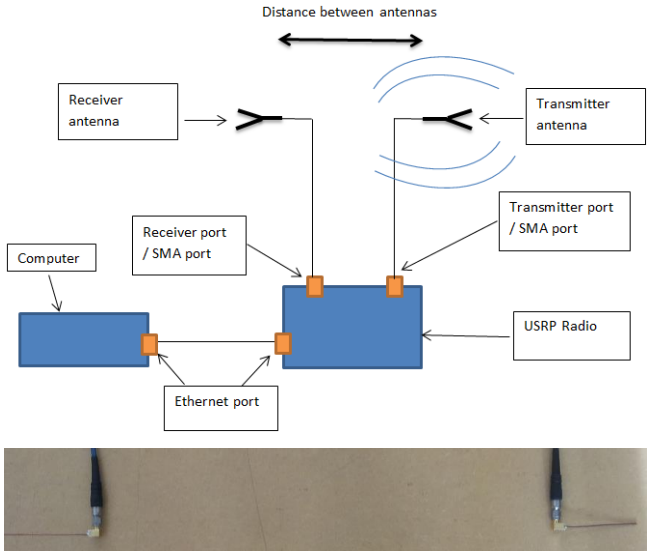


Figure M.6: The figure shows the antennas placed in a horizontal position with their bases facing each other.

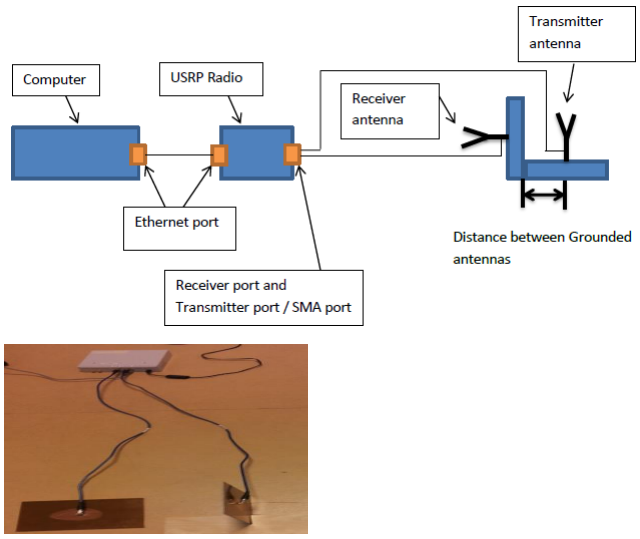


Figure M.7: Both antennas grounded to seperate PCB plates.

The shield is made of PCB plates, The shield has a dimension of 39.8 cm width, 40.7 cm height and 0.2 cm depth.

The wood plates have a dimension of 24 cm width, 48.3 cm height, 0.3 cm depth and the hole on the side of the plates has the dimension 0.2 cm width, 5.4 cm height and 0.6 cm depth.

The plastic box has a dimension of 22 cm width, 15.4 cm and 32 cm depth.

The shield is made of PCB plate, the alloy has from the plate been removed, so the plate is conductive to the antenna. The shield's dimension is 18 cm width, 18 cm height and 0.2 cm depth. The antenna is placed in the center of the PCB plates.

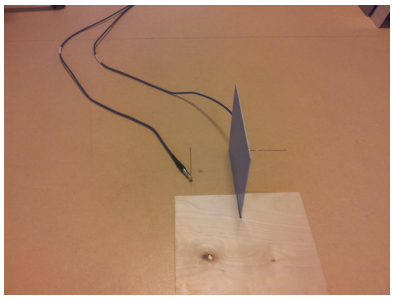


Figure M.8: Grounded receiver antenna, and un-grounded transmitter antenna.

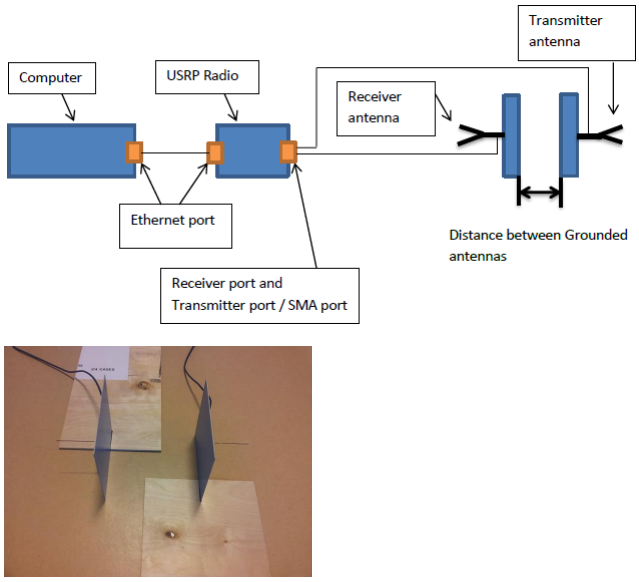


Figure M.9: Grounded antennas with bases facing each other

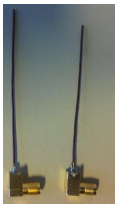


Figure M.10: The antennas used in the shield tests.

M.0.10 Measure the magnitude of the received signal

To test the attenuation effect of the shield and the horizontal position, the magnitude of the signal from the transmitter antenna is measured on the receiver antenna with different distances between the transmitter antenna and the receiver antenna. This test is first done without the use of the shield both for the vertical position and the horizontal position, then the shield is place between the antennas and grounded to the ground of the transmitter antenna, this is done for both for the vertical position and the horizontal position, this test the 3 scenarios with the grounded antennas is performed. The magnitude of

Figure M.11: The shield used in the shield tests.

Figure M.12: The figure shows the woodplate used to hold the shield in the shield tests.

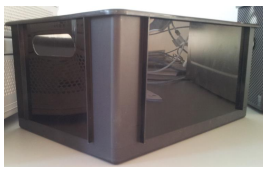


Figure M.13: The figure shows the plastic used to hold the woodplates in the shield tests.

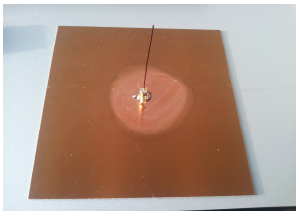


Figure M.14: The figure shows the antenna with a ground plate used used in the shield tests.

the signal received by the receiver antenna is measured for all scenarios, and the vertical position with no shield is used as the reference point of the signal, since no attenuation method were used here.

A constant signal is sent on the frequency 868MHz from the transmitter antenna, and the results is read from a graph on the receiver USRP, showing the magnitude of the received signal. The program used to send the signal, is made in gnuradio companion, it is made of a Signal source signal block generating the constant signal, this signal is passed on to a UHD USRP Sink block, which sends the signal out in the air. The program with its setting can be seen in the figure M.15

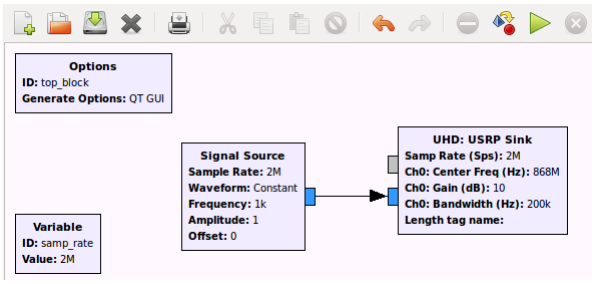


Figure M.15: The figure shows the program made in GNU radio companion, which sends a constant signal

The program used to receive signal and measure the magnitude of the received signal is made of a UHD source block, which received the signal from the air, this signal is passed on to QT sink gui block, which drew the magnitude of the received signal. The program with its setting can be seen in the figure M.16

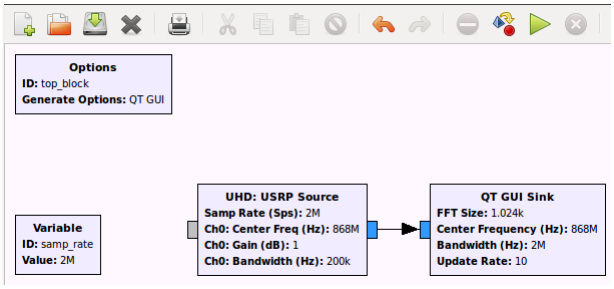


Figure M.16: The GNU Radio Companion program, which plots the received signal

The test results from the program was read by moving the cursor of the program to where the magnitude of signal was and the value was printed out on the screen besides the cursor. Since the signal was constant, the data retrieved was consistent, with a small error of placing the cursor within 1 DB tolerance, this can be seen in figure M.17

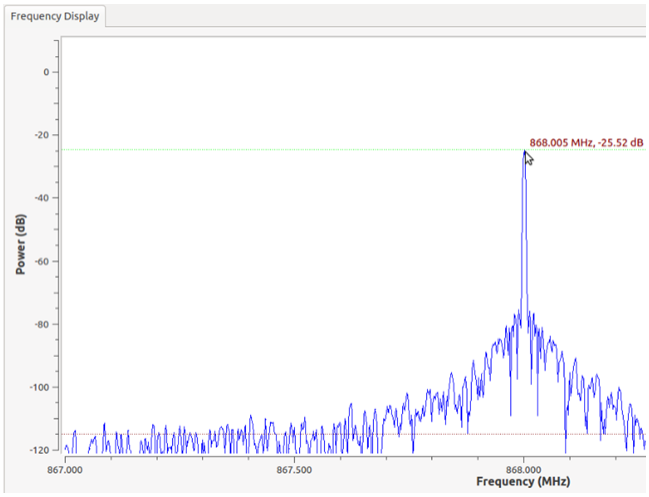


Figure M.17: The figure shows the received signal drawn on a QT GUI, with the frequency as the x-axis and magnitude as the y-axis

M.0.11 Results

The results for the attenuation effect of the shield method, horizontal method and grounded antennas method can be seen in figure M.18 and figure M.19.

The top graph in figure M.18 shows the attenuation effect on the received signal by placing a shield between the two antennas. The bottom graph in figure M.18 shows the attenuation effect on the received signal by placing the antennas in the horizontal position and also placing a shield between the two antennas.

These measurements used the vertical position with no shield as the reference point, so when a graph shows an attenuation of 5 dB, it means that the signal for the given attenuation method, at the given distance, is 5 dB weaker compared to the vertical position with no shield at the same distance and if it shows an attenuation of -5 dB, then the signal is 5 dB stronger.

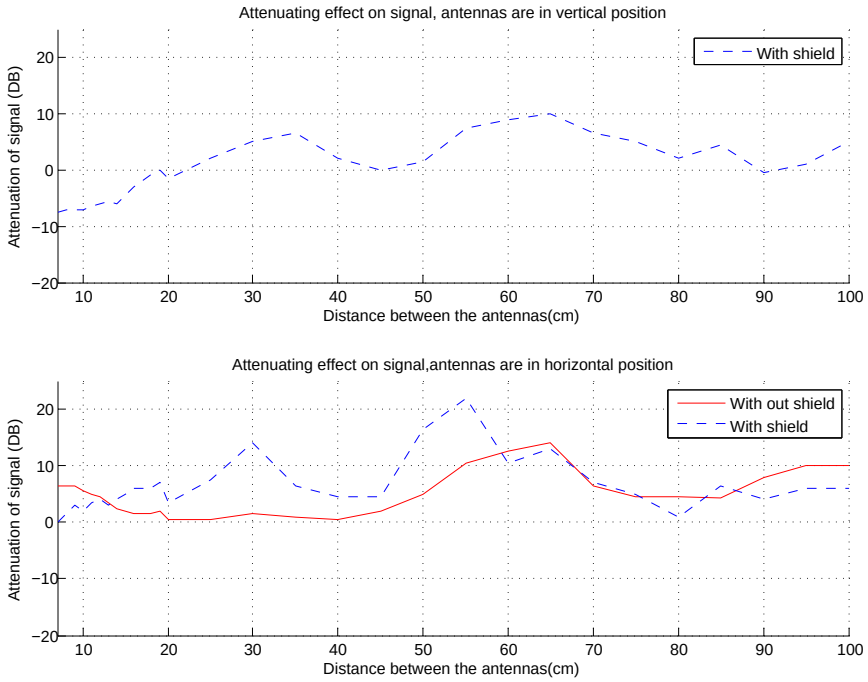


Figure M.18: The figure shows the attenuation effect on the received signal with the use of a shield for both vertical and horizontal position

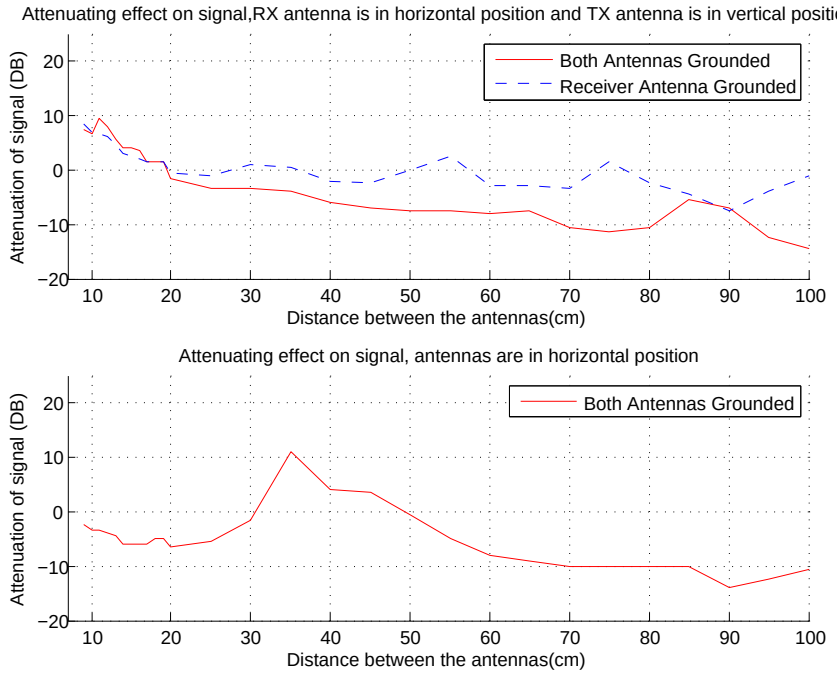


Figure M.19: The figure shows the attenuation effect on the received signal with the use of grounded antennas

The attenuation methods prove very difficult to use, because the results varied a lot. The test with the antennas in vertical position and with use of the shield, showed that within a distance of 20 cm, the signal was amplified and after the 20 cm distance the signal weakened between 0-10 dB for the various distance with no clear relationship between the attenuation effect and the distance.

The test with the antennas in horizontal position, was the only test where the signal did not get amplified, but the attenuation effect still varied a lot for the various distances. The shield did not provide any clear result here either. At a distance of 55 cm the shield weakened the signal with 11 dB more than without the use of it, but at a distance of less than 12 cm, the signal was weaker without the use of the shield than with it.

The test with the receiver antenna in a horizontal position and the transmitter antenna in a vertical position, proved even more inconclusive. At a distance larger than 20 cm the signal was amplified between 0-10 dB. The amplification was larger when both the antennas were grounded compared to when only the receiver antenna was grounded.

The test with both the antennas grounded and their bases facing each other also came with inconclusive results. At a distance of 35 cm the signal was weaken by 10 dB, but

here the attenuation effect spiked and if the antennas were closer to each other or moved away from each other, the attenuation effect would decrease.

The tests in this section have been restricted by the available tools:

1. We would have liked to perform the tests in a sound room, where the walls could have absorbed the signals. The room, where the tests were performed, was a standard office setting with no control over the signals, when they reflected from the surface of the surrounding environment. It is very likely that this had an influence on the reading, since the reflection of the signals could have been in the received signals.

2. We would have liked to do the measurements in dBm and not dB, since that is a absolute value for describing the signal strength and not a relative values that dB is. Unfortunately this information is not available via GNU Radio[36] and the only thing one can measure is the amplitude or magnitude of the signal. To measure the received signal in dBm one would have to place a power meter or spectrum analyser between the receiver antenna and the radio.

3. We would also have liked to test how strong the signal should be for enabling the RFID tags, since none of the tag providers have provided that information about the tags. To get that information one would have to place a voltmeter between the antennas, to check when the voltage is high enough to enable the micro controller on the RFID tag and a power meter or spectrum analyser to get the signal strength.

Appendix N

Pyplot example script and illustration

N.1. Plot window screenshot

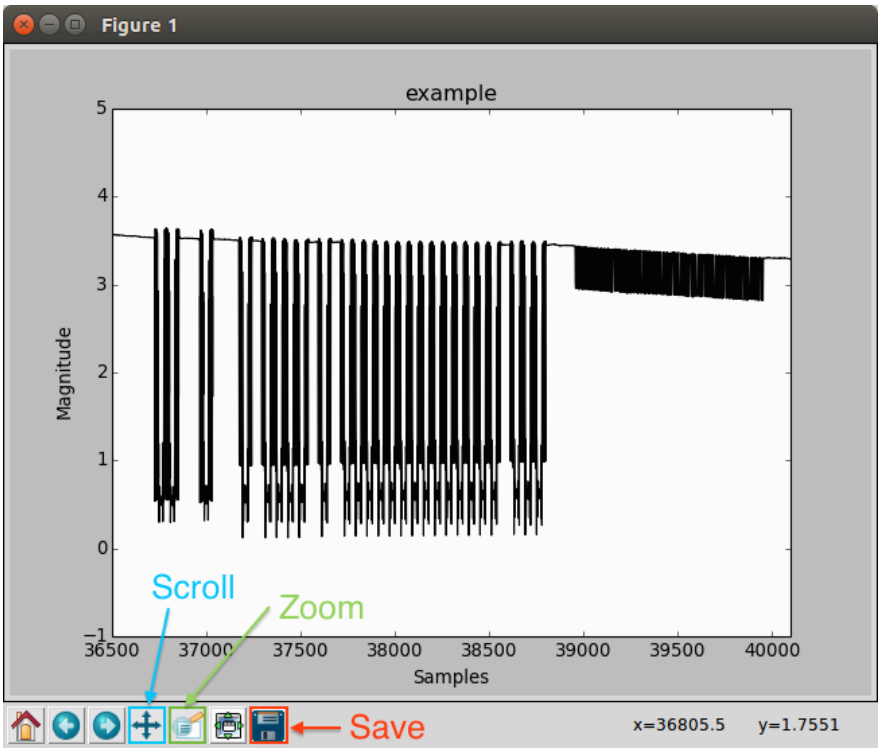


Figure N.1: Example of what a pyplot window looks like. The most important tools (*Zoom*, *Scroll* and *Save*) are annotated with colored arrows and text.

N.2. Pyplot save-file formats

Pyplot supports saving to the following file formats out-of-the-box: .png, .eps, .jpg/jpeg, .pdf, .ps, .raw/rgba, .svg/svgz, .tif/tiff

N.3. Script

```
#!/usr/bin/env python
import scipy
import sys
import numpy as np
import matplotlib.pyplot as plt
import getopt

def main(argv):
# parse input parameters
   inputfile = ''
   try:
      opts, args = getopt.getopt(argv,"hi:",["ifile="])
   except getopt.GetoptError:
      print 'pyplot_example_script.py -i <inputfile>'
      sys.exit(2)
   for opt, arg in opts:
      if opt in ("-i", "--ifile"):
         inputfile = arg
   np.set_printoptions(threshold=sys.maxint)
# read signal from file
   x = scipy.fromfile(open(inputfile),dtype=scipy.float32)
# plot signal
   plt.plot(x, color='k')
   plt.xlim(3.65e4,4.01e4])
   plt.ylim([-1,5])
   plt.ylabel('Magnitude')
   plt.xlabel('Samples')
   plt.title("example")
   plt.show()

if __name__ == "__main__":
   main(sys.argv[1:])
```

Bibliography

[1] C Angerer and R Langwieser. “Flexible evaluation of RFID system parameters using rapid prototyping”. In: *RFID* (2009).

[2] C Angerer, Robert Langwieser, and M Rupp. “RFID Reader Receivers for Physical Layer Collision Recovery”. In: *Communications, IEEE Transactions on* 58.12 (Dec. 2010), pp. 3526–3537.

[3] Christoph Angerer et al. “A flexible dual frequency testbed for RFID”. In: (2008), p. 3.

[4] A Briand, B B Albert, and E C Gurjao. *Complete software defined RFID system using GNU radio*. IEEE, 2012.

[5] M Buettner and D Wetherall. “A flexible software radio transceiver for UHF RFID experimentation”. In: *Univ. Washington, Seattle, WA, UW TR: UWCSE-09-10-02* (2009).

[6] M Buettner and D Wetherall. *A software radio-based UHF RFID reader for PHY/-MAC experimentation*. IEEE, 2011.

[7] Michael Buettner and David Wetherall. “A Gen 2 RFID monitor based on the USRP”. In: *ACM SIGCOMM Computer Communication Review* 40.3 (2010), pp. 41–47.

[8] Michael Buettner and David Wetherall. *An empirical study of UHF RFID performance*. New York, New York, USA: ACM, Sept. 2008.

[9] Frank Buschmann, Kelvin Henney, and Douglas Schimdt. *Pattern-oriented Software Architecture: On Patterns and Pattern Language*. Vol. 5. John Wiley & Sons, 2007.

[10] L Catarinucci et al. “Performance analysis of passive UHF RFID tags with GNU-radio”. In: *2011 IEEE Antennas and Propagation Society International Symposium and USNC/URSI National Radio Science Meeting* (2011), pp. 541–544.

[11] D De Donno, F Ricciato, and L Catarinucci. “Challenge: towards distributed RFID sensing with software-defined radio”. In: 2010.

[12] D De Donno, F Ricciato, and L Tarricone. “Listening to Tags: Uplink RFID Measurements With an Open-Source Software-Defined Radio Tool”. In: *Instrumentation and Measurement, IEEE Transactions on* 62.1 (Jan. 2013), pp. 109–118.

[13] D De Donno, L Tarricone, and L Catarinucci. “Performance enhancement of the RFID epc gen2 protocol by exploiting collision recovery”. In: *Progress In ...* (2012).

[14] D De Donno et al. "Design and applications of a Software-Defined listener for UHF RFID systems". In: *2011 IEEE/MTT-S International Microwave Symposium - MTT 2011* (2011), pp. 1–4.

[15] D De Donno et al. "Increasing performance of SDR-based collision-free RFID systems". In: *2012 IEEE/MTT-S International Microwave Symposium - MTT 2012* (2012), pp. 1–3.

[16] D De Donno et al. "Multipacket reception MAC schemes for the RFID EPC Gen2 protocol". In: *2012 9th International Symposium on Wireless Communication Systems (ISWCS 2012)* (2012), pp. 311–315.

[17] Sylvain Munaut Dimitri Stolnikov with contributions from Hoernchen Steve Markgraf and Nuand LLC. *gr-osmoSDR open source driver for GNU Radio*. `http://sdr.osmocom.org/trac/wiki/GrOsmoSDR`. [Online; accessed 23-Feb-2015]. 2012.

[18] Daniel M Dobkin. *The RF in RFID: UHF RFID in Practice*. Newnes, 2012.

[19] *Electromagnetic compatibility and Radio spectrum Matters (ERM); Radio Frequency Identification Equipment operating in the band 865 MHz to 868 MHz with power levels up to 2 W and in the band 915 MHz to 921 MHz with power levels up to 4 W*. 2.1.1. [Deliverable No.: EN 302 208 (1 and 2)]. ETSI. Feb. 2015.

[20] *EPC/RFID UHF Air Interface Protocol*. 2.0.0. GS1. Nov. 2013.

[21] Fairwaves. *UmTRX v2.2*. `http://umtrx.org/hardware/`. [Online; accessed 23-Feb-2015].

[22] Bruce A Fette. *Cognitive radio technology*. Academic Press, 2009.

[23] SDR Forum (now Wireless Innovation Forum). *SDRF Cognitive Radio Definitions v1.0.0*. `http://www.sdrforum.org/pages/documentLibrary/documents/SDRF-06-R-0011-V1_0_0.pdf`. [Online; accessed 23-Feb-2015]. Nov. 2007.

[24] Wireless Innovation Forum. *What is Software Defined Radio?* `http://www.wirelessinnovation.org/what_is_sdr`. [Online; accessed 23-Feb-2015].

[25] GS1. *Regulatory status for using RFID in the EPC Gen 2 band (860 to 960 MHz) of the UHF spectrum*. `http://www.gs1.org/docs/epc/UHF_Regulations.pdf`. [Online; accessed 23-Feb-2015]. Oct. 2014.

[26] Wireless Innovation. *Wireless Innovation*. `http://www.wirelessinnovation.org`. [Online; accessed 23-Feb-2015].

[27] Texas instruments. *MSP430F2132*. `http://www.ti.com/product/MSP430F2132/description`. [Online; accessed 28-Feb-2015]. 2013.

[28] J Kimionis, A Bletsas, and J N Sahalos. *Design and implementation of RFID systems with software defined radio*. IEEE, 2012.

[29] H Liu et al. "Towards adaptive continuous scanning in large-scale rfid systems". In: *in Proc of IEEE INFOCOM* (2014).

[30] P. V. Nikitin and K. V. S. Rao. "LabVIEW-Based UHF RFID Tag Test and Measurement System". In: *Industrial Electronics, IEEE Transactions on* 56.7 (July 2009), pp. 2374–2381.

[31] P V Nikitin and KVS Rao. "Antennas and propagation in UHF RFID systems". In: *challenge* (2008).

[32] Nuand. *bladeRF*. `http://nuand.com/`. [Online; accessed 23-Feb-2015]. 2013.

[33] George Nychis et al. "Enabling MAC Protocol Implementations on Software-Defined Radios." In: *NSDI*. Vol. 9. 2009, pp. 91–105.

[34] Michael Ossmann. *HackRF One*. `https://greatscottgadgets.com/hackrf/`. [Online; accessed 23-Feb-2015]. 2012.

[35] GNU Radio. *GNU Radio Hardware Guide*. `https://gnuradio.org/redmine/projects/gnuradio/wiki/Hardware`. [Online; accessed 23-Feb-2015]. 2015.

[36] GNU radio. *Frequently Asked Questions*. `http://gnuradio.org/redmine/projects/gnuradio/wiki/FAQ`. [Online; accessed 27-Feb-2015]. 2006.

[37] Ettus Research. *USRP Hardware Driver for USRP*. `http://code.ettus.com/redmine/ettus/projects/uhd/wiki`. [Online; accessed 27-Feb-2015]. 2006.

[38] Ettus Research. *USRP N210*. `http://www.ettus.com/product/details/UN210-KIT`. [Online; accessed 23-Feb-2015].

[39] Ettus Research. *USRP1*. `http://www.ettus.com/product/details/USRPPKG`. [Online; accessed 23-Feb-2015].

[40] Thomas Schmid, Oussama Sekkat, and Mani B Srivastava. *An experimental study of network performance impact of increased latency in software defined radios*. ACM, Sept. 2007.

[41] Joshua R Smith et al. "A Wirelessly-Powered Platform for Sensing and Computation." In: *Ubicomp* 4206.Chapter 29 (2006), pp. 495–506.

[42] S Tung and A K Jones. "Physical layer design automation for RFID systems". In: *Distributed Processing Symposium (IPDPS)* (2008), pp. 1–8.

[43] C.R. Valenta and G.D. Durgin. "R.E.S.T. A flexible, semi-passive platform for developing RFID technologies". In: *Sensors, 2012 IEEE*. Oct. 2012, pp. 1–4.

[44] Gunjan Verma and Paul Yu. *A MATLAB Library for Rapid Prototyping of Wireless Communications Algorithms with the Universal Software Radio Peripheral (USRP) Radio Family*. Tech. rep. DTIC Document, 2013.

[45] Jue Wang and Dina Katabi. "Dude, where's my card?: RFID positioning that works with multipath and non-line of sight". In: *ACM SIGCOMM Computer Communication Review*. Vol. 43. 4. ACM. 2013, pp. 51–62.

[46] Jue Wang et al. "Efficient and reliable low-power backscatter networks". In: *ACM SIGCOMM Computer Communication Review* 42.4 (Sept. 2012), pp. 61–72.

[47] Jue Wang et al. "Securing deployed rfids by randomizing the modulation and the channel". In: (2013).

[48] Wikipedia. *List of software-defined radios*. `http://en.wikipedia.org/wiki/List_of_software-defined_radios`. [Online; accessed 23-Feb-2015]. 2015.

[49] P Zhang and D Ganesan. "Enabling bit-by-bit backscatter communication in severe energy harvesting environments". In: *Proceedings of the 11th USENIX Symposium*. 2014.

[50] Pengyu Zhang, Jeremy Gummeson, and Deepak Ganesan. *BLINK: a high throughput link layer for backscatter communication*. a high throughput link layer for backscatter communication. New York, New York, USA: ACM, June 2012.

[51] Yuanqing Zheng. *usrp2reader*. `https://github.com/yqzheng/usrp2reader`. [Online; accessed 23-Feb-2015]. June 2013.

[52] Yuanqing Zheng and Mo Li. "ZOE: Fast cardinality estimation for large-scale RFID systems". In: *IEEE INFOCOM 2013 - IEEE Conference on Computer Communications* (2013), pp. 908–916.