POLITECNICO DI TORINO

Master of Science Program in Electronic Engineering

Master's Degree Thesis

Software-Defined Radio Implementation of a LoRa Detector and Transceiver

Supervisors (PoliTO)
Dr. Roberto Garello
Dr. Daniel Gaetano Riviello
M.Sc. Simone Scarafia
Supervisor (USP)
Dr. Cristiano Magalhães Panazio

Student
João Pedro de Omena Simas

Academic Year 2020-2021

Abstract

The number of applications of low-power wide-area networks (LPWANs) has been growing quite considerably in the past few years and so has the number of protocol stacks. Despite this fact, there's still no fully open LPWAN protocol stack available to the public, which limits the flexibility and ease-of-integration of the existing ones. The closest to being fully open is LoRa, however only its medium access control (MAC) layer, know as LoRaWAN, is open and its physical and logical link control layers, also known as LoRa PHY, are still only partially understood.

In this thesis, the essential missing aspects of LoRa PHY are not only reverse-engineered, but also a new design of the transceiver and its sub-components is proposed and implemented in a modular and flexible way using GNU Radio.

Finally, some examples of applications of both the transceiver and its components, which are made to be run in a simple setup using cheap widely available of-the-shelf hardware, are given to show how the library can be used and extended.

Contents

List of Figures

List of Tables

List of Acronyms

BPSK	Binary phase shift keying
BW	Bandwidth
Cr	Code rate
CRC	Cyclic redundancy check
CSS	Chirp spread spectrum
DBPSK	Differential binary phase shift keying
DFT	Discrete Fourier transform
DTFT	Discrete-time Fourier transform
GF(2)	Galois Field of order 2
HN	High Nibble
IETF	Internet Engineering Task Force
IoT	Internet of things
IP	Internet Protocol
IC	Integrated circuit
ISM	Industrial, scientific and medical radio nand
LFSR	Linear-feedback shift register
LLC	Logic link control
LN	Low nibble
LPWAN	Low-power wide-area network
LSB	Least-significant bit

MAC	Medium access control
MSB	Most-significant bit
NLMS	Normalized least mean squares
OSI	Open Systems Interconnection
QPSK	Quadrature phase shift keying
RAM	Random-access memory
RFTDMA	Random frequency time-division multiple access
SDR	Software defined radio
SF	Spreading factor
TCP	Transmission control protocol

Chapter 1

Introduction

Low-power wide-area networks, commonly referred to as LPWANs, are networks whose main characteristics are having large coverage areas, low-power consumption (many times involving battery operated devices) and low data rates. The number of applications of this kind of technology has been growing in the past few years, especially with the rise in interest on Internet of Things applications, many times related to the implementation of wireless sensor networks.

Despite this big interest in this field and the growing number of applications of LPWANs, it still lacks a completely open protocol stack. The closest to this available at the moment is LoRa, better said the LoRa protocol stack, however only its upper layer, i.e. the medium access control layer, is open.

The LoRa Protocol stack, usually referred simply as LoRa, is usually subdivided in two parts: LoRa PHY, which, contrary to what the name implies, encompasses not only the physical layer aspects of this protocol stack, but also the logical link control sublayer of the data link layer, using a more conventional Open Systems Interconnection (OSI) model nomenclature; and LoRaWAN, which comprises the medium access control sublayer of the OSI data link layer.

While LoRaWAN is open and publicly available [1], LoRa PHY is not. Some attempts have been made to reverse-engineer it and or propose designs for its demodulator [2], [3], but some details still have remained not fully understood which has prevented the implementation of a fully working free and open-source LoRa PHY transceiver.

The main objective of this thesis is to continue the previous work on reverse-engineering the missing details of LoRa's lower layers and implementing a fully free and open-source software-defined transceiver, with the goal of allowing for further research and development to be done on this protocol stack by the scientific community. Also, this implementation is made to be modular, to give more flexibility and ease the implementation of custom solutions in the lower levels to users of this technology.

Finally, four example applications of the developed receiver and transmitter are

given, both in terms of hardware and extra software. The first one to illustrate how individual blocks can be used to implement simpler applications, which do not require the full transmitter/receiver, in the form of a detector that can detect frames being transmitted and work out which spreading factor and which channel was used to transmit it and derive statistics from it, in order to analyze the traffic in the network. And the last three to illustrate how the developed receiver and transmitter blocks can be used to implement more complex systems, that extend its functionality using GNU Radio's stock blocks.

Chapter 2

LPWANs and the LoRa Protocol Stack

In order to illustrate the context within which the LoRa protocol stack is situated in the framework of LPWANs, a brief description on some key topics is given in this section.

2.1 Low-Power Wide-Area Networks

Given that there's no standard defining what exactly classifies as a Low-Power Wide-Area Network, commonly referred to as a LPWAN, it is not simply a Network that covers a wide area and employs low-power devices. One attempt to define it, given the current applications that identify themselves as LPWANs, is the RFC8375 [4] published by the Internet Engineering Task Force (IETF). It sums up the their main characteristics as:

> Most technologies in this space aim for a similar goal of supporting large numbers of very low-cost, low-throughput devices with very low power consumption, so that even battery-powered devices can be deployed for years. LPWAN devices also tend to be constrained in their use of bandwidth, for example, with limited frequencies being allowed to be used within limited duty cycles (usually expressed as a percentage of time per hour that the device is allowed to transmit). As the name implies, coverage of large areas is also a common goal. So, by and large, the different technologies aim for deployment in very similar circumstances.

Also, some important structural similarities between all the existing LPWAN "technologies", name given to protocol stacks combined with network topologies, are pointed out, mainly them being organized in terms of:

- **End devices** that communicate with Radio Gateways via a wireless link;
- **Radio Gateway** that connects to **end devices** using the LPWAN protocol and to a **network gateway** using TCP/IP;
- **Network Gateway** that connects the **radio gateway** to the internet (i.e. to an Application server);
- **Authentication Server** that handles authentication, the joining of new devices to the network and the assignment of encryption keys. This might be implemented in the same hardware as the Network Gateway.

Some of the most widely used LPWAN technologies include LoRa (i.e. LoRaWAN + LoRa PHY), Sigfox and NB-IoT. Below is a brief comparison table of these three technologies, in terms of their uplink communication characteristics, in the physical, LLC and MAC layers:

Table 2.1: Comparison of selected Characteristics of LoRa, NB-IoT and Sigfox.

	LoRaWAN + LoRa PHY
Band	Industrial, scientific and medical radio nand (ISM)
Modulation	CSS (see section 2.3)
Coding	Parity Bit (code rate (Cr) = 4/5) (See section 3.2.4.1)
Bandwidth	125, 250 or 500 kHz [5]
Physical Bit Rate	0.37 - 27.4 kbit/s (See section 2.3)
Multiplexing	Different-SF Chirp Interference Resistance (See section 4.1)
Channel Access Mechanism	Deterministic Time Slots [1]
	NB-IoT (Random Access Channel, Single Subcarrier)
Band	Same as LTE
Modulation	SC-FDMA (BPSK/QPSK) [6]
Coding	Turbocode (Cr = 1/3)
Bandwidth	3.75 or 15 kHz [6]
Physical Bit Rate	3.75, 7.5, 15 or 30 kbit/s [6]
Multiplexing	Subcarrier hopping [7]
Channel Access Mechanism	4-way handshake [7]
	Sigfox
Band	ISM
Modulation	Differential binary phase shift keying (DBPSK) [8]
Coding	None or Convolutional-(1,5,7) (Cr = 1/3) [8]
Bandwidth	0.1 kHz or 0.6 kHz [8]
Physical Bit Rate	0.1 kbit/s or 0.6 kbit/s [8]
Multiplexing	RFTDMA [8]
Channel Access Mechanism	Deterministic Time Slots [9]

As it can be seen, although each of the three can be classified as LPWAN technologies, they do so while following very distinct design paradigms.

Sigfox takes a minimalistic approach, employing one of the simplest modulations possible, making the hardware equally simple, but ending up sacrificing bit rate.

NB-IoT is designed around the constraint of implementing an internet of things (IoT)/LPWAN oriented protocol stack, while re-using as much as possible all the existing LTE standards. This allows for existing components, software, and, more importantly, infrastructure built for LTE to be re-used for NB-IoT. This constraint, however, makes it so NB-IoT inherits much of the complexity of the LTE protocol stack that is arguably unnecessary in this context.

Finally, LoRaWAN is built with the main goal of being a flexible LPWAN protocol stack that takes advantage of the benefits of the LoRa chirp spread spectrum (CSS) modulation. This gives it all the flexibility that Sigfox lacks, while, because it is designed basically specifically for LPWAN applications, not having the complexity that comes with NB-IoT, and on top of that carries all the benefits the CSS modulation has over conventional ones.

2.2 LoRaWAN

According to LoRaWAN's specification [1], LoRaWAN is a MAC layer protocol made to run on top of the LoRa PHY physical layer, with the main goal of running networks of battery-powered end-devices, which should run continuously for a long time.

In terms of network topology, usually LoRaWAN network follows a "star-of-stars" topology with the usual organization described in the previous section on LPWANs 2.1, where multiple gateways connect to a network server and the end devices connect to one or more gateways.

It is also important to note that LoRaWAN is designed to operate reliably on ISM bands which are to be shared with other devices using a variety of other protocols. To cope with this issue, in addition to the LoRa CSS modulation, it uses some MAC-layer techniques, such as transmitting the message multiple times, hopping between channels in a pseudo-random fashion and changing spreading factors, and consequently bit rates, in order to improve transmission robustness if needed.

Finally an important aspect of LoRaWAN, is that it up uplink-focused, i.e. it gives special importance to messages between the end-devices and the gateways and most of the MAC behavior is triggered by these. The main reflection of this fact is that in the most common form of operation (communication between A-class devices and gateways), downlink communication is only allowed in time slots located at fixed delays after the transmission of an uplink message.

2.3 LoRa Chirp Spread Spectrum Modulation

The CSS (Chirp Spread Spectrum) modulation used by LoRa PHY consists in using as symbols linear chirps, that is signals with the form:

$$x_i(t) = Ae^{j\pi\beta mod(t-T_s\frac{i}{N_{sym}},Ts)^2}, i \in \{0, ..., N_{sym}\}, t \in [0, T_s[\tag{2.1}$$

Where:

- $\mathbf{T_s}$ is the symbol period
- $\boldsymbol{\beta}$ is a parameter which determines the speed of growth of the instantaneous frequency of that signal, which will be referred in the rest of this thesis as the *chirp rate*.
- **mod(a, b)** is the real number remainder function defined as:

$$mod(a, b) = min\{r \in \mathbb{R}^+, \exists q \in \mathbb{Z}, a = qb + r\} \tag{2.2}$$

- $\mathbf{N_{sym}}$ The total number of symbols in the CSS modulation. The parameters used my the LoRa PHY modulation are:
  - **Bandwidth**, the mean bandwidth of the signal, which relates with T_s and β as:

$$BW_{CSS} = \beta T_s \tag{2.3}$$

    This can be set to a set of predefined values, described in [10], however the LoRaWAN specification only uses 125kHz and optionally 250kHz for uplink messages and 500kHz for downlink messages.
  - **Spreading Factor** It is not only the number of bits each symbol encodes, i.e $N_{sym} = 2^{SF}$, but is also constrained to relate to the previously mentioned parameters, such that:

$$2^{SF} = BW_{CSS}T_s \tag{2.4}$$

To illustrate the effects of these parameters, below a table and a graph with the resulting physical bit rates with some of the possible parameter combinations, mainly the ones used in LoRaWAN, are shown below:

Table 2.2: Physical bit rate values (in kbit/s) for selected combinations of bandwidth (BW) and spreading factor (SF) ($R_b = BW \cdot 2^{-SF} \cdot SF$).

BW(kHz) \ SF	7	8	9	10	11	12
125	6.84	3.91	2.2	1.22	0.67	0.37
250	13.67	7.81	4.39	2.44	1.34	0.73
500	27.34	15.63	8.79	4.88	2.69	1.46

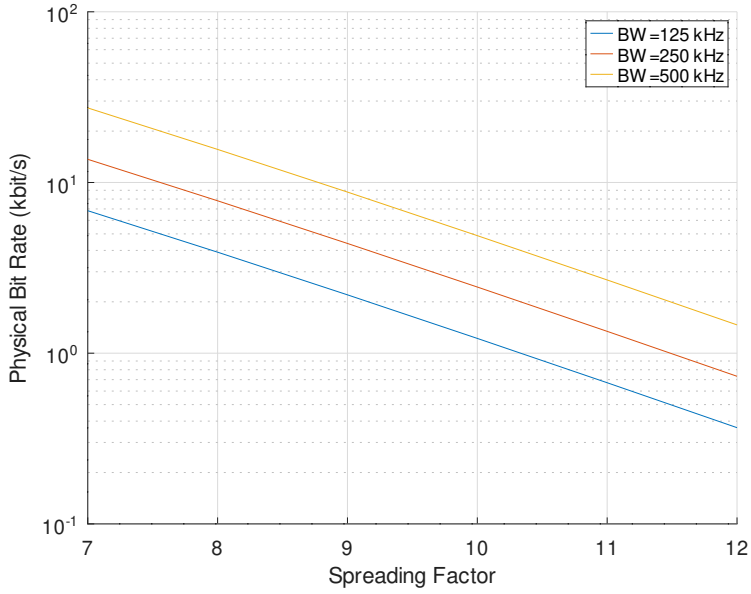

Figure 2.1: Physical bit rate in function of spreading factor (SF) and bandwidth (BW) for some of the possible values.

Chapter 3

Reverse Engineering The LoRa PHY

3.1 Previous Works

3.1.1 General Transmitter Structure

In [2], [3] similar structures for the LoRa receiver are proposed. Given those and the information provided in [10], we can summarize those structures as being composed of the following blocks connected sequentially:

- **Linear Encoder**

  Takes the bits and encodes them with a (CR + 4)/4 code, except for the data bits, which are always encoded with a 8/4 code.

- **Interleaver**

  An usual diagonal interleaver, that turns the sequence (CR + 4)-bit code words into a sequence of SF-bit length words, with the exception of the header bits, which are interleaved with a 8-bit to (SF - 2)-bit interleaver [1].

[1] As mentioned above, the header bits are interleaved in a different manner than the payload bits. Actually, as mentioned in [3], it is the header bits plus the number of payload bits to complete 8 symbols, the 5 header code words plus SF - 7 payload code words with 4 - CR zeros added to most-significant bits (MSBs) to extend them to also have 8 bits, therefore resulting in 8*(SF - 2) bits. They are then interleaved with a SF - 2 interleaver and the resulting symbol numbers are multiplied by 4, resulting in eight symbols in the $\{0; 4; 8; ...; 2^{SF} - 4\}$ range. This makes the header bits have more robustness against noise.

- **Randomizer / Data Whitening**

  Takes a sequence of pseudo-random numbers with the same bit-length as the output of the Interleaver, referred in [2] and [3] as a whitening sequence, and exclusive-ors it with the said output.

- **Gray Decoder**

  An usual gray decoder that turns the SF-bit (or (SF - 2) if inside the header) words into symbol numbers.

- **Modulator**

  A modulator that generates the chirps corresponding to each symbol number.

- **Append Preamble**

  Adds a fixed preamble the beginning of the resulting signal.

3.1.2 Physical Layer - Packet Structure

The LoRa PHY packet is composed of a preamble, followed by LoRa PHY's logical link control layer frame, modulated in CSS.

1. **Preamble**

   The preamble is composed by:

   - A sequence of consecutive up chirps, which can be of length 6 to 65535, as mentioned in [10];
   - Two symbols, referred as a sync word, used for network identification;
   - Two and a quarter down chirps;

3.1.3 Logic Link Control Layer Structure

1. **Coding**

   Both [2] and [3] cite that the linear encoding used in LoRa PHY is some form of Hamming encoding and in [3] explicitly describe it for CR = 4, as the usual form of Hamming(7, 4), mainly a code with generator matrix:

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

   With a permutation with the following permutation applied to the coded word:

$$\begin{pmatrix} 5 & 0 & 1 & 2 & 4 & 3 & 6 & 7 \end{pmatrix}$$

Represented in one-line notation.

2. **Randomizer**

   Also referred as a whitening block, the randomizer is partially reverse-engineered in both [2] and [3] in a direct approach by assuming randomization is done by exclusive "or'ing" a fixed arbitrary sequence with the output of the interleaver. These sequences then were then extracted by sending all-zeros payload messages and looking at the resulting de-interleaved symbols, that should be the same as the whitening sequence. Note that, in this way, they depend on the modulation parameters (SF, CR and if lowDataRate is used). These sequences can be found in [11].

3. **Frame Structure**

   In [3] the following header structure is proposed:

   Table 3.1: General Structure of the LoRa LLC frame.

Starting Bit	Function
0	Payload Length (1 byte)
8	CR (3 bits)
11	CRC Present (1 bit)
12	Header Checksum HN (4 bits)
16	Header Checksum LN (4 bits)
20	Payload (0 to 255 bytes)
20 + 8*(Payload Length)	Payload "CRC" (2 bytes) (optional)

   Where HN (bits 4 to 7) indicates the high nibble and LN the low nibble (bits 1 to 3) of a byte.

   Each part of the Frame is described in detail in the following sections.

   (a) **Payload Length**

   The length of the payload in bytes.

   (b) **CR**

   The CR parameter of the LoRa PHY modulation as described in the Semtech documentation [12], which is not the code rate, but the difference between the code length and the code rank, this second being always 4.

   (c) **CRC Present**

   Single bit that indicates if the payload "CRC" is present or not. If it is '1', the it is present, if it is '0' it is not.

(d) **Header Checksum**

A checksum calculated from the first 12 bits of the header. Its presence is mentioned in [3], but its exact structure is not mentioned, only that its 5 least-significant bits (LSBs) are non-zero.

(e) **Payload "CRC"**

The cyclic redundancy check (CRC) sum of the payload. It's mentioned in [2] and [3], but its exact structure was not known.

3.2 Findings

3.2.1 Experimental Setup

To do the following analysis, a combination of the data provided in gr-lora-samples and data captured with a simple experimental setup was used.

This setup consisted in a board based on the Atmega328P micro-controller connected to a module based on the SX1278 IC, which, among some other features, is a LoRa transceiver, and a RTL-SDR USB software defined radio receiver connected to a PC. The first was programmed to transmit signals with the desired parameters and then they were received and recorded using the RTL-SDR and a simple flow-graph in GNU Radio, to be subsequently decoded using a piece of software written on MATLAB/Octave developed for this project [13].

3.2.2 General Transmitter Structure

After reverse engineering the whole protocol, process whose details will be described on following sections, the new revised structure below is proposed.

- **Randomizer / Data Whitening**

  A randomizer which adds (modulo-2 addition, i.e. exclusive or) a pseudo random sequence of bytes generated by a linear-feedback shift register (LFSR) which is independent from the modulation parameters. Note that this sequence is only added to the payload bytes and not to the header nor the payload "CRC" (refer to the section on the structure of the frame header (3.2.4.3) for more details).

- **Linear Encoder**

  The exact same as described in section 3.1.1, but in the following section on coding a new, and more intuitive, description of the codes used is given.

- **Gray Decoder**

  The exact same as described in section 3.1.1.

- **Interleaver**

  The exact same as described in section 3.1.1.

- **Modulator**

  The exact same as described in section 3.1.1.

- **Append Preamble**

  The same as described in section 3.1.1, with the exception of a slight change on the size of the section of downchirp (see following section on packet structure (3.2.3) for more detail).

3.2.3 Physical Layer - Packet Structure

As previously mentioned in the section on previous works (3.1), the packet is composed by the LLC payload preceded by a preamble. The structure described there seems to be correct, with a slight correction: the down chirps where found to be slightly shorter than two and a quarter. A length of $2 + 1/4 - 2^{-SF}$ symbols was determined empirically. The need for this change arose by observations during testing of the proposed receiver structure (see section 4), as there was a systematic time offset in synchronization. Also, while using the receiver implemented in [11], that is based on the description from [3], for large frames (with payloads with length close or equal to 255 bytes) with a known input and looking at how the received symbol numbers drift monotonically from the expected ones, we can see that there's some inaccuracy in the synchronization, further giving evidence to this hypothesis.

As a side-note, this detail is probably the reason why the whitening/randomizer was not reverse engineered on previous papers. For more details on this look in the section on the randomizer (3.2.4.2).

3.2.4 Logic Link Control Layer Structure

1. **Coding**

   By applying the bit permutations to the to the generator matrices of the codes proposed in [2] and [3] and implemented in [11], in order to not need the extra bit-permutation step new generator matrices were obtained. Also the nibbles of the coded word were inverted, to get a canonical code, which also resulted in a more intuitive structure of the randomizer (see section 3.2.4.2) giving more evidence that this is indeed the intended bit ordering. For each of the possible CR values, the obtained generator matrices are listed below:

- **CR = 1**

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

  Which is a simple parity bit code.

- **CR = 2**

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

- **CR = 3**

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

- **CR = 4**

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Note that all the obtained linear codes are canonical and, with the exception of the first code, which is a simply a parity bit, the three others are all derived from Hamming(7,4). For CR = 3, the code is a canonical version of it, whereas for CR = 2 the code is a reduced version this code obtained by removing the last bit, which results in a canonical (6,4) code and the last one (CR = 4) is a form of the extended Hamming(8,4) code obtained by adding an extra parity bit to the form of Hamming(7,4) used for CR = 3.

2. **Randomizer**

   By taking the whitening sequences obtained in [3], used in [11], passing them through the decoder and then grouping the obtained bits in a 8xN column bit matrix and then running Berkelamp-Massey's algorithm [2] to the first 100 bytes, it can be seen that the sequence can be generated by an degree-8 LFSR with the following polynomial:

[2]Berkelamp-Massey's algorithm is an algorithm that finds the shortest LSFR that encodes a given sequence in GF(2). This approach is equivalent to solving the linear system that arises from the LSFR structure for a number of data points ($b_{i+N} = \sum_{i=0}^{N-1} b_{i-1-k}a_k$ for $i \in \{0, ..., M-1\}$) using Gaussian elimination in GF(2).

$$P(x) = x^0 + x^3 + x^4 + x^5 + x^7 \tag{3.1}$$

The state of this LFSR is modulo-2 added to each the data bytes to act as a randomizer.

Furthermore, it's worth noticing that the sequences provided by Robyns in [11] diverge from those generated by this LFSR after a certain position, but this is probably due to a symbol rate offset or an imprecision on the alignment that caused the last symbols of the message not to be aligned when using the receiver used in [11]. This conclusion was reached because the symbol numbers at the end of a 255-byte length frame slowly drift from the ones obtained in this work.

It has to be pointed out that the whitening sequences proposed in [3] do work. Even with this problem, as the symbol number offsets caused by the time drift due to the inaccuracy in synchronization are deterministic and therefore can be compensated in the de-randomization process by changing the whitening sequence, like it was done, unknowingly, in [3].

3. **Frame Structure**

By analyzing the structure of the header, and given the previous description given in [3], the structure of the frame was found to be:

Table 3.2: Proposed General Structure of the LoRa LLC frame.

Starting Bit	Function
0	Payload Length HN (4 bits)
4	Payload Length LN (4 bits)
8	CR (3 bits)
11	CRC Present (1 bit)
12	Header Checksum HN (4 bits)
16	Header Checksum LN (4 bits)
20	Payload (0 to 255 bytes)
20 + 8*(Payload Length)	Payload "CRC" (2 bytes) (optional)[3]
20 + 8*(Payload Length + 2)	Padding Nibbles [4]

[3]The payload "CRC" was found to not be in the usual ordering and not to be an usual CRC sum. Check its section for more details.

[4]These padding nibbles are added so the encoding of the payload plus the payload nibbles plus the CRC, if present, plus these extra nibbles are a multiple of SF and therefore give rise, after the interleaver, to a whole number of symbols. How exactly this numbers are generated in real hardware is not clear, but they seem to make no difference as, while testing the implemented

Where HN (bits 4 to 7) indicates the high nibble and LN the low nibble (bits 1 to 3) of a byte.

Note that the nibbles might seem to be inverted to that described in [3] and in section 3.1.3.3. This is due to the change in the description of the coding.

Each part of the Frame is described in detail in the following sections.

(a) **Payload Length**
The length of the payload in bytes. Note that the two nibbles are in a "little endian-like" ordering with the upper nibble coming first and then the lower nibble.

(b) **CR**
The CR parameter of the LoRa PHY modulation as described in the Semtech documentation [12], which is not the code rate, but the difference between the code length and the code rank, this second always being equal to 4.

(c) **CRC Present**
Single bit that indicates if the payload "CRC" is present or not. If it is '1', then it is present, if it is '0' it is not.

(d) **Header Checksum**
A checksum calculated from the first 12 bits of the header. The same "little endian-like" ordering used in the payload length was assumed. As described in Robyn's work [3], 3 of the bits of the checksum are always 0, which in the assumed ordering are the last 3 bits (5 to 7). After doing some analysis it was found that the checksum can be calculated as:

$$c = Gh \tag{3.2}$$

Where c is the bit column vector representation of the checksum and h the one of the first 12 bits of the header and G a matrix in $\mathbb{M}_{8\times12}(GF(2))$, where GF(2) denotes the order-2 Galois field. And G is equal to:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

transmitter, they were set to zero, which is not what usually what happens in the commercial transmitters, and still functioned normally.

i. **Reverse-Engineering Methodology**
In order to reverse engineer this checksum, initially the reasonable assumption that the bits of the header checksum is a linear function (in modulo-2 arithmetic) of the data bits of the header was made. Then by decoding the headers of frames from captured waveform files, putting them as rows of a matrix and then applying the Gauss-Jordan elimination algorithm in modulo-2 arithmetic. In this way, if we start with a matrix with 12 linearly independent rows (in the canonical linear space in $GF(2)^{12}$) we get a matrix M, such that:

$$M = \begin{pmatrix} I & G^T \end{pmatrix}$$

Where I is an 12x12 identity matrix.

(e) **Payload "CRC"**

The payload "CRC" was found to be calculated by taking the polynomial in GF(2) relative to the payload data in little endian ordering and taking the remainder of its division with the polynomial $x^0 + x^5 + x^{12} + x^{16}$ and then taking the corresponding bit string and storing it in big endian ordering. Note that this division is equivalent to computing the CRC sum of the data starting from the second byte (in little endian order) with the above mentioned polynomial and using as initial value of the algorithm the first byte.

i. **Reverse-Engineering Methodology**
Initially, given what was already known, it was assumed that the CRC sum was really a CRC sum and had a degree-16 polynomial. In order to reverse engineer the payload CRC, using the before mentioned test setup, frames whose payload was the powers of two, i.e. only one '1' byte and all the others '0', with length 1 up to 7 were transmitted and captured and decoded. For the sake of simplicity all byte orderings referred in the rest of this description are big endian. When observing the data, it can be seen that when only the last 2 bytes are non-zero, the CRC is the same as those 2 last bytes, but with the two bytes in opposite ordering. Given this, three things were inferred:

- That the byte ordering of the CRC was the the opposite of the one of the data.
- That the checksum was not a direct CRC sum, but the direct remainder of the polynomial division of the data without multiplying its polynomial by x^n, where n is the order of the CRC.
- That the data is used in this calculation is in little endian order, because the CRC sum was the same as the data but inverted when only two non-zero bytes are transmitted.

Given those three, if the last bytes of the payload are zero, the checksum is equivalent to an usual CRC sum of the data in little endian mode and ignoring those last two bits. Therefore, it was possible to use the open source tool CRC RevEng [14] to find the polynomial, if this checksum was indeed a CRC. This program tests a collection of know used CRC polynomials and checks if any are consistent with the given data + CRC's. By running it with payload data, the program yielded the polynomial 0x1021 and indicated that the data was indeed taken in little endian ordering. For further testing, with all power-of-two-length payloads it was verified that this CRC coincided with the one calculated by the commercial transceiver for payload lengths 1 up to 7 bytes.

Chapter 4

Receiver Structure

As the transmitter structure was already described on section 3.2.2 to illustrate the structure of the frame, in this section, an structure for the receiver is now proposed to fill in the missing elements needed to effectively implement a receiver that are not present in the transmitter, namely synchronization, demodulation and decision.

In order to structure the analysis of the Receiver, the system was divided into 4 main blocks:

- Frequency Estimator
- Correlation Synchronizer
- Symbol Decision
- Frame Decoder

4.1 Frequency Estimator

The goal of this block is to estimate the instantaneous frequency of the signal, given some known chirp rate, or equivalently an symbol factor and a CSS bandwidth. This can be formally defined as finding some $\omega_k \in]-\pi;\pi]$, such that minimizes:

$$J(\omega_k) = E[\Sigma_{i\in -N}^{N} |x_{i+k} - cos\left(\pi\beta i^2\right) e^{j\omega_k i}|^2] \tag{4.1}$$

Or, equivalently, that maximizes:

$$K(\omega_k) = E[|\Sigma_{i\in -N}^{N} x_{i+k} cos\left(\pi\beta i^2\right) e^{-j\omega_k i}|^2] = E[|DTFT[(x_{i+k} cos\left(\pi\beta i^2\right))_{i\in\{0,\dots,N-1\}}](\omega_k)|^2] \tag{4.2}$$

for some $N \in \mathbb{N}$, where x_i is the received signal.

Note that $cos(\pi\beta i^2) = 0.5(e^{\pi\beta i^2} + e^{-\pi\beta i^2})$ is used instead of $e^{-\pi\beta i^2}$ as the LoRa CSS signal uses not only up-chirps, but also down-chirps, these second for synchronization, as described in the section about the modulation, and by using the sum of a down- and upchirp, when multiplied by this reference signal, the signal vector will present a peak on it's spectrum at it's middle frequency when either chirp is present.

To estimate this value two approaches are proposed.

1. **Stochastic Gradient Descent**

   If it is assumed there's no interfering signal other than Gaussian noise, this problem can be approximated with a simpler optimization problem using only one sample, i.e. to minimize the cost function:

   $$J(w_i) = E[|x_{i+1}e^{-j2\pi\beta} - w_i^* x_i|^2], w \in \mathbb{C} \tag{4.3}$$

   And using $\hat{f} = -\frac{arg(w)}{2\pi}$ as the instantaneous angular frequency estimate. The solution of this problem can be easily estimated using stochastic gradient descent, i.e. a order-1 adaptive filter. For the reverse-engineering done in this work, an order-1 normalized least mean squares (NLMS) filter was used, whose coefficient-update function reduces to:

   $$w_{i+1} = w_i(1-\mu) + \mu(\frac{x_{i+1}e^{-j2\pi\beta}}{x_i})^*, \mu \in \mathbb{R}^+ \tag{4.4}$$

   Which, is equivalent to passing the signal $\frac{e^{-j2\pi\beta}x_{i+1}}{x_i}$ through a single pole IIR low pass filter.

   Also, to avoid division-by zero errors, the alternative equation:

   $$w_{i+1} = w_i(1-\mu) + \mu \cdot sign(e^{-j2\pi\beta}x_{i+1}^* x_i), \mu \in \mathbb{R}^+ \tag{4.5}$$

   can be used.

   An example of output of this block when receiving a signal together with the spectrogram of the input can be seen below:

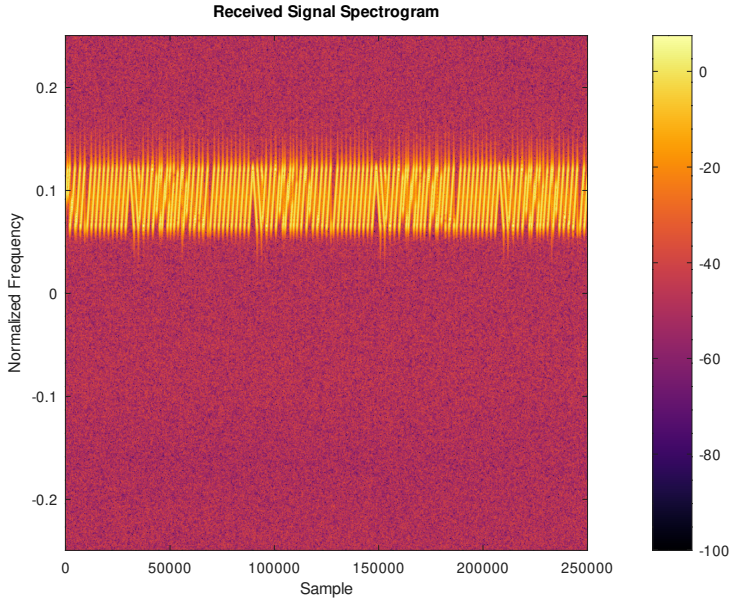

Figure 4.1: Spectrogram of he input signal composed of a sequence of LoRa frames with SF = 7, CRC on, and payloads of a single byte containing powers of 2 (1, 2, . . . , 128).

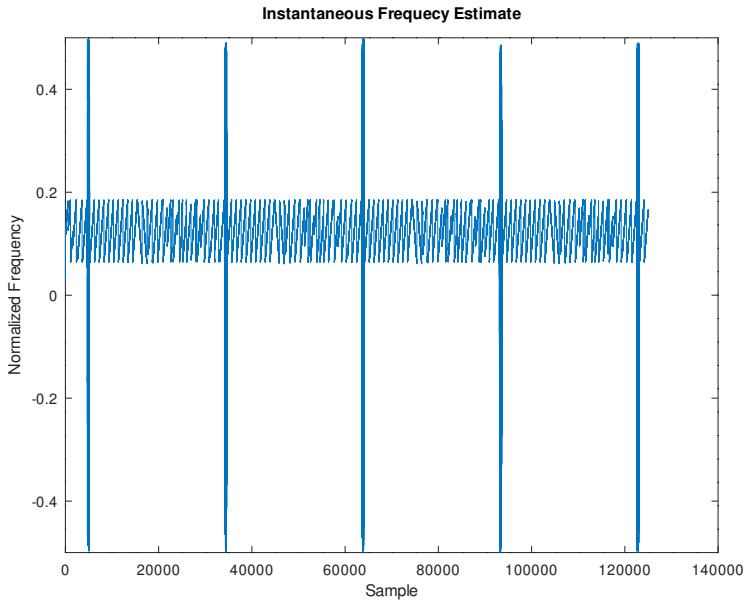

Figure 4.2: Output signal of the stochastic gradient descent frequency tracker given the described test signal at its input.

2. **DFT Peak**

   The previous method presents some problems as if multiple local maxima are present in the short-time spectrum, e.g. if there's an interfering signal, the frequency estimate will converge to an intermediate solution that minimizes the square error of the simplified cost function and not the global maximum that optimizes the original function.

   If the magnitude squared of discrete-time Fourier transform (DTFT) of the signal windowed by a (2N + 1)-sample rectangular window around each sample is seen as a the probability density of the instantaneous frequency process, the original problem can be interpreting as finding its mode, whereas the simplified process converges to its mean, therefore only working if the interfering signal's mean is equal to its mode, as it happens for a signal with Gaussian interference.

   To cope with this problem, we can use the use the following estimator for the frequency:

$$\hat{f}_k = N^{-1} argmax_n(|DFT[(x_{i+k} cos(\pi\beta(i - \frac{N-1}{2})^2))_{i\in\{0,\dots,N-1\}}](n)|^2) \quad (4.6)$$

   Which is equivalent to computing the original cost function at N points and taking the point that maximizes it.

Additionally, to compute the cost function at more frequency points, the order of the discrete Fourier transform (DFT) can be increased while applying a small rectangular window to the signal, i.e.

$$\hat{f}_k = N'^{-1} argmax_n(|DFT[(x_{i+k} w_i cos(\pi\beta(i - \frac{N'-1}{2})^2))_{i\in\{0,\dots,N'-1\}}](n)|^2) \tag{4.7}$$

Where:

$$w_i = \begin{cases} 1 & \text{if } |i| \leq N' \\ 0 & \text{if } |i| > N_w \end{cases} \tag{4.8}$$

Where $N_w \leq N'$.

This method presents some issues in terms of computational cost, however it has much better performance in terms of resistance to interference from signals with different symbol factors, as multiplying the signal by $cos(\pi\beta i^2)$, spreads the spectrum of the interfering signals and collapses the desired signal to a single (or two, in case of the CSS modulation at the transition of two symbols) peak.

To address the performance issues, the $\hat{f}_k$ estimator can be computed at a lower rate than the input signal, i.e.

$$\hat{f}_k = N'^{-1} argmax_n(|DFT[(x_{i+\delta k} cos(\pi\beta(i - \frac{N'-1}{2})^2))_{i\in\{0,\dots,N'-1\}}](n)|^2), \delta \in \mathbb{N} \tag{4.9}$$

As with the previous method, an example of output of this block when receiving the beginning of the same test signal is given below:

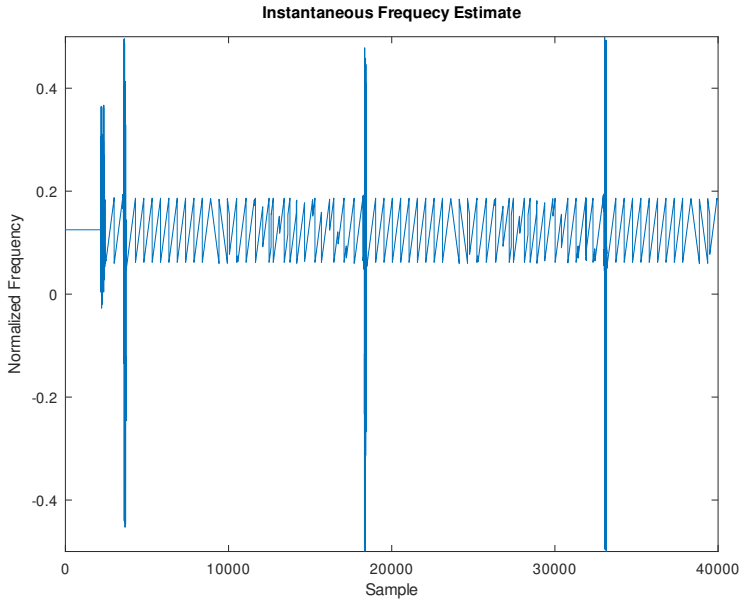

Figure 4.3: Output signal of the DFT peak frequency tracker given the described test signal at its input.

4.2 Correlation Synchronizer

This stage is split into two sections: the calculation of the correlation and the actual synchronization/alignment.

1. **Correlator**

   The correlation is calculated by taking the most recent $n_{preamble}$ samples of the instantaneous frequency estimated generated by the frequency estimator, where $n_{preamble}$ is the size of the fixed part of the preamble, i.e. the sync word plus the downchirps and computing its normalized correlation with the expected preamble, i.e.:

$$c_i = \frac{\vec{x_i} \cdot \vec{p_i}}{||\vec{x_i}||||\vec{p_i}||} \tag{4.10}$$

   Then this value and the first sample of the input vector are passed on to the next stage.

   Below an example of the outputs of this block when receiving the end of the preamble of a LoRa PHY Frame is given below:

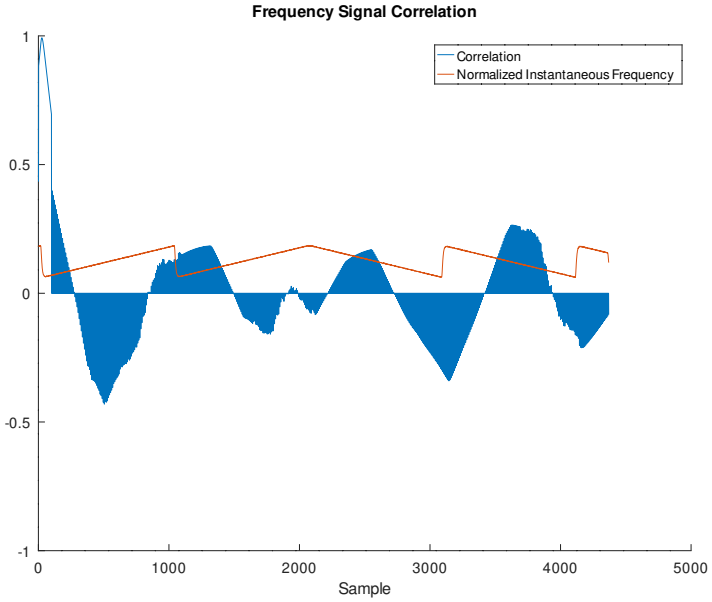

Figure 4.4: Output signals of the correlator when receiving the end of the preamble a LoRa frame with SF = 7, CRC on, and payload of a single byte containing a power of 2 (1, 2, . . . , 128).

2. **Synchronizer**

   This block takes the two values generated by the correlator and looks for a local maximum, given two thresholds. When the correlation is higher then the first threshold, it starts trying to find the local maximum and when the correlation gets lower than the second threshold, it stops this detection and outputs the signal starting from the point of maximum correlation between these two instants, already grouped in vectors of size n_{sym}, i.e. the number of samples in a symbol.

   Below the resulting output when the previously-shown output of the correlator is input to the synchronizer can be seen:

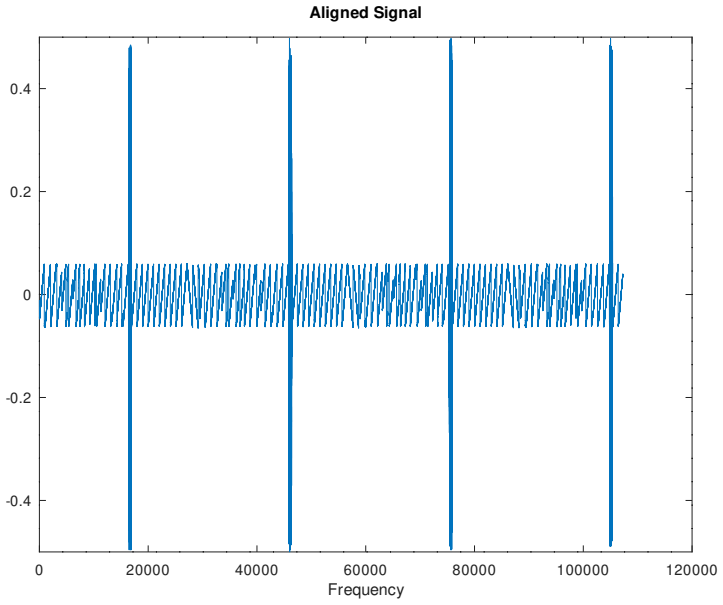

Figure 4.5: Synchronized signal, as generated by the synchronizer when its input is the previously shown test signal.

3. **Time-Frequency Shift Compensation**

   In this step, two parameters are estimated and compensated for by adding an offset to the instantaneous frequency signal: the frequency offset, i.e the mean frequency of the signal and the fractional time offset, i.e the remaining, non integer time offset of the signal after synchronization.

   Initially, the two sync-word symbols and the two downchirps are demodulated using a method similar to that described in the section on the multiple detection approach (section 4.3, item 2) for symbol detection, with the difference that the sync word samples get subtracted by the expected sync word and the

downchirps get demodulated considering downchirp symbols. Also, two obtained new sync word symbols are demodulated together generating a single offset value and the same is done for the two downchirps.

With these two values in hand, their average scaled by $\frac{1}{n_{sym}}$ gives an estimate of the frequency offset and their difference scaled by $\frac{1}{2\beta n_{sym}}$ gives an estimate of the fractional time shift. i.e.

$$\delta_f = \frac{sym_{upchirps} + sym_{downchirps}}{2n_{sym}} \tag{4.11}$$

$$\delta_t = \frac{sym_{upchirps} - sym_{downchirps}}{2\beta n_{sym}} \tag{4.12}$$

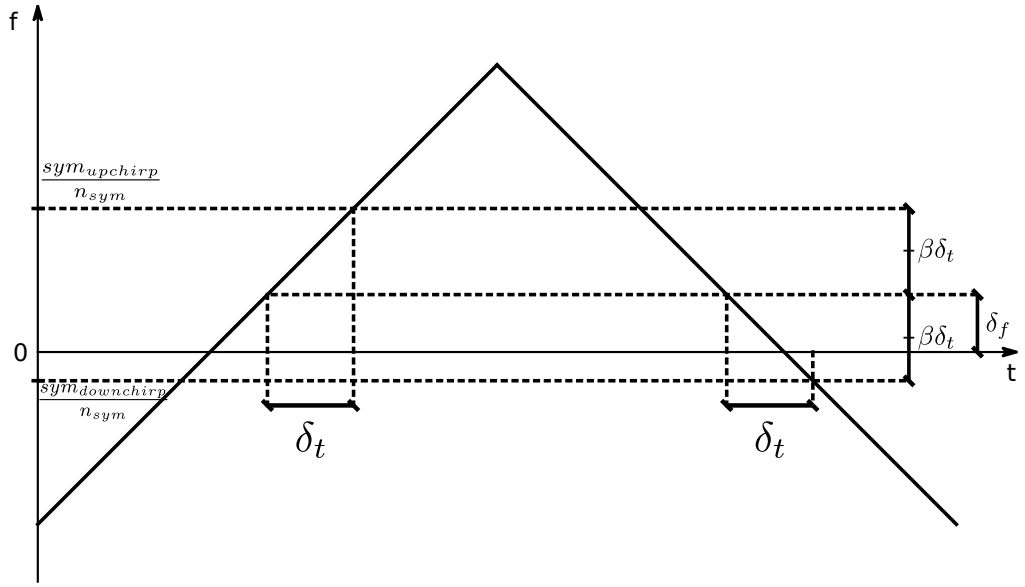

Figure 4.6: Illustration of the offset estimation procedure with a single upchirp (as sync word) and downchirp.

Finally, the calculated frequency offset is subtracted from the frequency estimates signal, and, to avoid complex fractional resampling operations, the time offset is compensated by adding an extra equivalent offset, i.e.

$$\delta_{f,t,eq} = \beta\delta_t \tag{4.13}$$

$$\delta_{f,total} = \delta_f + \delta_{f,t,eq} \tag{4.14}$$

$$f_{i,compensated} = f_i - \delta_{f,total} \tag{4.15}$$

Where β is the chirp rate relative to the current modulation parameters.

This relies the local linearity of the frequency-time waveform which allows the time offset to be compensated by shifting in frequency.

4.3 Symbol Decision

In a similar manner to the frequency estimator, two methods of symbol demodulation were proposed, one based on the mean instantaneous frequency of the symbol being detected and one based on the mode.

1. **Minimum Squares Approach**

   This method is based on the one described in [3], but uses the previously mentioned frequency estimation techniques. In this method, the symbol is determined by computing the inner product of the symbol frequency samples with each of the expected symbols and taking the one that maximizes this result. Because the symbols are rotations of each other, this can be done by computing the circular correlation of the symbol with the 0-th symbol, i.e the time-frequency representation of the base up-chirp, which reduces its computational complexity of this calculation from $O({n_{sym}}^2)$ to $O(n_{sym}\ \log(n_{sym}))$, where n_{sym} is the number of frequency samples per symbol, if a DFT based approach is employed.

   The main advantage of this approach is that it is insensitive to both frequency offsets and frequency scaling, as the instantaneous frequency vector of the base up-chirp has zero mean.

   Below, an example of the vector used as input to the symbol decision block together with the samples of the decided symbol can be seen.

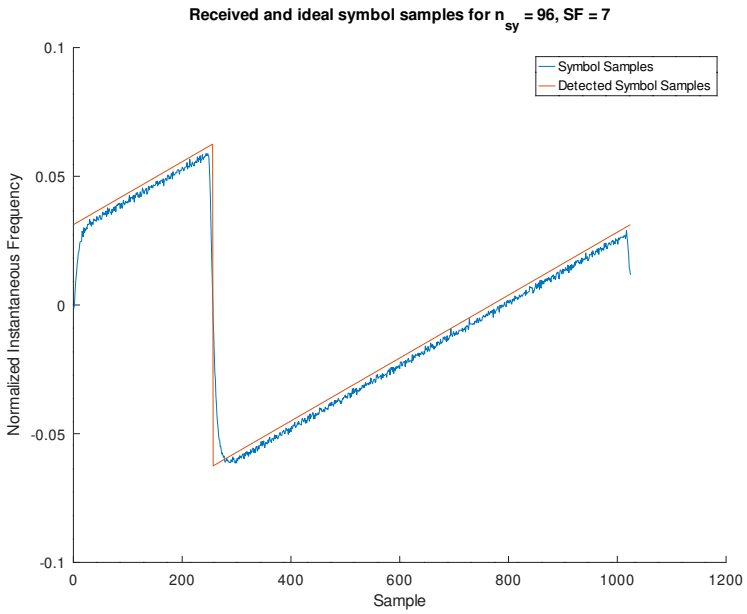

Figure 4.7: Synchronized frequency samples of a received LoRa symbol together with the samples of the symbol detected by the minimum squares decider block when given the former as an input.

2. **Multiple Detection Approach**

   The previous method assumes the interfering signal has zero mean frequency within the observation window, which is not necessarily true when using the DFT-based approach for frequency estimation, as the behavior of the estimate in the transition is not very predictable, as in this region there are two spectral peaks, with magnitudes close two each other, therefore, in this region, the maximum will alternate between them. To cope with this problem, a new method is proposed, that makes a decision for each of the points of the symbol and takes the most frequent one, using the following estimator:

$$s_i = mod\left(\left[\left(\hat{f}_i - \beta\left(i - \frac{n_{sym}-1}{2}\right)\right)\frac{2^{SF}}{BW}\right], 2^{SF}\right) \tag{4.16}$$

   Where the brackets represent nearest integer rounding and $\hat{f}_i$ is the vector of frequencies estimated for the current symbol.

   The main issue with this method is that it requires time-frequency alignment, like the procedure described in 3, to be done beforehand.

   Also, to improve performance, only the middle half samples of the frequency waveform are used, i.e. samples at instants $i \in \{\frac{n_{sym}}{4}, ..., \frac{3n_{sym}}{4} - 1\}$. This also helps avoid interference from neighboring symbols at the edges of the symbol due to time shifts.

4.4 Frame Decoder / Receiver Controller

This section implements the inverse of the steps described in section 3.2.2, reads the frame header and performs the CRC check if needed. It executes the following operations, in order:

- Gray Encoding
- Deinterleaving
- Decoding
- Randomization
- "CRC" Calculation

Chapter 5

Transceiver Implementation

5.1 GNU Radio

For the implementation of the receiver, GNU Radio was used. It is an free and open-source software development toolkit used for the development of signal processing blocks that can be connected in a flowgraph. This custom blocks can be written in C++ or Python and the flowgraphs can be either created using the GNU Radio Companion graphical interface and then exported to the above-mentioned languages or directly written in those. All the GNU Radio based Code developed for this project can be found at [15] and the most important sections, mainly the *work()* functions of each block, can be found at their respective appendix.

5.2 Implementation

5.2.1 Receiver

In order to keep the design modular, not only so it can be more easily modified and but also to take advantage of the multi-threading capabilities GNU Radio provides, as each block runs in a separate thread, each of the sections described in the previous section have been implemented in separate blocks.

In addition to these, two extra blocks were added:

- A **Receiver Controller** that, in order to control the flow of data and pass some necessary information between the blocks, controls some aspects of all other blocks using message ports, which are a asynchronous way of passing information between blocks that GNU Radio provides.
- A **Chirp Detector** that using an approach similar to that described in the DFT-based symbol decision, takes the DFT of the signal multiplied by the the linear chirp relative to the selected set of modulation parameters (BW_{CSS},

SF and consequently the chirp rate) computes the chirp-windowed DFT and computes the ratio between the energy in the maximum bin and the mean energy on the remaining bins and checks whether this value is higher to some set threshold. Then it uses this information to only allow the flow of data downstream in the flowgraph when this detection happens, in order to avoid unnecessary calculations and consequently unnecessary power consumption.

Finally, all these blocks were connected together in a hierarchical flowgraph to create a receiver block. An image of the flowgraph of the receiver is show below:

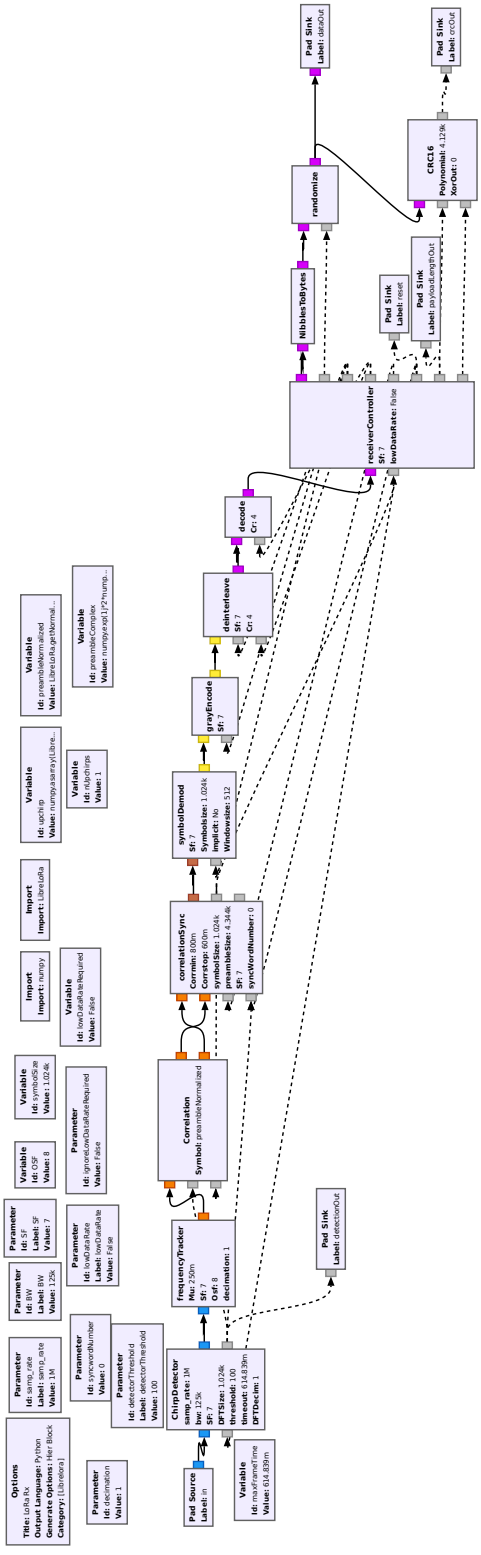

Figure 5.1: Flowgraph of the receiver in GNU Radio companion.

5.2.2 Transmitter

The transmitter follows the same design approach as the receiver, that is, of following the proposed structure and trying to be as modular as possible. Also, in addition to what was previously described two blocks had to be added:

- A **Transmitter Controller** that controls all other blocks, by setting the required parameters, according to parameters given to the transmitter and depending on which part of packet is being transmitted. This block, differently to the controller in the receiver, mainly uses tags to control the blocks to keep the implementation cleaner (tags are another mechanism of asynchronous message passing on GNU Radio that embeds itself into existing data streams instead of requiring an extra output in the block). Also, this block (together with the other blocks) supports dynamically setting all modulation parameters (SF, CR, BW, payload size), by sending a special tag to it.
- The **Append Silence** block. This is needed, because of how GNU Radio works and how most stock sink blocks are implemented, a continuous stream of data is required at the output of the receiver, so this block is controlled by the transmitter controller and generates silence samples (i.e. of value zero) whenever the rest of the blocks are not outputting any packets and outputs that data when it is available.

An image of the flowgraph of the receiver is show below:

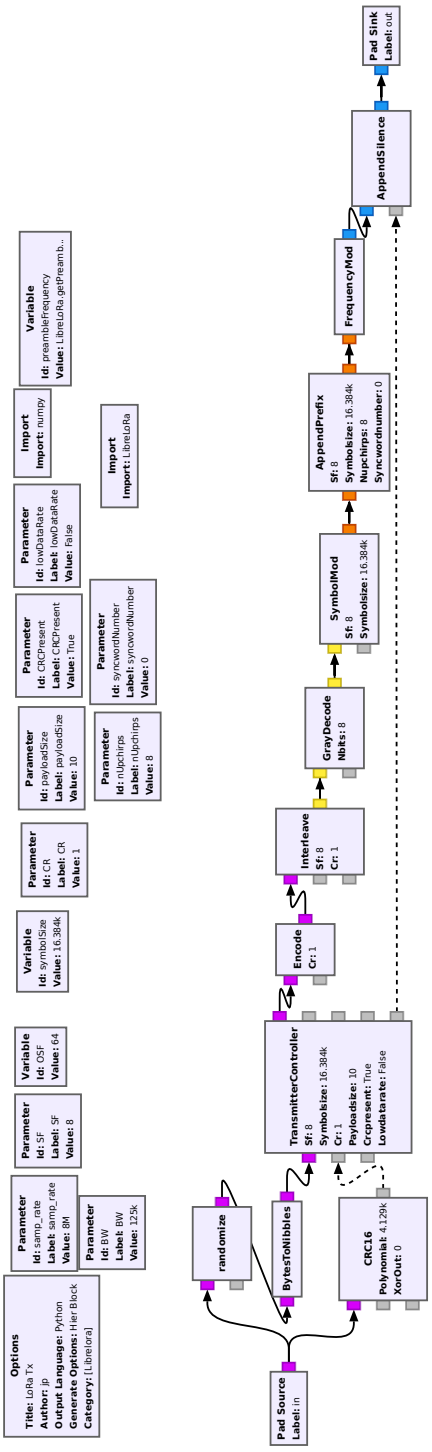

Figure 5.2: Flowgraph of the transmitter in GNU Radio companion.

In addition to the blocks themselves, another change was introduced. As it is not trivial in GNU Radio to instantiate multiple instances of the receiver and use them with a single sink, a mechanism to update the parameters of the transmitter via tags sent through its input was implemented, so a single block can be used and receive data with different modulation parameters to be controlled by external blocks.

Chapter 6

Hardware Implementation and Example Applications

6.1 Hardware Setup

The target hardware chosen was a Raspberry Pi 3A+ as the computer to run GNU Radio on, together with a RTL2832U based USB software defined radio (SDR) (commonly referred as an RTL-SDR) to work as a receiver and a HackRF One SDR transceiver to work as a transmitter. The main drivers behind this choice were the relative low cost and wide availability of these devices. Below a simple diagram depicting this setup is shown:

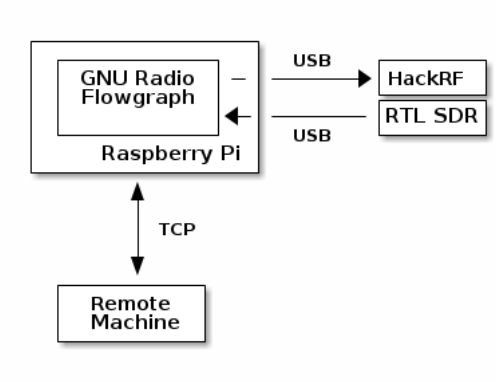

Figure 6.1: Block diagram of employed hardware setup.

6.2 Example Applications

To showcase how the developed library can be used, not only to implement the functionality of a simple transceiver, but also how it can be employed in more complex applications that take advantage of the flexibility built into the blocks.

6.2.1 LoRa Detector: Another Application of the Chirp Detector

With some simple modifications, the before-mentioned Chirp Detector (in section 5.2.1) can be also used for detecting the presence of different SF signals in multiple channels. This is especially useful, as it allows for obtaining useful statistics about the local channel/network without all the computational resources required by multiple receivers running in parallel. This was done by making its data output optional and adding a message output port that, whenever it detects a transmission, is used to send out a message containing the parameters of the block (SF, BW and sample rate) and the normalized center frequency of the band where the detection happened. In this way, by running multiple of these detectors in parallel, while keeping the sample rate high enough so multiple channels can be observed simultaneously, and sending the messages they generate to a block that receives and interprets these messages, one can easily do statistics on channel usage for each SF and band. In order to demonstrate this, a simple block that takes these messages and counts the detected transmissions on each channel-SF pair was developed. An image of the entire flowgraph developed for this application is shown below:

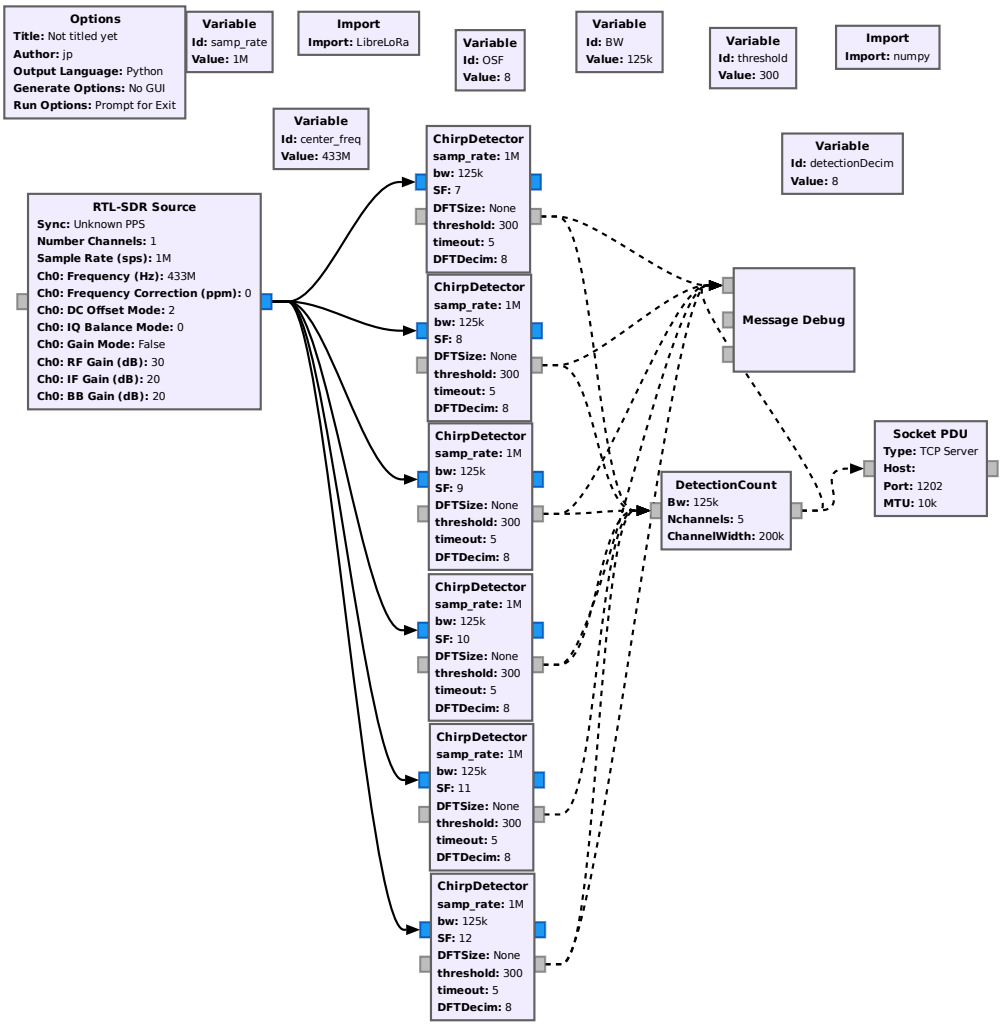

Figure 6.2: Flowgraph of the LoRa multi-channel multi-SF detector.

Also, it is worth pointing out that this extra information that is output by the chirp detectors could be used in future implementations for controlling the center frequency of a filter to which the input signal is fed to, making it possible to select the channel in which data is being transmitting with a certain SF and BW and forward it to an appropriate receiver, thus allowing for the implementation an efficient multi-channel receiver system.

6.2.2 Multi-Parameter, Multi-Channel Receiver

In this example application, five receivers are run in parallel, each with a different spreading factor and with a sample rate of 1 MS/s. In this way, any signal transmitted in any channel within an 1MHz band with any spreading factor can be received.

In addition, in order to allow simple integration with other applications, a TCP interface is also added. In this particular setup, one can run the receiver at a dedicated Raspberry Pi 3A+ and receive the data via TCP in any network connected device.

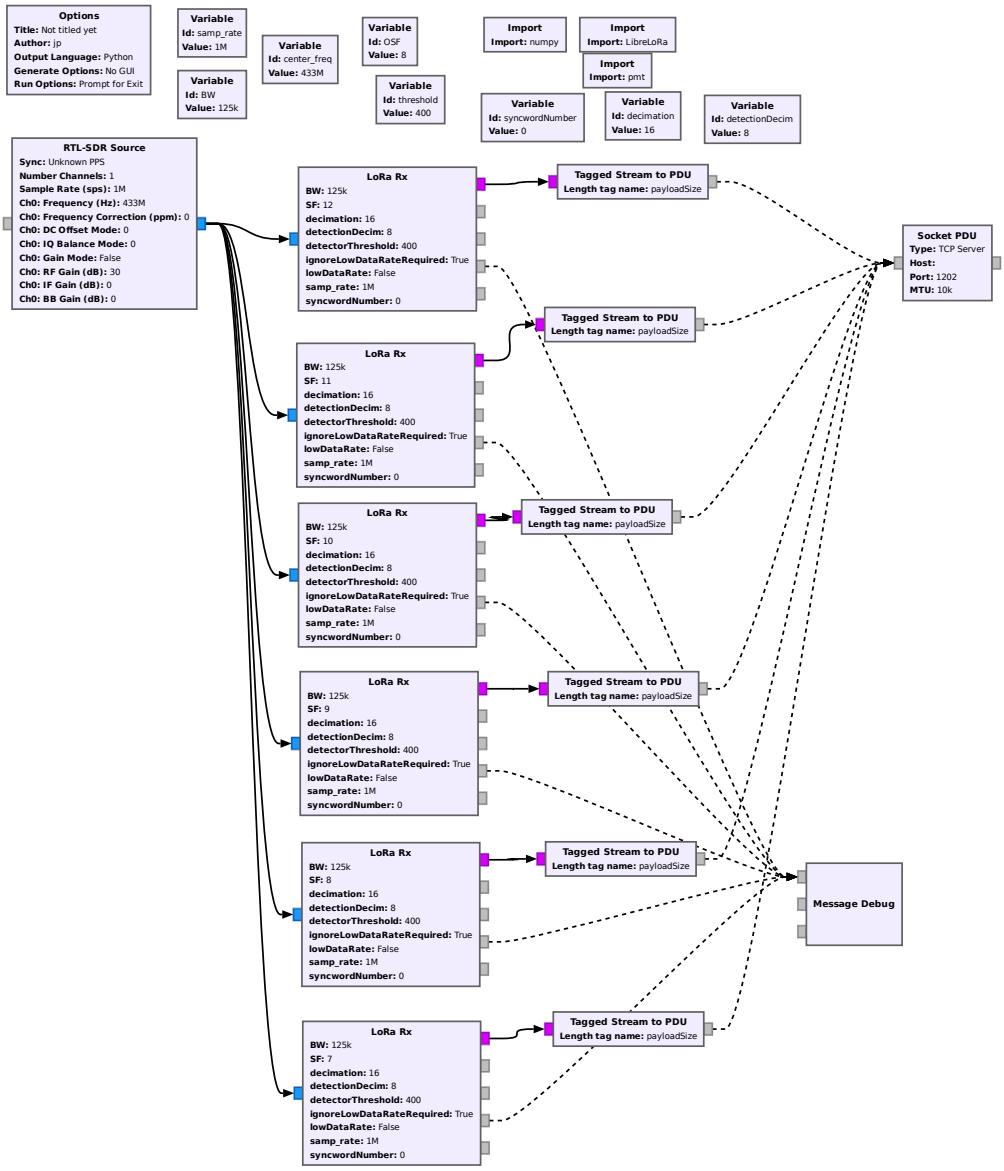

Figure 6.3: Flowgraph of the Multi-Parameter, Multi-Channel Receiver.

6.2.3 Variable Parameter Transmitter

In order to make the transmitter block able to have its parameters changed during runtime, an interface based on GNU Radio's tag propagation mechanism was implemented. This allows for metadata containing modulation and band parameters to be optionally propagated together with the data stream in order to change the parameters when needed.

In addition, an interface to translate a packet with a special format [1] to GNU Radio tags, extract the band information, if present, and generate a message to control a GNU Radio Signal source, in order to select the band, was also added, to allow the transmitter to be controlled by external devices via network.

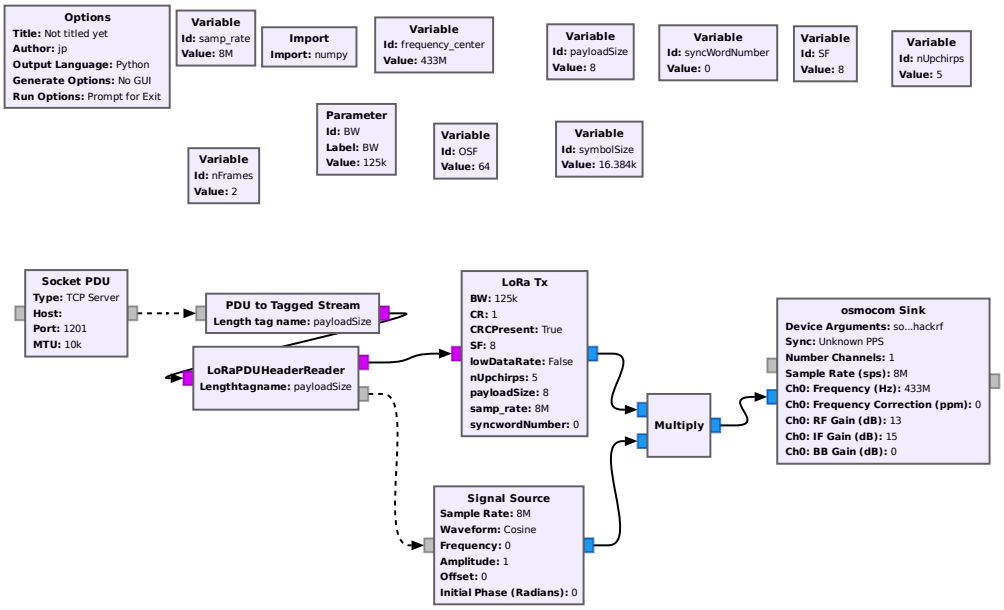

Figure 6.4: Flowgraph of the Variable Parameter Transmitter.

[1] It consists of the struct: struct loraPDUHeader {int8_t hasHeader; int8_t SF; uint8_t CR; bool payloadCRCPresent; bool lowDataRate; float BW; uint8_t syncWordNum; float fOffset; }; followed by the actual payload data to be transmitted via the LoRa transmitter. The hasHeader field is always 0x01 if a header with configuration is present, therefore if one is not present, the user should sent a single byte before the payload data with any value other than 0x01.

6.2.4 Multi-Parameter, Multi-Channel Transceiver

By joining the two previously mentioned flowgraphs, a full transceiver that can both receive and transmit with multiple modulation parameters and channels, controlled by a TCP interface was implemented. The idea behind this is that it could potentially be used for implementing fully functional LoRaWAN nodes and even gateways, with the physical and logical link control layers, i.e. LoRa PHY, running in a remote device, and the MAC layer protocols running in the local sender device. However, it is worth noting that, in its current state, this application cannot be run stably in the Raspberry Pi 3A+, due to RAM speed and size limitations, but it is very likely that with some extra optimization effort it could be made to run successfully in this device.

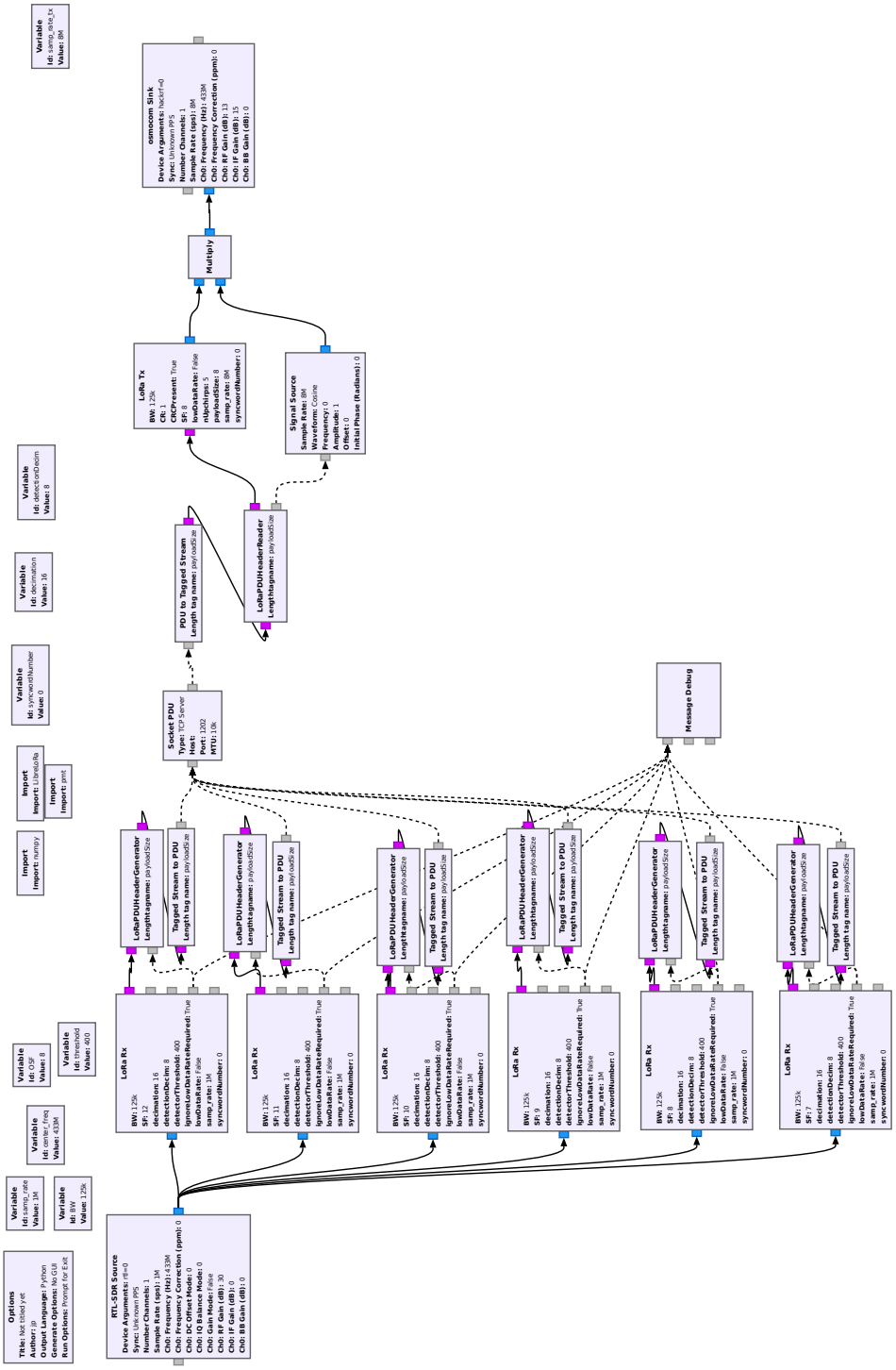

Figure 6.5: Flowgraph an example of the full Multi-Parameter, Multi-Channel Transceiver

Chapter 7

Conclusion

This thesis aimed to continue on previous research on reverse-engineering the lower layers of LoRa, commonly referred as LoRa PHY, propose an structure for its transceiver and implement this structure, using the GNU Radio platform, to be used with widely-available software-defined radios.
To what concerns the reverse-engineering of LoRa PHY, all the missing details were revealed, together with the methodology used to find them, building a full picture to how this part of this protocol stack works.
For the transceiver structure, new demodulation and synchronization methods were proposed, which potentially bring better interference resistance performance in relation to previously proposed methods.
To what concerns the actual implementation of the transceiver, not only the feasibility of the proposed methods were tested with real hardware, but also a completely free and open-source LoRa PHY transceiver implementation was made available to serve as base to further research and development on this protocol stack. Also, an example of hardware implementation was proposed and tested, showing how this implementation of the transceiver can be used, extended and integrated using the capabilities of the GNU Radio library.
As a final remark, some technical details and challenges still remain to be solved by future research, mainly to optimize the transceiver's algorithms and code for it to be able to run even simpler hardware, increasing its range of applications and to implement features left out of this project, such as support for LoRa's implicit mode.

Bibliography

[1] "Lorawan™ 1.1 specification," Lora Alliance, Standard, Oct. 2017. [Online]. Available: https://lora-alliance.org/resource_hub/lorawan-specification-v1-1/

[2] M. Knight and B. Seeber, "Decoding lora: Realizing a modern lpwan with sdr," *Proceedings of the GNU Radio Conference*, vol. 1, no. 1, 2016. [Online]. Available: https://pubs.gnuradio.org/index.php/grcon/article/view/8

[3] P. Robyns, P. Quax, W. Lamotte, and W. Thenaers, "A multi-channel software decoder for the lora modulation scheme," in *Proceedings of the 3rd International Conference on Internet of Things, Big Data and Security*, - 2018, p. nil.

[4] "Low-power wide area network (LPWAN) overview," Tech. Rep., May 2018.

[5] "Lorawan™ 1.1 regional parameters," Lora Alliance, Standard, Oct. 2017. [Online]. Available: https://lora-alliance.org/resource_hub/lorawan-regional-parameters-v1-1ra/

[6] J. Schlienz and D. Raddino, "Narrowband internet of things whitepaper," *White Paper, Rohde&Schwarz*, pp. 1–42, 2016. [Online]. Available: https://www.rohde-schwarz.com/hk/applications/narrowband-internet-of-things-white-paper_230854-314242.html

[7] S. Martiradonna, G. Piro, and G. Boggia, "On the evaluation of the nb-iot random access procedure in monitoring infrastructures," *Sensors*, vol. 19, no. 14, p. 3237, 2019.

[8] SIGFOX, "Sigfox connected objects: Radio specifications," Feb. 2020. [Online]. Available: https://build.sigfox.com/sigfox-device-radio-specifications

[9] ——, "Sigfox device cookbook - communication configuration," Nov. 2018. [Online]. Available: https://build.sigfox.com/sigfox-device-cookbook

[10] J. C. Liando, A. Gamage, A. W. Tengourtius, and M. Li, "Known and unknown facts of lora: Experiences from a large-scale measurement study," *ACM Trans. Sen. Netw.*, vol. 15, no. 2, Feb. 2019.

[11] W. L. Pieter Robyns, Peter Quax and W. Thenaers, "gr-lora: An efficient lora decoder for gnu radio," Sep. 2017.

[12] Semtech, "An1200.22 lora modulation basics," 2015.

[13] J. P. de Omena Simas, "lorasim-matlab: A matlab/octave-based lora

phy simulator," 2020. [Online]. Available: https://gitlab.com/jpsimas/lorasim-matlab.git

[14] G. Cook, "Crc reveng: arbitrary-precision crc calculator and algorithm finder," 2019. [Online]. Available: https://reveng.sourceforge.io/

[15] J. P. de Omena Simas, "Librelora: A gnuradio based lora phy receiver and transmitter implementation," 2020. [Online]. Available: https://gitlab.com/jpsimas/librelora.git

Appendix A

Code Listings

A.1 Chirp Detector

This block is implemented as a flowgraph written using Python due to limitations in GNU Radio's grc file format. Below is the main part of its code plus those from all the custom blocks used in it.

```
class ChirpDetector(gr.hier_block2):
    def __init__(self, samp_rate, BW=125e3, SF=7, DFTSize=None, threshold=200,
        timeout=5, DFTDecim=1):
        gr.hier_block2.__init__(
            self, "ChirpDetector",
                gr.io_signature(1, 1, gr.sizeof_gr_complex*1),
                gr.io_signature(1, 1, gr.sizeof_gr_complex*1),
        )
        if (DFTSize == None):
            self.DFTSize = DFTSize = int((1 << SF)*(samp_rate/BW))

        self.message_port_register_hier_in("reset")
        self.message_port_register_hier_out("detectOut")

        ##################################################
        # Parameters
        ##################################################
        self.samp_rate = samp_rate
        self.BW = BW
        self.DFTSize = DFTSize
        self.SF = SF
        self.threshold = threshold
        self.timeout = timeout

        ##################################################
        # Variables
        ##################################################
        self.windowSize = windowSize = DFTSize
        self.chirpRate = chirpRate = ((BW/samp_rate)**2)*(2**(-SF ))
        self.chirpWindow = chirpWindow = numpy.real(LibreLoRa.getChirpWindow(
            DFTSize, windowSize,0, numpy.sqrt(1/chirpRate)))*numpy.sqrt(DFTSize/
            windowSize)

        ##################################################
        # Blocks
```

```
        ##################################################
        self.fft_vxx_0 = fft.fft_vcc(DFTSize, True, numpy.ones(DFTSize), False, 1)
        # self.blocks_stream_to_vector_0 = blocks.stream_to_vector(gr.
            sizeof_gr_complex*1, DFTSize)
        self.blocks_stream_to_vector_0 = LibreLoRa.streamToHistoryVector_cc(
            DFTSize, DFTSize*DFTDecim);
        self.blocks_multiply_const_vxx_0 = blocks.multiply_const_vcc(chirpWindow)
        self.blocks_complex_to_mag_squared_0 = blocks.complex_to_mag_squared(
            DFTSize)
        self.blocks_complex_to_mag_squared_0.set_min_output_buffer(int(timeout*
            samp_rate/DFTSize))
        self.LibreLoRa_ToneDetector_0 = LibreLoRa.ToneDetector(DFTSize)
        self.LibreLoRa_PowerDetector_0 = LibreLoRa.PowerDetector(samp_rate,
            threshold, timeout, DFTSize*DFTDecim, pmt.to_pmt((SF, BW, samp_rate)))
        self.LibreLoRa_PowerDetector_0.set_min_output_buffer(int(numpy.ceil(
            timeout*samp_rate)))

        ##################################################
        # Connections
        ##################################################
        self.msg_connect((self.LibreLoRa_PowerDetector_0, 'detectOut'), (self, '
            detectOut'))
        self.msg_connect((self, 'reset'), (self.LibreLoRa_PowerDetector_0, 'reset'
            ))
        self.connect((self.LibreLoRa_PowerDetector_0, 0), (self, 0))
        self.connect((self.LibreLoRa_ToneDetector_0, 0), (self.
            LibreLoRa_PowerDetector_0, 1))
        self.connect((self.blocks_complex_to_mag_squared_0, 0), (self.
            LibreLoRa_ToneDetector_0, 0))
        self.connect((self.blocks_multiply_const_vxx_0, 0), (self.fft_vxx_0, 0))
        self.connect((self.blocks_stream_to_vector_0, 0), (self.
            blocks_multiply_const_vxx_0, 0))
        self.connect((self.fft_vxx_0, 0), (self.blocks_complex_to_mag_squared_0,
            0))
        self.connect((self, 0), (self.LibreLoRa_PowerDetector_0, 0))
        self.connect((self, 0), (self.blocks_stream_to_vector_0, 0))
```

A.1.1 Tone Detector

```
int
ToneDetector_impl::work (int noutput_items,
                  gr_vector_const_void_star &input_items,
                  gr_vector_void_star &output_items)
{
  const float *dftIn = (const float *) input_items[0];
  gr_complex *out = (gr_complex *) output_items[0];

  for(auto i = 0; i < noutput_items; i++) {
    const float* vect = dftIn + i*DFTSize;
    float totalPower;
    volk_32f_accumulator_s32f(&totalPower, vect, DFTSize);
    uint32_t maxInd;
    volk_32f_index_max_32u(&maxInd, vect, DFTSize);

    add_item_tag(0, nitems_written(0) + i, tagKey, pmt::from_float(fmod(float(
        maxInd)/DFTSize + 0.5f, 1.0f) - 0.5f));

    // out[i] = vect[maxInd]/totalPower;
    out[i] = (DFTSize - 1)*vect[maxInd]/(totalPower - vect[maxInd]);
  }
```

```
  return noutput_items;
}
```

A.1.2 Power Detector

```
    int
    PowerDetector_impl::general_work (int noutput_items,
                      gr_vector_int &ninput_items,
                      gr_vector_const_void_star &input_items,
                      gr_vector_void_star &output_items)
    {
      const gr_complex *in = (const gr_complex *) input_items[0];
      const float *powerIn = (const float *) input_items[1];
      gr_complex *out;
      if(output_items.size() > 0)
        out = (gr_complex *) output_items[0];

      size_t i;
      const size_t nOutput = ((noutput_items + decimation - 1)/decimation)*
          decimation;
      switch(state) {
      case detection:
        for(i = 0; i < (noutput_items + decimation - 1)/decimation; i++) {
          if(powerIn[i] > threshold) {
            state = started;
            samplesToRead = maxSamplesToRead;
            // time = clock();

            std::vector<gr::tag_t> tags;
            auto nr =  nitems_read(1);
            get_tags_in_range(tags, 1, nr + i, nr + i + 1);
            if(tags.size() != 0)
              message_port_pub(detectOutPort, pmt::make_tuple(message, tags[0].
                  value));
            else
              message_port_pub(detectOutPort, message);
#ifndef NDEBUG
            std::cout << "PowerDetector: started" << std::endl;
#endif
            break;
          }
        }

        consume(0, i*decimation);
        consume(1, i);

        return 0;

      case started:
        if(output_items.size() > 0) {
          //propagate tags
          auto nr =  this->nitems_read(0);
          std::vector<gr::tag_t> tags;
          this->get_tags_in_range(tags, 0, nr + i, nr + nOutput);
          for(auto tag : tags) {
            this->add_item_tag(0, this->nitems_written(0) + tag.offset - nr, tag.
                key, tag.value);
          }
        }

        for(i = 0; i < nOutput; i++) {
```

```
          samplesToRead--;
          if(output_items.size() > 0) {
            out[i] = in[i];
          }

          //if(clock() > time + timeout) {
          if(samplesToRead == 0) {
            state = waiting;
            break;
          }
        }

        // consume(0, ((noutput_items + decimation - 1)/decimation)*decimation);
        // consume(1, (noutput_items + decimation - 1)/decimation);
        consume(0, i);
        consume(1, i/decimation);
        if(output_items.size() > 0)
          // return ((noutput_items + decimation - 1)/decimation)*decimation;
          return i;
        else
          return 0;
      case waiting:
        for(i = 0; i < ((noutput_items + decimation - 1)/decimation)*decimation; i
            ++) {
          if(powerIn[i] < threshold) {
            state = detection;
#ifndef NDEBUG
            std::cout << "PowerDetector:␣stopped␣(timeout)" << std::endl;
#endif
            break;
          }
        }

        consume(0, i*decimation);
        consume(1, i);
        return 0;
      }
      return 0;
    }
```

A.2 Frequency Estimator

A.2.1 Stochastic Gradient Descent (frequencyTracker class)

```
template<typename T>
int
frequencyTracker_impl<T>::work (int noutput_items,
                                gr_vector_const_void_star &input_items,
                                gr_vector_void_star &output_items)
{
  const gr_complex *in = (const gr_complex *) input_items[0];
  T *out = (T *) output_items[0];

  for(int i = 0; i < noutput_items; i++) {
    for(int j = 0; j < decimation; j++) {
      w *= wStep;

      w = (1 - mu)*w + mu*(in[i*decimation + j + 1]/(in[i*decimation + j] + 1e-6f)
          );
    }
```

```
    out[i] = calcFreq(w);
  }

  // Tell runtime system how many input items we consumed on
  // each input stream.
  // this->consume_each (noutput_items*decimation);

  // Tell runtime system how many output items we produced.
  return noutput_items;
}


template <>
float frequencyTracker_impl<float>::calcFreq(gr_complex w) {
  return std::arg(w)/(2*M_PI);
}
```

A.2.2 DFTPeak

The DFTPeak Block was developed using a GNU Radio flowgraph that mixes both stock and custom blocks.

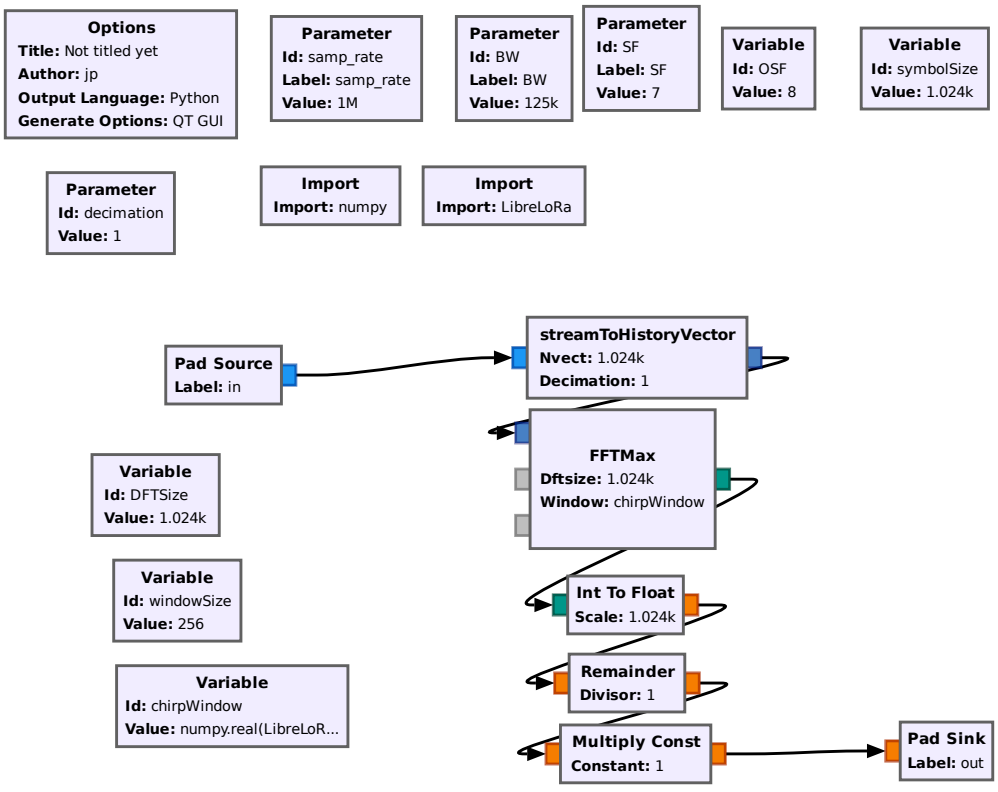

Figure A.1: Flowgraph of the DFTPeak block in GNU Radio companion.

The work functions of the used custom blocks can be found below:

1. streamToHistoryVector

```
template<typename T>
int
streamToHistoryVector_impl<T>::work(int noutput_items,
                                    gr_vector_const_void_star &input_items,
                                    gr_vector_void_star &output_items)
{
  const T *in = (const T *) input_items[0];
  T *out = (T *) output_items[0];

  // Do <+signal processing+>
  for(size_t i = 0; i < noutput_items; i++) {
    // for(size_t j = 0; j < nVect; j++)
    //   out[nVect*i + j] = in[i + j];
    memcpy(out + nVect*i, in + i*decimation, nVect*sizeof(gr_complex));
  }

  // Tell runtime system how many output items we produced.
  return noutput_items;
}
```

2. FFTMax

```
int
FFTMax_impl::work(int noutput_items,
    gr_vector_const_void_star &input_items,
    gr_vector_void_star &output_items)
{
  const gr_complex *in = (const gr_complex *) input_items[0];
  uint32_t *out = (uint32_t *) output_items[0];

  for(auto i = 0; i < noutput_items; i++) {
    if(enabled) {
      volk_32fc_x2_multiply_32fc(fftIn, in + i*size, fftWindow, windowSize);
      // memcpy(fftIn, in + i*size, size*sizeof(gr_complex));
      fft_execute(fftPlan);
      volk_32fc_index_max_32u(out + i, fftOut, size);
    } else
      out[i] = 0;
  }

  // Tell runtime system how many output items we produced.
  return noutput_items;
}
```

3. Remainder

```
int
Remainder_impl::work(int noutput_items,
    gr_vector_const_void_star &input_items,
    gr_vector_void_star &output_items)
{
  const float *in = (const float *) input_items[0];
  float *out = (float *) output_items[0];

  for(auto i = 0; i < noutput_items; i++)
    out[i] = std::remainder(in[i], divisor);

  // Tell runtime system how many output items we produced.
```

```
      return noutput_items;
    }
```

A.3 Correlator

```
    template<>
    int
    Correlation_impl<float>::work(int noutput_items,
                                  gr_vector_const_void_star &input_items,
                                  gr_vector_void_star &output_items)
    {
      const float *in = (const float *) input_items[0];
      float *corr_out = (float *) output_items[0];
      float *data_out = (float *) output_items[1];

      // Do <+signal processing+>

#ifndef NDEBUG
      ///std::cout << "Correlation: work called. noutput_items: " << noutput_items
          << std::endl;
#endif

      if(enabled) {
        float corr;
        float sumSq;
        float sum;
        const float* samples = in;

        volk_32f_x2_dot_prod_32f(&corr, samples, symbol.data(), symbol.size());
        volk_32f_x2_dot_prod_32f(&sumSq, samples, samples, symbol.size());
        volk_32f_accumulator_s32f(&sum, samples, symbol.size());

        corr_out[0] = corr/sqrt(sumSq - sum*sum/symbol.size());
        // data_out[0] = samples[0];

        for(size_t k = 1; k < noutput_items; k++) {
          sumSq -= samples[0]*samples[0];
          sum -= samples[0];

          samples++;

          sumSq += samples[symbol.size() - 1]*samples[symbol.size() - 1];
          sum += samples[symbol.size() - 1];

          volk_32f_x2_dot_prod_32f(&corr, samples, symbol.data(), symbol.size());

          corr_out[k] = corr/sqrt(sumSq - sum*sum/symbol.size());
          // data_out[k] = samples[0];
        }
      } else {
        for(auto k = 0; k < noutput_items; k++)
          corr_out[k] = 0;
      }

      for(auto k = 0; k < noutput_items; k++)
        data_out[k] = in[k];


      // Tell runtime system how many output items we produced.
      return noutput_items;
```

```
    }
```

A.4 Synchronizer

```
    template<typename T>
    int
    correlationSync_impl<T>::general_work (int noutput_items,
                                           gr_vector_int &ninput_items,
                                           gr_vector_const_void_star &input_items,
                                           gr_vector_void_star &output_items)
    {
      const T *data_in = (const T *) input_items[0];
      const T *corr = (const T *) input_items[1];
      T* data_out = (T*) output_items[0];
      // bool* syncd_out = (bool*) output_items[1];

#ifndef NDEBUG
      // std::cout << "correlationSync: work called: noutput_items = " <<
          noutput_items << std::endl;
#endif

      // *syncd_out = false;
      // Do <+signal processing+>

      if(!syncd) {
        bool foundFirstPt = false;
        float corrMaxNorm = 0;
        corrMax = 0;
        size_t maxPos = 0;

        for(size_t i = 0; i < 2*symbolSize; i++) {
          if(foundFirstPt) {
            if(norm(corr[i]) <= corrStop) {
              corrMax = conj<T>(corrMax)/std::abs(corrMax);
              foundFirstPt = false;
              this->consume_each(maxPos);
              syncd = true;
              preambleConsumed = false;
              preambleSamplesToConsume = preambleSize;
#ifndef NDEBUG
              std::cout << "correlationSync: sync'd" << std::endl;
#endif
              // *syncd_out = true;
              // produce(1, 1);
              // return WORK_CALLED_PRODUCE;

              this->message_port_pub(syncPort, pmt::PMT_NIL);
#ifndef NDEBUG
              std::cout << "correlationSync: produced syncd signal" << std::endl;
#endif
              return 0;
            } else if(norm(corr[i]) > corrMaxNorm) {
              if(i < symbolSize) {
              // if(i < ninput_items[0] - symbolSize) {
                corrMaxNorm = norm(corr[i]);
                maxPos = i;
              } else
                break;
            }
          } else if(norm(corr[i]) >= corrMin) {
            if(i < symbolSize) {
```

```
            // if(i < ninput_items[0] - symbolSize) {
              foundFirstPt = true;
              corrMax = corr[i];
              corrMaxNorm = norm(corr[i]);
            } else
              break;
          }
        }//for

        this->consume_each(symbolSize);
        // this->consume_each(ninput_items[0] - symbolSize);

        return 0;
      }
```

A.5 Symbol Detector

A.5.1 Minimum Squares

```
    template<>
    int
    symbolDemod_impl<float>::general_work (int noutput_items,
                                    gr_vector_int &ninput_items,
                                    gr_vector_const_void_star &input_items,
                                    gr_vector_void_star &output_items)
    {
      const float *dataIn = (const float *) input_items[0];
      uint16_t *dataOut = (uint16_t *) output_items[0];

#ifndef NDEBUG
      std::cout << "demodulating␣" << noutput_items <<  "␣symbols,␣SF␣=␣" << SF <<
           std::endl;
#endif
      if(started) {
        // Do <+signal processing+>
        for(size_t i = 0; i < noutput_items; i++) {
          //size_t i = 0;
          float corrMax = 0;
          size_t jMax = 0;

          for(size_t j = 0; j < /*symbolSize*/(1 << SF); j++) {
            float corrJ;
            volk_32f_x2_dot_prod_32f(&corrJ, dataIn + i*symbolSize, upchirps.data
                () + j*(symbolSize >> SF), symbolSize);

            //volk_32f_x2_dot_prod_32f(&corrJ, dataIn + i*symbolSize, getSymbol<
                float>(j, SF, symbolSize).data(), symbolSize);

            if(corrJ >= corrMax) {
              corrMax = corrJ;
              jMax = j;
            }
          }

          // float err = jMax - (symbolSize >> SF)*std::round(float(jMax)/(
              symbolSize >> SF));
          // offset += 0.1*err;

          // dataOut[i] = uint16_t(std::round((jMax /*- std::round(offset)*/)*(1
              << SF)/float(symbolSize)))%uint16_t(1 << SF);
```

```
          dataOut[i] = jMax;
#ifndef NDEBUG
          std::cout << "demodulated␣symbol:␣" << std::dec << dataOut[i] << ",␣SF␣=
              ␣" << SF << std::endl;
#endif
        }

        // Tell runtime system how many input items we consumed on
        // each input stream.
        consume_each (noutput_items);

        // Tell runtime system how many output items we produced.
        return noutput_items;
        //consume_each(1);
        //return 1;
      }
```

A.5.2 Multiple Detection

```
    int
    symbolDemodNew_impl::work(int noutput_items,
        gr_vector_const_void_star &input_items,
        gr_vector_void_star &output_items)
    {
      const float *dataIn = (const float *) input_items[0];
      uint16_t *dataOut = (uint16_t *) output_items[0];

      // Do <+signal processing+>
      for(size_t i = 0; i < noutput_items; i++) {

        // size_t minErr = (1 << SF);
        // size_t minSym = 0;
        // for(size_t sym = 0; sym < (1 << SF); sym++) {
          //gr_complex mean = 0;

          for(auto& x : count)
            x = 0;

          for(size_t j = startingIndex; j < startingIndex + windowSize; j++){
            // mean += std::polar<float>(1.0, 2*M_PI*OSF*(dataIn[i*symbolSize + j]
                 - twoUpchirps[j]));;
            // int16_t decision = 2*(1 << SF) - int16_t(std::round(symbolSize*(std
                ::abs(dataIn[i*symbolSize + j] - twoUpchirps[j + sym*OSF])))%
                int16_t(2*(1 << SF));

            float decisionf = std::round(float(1 << SF)*OSF*(dataIn[i*symbolSize +
                 j] - twoUpchirps[j]));

            int16_t decision = (int32_t(decisionf)%int32_t(1 << SF) + int32_t(1 <<
                 SF))%int32_t(1 << SF);

            count.at(decision)++;
          }


          // dataOut[i] = uint16_t(round((1 << SF)*std::arg(mean)/(2.0*M_PI)))%
              uint16_t(1 << SF);
          size_t maxK = 0;
          size_t maxCount = 0;
          for(auto k = 0; k < count.size(); k++)
            if(count[k] > maxCount){
              maxK = k;
```

```
              maxCount = count[k];
            }

          dataOut[i] = maxK;
// #ifndef NDEBUG
//         std::cout << "err: " << maxK << std::endl;
// #endif
        //   if(maxK < minErr){
        //     minSym = sym;
        //     minErr = maxK;
        //   }
        // }

        // dataOut[i] = minSym;
#ifndef NDEBUG
        std::cout << "symbolDemodNew: demodulated symbol: " << std::dec << dataOut
            [i] << ", SF = " << SF << std::endl;
        // std::cout << "symbolDemodNew: counts: ";
        // for(auto& x : count)
        //   std::cout << x << ", ";
#endif
      }


      // Tell runtime system how many output items we produced.
      return noutput_items;
    }
```

A.6 Gray Encoder

```
    int
    grayEncode_impl::work (int noutput_items,
                           gr_vector_const_void_star &input_items,
                           gr_vector_void_star &output_items)
    {
      const uint16_t *in = (const uint16_t *) input_items[0];
      uint16_t *out = (uint16_t *) output_items[0];

#ifndef NDEBUG
      std::cout << "grayEncode: work called: noutput_items = " << noutput_items <<
          std::endl;
#endif

      // Do <+signal processing+>
      for(size_t i = 0; i < noutput_items; i++)
        out[i] = (in[i] ^ (in[i] >> 1)) & ((1 << SF) - 1);

      // Tell runtime system how many input items we consumed on
      // each input stream.
        //consume_each (noutput_items);

      // Tell runtime system how many output items we produced.
        return noutput_items;
    }
```

A.7 Deinterleaver

```
    int
    deinterleave_impl::general_work (int noutput_items,
                                     gr_vector_int &ninput_items,
                                     gr_vector_const_void_star &input_items,
                                     gr_vector_void_star &output_items)
    {
      const uint16_t *in = (const uint16_t *) input_items[0];
      uint8_t *out = (uint8_t *) output_items[0];

#ifndef NDEBUG
      //std::cout << "deinterleave: work called: noutput_items = " <<
          noutput_items << std::endl;
#endif

      const size_t blocksToProduce = (noutput_items + SF - 1)/SF;

#ifndef NDEBUG
      if(blocksToProduce != 0) {
        std::cout << "deinterleave:␣producing:␣" << blocksToProduce << "␣blocks␣("
             << "nouput_items␣=␣" << noutput_items << ",␣SF␣=␣" << SF << ")" <<
            std::endl;

        // Do <+signal processing+>
      }
#endif

      for(size_t k = 0; k < SF; k++)
        out[k] = 0;

      for(size_t i = 0; i < blocksToProduce; i++){

#ifndef NDEBUG
        std::cout << "deinterleave:␣deinterleaving␣symbols:␣";
        for(size_t j = 0; j < codeLength; j++)
          std::cout << std::hex << in[i*codeLength + j] << "␣";
        std::cout << std::endl;
#endif

        for(size_t j = 0; j < codeLength; j++)
          for(size_t k = 0; k < SF; k++)
            out[i*SF + (j + k)%SF] |= (in[i*codeLength + j] >> k & 0x01) << j;

      }
      // Tell runtime system how many input items we consumed on
      // each input stream.
      // consume_each (noutput_items);
      consume_each(codeLength*blocksToProduce);

      // Tell runtime system how many output items we produced.
      // return noutput_items;
      return SF*blocksToProduce;
    }
```

A.8 Decoder

```
    int
    decode_impl::work (int noutput_items,
                       gr_vector_const_void_star &input_items,
                       gr_vector_void_star &output_items)
    {
      const uint8_t *in = (const uint8_t *) input_items[0];
      uint8_t *out = (uint8_t *) output_items[0];

#ifndef NDEBUG
      std::cout << "decode: work called: noutput_items = " << noutput_items << std
          ::endl;
#endif

      // Do <+signal processing+>
      for(size_t i = 0; i < noutput_items; i++) {
        uint8_t syndrome = calculatePairity(in[i], pairityMatrix) ^ in[i];
        out[i] = (in[i] ^ cosetLeader[syndrome]) & 0x0f;

#ifndef NDEBUG
        std::cout << std::hex << "decoded: in: " << unsigned(in[i]) << ", out: "<<
             std::hex << unsigned(out[i]) << ", CR = " << CR << std::endl;
#endif

      }

      // Tell runtime system how many input items we consumed on
      // each input stream.
      // consume_each (noutput_items);

      // Tell runtime system how many output items we produced.
      return noutput_items;
    }
```

A.9 Frame Decoder / Receiver Controller

```
    void
    receiverController_impl::forecast (int noutput_items, gr_vector_int &
        ninput_items_required)
    {
      // ninput_items_required[0] = started? (gotHeader? nibblesToRead : SFcurrent
          ) : 0;
      // ninput_items_required[1] = started? 0 : 1;
      switch(currentState) {
      case waitingForSync:
        //no samples are really needed, but this avoinds work being called when it
            is not necessary
        ninput_items_required[0] = 1;//SFcurrent;//0;
        // ninput_items_required[1] = 1;
        break;
      case readingHeader:
        ninput_items_required[0] = SFcurrent;
        // ninput_items_required[1] = 0;
        break;
      case sendingPayload:
        ninput_items_required[0] = payloadNibblesToRead + extraNibblesToConsume +
            (payloadCRCPresent? 2*payloadCRCSize : 0);
        // ninput_items_required[1] = 0;
        break;
```

```
      }
    }

    int
    receiverController_impl::general_work(int noutput_items,
                                          gr_vector_int &ninput_items,
                                          gr_vector_const_void_star &input_items,
                                          gr_vector_void_star &output_items)
    {
      const uint8_t *nibblesIn = (const uint8_t *) input_items[0];
      // const bool *syncdIn = (const bool *) input_items[1];
      uint8_t *nibblesOut = (uint8_t *) output_items[0];

#ifndef NDEBUG
      std::cout << "receiverController: work called: noutput_items = " <<
          noutput_items << std::endl;
#endif

      int produced = 0;
      size_t payloadNibblesToProduce = 0;
      // Do <+signal processing+>
      switch(currentState) {
      case waitingForSync:
        // if(*syncdIn)
        //   startRx();

        // consume(1, 1);
        consume_each(1);
        break;

      case readingHeader:
        //consume(1, 1);
        readHeader(nibblesIn, nibblesOut);
        //consume(0, SFcurrent);
        //produced = SFcurrent - 5;
        consume(0, 5);
        produced = 0;

        if(!headerCheckSumValid || CR == 0 || CR > 4) {
          stopRx();
        } else {
#ifndef NDEBUG
          std::cout << std::dec;
          std::cout << "Got Valid Header" << ": ";
          std::cout << "length: " << unsigned(payloadLength) << ", ";
          std::cout << "CR: " << unsigned(CR) << ", ";
          std::cout << "CRC " << (payloadCRCPresent? "Present" : "Not Present") <<
               std::endl;
#endif

          message_port_pub(lfsrStatePort, pmt::from_long(0xff));
          message_port_pub(payloadLengthPort, pmt::from_long(payloadLength));

          // send header info as a loraFrameParams tag
          pmt::pmt_t message = pmt::make_tuple(pmt::from_long(SF),
                                               pmt::from_long(CR),
                                               pmt::from_bool(payloadCRCPresent),
                                               pmt::from_bool(lowDataRate));

          add_item_tag(0, nitems_written(0), pmt::intern("loraFrameParams"),
              message);
          // send payloadLength as a separate tag
```

```
          add_item_tag(0, nitems_written(0), pmt::intern("payloadSize"), pmt::
              from_long(payloadLength));

          // if(2*payloadLength <= (SFcurrent - 5))
          //   stopRx();
          // else {
          payloadNibblesToRead = 2*payloadLength;// - (SFcurrent - 5);
          size_t nibblesToRead = payloadNibblesToRead + 2*(payloadCRCPresent?
              payloadCRCSize : 0);
            // extraNibblesToConsume = SF*ceil(nibblesToRead/float(SF)) -
                nibblesToRead;
            //extraNibblesToConsume = (SF - nibblesToRead%SF)%SF;

          setSFcurrent(lowDataRate? (SF - 2) : SF);
            //setSFcurrent(SF);
          if(nibblesToRead > (SF - 7))
            extraNibblesToConsume = (SFcurrent - (nibblesToRead - (SF - 7))%
                SFcurrent)%SFcurrent;

#ifndef NDEBUG
          std::cout << "nibbles␣to␣read:␣" << nibblesToRead << ",␣SF␣=␣" << SF <<
              std::endl;
          std::cout << "extra␣nibbles:␣" << extraNibblesToConsume << ",␣SF␣=␣" <<
              SF << std::endl;
#endif

            currentState = sendingPayload;
            // synchronizer->setNOutputItemsToProduce((extraNibblesToConsume +
                nibblesToRead)*(CR + 4)/SF);
            if(nibblesToRead + extraNibblesToConsume > (SF - 7))
              message_port_pub(synchronizerSetNPort, pmt::from_long((
                  extraNibblesToConsume + nibblesToRead - (SF - 7))*(CR + 4)/
                  SFcurrent));
            // message_port_pub(synchronizerResetPort, pmt::PMT_NIL);
            // randomizer->reset();
          // }
        }
        break;
      case sendingPayload:

        //message_port_pub(lfsrStatePort, pmt::from_long(0xff));
        for(size_t i = 0; i < payloadNibblesToRead; i++) {
          nibblesOut[i] = nibblesIn[i];

#ifndef NDEBUG
          std::cout << "receiverController:␣produced␣data␣nibble:" << std::hex <<
              unsigned(nibblesOut[i]) << std::endl;
#endif

        }

        produced = payloadNibblesToRead;

        if(payloadCRCPresent) {
          uint16_t CRC = nibbles2bytes<uint16_t>(*((uint32_t*)(nibblesIn +
              payloadNibblesToRead)));

#ifndef NDEBUG
          std::cout << "receiverController:␣read␣CRC:␣" << unsigned(CRC) << std::
              endl;
#endif

          message_port_pub(crcPort, pmt::from_long(CRC));
```

```
        }

#ifndef NDEBUG
        for(size_t i = 0 ; i < extraNibblesToConsume; i++)
          std::cout << "receiverController:␣extra␣nibble:" << std::hex << unsigned
              (nibblesIn[payloadNibblesToRead + (payloadCRCPresent? 2*
              payloadCRCSize : 0) + i]) << std::endl;
#endif

        consume(0, payloadNibblesToRead + extraNibblesToConsume + (
            payloadCRCPresent? 2*payloadCRCSize : 0));
        stopRx();

        break;
      default:
        ;
      }

      // Tell runtime system how many output items we produced.
      return produced;
    }
```

A.10 Nibbles To Bytes

```
int
NibblesToBytes_impl::work(int noutput_items,
    gr_vector_const_void_star &input_items,
    gr_vector_void_star &output_items)
{
  const uint8_t *in = (const uint8_t *) input_items[0];
  uint8_t *out = (uint8_t *) output_items[0];

  // Do <+signal processing+>
  for(size_t i = 0; i < noutput_items; i++)
    out[i] = (in[2*i + 1] << 4)|in[2*i];

  // Tell runtime system how many output items we produced.
  return noutput_items;
}
```

A.11 Randomizer

```
    int
    randomize_impl::work(int noutput_items,
        gr_vector_const_void_star &input_items,
        gr_vector_void_star &output_items)
    {
      const uint8_t *in = (const uint8_t *) input_items[0];
      uint8_t *out = (uint8_t *) output_items[0];

      //check for payloadSize tags
      std::vector<gr::tag_t> tags;
      get_tags_in_range(tags, 0, nitems_read(0), nitems_read(0) + 1, pmt::intern("
          payloadSize"));
      if(tags.size() != 0) {
        size_t payloadSizeNew = pmt::to_long(tags[0].value);
        if(payloadSizeNew <= 255) {
          setPayloadSize(payloadSizeNew);
#ifndef NDEBUG
```

```
          std::cout << "randomize:␣set␣payloadSize␣to:␣" << payloadSize << std::
              endl;
#endif
        } else {
          setPayloadSize(1);
#ifndef NDEBUG
          std::cout << "randomize:␣got␣invalid␣payloadSize.␣set␣payloadSize␣to␣1."
               << payloadSize << std::endl;
#endif
        }
      }

      for(size_t i = 0; i < noutput_items; i++) {
        out[i] = in[i] ^ lfsrState;

#ifndef NDEBUG
        std::cout << "randomize:␣in␣=␣" << std::hex << unsigned(in[i]) << ",␣out␣=
            ␣" << unsigned(out[i]) << ",␣state␣=␣" << unsigned(lfsrState) << std::
            endl;
#endif

        lfsrState = (lfsrState << 1) | pairity(lfsrState&0xB8);

        byteCount++;
        //reset state at the end of each frame
        if(byteCount == payloadSize) {
#ifndef NDEBUG
          // std::cout << "randomize: end of payload." << std::endl;
#endif
          byteCount = 0;
          lfsrState = lfsrInitialState;
        }
      }

      return noutput_items;
    }
```

A.12 CRC-16

```
    int
    CRC16_impl::general_work (int noutput_items,
                      gr_vector_int &ninput_items,
                      gr_vector_const_void_star &input_items,
                      gr_vector_void_star &output_items)
    {
      const uint8_t *in = (const uint8_t *) input_items[0];
      // uint16_t *out;
      // if(output_items.size() > 0)
      //         out = (uint16_t *) output_items[0];

// #ifndef NDEBUG
//       std::cout << std::dec << "CRC16: work called, noutput_items = " <<
    noutput_items << ", payloadSize = " << payloadSize << std::endl;
// #endif

      //check for payloadSize tags
      std::vector<gr::tag_t> tags;
      get_tags_in_range(tags, 0, nitems_read(0), nitems_read(0) + 1, pmt::intern("
          payloadSize"));
      if(tags.size() != 0) {
        size_t payloadSizeNew = pmt::to_long(tags[0].value);
```

```
        if(payloadSizeNew <= 255) {
          setPayloadSize(payloadSizeNew);
        } else {
          setPayloadSize(1);
#ifndef NDEBUG
          std::cout << "CRC16: got invalid payloadSize." << payloadSize << std::
              endl;
#endif
        }
      }

      if(payloadSize == 0){
        consume_each(noutput_items);
        return 0;
      }

      for(size_t j = 0; j < noutput_items; j++) {
        // const uint8_t* inJ = in + j*payloadSize;

        // uint32_t crc = 0x0000;
        // for(size_t i = 0; i < payloadSize; i++) {
        crc <<= 8;
          // crc = polDivRem(crc^uint16_t(inJ[i]), polynomial);
        crc = polDivRem(crc^uint16_t(in[j]), polynomial);
        // }
        byteCount++;
        // if(output_items.size() > 0)
        //   out[j] = crc^xorOut;

        if(byteCount == payloadSize) {
          message_port_pub(crcOutPort, pmt::from_long(crc^xorOut));

#ifndef NDEBUG
          std::cout << std::hex << "CRC16: calculated CRC: " << /*unsigned(out[j])
              */unsigned(crc^xorOut) << std::endl;
#endif

          byteCount = 0;
          crc = 0x0000;
        }
      }
      // Tell runtime system how many input items we consumed on
      // each input stream.
      // consume_each (payloadSize*noutput_items);
      consume_each(noutput_items);

      // payloadSize = 0;
      // Tell runtime system how many output items we produced.
      return noutput_items;
    }
```

A.13 Bytes To Nibbles

```
    int
    BytesToNibbles_impl::work(int noutput_items,
        gr_vector_const_void_star &input_items,
        gr_vector_void_star &output_items)
    {
      const uint8_t *in = (const uint8_t *) input_items[0];
      uint8_t *out = (uint8_t *) output_items[0];
```

```
      for(size_t i = 0; i < noutput_items/2; i++){
        out[2*i] = in[i]&0x0f;
        out[2*i + 1] = (in[i] >> 4)&0x0f;

#ifndef NDEBUG
        std::cout << "BytesToNibbles: in = " << unsigned(in[i]) << ", out = " <<
            unsigned(out[2*i]) << ", " << unsigned(out[2*i + 1]) <<  std::endl;
#endif
      }

      // Tell runtime system how many output items we produced.
      return noutput_items;
    }
```

A.14 Transmitter Controller

```
    int
    TransmitterController_impl::general_work (int noutput_items,
                      gr_vector_int &ninput_items,
                      gr_vector_const_void_star &input_items,
                      gr_vector_void_star &output_items)
    {
      const uint8_t *in = (const uint8_t *) input_items[0];
      uint8_t *out = (uint8_t *) output_items[0];

      std::vector<gr::tag_t> tags;

      switch(currentState) {
      case sendingHeader:
        //check for payloadSize tags
        get_tags_in_range(tags, 0, nitems_read(0), nitems_read(0) + 1, pmt::intern
            ("payloadSize"));
        if(tags.size() != 0) {
          size_t payloadSizeNew = pmt::to_long(tags[0].value);
          if(payloadSizeNew <= 255) {
            setPayloadSize(payloadSizeNew);
#ifndef NDEBUG
            std::cout << "TransmitterController: set payloadSize to: " <<
                payloadSize << std::endl;
#endif
          } else {
            setPayloadSize(1);
#ifndef NDEBUG
            std::cout << "TransmitterController: got invalid payloadSize. set
                payloadSize to 1." << payloadSize << std::endl;
#endif
          }
        }

        //check for loraFrameParams tag
        get_tags_in_range(tags, 0, nitems_read(0), nitems_read(0) + 1, pmt::intern
            ("loraFrameParams"));
        if(tags.size() != 0) {
            pmt::pmt_t message = tags[0].value;
            SF = pmt::to_long(pmt::tuple_ref(message, 0));
            CR = pmt::to_long(pmt::tuple_ref(message, 1));
            CRCPresent = pmt::to_bool(pmt::tuple_ref(message, 2));
            lowDataRate = pmt::to_bool(pmt::tuple_ref(message, 3));
            float BW = pmt::to_float(pmt::tuple_ref(message, 4));
            symbolSize = std::round(float(1 << SF)/BW);
            //syncWord still not implemented
```

```
            calculateConstants();
        }

#ifndef NDEBUG
        std::cout << "TransmitterController: sending header." << std::endl;
#endif

        setSFCurrent(SF-2);
        setCRCurrent(4);

        sendParamsTag(true);
        message_port_pub(transmissionStartPort, pmt::PMT_NIL);

        //output header
        memcpy(out, headerNibbles, 5*sizeof(uint8_t));

        if(nNibbles > (SF - 7)) {
          memcpy(out + 5, in, (SF - 7)*sizeof(uint8_t));
          currentState = sendingPayload;
          consume_each(SF - 7);
        } else {
          memcpy(out + 5, in, 2*payloadSize*sizeof(uint8_t));

          if(CRCPresent) {
            uint32_t crcNibbles = bytes2nibbles<uint32_t, uint16_t>(crc);
            memcpy(out + 5 + 2*payloadSize, &crcNibbles, 2*CRCSize*sizeof(uint8_t)
                );
          }

          memset(out + 5 + nNibbles, 0, nNibblesTotal - nNibbles);
          consume_each(2*payloadSize);

          //send end of frame tag
          static const pmt::pmt_t tagKey = pmt::intern("loraEndOfFrame");
          add_item_tag(0, nitems_written(0) + (SF - 2) - 1, tagKey, pmt::PMT_NIL);
        }

#ifndef NDEBUG
        // std::cout << "TransmitterController: produced nibbles: " << std::endl;
        // for(auto i = 0; i < (SF - 2); i++)
        //   std::cout << std::hex << unsigned(out[i]) << " ";
        // std::cout << std::endl;
#endif

        return (SF - 2);
      case sendingPayload:

#ifndef NDEBUG
        std::cout << "TransmitterController: sending payload." << std::endl;
#endif

        setSFCurrent(lowDataRate? (SF - 2) : SF);
        setCRCurrent(CR);

        sendParamsTag();

        //output payload
        memcpy(out, in, (2*payloadSize - (SF - 7))*sizeof(uint8_t));

#ifndef NDEBUG
        std::cout << "TransmitterController: sent payload." << std::endl;
#endif
```

```
        if(CRCPresent) {
           //caculate crc
          // uint16_t crc = calculateCRCFromNibbles(in);
          uint32_t crcNibbles = bytes2nibbles<uint32_t, uint16_t>(crc);

          memcpy(out + (2*payloadSize - (SF - 7)), &crcNibbles, 2*CRCSize*sizeof(
              uint8_t));
        }

#ifndef NDEBUG
        std::cout << "TransmitterController: sent crc." << std::endl;
        std::cout << (nNibblesTotal - ((2*payloadSize - (SF - 7)) + 2*(CRCPresent
            ? CRCSize : 0))) << std::endl;
#endif

        memset(out + (2*payloadSize - (SF - 7)) + 2*(CRCPresent ? CRCSize : 0), 0,
               (nNibblesTotal - ((2*payloadSize - (SF - 7)) + 2*(CRCPresent ?
                   CRCSize : 0)))*sizeof(uint8_t));

#ifndef NDEBUG
        std::cout << "TransmitterController: sent extra nibbles." << std::endl;
#endif

#ifndef NDEBUG
        // std::cout << "TransmitterController: produced nibbles: " << std::endl;
        // for(auto i = 0; i < nNibblesTotal - (SF - 7); i++)
        //   std::cout << std::hex <<  unsigned(out[i]) << " ";
        // std::cout << std::endl;
#endif

        //send end of frame tag
        static const pmt::pmt_t tagKey = pmt::intern("loraEndOfFrame");
        add_item_tag(0, nitems_written(0) + nNibblesTotal - (SF - 7) - 1, tagKey,
            pmt::PMT_NIL);

        currentState = sendingHeader;

        // Tell runtime system how many input items we consumed on
        // each input stream.
        consume_each (2*payloadSize - (SF - 7));

        // Tell runtime system how many output items we produced.
        return nNibblesTotal - (SF - 7);
      }

      return 0;
    }
```

A.15 Encoder

```
    int
    Encode_impl::work(int noutput_items,
        gr_vector_const_void_star &input_items,
        gr_vector_void_star &output_items)
    {
      const uint8_t *in = (const uint8_t *) input_items[0];
      uint8_t *out = (uint8_t *) output_items[0];

#ifndef NDEBUG
      std::cout << "Encode: encoding: " << std::dec << noutput_items << " items"
          << ", CR = " << CR << std::endl;
```

```
#endif

      static const pmt::pmt_t tagKey = pmt::intern("loraParams");

      for(size_t i = 0; i < noutput_items; i++) {

        std::vector<gr::tag_t> tags;
        auto nr =  nitems_read(0);
        get_tags_in_range(tags, 0, nr + i, nr + i + 1, tagKey);
        if(tags.size() != 0) {

          pmt::pmt_t message = tags[0].value;
          size_t CRnew = pmt::to_long(pmt::tuple_ref(message, 1));
          setCR(CRnew);
#ifndef NDEBUG
          std::cout << "Encode:␣" << "changed␣CR␣to:␣" << CR << "␣(tag)." << std::
              endl;
#endif
        }

        out[i] = calculatePairity(in[i], pairityMatrix);

#ifndef NDEBUG
        std::cout << std::hex << "Encode:␣in:␣" << unsigned(in[i]) << ",␣out:␣"<<
             std::hex << unsigned(out[i]) << ",␣CR␣=␣" << CR << std::endl;
#endif
      }


      // Tell runtime system how many output items we produced.
      return noutput_items;
    }
```

A.16 Interleaver

```
    int
    Interleave_impl::general_work (int noutput_items,
                      gr_vector_int &ninput_items,
                      gr_vector_const_void_star &input_items,
                      gr_vector_void_star &output_items)
    {
      const uint8_t *in = (const uint8_t *) input_items[0];
      uint16_t *out = (uint16_t *) output_items[0];

      set_relative_rate(codeLength, SF);

      size_t codeLengthInitial = codeLength;
      size_t SFInitial = SF;

      const size_t blocksToProduce = (noutput_items + codeLength - 1)/codeLength;

#ifndef NDEBUG
      // std::cout << "Interleave: blocksToProduce = " << blocksToProduce << std::
          endl;
#endif

      for(size_t k = 0; k < codeLength; k++)
        out[k] = 0;

      size_t i;
      for(i = 0; i < blocksToProduce; i++) {
```

```
        if(!changedParams || i != 0) {

          //read loraParam tag if present
          std::vector<gr::tag_t> tags;
          auto nr =  nitems_read(0);
          static const pmt::pmt_t tagKey = pmt::intern("loraParams");
          get_tags_in_range(tags, 0, nr + i*SF, nr + i*SF + 1, tagKey);
          if(tags.size() != 0) {
            // add_item_tag(0, nitems_written(0) + i*codeLength, tagKey, tags[0].
                value);

            pmt::pmt_t message = tags[0].value;
            size_t SFnew = pmt::to_long(pmt::tuple_ref(message, 0));
            setSF(SFnew);
            size_t CRnew = pmt::to_long(pmt::tuple_ref(message, 1));
            setCR(CRnew);
            changedParams = true;

            break;
          }
        } else {

          changedParams = false;
          set_relative_rate(codeLength, SF);
          codeLengthInitial = codeLength;
          SFInitial = SF;

#ifndef NDEBUG
          std::cout << "Interleave:␣updated␣parameters." << std::endl;
#endif
        }

#ifndef NDEBUG
        std::cout << "Interleave:␣propagating␣Tags.␣nBlocks:␣" << blocksToProduce
            << std::endl;
#endif
        //propagate tags in the beggining of blocks
        std::vector<gr::tag_t> tags;
        auto nr =  nitems_read(0);
        get_tags_in_range(tags, 0, nr + i*SF, nr + i*SF + 1);
        for(auto tag : tags)
          add_item_tag(0, nitems_written(0) + i*codeLength, tag.key, tag.value);

        //propagate tags in the end of blocks
        tags.clear();
        get_tags_in_range(tags, 0, nr + i*SF + SF - 1, nr + i*SF + SF - 1 + 1);
        for(auto tag : tags)
          add_item_tag(0, nitems_written(0) + i*codeLength + codeLength - 1, tag.
              key, tag.value);

#ifndef NDEBUG
        std::cout << "Interleave:␣interleaving␣symbols:␣";
        for(size_t j = 0; j < SF; j++)
          std::cout << std::hex << unsigned(in[i*SF + j]) << "␣";
        std::cout << ",␣SF␣=␣" << SF << ",␣CR␣=␣" << codeLength - 4 << ",␣nBlocks:
            ␣" << blocksToProduce << std::endl;
#endif

        for(size_t j = 0; j < SF; j++)
          for(size_t k = 0; k < codeLength; k++)
            out[i*codeLength + k] |= (in[i*SF + (j + k)%SF] >> k & 0x01) << j;
```

```
#ifndef NDEBUG
        // std::cout << "Interleave: interleaved symbols: ";
        // for(size_t k = 0; k < codeLength; k++)
        //   std::cout << std::hex << unsigned(out[i*codeLength + k]) << " ";
        // std::cout << std::endl;
#endif
      }

      // Tell runtime system how many input items we consumed on
      // each input stream.
      // consume_each (noutput_items);
      consume_each(SFInitial*i);

      // Tell runtime system how many output items we produced.
      // return noutput_items;
      return codeLengthInitial*i;
    }
```

A.17 Gray Decoder

```
    int
    GrayDecode_impl::work(int noutput_items,
        gr_vector_const_void_star &input_items,
        gr_vector_void_star &output_items)
    {
      const uint16_t *in = (const uint16_t *) input_items[0];
      uint16_t *out = (uint16_t *) output_items[0];

      // Do <+signal processing+>
      for(size_t i = 0; i < noutput_items; i++) {

        std::vector<gr::tag_t> tags;
        auto nr =  nitems_read(0);
        static const pmt::pmt_t tagKey = pmt::intern("loraParams");
        get_tags_in_range(tags, 0, nr + i, nr + i + 1, tagKey);
        if(tags.size() != 0) {
          pmt::pmt_t message = tags[0].value;
          size_t SFnew = pmt::to_long(pmt::tuple_ref(message, 0));
          setNBits(SFnew);
        }

        out[i] = grayDecode(in[i], nBits);

#ifndef NDEBUG
        std::cout << "GrayDecode: in: " << std::hex << in[i] << ", out: " << out[i
            ] << ", nBits = " << nBits << std::endl;
#endif
      }

      // Tell runtime system how many output items we produced.
      return noutput_items;
    }
```

A.18 Symbol Modulator

```
    int
    SymbolMod_impl::work(int noutput_items,
        gr_vector_const_void_star &input_items,
        gr_vector_void_star &output_items)
    {
      const uint16_t *in = (const uint16_t *) input_items[0];
      float *out = (float *) output_items[0];

#ifndef NDEBUG
      std::cout << "SymbolMod: modulating " << noutput_items/symbolSize <<  "
          symbols, SF = " << SF << std::endl;
#endif

      for(auto i = 0; i < noutput_items/symbolSize; i++){
        //propagate endOfFrame Tag
        std::vector<gr::tag_t> tags;
        auto nr =  nitems_read(0);
        get_tags_in_range(tags, 0, nr + i, nr + i + 1, pmt::intern("loraEndOfFrame
            "));
        for(auto tag : tags)
          add_item_tag(0, nitems_written(0) + i*symbolSize + symbolSize - 1, tag.
              key, tag.value);

        static const pmt::pmt_t tagKey = pmt::intern("loraParams");
        get_tags_in_range(tags, 0, nr + i, nr + i + 1, tagKey);
        if(tags.size() != 0) {
          pmt::pmt_t message = tags[0].value;
          size_t SFNew = pmt::to_long(pmt::tuple_ref(message, 0));
          const bool isBeginning = pmt::to_bool(pmt::tuple_ref(message, 2));
          if(isBeginning) {
             setSF(SFNew + 2);
          }
          setSFCurrent(SFNew);

          size_t symbolSizeNew = pmt::to_long(pmt::tuple_ref(message, 3));
          setSymbolSize(symbolSize);

          add_item_tag(0, nitems_written(0) + i*symbolSize, tagKey, message);
        }

        const std::vector<float> symi = getSymbol<float>(in[i]*(1 << (SF -
            SFCurrent)), SF, symbolSize);
        memcpy(out + i*symbolSize, symi.data(), symbolSize*sizeof(float));

#ifndef NDEBUG
        std::cout << "SymbolMod: modulated symbol: " << std::dec << in[i] << ", SF
            = " << SFCurrent << std::endl;
#endif
      }

      // Tell runtime system how many output items we produced.
      return (noutput_items/symbolSize)*symbolSize;
    }
```

A.19 Append Prefix

```
    int
    AppendPrefix_impl::general_work (int noutput_items,
                      gr_vector_int &ninput_items,
                      gr_vector_const_void_star &input_items,
                      gr_vector_void_star &output_items)
    {
      const float *in = (const float *) input_items[0];
      float *out = (float *) output_items[0];

      size_t nFrames = 0;
      size_t produced = 0;

      size_t i;
      for(i = 0; i < noutput_items; i++) {

        //propagate tags
        auto nr =  this->nitems_read(0);
        this->get_tags_in_range(tags, 0, nr + i, nr + i + 1);
        for(auto tag : tags) {
          this->add_item_tag(0, this->nitems_written(0) + i + nFrames*prefix.size
              (), tag.key, tag.value);
        }

        tags.clear();
        static const pmt::pmt_t tagKey = pmt::intern("loraParams");
        this->get_tags_in_range(tags, 0, nr + i, nr + i + 1, tagKey);

        if(tags.size() != 0) {
#ifndef NDEBUG
          // std::cout << "AppendPrefix: got tag. tags.size() = " << tags.size()
              << std::endl;
#endif

          const bool isBeginning = pmt::to_bool(pmt::tuple_ref(tags[0].value, 2));
          if(isBeginning) {
            const size_t SFNew = pmt::to_long(pmt::tuple_ref(tags[0].value, 0)) +
                2;
            if(SFNew != SF) {
              SF = SFNew;
              calculatePrefix();
              this->set_max_noutput_items(prefix.size());
              this->set_min_output_buffer(this->max_noutput_items() + prefix.size
                  ());
            }

            memcpy(out + i + nFrames*prefix.size(), prefix.data(), prefix.size()*
                sizeof(float));

            // this->add_item_tag(0, this->nitems_written(0) + i + nFrames*prefix.
                size(), tagKey, tags[0].value);

            nFrames++;
            produced += prefix.size();
#ifndef NDEBUG
            std::cout << "AppendPrefix: prefix inserted." << std::endl;
#endif
          }
        }

        out[i + nFrames*prefix.size()] = in[i];
```

```
        produced++;

        if(produced >= noutput_items) {
#ifndef NDEBUG
          // std::cout << "AppendPrefix: sent everything. i = " << i << std::endl;
#endif
          break;
        }
      }

#ifndef NDEBUG
      // std::cout << "AppendPrefix: work ended. i = " << i << ", noutput_items =
          " << noutput_items << ", produced = " << produced <<  std::endl;
#endif
      // Do <+signal processing+>
      // Tell runtime system how many input items we consumed on
      // each input stream.
      // gr::block::consume_each (noutput_items);
      gr::block::consume_each(i + 1);

      // Tell runtime system how many output items we produced.
      // return nFrames*prefix.size() + noutput_items;
      return produced;
    }
```

A.20 Frequency Modulator

```
int
FrequencyMod_impl::work(int noutput_items,
    gr_vector_const_void_star &input_items,
    gr_vector_void_star &output_items)
{
  const float *freqIn = (const float *) input_items[0];
  gr_complex *out = (gr_complex *) output_items[0];

  for(auto i = 0; i < noutput_items; i++){
    w *= std::polar<float>(1.0f, 2*M_PI*freqIn[i]);
    out[i] = w;
  }


  // Tell runtime system how many output items we produced.
  return noutput_items;
}
```

A.21 Append Silence

```
    int
    AppendSilence_impl::general_work (int noutput_items,
                      gr_vector_int &ninput_items,
                      gr_vector_const_void_star &input_items,
                      gr_vector_void_star &output_items)
    {
      const gr_complex *in = (const gr_complex *) input_items[0];
      gr_complex *out = (gr_complex *) output_items[0];

      size_t nSamplesUsed = 0;

      if(!transmittingFrame) {
        memset(out, 0, noutput_items*sizeof(gr_complex));
```

```
      } else
        for(auto i = 0; i < noutput_items; i++) {
          if(transmittingFrame) {
            out[i] = in[i];

            nSamplesUsed++;

            //check is next sample has endOfFrame tag
            std::vector<gr::tag_t> tags;
            auto nr =  nitems_read(0);
            static const pmt::pmt_t tagKey = pmt::intern("loraEndOfFrame");
            get_tags_in_range(tags, 0, nr + i, nr + i + 1, tagKey);
            if(tags.size() != 0) {
#ifndef NDEBUG
              std::cout << "AppendSilence: got end of frame." << std::endl;
#endif
              transmittingFrame = false;
            }
          } else {
            out[i] = 0;
          }
        }


      // Tell runtime system how many input items we consumed on
      // each input stream.
      consume_each (nSamplesUsed);

      // Tell runtime system how many output items we produced.
      return noutput_items;
    }
```