Telecommunications Circuits Laboratory
EPFL, Lausanne

Tapparel Joachim

Complete Reverse Engineering of LoRa PHY

Supervisor: Prof. Burg Andreas, EPFL-STI-IEL-TCL
Advisor: Afisiadis Orion, EPFL-STI-IEL-TCL, Alexios Balatsoukas-Stimming TUe Eindhoven

Contents

Chapter 1

Introduction

This report describes the reverse engineering of the entire LoRa PHY transceiver chains. These reverse engineering details are used in our fully fanctional GNU Radio LoRa prototype. In the reverse engineering process we used a commercial LoRa transceiver (Adafruit Feather 32u4 RFM95W) along with an NI USRP-2920. This project is the continuation of work previously done in the laboratory [1] [2] [3].

Chapter 2

LoRa Physical layer

2.1 LoRa modulation

As the main focus of this project is the decoding of LoRa symbols, we will only briefly introduce the lora modulation concepts that are required to understand the reverse engineering steps that will be described later. A more in-depth presentation of the modulation and demodulation processes is available in [1].

The LoRa modulation is based on chirp spread spectrum (CSS) which is a method consisting of sweeping a sinusoidal signal frequency in a defined bandwidth. In our case this shift is performed linearly and can be done increasingly or decreasingly which is what we call either an upchirp or a downchirp. This method provides many advantages such as bandwidth scalability, low-power consumption with constant envelope, high robustness toward fast frequency hopping spread spectrum systems, resistant to multipath, fading and Doppler effect and long range capability [4]. A chirp is defined by the bandwidth (BW) it covers, and the spreading factor (SF) it uses. With those two parameters we get a symbol period of $T_s = \frac{2^{\text{SF}}}{\text{BW}}$ secs. The method used to encode a symbol S in a chirp is to add a starting offset to the frequency sweep which is given by the following relation:

$$f_{\text{offset}} = \frac{BW}{2^{\text{SF}}} * \text{S} \quad \text{with S} \in [0, 2^{\text{SF}} - 1] \tag{2.1}$$

Since the transmission is restricted by a bandwidth going from $-\frac{\text{BW}}{2}$ to $\frac{\text{BW}}{2}$, by choosing an offset greater than zero we will attain the higher frequency of the bandwidth before the end of the symbol period. By setting the sampling frequency to BW, we will fold the frequencies higher than $\frac{\text{BW}}{2}$ back to $-\frac{\text{BW}}{2}$ because of the disrespect of the Nyquist–Shannon sampling theorem. Therefore we can express a modulated LoRa symbol in two parts, one before that the folding occurs at a time $t_{\text{fold}} = \frac{2^{\text{SF}} - S}{\text{BW}}$ and one after, by the following expression [5]:

$$x_s(t) = \begin{cases} e^{j2\pi\left(\frac{\text{BW}}{2T_s}t^2 + (f_{\text{offset}} - \frac{\text{BW}}{2})t\right)} & \text{for } 0 \leq t \leq t_{\text{fold}} \\ e^{j2\pi\left(\frac{\text{BW}}{2T_s}t^2 + (f_{\text{offset}} - \frac{3\text{BW}}{2})t\right)} & \text{for } t_{\text{fold}} \leq t \leq T_s \end{cases} \tag{2.2}$$

The figure 2.1 shows an example of modulated LoRa symbols visualized on a spectrogram.

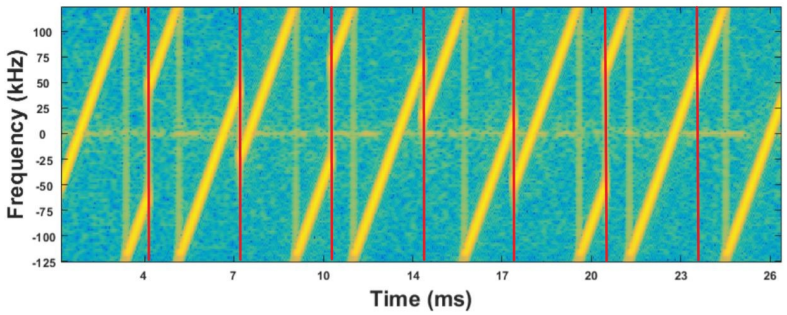

Figure 2.1: LoRa symbols observed with a spectrogram

The method used to recover the integer value encoded in each chirp is to multiply each received chirp by the complex conjugate of the chirp encoding the value 0, which corresponds to a downchirp. We can then apply a DFT on this dechirped version of our symbol to find the main frequency present which should be no other than f_{offset}.

2.2 Preamble

Each LoRa packet is initiated with a preamble used for transmission detection, frame synchronization and frequency synchronization. From the LoRa patent [6], we know that this preamble is composed of a series of unmodulated chirps, some symbols having a predetermined modulation value for frame synchronization and finally some symbols being the complex conjugate of the unmodulated symbol for frequency synchronization.

The figure 2.2 present a more exhaustive description of the preamble structure, as outlined by Pieter Robyns [7]. The structure being a repetition of unmodulated upchirps which number can be define by the user, 2 modulated upchirps and 2.25 unmodulated downchirps.

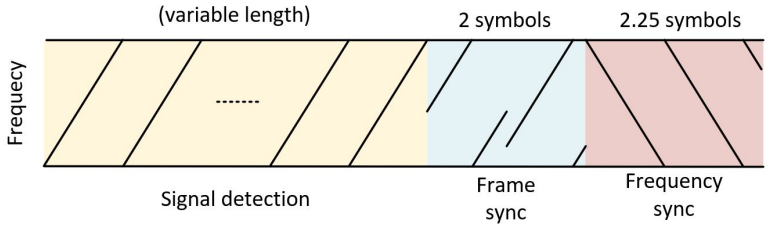

Figure 2.2: Preamble structure

2.3 LoRa encoding scheme

The encoding and decoding process of the data bits are performed through four steps, which are presented in figure 2.3. We will describe the operations performed by each of them as well as the method used to highlight their functioning principle. In order to better understand how their functioning was deduced, we will not present them in their sequential order but in the order allowing their revere engineering.

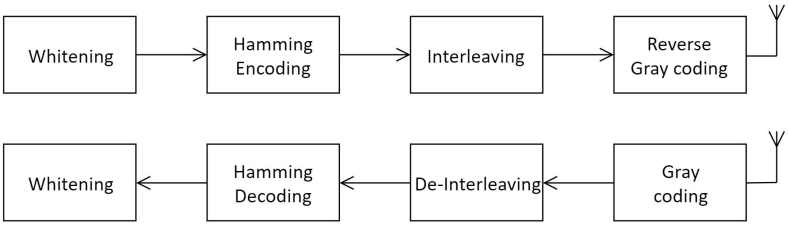

Figure 2.3: Encoding/Decoding scheme

2.3.1 Gray coding

A gray coding is a mapping between a symbol in any numeric representation to a binary sequence. The particularity of the obtained binary sequence is that adjacent symbols in the original representation only differ by one bit in the gray representation. This property of a gray codes make them very useful in many wireless communication modulations where it is more likely to misinterpret a symbol by an adjacent one than another random one. It is noteworthy that in the case of CSS modulation it is not the white noise that causes this misinterpretation between adjacent symbols but the carrier and sampling frequency offset, as explained in [5]. In the case of LoRa, the symbols can be seen as integers between 0 and $2^{\mathrm{SF}} - 1$ and by making usage of gray coding to map them to binary strings, we increase the performance of the forward error correction mechanisms capable of correcting one erroneous bit in a codeword.

Unfortunately, it is possible to create different binary string sequences respecting the conditions to be considered as a gray code. Furthermore it is just a mapping between decimal numbers and binary string, meaning that once we have found a sequence of 2^{SF} binary string differing by only one bit, we can still define any of them to by mapped to 0. Finally the direction of mapping of the following symbol can also be chosen freely. This leads to $2 * 2^{SF}$ possibilities for every possible gray sequence. The method used in [2] was a brute force approach, which found out that the method of the figure 2.4 is used. The standard gray code being used is given by $C_n = B_n$ XOR $(B_n >> 1)$ where B_n is the left MSB binary representation of n. On top of the mapping using this gray code, a shift of -1 is used.

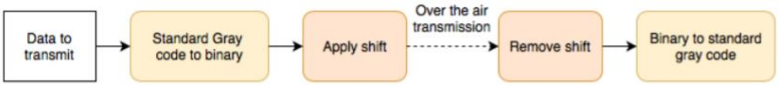

Figure 2.4: LoRa ”reverse” Gray coding algorithm

We can note that the gray coding doesn't occur in the conventional order but in reverse which still holds the property that one adjacent symbol mistake leads only to one bit being wrong.

The figure 2.5 present the binary values obtained after gray encoding of the received LoRa symbols line by line. The white and black square are respectively ones and zeros.

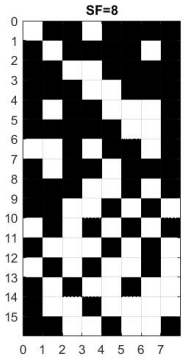

Figure 2.5: Gray coded LoRa symbols line by line
with ones as white square and zeros in black

2.3.2 Interleaving

During the transmission a symbol can be corrupted by noise or fading, leading to a multiple bits errors after demodulation. Because all errors are caused by one symbol, they are highly correlated. The issue with such errors is that they are not well handled by error correction codes, designed to correct random errors in codeword. The objective is to break this correlation between the erroneous bit by distributing the errors over multiple codewords. It is possible by spreading the bits constituting a codeword between multiple LoRa symbols. This operation increases the effectiveness of the error correction code at the cost of an increased latency caused by the necessity of receiving multiple symbols before being able to recover one full codeword.

Reverse engineering

Since we want to only observe the effect of the interleaver on our data, we have to make abstraction of the others blocks. Thanks to the LoRa patent [6] we know that some variant of Hamming coding is used and one particularity of Hamming code is that all zero data gets all zero parity bits, as well as all ones data gets all ones parity bits.

Only the whitening still occurs and require a method in order to make abstraction of his effect. Since the whitening consist of a XOR between the data and a pseudo random sequence, we can use the following property:

$$C_1 \oplus C_2 = M_1 \oplus W \oplus M_2 \oplus W = M_1 \oplus M_2 \tag{2.3}$$

were C is a whitened message, M the original message and W the whitening sequence. The result of $M_1 \oplus M_2$ will be refereed as a difference message.

The figure 2.6 shows the 16 first symbols of the difference messages resulting of payloads entirely composed of zeros and others of all ones. The parameters used are a constant coding rate of 4/8 and a spreading factor of seven, eight and nine. The objective of those messages is to indicate the position of the payload bits inside a packet. Indeed in the difference message, we expect to have a one when changing the payload effectively affected this particular bit. The first observation is that there exist 80 bits in the first 16 symbols of the packet that do not depends on the payload content. It is unexpected since we send these messages in implicit header mode, which in theory shouldn't include any header and the packet content should only depend on our payload. From now on we will refer to this part as the implicit header, and its content will be discussed further in the section 2.4.

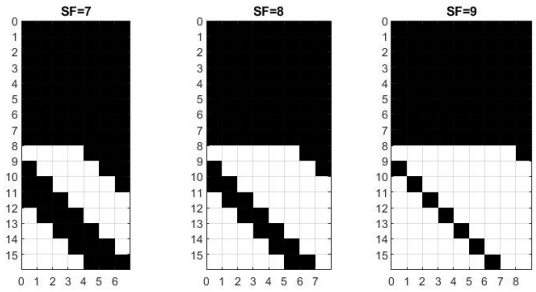

Figure 2.6: Difference messages for a coding rate of 4/8 and different spreading factors

By changing the coding rate to 4/6 we obtain the pattern of the figure 2.7. From this message, we can observe that the coding rate impact the number of symbols that are interleaved together. Moreover, as highlighted by the red boxes, there is a first bunch of 8 symbols that are interleaved using a coding rate of 4/8 independently of the chosen coding rate. Only after this first bunch the desired coding rate is used. This reduce rate for the beginning of the packet is mostly destined to the explicit header mode for which it is very important to receive a correct header since the decoding of the rest of the message depends on it. With a coding rate of 4/6, we get another number of implicit header bits but by taking into account that the first 8 symbols are coded using a coding rate of 4/8 and only the following ones are using 4/6, we get $\frac{4}{8} \cdot 64 + \frac{4}{6} \cdot 12 = 40$ which is equal to the previous $\frac{4}{8} \cdot 80$ data bits.

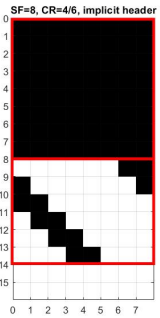

Figure 2.7: Difference message for a Coding rate of 4/6 and a spreading factor of 8

The next step will be to extract the interleaving method that is used to distribute the bits of each codeword in different symbols. By generating difference messages coming from payload varying only by one nibble, we should be able to observe the position of each codeword in the packet after the interleaving.

The figure 2.8 presents the differences messages obtained by inverting the first eight nibbles one at a time, using a coding rate of 4/8 and a spreading factor of 8. We can effectively observe that every bits of a codeword get distributed to a different LoRa symbol. Furthermore it appears that when the original bytes are split into two nibbles, the nibble containing the LSBs is sent first. This reordering becomes evident by looking at the figure 2.9 which present the same patterns as in the previous figure swapped two by two. In that order the pattern appears to be shifted by one to the left each time the next nibble of the payload is flipped.

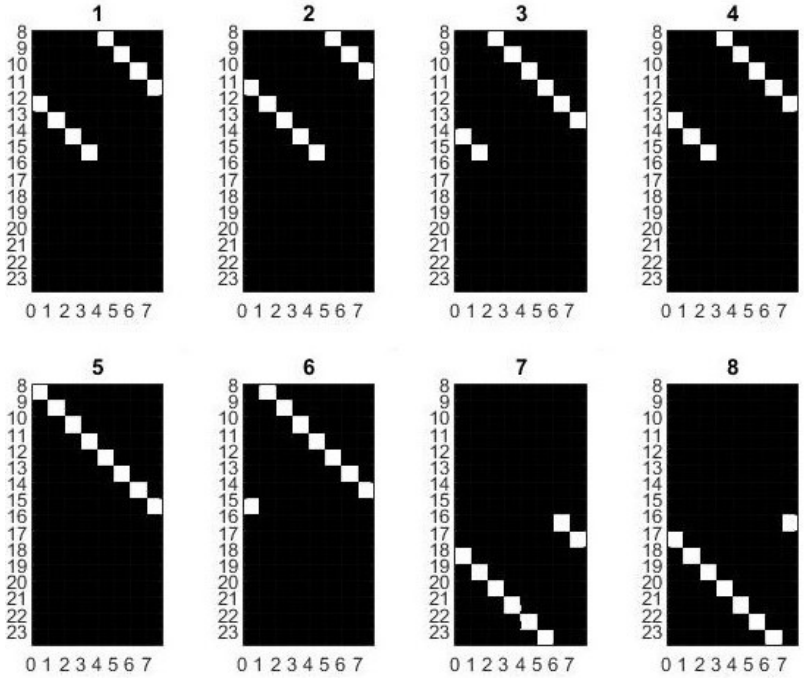

Figure 2.8: Interleaving pattern for SF=8 and CR=4/5

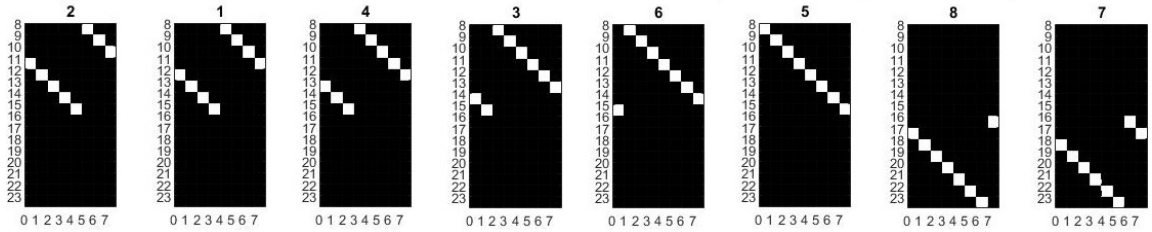

Figure 2.9: Interleaving pattern swapped 2 by 2 for SF=8 and CR=4/5

The figure 2.10 present the codeword before and after interleaving that can be extracted from the pattern obtained previously. The patterns for SF=7, CR=4/8 and SF=9, CR=4/5 have been obtain following the same procedure. We can note that during the interleaving we get from a matrix of dimension SF$\times\frac{4}{\text{CR}}$ to $\frac{4}{\text{CR}}\times$SF, the latter being able to be used line by line to modulate a LoRa symbol of spreading factor = SF. From these patterns, we can finally extract the following general interleaving formula:

$$I_{i,j} = D_{j,(i-(j+1)\%\text{SF})} \tag{2.4}$$

where I is the interleaved matrix and D the deinterleaved one.

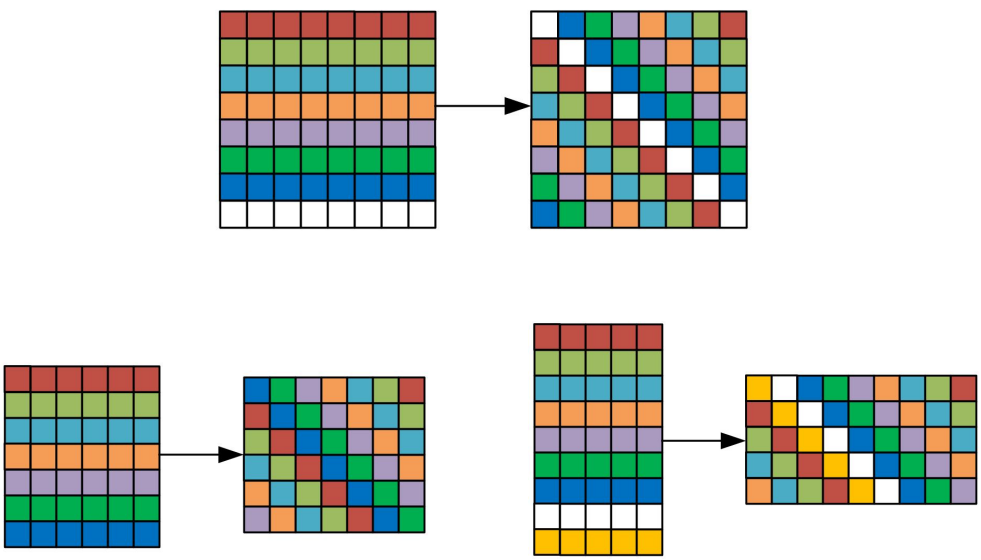

Figure 2.10: Interleaving pattern for SF=8, CR=4/8; SF=7, CR=4/6 and SF=9, CR=4/5

Now that we are able to deinterleave the codewords, we can start to tackle the whitening block since based on Pieter Robyns paper [7] it should be the block coming just before interleaving.

2.3.3 Whitening

Whitening is a method consisting of XORing bits with a pseudo random sequence with the objective of removing DC-bias in the transmitted data. Unlike Manchester coding, it provides the advantage of keeping the same data rate at the cost of not having the guarantee of removing any DC-bias but only a very high probability of doing it.

Reverse engineering

Recovering the whitening sequence is fairly straight forward since $a \oplus 0 = a$, meaning that by sending an all zeros message, the received bits will correspond to the whitening sequence. Except for the portion of the whitening sequence that corresponds to the implicit header since that portion can't be set to all-zeros like the rest of the payload. By recovering the whitening matrices of different coding rates and spreading factors, we get the same sequence for every spreading factor. This invariance toward the spreading factor proves that the interleaving effectively occurs after the whitening since if the opposite order was correct, no invariance toward spreading factor or coding rate could have been observed after the deinterleaving.

The figure (fig.2.11) shows the whitening sequence obtained for the coding rates from 4/8 to 4/5. Since we recover the packet content in a bi-dimensional way, the whitening can't be considered as a sequence but as a matrix. It can be observed that the matrix of CR=4/8 appears to be the most complete one and that only the required first columns are kept by other coding rates. This observation however doesn't work for the 4/5 coding rate, indeed the last column is different from the fifth columns of the other whitening matrices. The exact reason of the different bits in the last columns of the 4/5 coding rate will be presented in the the following section and are caused by a special coding used only for that coding rate.

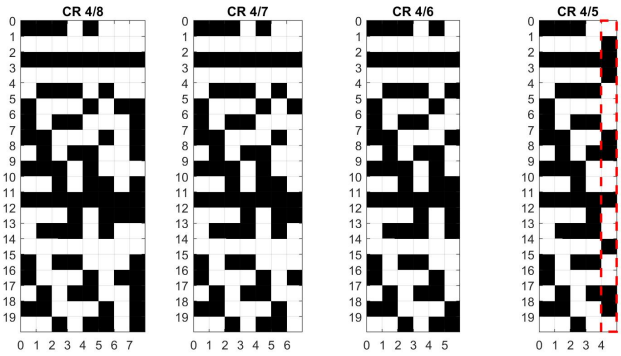

Figure 2.11: Whitening matrices for different coding rates

Even if we can use the observations described above to correctly dewhiten the data, the exact causes are let unexplained at this point as they require a deeper understanding of the Hamming coding used. However we have to be able to dewhiten the data to acquire that understanding and therefore we will use the whitening matrices obtained without questioning there correctness.

2.3.4 Hamming Coding

From the patent [6], we know that some kind of hamming coding is used. Hamming code is an error correcting code meaning that he should be able to correct errors by adding redundancy in each codeword. This can be possible with a coding rate of 4/8 and 4/7 but can't be done for the other ones, meaning that the Hamming encoding mentioned in the patent can only be a variation of a traditional Hamming coding.

Reverse engineering

Until now the bit position inside each codeword was undefined since the exact composition of the codeword was not known. After sending known data and deinterleaving them, it appears that we receive them in the order presented in figure 2.12. There are two important things to be aware of: the first is that the names given to the parity bits are arbitrary since any other naming will only require to define different generator and parity check matrices. The second is that d0 is the LSB of the nibble.

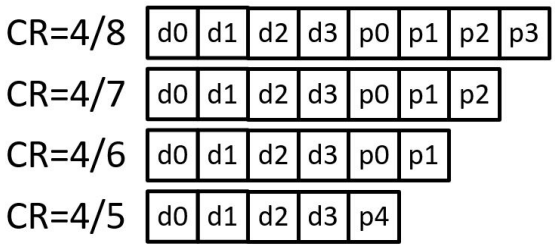

Figure 2.12: Bits in each deinterleaved codeword

The parity bits 0 to 3 are calculated as follows:

$$p_0 = d_0 \oplus d_1 \oplus d_2 \tag{2.5}$$
$$p_1 = d_1 \oplus d_2 \oplus d_3 \tag{2.6}$$
$$p_2 = d_0 \oplus d_1 \oplus d_3 \tag{2.7}$$
$$p_3 = d_0 \oplus d_2 \oplus d_3 \tag{2.8}$$

However for a coding rate of 4/5, p_4 seems to be a checksum of the whitened nibble. In order to verify this specificity of this coding rate, we sent random data and since there is only 16 possible configurations, we extract them from the received codewords and plotted them on figure 2.13. As expected, the last columns correspond indeed to the parity bit of the whitened nibble, meaning that the last column is calculated after that the whitening occurs.

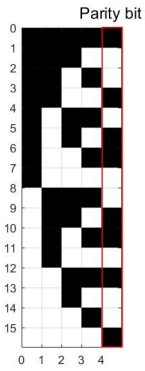

Figure 2.13: Exhaustive list of whitened codewords

Generator matrix To generate the codewords for the coding rates 4/6 to 4/8 we can simply do the following multiplication and keep only the firsts $\frac{4}{\text{CR}}$ bits:

$$\begin{bmatrix} d_3 & d_2 & d_1 & d_0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \bmod 2 = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 & p_0 & p_1 & p_2 & p_3 \end{bmatrix}$$

Parity check matrix For the error detection and correction, there are multiple cases to consider:

The cases of a coding rate of 4/8 and 4/7 offer both the possibility to correct one wrong bit. However with the additional parity bit p_3 present for the 4/8 coding rate, we can avoid to swap a correct bit in the case of an even number of wrong bits. This come from the fact that:

$$p_3 = d_0 \oplus d_2 \oplus d_3 = d_0 \oplus d_1 \oplus d_2 \oplus d_3 \oplus p_0 \oplus p_1 \oplus p_2 \tag{2.9}$$

Which can act like a parity checksum. Once we know if there is an odd number of mistakes, most probably only 1, we can drop p_3 and treat both coding rates exactly the same way. One property that can be achieved with Hamming coding is that the syndrome that comes out of the multiplication of the codeword with the parity check matrix can give the position of the bit to swap in the codeword. To achieve that we have to reorder the bits inside the codeword to match the syndrome output. Here again we have to choose an arbitrary ordering of the parity bit position in the syndrome calculation, in our case we choose to use the following syndrome:

$$S = \begin{bmatrix} p_2 \oplus d_0 \oplus d_1 \oplus d_3 & p_1 \oplus d_1 \oplus d_2 \oplus d_3 & p_0 \oplus d_0 \oplus d_1 \oplus d_2 \end{bmatrix} \tag{2.10}$$

which will require to change the codeword bit order to the following:

$$\begin{bmatrix} p_0 & p_1 & d_2 & p_2 & d_0 & d_3 & d_1 \end{bmatrix} \tag{2.11}$$

And finally the parity check matrix to use is:

$$H = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{2.12}$$

In the case of a coding rate of 4/6, only the detection of up to two errors can be performed. The error detection can be realized by using the following syndrome:

$$S = \begin{bmatrix} p_1 \oplus d_1 \oplus d_2 \oplus d_3 & p_0 \oplus d_0 \oplus d_1 \oplus d_2 \end{bmatrix} \tag{2.13}$$

The syndrome can be calculated by the following equation:

$$\begin{bmatrix} d_0 & d_1 & d_2 & d_3 & p_0 & p_1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{2.14}$$

which will return a non-null syndrome in the case of a detected error.

Whitening order Having understood how the Hamming coding block works, we can go back to the whitening matrix previously found and understand why for bigger coding rates, only the first columns where used and why in the case of a coding rate of 4/5 the last columns was different. In fact the first four columns of the whitening matrix are going through the Hamming encoder but this wasn't an issue since:

hamming(Data) XOR hamming(Whitening) = hamming(Data XOR Whitening)

Which leads to the encoding order being: Whitening, Hamming encoding, Interleaving and Gray indexing. This order seems to be the correct one since there isn't any particular configuration requiring a special treatment somewhere else than inside each block separately.

2.4 Header

From the LoRa patent [6], we know that the header should always be encoded using the lowest coding rate (4/8) to make use of all the error correction capabilities of the hamming code. The objective is to increase the chance of decoding the header content correctly since crucial informations about the transmission are specified in it. Moreover, heaving a defined coding rate for the header is mandatory because the coding rate of the message is not known in advance by the receiver. The patent also describe what could be included in the header: "According to a possible implementation of the invention, the header fields are coding rate, enable of payload CRC, payload length, CRC, a burst mode indication, a compressed mode indication, some reserved bits, and a ranging bit". Until now we have only received packets in implicit header mode and since we have already seen the presence of an implicit header, we need to know where is added the explicit header.

The figure 2.14 present the 16 first codeword of a difference message obtained by sending the packet varying only the coding rate used. The objective is to see which part of the header didn't stay constant, meaning that the coding rate information and by extension the explicit header is present at that position.

We can see that the differences are situated in the firsts height symbols and in the 16th one. The last row is different because, for the bigger coding rate, it is the parity bits p_3 of each interleaved codewords and for the smaller coding rate, it is the data bits d_0, meaning that it is not cause by the header position. Because of that we can deduce that the explicit header is situated in the first height symbols and we will only have to examine the first height symbols to extract the explicit header data.

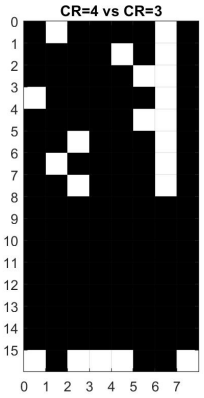

Figure 2.14: Difference message between two coding rates in explicit header mode

In order to find the position of each element in the header, we sent packet with a lot of different configuration, varying coding rate, payload CRC presence and payload length. We then take all the received headers and create the matrix shown on figure 2.15. In this matrix, all bits that stayed to zero independently of the parameters aforementioned are in black, the ones that stayed at one are white and the others are grey. We can already see that something is particular with the lasts 2 columns and they require further investigation.

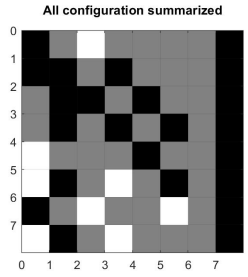

Figure 2.15: Summary matrix of many different headers

The figure 2.16 present the header received with a packet of 16 bytes, no payload CRC and a coding rate of 4/7. We can observe that the last column is as expected always zeros. For the columns marked in red, it appears that it contains the checksum of each row.

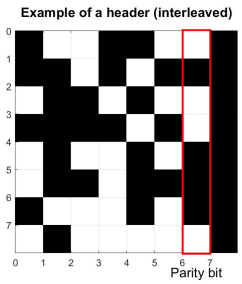

Figure 2.16: Header received for CR=4/7, Payload length=16B and CRC= OFF

Coding rate The figure 2.17 present the matrices created in the same manner as explained above. each of these matrices being created from header of packet using the same coding rate. We can finally find the bit that are constant (black or white) for each individual coding rate but different between two of them, if such bit exist they are the one containing the information about the coding rate.

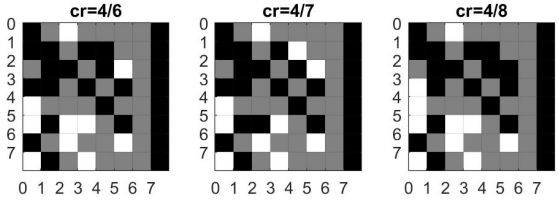

Figure 2.17: Summary matrices for 3 different coding rates

The figure 2.18 present the mask obtained by this method, on which the white squares represent the position of the bits indicating the coding rate. By applying this mask on top of the headers of figure 2.17, meaning to look only at the bits having the same position as the white ones in the mask, we get the three pattern presented alongside the mask. From our understanding of the interleaving method, we know that the data bits should be in the first four rows, the other ones being parity bits.

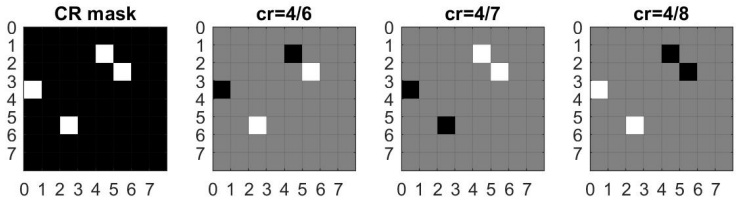

Figure 2.18: Mask and bit corresponding to the coding rate information

The coding rate should be encoded in the bits of rows two to four which are in our case :

- 0 1 0 for CR = 4/6
- 1 1 0 for CR = 4/7
- 0 0 1 for CR = 4/8

The bits recovered correspond to the LSB first version of the way the coding rate is specified to the chip. As explained in the chip datasheet [8] the coding rate has to be specified by the number n as CR=$\frac{4}{4+n}$. This also explain why only 3 bits are required since n can be between 1 and 4.
Lastly, we can conclude that the header is not whitened, which can also be deduced by comparing the whitening sequence obtained for the implicit and explicit header mode (fig. 2.19). By looking at the patterns we clearly see that for the explicit header the whitening sequence is shifted by 5 rows, meaning that the whitening starts after the explicit header which is also 5 codewords long.

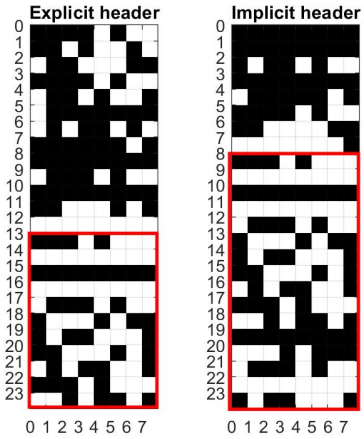

Figure 2.19: Deinterleaved whitening sequence

We can also note that the lasts two columns are added to the header bloc after that the data has been interleaved. And in order to account for those two added bits in the end of each symbols, knowing that total number of bits per symbol should be equal to the spreading factor, the interleaving should be performed in a modified way. This modification only consist of interleaving SF-2 codeword together instead of SF in the normal case, what can be achieved by simply replacing the term term SF in the equation 2.4 by SF-2. The advantage of decreasing the number of values that can modulate the LoRa symbols is to provide a greater distance between the candidate values during the demodulation.

CRC presence The same way as for the coding rate, we recover the mask of the CRC presence indicator position. The result is shown on the figure 2.20 and shows that only one bit indicate the MAC CRC presence which was expected. Now that we are aware of the special interleaving that is used, we can look at the deinterleaved version of the header to better see what is contain in the rest of it.

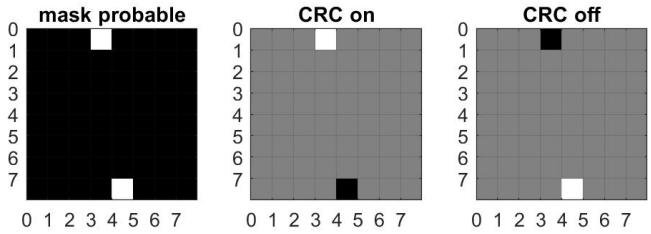

Figure 2.20: Mask and bit corresponding to the MAC CRC presence

Payload length From the chip datasheet [8] we know that the maximal payload length is 255 Bytes, meaning that the payload length information would require 16 bits (with the parity bits included) of the header. By looking at the mask found through the same method as before, shown on the figure 2.21, it appears that the two first rows contains the payload length, since they are white in the calculated mask. After looking to the value given by the payload length in the header, we get:

- 16 Bytes: 0b0001 0100 $\Rightarrow$ 20
- 17 Bytes: 0b0001 0101 $\Rightarrow$ 21
- 127 Bytes: 0b1000 0011 $\Rightarrow$ 131

An offset of 4 appears, and is due to the addition of the implicit header which seems to be added in the beginning of our payload and which is indeed 4 Bytes long.

Since we know that the smaller spreading factor is 7 and that the header block uses an effective interleaving spreading factor of SF-2, 5 in the worst case, we know that we there still is two codewords of the header that are not known.

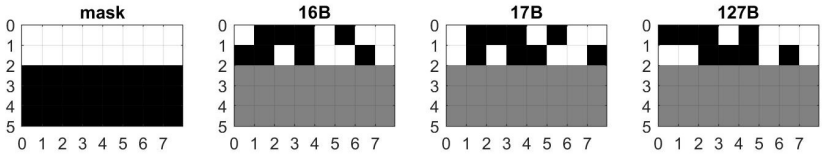

Figure 2.21: Deinterleaved mask and bits corresponding to the Payload length

Header CRC The patent [6] also mentioned the presence of a CRC in the header. We can now look only at the four first columns since the other ones are parity bits. By varying each parameters, we observe that the last two nibbles change, highly encouraging us to think that the CRC is present there. But as show in the figure 2.22, only the 5 LSBs seemed to change depending on the parameters. We weren't able to make the 3 other bits change to non-zero, even by changing the values described as potentially in the header which where modifiable on the chip we used (RFM95W).

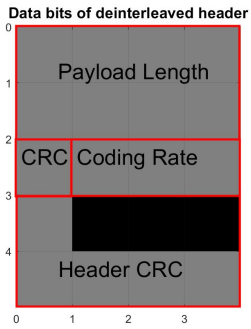

Figure 2.22: Header structure

Since the patent talk about a CRC, we considered it as true and started from there. To calculate the CRC of a bitstream, there are mainly 3 parameters to know: The divisor polynomial, the initial CRC value and the final XORing value [9].
We firstly want test if the initial and final XORing values are non-zero. This can be easily verified by sending an all-zeros messages and verifying if the corresponding CRC is also all-zeros. Unfortunately we can't do that in our case since we have to send at least 4 bytes of implicit header, leading to at least one non-zero value in the explicit header. An alternative is to send three headers, namely H1, H2 and H3. H1 and H2 can be any two different headers and H3 should be H1 $\oplus$ H2. If the CRC of H3 is not equal to the (CRC of H1) $\oplus$ (CRC of H2), those two parameters are non all-zeros. This verification is based on the superposition principle that holds for homogeneous CRCs (CRC without initial value and final XOR).

The figure 2.23 present three different headers received. The third was constructed to be the XOR of the two previous one and by looking at the CRC obtained (under the red line), we can deduce that an homogeneous CRC is used

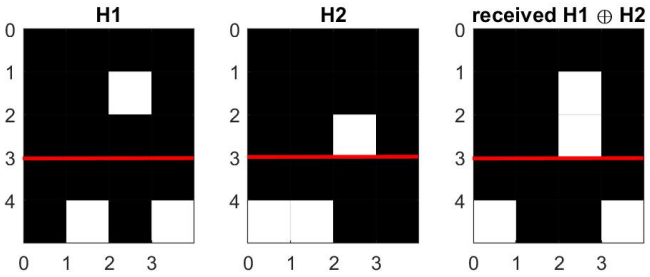

Figure 2.23: Header received with 3 different configurations

Only the polynomial used was still unknown which can easily be found knowing the CRC calculation algorithm. If we take two messages in which only one bit is 1 and that the position of those ones is adjacent, their CRCs should differ only by a shift-XOR operation. Meaning that if we take the two CRCs, shift one of them by one (the direction being another parameter of the CRC calculation but could have only two values) and XOR them together we will either get all 0 if the LSB of the non-shifted one was 0. Indeed if the lsb was zero, the shift-XOR operation wouldn't have happened. In the case of a 1 as LSB, we would recover the polynomial used.
Unfortunately in our case the result wasn't what was expected and no polynomial seemed to get out of this procedure. And by looking at figure 2.24 the CRC is not a shifted version of an adjacent one when this

adjacent one end or begin with a zero (we should look at both possibilities since we don't know if the CRC is reversed or not in the end).

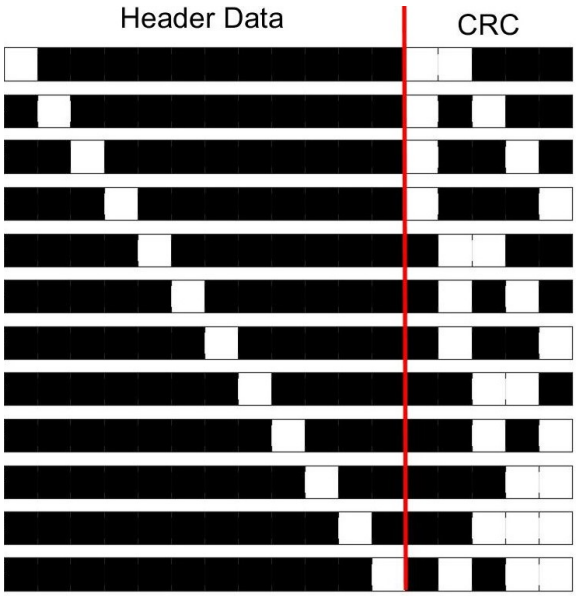

Figure 2.24: 'Impulse response' of the CRC

This lead us to conclude that it isn't a conventional CRC but since it is linear as CRC(x) $\oplus$ CRC(y) = CRC(x $\oplus$ y), and we have what we could consider as the impulse response of each of the 12 bits, we can easily calculate the CRC value thanks to the following equation:

$$\begin{bmatrix} h_{11} & h_{10} & \dots & h_0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix} \bmod 2 = \begin{bmatrix} crc_4 & crc_3 & crc_2 & crc_1 & crc_0 \end{bmatrix} \tag{2.15}$$

Implicit header After observing that this implicit header was not at all data dependant and treated exactly as the rest of the payload, we started to doubt that the radio was handling it. By looking at the library we used to program the chip, it appears that during the sending operation 4 bytes are sent to the chip before our payload and are namely:

- To
- From
- Id
- Flags

meaning that this implicit header is not part of LoRa PHY.

2.5 Payload CRC

The only still unknown part of the received packets is the payload CRC which has a length of 2 Bytes, as specified by the chip datasheet [8]. We will start by considering that it is a conventional CRC and perform in the same way as for the header CRC. By sending the same message in implicit and explicit mode, we observed that the CRC calculation in only performed on the payload.

In order to find the polynomial used, we have to generate what we will call difference message in order to make abstraction of the initial and XOR values. To achieve that, we will require more theory about CRC [10] which imply that a CRC C can be expressed as follows $C = T^n I \oplus D \oplus F$ with I being the initial value, F the final XOR, n the message length, D the CRC if I and F where null and $T^i x$ the result of applying i shift-xor steps to register contents x with zero data. And if we XOR two messages, we get:

$$C1 \oplus C2 = (T^n I \oplus D_1 \oplus F) \oplus (T^n I \oplus D_2 \oplus F) = D1 \oplus D2 \tag{2.16}$$

Which means that by XORing two messages and their CRCs together, we get a new message and CRC. But this new CRC will be the one corresponding to the new message calculated using the original polynomial but no initial value and final XOR.

By generating these difference messages in a way that the have only one 1 in them and that these ones are adjacent in each message, we get the CRCs shown on the figure 2.25. We can recognize the CRC characteristic pattern where two adjacent CRCs are just a shifted version of one another in the case of the first one ending with a zero. We only need to shift each line by one and XOR it with the next one to either get zeros or the polynomial used. In our case we obtain 0x8810 which correspond to the polynomial named CCITT-16, being $x^{16} + x^{12} + x^5 + 1$.

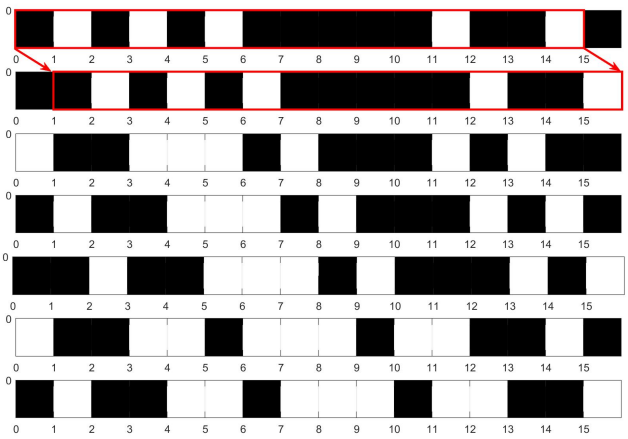

Figure 2.25: CRCs of adjacent one-hot difference messages

By sending a message containing only zeros we can easily test if the initial value and final XOR are null since in such a case, the CRC corresponding will be all zero. The result of this test gave us not all-zeros CRC but after taking a closer look we found out that it was the whitening sequence that appears, meaning that the CRC should not have been de-whitened. By not dewhitening the CRC we received every time all-zeros CRCs for different length of all-zero messages.

Unfortunately, the CRC calculation wasn't working with message that weren't all-zeros ones. By sending one hot messages, and comparing their CRCs to the one we generate using our polynomial, we observed that we get the correct value when we shift the one hot position by 16. Since adding zeros in the beginning of a payload should have no impact on the CRC value, it means that the CRC is calculated without using the lasts 2 Bytes of the payload. Which leads us to ask what comes of those two bytes which are not taken into account for the CRC calculation, and we observed that they are used as the final XOR value. Using this method, we are now able to calculate the payload CRC and use it to verify packet integrity.

Bibliography

[1] G. Jaume and M. Cotting, "Lora phy implementation and analysis," 2016.

[2] M. Joffrey, "Usrp implementation of lora lpwan physical layer," 2017.

[3] J. A. M. Galicia, "Reverse engineering lora header and crc," 2018.

[4] S. Corporation, *AN1200.22 LoRa™ Modulation Basics*, 2015.

[5] R. Ghanaatian, O. Afisiadis, M. Cotting, and A. Burg, "Lora digital receiver analysis and implementation," 2018.

[6] O. Seller and N. Sornin, "Low power long range transmitter," Patent EP 2 763 321 A1, 2014.

[7] P. Robyns, P. Quax, W. Lamotte, and W. Thenaers, "A multi-channel software decoder for the lora modulation scheme," 2017.

[8] *RFM95/96/97/98(W) - Low Power Long Range Transceiver Module.*

[9] R. N. Williams, "A painless guide to crc error detection algorithms," 1993.

[10] G. Ewing, "Reverse-engineering a crc algorithm," 2010.