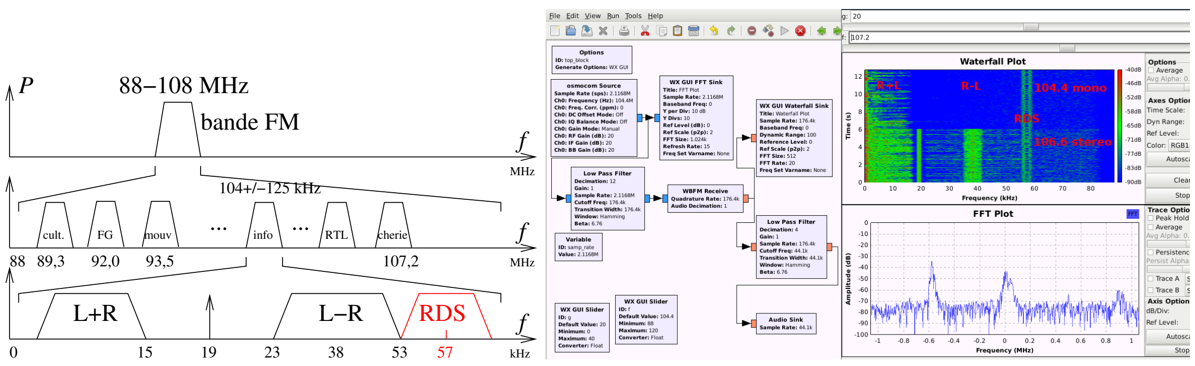
Radio Data System (RDS) – investigation of the digital channel used by commercial FM broadcast stations, introduction to error correcting codes

J.-M Friedt, April 2, 2017

RDS – Radio Data System – is the digital communication protocol used by commercial FM station in the 88–108 MHz band to inform the listener of information such as the name of the station being broadcast, free text such as the title of the current program or music, as well as time or the kind of program being broadcast. We consider addressing the full decoding scheme, from recording the analog signal with a DVB-T dongle used as a general purpose radiofrequency receiver, to understanding the various demodulation and decoding schemes, to finally conclude with an exploration of means of detecting and correcting errors.

1 Introduction

The commercial FM broadcast band, between 88 and 108 MHz, is divided to allocate 200 kHz wide bands to each station (Fig. 1, left). Each station sub-divides its spectrum fraction in three sub-segments: sound, with first the left plus right audio signals, then the stereo signal including left minus right audio signals (so that a mono-receiver can still receive a stereo program), and finally a digital signal – RDS (*Radio Data System*, Fig. 1 right) – which includes information such as the name of the station (Fig. 2), or free text such as the title of the program or the music. A receiver is noticed that a broadcast is stereo by a pilot – a continuous periodic signal – at 19 kHz after demodulation of the FM signal. The digital sub-carrier is at 57 kHz, generated as three times the pilot signal if the broadcast is stereo, an assumption we will not consider during our processing in which we will aim at reproducing a local copy of the 57 kHz subcarrier. The digital signal bandwidth is about 5 kHz.



```
Options
ID: top_block
Generate Options: WX GUI

osmocom Source
Sample Rate (sps): 2.1168M
Ch0: Frequency (Hz): 104.4M
Ch0: Freq. Corr. (ppm): 0
Ch0: DC Offset Mode: Off
Ch0: IQ Balance Mode: Off
Ch0: Gain Mode: Manual
Ch0: RF Gain (dB): 20
Ch0: IF Gain (dB): 20
Ch0: BB Gain (dB): 20

WX GUI FFT Sink
Title: FFT Plot
Sample Rate: 2.1168M
Baseband Freq: 0
Y per Div: 10 dB
Y Divs: 10
Ref Level (dB): 0
Ref Scale (p2p): 2
FFT Size: 1.024k
Refresh Rate: 15
Freq Set Varname: None

Low Pass Filter
Decimation: 12
Gain: 1
Sample Rate: 2.1168M
Cutoff Freq: 176.4k
Transition Width: 176.4k
Window: Hamming
Beta: 6.76

WBFM Receive
Quadrature Rate: 176.4k
Audio Decimation: 1

WX GUI Waterfall Sink
Title: Waterfall Plot
Sample Rate: 176.4k
Baseband Freq: 0
Dynamic Range: 100
Reference Level: 0
Ref Scale (p2p): 2
FFT Size: 512
FFT Rate: 20
Freq Set Varname: None

Low Pass Filter
Decimation: 4
Gain: 1
Sample Rate: 176.4k
Cutoff Freq: 44.1k
Transition Width: 44.1k
Window: Hamming
Beta: 6.76

Variable
ID: samp_rate
Value: 2.1168M

WX GUI Slider
ID: g
Default Value: 20
Minimum: 0
Maximum: 40
Converter: Float

WX GUI Slider
ID: f
Default Value: 104.4
Minimum: 88
Maximum: 120
Converter: Float

Audio Sink
Sample Rate: 44.1k
```

Figure 1: Left: the frequency band of interest – RDS – is located at 57 kHz in each FM station sub-band: it is thus accessible by frequency translation, to bring the digital signal to baseband (around the 0-frequency), *after* demodulation of the FM signal. Right: the various signals broadcast by a commercial FM station are visible on a *waterfall* following the WBFM demodulator. We have switched stations halfway during the acquisition between a stereo and a mono station, both emitting an RDS identification signal.

A reader who only wants to read the content of the digital information might be content with using one of the readily available integrated chips doing the job, such as the ST TDA7330 [1] or TDA7478 (single chip RDS decoders), or the integrated FM receiver RDS5807 by RDA microelectronics. [2] uses similarly a TEA5764 to synchronize a sensor network, despite all these references appearing now as obsolete and no longer provided by major suppliers. Doing so, we would not have learned anything about the encoding scheme, the kind of information transmitted but most significantly the means for recovering the bits corrupted during transmission. All these concepts will be tackled during software decoding of the sentences, analyzing step by step the path to convert a frequency modulated signal (commercial FM broadcast) to understandable sentences. As usual, such an understanding help assessing malicious use of the protocol or how to divert it from its original purpose [1, 3, 4]. All digital signal processing prototyping will be performed on GNURadio [1] used for acquiring the signal, followed by an implementation of the processing and decoding algorithms with GNU/Octave [2] with the aim of getting back to basics, without using high level libraries such as the communication toolbox [3] which would hinder the detailed understanding of the processing flow.

[1] gnuradio.org
[2] www.gnu.org/software/octave/
[3] octave.sourceforge.io/communications/overview.html

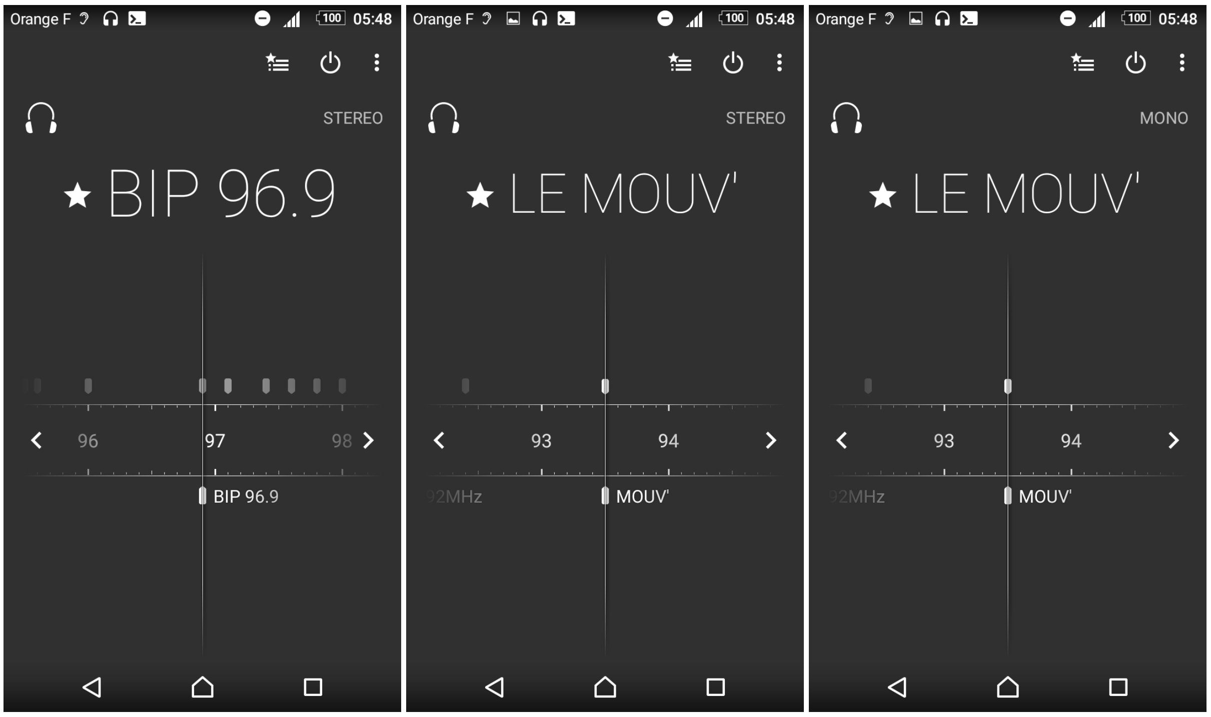


Figure 2: Reception of various FM commercial broadcast, with the display of the name of each station as transmitted by RDS. Notice that Le Mouv' is sometimes identified as stereo, sometimes as mono, without preventing the display of its identifier.

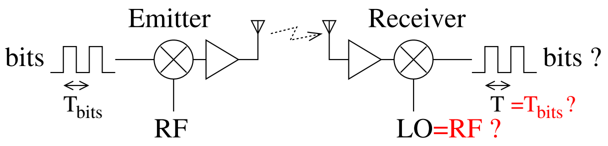


Figure 3: Problem of synchronization with the remote oscillator carrying the information and clocking the databit rate. Decoding will only become possible once the two oscillators on the receiver – LO (Local Oscillator) and digital sampling rate – are locked on their remote counterpart on the emitter.

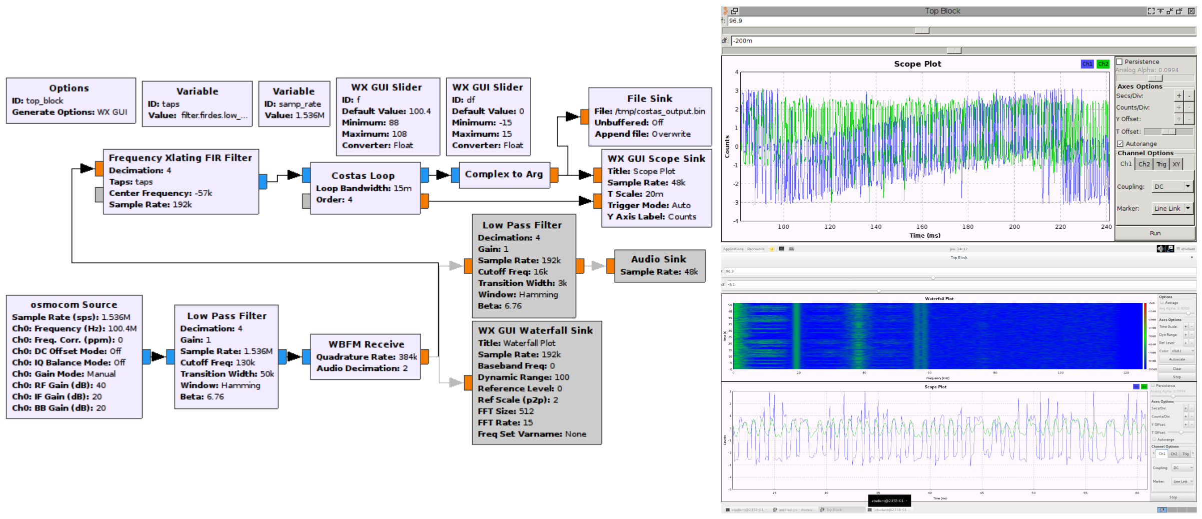
Decoding a digital communication on a radiofrequency link always requires solving the issues of synchronizing remote oscillators, namely the radiofrequency carrier on which the information is transmitted, and the data rate. The receiver translates the radiofrequency signal to baseband by mixing the received signal with a local oscillator LO (Fig. 3). The phase which encodes the information on the carrier is only usable once a copy of the emitter carrier – RF – is locally generated: LO is controlled by RF using schemes described later in the text (section 2). Once the bits are visible at the output of the demodulation, the digital information is periodically sampled to extract each bit value and assemble sentences. Here again, the decoding rate has no reason of being synchronous with the encoding rate: even if the nominal bitrate is known, any offset between the oscillator generating the data stream with the local oscillator will necessarily end up generating a loss of synchronization with the receiver. Here again, some feedback control will be necessary, as described in section B.

In the GNURadio acquisition scheme depicted in Fig. 4 which aims at demodulating an FM signal, extract the digital sub-carrier, and store the resulting stream in a binary file for post-processing, two characteristics define the quality of the demodulation and the computational power needed to implement the algorithm: the cutoff frequency to isolate the frequency band of interest using successive filterings, and the decimation factors (only taking one in every N sample for a decimation factor of N) to reduce the data rate. A DVB-T receiver based on a RTL2832U analog to digital converter can only operate with sampling rates between 1.5 and 2.4 MHz, which are much more than the spectral bandwidth occupied by a single FM broadcast station. Our first task hence consists in decimating this data stream to only keep about 200 ksamples/s, hence meeting the requirements of the spectral bandwidth allocated to each FM station. However, decimating is only possible after attenuating the signal of neighboring stations located more than 100 kHz away from the band we are interested in, otherwise aliasing [4] would bring their signals into baseband during decimation.

Hence, we start with a low-pass filter which isolates the frequency range of interest, and *only then* decimate enough to reduce the data rate to about 200 ksamples/s. The other advantage of decimating is to allow for the

[4]aliasing is a consequence of discrete periodic sampling at times $1/f_s$, which assumes that the spectrum between $-f_s/2$ and $f_s/2$ repeats every f_s. Thus, decimating by a factor N would bring all spectral components between $f_s/(2N)$ and $f_s/2$ to the band between $-f_s/(2N)$ and $f_s/(2N)$ by translating the spectra by $f_s/(2N)$ steps. This effect is avoided by including prior to any decimation a low-pass filtering step which efficiently eliminates all signal components above $f_s/(2N)$.

transition width of subsequent filters to be sharper with a same number of coefficients: indeed, the transition width of a filter is about the sampling rate divided by the number of coefficients in the filter. Hence, the computational load of a sharp filter at a high sampling rate is heavy, while the same result is achieved with a lighter load if a preliminary decimation was performed previously. The 200 ksamples/s we have just obtained include all the information encoded by the FM station we are considering: the WBFM block demodulates the signal and provides a data stream which can again be decimated if only the audio signal is of interest. However, we wish to decode RDS located on a subcarrier around 57 kHz so we cannot decimate yet, but must bring this signal of interest to baseband using a `Xlating FIR Filter` [5] before again decimating, the 200 ksamples/s being far too much bandwidth for the few kHz needed by the digital signal. Hence, the FIR (Finite Impulse Response) filter is designed to isolate the digital signal in a bandwidth of a few kHz and reject all the other spectral components prior to the decimation which yields a data rate consistent with that needed for digital mode decoding. Warning: the sampling rate defining the FIR filter must be the sampling rate at the input of the `Xlating FIR Filter`, which must include the decimation factor of the first low-pass filter and possibly of the WBFM demodulator. In our case, this filter (`taps` variable called in the field with the same name of the `Xlating FIR Filter`) is defined by the Python expression: `filter.firdes.low_pass_2(1, samp_rate/8, 2000, 500, 60)` to indicate that the low pass filter already performed a decimation by 8, and no decimation was applied by the WBFM demodulator, and 2 kHz cutoff frequency and a 500 Hz transition width. Following this processing step, we have isolated the RDS band which includes the data we wish to decode.


Options
ID: top_block
Generate Options: WX GUI

Variable
ID: taps
Value: filter.firdes.low_...

Variable
ID: samp_rate
Value: 1.536M

WX GUI Slider
ID: f
Default Value: 100.4
Minimum: 88
Maximum: 108
Converter: Float

WX GUI Slider
ID: df
Default Value: 0
Minimum: -15
Maximum: 15
Converter: Float

File Sink
File: /tmp/costas_output.bin
Unbuffered: Off
Append file: Overwrite

Frequency Xlating FIR Filter
Decimation: 4
Taps: taps
Center Frequency: -57k
Sample Rate: 192k

Costas Loop
Loop Bandwidth: 15m
Order: 4

Complex to Arg

WX GUI Scope Sink
Title: Scope Plot
Sample Rate: 48k
T Scale: 20m
Trigger Mode: Auto
Y Axis Label: Counts

Low Pass Filter
Decimation: 4
Gain: 1
Sample Rate: 192k
Cutoff Freq: 16k
Transition Width: 3k
Window: Hamming
Beta: 6.76

Audio Sink
Sample Rate: 48k

osmocom Source
Sample Rate (sps): 1.536M
Ch0: Frequency (Hz): 100.4M
Ch0: Freq. Corr. (ppm): 0
Ch0: DC Offset Mode: Off
Ch0: IQ Balance Mode: Off
Ch0: Gain Mode: Manual
Ch0: RF Gain (dB): 40
Ch0: IF Gain (dB): 20
Ch0: BB Gain (dB): 20

Low Pass Filter
Decimation: 4
Gain: 1
Sample Rate: 1.536M
Cutoff Freq: 130k
Transition Width: 50k
Window: Hamming
Beta: 6.76

WBFM Receive
Quadrature Rate: 384k
Audio Decimation: 2

WX GUI Waterfall Sink
Title: Waterfall Plot
Sample Rate: 192k
Baseband Freq: 0
Dynamic Range: 100
Reference Level: 0
Ref Scale (p2p): 2
FFT Size: 512
FFT Rate: 15
Freq Set Varname: None

Figure 4: Left: processing sequence of a signal recorded in the commercial FM band. The waterfall mode (bottom-right) clearly exhibits the 19 kHz pilot and the RDS sub-carrier around 57 kHz. Top-right: without carrier synchronization scheme (blue), the phase encoding the signal exhibits a linear drift with a slope equal to the offset between the nominal frequency of 57 kHz and the digital oscillator frequency. Using a Costas loop (green) eliminates such a frequency offset: the phase now exhibits usable bit states (bottom right, blue).

RDS is actually readily available to GNURadio users through the `gr_rds` module, initiated by Dimitrios Symeonidis [6] and currently maintained by Bastian Bloessl [7]. Copying this code might bring some insight, even validate the proper operation of the receiving hardware setup, but will not help in understanding the various steps needed to demodulate and decode the protocol, as will be addressed in this document. The various contributors to `gr_rds` do not seem to have considered useful to document in detail the underlying principles of their software in an associated publication, making the analysis of the source code a bit tedious, especially for the beginner which is our current status at this point of the protocol analysis. Indeed, we will meet several OSI layers before getting a readable message, and this clear separation of the abstraction layers in the technical documentation [6] remains confusing at first by postponing in the appendix the layer 2 while the main text addresses layers 1 and 3. Furthermore, all source codes found on the web are inspired from an implementation of the error correcting code as a linear feedback shift register, a natural solution for the electronics engineer but poorly suited to signal processing prototyping under GNU/Octave in which problems are naturally expressed as linear algebra matrix computations. We will hence describe an implementation

[5]we have introduced the `Xlating FIR Filter` as a block including a core signal processing function during demodulation, namely frequency translation, filtering and decimating [5]. As a reminder, the low pass filter aims at attenuating spectral components after mixing – frequency translation – above the sampling rate achieved following decimation: without this filter, all signal components above the new sampling rate would be brought to baseband by aliasing during the decimation and would prevent further processing.

[6]`https://bitbucket.org/azimout/gr-rds/`

[7]`https://github.com/bastibl/gr-rds`

that seems to be original to synchronize sentences and for correcting errors that we have not found readily implemented in the documentations we have read during this investigation, but whose compacity aims at helping understand underlying concepts tricky to address at the logic gate level.

The modulation scheme is described in a somewhat confusing way in the technical documents describing RDS [6], explaining that a carrier-less amplitude modulation [7] is generated by using two signal in phase opposition [6, section 1.4]. The alternative explanation, considering information being carried by a ±90° phase modulation, completely changes our demodulation strategy (see appendix A). While an amplitude modulation only requires a rough estimate of the carrier and a rectification/filtering with a bandwidth wide enough for any frequency offset of the emitter with respect to the receiver to be included, a phase modulation requires a strategy in which *a local copy of the unmodulated emitted carrier oscillator is recovered*, which will be the topic of the first part of this document.

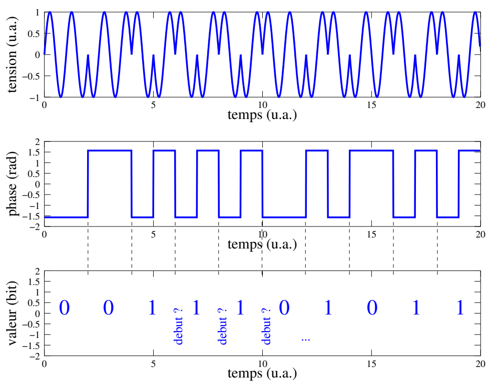


Figure 5: Processing sequence for decoding sentences: going from the middle to the bottom steps is consistent with Tab. 2 of [6] with a conversion meeting the definition of a differential Manchester encoding.

Having generated useful signal – phase of the carrier – representative of the successive bit values, we will group these bits into sentences (Fig. 5). Since bits are continuously transmitted, we must find a strategy for identifying the beginning of sentences in this continuous bit stream. Finally, having synchronized our receiver on the bit-stream, we will interpret the payload and observer that the results are consistent. We observe on Fig. 4, which consists in a GNURadio processing flow for demodulating the FM station, extract the 57 kHz sub-carrier and display the phase and magnitude, that the phase does not exhibit a visually appealing structure representative of a bit sequence. The spectrum of the phase information hints that the binary phase shift keying (BPSK [8, 9]) modulation with two states separated by 180° is the right approach: the spectrum is spread around 1200 Hz by the phase modulation, but the fine peak at 2375 Hz confirms that any non-linear process which has produced the square of the incoming signal generates a spectrally pure output, as would be expected from a BPSK modulation.

2 Translation and reproducing the carrier

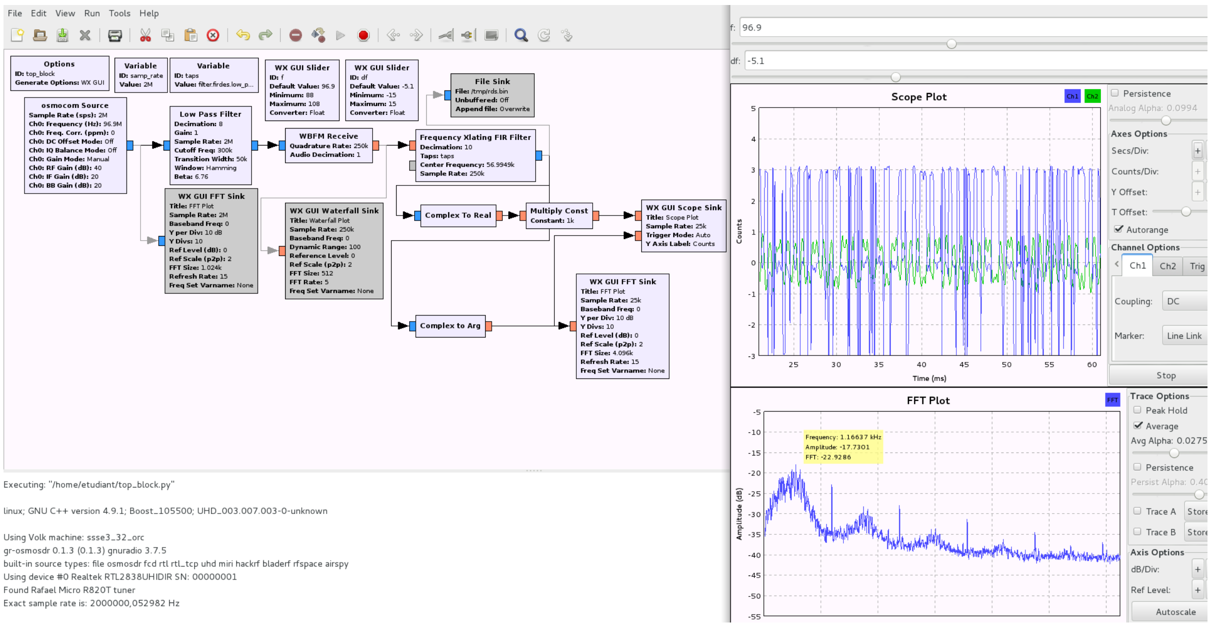
We have already seen, during our investigation of GPS [9], that demodulating BPSK which is characterized by two phase states with 0 and 180° values to represent the two bit states 0 and 1 for example, is achieved by reproducing a local copy of the emitter unmodulated carrier. In order to achieve such a feat, we had considered various means of extracting a phase from the complex generated from the I and Q coefficient using function insensitive to 180° phase rotations (`atan` in GNU/Octave which does not account for the quadrant in which I and Q individually are located but is only concerned with the Q/I ratio, as opposed to `atan2` which exploits each component individually, allowing it to recover a phase including 180° rotation). An alternative approach was to raise the signal to its square (multiplying it by itself) in order to remove the phase modulation, since squaring an harmonic signal creates the double of its argument, and $2 \times 180° = 360 = 0[360]$ (Fig. 6, bottom right). The result, with a double frequency than that of the sub-carrier, generates a local copy of the emitted carrier prior to its modulation by feeding a counter which divides by two the frequency. This latter method is implemented in the signal processing block named Costas loop [9], which outputs on the one hand the signal corrected from any offset between the emitter oscillator and the local receiver oscillator, and on the other hand an estimate of this error.

Our processing strategy is thus summarized as:

1. transpose the signal located around 57 kHz at the output of the FM demodulator to bring the digital information into baseband, for example using the GNURadio `Xlating FIR Filter`, using a free running local oscillator. While we usually have to experiment with the sign of the transposition frequency when handling complex input signals to bring the output to baseband, the problem exceptionally does not happen in this case in which the FM demodulation output is a real signal, so that the magnitude of the Fourier transform is even. The information we are interested in around +57 kHz also exists around -57 kHz: both solutions are acceptable and yield the same result,

2. extract the phase of the resulting signal, phase which includes two components: the information encoded in the BPSK encoding scheme at ±90°, and the linear drift due to the offset between emitter and receiver frequencies Δf,

3. feed a Costas loop with this signal so it estimates Δf and compensates for it.



```
Options
ID: top_block
Generate Options: WX GUI

Variable
ID: samp_rate
Value: 2M

Variable
ID: taps
Value: filter.firdes.low_p...

WX GUI Slider
ID: f
Default Value: 96.9
Minimum: 88
Maximum: 108
Converter: Float

WX GUI Slider
ID: df
Default Value: -5.1
Minimum: -15
Maximum: 15
Converter: Float

File Sink
File: /tmp/rds.bin
Unbuffered: Off
Append file: Overwrite

osmocom Source
Sample Rate (sps): 2M
Ch0: Frequency (Hz): 96.9M
Ch0: Freq. Corr. (ppm): 0
Ch0: DC Offset Mode: Off
Ch0: IQ Balance Mode: Off
Ch0: Gain Mode: Manual
Ch0: RF Gain (dB): 40
Ch0: IF Gain (dB): 20
Ch0: BB Gain (dB): 20

Low Pass Filter
Decimation: 8
Gain: 1
Sample Rate: 2M
Cutoff Freq: 300k
Transition Width: 50k
Window: Hamming
Beta: 6.76

WBFM Receive
Quadrature Rate: 250k
Audio Decimation: 1

Frequency Xlating FIR Filter
Decimation: 10
Taps: taps
Center Frequency: 56.9949k
Sample Rate: 250k

WX GUI FFT Sink
Title: FFT Plot
Sample Rate: 2M
Baseband Freq: 0
Y per Div: 10 dB
Y Divs: 10
Ref Level (dB): 0
Ref Scale (p2p): 2
FFT Size: 1.024k
Refresh Rate: 15
Freq Set Varname: None

WX GUI Waterfall Sink
Title: Waterfall Plot
Sample Rate: 250k
Baseband Freq: 0
Dynamic Range: 100
Reference Level: 0
Ref Scale (p2p): 2
FFT Size: 512
FFT Rate: 5
Freq Set Varname: None

Complex To Real

Multiply Const
Constant: 1k

WX GUI Scope Sink
Title: Scope Plot
Sample Rate: 25k
Trigger Mode: Auto
Y Axis Label: Counts

Complex to Arg

WX GUI FFT Sink
Title: FFT Plot
Sample Rate: 25k
Baseband Freq: 0
Y per Div: 10 dB
Y Divs: 10
Ref Level (dB): 0
Ref Scale (p2p): 2
FFT Size: 4.096k
Refresh Rate: 15
Freq Set Varname: None
```

```
Executing: "/home/etudiant/top_block.py"

linux; GNU C++ version 4.9.1; Boost_105500; UHD_003.007.003-0-unknown

Using Volk machine: ssse3_32_orc
gr-osmosdr 0.1.3 (0.1.3) gnuradio 3.7.5
built-in source types: file osmosdr fcd rtl rtl_tcp uhd miri hackrf bladerf rfspace airspy
Using device #0 Realtek RTL2838UHIDIR SN: 00000001
Found Rafael Micro R820T tuner
Exact sample rate is: 2000000,052982 Hz
```

Figure 6: Left: FM reception, followed by the extraction of the RDS sub-band. Lacking a carrier synchronization strategy, the phase varies too quickly to look like a digital signal.

The digital modulation is at a nominal rate of 1187.5 bits/s, and we shall see later that the encoding scheme is differential which hence requires a bandwidth of at least $1187.5 \times 2 = 2375$ Hz, defining the width of the low-pass filter of the `Xlating FIR Filter` as well as its decimation factor. We aim at having at least 5 samples per period, or at least 11875 samples/s.

3 From baseband signal to bits

We now assume to have recovered a stream representative of the digital signal. We wish to extract a bit sequence from phase values. We initially learn [6, section 1.6] that the signal is encoded with a differential strategy (reminiscent of a differential Manchester [8]), encoding) as confirmed by our observation that the phase variation rate is twice the expected bit rate of 1187.5 Hz. We will thus threshold the phase after removing the mean value – considering that a negative value becomes null, and otherwise assigned to a bit value of 1 (*bit slicer* in GNURadio), and once the threshold value is obtained, we apply the conditions that two equal adjacent values become a bit equal to 0 while if a transition is observed, the resulting bit is assigned a value of 1. The two subtleties here lie on the one hand in mistakenly synchronizing on the wrong transition when searching for the half-bit used to start the analysis – in such a case all adjacent pairs exhibit a transition (as would be expected from a Manchester encoding) – and on the other hand synchronize the bit rate if the data rate is not locked on our local oscillator. We search for the first transition (`debut`), then move forward in the data stream by considering the quarter period and three-quarters (second state) after the transition. By analyzing whether these two states are equal or different (`s1` and `s2`), we deduce `so` the output bit state, as an exclusive OR (XOR) of these two bit-states, which can also be expressed as a sum modulo 2, a statement more natural for GNU/Octave programmers.

```
fe=24000;          % sampling rate -- cf gnuradio sink
bitlength=fe/1187.5  % 1 bit = 2 transitions
bitsur2=bitlength/2;
bitsur4=bitlength/4; % half transition width
r=read_complex_binary('costas_output100p4_24kHz.bin');% Virgin

p=angle(r);
p=conv(p,ones(4,1)/4);p=p(2:end-2);  % normalized low pass filter
p=p(2000:end);                       % time needed for costas to lock
p=p-mean(p);
k=find(diff(p(11:1000))>0.5);debut=k(1)+10; % first transition index
s=(p>0);s=s-mean(s);                 % binary slicer
l=debut;
for k=1:length(s)/bitlength-bitlength
```

[8]www.mathworks.com/examples/simulink-communications/mw/rtlsdrradio_product-RBDSSimulinkExample-rbds-rds-and-radiotext-plus-rt-fm-receiver

```
  s1(k)=s(floor(l+bitsur4));      % first state
  s2(k)=s(floor(l+bitsur2+bitsur4)); % 2nd state 1/2 a period later
  l=l+bitlength;              % move fwd one period
  transition=abs(diff(s(floor(l-2):floor(l+1))));
  [val,pos]=max(transition);
  if (val>0)               % synchronization attempt
    if (pos==3) l=l+1;end       % we are a bit early
    if (pos==1) l=l-1;end       % we are a bit late
  end
end
s1=(s1>0); s2=(s2>0);
so=mod(s1+s2,2);              % 2 bits -> 1 state
```

The GNURadio processing sequence provides a signal synchronous to the radiofrequency carrier, but does not address the issues of synchronizing the datarate (time interval between two bits). In the example provided above, a naive approach consists in checking whether a transition from one bit state to another occurs one sample period before or after the expected moment, and if such an offset occurs, to shift the counter which is incremented along the sentence (variable `l`). A more rigorous, but more complex, solution is discussed in appendix B, using the appropriate GNURadio signal processing block provided by `MPSK Decoder` or `Clock Recovery MM`, as presented in `http://gnuradio.4.n7.nabble.com/Clock-Recovery-MM-documentation-td55117.html`. Although this digital datastream synchronization was not used while writing this article, hence requiring the investigations on error correcting codes to recover erroneous bits as identified while decoding the sentences, its late implementation yielded excessively decoding efficiency as demonstrated at the end of appendix B.

4 From bits to sentences: synchronization

We have obtained a continuous stream of bits. However, a sentence starts and ends at some boundary that must be identified: we have for example seen when exploring the ACARS protocol [10] that each sentence starts with a preamble used for synchronizing the receiver with the bitrate of the emitter. In the case of RDS, finding the beginning of each sentence is described in the reference document [6, annexe C]: locating this issue that far in the appendix of the document makes its reading complex since it promotes attempting doomed decoding strategies [6, section 2] before being even sure that sentences are consistent. We learn between these two sections that each RDS sentence is made of 16-bit long data payload followed by a 10-bit long error correcting code (an alternative synchronization approach is given in C). The basic information is thus grouped in 26-bit long sentences, differentiated as four successive blocks named A, B, C, and D which follow to make one big message. The means of synchronization described in [11, chap.12] is thus summarized as

1. consider 26 adjacent bits in the acquired stream
2. compute their error correcting code, as described bellow, for these 16+10 bits, assuming they represent an A-block
3. if the last ten bits of the sequence indeed match the error correcting code, we might have reached a synchronization condition, which is validated by repeating the error correcting code computation on the next block (B, next 26 bits) and then D (26 adjacent bits separated by 78 positions from the current position). If all three error correcting code conditions are met, we have very probably met a synchronization condition, and hence identified where the first bit of a sentence is located. Block C has been omitted since it is split in two possible options, C and C' depending on the content of the message, with different syndromes in both cases, expanding the number of cases to be tested.
4. if the error correction code computation on the first 16 bits does not match the last 10 bit sequence, synchronization is not achieved: we move forward one bit of the recorded sequence and restart the whole procedure.

This sequence must end up converging when we meet a condition in which the last 10 bits of each block match the error correcting code of each preceding 16 bits. If that condition is never reached, then the bit decoding sequence from the raw analog data (previous step) must have failed, and we must reconsider the procedure used to convert the phase information to digital information. Successful decoding is the source of the time synchronization proposed in [2]. Once the beginning of a block identified, the payload interpretation of each block depends on the kind of message: while all A-block contain the same station identifier unique to each FM broadcaster (PI code, appendix C), the content of the other blocks depend on the nature of the transmitted data, as described in [6, section 3].

A key issue in the coming discussion lies with the error correcting code. All implementations we have found use a Linear Feedback Shift Register (LFSR) approach, a natural representation of the problem when configuring an FPGA or programming a microcontroller in assembly language [12], but poorly suited to GNU/Octave which best expresses problems in a matrix representation (see inset below). In order not be accused of plagiarism – all free code examples found on the internet for implementing the error correcting code look the same as found at `https://github.com/bastibl/gr-rds/blob/master/lib/decoder_impl.cc#L69-L80` (Fig. 7) – we demonstrate an error correcting code computation as a matrix computation as described in [6, section B.2.1]: a 26×10 matrix H provides the linear relationship between each payload bit sequence and its associated correcting code bit.

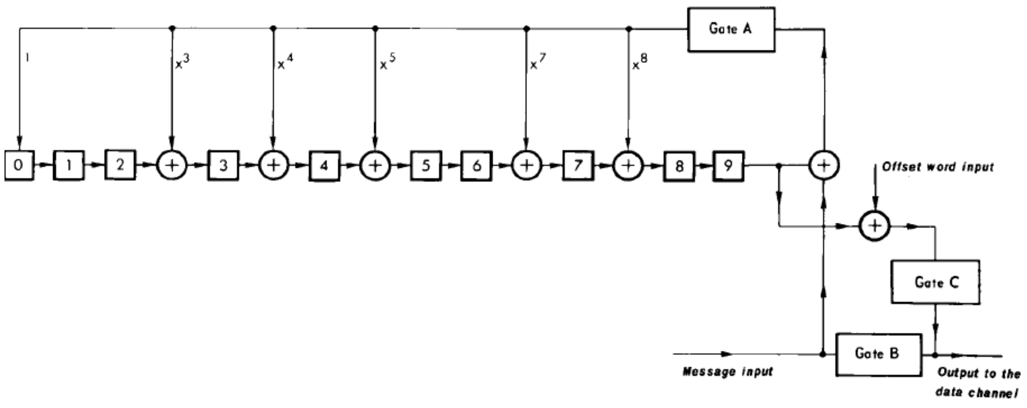


Figure 7: Linear feedback shift register implementation of the error correcting code. We discuss an alternative solution of pre-computing all possible states of the error correcting code in order to fill a matrix, better suited for expressing a problem to be solved by GNU/Octave. Each "⊕" symbol must be understood as a binary sum, or an exclusive OR. This chart was copied from [6, p.62].

Converting the polynomial expression to the matrix expression of the error correcting code might not seem obvious at first sight. The basic approach of linear algebra (matrix expression) is to decompose a problem on each element of a base and compute the global solution as a linear combination of each solution found for each element of the base. In our case, the base of the problem is represented as a message with one bit set to the value 1 at the nth position of the sentence and all other bits set to 0. We hence apply the error correcting code computation to each vector [0 ... 0 1 0 ... 0] and shift the position of the bit set to 1 in this vector, also expressed in a polynomial form as x^n. Computing the error correcting code is the remainder of the polynomial division [13] of x^n, $n \in [0..25]$ (the 26 possible positions of the bit in the emitted message) with the polynomial expression representing the BCH code [14, p.251] $x^{10} + x^8 + x^7 + x^5 + x^4 + x^3 + 1$ which is written under GNU/Octave as [1 0 1 1 0 1 1 1 0 0 1] (with the highest power to the left). As a reminder, the polynomial division is performed as a classical integer division, with the denominator multiplied by the appropriate power to cancel the term with the highest power of the numerator, and the remainder is computed as the subtraction of these two terms. The procedure continues until the remainder reaches a power lower than that of the denominator. For example $\underbrace{x^4+x^2+x+1}_{numeerator} = \underbrace{(x^2+1)}_{denominator} \times \underbrace{x^2}_{quotient} + \underbrace{(x+1)}_{remainder}$

The remainder of the polynomial division is found with GNU/Octave (and Matlab) as `[b,r]=deconv(N,D);` with `N` and `D` the vectors representing respectively the numerator and denominator polynomial functions, and `r` the remainder of the division we are interested in. In the previous example, `[d,r]=deconv([1 0 1 1 1],[1 0 1])` yields `d=1 0 0` and `r=0 0 0 1 1`. Hence, matrix G of [6] is found using the following GNU/Octave script:

```
vecteur=[1 zeros(1,10)];       % first element of the base: 1*x^{10}
polynome=[1 0 1 1 0 1 1 1 0 0 1]; % x^10+0+x^8+x^7+0+x^5+x^4+x^3+0+0+x
for k=1:16                  % position of the bit set to 1
 [a,b]=deconv(vecteur,polynome); % polynomial division
 mod(abs(b(end-9:end)),2)      % modulo x^10
 vecteur=[vecteur 0];          % next element in the base: add a 0
end
```

Since the identity matrix at the left of G places the least significant bit to the bottom, this script computes the right part of the lines of G from bottom to top.

```
% p.75 US, 64 EU
sA2=[1 1 1 1 0 1 1 0 0 0]; % A=  [0 0 1 1 1 1 1 1 0 0];
```

```
3  sB2=[1 1 1 1 0 1 0 1 0 0]; % B=  [0 1 1 0 0 1 1 0 0 0];
4  sC2=[1 0 0 1 0 1 1 1 0 0]; % C=  [0 1 0 1 1 0 1 0 0 0];
5  Cp= [1 1 0 1 0 1 0 0 0 0];
6  % sCp=[1 0 1 1 1 0 1 1 0 0];
7  sCp=[1 1 1 1 0 0 1 1 0 0]; % an495
8  D=  [0 1 1 0 1 1 0 1 0 0];
9  sD2=[1 0 0 1 0 1 1 0 0 0]; % inconsistent with an495
10 % sD2=[0 1 0 1 0 1 1 0 0 0]; % an495
11
12 % p.74
13 H=[
14  1 0 0 0 0 0 0 0 0 0 ; % this
15  0 1 0 0 0 0 0 0 0 0 ; % identity
16  0 0 1 0 0 0 0 0 0 0 ; % sub-
17  0 0 0 1 0 0 0 0 0 0 ; % matrix
18  0 0 0 0 1 0 0 0 0 0 ; % will be
19  0 0 0 0 0 1 0 0 0 0 ; % replaced
20  0 0 0 0 0 0 1 0 0 0 ; % with
21  0 0 0 0 0 0 0 1 0 0 ; % eye()
22  0 0 0 0 0 0 0 0 1 0 ; % in the
23  0 0 0 0 0 0 0 0 0 1 ; % next codes.
24  1 0 1 1 0 1 1 1 0 0 ;
25  0 1 0 1 1 0 1 1 1 0 ;
26  0 0 1 0 1 1 0 1 1 1 ;
27  1 0 1 0 0 0 0 1 1 1 ;
28  1 1 1 0 0 1 1 1 1 1 ;
29  1 1 0 0 0 1 0 0 1 1 ;
30  1 1 0 1 0 1 0 1 0 1 ;
31  1 1 0 1 1 1 0 1 1 0 ;
32  0 1 1 0 1 1 1 0 1 1 ;
33  1 0 0 0 0 0 0 0 0 1 ;
34  1 1 1 1 0 1 1 1 0 0 ;
35  0 1 1 1 1 0 1 1 1 0 ;
36  0 0 1 1 1 1 0 1 1 1 ;
37  1 0 1 0 1 0 0 1 1 1 ;
38  1 1 1 0 0 0 1 1 1 1 ;
39  1 1 0 0 0 1 1 0 1 1 ;
40 ];
41
42 texte="";
43 station="";
44 debut=0
45 so=(1-so);
46 for k=1:length(so)-104
47   data1=(so(k:k+25));          % A
48   data2=(so(k+26*1:k+25+26*1)); % B
49   data3=(so(k+26*2:k+25+26*2)); % C(')
50   data4=(so(k+26*3:k+25+26*3)); % D
51   HI1=mod(data1*H,2);
52   HI2=mod(data2*H,2);
53   HI3=mod(data3*H,2);
54   HI4=mod(data4*H,2);
55   pa=findstr((sA2),(HI1));
56   pb=findstr((sB2),(HI2));
57   pc=findstr((sC2),(HI3));
58   pcp=findstr((sCp),(HI3));
59   pd=findstr((sD2),(HI4));
60   if (!isempty(pa) && !isempty(pb) && !isempty(pd))
61     printf("synchronization\n");
62   end
63 end
```

This example, a bit lengthy due to the expression of the decoding matrix `H`, operates as follows:

1. for each consecutive 104 bit long sequence, we split four adjacent 26 bit long blocks (ll.47–50). Each block is assumed to be composed of a 16-bit payload followed by a 10-bit error correcting code to which a block identification value was added,

2. we compute the 10-bit syndrome of each 26 bit block by applying the matrix multiplication with `H` (ll. 51–54),

3. we check if the syndrome we have found matches the expected block syndrome (ll. 55–59). If this condition is met, a valid block has been found, and we assume we have met a synchronization condition if the syndromes of the four consecutive blocks have been found.

4. If a block syndrome does not match the expected error correcting code value, we assume we are not yet synchronized: we move forward one step in the acquired bit sequence to define a new 104-bit long dataset and restart the computation.

Converting G to H as found in [6] is also not trivial. While converting G to a shape of H that allows checking the integrity of a received message x by computing $H \cdot x = 0$ as described in [14, p.244] or [15, p.70] is only a matter of transposing the non-identity part of G, the expression of the control matrix found in [6, p.63] does not match this expression by putting the identity part of the matrix to the left of the decoding matrix. Means of converting one expression to the other is explained at `http://www.di-mgt.com.au/cgi-bin/matrix_stdform.cgi` which uses the algorithm described in [15, pp.52,71] to convert from one expression (deduced by transposing part of the encoding matrix) to the other (so called standard) by linear combinations and permutations of the lines and columns. Using the afore-mentioned site, converting H as found in [6, p.63]

```
Input:
1 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 0 1 1 0 0 1 1 1
0 1 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 0 1 1 0 0 1 1
0 0 1 0 0 0 0 0 0 0 1 0 1 1 1 0 0 0 1 0 1 1 1 1 1 0
0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1 1 1 0 0 0
0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1 1 1 0 0
0 0 0 0 0 1 0 0 0 0 1 0 1 0 1 1 1 1 1 0 1 0 1 0 0 1
0 0 0 0 0 0 1 0 0 0 1 1 0 0 1 0 0 0 1 0 1 1 0 0 1 1
0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 0 1 1 0 0 1 1 1 1 1 0
0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 0 1 1 0 0 1 1 1 1 1
0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 0 1 1 0 0 1 1 1 1
```

to that found in [14, p.244] by transposing the part of G at the right of the identity (the left-most column below is indeed the end of the first line of G for example)

```
0 1 1 1 1 1 0 0 0 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0
0 0 1 1 1 1 1 0 0 0 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0
0 1 1 0 0 0 1 1 0 0 1 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0
1 1 0 0 1 1 0 1 1 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0
1 1 1 0 0 1 1 0 1 1 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0 0
1 0 0 0 1 1 1 1 0 1 0 1 0 1 1 1 0 0 0 0 0 1 0 0 0 0
0 0 1 1 1 0 1 1 1 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 0 0
1 1 1 0 0 0 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 1 0 0
1 1 1 1 0 0 0 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 1 0
1 1 1 1 1 0 0 0 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 1
```

using this set of transforms (notice how the identity sub-matrix was shifted from left to right in the expression of matrix H).

5 Message interpretation

The most difficult part is now behind us: we have a sequence of bits, whose integrity is guaranteed following the error correcting code. We are simply left with analyzing the payload of each message, by replacing lines 60–62 of the previous example. All transactions are performed with the most significant bit first (MSB). We have limited our decoding ability to a few simple examples to keep the source code short: name of the station, clock (day/hour/minutes) or free text. Since we want to be able to identify the station being received, the first type of message we wish to decode is 0A or 0B as defined in the first 4 bits of block B: if the first 4 bits of B are zero, then block D contains the ASCII code of a string containing the name of the broadcast FM station. Block B is the second block whose payload is stored in variable `data2`, yielding the interpretation of the content of the last block as two ASCII characters, MSB first:

```
puissances=2.^[7:-1:0]'; % 128 64 32 6 8 4 2 1
 if (!isempty(pa) && !isempty(pb) && !isempty(pd))
   if ((data2(1:4)*puissances(end-3:end))==2)  % RDS text
     texte=[texte char(data3(1:8)*puissances)] % 2chars in C
     texte=[texte char(data3(9:16)*puissances)]
     texte=[texte char(data4(1:8)*puissances)] % 2chars in D
     texte=[texte char(data4(9:16)*puissances)]
   end
   if (sum(data2(1:4))==0)              % station name
     station=[station char(data4(1:8)*puissances)];
     station=[station char(data4(9:16)*puissances)]
   end
   if ((data2(1:5)*puissances(end-4:end))==2)  % 1A
      day=data4(1:5)*puissances(end-4:end);
      heure=data4(6:10)*puissances(end-4:end);
      minute=data4(11:16)*puissances(end-5:end);
      printf("time1A %d %d %d\n",day,heure,minute);
   end
```

```
    if ((data2(1:5)*puissances(end−4:end))==8)  % 4A
      day=data2(15:16)*2.^[16:−1:15]'+data3(1:15)*2.^[14:−1:0]';
      heure=data3(16)*2^4+data4(1:4)*puissances(end−3:end); % UTC
      minute=data4(5:10)*puissances(end−5:end);
      offset=data4(12:16)*puissances(end−4:end); % *30 min−>local
      printf("time4A␣%d␣%d␣%d␣%d\n",day,heure,minute,offset);
    end
  else % printf(".");
  end
end
printf("\n");
```

Notice the slight subtlety when converting the bit array to a number by using the dot-product (and not one to one product) of `data` with `puissance`, the latter containing the powers of 2 from 128 to 1 (most significant bit first).

The result of this computation, for stations received in Besançon (France), is

```
station = IRN  VIRGIGIN GIGIGIN GIN GIIRGIIRGIN  VGI VIR
```

on 100.4 MHz,

```
texte =  ES (FEAT. GUCCI MANE)      MOUVAE SREMMURD - BLACK BEAT(FEAT. GECCI MANE)      MOUV',
RAE SREMMURD - BLACK BEAT(FEAT. G]CCI MANE)          MOUVAE SREMMERD - BLACK BEATLES (FEAT. G MAN    MOUV'( RAE
station = LEOUOU MLEV'V' MOULE MOUV'OUV'LE MV'LEOUV'LE MOUV' MOUV'LE MOUV'LEV'LE MOUV'LEOUV'LE MOULE MOULEV' MOUV'
MOUV'LE MOUV'LE MV'LE MOUV'LE MOUV'LE MV'LE MOUV'LEOUV'LEOULE MV'LE MOUV
```

on 93.5 MHz, and

```
station = .9.9P BI.996.9BIP BI96BIP .9BI96.9P 96.9BIBI.9BI.996BIP .9BI.9BIP
```

on 96.9 MHz, which hints at records of the signal emitted by Virgin, Le Mouv' and BIP. Finally Rire & Chanson on 91.0 yields

```
station = ELLI& E &  RIR RE &  RIRE &  RE &  RIR&  R SLI SELLIG  SELLIG  SELLIG  SELLI RIRE &  RIRE &  RIRE &
RIRE &  RIRE &  RIRE  RIRE  R& IRE & IR R
```

Sellig seems to be a humorist so his name in the title of this broadcast station in not unlikely. More interesting, France Info yields

```
texte = N MOCH - SUR(LA CART FRAFRANCE INFO  14  : JULIEN MOCH - SUR LA CARTE DE FRAFRANCE INFO - LE| 17 : JULIEN
MOCH - SUR LA CARTE DE FRANCE FRANKE INFO - LE 14 | 17 : JULIEC@ - SUR LA CARTE DE FRANCE - LE 14 | 17 : JN MOCH -
SUR LA Cï£¡RTE DE FRACE INFO - LE 14 | 17 : JULIE
station =      FO    INFO    FO    INFOINININFO    INFO    FO  INFO        FO    INFO  INFO    FO    INFO
INFO    IN    INFO    INFO    INFO    INFO  INFO    INFO      INFO    IN    INFO    INFO    INFO    FO    INFO
FO  INFO    INFO    INFO  IN  INFO    INFO    INFO    FO
```

indicating that in addition to the title of the station, a free text with the title of the current program being broadcast is sent.

Without being perfect, these sentences demonstrate that the concept is sound and properly implemented, with results in agreement with those provided by `gr_rds` (Fig. 8).

6 Error correction

We have decoded RDS messages following the synchronization procedure to align sentences on a continuous bit stream, what more could we expect ? The 10-bit error correcting code added at the end of each 16 bit payload has only been used so far for synchronization and make sure of the integrity of the transmitted message. The RDS signal is noisy, and some sentences are eliminated of this investigation because their error correcting code does not match the expected value. Can we improve the reception decoding yield by attempting to correct transmission errors thanks to redundancies introduced by the error correcting code [16], process which contributed to the communication bandwidth increase of deep space probes (Fig. 9) [17] ? This will take us back in the processing sequence with respect to the last section, back to layer 2 of the OSI model, but will provide the opportunity for an experimental demonstration of implementing

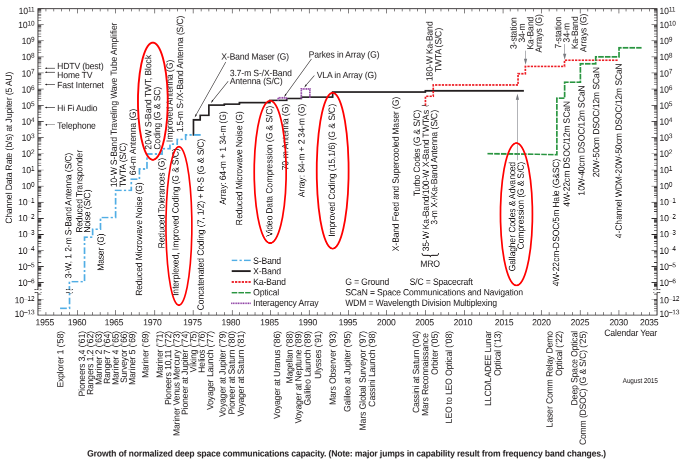
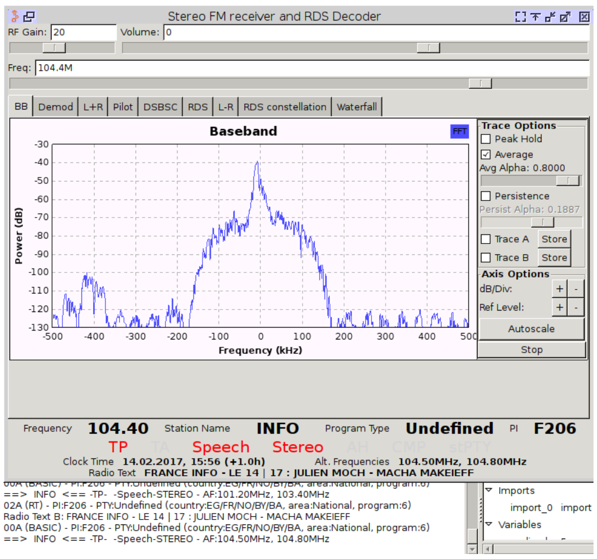


Figure 9: Evolution of the communication bandwidth of deep space probes.

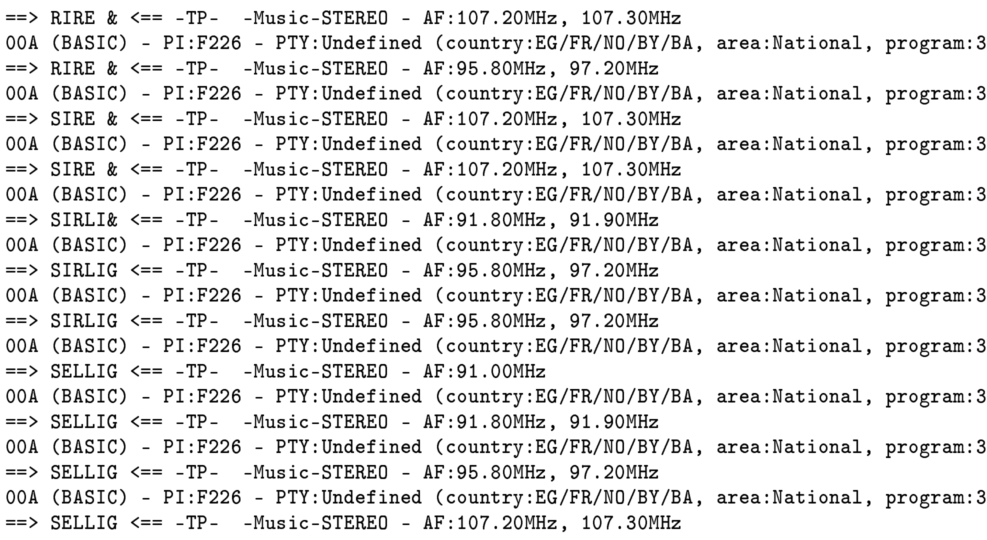


Frequency **104.40** Station Name **INFO** Program Type **Undefined** PI **F206**

Clock Time **14.02.2017, 15:56 (+1.0h)** Alt. Frequencies **104.50MHz, 104.80MHz**

Radio Text **FRANCE INFO - LE 14 | 17 : JULIEN MOCH - MACHA MAKEIEFF**

```
00A (BASIC) - PI:F206 - PTY:Undefined (country:EG/FR/NO/BY/BA, area:National, program:6)
==>  INFO  <== -TP-  -Speech-STEREO - AF:101.20MHz, 103.40MHz
02A (RT) - PI:F206 - PTY:Undefined (country:EG/FR/NO/BY/BA, area:National, program:6)
Radio Text B: FRANCE INFO - LE 14 | 17 : JULIEN MOCH - MACHA MAKEIEFF
00A (BASIC) - PI:F206 - PTY:Undefined (country:EG/FR/NO/BY/BA, area:National, program:6)
==>  INFO  <== -TP-  -Speech-STEREO - AF:104.50MHz, 104.80MHz
```



```
==> RIRE & <== -TP-  -Music-STEREO - AF:107.20MHz, 107.30MHz
00A (BASIC) - PI:F226 - PTY:Undefined (country:EG/FR/NO/BY/BA, area:National, program:3
==> RIRE & <== -TP-  -Music-STEREO - AF:95.80MHz, 97.20MHz
00A (BASIC) - PI:F226 - PTY:Undefined (country:EG/FR/NO/BY/BA, area:National, program:3
==> SIRE & <== -TP-  -Music-STEREO - AF:107.20MHz, 107.30MHz
00A (BASIC) - PI:F226 - PTY:Undefined (country:EG/FR/NO/BY/BA, area:National, program:3
==> SIRE & <== -TP-  -Music-STEREO - AF:107.20MHz, 107.30MHz
00A (BASIC) - PI:F226 - PTY:Undefined (country:EG/FR/NO/BY/BA, area:National, program:3
==> SIRLI& <== -TP-  -Music-STEREO - AF:91.80MHz, 91.90MHz
00A (BASIC) - PI:F226 - PTY:Undefined (country:EG/FR/NO/BY/BA, area:National, program:3
==> SIRLIG <== -TP-  -Music-STEREO - AF:95.80MHz, 97.20MHz
00A (BASIC) - PI:F226 - PTY:Undefined (country:EG/FR/NO/BY/BA, area:National, program:3
==> SIRLIG <== -TP-  -Music-STEREO - AF:95.80MHz, 97.20MHz
00A (BASIC) - PI:F226 - PTY:Undefined (country:EG/FR/NO/BY/BA, area:National, program:3
==> SELLIG <== -TP-  -Music-STEREO - AF:91.00MHz
00A (BASIC) - PI:F226 - PTY:Undefined (country:EG/FR/NO/BY/BA, area:National, program:3
==> SELLIG <== -TP-  -Music-STEREO - AF:91.80MHz, 91.90MHz
00A (BASIC) - PI:F226 - PTY:Undefined (country:EG/FR/NO/BY/BA, area:National, program:3
==> SELLIG <== -TP-  -Music-STEREO - AF:95.80MHz, 97.20MHz
00A (BASIC) - PI:F226 - PTY:Undefined (country:EG/FR/NO/BY/BA, area:National, program:3
==> SELLIG <== -TP-  -Music-STEREO - AF:107.20MHz, 107.30MHz
```

Figure 8: Left, gr_rds decoding of France Info: the graphical layout is slightly more attractive than executing a GNU/Octave script but the displayed result are about the same. We notice during its execution that the station name and free text display slowly accumulate letters, in agreement with the difficulty we have also met of obtaining a complete uncorrupted sentence. Right: the gr_rds output when listening at Rire & Chanson, confirming the inclusion of the name of the program being broadcast in the name of the station.

error correcting capability.

The implemented procedure not only allows identifying a corrupted message, but also to identify which bit was flipped: a Hamming code [15, chap.8] is able of such a feat for a unique error only. A BCH code [14, p.252],[15, chap.11] extends the concept to more than a single flipped bit: here, the implemented code allows for correcting up to two bits corrupted during transmission.

6.1 One error

Appendix C of [6] explains that the error correcting code is implemented, during emission, as a linear combination (sum modulo 2, or XOR) of the emitted bits.

A linear transform is implemented either as a linear feedback shift register with taps feeding XOR gates (Fig. 7), but since we prototype using GNU/Octave, we keep on using the matrix approach [14, p.244]: a 16×10 matrix computes 10 output bits considering 16 input bits. We have seen that a block identifier for synchronization was added to this error correcting code output: the transmitted message is made of 16 data bits followed by the 10-bit error correcting code to which the block identifier was added. Such a procedure is implemented in GNU/Octave as

```
A=  [0 0 1 1 1 1 1 1 0 0]; % A block emission
data=[0 1 0 0 1 0 1 0 0 1 0 0 1 1 0 1]; % JM
G=[0 0 0 1 1 1 0 1 1 1;  % 16x10: emitted message
  1 0 1 1 1 0 0 1 1 1;
  1 1 1 0 1 0 1 1 1 1;
  1 1 0 0 0 0 1 0 1 1;
  1 1 0 1 0 1 1 0 0 1;
  1 1 0 1 1 1 0 0 0 0;
  0 1 1 0 1 1 1 0 0 0;
  0 0 1 1 0 1 1 1 0 0;
  0 0 0 1 1 0 1 1 1 0;
  0 0 0 0 1 1 0 1 1 1;
  1 0 1 1 0 0 0 1 1 1;
  1 1 1 0 1 1 1 1 1 1;
  1 1 0 0 0 0 0 0 1 1;
  1 1 0 1 0 1 1 1 0 1;
  1 1 0 1 1 1 0 0 1 0;
  0 1 1 0 1 1 1 0 0 1];
mI=mod(data*G,2)
mI=mod(mI+A,2);  % add A identifier
envoi=[data mI]
```

If all goes well, this message 0 1 0 0 1 0 1 0 0 1 0 0 1 1 0 1 0 1 0 0 1 0 1 0 1 0 which contains the two ASCII characters "JM", is transmitted and received without corruption, so that decoding on the receiver side is achieved with

```
% p.75 US, 64 EU
sA2=[1 1 1 1 0 1 1 0 0 0]; reception syndrome
% p.74
H=[
 eye(10); % identity matrix
 1 0 1 1 0 1 1 1 0 0;
 0 1 0 1 1 0 1 1 1 0;
 0 0 1 0 1 1 0 1 1 1;
 1 0 1 0 0 0 0 1 1 1;
 1 1 1 0 0 1 1 1 1 1;
 1 1 0 0 0 1 0 0 1 1;
 1 1 0 1 0 1 0 1 0 1;
 1 1 0 1 1 1 0 1 1 0;
 0 1 1 0 1 1 1 0 1 1;
 1 0 0 0 0 0 0 0 0 1;
 1 1 1 1 0 1 1 1 0 0;
 0 1 1 1 1 0 1 1 1 0;
 0 0 1 1 1 1 0 1 1 1;
 1 0 1 0 1 0 0 1 1 1;
 1 1 1 0 0 0 1 1 1 1;
 1 1 0 0 0 1 1 0 1 1;
];

mIr=abs(mod(envoi*H,2)−sA2)
```

and the received message `mIr` must be 0 after subtraction of the syndrome of block A, here named sA2. This little numerical experiment validates the proper decoding sequence.

Let us assume now that between emission and reception, an error occurred:

```
N=3;envoi(N)=1−envoi(N); % error introduction
```

Can we do better than reject the sentence ? By displaying the result of the syndrome computation, we note that flipping the third bit yields a syndrome

```
mIr = 0    0    1    0    0    0    0    0    0    0
```

which indeed tells us that the third bit was corrupted. This is due to the fact that the first ten lines of `H` are the identity matrix. More generally, and error on bit N is detected by a syndrome equal to the Nth line of `H`. Indeed, and error might occur on any of the 26 bits of the message: if bit 25 is flipped for example

```
N=25;envoi(N)=1−envoi(N); % error introduction
```

then we obtain

```
mIr = 1    1    1    0    0    0    1    1    1    1
```

which is indeed the line before last of `H`. Searching for the corrupted bit is hence generalized with

```
[num,ligne]=ismember(mIr,H,'rows')
% if not 0, this is the pattern of the line of the matrix in which the error lies
```

with `num` telling us if the syndrome was identified as a line of `H`, and if so, `ligne` tells us which bit was flipped. We hence improve the decoding yield of RDS, with for example 24 characters in addition to the 680 already properly decoded when listening at Le Mouv' (+4%), 9 characters in addition to the 79 already acquired on Virgin (+11%) and an additional 10 to the 60 acquired when listening to BIP (+17%).

Experimenting with the two shapes of the H decoding matrix discussed earlier allow us to be convinced that they operate similarly, since the code properties are kept even if lines and columns are switched. Hence, we observe with

```
A=  [0 0 1 1 1 1 1 1 0 0]; % A block emission
sA2=[1 1 1 1 0 1 1 0 0 0]; % syndrome de reception

data=[0 1 0 0 1 0 1 0 0 1 0 0 1 1 0 1]; % JM
G=[0 1 1 1 1 1 0 0 0 0 1 1 1 1 1 0;
  0 0 1 1 1 1 1 0 0 0 0 1 1 1 1 1;
  0 1 1 0 0 0 1 1 0 0 1 1 0 0 0 1;
  1 1 0 0 1 1 0 1 1 0 1 0 0 1 1 0;
  1 1 1 0 0 1 1 0 1 1 0 1 0 0 1 1;
  1 0 0 0 1 1 1 1 0 1 0 1 0 1 1 1;
  0 0 1 1 1 0 1 1 1 0 0 1 0 1 0 1;
  1 1 1 0 0 0 0 1 1 1 1 1 0 1 0 0;
  1 1 1 1 0 0 0 0 1 1 1 1 1 0 1 0;
  1 1 1 1 1 0 0 0 0 1 1 1 1 1 0 1];
```

```
15
16 mI=mod(data*G',2);     % on a pris transposee de G pour prendre moins de place
17 mI=mod(mI+A,2);        % encodage du message
18 envoi=[data mI]        % message + code (identit'e a gauche de G')
19 N=5;envoi(N)=1−envoi(N);  % introduction d'une erreur
20
21 H1=[  % forme fournie dans document RDS => on en deduit le syndrome de A
22 1 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 0 1 1 0 0 1 1 1 ;
23 0 1 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 0 1 1 0 0 1 1 ;
24 0 0 1 0 0 0 0 0 0 0 1 0 1 1 1 0 0 0 1 0 1 1 1 1 1 0 ;
25 0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1 1 1 0 0 0 ;
26 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1 1 1 0 0 ;
27 0 0 0 0 0 1 0 0 0 0 1 0 1 0 1 1 1 1 1 0 1 0 1 0 0 1 ;
28 0 0 0 0 0 0 1 0 0 0 1 1 0 0 1 0 0 0 1 0 1 1 0 0 1 1 ;
29 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 0 1 1 0 0 1 1 1 1 1 0 ;
30 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 0 1 1 0 0 1 1 1 1 1 ;
31 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 0 1 1 0 0 1 1 1 1 ];
32
33 H2=[  % forme issue de [I | tG] => on en deduit A (matrice de controle)
34 0 1 1 1 1 1 0 0 0 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 ;
35 0 0 1 1 1 1 1 0 0 0 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 ;
36 0 1 1 0 0 0 1 1 0 0 1 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 ;
37 1 1 0 0 1 1 0 1 1 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 ;
38 1 1 1 0 0 1 1 0 1 1 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0 0 ;
39 1 0 0 0 1 1 1 1 0 1 0 1 0 1 1 1 0 0 0 0 0 1 0 0 0 0 ;
40 0 0 1 1 1 0 1 1 1 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 0 0 ;
41 1 1 1 0 0 0 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 1 0 0 ;
42 1 1 1 1 0 0 0 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 1 0 ;
43 1 1 1 1 1 0 0 0 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 1 ];
44
45 res=mod(envoi*H2'−A,2)  %
46 [tmp,col]=ismember(res,H2',"rows")
47 res=mod(envoi*H1'−sA2,2) % sA
48 [tmp,col]=ismember(res,H1',"rows")
```

that by encoding a message, including the block identifier (optional in these numerical experiments), we are on the one hand able to validate a transmission without error: comment the bit flip instruction in line 19 to observe the two decoding results, using $H1$ or $H2$, in both cases with null results. If on the other hand a bit is flipped, the result is indeed the line whose index is equal to the position of the erroneous bit.

6.2 Two errors

Recovering one corrupted bit seems to work properly, and we have understood how to identify which bit was corrupted by searching the line of `H` which contains the syndrome after decoding with one bit flipped. However, what happens if two bits are flipped ?

A quick numerical experience again puts us on the track:

```
N=2;envoi(N)=1−envoi(N); % error introduction
N=5;envoi(N)=1−envoi(N); % error introduction
```

yields

```
mIr = 0   1   0   0   1   0   0   0   0   0
```

Two bits are now set to 1 after decoding the syndrome, those of the second and fifth lines. The analysis is hence trivial as long as two of the first ten bits have flipped: the bit numbers appear as the indices of the ones in the decoded syndrome at the reception, again because the first ten lines of `H` are the identity matrix. This concept is generalized as found in the syllabus of [18]:"Suppose we wish to correct all patterns of t errors. In this case, we need to pre-compute more syndromes, corresponding to 0, 1, 2,... t bit errors. Each of these should be stored by the decoder."

We hence compute a new matrix, deduced from `H` and which we call `m2r`, which includes all 10-bits syndromes obtained as all possible combinations of line pairs of `H`. An approach using `for` loops, not necessarily the best when programming with GNU/Octave but which provides a good result, is

```
m2r=[0 0 0 0 0 0 0 0 0 0]; % seed
l=1;
for k=1:length(H)−1     % compute all combinations
 for j=k+1:length(H)
   m2r=[m2r ; mod(H(k,:)+H(j,:),2)];
   solution(l,1)=k;    % which line is to be corrected ?
   solution(l,2)=j;
   l=l+1;
 end
```

```
end
m2r=m2r(2:end,:);      % remove seed
```

Of course m2r is much larger than H since this time we have all possible error pairs represented as the lines of the new matrix. We check that all the resulting lines are unique – an expected property of a code designed to correct two communication errors – with

```
for k=1:length(m2r)
 [num,ligne]=ismember(m2r(k,:),m2r,'rows');
 if (num>1)
   printf("duplicate␣%d␣",num)    % never twice the same row
 end
end
```

which does not print any "duplicate" message (question: why does `[m2runiq,m,n]=unique(m2r,'rows');` return fewer lines in m2runiq than those found in m2r, although all lines are unique ?).

We can now search which pairs of bits flipped during transmission with

```
[num,ligne]=ismember(mIr,m2r,'rows')
if (num>0) solution(ligne,:),end % display which bits have flipped
```

and since we have taken care of filling solution with the couple j and k of the lines of H which were summed to yield a line of m2r, we know the indices k and j of the flipped bits. Indeed,

```
N=21;envoi(N)=1−envoi(N); % error introduction
N=25;envoi(N)=1−envoi(N); % error introduction
```

indeed yields

```
mIr = 0   0   0   1   0   1   0   0   1   1
num = 0
ligne = 0
num =  1
ligne =  314
ans = 21   25
```

in which we observe that looking for syndrome mIr in matrix H (one modification) yields no result (num=0) but the search in m2r indeed yields one result, with the syndrome found as line 314 of m2r, which was computed as the sum of the two lines when j=21 and k=25. Decoding hence operates properly. RDS only claims the ability to correct 1 or 2 bits flipped during transmission, so we stop here our investigation of error correcting codes.

The number of bits that can be corrected by a code is defined by the concept of Hamming distance, which graphically represents the distance between the "good" message and the corrupted message, below which the correction is possible (fr.wikipedia.org/wiki/Code_correcteur).

The ability of correcting corrupted transmission is a core issue allowing increased datarates (Fig. 10), whether during communication with deep space probes [17] or for short range low power communication [19]. This quick overview of a simple example provides the opportunity to further investigate a topic still currently active: without claiming we understand how to generate the codes, experimenting with real data with concrete results provides the motivation to further read more theoretical books dealing with this topic such as [14] and its excellent presentation of the relation between compression, cryptography and error correction.

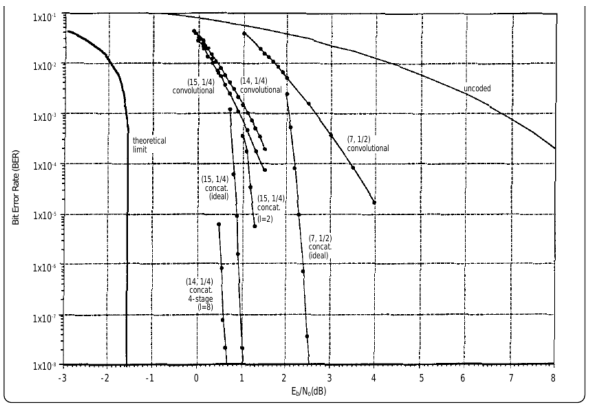


Figure 7-11. Telemetry communication channel performance for various coding schemes. The first set of curves shows the *Voyager* k = code, both alone and in combination with the Reed-Solomon code. The second set of codes illustrates the k = 15 code, which was to be demonstrated with Galileo's original high-rate channel, shown alone and in combination with the Reed-Solomon code, either as constrained by the *Galileo* spacecraft data system (I = 2) or in ideal combination. The third set shows the k = 14 code, devised by the Advanced Systems Program researchers for the actual Galileo low-rate mission, both alone and in combination with the selected variable-redundancy Reed-Solomon code and a complex four-stage decoder. The added complexity of the codes, which has its greatest effect in the size of the ground decoder, clearly provides increased reliability for correct communication.

Figure 10: Evolution of the bit error rate as a function of the signal to noise ratio, for various error correcting code types, as found in [17, Fig. 7-11, p.477]

7 Conclusion

We have investigated the RDS protocol, exploited for transmitting digital information associated with commercial FM broadcast station including the name of the station or the kind of program, by starting with the acquisition of the raw analog signal using a digital video broadcast (DVB-T) receiver, then extracted the digital information on the 57 kHz sub-carrier, before extracting sentences (after solving the synchronization issue) and interpret their content. We have finally tackled the error correction issue by showing that we could introduce error and identify which bits had been flipped. Using this error correction method on experimental data, we could recover an additional 10% characters with respect to the decoding yield limited to un-corrupted sentences.

Many more developments would be needed before this proof of concept could be considered robust enough to be used practically, especially on the 1187.5 bps bitrate synchronization, but that would make the decoding code longer and less readable. We consider the proof of concept to be explicit enough to demonstrate understanding of digital communication basics with decoded sentences consistent with each broadcast station. This exercise was completed in about 2 months: it is hence a topic worth investigating for grasping the whole processing chain of a communication protocol, from the hardware to the session layers.

A disappointing issue is the poor time transfer capability through this medium. We have only received few timing sentences (CT, group 4A) which provide the date and time with a resolution of $\pm 0,1$ s at the beginning of each minute. This poor time-transfer capability will be tackled soon with software decoding of DCF77 presented elsewhere.

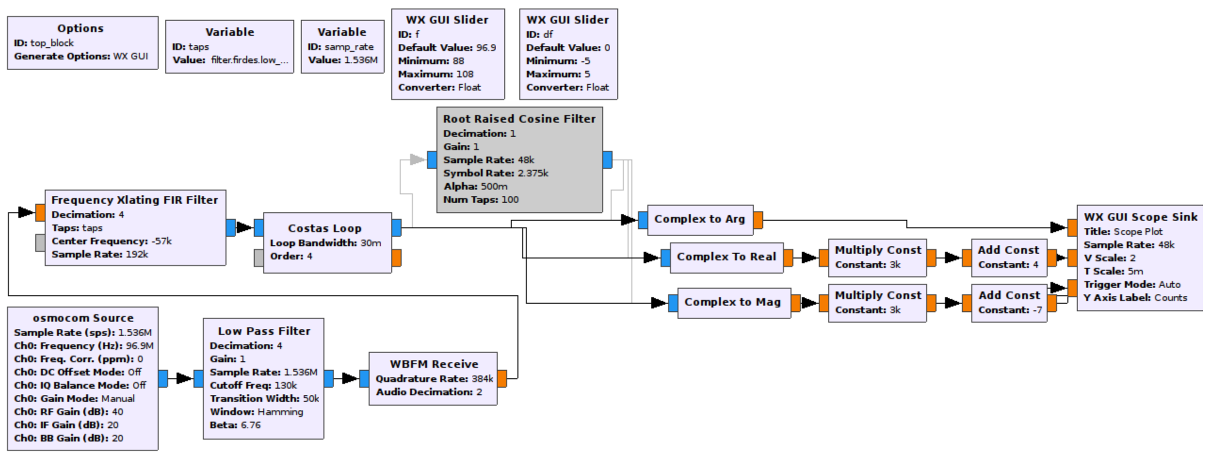
An archive with the GNU/Octave scripts, GNURadio Companion configurations and example files, is available at `http://jmfriedt.free.fr/lm_rds.tar.gz`.

Acknowledgement

Thomas Lavarenne has motivated this study through his questions and constructive comments, as did Bastien Bloessl by commenting that RDS decoding is trivial [23], an unacceptable comment as long as this protocol had resisted our investigations. This deficiency is now corrected. G. Cabodevila has improved the GNU/Octave code quality during his proof reading, and É. Carry helped clarify some sections. Despite having bought a paper copy of [14], all references were collected from the Library Genesis, `gen.lib.rus.ec`, an essential resource to our research activities.

A Phase or amplitude ?

We have selected throughout this document to consider RDS to be phase-modulated since our initial attempts at displaying the magnitude of the filtered signal (Band Pass Filter around 57 ± 2.5 kHz) yielded unusable results. Although phase and frequency control using Costas loop remain mandatory, displaying the real part of the output instead of the phase provides a usable solution (Fig. 11). A few oscillations are visible on the longest symbols, which might be interpreted erroneously of a short symbol if care is not taken: an optimized low-pass filter reduces this fluctuation amplitude and improves the detection chances of the transmitted signal. Indeed, filtering with a rectangular window in the frequency domain induces, through the Fourier transform of the rectangle window, by a convolution with a cardinal sine (or sinc, $\sin(x)/x$) and hence a spreading in the time domain of each symbol on its neighbors considering the slow decay of the filter. Whatever the filtering used, the magnitude of the complex signal always yields unusable results for detecting bits: only the phase or real part are usable.



Options
ID: top_block
Generate Options: WX GUI

Variable
ID: taps
Value: filter.firdes.low_...

Variable
ID: samp_rate
Value: 1.536M

WX GUI Slider
ID: f
Default Value: 96.9
Minimum: 88
Maximum: 108
Converter: Float

WX GUI Slider
ID: df
Default Value: 0
Minimum: -5
Maximum: 5
Converter: Float

Root Raised Cosine Filter
Decimation: 1
Gain: 1
Sample Rate: 48k
Symbol Rate: 2.375k
Alpha: 500m
Num Taps: 100

Frequency Xlating FIR Filter
Decimation: 4
Taps: taps
Center Frequency: -57k
Sample Rate: 192k

Costas Loop
Loop Bandwidth: 30m
Order: 4

Complex to Arg

Complex To Real

Complex to Mag

Multiply Const
Constant: 3k

Multiply Const
Constant: 3k

Add Const
Constant: 4

Add Const
Constant: -7

WX GUI Scope Sink
Title: Scope Plot
Sample Rate: 48k
V Scale: 2
T Scale: 5m
Trigger Mode: Auto
Y Axis Label: Counts

osmocom Source
Sample Rate (sps): 1.536M
Ch0: Frequency (Hz): 96.9M
Ch0: Freq. Corr. (ppm): 0
Ch0: DC Offset Mode: Off
Ch0: IQ Balance Mode: Off
Ch0: Gain Mode: Manual
Ch0: RF Gain (dB): 40
Ch0: IF Gain (dB): 20
Ch0: BB Gain (dB): 20

Low Pass Filter
Decimation: 4
Gain: 1
Sample Rate: 1.536M
Cutoff Freq: 130k
Transition Width: 50k
Window: Hamming
Beta: 6.76

WBFM Receive
Quadrature Rate: 384k
Audio Decimation: 2

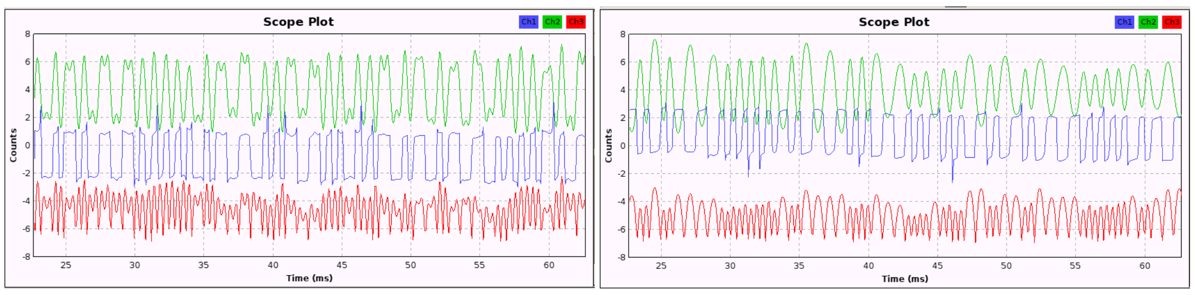


Figure 11: Top: processing sequence to extract the various components of the RDS sub-carrier following the correction of the local oscillator offset with a Costas loop. The phase will be displayed in blue, and the real part in green. Bottom: left without low-pass RRC, the real part exhibits some oscillations on the longest symbols. Bottom middle: a RRC filter attenuates such artifacts and yields a usable real part. In all cases, the magnitude of the complex (red) is unusable.

The Root Raised Cosine (RRC [20]) filter was designed to correct such a deficiency: a low pass filter able to reduce occupation both in the time and frequency domains, with the ability to finely select a given frequency band while preventing too slow a decay. While a rectangular filter in the frequency domain (blue on Fig. 12, left, case $\alpha = 0$) provides utmost spectral selectivity, its time domain response is very long with a null at the position of adjacent symbols, but any error on the timing between emission and sampling on the receiver side (Fig. 12, left of the time-domain curve) induces some leakage of energy on neighboring symbols, with the larger the contribution the further away the neighbors are. The RRC also exhibits zeros on neighbor symbol positions, but its decay is much faster (Fig. 12, right, case $\alpha = 1$), preventing the pollution of neighbors if synchronization errors occur [8, p.136],[21]. The α parameter provides a degree of freedom for continuously shifting from utmost spectral selection to optimal tradeoff between spectral occupation and time-domain decay of the shaped pulse. Thus, RRC aims at optimizing the spectral occupation of the communication channel (reduction of the bandwidth B) used for transmitting the digital stream, while maximizing the bitrate by allowing the reduction of the time interval $1/B$ between successive symbols.

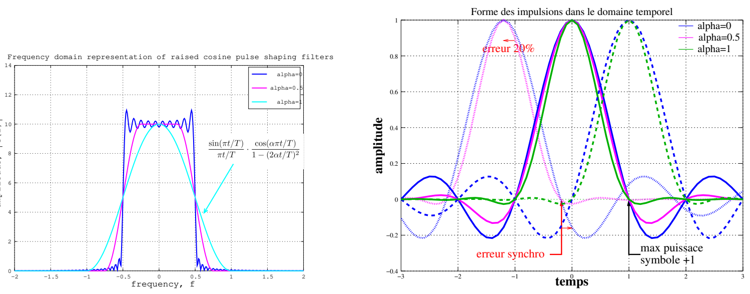


Figure 12: Left: spectral response of RRC filters with various shapes, from rectangular (best frequency selectivity) to the segment of trigonometric functions. Right: multiple successive pulses in the time domain, filtered with the same RRC shapes depicted in the frequency domain, with the left-pulse shifted by 20% with respect to its nominal timing assuming a constant bitrate.

B Bitrate control

Phase modulation requires a local copy of the unmodulated radiofrequency carrier, as generated for example by the Costas loop which eliminates the phase modulation from the received signal. Once bits have become visible as 0 or π phase values (for BPSK), the bitrate synchronization remains to identify when to sample the phase. The clock generating the digital transmission (for example a microcontroller on the emitter) has no reason to be clocked on the same source than the radiofrequency carrier, and hence another feedback loop much control the sampling rate on the receiver side to reproduce the emitter bitrate clock for the decoding to be possible (Fig. 13). Various strategies are available, such a maximizing the number of transitions on the emitted signal to ease the control of the receiver clock (case of the Manchester encoding which provides at least one transition during each bit transmission). GNURadio provides various blocks for bitstream synchronization, such as `Clock Recovery` (using the algorithm published in [22]) or `MPSK Receiver`. These blocks are fed with the output of the Costas loop which has already taken care of correcting the coarse frequency error, and now handle digital symbols generated by the previous processing steps to adjust the bitrate. These blocs output one sample per symbol, making later processing steps much easier (GNURadio `Binary Slicer` by saturating and comparing – for further analysis of the resulting digital stream). We validate the proper operation of these blocks by displaying the constellation diagrams, which exhibit on their ordinate the imaginary part of the output and the real part in abscissa. Since in the complex plane (phase diagram) the angle with the abscissa axis represents the complex number phase and the distance to the origin the magnitude, a point cloud at constant angle and constant distance indicate efficient synchronization. A "dancing" point cloud or shaped as a circle indicate no or poor synchronization. The output of the Costas loop illustrates the latter case (Fig. 14, right) while the output of the processing by one of the synchronization blocks for locking the bitrate indeed yields two distinct clouds (Fig. 14, left) which separate the two possible states expected from a BPSK modulation. We note here that the RRC filter described earlier. despite not significantly changing the visual aspect of the time-domain signal, helps in bitrate clock synchronization by reducing leakage of power from one symbol to its neighbors. A low pass filter induces the same result, but less efficiently.

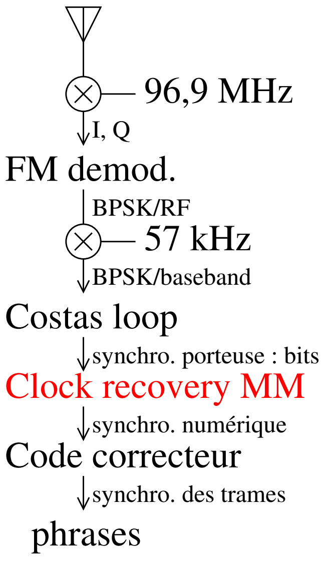
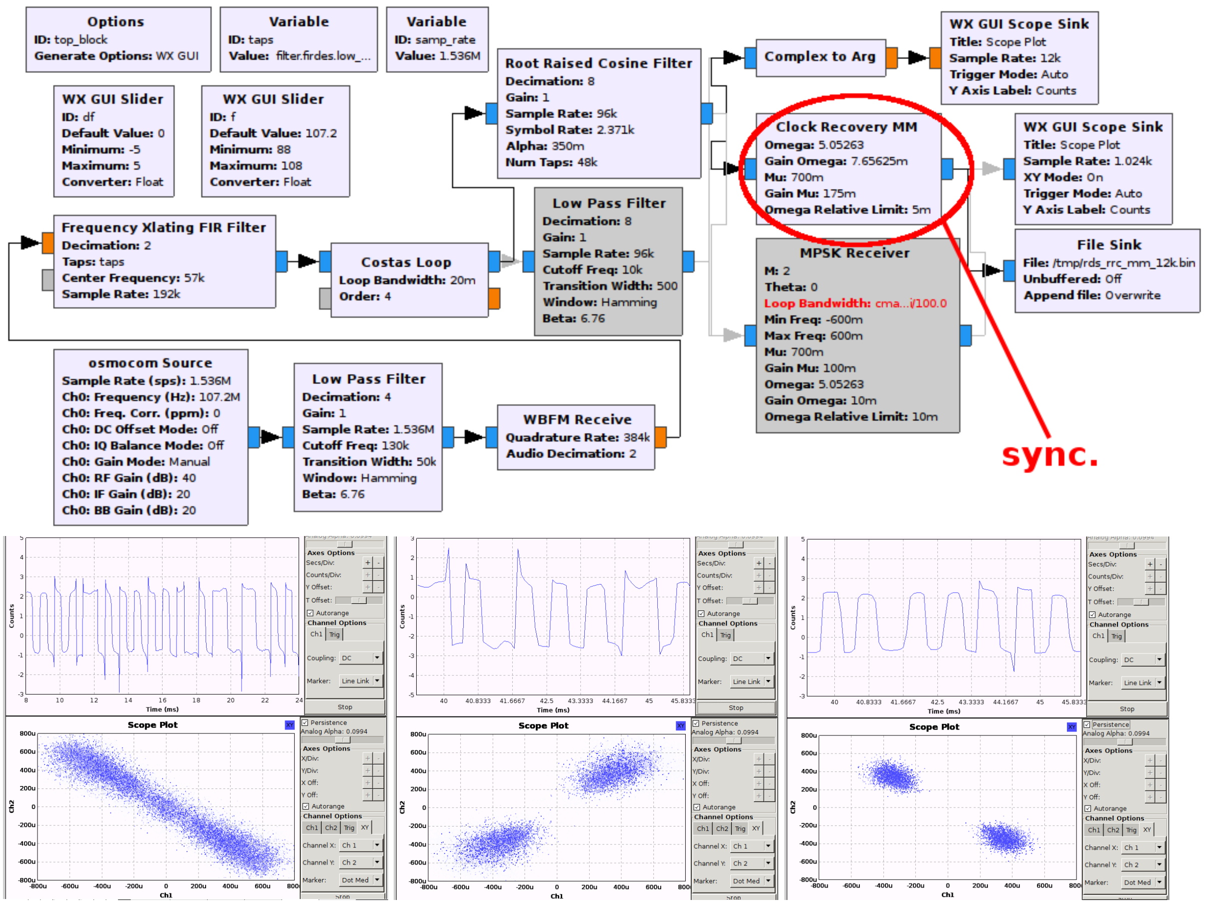


Figure 13: Signal processing chain for processing the information collected by the antenna in order to recover messages (sentences): in this appendix, we are interested in the bitrate synchronization, in red.

Configuring synchronization blocks as described at `http://gnuradio.org/redmine/projects/gnuradio/wiki/Guided_Tutorial_PSK_Demodulation` requires minimizing the number of samples per symbol, while


Options
ID: top_block
Generate Options: WX GUI

Variable
ID: taps
Value: filter.firdes.low_...

Variable
ID: samp_rate
Value: 1.536M

Root Raised Cosine Filter
Decimation: 8
Gain: 1
Sample Rate: 96k
Symbol Rate: 2.371k
Alpha: 350m
Num Taps: 48k

Complex to Arg

WX GUI Scope Sink
Title: Scope Plot
Sample Rate: 12k
Trigger Mode: Auto
Y Axis Label: Counts

WX GUI Slider
ID: df
Default Value: 0
Minimum: -5
Maximum: 5
Converter: Float

WX GUI Slider
ID: f
Default Value: 107.2
Minimum: 88
Maximum: 108
Converter: Float

Clock Recovery MM
Omega: 5.05263
Gain Omega: 7.65625m
Mu: 700m
Gain Mu: 175m
Omega Relative Limit: 5m

WX GUI Scope Sink
Title: Scope Plot
Sample Rate: 1.024k
XY Mode: On
Trigger Mode: Auto
Y Axis Label: Counts

Low Pass Filter
Decimation: 8
Gain: 1
Sample Rate: 96k
Cutoff Freq: 10k
Transition Width: 500
Window: Hamming
Beta: 6.76

Frequency Xlating FIR Filter
Decimation: 2
Taps: taps
Center Frequency: 57k
Sample Rate: 192k

Costas Loop
Loop Bandwidth: 20m
Order: 4

MPSK Receiver
M: 2
Theta: 0
Loop Bandwidth: cma...i/100.0
Min Freq: -600m
Max Freq: 600m
Mu: 700m
Gain Mu: 100m
Omega: 5.05263
Gain Omega: 10m
Omega Relative Limit: 10m

File Sink
File: /tmp/rds_rrc_mm_12k.bin
Unbuffered: Off
Append file: Overwrite

osmocom Source
Sample Rate (sps): 1.536M
Ch0: Frequency (Hz): 107.2M
Ch0: Freq. Corr. (ppm): 0
Ch0: DC Offset Mode: Off
Ch0: IQ Balance Mode: Off
Ch0: Gain Mode: Manual
Ch0: RF Gain (dB): 40
Ch0: IF Gain (dB): 20
Ch0: BB Gain (dB): 20

Low Pass Filter
Decimation: 4
Gain: 1
Sample Rate: 1.536M
Cutoff Freq: 130k
Transition Width: 50k
Window: Hamming
Beta: 6.76

WBFM Receive
Quadrature Rate: 384k
Audio Decimation: 2

Figure 14: Top: signal processing chain, with the addition of the bitrate synchronization blocks following the carrier synchronization (Costas). Bottom: constellation diagrams (XY mode oscilloscope fed with complex data) for various processing conditions – from left to right, output of the Costas loop (no bitrate synchronization), bitrate synchronization following a low-pass filter, and finally following and RRC. Two distinct point clouds ensure proper symbol separation.

keeping that number above 2. Doing so is achieved by introducing an RRC between the Costas loop and the bitrate synchronization block, with a decimation factor large enough to reach an Omega parameter (ratio of the sampling rate to the bitrate) close but above 2. In our example, Omega is equal to `samp_rate/128/(1187.5*2)` with 128 the product of all decimation factors of previous processing blocks between source and bitstream synchronization blocks, $1187,5\times2$ the bitrate when using a differential Manchester encoding, and `samp_rate` the sampling rate of the source. Other processing parameters of the synchronization block remain unchanged. The RRC is configured with an input rate equal to the source sample rate divided by the product of the various decimation factors found in the preceding processing chain, and again a symbol rate equal to $1187,5\times2$, defining its cutoff frequency.

The file recorded at the output of the carrier synchronization (Costas) and bitstream synchronization (Clock Recovery or MPSK) is perfectly decoded with the scripts described in the main text (section 3), after removing the first 4 lines since we already now have one sample for each symbol, and only keeping from lines 8 to 24 the following part
`p=angle(r);p=p-mean(p);s=(p>0);s=s-mean(s);s1=s(2:2:end);s2=s(1:2:end-1);`. Indeed, s is the saturated version of phase p, analyzed as a differential Manchester encoding in which successive pairs are considered to generate the bitstream. After that, the whole processing sequence (synchronization on the syndrome of each 16 bit sentences) remains unchanged, yielding for example

```
station = P 96.9BIP 96.9BIP 96.9BIP 96.9BIP 96.9BIP 96.9BIP 96.9BIP 96.9BIP
```

which is this time perfect, or

```
station = EUROPE 1EUROPE 1EUROPE 1EUROPE 1EUROPE 1EUROPE 1EUROPE
```

again without corrupted sentences, and even

```
station = RIRE &  RIRE  RIRE &  RIRE &  RIRE &  RIRE   JEAN    JEAN    JEAN    JEAN    JEAN
JEDUJARDINDUJARDINDUJARDINDUJARDINDUJARDINDUJARD
temps4A 57811 7 51 2
```

or

```
texte =  FRANNFO - LE 9 : FABIENN - 8APHATIE        FRANCE INFO - LE 7 | 9 : FABIENNE SINTES - 8H30 APHATIE
  FRANCE INFO - LE 7 | 9 :IENNE SINTES - 8H30 APHATIE        FRANCE INFO - LE 7 | 9 : FABIENNE SINTES - 8H30 APHATIE
  FRANCE INFO - LE 7 | 9 : FABIENNE SINTES - 8H30 APHATIE
station =FO    INFO    INFO    INFO    INFO    INFO    INFO    INFO    INFO    INFO    INFO    INFO    INFO    INFO
  INFO    INFO    INFO    INFO    INFO    INFO    INFO    INFO    INFO    INFO    INFO    INFO    INFO    INFO    INFO
temps4A 57811 7 54 2
```

which are close to perfect. The date is in agreement with our expectation: the date is given in modified Julian date as day 57811, which `http://www.csgnetwork.com/julianmodifdateconv.html` converts to February 27 2017, and the time is 7h5{1,4} shifted by 2 half-hours, which is indeed 8h5{1,4}, the time at which the signal was recorded. Sentence 4A [6, p.28] is hence properly decoded and allows time transfer using RDS, with an update every minute claimed to be accurate to within ±0,1 s. It is only after implementing bitrate synchronization that we managed to obtain timing sentence shaped following the 4A format since they are only emitted once every minute. Too low a sentence decoding yield reduces the chances of grasping this relatively rare transmission:

```
station = RT  RTL2    RT  RTL2    RT  RTL2    RT  RTL2    RT  RTL2    RT  RTL2    RT  RTL2    RT  RTL2    RT  RTL2
temps4A 57811 8 3 2
```

All examples presented in this appendix were recorded in a complex format (2×4 bytes) representing each symbol, as provided at the output of the `Clock Recovery MM` block. At a rate of 1187.5 × 2 = 2375 Hz, records are typically files of a few hundreds of kilo-bytes for acquisitions lasting a few tens of seconds (19000 kB/s).

C PI code

Each RDS transmission includes, in its first block (A [6, p.15]), the PI code of the received station. [6, p.66] hints that knowing the PI code, unique to each broadcast station, one might be able to synchronize the bitstream on this bit sequence which repeats every 104 bits (4 blocks 26 bit long each), instead of struggling with error correcting codes to be computed on each 26 bit sequences of the received bitstream and checking if the last 10 bits match the syndrome of the first 16 bits as was done here. We could have changed the story of our investigation when writing this article by demonstrating first, before addressing the error correcting code issue, the synchronization on the PI code, but the truth is that the list of PI codes was only recovered at the very end of writing this text. Indeed, the list of PI codes for all authorized broadcast stations is made available by the French regulating agency CSA at `www.csa.fr/maradiofm/radiords_tableau`. As an example for Radio Campus in Besançon, we learn that the PI code is FC3A. Adding in the sentence decoding loop, similarly to the free text or station name decoding strategy, the PI code decoding with

```
if (PI==0) PI=dec2hex(data1(1:16)*2.^[15:-1:0]')  % PI
```

and having taken care of initializing `PI=0` out of the loop, our GNU/Octave script indeed decodes and identifies `PI = FC3A` when listening at 102.4 MHz. The code was properly identified.

References

[1] A. Barisani & D. Bianco, *Hijacking RDS-TMC Traffic Information signals*, Phrack **64** (5) (2011) at `phrack.org/issues/64/5.html#article`, and associated archive at `dev.inversepath.com/rds/`

[2] L. Li, G. Xing, L. Sun, W. Huangfu, R. Zhou, & H. Zhu, *Exploiting FM radio data system for adaptive clock calibration in sensor networks*, Proc. of the 9th ACM International Conference on Mobile systems, applications, and services, pp. 169–182 (2011)

[3] D. Symeonidis, *RDS-TMC spoofing using GNU Radio*, Proc. 6th Karlsruhe Workshop on Software Radios (pp. 87–92), Mars 2010, with a copy provided by the author and made available at `jmfriedt.free.fr/Symeonidis.pdf`

[4] E. Fernandes, B. Crispo, & M. Conti, *FM 99.9, Radio Virus: Exploiting FM Radio Broadcasts for Malware Deployment*, IEEE Trans. on Information Forensics and Security **8** (6), pp.1027–1037 (2013)

[5] J.-M Friedt, *La réception de signaux venus de l'espace par récepteur de télévision numérique terrestre*, OpenSilicium **13** (Dec 2014/Jan-Feb 2015)

[6] European Standard EN 50067, *Specification of the radio data system (RDS) for VHF/FM sound broadcasting in the frequency range from 87,5 to 108,0 MHz* (April 1998), available at `www.interactive-radio-system.com/docs/EN50067_RDS_Standard.pdf`, or for its American version `http://www.nrscstandards.org/DocumentArchive/NRSC-4%201998.pdf`

[7] S.A. Tretter, *Communication System Design Using DSP Algorithms, With Laboratory Experiments for the TMS320C30*, Chap. 6 "Double-Sideband Suppressed-Carrier Amplitude Modulation and Coherent Detection", Springer (1995)

[8] J.R. Barry, E.A. Lee & D.G. Messerschmitt, *Digital Communication*, Springer (2004)

[9] J.-M Friedt, G. Cabodevila, *Exploitation de signaux des satellites GPS reçus par récepteur de télévision numérique terrestre DVB-T*, OpenSilicium **15**, July-Sept. 2015

[10] J.-M Friedt, G. Goavec-Mérou, *La réception radiofréquence définie par logiciel (Software Defined Radio – SDR)*, GNU/Linux Magazine France **153** (October 2012), pp.4–33

[11] W. Wesley Peterson & E.J. Weldon, *Error-Correcting Codes, 2nd Ed.*, MIT Press (1996)

[12] P. Topping, *RDS decoding for an HC11-controlled radio*, Motorola AN495/D (1994)

[13] Y. Guidon, *Comprendre les générateurs de nombres pseudo-aléatoires*, GNU/Linux Magazine France **81**, pp.64–76 (2006)

[14] J.G. Dumas, J.L. Roch, S. Varrette, & E. Tannier, *Théorie des codes : compression, cryptage, correction*, Dunod (2007)

[15] R.A. Hill, *A First Course in Coding Theory*, Oxford University Press (1990)

[16] N. Patrois, *Réparer un code QR*, GNU/Linux Magazine **198** (Nov. 2016)

[17] `http://descanso.jpl.nasa.gov/performmetrics/profileDSCC.html` includes digital encoding schemes in some of the significant bandwidth increase steps, while Fig. 7-11 (page 477) of `https://history.nasa.gov/SP-4227/Chapter07/Chapter07.PDF` shows the drop in bit error rate as channel encoding schemes are improved.

[18] Syllabus of course MIT 6.02, H. Balakrishnan & G. Verghese, *Introduction to EECS II: Digital Communication Systems*, chapter *6: Linear Block Codes: Encoding and Syndrome Decoding*, available at `https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-02-introduction-to-eecs-ii-digital-communication-systems-fall-2012/readings/MIT6_02F12_chap06.pdf` (2012)

[19] E. Tsimbalo, X. Fafoutis & R. Piechocki, *Fix it, don't bin it! CRC error correction in Bluetooth Low Energy*, Proc. 2nd IEEE World Forum on Internet of Things (WF-IoT), 2015 at `http://eis.bris.ac.uk/~xf14883/files/conf/2015_wfiot_crc.pdf`

[20] E. Cubukcu, *Root Raised Cosine (RRC) Filters and Pulse Shaping in Communication Systems* (2012), at `https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20120008631.pdf`

[21] `http://www.dsplog.com/2008/04/22/raised-cosine-filter-for-transmit-pulse-shaping/` provides a GNU/Octave script for experimenting with RRC filters.

[22] H. Meyr, M. Moeneclaey, S.A. Fechtel, *Digital Communication Receivers: Synchronization, Channel Estimation, and Signal Processing*, John Wiley & Sons (1998)

[23] B. Bloessl, *First Steps in Receiving Digital Information with RDS/TMC*, FOSDEM (2015), at `https://archive.fosdem.org/2015/schedule/event/sdr_rds_tmc/`