E4k and RTL2832U based Software Defined Radio – SDR)

J.-M Friedt, G. Goavec-Mérou,
3 décembre 2012

The wide diffusion of radio communication in consumer electronics yields the availability of components with capabilities compatible with these communication modes (operating at least in the 50 to 2500 MHz range, several MHz bandwidth) and affordable thanks to mass production. This trend is the opportunity to present the use of a digital broadcast television (DVB) receiver – based on the Elonics E4000 chip – whose receiving stage is so simple (zero-intermediate frequency) that it appears compatible with many analog and digital radiofrequency transmission modes. The raw data flow generated by the receiver is processed using classical algorithms implemented as blocs in the `gnuradio` software and the associated graphical user interface `gnuradio-companion`. We will demonstrate, beyond the use of the available signal processing blocs, how to implement our own algorithms for decoding the content of the incoming data flow. We will particularly focus on the *packet* communication protocol, ACARS and radiomodems communicating using FSK and AFSK modulation.

1 Introduction to software defined radio (SDR)

Software Defined Radio (SDR) aims at processing radiofrequency (RF) signals by using as much software as possible instead of being dependent upon hardware [1, 2, 3, 4].

Considering the frequencies and the data flow when communicating through wireless RF media, most signal processing steps had to rely on dedicated hardware, with software only handling the resulting low bandwidth data flow, usually in the audio range (<100 kHz). FM and AM receivers are amongst such usual receivers, in which no software at all is required, while several digital communication modes including the commonly used amateur radio (*packet*) mode is decoded by processing the digitized audio output (sound card input) of the RF receiver [1]. The same approach has been demonstrated for image transmission (Slow Scan TeleVision – SSTV – decoded by the now defunct JVFAX under MS-DOS, more recent tools being probably available nowadays under GNU/Linux) or satellite weather data reception as described in [5].

Classically, software processing of information transferred through a wireless link is confined to audio-frequency due to the wide availability of an acquisition interface – the sound card fitted in most personal computers manufactured in the last ten years – and the processing power which is far from negligible, even for such low sampling rates as 100 kHz. Indeed, the processing speed is defined by the incoming data stream rate, and any gap (for example for processing other tasks requested by the operating system) yields irreversible data loss. Since none of the consumer operating systems, and especially not GNU/Linux in its basic implementation, aims at providing bounded latency (real time capability), the availability of excessive computational power is the only reliable mean of preventing information loss during the data processing.

The bandwidth limitation on the acquisition interfaces was partly removed with the availability of video acquisition cards (*frame grabber*) : compatible with rates of several million samples per second, the data stream becomes compatible with a software analysis of radiofrequency bands several MHz wide. However, the initial stage of radiofrequency signal processing – amplifier and shifter to a frequency range low enough to be compatible with an analog to digital converter – remains hardware. Recently, a Digital Video Broadcast television (DVB) receiver available as a USB dongle has renewed interest in the field of free, opensource SDR since it appeared that the data stream provided to the video decoding software happens to be very general (we will later explain the meaning of the I/Q data flow provided by this hardware interface) : why then restrict the analysis of the recovered data to the television frequency bands, and not probe all the other bands accessible to the radiofrequency tuner, in order to interpret the content of the resulting stream using software digital signal processing ?

1. `http://www.baycom.org/~tom/ham/linux/multimon.html`

In order to reach such a target, a convenient development environment is `gnuradio`[2], which provides most of the basic processing blocs, allowing for quickly getting started in tackling the proposed issue.

The analysis sequence we propose first starts with a brief description of the hardware, before using some of the signal processing blocks provided by `gnuradio` through its graphical interface `gnuradio-companion`[3]. Since *using* available processing blocks is hardly a technical challenge for the developer beyond the entertainment aspect, we then focus on processing an unknown data stream, namely the for extracting information transmitted by a radiomodem on the one hand, and by planes using the ACARS protocol on the other hand. Finally, we will see how the signal post-processing algorithm are implemented as real time processing blocks compatible with `gnuradio-companion`.

2 Hardware aspects

Historically, SDR development on open hardware has focused on successive generations of Universal Software Radio Peripherals (USRP) developed by Ettus Research Ettus Research[4]. Although at a price of a fraction of the cost of professional systems, the investment of a few thousand euros remains confined to the dedicated amateur whose needs are justified by the performance of such hardware (wide operating bandwidth, sensitivity, emission in addition to reception). The USRP related functionalities supported by `gnuradio` are implemented in the `gr-uhd` drivers (UHD meaning *USRP Hardware Driver*) : this comment means that any tool or library including the name uhd will be *useless* in the following discussion and do *not* need to be compiled.

Indeed, while the USRP was made possible by the drop of the price of RF components developed for wireless digital communication (frequency synthesizers, I/Q modulators and demodulators operating with bandwidths from a few MHz up to several GHz, especially by companies such as Analog Devices, Maxim or Semtech), this trend now accelerates with the availability of compact integrated circuits including all the analog RF front end functionalities classically implemented using discrete components.

One newer chip in particular, the Elonics E4000 (direct RF receiver with zero-intermediate frequency operating in the 64 to 1700 MHz range) has generated a massive set of low cost circuits compatible with SDR applications. Being such a simple circuit – an RF amplifier located right after the antenna, a mixer with a local oscillator for frequency transposition[5] followed by low pass filters to only keep the low frequency component – most processing functionalities are necessarily moved to a software processing level. This *analog* component thus generates the I (Indentity) and Q (quadrature) outputs with a bandwidth of up to 8 MHz.

The analog signals must then be digitized for the digital data processing to be possible. A first project targets high resolution analog-to-digital conversion by using the same chip as would be used on sound cards : Funcube[6]. The low production volumes of the board yield a cost in the one hundred euros range, for a receiver with good sensitivity compatible with the reception of signals of emitted by satellite orbiting the Earth, but with a reduced bandwidth as found in a sound card of about 100 kHz.

In order to improve this bandwidth limitation, another project introduces a Field Programmable Gate Array (FPGA) to increase the sampling rate : OsmoSDR[7] involves such well known developers as Harald Welte[8], guarantee of seriousness of the project. Software developments around this project have thus benefitted from the commercial availability of the DVB receiver dongles, exhibiting poor sensitivity but whose large-scale manufacturing yields affordable products. Indeed, porting the driver for these peripherals to V4L has demonstrated that the data decoding is performed at the software level (a sad reminder of the software defined modems – also known as winmodems – which already targeted cheaper products by reducing the number of hardware components but were unusable in a free environment due to lack of the associated drivers[9]) and that the raw I/Q stream is appropriate for application much more interesting than viewing television programs.

2. `gnuradio.org`
3. `http://www.joshknows.com/grc`
4. `http://www.ettus.com/`
5. the basic principle of the mixer is to shift an incident signal of frequency f and a local oscillator reference signal f_0 by generating $f \pm f_0$: the sum-frequency component is removed using a low-pass filter.
6. `http://tetra.osmocom.org/trac/wiki/Funcube_Dongle` and `http://www.funcubedongle.com/`
7. `sdr.osmocom.org`
8. `https://har2009.org/program/speakers/253.en.html`
9. `http://en.wikipedia.org/wiki/Softmodem`

Getting involved in the fantastic field of SDR now only requires a hardware investment of about 20 euros : we have experimented with the EZCAP (DVB-T/DAB/FM) dongle as (for example) available at `http://www.dealextreme.com/p/mini-dvb-t-digital-tv-usb-2-0-dongle-with-fm-dab-remote-controller-92C` or `https://www.cosycave.co.uk/product.php?id_product=104`.

The project describing the analog to digital converter embedded on such dongles, the Realtek RTL2832U, is a subcomponent of OsmoSDR and described at `http://sdr.osmocom.org/trac/wiki/rtl-sdr`.

2.1 Installing the gnuradio tools

The `gnuradio` environment is quickly evolving, especially when using the tools needed for receiving RF signal with the digital television receiver dongles. Compiling the latest release of the software from the source available of the web is thus advisable. This task has been automated thanks to a well written script available at `http://www.sbrac.org/files/build-gnuradio`. However, a surprising aspect of the compilation of `gnuradio` should be emphasized : rather than defining the requirements to achieve a given targetted set of functionalities, including or excluding parts of the gnuradio software is automatically selected at configuration time by searching for the available libraries already installed on the system. Thus at first the compilation excludes most functionalities of gnuradio, and only by installing one after the other the missing library dependencies (`libfftw3-dev`, Python modules) can all the required components be built, except for the uhd related tools which are not needed. One significant part of the compilation process is to build the graphical user interface `gnuradio-companion`, associated with `gnuradio`, which will be intensively used when assembling signal processing blocs and for automating the generation of python code.

2.2 The RTL2832U as an oscilloscope

Having built the gnuradio related tools and more specifically `gnuradio-companion`, the issue of the behavior of the DVB reception dongle EZCAP can be tackled. The first observation is that the 75 Ω television antenna is hardly usable with most amateur radio antenna and the classically available RG58 BNC-ended cables we will be using for our own experiments : we first remove this connector and replace it with a female BNC connector.

In order to understand the distinct functionalities provided by the analog E4000 chip on the one hand, and the analog to digital converter followed by USB communication RTL2832U on the other hand, we will inject a periodic sine wave signal on parts of the circuit most obviously identified as the balanced I and Q (*i.e.* I+ and I- as well as Q+ et Q-). In addition to this hardware modification, we generate the most simple signal processing graph under `gnuradio-companion` : a data source as the `OsmoSDR Source` bloc (accessed in the `OsmoSDR` menu within the functions displayed at the right of `gnuradio-companion`) – this menu is missing if the tool compilation was not done correctly – ad a sink as as graphical oscilloscope found in the `WX GUI Widget` menu (`WX GUI Scope Sink` bloc).

We observe on Fig. 1 that the I and Q channels indeed exhibit a periodic sine-shaped signal, in agreement with the injected signal, at a frequency of 200 kHz and sampled at about 2 MS/s (10 points/period). The quartz internal to the EZCAP and defining the converter sampling rate, and the clock signal defining the frequency synthesizer output frequency, agree to within 2 parts per million (2 ppm), since the output frequency of the synthesizer had to be set to 199999.720 Hz for the sampling to be exactly 10 points/period at 2 MS/s sampling rate (lacking a trigger on an edge, this condition is visually observed when the oscilloscope sine output of `gnuradio-companion` stops moving horizontally with a rate equal to the inverse of the frequency difference). Such an accuracy is impressive considering how simple the electronic board is, exploiting only an industrial grade quartz resonator to clock the digital electronics.

We have stated that the RTL2832U sampling rate is increased by a factor of $2.5\times10^6/48000 \simeq 50$ with respect to that of a sound card. This value should not hide, though, the fact that the 8-bit RTL2832U measurement dynamics is poor : converting to a logarithmic value, the dynamics of $20\times\log_{10}(2^8) = 48$ dB is hardly acceptable considering that most radiofreqency signals exhibit often dynamics in the 60 dB range. A preliminary gain control stage is thus necessary, either with automatic or manual gain control as is possible with the E4000. Oversampling by a factor of 50 does not compensate for the poor resolution of this converter with respect to the performance of a sound card. Indeed, it is well known [6] that by applying a sliding average over N sampled points, the standard deviation on these filtered samples (assumed to be polluted by uniformly distributed noise) decreases as $\sqrt{N}$. Thus, the sampling rate must

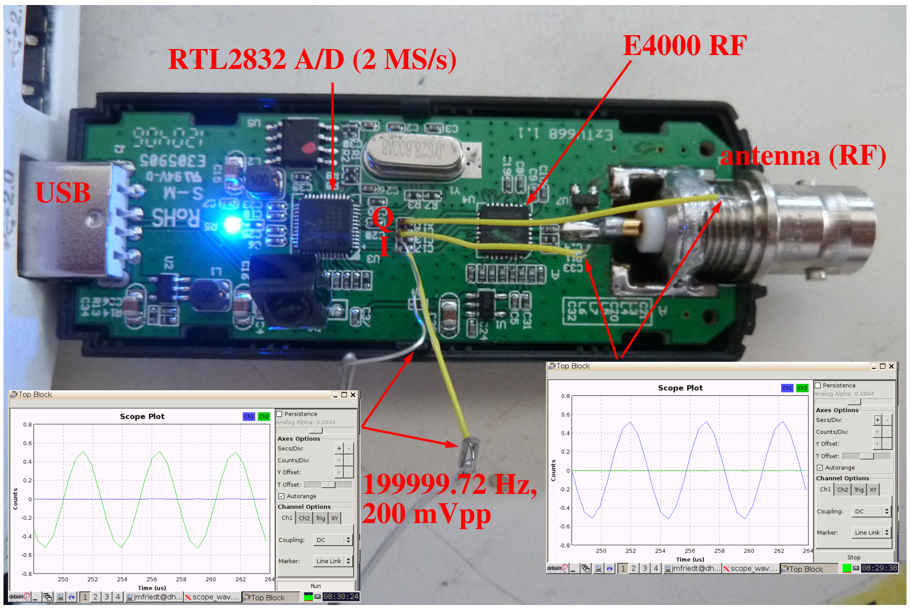


FIGURE 1 – Using a DVB receiver as an oscilloscope providing more than 2 MHz bandwidth.

be multiplied by 4 for the noise level on the acquired signal to be divided by 2 and hence add one significant bit to the analog to digital converter resolution.

[6] provides the generic mathematical expression of this statement by noting that for a sampling frequency f_{se}, the resolution improvement (p bits) obtained by a sliding average with respect to a sampling rate at f_e is

$$f_{se} = 4^p \times f_e$$

or, by converting to a logarithmic scale, $10 \log_{10} (f_{se}/f_e) = 10 \log_{10}(4) \times p$. Hence, noting that $10 \log_{10}(4) \simeq 6$, we indeed find [3] that the resolution gain of p bits for an over-sampling rate of f_{se}/f_e is $1/6 \times 10 \log_{10} (f_{se}/f_e)$. In any case, we are far from reaching the 16-bit resolution sound card, since $10 \log_{10}(50)/6 \simeq 3$ or a total of 11 bits on the analog to digital converter operating at the same frequency than a sound card.

This short calculation aims at reminding that sampling rate is not necessarily the most important characteristics of an analog to digital converter, depending on the application. In this case, the more than 2 Msamples/s is only useful for analyzing wide bandwidth signals with reduced dynamics. If such a large bandwidth is not needed, then a sound card will be more efficient for sampling signals with an increased dynamic range.

All computations will be performed on an eeePC701 netbook computer (630 MHz CPU speed, 1260 bogomips as provided by GNU/Linux) and thus do not require significant processing or memory resources.

3 First steps : using gnuradio-companion

The targeted signals we wish to recover will be much less powerful than the commercial frequency modulated (FM) broadcast radio. Most significantly, we shall be interested in receiving images from satellites operating around a 137 MHz frequency band. Most of the signals we will be interested in happen to fall within this value – 88 to 108 MHz for commercial FM, 108 to 137 MHz for air traffic, 154 MHz for *packet radio* – and we will thus use in all cases a dipole antenna tuned to resonate at 137 MHz.

In order to keep the mobility of the small radiofrequency receiver dongle connected to the USB port, the antenna should fit in a backpack once dismantled. 8 mm diameter hollow copper tubing and cut as 55 cm long ($300/137/4 \simeq 0,55$ with 300 m/μs the velocity of an electromagnetic wave propagating in vacuum and 137 the considered frequency in MHz) segments, finishing with 4 mm diameter "banana" connectors, fit these requirements. This rugged setup is conveniently assembled on a plastic holder made of 4 mm female connectors to which an as short as possible RG58 coaxial cable is soldered, ended with a BNC connector (Fig. 2).

We might try and add a low noise amplifier (LNA) Hittite HMC478 (5 V power supply as provided by a second USB port, 2 dB noise factor) between the antenna and the EZCAP receiver, although the improvement brought by such a setup still remains to be demonstrated.

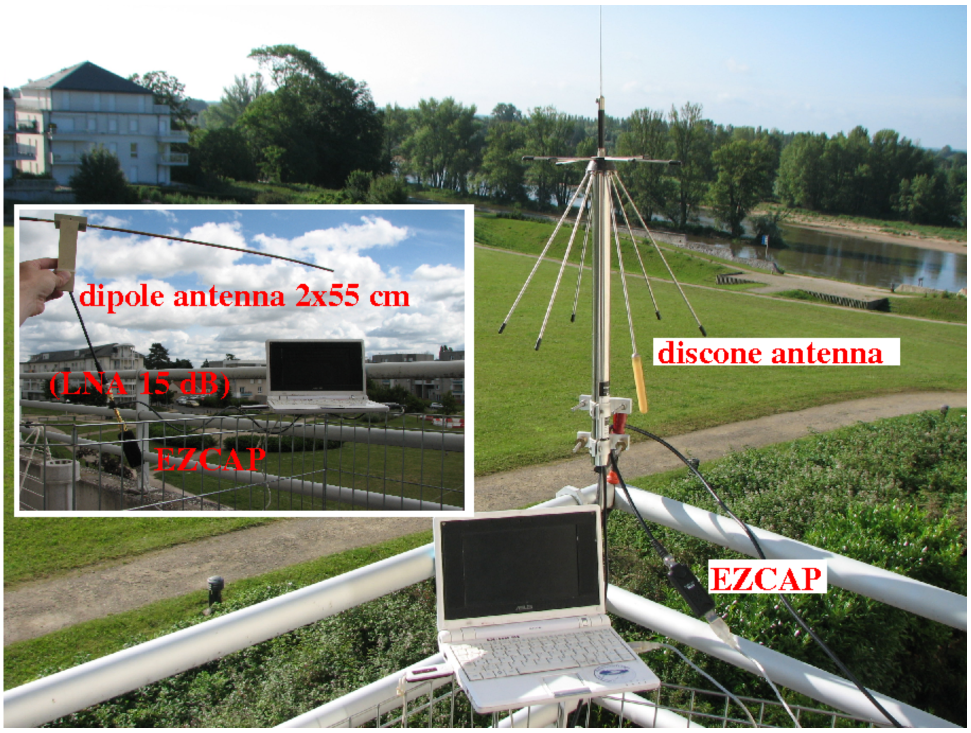


FIGURE 2 – Two antenna setups tested with the EZCAP receiver : one dipole antenna easy to store in a backpack when listening to frequencies around the 137 MHz range, and otherwise a wideband dipole antenna. As important as the antenna configuration, the antenna must be located in an environment free of obstacles. Here, a balcony located at the third floor of a building oriented eastward is usable for receiving commercial FM and planes, but too cluttered for receiving satellite images.

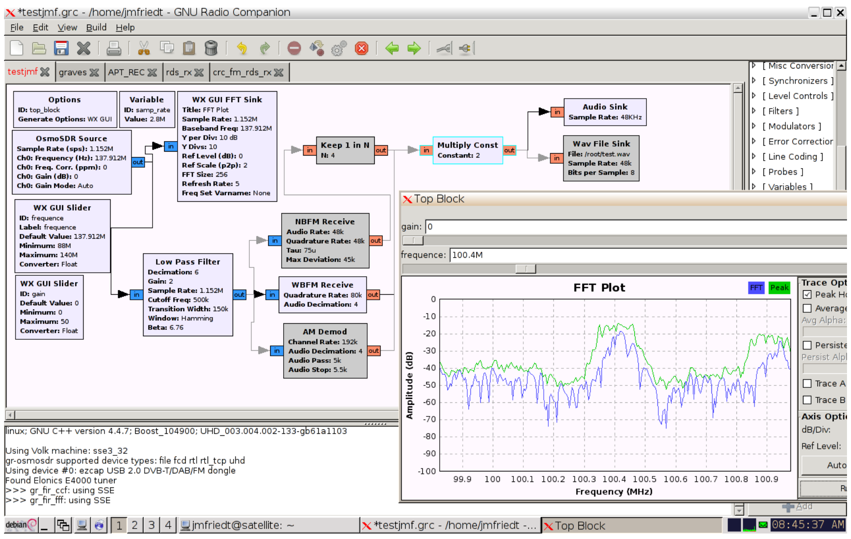
In the case of ADS-B operating around 1090 MHz, the a wideband Diamond Antenna D190 discone antenna was used since sold as compatible with reception in the 100 to 1500 MHz range (Fig. 2).

3.1 Commercial Frequency-Modulated (FM) broadcast

The most simple test for checking the operation conditions of the receiver is the commercial FM band since emission is continuous with a strong signal found between 88 and 108 MHz in Western Europe. Signals in this frequency range are qualitied as Wideband (WFM) since the spectral width allocated to each station is large (180 kHz). The channels have thus been defined with a spacing of 200 kHz 200 kHz [9, p.138].

Receiving commercial broadcast FM is useful for making sure the installation is operational : graphical user interface `gnuradio-companion`, OsmoSDR source block (as was used previously in the oscilloscope example), WFM demodulator block and data sink through the sound card on the one hand (after demodulation of the audio signal) and spectral analysis of the recorded data (before demodulation) in order to identify the emission frequencies of nearby channels on the other hand.

This example provides the opportunity to introduce the most significant requirement when assembling processing blocks : *data rate continuity must be propagated along the processing chain*, and thus an output



Options ID: top_block; Generate Options: WX GUI

Variable ID: samp_rate; Value: 2.8M

WX GUI FFT Sink Title: FFT Plot; Sample Rate: 1.152M; Baseband Freq: 137.912M; Y per Div: 10 dB; Y Divs: 10; Ref Level (dB): 0; Ref Scale (p2p): 2; FFT Size: 256; Refresh Rate: 5; Freq Set Varname: None

OsmoSDR Source Sample Rate (sps): 1.152M; Ch0: Frequency (Hz): 137.912M; Ch0: Freq. Corr. (ppm): 0; Ch0: Gain (dB): 0; Ch0: Gain Mode: Auto

WX GUI Slider ID: frequence; Label: frequence; Default Value: 137.912M; Minimum: 88M; Maximum: 140M; Converter: Float

WX GUI Slider ID: gain; Default Value: 0; Minimum: 0; Maximum: 50; Converter: Float

Low Pass Filter Decimation: 6; Gain: 2; Sample Rate: 1.152M; Cutoff Freq: 500k; Transition Width: 150k; Window: Hamming; Beta: 6.76

NBFM Receive Audio Rate: 48k; Quadrature Rate: 48k; Tau: 75u; Max Deviation: 45k

WBFM Receive Quadrature Rate: 80k; Audio Decimation: 4

AM Demod Channel Rate: 192k; Audio Decimation: 4; Audio Pass: 5k; Audio Stop: 5.5k

Keep 1 in N N: 4

Multiply Const Constant: 2

Audio Sink Sample Rate: 48KHz

Wav File Sink File: /root/test.wav; Sample Rate: 48k; Bits per Sample: 8

gain: 0

frequence: 100.4M

```
linux; GNU C++ version 4.4.7; Boost_104900; UHD_003.004.002-133-gb61a1103

Using Volk machine: sse3_32
gr-osmosdr supported device types: file fcd rtl rtl_tcp uhd
Using device #0: ezcap USB 2.0 DVB-T/DAB/FM dongle
Found Elonics E4000 tuner
>>> gr_fir_ccf: using SSE
>>> gr_fir_fff: using SSE
```

FIGURE 3 – Example of signal processing chart for decoding signals broadcasted in the commercial FM (88-108 MHz) band, or in the frequency band allocated to communication with planes – above the commercial FM band from 108 to 137 MHz – in which the modulation is on the amplitude (AM). The narrowband FM demodulation block did not provide signficantly different results compared to the wideband FM (WFM), but is included to demonstrate the use of the decimation of its output by keeping only one every 4 samples (and thus provides a sample rate compatible with the requirements of the sound card). The spectrum of the sample signal, visible bottom right, extends from $-f_e/2$ to $f_e/2$ with $f_e = 1152$ kHz the sampling rate. The peak associated with the channel being recorded is obvisously visible (100.4 MHz) as well as a peak around the neighbour station (100.9 MHz), which does not interfere with the demodulation algorithm thanks to the low-pass filter located after the spectrum analyzer.

block must neither provide too many data which would not be processed in time by the receiver block, nor make the next block wait for new data. Hence, in the example displayed in Fig. 3, starting from the end of the processing chain (the data sink is the sound card of the personal computer), we *must* provide a continuous data rate at the selected sampling rate of 48 kHz. Since we observe that the WFM demodulation block performs a decimation by a factor of 4 (*i.e.* generates 4 times fewer data than received), the input rate of this block must be $4 \times 48 = 192$ kHz. The low pass filter performs a decimation by a factor of 6, so the input rate must be 192 $\times$ 6 =1152 kHz, which is indeed the value found un the sampling rate of this block. Since the OsmoSDR source block is directly connected to the low-pass filter, its sampling rate is also selected to be 1152 kS/s. One limitation of the OsmoSDR is a lower sampling rate range limit of a few hundred kHz : a sampling rate between 1 and 2.5 MHsamples/s should be selected, possibly followed by a low-pass filtering decimation block if the next processing steps only require low bandwidth data rates. The reader is encouraged to change sampling frequencies or decimation factors on such a processing chart to assess the effect of these erroneous parameter values.

One extension of WFM signal reception provided as part of the examples of `gnuradio-companion` is the decoding of RDS (*Radio Data System*) [10] which provides an information in digital format concerning the channel being listened at (name of the broadcast station, frequencies propagating the same program in the surroundings, kind of program being broadcasted, possibly traffic information ...). This example is interesting because it emphasizes how flexible software decoding of the transmitted radiofrequency signals is. Indeed, the same dataset is on the one hand decoded as an audio signal, and on the other hand as digital information, with no need to change the hardware (Fig. 4). Furthermore, notice that the author of the RDS decoder has implemented an FM demodulator as would be done with discrete analog components, by controlling a phase locked loop (PLL) with the output of a low pass filtered radiofrequency signal, rather

10. `https://cgran.org/wiki/RDS`. Be aware that, due to a misbehaviour of the `-fvar-tracking-assignement` option of some of the latest releases of GCC, the `-g -O2` optimization options must be disabled during the compilation of the RDS decoding block (*i.e.* keep the default), in order to avoid a compilation failure after all the available memory has been exausted.

than using the ready-made processing block found with `gnuradio-companion`. The various bandpass filters are designed to select which information is processed in the various audio-frequency bands (sound or RDS).

We here provide below a few examples of digital data transmitted on the commercial FM band next to the audio broadcast, illustrating some of the messages sent through this communication medium.

```
00A (BASIC) - PI:F211 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:17)
==>  RTL   <== -TP-  -Music-STEREO - AF:
00A (BASIC) - PI:F211 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:17)
==>  RTL   <== -TP-  -Music-STEREO - AF:
00A (BASIC) - PI:F211 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:17)
==>  RTL   <== -TP-  -Music-STEREO - AF:104.00MHz
00A (BASIC) - PI:F211 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:17)
==>  RTL   <== -TP-  -Music-STEREO - AF:92.40MHz, 93.90MHz
00A (BASIC) - PI:F211 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:17)
==>  RTL   <== -TP-  -Music-STEREO - AF:95.00MHz, 97.50MHz
00A (BASIC) - PI:F211 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:17)
==>  RTL   <== -TP-  -Music-STEREO - AF:101.20MHz, 101.50MHz
00A (BASIC) - PI:F211 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:17)
==>  RTL   <== -TP-  -Music-STEREO - AF:102.20MHz, 103.20MHz
00A (BASIC) - PI:F211 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:17)
@@@@@ Lost Sync (Got 46 bad blocks on 50 total)
@@@@@ Sync State Detected
@@@@@ Lost Sync (Got 46 bad blocks on 50 total)
@@@@@ Sync State Detected
00A (BASIC) - PI:F219 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:25)
==>  IRL   <== -TP-  -Music-STEREO - AF:99.60MHz, 99.80MHz
00A (BASIC) - PI:F219 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:25)
==>  IRL   <== -TP-  -Music-STEREO - AF:99.60MHz, 99.80MHz
00A (BASIC) - PI:F219 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:25)
==>  IRL N <== -TP-  -Music-STEREO - AF:97.80MHz, 98.40MHz
00A (BASIC) - PI:F219 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:25)
==> VIRL N <== -TP-  -Music-STEREO - AF:100.40MHz
@@@@@ Lost Sync (Got 47 bad blocks on 50 total)
@@@@@ Sync State Detected
@@@@@ Lost Sync (Got 49 bad blocks on 50 total)
@@@@@ Sync State Detected
00A (BASIC) - PI:FC67 - PTY:None (country:EG/FR/NO/BY/BA, area:Regional 9, program:103)
==>  GRAY  <== -TP-  -Music-STEREO - AF:100.10MHz
00A (BASIC) - PI:FC67 - PTY:None (country:EG/FR/NO/BY/BA, area:Regional 9, program:103)
==>  GRAY  <== -TP-  -Music-STEREO - AF:97.10MHz
02A (RT) - PI:FC67 - PTY:None (country:EG/FR/NO/BY/BA, area:Regional 9, program:103)
Radio Text A:  RADIO    STAR  TOUS LES  HITS  BESANCON  106.6   GRAY    100.2
00A (BASIC) - PI:FC67 - PTY:None (country:EG/FR/NO/BY/BA, area:Regional 9, program:103)
==>  10AY  <== -TP-  -Music-STEREO - AF:100.70MHz, 87.80MHz
```

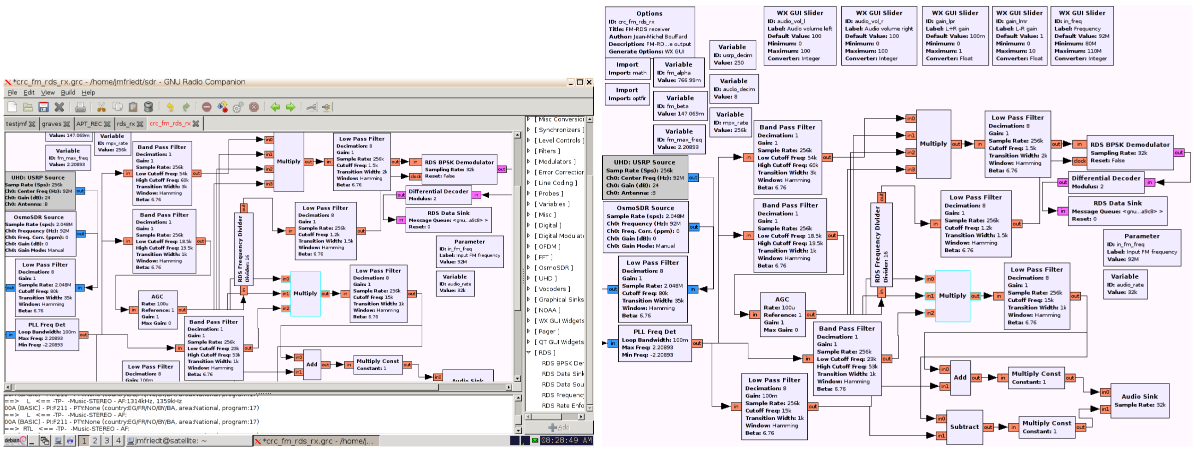
The selected sentences illustrate on the one hand the AF field (*alternative frequencies*) thanks to which a radiofrequency receiver knows what channel to lock on when the current link budget becomes too weak, and on the other hand the RT field (*radio text*) in which a free text containing up to 64 characters is transmitted. We have not had the opportunity of recording TA (*traffic announcement*) messages, probably one of the most useful application of this digital message broadcast over the commercial FM band.

One last example illustrates the transfer of the current time (CT field) as well as the free text field (RT) to display the music title being broadcasted :

```
04A (CT) - PI:F221 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:33)
Clocktime: 10.08.2012, 17:27 (+2.0h)
02A (RT) - PI:F221 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:33)
Radio Text A: Mozart : Concerto pour piano n 14:1er mvt
```

3.2 ADS-B

ADS-B (*Automatic Dependent Surveillance-Broadcast*) is a digital communication format between planes and ground initiated by a request emitted by a RADAR requesting a position information from an aircraft fitted with a GPS receiver. From the ground, a receiver (us) can listen to the answer of the plane to the request : this broadcast is performed at 1090 MHz. Thus, ADS-B provides a complementary



```
==>   L   <== -TP-  -Music-STEREO - AF:1314kHz, 1359kHz
00A (BASIC) - PI:F211 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:17)
==>   L   <== -TP-  -Music-STEREO - AF:
00A (BASIC) - PI:F211 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:17)
==>  RTL   <== -TP-  -Music-STEREO - AF:
```

FIGURE 4 – Left : screenshot of the RDS decoder running. Notice the messages being displayed in the terminal on the bottom of the window (the same messages are displayed in the terminal from which the `gnuradio-companion` application was launched), and on the right-menu the RDS related functions which were added by compiling the blocks provided at `https://cgran.org/wiki/RDS`. Right : complete processing chain schematic as found at `http://mmbtools.crc.ca/content/view/45/73/#fm_rds_rx` but with a few options requiring too much processing power for an Asus eeePC 701 removed.

information to the passive reflection of the incoming RADAR electromagnetic wave by the plane : instead on solely relying on the time of flight and antenna directivity and orientation when the probe pulse is generated, the position of the plane is also given by its GPS coordinates.

Decoding this protocol has been implemented as a script using `gnuradio` libraries, available at `https://www.cgran.org/wiki/gr-air-modes` and described in details in the associated slides [10].

Combining the information gathered by multiple such ADS-B receivers aims at mapping the position of planes in the world independently of Air Traffic Control, as done for example at `http://www.radarvirtuel.com/`. The software provided by N. Foster for `gnuradio` also outputs KML files compatible with Google Earth and Google Maps (Fig. 5).

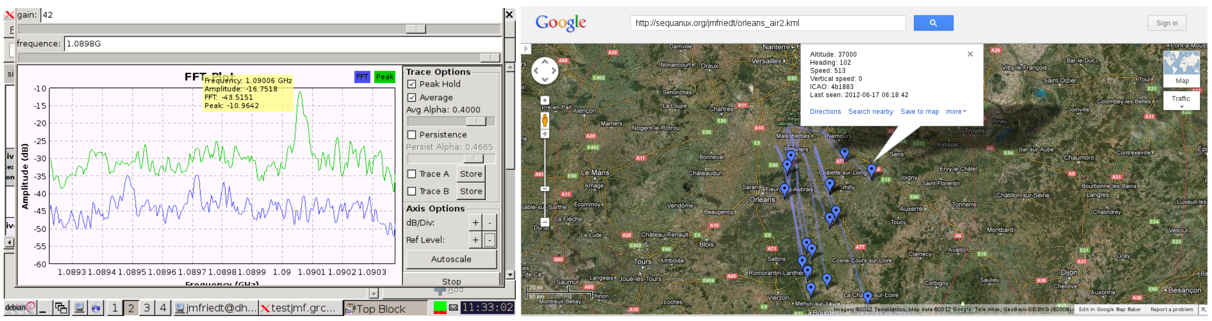


FIGURE 5 – Left : spectrum acquired while tuning to the ADS-B transmission frequeny, with a central frequency slightly offset with respect to the targeted frequency. Right : map of the planes tracked with a receiver located close to Orléans (France), using an antenna located on a balcony with a view towards the East only.

4 Decoding digital modes

Using existing functionalities is hardly of any interest to the developer who is quickly bored with the modules available on the project archive associated with `gnuradio` CGRAN [11]. Our true aim is to understand the demodulation methods in order to adapt these methods to strategies not yet available, and

11. `https://cgran.org`

thus use the `gnuradio` opensource environment as a flexible prototyping tool. The opensource aspect of the project is a core aspect of the learning capability by reading source code written by others [12]. Three digital communication modes will be of interest : FSK as used in radiomodems, AFSK as used in amateur or commercial packet radio, and the ACARS protocol used by civil aircraft (and thus military since compatible with the civilian network). This protocol is older but considered more "interesting" than ADS-B since it carries messages from the crew to ground or concerning the status of the plane.

Although all these modes were designed with bandwidths compatible with sound card demodulation, the digital broadcast receiver (DVB) provides a solution which prevents buying a radiofrequency scanner. Although a scanner is a flexible tool with much better sensitivity and the USB dongle, its cost might prevent the amateur from getting into digital mode decoding due to the significant financial investment. Furthermore, digital decoding allows for dynamically adapting filter characteristics by only being limited by available processing power, whereas a scanner only provides a finite number of decoding modes and usually only 2 or 3 filter bandwidths implemented at the hardware level and thus not tunable by the user.

4.1 Modulation strategies and signal processing

An introduction to the basics of digital signal processing for modulation decoding is beyond the reach of this document. In order to help the reader who is not familiar with the terms used throughout this document, let us notice that a periodic signal $s(t)$ is characterized by its time t behavior through three quantities : its amplitude A, its frequency f and its phase φ by following $s(t) = A(t) \times \sin(2\pi f(t) + \varphi(t))$. These three quantities might be themselves variable with time, individually or simultaneously, for coding a transmitted signal : $A(t)$ means an amplitude modulation (AM when talking of analog modulation, or ASK – *Amplitude Shift Keying* – when considering digital modes), $f(t)$ for a frequency modulation (FM for analog transmissions, or FSK – *Frequency Shift Keying* – for digital modes), and $\varphi(t)$ for phase modulation (PSK – *Phase Shift Keying*) [11]. Each one of these modulation modes is characterized by its spectral occupation, robustness to various noise sources and ease of use yielding educated guesses of the communication parameters depending on the targeted application. One particular we will discuss later is AFSK – Audio Frequency Shift Keying – in which coding does not directly modify the phase or frequency of the carrier, but an audio frequency variation of the carrier is applied. Such a modulation scheme was specifically developed for transferring data over communication channels designed for voice (telephone in the case of modems, or radiofrequency transceivers) which only require an output spanning the audio frequency range (typically 1000-5000 Hz).

Based on this introduction concerning digital signal processing, we conclude that demodulation is a matter of extracting one of these parameters in order to plot its time evolution and thus extract the transmitted data. In the most simple case of ASK, the receiver is locked to a given carrier frequency (possibly with a feedback loop to compensate for the drift between the emitter oscillator and the receiver oscillator), and a low pass filter for envelope detection. The amplitude is then representative of the transmitted bit value. FSK is slightly more complex since we use the control signal of the local oscillator phase locked on the received signal, but its application is made trivial by the availability under `gnuradio-companion` of the `Modulators` $\rightarrow$ `WBFM receive` processing block.

Demodulating a signal first requires the generation of the I (*In-phase*) and Q (*quadrature*) signals as described earlier (Fig. 6) : the incoming signal (called RF – RadioFrequency) is mixed with a local reference (called LO – Local Oscillator) in order to generate $I = A(t)\sin((RF - LO) \times t)$ and $Q = A(t)\cos((RF - LO) \times t)$. Then $I + j \times Q = A(t)\exp(j \times 2\pi(RF - LO) \times t)$ with $j^2 = -1$ [12] (notice that sin and cos are out of phase by 90^o, so that a practical implementation of the I/Q demodulator uses the mixing with a direct LO signal and a 90^o phase shifted copy of this signal).

These two signals are available at the output of the Elonics E4000 chip after the user has configured the LO frequency. One simple representation of the concept of modulation is as just described is summarized in the constellation diagram [12, p.11]. This chart plots along the abscissa the I component and along the ordinate the Q component as defined earlier, and provides a description in the complex plane whose properties are well known : the distance of a point to the origin represent the modulus of the complex number (in our case $A(t)$) and the angle between the abscissa axis (I) and the point in the complex plane represents the phase $\varphi(t)$. Thus, a constellation diagram represents in the complex plane (I, Q)

12. `http://gnuradio.org/redmine/projects/gnuradio/wiki/OutOfTreeModules`

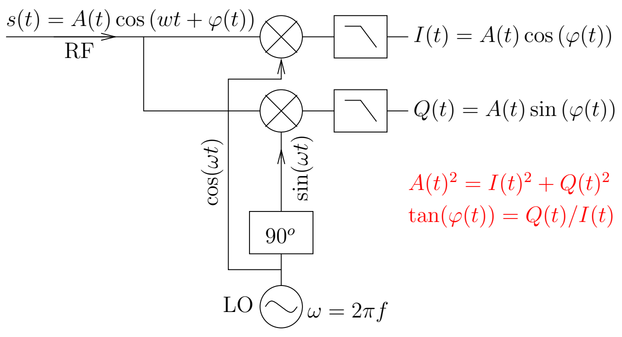

$s(t) = A(t)\cos(wt + \varphi(t))$
RF
$I(t) = A(t)\cos(\varphi(t))$
$Q(t) = A(t)\sin(\varphi(t))$
$\cos(\omega t)$
$\sin(\omega t)$
$A(t)^2 = I(t)^2 + Q(t)^2$
$\tan(\varphi(t)) = Q(t)/I(t)$
90^o
LO
$\omega = 2\pi f$

FIGURE 6 – Principle of IQ demodulation for extracting the two quantities representative of the input radiofrequency signal $s(t)$ assuming its frequency f (or angular frequency $\omega = 2\pi f$) is known, namely its amplitude $A(t)$ and its phase $\varphi(t)$, both of which can be used to code information when varying over time t. The cutoff frequency of the two low-pass filter following the mixers defines the bandwidth of the detector (a core aspect of the communication bandwidth since it defines the rate at which the symbols defined by $A(t)$ and/or $\varphi(t)$ can change).

the various possible codes. A phase modulation is represented as points distributed along a circle of constant radius centered on the origin, while an amplitude modulation is represented as points whose distance to the origin vary. Frequency modulation is a particular case since the frequency is the derivate of the phase and thus points in the complex plane rotate along a circle centered on the origin.

We will, in the following sections, illustrate some of these modulation schemes on practical examples used on commercially available emitters whose transmitted information we wish to decode. The Semtech XE1203F emits data following a frequency modulation, packet radio emits audio-frequency modulated signals, and the protocol used by plane to communicated with ground stations called ACARS is amplitude modulated.

4.2 FSK on the Semtech XE1203F

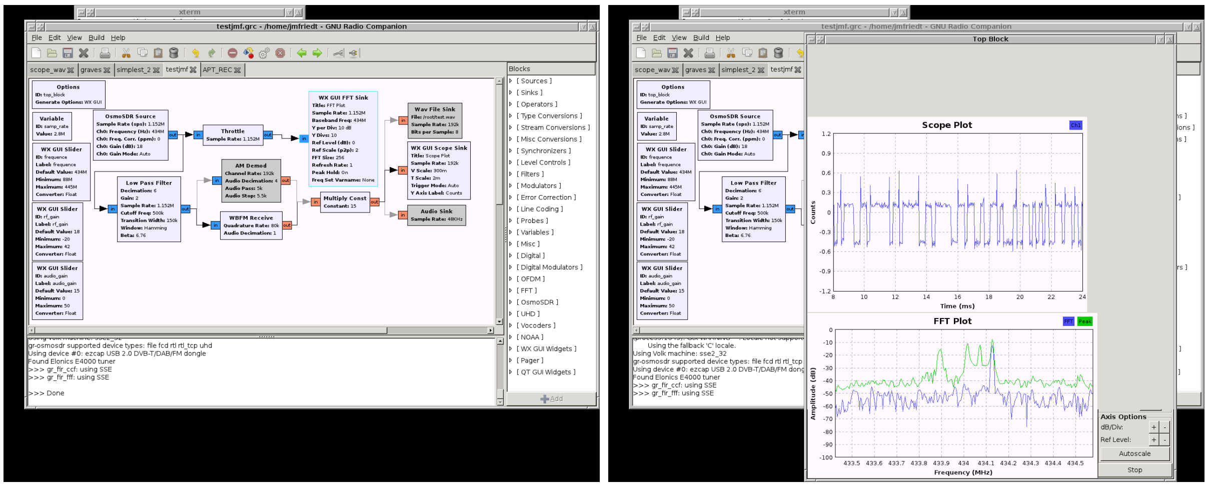
The Semtech XE1203F radiomodem [13] is a half-duplex transceiver (either it emits, or it receives) able to modulate the frequency of an oscillator (FSK) in order to code the two possible states during digital data transmission. The carrier frequency is programmed with a 500 Hz resolution : we select 434 MHz, right in the middle of the Industrial, Scientific and Medical (ISM) band. The frequency modulation excursion is software configurable – we select 55 kHz, and the digital data flow rate is defined by the baudrate of the microcontroller (in our case a STM32F103) asynchronous interface (UART) generating the bit stream through the DATA and DATAIN pins of the radiomodem. Such pin connections seem to be the most convenient means of communicating between a microcontroller and the XE1203F radiomodem, even though the start and stop bits yield some subtleties in the way the radiomodem handles part of this information (pattern encoding and decoding).

Using the WFM demodulator, well suited for such a large excursion as 55 kHz, yields the signal shown on Fig. 7 : we observe obvious state transitions representing the bit stream on the oscilloscope connected to the demodulator output. FSK requires no specific processing step to recover these bits since the radiofrequency oscillator is directly modified by the digital signal to be transmitted : the demodulator provides a low frequency signal proportional to the error signal from the phase locked loop, and thus a voltage proportional to the frequency encoding each bit.

Furthermore, the time interval between two transitions is in agreement with the 4800 bits/s data rate. The analysis of this dataset in order to extract a usable information requires a detailed understanding of the emission sequence implemented in the microcontroller following the Semtech datasheet instructions :

1. transmission starts with some oscillations between 0 and 1 called bit synchronizer for the receiver to lock on the emitted signal. Due to the start bit being at 0 level and the stop bit at 1, a sequence of alternating 0 and 1 is obtained by sending the value 0x55 (the least significant bit being sent first), *i.e.* the “U” character,

13. `http://www.semtech.com/apps/filedown/down.php?file=xe1203f.pdf`



```
gr-osmosdr supported device types: file fcd rtl rtl_tcp uhd
Using device #0: ezcap USB 2.0 DVB-T/DAB/FM dongle
Found Elonics E4000 tuner
>>> gr_fir_ccf: using SSE
>>> gr_fir_fff: using SSE

>>> Done
```

```
        Using the fallback 'C' locale.
Using Volk machine: sse2_32
gr-osmosdr supported device types: file fcd rtl rtl_tcp
Using device #0: ezcap USB 2.0 DVB-T/DAB/FM dong
Found Elonics E4000 tuner
>>> gr_fir_ccf: using SSE
>>> gr_fir_fff: using SSE
```

FIGURE 7 – Left : schematic of the blocks needed to decode the data stream transmitted by a FSK-modulated radiomodem (the same WFM demodulating block than used earlier for listening to the commercial broadcast FM is again used here). Right : oscilloscope output while recording the data stream. The two bit states are clearly visible and the decoding process will thus be easily automated.

2. we use the facility of coding the address of the emitting radiomodem, especially after observing that the surrounding radiofrequency noise is detected as an erroneous transmission when this functionality is not active (*Pattern recognition block*),
3. after detecting the pattern, the message itself is decoded.

We check for each transmitted byte that the last bit (stop bit) is at the level 1, otherwise we indicate an error has occurred in the decoding process. Checking the stop bit level partly solves some of the propagation channel noise issues which might indicate that a start bit was triggered (1 to 0 transition), noise which would hardly be correlated at the stop bit position without an actual data transmission.

In order to ease the understanding of a new coding scheme, the development strategy we adopt consists in always recording in a binary file (or WAV when adding a header including the data size and sampling rate) the data stream, then first applying the decoding algorithm written for GNU/Octave to this dataset, before translating this decoding script to C, and finally encapsulating this algorithm in the framework required by a `gnuradio` block (section 5) before finally processing a "real" data stream provided by the radio receiver.

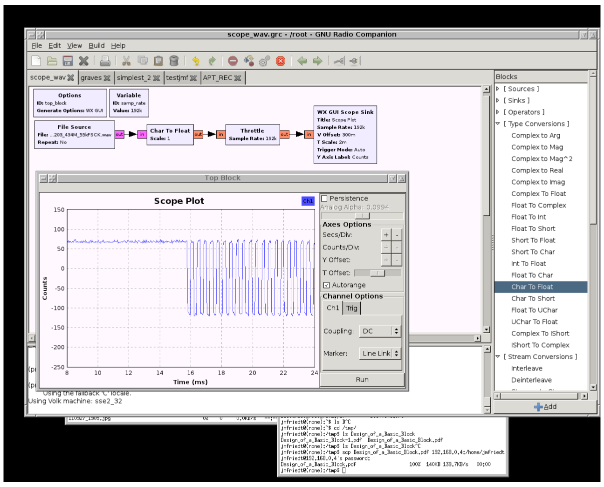
Fig. 8 displays one example of processing block assembly designed to validate the recorded dataset and display in oscilloscope mode the values stored in a binary file (we have throughout this document used the default single channel recording – as opposed to dual channel stereo – with 8-bit/datum).

4.3 AFSK decoding

Audio Frequency Shift Keying encodes the two possible bit states with audio frequencies, typically between 1000 and 1550 Hz for one state, and 2000 to 2500 Hz for the other. One particular case of interest is the *packet* mode which inherits from the older times of digital communications over analog telephone lines using modems.

Our interest in this modulation modes is twofold : on the one hand it is a digital communication mode widely used in the world of amateur radio enthusiasts, and on the other hand the Besançon (France) bus network Ginko uses modems transmitting such a protocol to locate its busses (emissions at 154.150 and 154.154 MHz).

The most common encoding scheme of *packet radio* meets the requirements of Bell 202 by encoding the two bit states with 2200 and 1200 Hz sine waves, and a transmission rate of 1200 bits/s. On the example displayed in Fig. 9, blue circles have been sample every 40 recorded points or, for a sampling rate of 48 kS/s, an information rate of 48000/40=1200 bits/s. Since various modulation schemes are available


scope_wav.grc - /root - GNU Radio Companion

Options
ID: top_block
Generate Options: WX GUI

Variable
ID: samp_rate
Value: 192k

File Source
File: ...203_434M_55kFSCK.wav
Repeat: No

Char To Float
Scale: 1

Throttle
Sample Rate: 192k

WX GUI Scope Sink
Title: Scope Plot
Sample Rate: 192k
V Offset: 300m
T Scale: 2m
Trigger Mode: Auto
Y Axis Label: Counts

Using Volk machine: sse2_32

FIGURE 8 – Block processing sequence to replay the data recorded in a binary file including a WAV header : notice the use of the `throttle` block to impose the rate at which data read from the file will be played.

(for example coding the bit states with 1300 and 2100 Hz sine waves as defined in ITU-V.23 [14]), we have decided to apply band-pass filters with a cutoff frequency around 1700 Hz, yielding compatibility with most of the modes described on the web.

The following GNU/Octave script introduces some of the first steps for decoding signals recorded with `gnuradio` following a narrow-band FM (NFM) demodulation in order to extract the values of the successive bits. A pair of band-pass filters centered around the two frequencies assumed to encode each bit state filter the incoming data stream, providing a flexible approach which can be adapted to any set of audio frequencies.

```
fe=48000;
deb=  338100;
fin=  488000;
SEUIL=200;

f=fopen(filename='120723_ginko.wav');d=fread(f,inf,'uint8');
d=d(deb:fin);N=fin(m)-deb(m);
f=linspace(0,fe,N);
d=d-mean(d);
```

Variable initialization – including the sampling rate `fe` – and reading a segment of binary data recorded using the tt wav output block of `gnuradio-companion` (8 bits/sample, single channel).

```
c=ones(30,1)/30;
h=firls(42,[0 500 1000 1500 1900 fe/2]/fe*2,[0 0 1 1 0 0]);
[H,freq]=freqz(h,1,512,fe);
y12=filter(h,1,d(1:N));
yc12=conv(abs(y12),c);yc12=yc12(15:end-15);plot(yc12,'c');hold on
```

The Finite Impulse Response (FIR) filter whose template is defined through the amplitudes in the second vector for frequencies defined in the former vector. In this example, the bandpass is from 1000 to 1500 Hz (amplitude 1) and any signal beyond these boundaries should be cut. A digital filter is always defined with frequencies normalized to the sampling rate. Finally, following the band pass filter, and low pass filter (averaging) is applied as a convolution by a rectangular window 30 samples long.

```
h=firls(42,[0 1700 2000 2600 4600 fe/2]/fe*2,[0 0 1 1 0 0]);
y24=filter(h,1,d(1:N));
yc24=conv(abs(y24),c);yc24=yc24(15:end-15);plot(yc24,'m')
```

This procedure is applied once more for a bandpass filter set between 2000 and 2600 Hz.

```
ind=[9160:40:48000];                      % http://wiki.ham.fi/AFSK.en
v1=find(yc12(ind)>yc24(ind));tout(v1)=1;  % 1300 Hz = 1 = mark
v2=find(yc12(ind)<=yc24(ind));tout(v2)=0; % 2400 Hz = 0 = space
res(1)=0;
for k=1:length(tout)
    if (tout(k)==0) res(k+1)=1-res(k); else res(k+1)=res(k);end
end
plot(ind,tout,'o');
```

`v1` and `v2` define the most probable bit state obtained by comparing the power output of both bandpass filters. As described at `http://n1vg.net/packet/index.php`, a 0 state (2400 Hz) encodes a bit state change with respect to the previous value, and a 1 (1300 Hz) copies the previous bit state in the

14. `http://wiki.ham.fi/AFSK.en`

current value. Selecting only one sample every 40 meets the requirement of the sampling rate (48 kHz) to the assumed bitrate (1200 bit/s) ratio.

```
binaire=reshape(tout(1:floor(length(tout)/8)*8),8,floor(length(tout)/8))';
code_asc=binaire(:,1)+binaire(:,2)*2+binaire(:,3)*4+binaire(:,4)*8+binaire(:,5)*16+binaire(:,6)*32+binaire→
    ↪(:,7)*64; % +binaire(:,8)*128;
printf('%02x  ',code_asc);
printf('\n');
```

Finally, the data are reorganized assuming 8 bit/sample, and the result displayed both as hexadecimal values (`%02x`) or as ASCII characters.

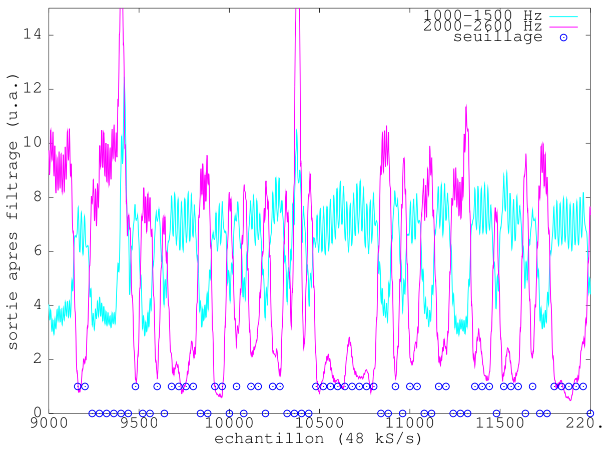


FIGURE 9 – Result of the decoding, using two band-pass filters, of the NFM (Narrow-band FM)-demodulated signals. The successive bit values are clearly visible, consistent with a 1200 bit/s datarate.

Obtaining the bit sequence is only among the first steps in understanding the content of the exchanged data : one must then be able to extract a usable information out of this bit stream, by identifying the baud rate (we have assumed so far, inspired by packet radio, a 1200 bit/s rate as observed on Fig. 9), the number of bits/data and whether parity bit(s) have been added at the end of each transmission, and finally the meaning of each byte. These issues have not yet been solved.

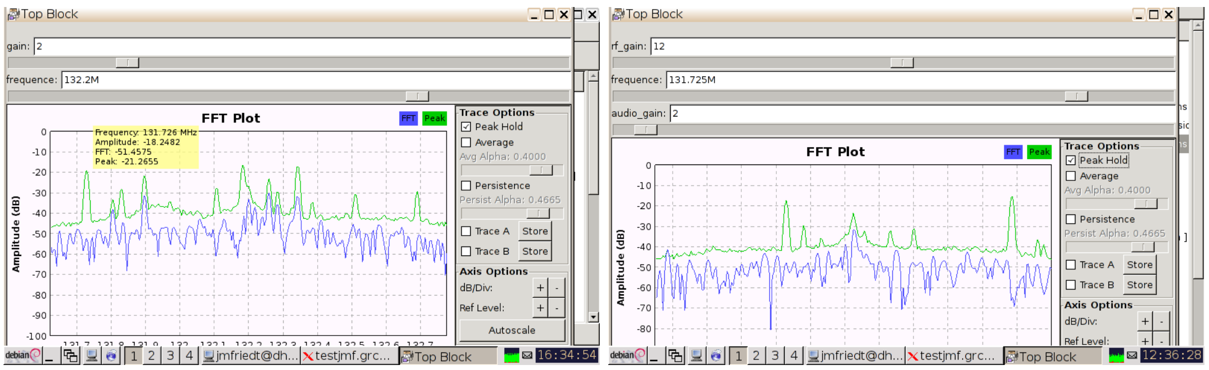
4.4 AM and the ACARS protocol

ACARS is a communication protocol internationally used by planes to transfer various information to ground during the flight, either automatically during well defined flight steps (taxi with data concerning the amount of fuel, during flight with data concerning the weather or estimated arrival time, during approach with the gate number and the status of the engines, after landing with the remaining weight of fuel, information concerning the crew or status of the computers) or manually with message concerning passengers.

Digital communication over Europe meeting the ACARS protocol is most usually performed at 131.725 MHz, or on the secondary frequencies of 131.525 MHz and 131.550 MHz. Considering the demodulation bandwidth of hardware associated with `gnuradio`, all these frequencies can be monitored simultaneously. For reasons we do not understand, all communications related to aeronautical bands are amplitude modulated (AM).

ACARS is a protocol old enough to be easy to decode, and is well documented at `http://files.radioscanner.ru/files/download/file4094/acars.pdf` or `http://www.tapr.org/aprsdoc/ACARS.TXT`. We learn from these document that two frequencies encode the two possible bit states of the transmitted data : 1200 and 2400 Hz are used to encode a bit stream transmitted at 2400 bit/second.

Few open implementations of ACARS exist : `acarsd` available at `http://www.acarsd.org/` is free but not opensource, while development of `acarsdec` at `http://sourceforge.net/projects/acarsdec/` and later `http://www.r-36.net/src/acarsdec/` seems to have stopped at the prototype level since 2007 (which still means it is worth reading for learning, but the provided example seems to be no longer functional and the person hosting the current code version seems not to know how to get the example to work).


gain: 2

frequence: 132.2M

Frequency: 131.726 MHz
Amplitude: -18.2482
FFT: -51.4575
Peak: -21.2655

Trace Options: [✓ Peak Hold] Average, Avg Alpha: 0.4000, Persistence, Persist Alpha: 0.4665

rf_gain: 12

frequence: 131.725M

audio_gain: 2

Trace Options: [✓ Peak Hold] Average, Avg Alpha: 0.4000, Persistence, Persist Alpha: 0.4665

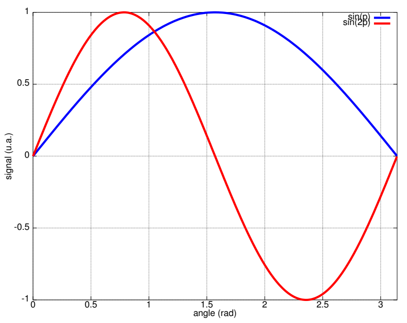
FIGURE 10 – Spectra in a aerial band ranging from 108 to 137 MHz, in which all communications are amplitude modulated (AM).

The selection of the modulation frequencies (1200 and 2400 Hz) with respect to the bit rate (2400 bit/second) might seem surprising at first, until the demodulator is actually implemented. The first naive approach, as used earlier in the case of packet radio when implementing two bandpass finite-impulse response (FIR) filters centered on 1200 and 2400 Hz, requires a fast sampling rate since even at 48 kHz (compatible with a sound card input), one period of a 2400 Hz sine wave only includes 20 samples, too few samples to implement an efficient filter with sharp edges, and most significantly a null coefficient at the 0 Hz frequency in order to remove the DC component which otherwise makes the use of a threshold complex (constant offset). This approach is functional but does not profit from the wise selection of the parameters we have just mentioned, considering the following aspects :

1. a bit rate proportional to the two encoding frequencies ensures the phase continuity between successive bits. Indeed, once the gate switching the bit output is synchronized with the audio oscillator, we are sure that the audio sine wave is always at 0 during a bit change,
2. encoding one bit state with half a wavelength of the audio signal and the other bit state with one full period allows to fully exploit the cross-correlation property when identifying the encoded bit.

This last point will be developed since it is a core aspect of a decoding scheme more efficient than the dual-band pass filter convolution approach. First of all, notice the following relationships which allows for an easy classification since the wrong bit state yields an integral with average value close to 0 (integral over one bit length, or 1/2400 s) while the detection of the right bit state yields a non-null average output, with an equal output power for both bit states :

- $\int_0^1 \sin(2\pi t)\sin(\pi t)dt \propto \int_0^1 (\cos(3\pi t)-\cos(\pi t))\,dt = sin(3\pi)-sin(0)-(sin(\pi)-sin(0))=0$
- $\int_0^1 \sin(2\pi t)\sin(2\pi t)dt = 1/2\times\int_0^1 (\cos(4\pi t)-\cos(0))\,dt = 1/2\times(sin(4\pi)-sin(0)+1)=1/2$
- $\int_0^1 \sin(\pi t)\sin(\pi t)dt = 1/2\times\int_0^1 (\cos(2\pi t)-\cos(0))\,dt = 1/2\times(sin(4\pi)-sin(0)+1)=1/2$



For those who have forgotten the basics of trigonometric rules, this calculation is a good opportunity to test the Wolfram website at `http://www.wolframalpha.com/` by requesting the computation of `int_0^1 sin^2(pi*x)dx`. Notice that without getting involved in the calculation, it is rather intuitive to realize that the integral over one period of the product of an even function with an odd function whose average values are null is null, and that the integral of the square of a function (all values being thus positive) is non-null (figure on the right).

We thus conclude that modulating one data value at $f_m = 1200$ Hz and at $2f_m = 2400$ Hz the other answers a need for efficient decoding schemes : the convolution of the recorded samples with $\sin(2\pi f_m t)$ generates a null average value if the data segment under investigation does not code the right state, and yields a value of 1/2 if the right bit status is met. Thus, once an initial synchronization of the incoming data stream with the local digital oscillator (sine wave) is performed – the reason for which the first 16-bytes provide an oscillating signal at 2400 Hz – the decoding is "simply" a matter of two products with the sequences $\sin(2\pi f_m t)$ and $\sin(\pi f_m t)$ in order to retrieve the successive bit values (Fig. 11).

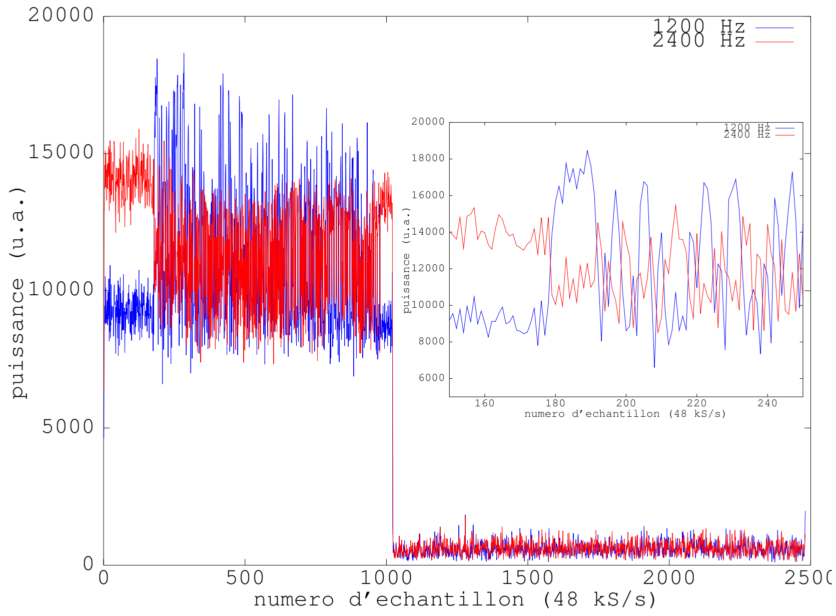


FIGURE 11 – Audio signals recorded during an ACARS transmission at 131.725 MHz after filtering at 1200 and 2400 Hz using the algorithms described in the text. Inserted to the right, a zoom on the first usable signals at the end of the receiver synchronization (sequence of multiple periods at 2400 Hz) : the identification of this starting point is a core requirement for the next steps of the decoding to operate properly on the whole sentence since it defines the first bit on which all the decoding is based.

Finally, having obtained a bit stream, its content remains to be interpreted : ACARS exploits a encoding in which *transitions* from one bit to another are notified, instead of coding directly the bit values. Thus, a 1200 Hz signal indicates that the bit state must change with respect to the previous value, while a 2400 Hz code indicates that the bit state remains constant [15].

An example of the prototyping of this algorithm under GNU/Octave is as follows :

```
function binaire=fft_decod(filename,deb,fin,seuil)
jmfdebug=0;
fe=48000;
f=fopen(filename);
d=fread(f,inf,'uint8');
d=d(deb:fin);
N=fin-deb;
```

We read the data segment included between the `deb` and `fin` indices from the file `filename`,

```
c=ones(60,1)/60;
dm=conv(d,c);dm=dm(60/2:end-60/2);
d=d-dm;   % remove sliding average over 3 periods
```

applies a sliding average (rectangular windows 60-sample long, or 1.25 ms for `fe` equal to 48 kHz),

```
t=[0:519]; % 2400 Hz at 48 kHz = 20 points/period *26
c2400x13=exp(i*t*2400/fe*2*pi); % search max of 13 periods at 2400 Hz
s=conv(c2400x13,d);
s=s(length(t)/2:end-length(t)/2);
[a,b]=max(real(s));   % max similarity of an offset b
% plot(d(b-260:b+260)/120,'g');hold on; plot(real(c2400x13),'r');
```

the keypoint for this algorithm to work properly is to synchronize the receiver on the oscillations generated by the emitter, and to do so we must maximize the cross-correlation between a reference signal (2400-Hz sine wave `c2400x13` sampled at `fe`) and the 2400-Hz synchronization sequence sent as part of the ACARS header. For a real time-symmetric signal, this cross-correlation is applied by running a convolution.

```
  b=mod(b,20)+5;  % rewinds by 2pi steps
  d=d(b+400:end); % good positioning is mandatory for the next steps
                  % would it be worth trying +/-1 ?
```

Having identified the starting point of the sequence for synchronizing the sine waves generated by the software with the recorded samples, we have selected to locate the cursor at the beginning of the

15. `leonardodaga.insyde.it/Corsi/AD/Documenti/ARINCTutorial.pdf`

recorded dataset by moving by steps of 20 samples, which is also equal to one 2400 Hz sine wave period (48000/2400). This step might not be useful, but prevents from locking on a long data sequence to multiple 2400 Hz periods in the middle of the ACARS message,

```
t=[0:19];  % 2400 Hz at 48 kHz = 20 points/period
c2400=exp(i*t*2400/fe*2*pi);
c1200=exp(i*t*1200/fe*2*pi);
s12=conv(c1200,d);
s24=conv(c2400,d);
% plot(d); hold on;plot(real(s12),'r'); plot(real(s24),'r');
```

We now look for individual bits, coded on a 2400-Hz period, or half a 1200-Hz period, and in all cases 20 samples,

```
t=[0:19];  % 2400 Hz dans 48 kHz = 20 points/period
fin20=floor(length(s12)/20)*20;
s12=s12(1:fin20);s24=s24(1:fin20);
rs12=reshape(abs(s12),20,length(s12)/20);
rs24=reshape(abs(s24),20,length(s24)/20);
rs12=sum(rs12);
rs24=sum(rs24);
```

as opposed to the previous case of processing the data generated by the XE1203F in which we only kept one out of 40 points to comply with the bit transmission rate, we will this time average the signal values within each bit to improve the signal to noise ratio (and thus the ability to decode a message within a noisy communication channel). To do so, we reorganize the two measurement vectors filtered with the sine wave sequences generated at 1200 and 2400 Hz shaped as 20-sample wide matrices (the sample length of one bit), and will sum the results to perform an average. `rs12` and `rs24` contain the data for identifying if a bit is most probably a 1 or a 0 et deserve being displayed in a preliminary debugging step (Fig. 11).

```
if (jmfdebug==1) plot(rs12,'bo-');end
if (jmfdebug==1) hold on;plot(rs24,'ro-');legend('1200','2400');end

seuil=max(rs24)*0.55;
l0=find((rs24+rs12)>seuil);   % only keep useful samples
rs12=rs12(l0);rs24=rs24(l0);
l1=find(rs24>seuil);l1=l1(1)
rs12=rs12(l1:end);
rs24=rs24(l1:end);
```

Finally, the sample vector is processed in order to generate the binary values : a threshold criterion defines which data is valid (signal above a given noise level, as above) and the comparison of the two filter outputs defines the bit value (convolutions, see below).

```
l=find(rs12>rs24);l=l(1)
rs12=rs12(l:end);
rs24=rs24(l:end);
pos12=find(rs12>rs24);
pos24=find(rs24>rs12);
toutd(pos12)=0;
toutd(pos24)=1;
```

The binary values are finally reorganized as bytes, following the method described earlier in the case of the radiomodem, with the input bit value defining whether the current output bit value is kept or changed.

```
n=1;
tout(n)=1;n=n+1;  % the first two 1s are forgotten since we sync on 1200
tout(n)=1;n=n+1;
for k=1:length(toutd)
  if (toutd(k)==0) tout(n)=1-tout(n-1); else tout(n)=tout(n-1);endif
n=n+1;
end
binaire=reshape(tout(1:floor(length(tout)/8)*8),8,floor(length(tout)/8))';
code_asc=binaire(:,1)+binaire(:,2)*2+binaire(:,3)*4+binaire(:,4)*8+binaire(:,5)*16+binaire(:,6)*32+binaire→
    ↪(:,7)*64;
checksomme=1-mod(sum(binaire(:,1:7)')',2); % verification
printf('%02x ',code_asc); printf('\n');
printf('%c',   code_asc); printf('\nCRC:'); printf('%d',checksomme-binaire(:,8)); printf('\n');
```

The decoded values are displayed as hexadecimal values, the associated ASCII code, and the parity but validation.

Summarizing all these ACARS sentence processing steps :

1. identify the beginning of the sentence found as a series of oscillations at 2400 Hz : we wish to find at least 13-periods of the 2400 Hz oscillations as the definition of the sentence beginning and start processing the acquired sequence,

2. identify the segments of bits equal to 1 (2400 Hz) and 0 (1200 Hz) by convolutions with one period of a sine wave of each period with the recorded stream. Only a single convolution is needed since the phase uncertainty (which would require 20 convolutions in this example in case of the 2400 Hz signal to identify among the 20 possible values which phase maximizes the cross-correlation) has already been solved during the previous step,

3. since the bit values represent a transition from one state to another and not the state itself, the bit stream resulting from the convolution remains to be converted to an sequence of ASCII characters. Notice that the documentation available at `http://www.pervisell.com/ham/raftmode.htm#I76`

is erroneous concerning the parity. It is also interesting to finally understand some of the strange entries in the ASCII table (`man ascii`) such as character 01 (beginning of header), 02 (beginning of text) or 03 (end of text) which are used by ACARS. Furthermore, coding bit transitions means that the first mistake in identifying a bit value will yield unreadable messages from then on. Thus, we will often obtain the identifier of the plane and flight at the beginning of the message, but obtaining the whole text of the message is challenging since dependent on the lack of bit identification error throughout the processing.

Applying a convolution between two datasets of lengths respectively M and N yields to an issue concerning the position of the origin. Indeed, the discrete digital implementation of the convolution c between u_i, $i \in [1..M]$ and v_j, $j \in [1..N]$, is

$$c(n) = \sum_{m=-\infty}^{+\infty} u_m \times v_{n-m}$$

with u and v equal to 0 outside of the intervals mentioned above. When going to the Fourier domain, the datasets u and v are complemented with 0 (zero-padding) in order to be of equal lengths (and reach the power of two closest to the initial number of samples in case of the Fast Fourier transform).

In the equation presented above, the integral with $\pm\infty$ bounds might be lengthy to compute, infinity being hard to reach. We will thus integrate, for two point series of length M and N, either between $M+N$ if the data series u is supposed to slide over the v series and thus generate a total of $M+N$ points, or a series of $\max(M,N)$ points if one of the series is complemented with 0s in order to match the length of both series. This latter approach is the one proposed in the C code translation developed below (section 5.2), since using the Fourier domain requires a dot product multiplication of the two sequences of equal length. The problem of locating the origin is thus a matter of the selected formalism, and is the main difficulty we met when translating GNU/Octave software to C (not to mention the various normalization conventions of the Fourier transform which require to compute new threshold values for the new implementation).

The GNU/Octave programme presented above identifies the beginning of the sentence (series of oscillations at 2400 Hz), displays the sequence of decoded hexadecimal values, their interpretation as characters following the ASCII code, and verifies the parity bit (even if this bit cannot act for correcting the propagated bit change value, it can be used at least to indicate that the decoded byte is no longer valid and might not be displayed). In the following result, we observe that all values have been correcty decoded except for the last 9 values, which are not relevant anyway since located after the ASCII character 0x03 (end of text).

We display as a first step the hexadecimal values of the decoded bytes (always beginning with the synchronization sequence 0x2b 0x2a 0x16 0x16 if decoding was successful), their interpretation as ASCII characters if within the set of character which can be displayed, and finally the parity bit validation (0 if the parity bit is consistent, ± 1 if an error occurs) :

```
binaire=fft_decod('acars_orleans.wav',101001+2.15e6,101001+2.15e6+40000,7000);

2b 2a 16 16 01 58 2e 47 2d 45 55 55 47 15 48 31 39 02 43 30 33 41 42 41 39 32 31 36 23 43 46 42
57 52 4e 2f 57 4e 31 32 30 36 33 30 30 38 33 33 30 30 33 34 30 30 30 30 30 36 4e 41 56 20 49 4c
53 20 32 20 46 41 55 4c 54 20 20 20 20 20 20 20 20 20 0d 03 6d 1d 7f 7f 7f 7f 7f 7f 7f
m*X.G-EUUGH19C03ABA9216#CFBWRN/WN12063008330034000006NAV ILS 2 FAULT
CRC:000000000000000000000000000000000000000000000000000000000000000000000000000000-100-1-1-1-1-1-1
```

It is always interesting to learn that a plane – here G-EUUG which is an A320 Airbus A320 owned by British Airways [16] going from London to Italy as part of the flight BA9216 – is flying with a fault on one of its secondary landing instruments [17].

Some of the abbreviations used in the messages are described at `http://www.angelfire.com/sc/scannerpost/acars.html`

16. `www.planespotters.net`
17. `http://www.scancat.com/Code-30_html_Source/acars.html` for the content of the sentences and in particular the header.

5 From prototyping to exploiting under gnuradio

`gnuradio-companion` converts a signal processing scheme defined as a graphical assembly of signal processing blocks to a Python script calling the associated functions. However, the data stream transiting between the blocks, *i.e.* the stream generated by the I/Q demodulator, only passes through from one block to another without being accessible from the Python script (code 1, corresponding to the graphical block assembly shown on Fig. 8). Each one of the blocks is itself programmed in Python or C(++), complying with a data exchange protocol [18]. We will thus translate the C program written for processing the FSK modulated data in order to comply with the conventions expected from a `gnuradio-companion` block and thus process the stream recovered from a `gnuradio` source in real time (instead of recording the data stream in a binary file for post-processing).

```
self.wxgui_scopesink2_0 = scopesink2.scope_sink_f(
            self.GetWin(),
            title="Scope Plot",
[...]
    )
self.Add(self.wxgui_scopesink2_0.win)
self.gr_throttle_0 = gr.throttle(gr.sizeof_float*1, samp_rate)
self.gr_file_source_0 = gr.file_source(gr.sizeof_char*1, "nom_du_fichier.wav", True)
self.gr_char_to_float_0 = gr.char_to_float(1, 1)

self.connect((self.gr_throttle_0, 0), (self.wxgui_scopesink2_0, 0))
self.connect((self.gr_char_to_float_0, 0), (self.gr_throttle_0, 0))
self.connect((self.gr_file_source_0, 0), (self.gr_char_to_float_0, 0))

tb = top_block()
tb.Run(True)
```

TABLE 1 – The Python code generated by `gnuradio-companion` does not provide access to the radiofrequency data flow but only defines blocks and the information transfer from one signal processing block output to the input of the next one.

Developing a `gnuradio` compatible block requires meeting some dedicated development environment constraints including a directory tree structure with dedicated sub-directories in which the various files associated with the block are stored.

The first step in the creation of one (or several) block(s) is thus to create this development environment. In order to avoid the time consuming step of manually creating the directory structure, we use the `gr-modtool.py` [19] script. This tool both generates the directories and fills them with minimalist files meeting the basic needs of a block.

Creating a new project is achieved with the following command :

```
gr_modtool.py create plop
```

The result of this command is a directory named `gr-projectname` in the current directory, including, among others, the directories :

– `grc` for the description files used by `gr-companion`,
– `include` for the headers,
– `lib` for the C++ source codes (with `.cc` extensions).

Adding a digital signal processing block to this project is achieved with the command (after entering the project directory) :

```
gr_modtool.py add -t general plup
```

The `-t` option defines the type of the block – in this case a generic type the other types exist including sinks, source, decimators ... as will be seen later, the type of the block defines the signature of some of the provided methods.

```
Enter valid argument list, including default arguments:
Add Python QA code? [Y/n]
Add C++ QA code? [Y/n]
```

The first line allows for the specification of the parameters to be received by the block when instanciated (data type, gain, threshold ...). The next two lines define whether (or not) unitary tests are added to the bock code.

At the end of this step, the directories have been filled with the necessary files and these files have been partially filled with the adapted template.

Three files are most important :

– `grc/ProjectName_BlocName.xml` : this file describes the bloc to `gnuradio-companion`. It defines, among other things, the name of the bloc (`name` field), in which class this block fits (`category`

18. `http://gnuradio.org/redmine/projects/gnuradio/wiki/BlocksCodingGuide`
19. `git://github.com/mbant/gr-modtool.git`

field), parameters (`param` field) as well as their names, types and associated variables, the inputs (`sink` fields) and the outputs (`source` fields). An example of such a file which provides a parameter (real value `seuil`) to the signal processing method `decoder` of the class `acars` reads as

```
<?xml version="1.0"?>
<block>
  <name>decodeur</name>
  <key>acars_decodeur</key>
  <category>acars</category>
  <import>import acars</import>
  <make>acars.decodeur($seuil)</make>
  <param>
    <name>Threshold</name>
    <key>seuil</key>
    <type>real</type>
  </param>
  <sink>
    <name>in</name>
    <type>float</type>
  </sink>
</block>
```

  - `include/ProjectName_BlocName.h` and `lib/ProjectName_BlocName.cc` : the class and its implementation. The default configuration is for these files to include
    - a private constructor,
    - a public destructor,
    - a function `ProjectName_make_BlocName` defined as `friend` with respect to the class (and thus allowed to access the constructor) and whose aim to to return an instance of the class ;
    - a public method `general_work`.

These abstract concepts of the directory structures will be illustrated later through some source code examples (section 5.3), and the reader is encouraged to consult the example code available at `http://jmfriedt.free.fr/gr-acars.tar.gz` or on the CGRAN web site at `www.cgran.org/wiki/ACARS`.

The last method might also be called `work` depending on the block type. This method is certainly the mist important since it receives the data stream from the previous block, performs the processing, and generates the new resulting data stream. Its signature is as follows :

```
int plop_plup::general_work (int noutput_items,
     gr_vector_int &ninput_items,
     gr_vector_const_void_star &input_items,
     gr_vector_void_star &output_items)
```

with `noutput_items` being, as its name does not indicate, the number of *input* items, `input_items` an array including the incoming data and which must be casted to the appropriate type, and `output_items` the array the block must fill with the processed data. `ninput_items` might not exist, depending of the block type, and does not seem to be of much use.

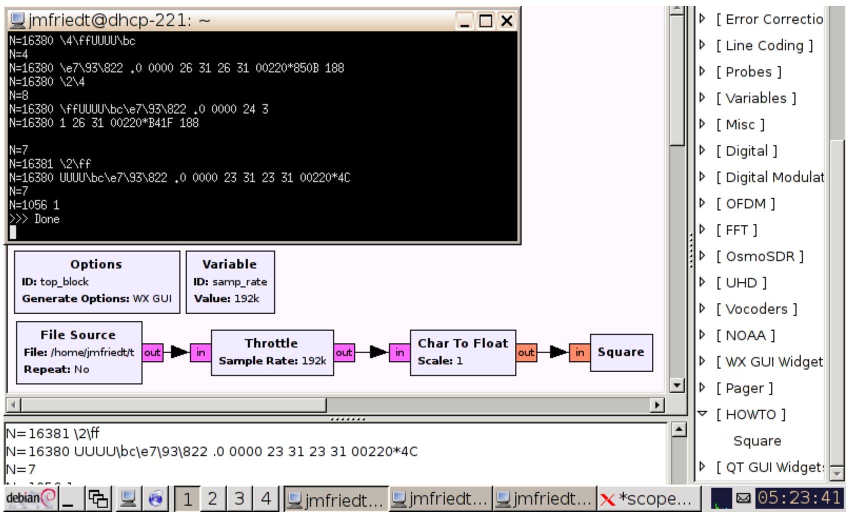
In order to compile our application, we perform as we have done with all other `gnuradio` archives met so far : create a sub-directory `build` in the source directory of the module to be compiled, execute `cmake ../` after entering the newly created directory, then `make VERBOSE=1 && make install`. Notice that once the block newly created in `gnuradio-companion` (seen in the list of menus in the right column), it is no longer necessary to restart the graphical user interface after each compilation of the block. Indeed, the Python code is regenerated each time the gear icon is clicked before being executed, and thus the newly compiled module reloaded. We will thus always run the latest revision of the block *after installation* by using the command `make install` (Fig.12).

5.1 Case of the XE1203F radiomodem

Having prototyped under GNU/Octave the filtering and signal processing algorithms, we wish to diplay in real time the output of the decoded stream. Porting this code is performed in 3 steps :

1. convert the GNU/Octave script to a C program and check that all functionalities remain operational by reading and processing the same audio record (or binary file if the sampling rate is not comptible with that of the sound card) as was used for developing the script. This development step might still benefit from the ability to read all of the data stream at once (sample code 2),
2. convert this C program to a version able to process data provided as successive segments of arbitrary size (Fig. 14), as will happen with the data stream provided by the previous `gnuradio` demodulation or filtering blocks,
3. integrate this code in the C(++) `gnuradio` framework.

The issue of handling a stream of data contained in segments of variable size is solved in the following way : we know that one byte contains 10 bits (start, 8 data bits, stop) or 400 samples for a signal sampled at 192000 Hz and transmitted at 4800 bits/s (192000/4800×10=400). We first search for the start bit



```
N=16380 \4\ffUUUU\bc
N=4
N=16380 \e7\93\822 .0 0000 26 31 26 31 00220*850B 188
N=16380 \2\4
N=8
N=16380 \ffUUUU\bc\e7\93\822 .0 0000 24 3
N=16380 1 26 31 00220*B41F 188

N=7
N=16381 \2\ff
N=16380 UUUU\bc\e7\93\822 .0 0000 23 31 23 31 00220*4C
N=7
N=1056 1
>>> Done
```

```
N=16381 \2\ff
N=16380 UUUU\bc\e7\93\822 .0 0000 23 31 23 31 00220*4C
N=7
```

FIGURE 12 – Decoding a data stream generated by a XE1203F radiomodem thanks to the software signal processing of the I and Q data in order to extract the digital data that were emitted, first by reading a binary file (.wav) in which the stream had been initially recorded before being played again in order to validate the proper functionality of the block (named here `square`). Notice, at the bottom of the menu available on the right side of the window, the addition of a `HOWTO` entry in which the `square` module is located, associated with the result of the compilation of our processing block. Modifying the `square` functions does not yield the need to re-launch `gnuradio-companion` since the block is re-loaded at each execution of the signal processing Python code.

(transition from the rest state – 1 – to the start bit value – 0 [20]), and if the length of samples remaining to be processed is larger than 400 bits, we keep on applying the processing algorithm aimed at testing the bit value every 40 samples (since 192000/4800=40). As is all asynchronous communication, the start bit is thus used to re-synchronize the receiver to the emitter for the next 9 bits : we will test at the end of the byte transmission that the stop bit is indeed at the 1 level, and then search by increasing the index sample by sample the next transition to 0 in order to re-launch a new decoding procedure. If the amount of data remaining to be processed when a start bit transition was met is below 400, we then copy at the beginning of the sample array the remaining dataset, and append the new data in order to launch the digital processing on a complete dataset longer than the length of one byte. If no start bit transition is detected on the current dataset, the search reaches the end of the array and the new data can be processed independently of any previous record. The result of this implementation (Fig. 13) is visible on table 3.

5.2 Decoding ACARS – *warning, addictive paragraph*

The philosophy we have described for converting software prototyped under GNU/Octave towards a processing scheme compatible with datasets of variable length requires the concatenation of the recorded data until a block of the appropriate size is generated, and thus return to conditions compatible with the initial data stream met under GNU/Octave. Whether a block is worth processing is determined by a threshold condition on the convolution of the recorded dataa with a filter centered on 2400 Hz. We have already mentioned that an ACARS sentence always starts with multiple periods of a 2400 Hz sine wave for the automatic gain control of the radiofrequency receiver to settle, and for synchronizing the receiver local clock with that of the plane. This signal is thus used for identifying the beginning of the sentence. We then concatenate the following datasets from the stream until the maximum ACARS message length is reached (we tried only concatenating as long as the received signal power is above a given threshold, but this approach seems to be *excessively* sensitive to the threshold value), *i.e* 258 characters when considering the maximum allowed text length.

20. `http://en.wikipedia.org/wiki/Asynchronous_serial_communication`

```
[... usual headers ...]
#define MAXSIZE 100000
#define NSEARCH 260  // 13 periods * 20 pts/period

void conv1(unsigned char* in,int* out,int N,int len)
{int somme=0,k;
 for (k=0;k<len;k++) somme+=(int)in[k];    // sliding average on len points
 for (k=len;k<N;k++) {somme=somme-in[k-len]+in[k];out[k-len]=somme/len;}
}

#define segment 8192

int main(int argc,char **argv)
{char filename[255];
 int deb,seuil,courant;
 int k,i,fe=192000,f,N;
 int dm[MAXSIZE];
 unsigned char d[MAXSIZE],code_asc;
 int b=0,l0=0,pos=0;

 if (argc<4) printf("%s filename deb seuil\n",argv[0]); else
   {sprintf(filename,"%s",argv[1]);deb=atoi(argv[2]);seuil=atoi(argv[3]);}

 deb=0;
 do {
   f=open(filename,O_RDONLY);  // read a data segment in the dataset file
   k=segment;
   i=lseek(f,deb,SEEK_SET);
   if (i!=deb) fprintf(stderr,"lseek error %d\n",k);
   N=read(f,&d[pos],k); printf("\nN=%d\n",N);
   close (f);

   conv1(d,dm,N,20);         // c=ones(20,1)/20;dm=conv(d,c);dm=dm(20/2:end-20/2);
   for (k=0;k<N;k++) if (dm[k]<seuil) dm[k]=1; else dm[k]=0;
       // k=find(dm<100);dm(k)=0; k=find(dm>=100);dm(k)=1;dm=1-dm;
       // % rest state is 1, start bit is encoded by going to 0

   courant=0;
   while (courant<(N+pos-380)) {    //   k=find(dm(courant:end)<1);dm=dm(k(1)+courant-1:end);
     do {courant++;} while ((dm[courant]==1)&&(courant<(N+pos-380)));
     if (courant<(N+pos-380)) // db=dm(20:40:20+40*9);        % 10 bits = START+9, 40 points/bit
       {code_asc=dm[courant+60]+dm[courant+100]*2+dm[courant+140]*4+dm[courant+180]*8+
                 dm[courant+220]*16+dm[courant+260]*32+dm[courant+300]*64+dm[courant+340]*128;
        if (code_asc>0) {
           if (dm[courant+380]==0) {} // printf("err "); // stop bit error
              else
              {if (((code_asc>=32)&&(code_asc<128))||(code_asc=='\n')||(code_asc=='\r'))
                  printf("%c",code_asc);
                  else printf("\\%x",code_asc);
              }
        }
        courant+=380;  // middle of the stop bit of the byte we have juste processed
       }
   }
   memcpy(d,&d[courant],N+pos-courant); // copy the remaining data not yet processed
   pos=N+pos-courant;
   deb+=segment;
 } while (N==segment); // loop on deb to simulate the data stream of a grc block
 printf("\n");
}
```

Table 2 – Translation to C of a code prototyped under GNU/Octave for decoding radiofrequency digital transmissions generated by a XE1203F radiomodem at a rate of 4800 bauds, using a frequency modulation (FSK). We have commented the main functions of the GNU/Octave script while translating to C. Notice how the file is read by segments of `segment` elements, and how this new data segment is appended to the remaining data values which have not yet been processed because they no longer contain a full byte : this strategy is representative of our approach for handling the data stream provided by `gnuradio` as segments during the real time decoding.

```
UUUU\bc\e7\fb\822 .0 \fc000 465 31 534 6002109
\c\f8\e0UUU\bc\e7\93\82\fe .0 00\fc0 558 31 500 7 02110
\fc\c0UUU\bc\e7\93\822 0 0000595 31 16\f9 8 02110
U\e0UUUU\bc\93\822 .0\fc0000 6\ff7 31 426 9 02110
\ff\b0E\e0UUUU\bc\e7\93\fe2 .0 0T00 225 31374 10 0210
\fc\f0UUUU\bc\e7\93\82\fe .0 000 333 31 24 12 02110
\ff\ff\e0\c1UUU\bc\e7\93\822 .0 0\bc00 203\bc31 419 13 02111
\ff\5\ff\f8\c0U5UU\bc\e7\93\822 .0 00\e40 367 \e71 246 14 02111
\f8\f0\e0UUU\bc\e7\93\822 .0 000\94 195 3\95 296 15 02111
\f8\f0UUUU\bc\e7\93\822 \fe0 0000\fc284 31 30\fd 17 02111
\ff\e0\5\f0\f0\e0UUUU\bc\e7\93\822 .0 0000\d02\f09 31 311 \859 02112
\ff\e0\f0\f0\e8\e0UUUU\bc\e7\93\822 .0 0000 336 31 \e741 20 021\e52
```

```
2 .0 0000 465 31 534 6 02109
2 .0 0000 558 31 500 7 02110
2 .0 0000 595 31 169 8 02110
2 .0 0000 637 31 426 9 02110
2 .0 0000 225 31 374 10 02110
2 .0 0000 333 31 324 12 02110
2 .0 0000 203 31 419 13 02111
2 .0 0000 367 31 246 14 02111
2 .0 0000 195 31 296 15 02111
2 .0 0000 284 31 305 17 02111
-> 2 .0 0000 201 31 325 18 02112
2 .0 0000 209 31 311 19 02112
2 .0 0000 336 31 341 20 02112
```

Table 3 – Comparison of the sentences decoded by `gnuradio` after processing the data received by the EZCAP receiver (left), and the sentences emitted by the radiomodem (right). All sentences on the left start with the synchronization sequence UUUU, followed by the identifier of the emitting modem, followed by the actual data sentence. Altough the processing result is not perfect, most of the sentences emitted (right) are found in the decoded signals (left). We have indicated in the right table by a line prefixed with an arrow one line which was completely omitted during the processing due to the failure to decode the transmission UUUU initialization sequence.

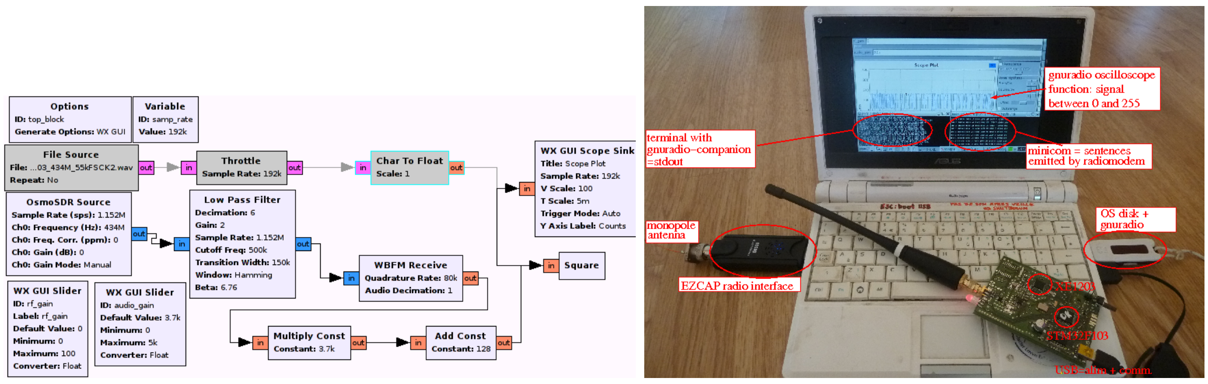


FIGURE 13 – Left : graphical `gnuradio-companion` block representation of the decoder of the sentences acquired by the EZCAP, and provided following demodulation to the processing block we have developed and tested by running an audio file (blocks shaded in dark grey). Notice the high value gain (about 3700) and offset (128) which must be added to tune the data range at the output of the demodulation block to match the values read from the .wav file used during the processing steps. Right : experimental setup, with a XE1203F radiomodem controlled by a STM32 microcontoller, and the EZCAP receiver for listening to the emitted data stream.

```
N=1361
N=5444
N=6805 \fcUUUU\bc\e7\93\2
N=1361 2 .
N=682 0
N=1361 000
N=1361 0 4\fe
N=1361 6 3
N=1361 1 4\fd
N=4083 0 31 01138
N=5444 *7C12 188

N=5444
N=1361 \f0
N=1361 \f0
N=1361
N=2722
N=1361
N=1361
N=5444 \10\fc
N=6805 UUUU\bc\e7\93\822
```

FIGURE 14 – The `gnuradio` compatible block is fitted, for debugging, with a display of the N variable including the number of data provided by the demodulation block : we observe that this value evolves more or less randomly, and that no hypothesis can be made on the length of data array to be processed. The proposed solution is described in the text.

Translating the GNU/Octave code to C does not involve any significant issue other than the efficient implementation of the convolution through a Fourier transform. Hopefully, a library is available so that we do not have to implement our own Fourier transform functions : `libfftw3`.

gnuradio uses a compilation environment aimed at being portable between un*x, MacOS X, and MS-Windows : cmake, or the *Cross Platform Make* (www.cmake.org). While adding a library in the classical Makefile used to be as simple as adding the library name preceded by -l, cmake is supposed to find by itself the right libraries in the appropriate directories, and do so whatever the development environment. In order to add the libfftw3 library, we have included parts of the script available at http://code.google.com/p/qmcpack/source/browse/trunk/CMake/FindFFTW.cmake, and added the necessary statements in CMakeLists.txt (which acts as the usual Makefile we are used to) by mentioning that headers should be searched with find_package(FFTW) and INCLUDE_DIRECTORIES($FFTW_INCLUDE_DIR) and library portability is met with LINK_LIBRARIES($FFTW_LIBRARIES).

```
[... entetes habituels ...]
#include <fftw3.h>
#define MAXSIZE 1000000
#define NSEARCH 260  // 13 periods * 20 pts/period

void remove_avg(unsigned char *d,int *out,int tot_len,int fil_len)
{int tmp,k,avg=0;
 for (k=0;k<fil_len;k++) avg+=d[k];    // initialise moyenne glissante
 for (k=0;k<tot_len-fil_len;k++) {out[k]=d[k]-avg/fil_len; avg-=d[k]; avg+=d[k+fil_len]; }
 for (k=tot_len-fil_len;k<tot_len;k++) out[k]=d[k]-avg/fil_len;
}
```

This first function of removing the average value – remember that a cross-correlation will be used to identify the sequence of 2400 Hz sine waves, thus assuming that the signal exhibits a null average – subtracts a sliding window average of length fil_len from the data array d to generate the output out. This pre-processing step is mandatory for all the following operations.

```
int main(int argc,char **argv)
{char filename[255];
 int deb,fin,seuil,k,i,fe=48000,f,N,t,*out,n,b=0,l0=0;
 unsigned char d[MAXSIZE],*toutd,*tout;
 double a=0.,*rs12,*rs24,*rc12,*rc24,c2400[20], c1200[20],s2400[20], s1200[20];
 fftw_complex *c2400x13,*fc2400x13,*fd,*s,mul,*ss;
 fftw_plan plan_a, plan_b, plan_R;

 if (argc<5) printf("%s filename deb fin seuil\n",argv[0]); else
   {sprintf(filename,"%s",argv[1]); deb=atoi(argv[2]); fin=atoi(argv[3]); seuil=atoi(argv[4]); }
 if ((fin-deb)>MAXSIZE) {fin=deb+MAXSIZE; fprintf(stderr,"deb=%d fin=%d\n",deb,fin); }
 f=open(filename,O_RDONLY); if (f<0) fprintf(stderr,"open error %d\n",f);
 k=lseek(f,deb,SEEK_SET);   if (k!=deb) fprintf(stderr,"lseek error %d\n",k);
N=read(f,d,fin-deb); close (f);             // d=d(deb:fin); N=fin-deb;
 out=(int*)malloc(sizeof(int)*N);
 remove_avg(d,out,N,60);  // c=ones(60,1)/60; dm=conv(d,c);dm=dm(60/2:end-60/2); d=d-dm;
```

Reading the data points recorded by gnuradio-companion is trivial – each data is one byte long so that no endianness issue arises – and the index of the beginning of the analyzed sequence will allow is to simulate the variable length dataset provided by the gnuradio demodulator output.

```
 c2400x13 = (fftw_complex *) fftw_malloc (sizeof (fftw_complex) * N);
[... same initialization method for fc2400x13, fd, s, ss ...]
 for (t=0;t<520;t++)  // t=[0:520]; c2400x13=exp(i*t*2400/fe*2*pi);
    {c2400x13[t][0]=cos((double)t*2400./fe*2*M_PI);
     c2400x13[t][1]=sin((double)t*2400./fe*2*M_PI);
    }
 for (t=520;t<N;t++) {c2400x13[t][0]=0;c2400x13[t][1]=0;} // 13 periodes 2400 Hz
 for (k=0;k<N;k++) {s[k][0]=(double)out[k];s[k][1]=0.;}
 plan_a=fftw_plan_dft_1d(N, c2400x13, fc2400x13, FFTW_FORWARD, FFTW_ESTIMATE);
 plan_b=fftw_plan_dft_1d(N, s, fd , FFTW_FORWARD, FFTW_ESTIMATE);
 plan_R=fftw_plan_dft_1d(N, fd,ss, FFTW_BACKWARD, FFTW_ESTIMATE);
 fftw_execute (plan_a); fftw_execute (plan_b);
 for (k=0;k<N;k++)       // produit des transformees de Fourier pour intercorrelation
    {mul[0]=fc2400x13[k][0]*fd[k][0]-fc2400x13[k][1]*fd[k][1];
     mul[1]=fc2400x13[k][1]*fd[k][0]+fc2400x13[k][0]*fd[k][1];
     fd[k][0]=mul[0]/(float)N;
     fd[k][1]=mul[1]/(float)N;
    }
 fftw_execute (plan_R);  // puis retour dans le domaine reel par FFT inverse
 fftw_destroy_plan (plan_a); fftw_destroy_plan (plan_b); fftw_destroy_plan (plan_R); // s=conv(c2400x13,d);
```

Calling the Fourier transform related functions provided by the fftw3 library, implementing some of the most complex function to program and validate otherwise, is achieved by allocating memory for the various arrays used during the intermediate steps of the cross-correlation computation by reaching the Fourier domain (remember that we have already described in [13, in French] the processing time gain – from N^2 to $N\ln(N)$ – when searching for a pattern in a dataset of the length N using a cross-correlation approach). The pattern we are looking for are defined as sine wave segments with a frequency normalized to the sampling rate fe=48000 Hz.

```
 for (k=0;k<N-NSEARCH;k++) if (ss[k+NSEARCH-2][0]>a) {a=ss[k+NSEARCH-2][0];b=k;} // [a,b]=max(real(s));
 printf("a=%f b=%d\n",a,b);
 b=b%20;
```

The index (time position) of the cross correlation maximum defines the position at which the pattern of 2400 Hz sine waves was most probably identified in the incoming data stream. Is identification is a key aspect for the next steps to be efficiently implemented since it solves all phase issues. Indeed, we have seen when presenting the general concepts around the I and Q output streams from the mixers that two unknown parameters are the magnitude and the phase. The cross-correlation allows for identitifying of both parameters, although with a heavy computational load when doing so. If the phase is already known, than only the magnitude remains to be identified, and this step only requires N multiplications by sliding the pattern made of one sine wave period over the experimental data *by one period step* (rather than by one sampling point step as would be required if the phase were unknown). This algorithm is implemented below with one period of a 2400 Hz sine wave and half a period of a 1200 Hz sine wave, each representing one of the possible bit values of the incoming data stream. We then consider the absolute value (modulus) of the convolution in order to avoid the case of the phase condition in which the magnitude of the result is large but negative.

```
 for (t=0;t<20;t++)  // t=[0:520]; c2400x13=exp(i*t*2400/fe*2*pi);
    {c2400[t]=cos((double)t*2400./fe*2*M_PI); //  t=[0:20];  % 2400 Hz dans 48 kHz = 20 points/periode
     s2400[t]=sin((double)t*2400./fe*2*M_PI); //  c2400=exp(i*t*2400/fe*2*pi);
     c1200[t]=cos((double)t*1200./fe*2*M_PI); //  c1200=exp(i*t*1200/fe*2*pi);
     s1200[t]=sin((double)t*1200./fe*2*M_PI);
    }

 rs12=(double*)malloc(sizeof(double)*(N-b)/20); // fin20=floor(length(s12)/20)*20;
[... idem pour rs24, rc12, rc24 ...]
 l0=0;
 for (k=b;k<N-20;k+=20)
    {rs12[l0]=0.; rs24[l0]=0.; rc12[l0]=0.; rc24[l0]=0.;
     for (t=0;t<20;t++)
       {rs24[l0]+=((double)out[k+t]*s2400[t]); rc24[l0]+=((double)out[k+t]*c2400[t]);
        rs12[l0]+=((double)out[k+t]*s1200[t]); rc12[l0]+=((double)out[k+t]*c1200[t]);
       }
    rs12[l0]=sqrt(rs12[l0]*rs12[l0]+rc12[l0]*rc12[l0]); rs24[l0]=sqrt(rs24[l0]*rs24[l0]+rc24[l0]*rc24[l0]);
    l0++;
    }
```

The raw data have been processed by the convolution filter, and the only remaining step is the comparison of the relative amplitude of the outputs of the two filters in order to estimate whether the bit value is most probably a 1 or a 0 :

```
 l0=0;        // all memory allocations (malloc) must be freed (free) ... omitted here
 do l0++; while ((rs24[l0]+rs12[l0])<2*seuil);  // cherche debut
 do l0++; while ((rs24[l0]+rs12[l0])>2*seuil);  // cherche fin
 fin=l0;
 l0=0;
 do l0++; while (rs24[l0]<seuil);    // l1=find(rs24>seuil);l1=l1(1);rs12=rs12(l1:end);rs24=rs24(l1:end);
 do l0++; while (rs12[l0]<rs24[l0]); // l =find(rs12>rs24); l=l(1);  rs12=rs12(l:end); rs24=rs24(l:end);
 toutd=(char*)malloc(fin-l0);
 tout=(char*)malloc(fin-l0+2);
 for (k=l0;k<fin;k++)  // pos12=find(rs12>rs24);pos24=find(rs24>rs12);toutd(pos12)=0;toutd(pos24)=1;
     if (rs24[k]>rs12[k]) toutd[k-l0]=1; else toutd[k-l0]=0;
```

and once the binary datastream contained in `toutd` have been obtained, decoding is completed by inverting the current bit values depending on the `toutd` values and displaying first as hexadecimal values the decoded bytes – an ACARS sentence must start with 2B 2A 16 16 01 – and then the characters among the sequence which can be printed are interpreted through their ASCII code.

```
 n=0; tout[n]=1;n++;tout[n]=1;n++; // the first two 1s are deleted since we sync on 1200
 for (k=0;k<fin-l0;k++) {if (toutd[k]==0) tout[n]=1-tout[n-1]; else tout[n]=tout[n-1];
                         n=n+1;
                        }
  for (k=0;k<fin-l0;k+=8)
      printf("%02x ",tout[k]+tout[k+1]*2+tout[k+2]*4+tout[k+3]*8+tout[k+4]*16+tout[k+5]*32+tout[k+6]*64);
  printf("\n");
  for (k=0;k<fin-l0;k+=8)
      printf("%c",   tout[k]+tout[k+1]*2+tout[k+2]*4+tout[k+3]*8+tout[k+4]*16+tout[k+5]*32+tout[k+6]*64);
} // checksomme=1-mod(sum(binaire(:,1:7)')',2); % verification
```

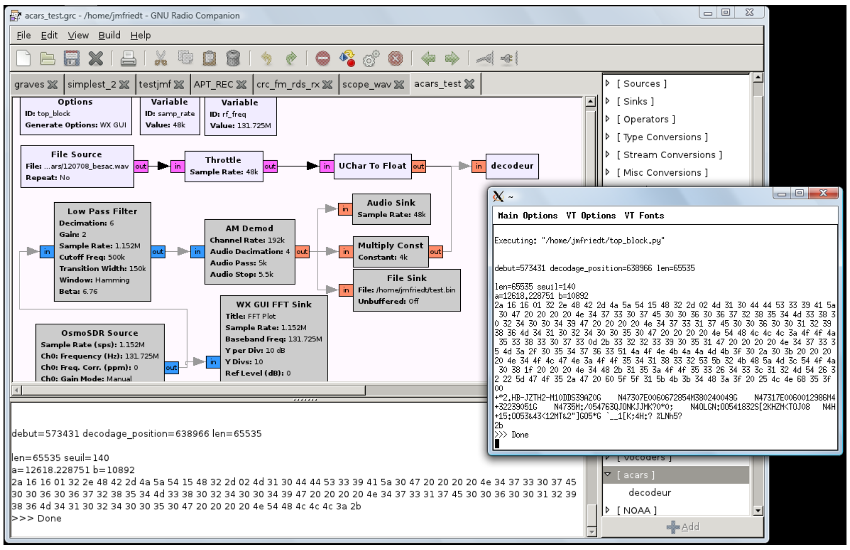
We have omitted the parity bit calculation which cannot be used anyway for retrieving the initial value of a byte corrupted during communication over a noisy channel : error-correcting codes allowing such a feat are much more complex than just an additional bit at the end of each transmitted byte.

Having validated that the resulting C code provides results consistent with the initial GNU/Octave script, we include this software in the C++ environment needed for compatibility with `gnuradio-companion` as shown earlier. Since the FFT requires a minimum dataset length to run, we have selected, rather than appending the data segment that has not been processed as has been done for the XE1203F radiomodem, to accumulate as many samples as would be contained in a message of maximum length as allowed by the ACARS protocol, namely 258 characters. Decoding the 13 adjacent 2400 Hz oscillations is performed continuously on the recorded data stream (here with the constraint of a minimum dataset length of 20 points/period×13 periods=260 points), and when a threshold condition is met (meaning the beginning of a sentence), we accumulate 40 points/bit×8 bit/character×258 charaacters or about 83000 points. The decoding algorithm we implemented in C is then applied on these 83000 points as if they had been read from a file (Fig. 15). We have however observed a strong sensitivity of the decoding performance with the threshold values, an aspect hardly satisfactory with this implementation which would deserve improvement by adding an automatic gain control. The decoded sentence slightly differs from the result obtained with the C program described earlier and includes

```
+*2.HB-JZTH2-M10DDS39AZ0G    N47307E0060672854M380240049G    N47317E0060012986M410240050G
N47333E005JLGL0HLrTLMMLF0JN8____1KHLJ2EP0JKHIL.50145524@0%0G____1K0381E00JKNGLML[2KHZ=C+0508
```

```
N4H+AJ90053&4LC12MT&2""80:*8_  N$4DK7E@_Z31J@
```

emitted by an A319 Airbus plane (HB-JZT) operated by the Swiss Easyjet (consistent with a listening station located in Besançon, close to the Swiss border) and seems to include some coordinate transmission (47.307 N, 6.607 E is indeed located in the North-Eastern area of the Franche-Comté region) whose meaning we are not aware of. The end of the message is probably corrupted.



Options
ID: top_block
Generate Options: WX GUI

Variable
ID: samp_rate
Value: 48k

Variable
ID: rf_freq
Value: 131.725M

File Source
File: ...ars/120708_besac.wav
Repeat: No

Throttle
Sample Rate: 48k

UChar To Float

decodeur

Low Pass Filter
Decimation: 6
Gain: 2
Sample Rate: 1.152M
Cutoff Freq: 500k
Transition Width: 150k
Window: Hamming
Beta: 6.76

AM Demod
Channel Rate: 192k
Audio Decimation: 4
Audio Pass: 5k
Audio Stop: 5.5k

Audio Sink
Sample Rate: 48k

Multiply Const
Constant: 4k

File Sink
File: /home/jmfriedt/test.bin
Unbuffered: Off

WX GUI FFT Sink
Title: FFT Plot
Sample Rate: 1.152M
Baseband Freq: 131.725M
Y per Div: 10 dB
Y Divs: 10
Ref Level (dB): 0

OsmoSDR Source
Sample Rate (sps): 1.152M
Ch0: Frequency (Hz): 131.725M
Ch0: Freq. Corr. (ppm): 0
Ch0: Gain Mode: Manual

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


debut=573431 decodage_position=638966 len=65535

len=65535 seuil=140
a=12618.228751 b=10892
2a 16 16 01 32 2e 48 42 2d 4a 5a 54 15 48 32 2d 02 4d 31 30 44 44 53 33 39 41 5a
 30 47 20 20 20 20 4e 34 37 33 30 37 45 30 30 36 30 36 37 32 38 35 34 4d 33 38 3
0 32 34 30 30 34 39 47 20 20 20 20 4e 34 37 33 31 37 45 30 30 36 30 30 31 32 39
38 36 4d 34 31 30 32 34 30 30 35 30 47 20 20 20 20 4e 54 48 4c 4c 4c 3a 4f 4f 4a
 35 33 38 33 30 37 33 0d 2b 33 32 32 33 39 30 35 31 47 20 20 20 20 4e 34 37 33 3
5 4d 3a 2f 30 35 34 37 36 33 51 4a 4f 4e 4b 4a 4a 4d 4b 3f 30 2a 30 3b 20 20 20
20 4e 34 4f 4c 47 4e 3a 4f 4f 35 34 31 38 33 32 53 5b 32 4b 48 5a 4d 3c 54 4f 4a
 30 38 1f 20 20 20 4e 34 48 2b 31 35 3a 4f 4f 35 33 26 34 33 3c 31 32 4d 54 26 3
2 22 5d 47 4f 35 2a 47 20 60 5f 5f 31 5b 4b 3b 34 48 3a 3f 20 25 4c 4e 68 35 3f
00
+*2.HB-JZTH2-M10DDS39AZ0G    N47307E0060672854M380240049G    N47317E0060012986M4
+32239051G    N4735M:/054763QJONKJJMK?0*0;    N40LGN:00541832S[2KHZM<TOJ08   N4H
+15:0053&43<12MT&2"]G05*G `__1[K:4H:? %LNh5?
2b
>>> Done
```

```
debut=573431 decodage_position=638966 len=65535

len=65535 seuil=140
a=12618.228751 b=10892
2a 16 16 01 32 2e 48 42 2d 4a 5a 54 15 48 32 2d 02 4d 31 30 44 44 53 33 39 41 5a 30 47 20 20 20 20 4e 34 37 33 30 37 45
30 30 36 30 36 37 32 38 35 34 4d 33 38 30 32 34 30 30 34 39 47 20 20 20 20 4e 34 37 33 31 37 45 30 30 36 30 30 31 32 39
38 36 4d 34 31 30 32 34 30 30 35 30 47 20 20 20 20 4e 54 48 4c 4c 4c 3a 2b
>>> Done
```

FIGURE 15 – Decoding using a `gnuradio-companion` block including the algorithms described in the main text for processing ACARS sentences recorded in an audio file sampled at 48 ksamples/s. Replacing the audio file with the output of the `osmosdr` block for real time signal processing (blocks shaded in dark grey) allows for the proper identification of the beginning of the transmission of a sentence, although the decoding of the message content often fails, probably due to a poor tuning of threshold values.

An example of real time decoding of a data stream processed by the block described in this paragraph is (first the values of the received bytes, and then their interpretation as printable characters through theur ASCII code) is

```
63 70 5c 72 27 7d 04 09 5c 42 43 26 7e 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f
2b 2a 16 16 01 47 2e 45 49 2d 44 54 4e 15 48 31 35 02 44 35 33 43 41 5a 30 32 34 33 23 44 46 42 33 2c 33
39 2c 30 35 39 2c 30 34 35 39 39 2c 30 31 33 36 2f 56 31 30 32 2c 31 32 2c 32 32 32 2c 32 39 32 2c 30 33
2c 30 31 2c 30 30 30 30 30 2f 56 32 30 32 2c 30 38 2c 30 38 31 2c 30 36 39 2c 30 30 2c 30 30 2c 30 30 30
30 30 2f 56 33 30 32 2c 30 36 2c 31 34 36 2c 33 31 32 2c 30 36 32 32 2f 56 34 31 30 2c 30 36 2c 32 32 33
2c 30 32 35 2c 30 36 32 31 2f 56 35 30 30 2c 30 30 2c 30 30 30 2c 30 30 30 2c 30 30 30 30 2f 56 36 30 30
2c 30 30 2c 30 30 30 2c 30 30 30 2c 30 30 30 30 2f 56 37 30 34 31 2c 30 38 37 2c 30 30 30 36 31 2c 32 32
32 32 32 32 32 32 32 32 32 47 31 31 2f 56 38 30 34 31 2c 30 38 36 2c 30 30 30 38 31 17 24 3e 7f
cp\r'}\BC&~+*G.EI-DTNH15D53CAZ0243#DFB3,39,059,04599,0136/V102,12,222,292,03,01,00000/V202,08,081,069,00,
00,00000/V302,06,146,312,0622/V410,06,223,025,0621/V500,00,000,000,0000/V600,00,000,000,0000/V7041,087,00
061,22222222222G11/V8041,086,00081$>
```

We observe that the recorded dataset is too long before the de 0x2b 0x2a 0x16 0x16 header, with a series of 0x7f which represent the 2400 Hz sine waves sent for synchronizing the receiver and the emitter. Nevertheless, decoding is successful, with the display of a sentence emitted by EI-DTN, an A320 operated by Alitalia as part of the AZ0243 trip between London and Milano (consistent with a receiving station located in Paris and a recording at 21h15 since this plane is supposed to leave, when not delayed, from London at 19h50 to land at Milano around 21h45). Here again, the content of the message seems consistent but its meaning is beyond our reach.

Thus, while not perfect, our implementation of the processing tools for real time decoding of ACARS sentences using `gnuradio-companion` allows for the identification of most planes flying over the receiver, sometimes with messages as long as

```
+*R.00-SNBH19C00ASN07LR#CFBFLR/FR12081915430036110006PRESS REG-V 4001HA1 OR  SOL 10HA1
OR SENSE LINE/IDBMC 1!
```

Although the result of “press the REG-V” button is unknown to us (apparently an issue with a valve regulating the cabin pressure), this request seems to be important for the pilot to inform ground, and appears regularly on various ACARS message archives found on the internet ! Such results are not significantly worse than the sound-card based decoders using the radiofrequency scanner output we have tested (the now defunct KRACARS[21] for example, running under DOS).

Improvements to the ACARS digital mode decoder mainly lie in an automated detection of a threshold parameter we use to identify the first bit of the first byte, out of which decoding the whole message relies on. This threshold is currenly set statically, a poor approach considering the wide variety of conditions met when decoding radiofrequency signals. On the other hand, parsing the sentences (plane identifier, flight identifier, transmitted text) is only basic in its current development and is only provided to help getting a first grip on the software by new users who might be put off by reading raw byte sequences. Still following the statements found at `http://www.scancat.com/Code-30_html_Source/acars.html`, having validated the 5 sychronization and header bytes (SOH, ASCII 0x01) at the beginning of the sentence, we extract the plane identifier (bytes 6 to 12), the beginning of text identifier (STX, ASCII code 0x03) before looking for the flight identifier (bytes 22 to 27) which are the most useful informations. Then, under the best circumstances, if the message can be decoded, the printable ASCII characters beyond the 28th byte are displayed.

Thus, the raw message

```
2b 2a 16 16 01 32 2e 2e 48 42 4a 4b 4c 15 31 37 30 02 4d 33 33 41 47 53 30 38 37 31 50 4f 41 30 31 47 53 30 38
37 31 2f 32 37 32 37 31 39 33 34 4c 53 47 47 45 4e 5a 56 2f 4e 34 37 20 31 37 2e 31 2f 45 20 20 36 20 31 2e 35
20 2f 32 36 32 2f 20 34 33 2f 33 30 32 2f 2d 20 33 39 2f 41 55 54 4f 52 50 2f 31 35 03 43 6a 7f
+*2..HBJKL170M33AGS0871POA01GS0871/27271934LSGGENZV/N47 17.1/E  6 1.5 /262/ 43/302/- 39/AUTORP/15Cj
```

is translated to

```
Aircraft=..HBJKL
STX
Seq. No=4d 33 33 41 M33A
Flight=GS0871
POA01GS0871/27271934LSGGENZV/N47 17.1/E  6 1.5 /262/ 43/302/- 39/AUTORP/15ETX
```

or a Falcon 2000 plane registered with the Swiss authorities[22] as located in Geneva, heard around 21h45 from a receiving station located in Besançon, with a flight identifier GS0871, again providing GPS coordinates consistent with the localization of the receiver. The reader owning 15 to 20 million dollars might consider buying second hand such a toy. In the other hand, such an investment also allows for building a 1000×1000 receiver matrix of EZCAP receivers and thus build a phased array RADAR receiver (`http://www.haarp.alaska.edu/`), much more interesting than a flying can.

Furthermore, notice how the message sequence is consistent

```
Mon Aug 27 22:03:31 2012
Aircraft=.PH-XRX
STX
Seq. No=53 35 32 41 S52A
Flight=HV5078
ETX
Mon Aug 27 22:04:30 2012
Aircraft=.PH-XRX
STX
Seq. No=53 35 33 41 S53A
Flight=HV5078
ETX
Mon Aug 27 22:09:00 2012
Aircraft=.PH-XRX
STX
Seq. No=53 35 35 41 S55A
Flight=HV50HG
...
```

indicating that this information (S52A, S53A, S55A) is well decoded.

If the reader, after reading these lines and implementing such concepts, leaves his PC running overnight, hoping to recover a message from a late plane, or wakes up with a start in the morning

21. `http://www.qsl.net/g4hbt/zips/krcrs12.zip`
22. `http://www.bazl.admin.ch/fachleute/luftfahrzeugregister/index.html?lang=en`

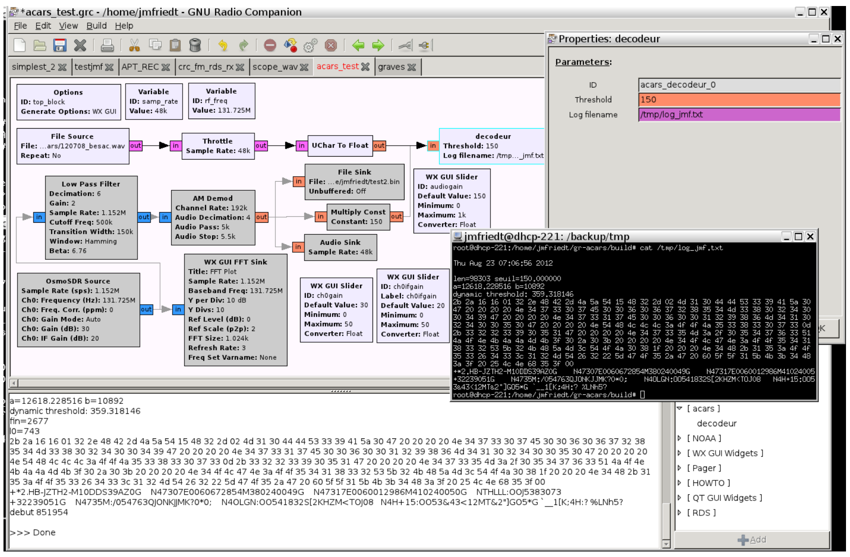
and checks how many planes have been heard – the addiction is set, the virus of radiofrequency transmitted message reception has infected a new victim, and the only medicine remains to ... read the souce code to learn and improve the decoding performance.

5.3 Parameter definition

The last point for easing the developer test cycle is to allow for the definition, by filling a field in the graphical user interface of the block shown in `gnuradio-companion`, of parameters used by the signal processing application, without requiring the compilation of the associated library. We observe that most blocks accept some sort of argument (numerical values, character strings) and wish to make the best use of such capabilities. It all starts by defining the arguments and adding the associated prototypes in the graphical user interface (`grc` directory), *i.e.* updating the XML block description file. The tags defining variables to be provided are `<param> ... </param>` whose arguments are the name of the associated variable `<key>` and the kind of variable involved, for example for a floating point scalar value `<type> real </type>`. This variable declaration *must* be done before `<sink>` parameter, or a (cryptic) error message will be displayed when loading the block. The remaining issue with the XML file is to provide the variable to the C++ program by filling the `<make>` method with the parameter, in our case `acars.decodeur($seuil)` (with `seuil` the name of the variable as defined in the `key` field).

The class declaration file (in the (`include` directory) including the entry function signature as well as the class constructor must be adapted to include the argument `acars_make_decodeur (float seuil1);` in all occurences. Finally, the C++ class constructor implementation itself (`lib` directory) recovers the argument through `acars_decodeur::acars_decodeur (float seuil1)`. In our case, the threshold setpoint is transferred from the `gnuradio-companion` graphical user interface to the C++ class without requiring a new compilation of the library (block).

A simple example of passing a single parameter is easily accessibe in the `gnuradio` archive in the `gnuradio/gr-digital/*/digital_diff_decoder_bb.*` directory : searching for the keyword `modulus` allows to identify the path followed by the variable in the various files defining the block, and get the inspiration for our own block implementation (Fig. 16).



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

Variable
ID: samp_rate
Value: 48k

Variable
ID: rf_freq
Value: 131.725M

File Source
File: ...ars/120708_besac.wav
Repeat: No

Throttle
Sample Rate: 48k

UChar To Float

decodeur
Threshold: 150
Log filename: /tmp..._jmf.txt

File Sink
File: ...e/jmfriedt/test2.bin
Unbuffered: Off

WX GUI Slider
ID: audiogain
Default Value: 150
Minimum: 0
Maximum: 1k
Converter: Float

Low Pass Filter
Decimation: 6
Gain: 2
Sample Rate: 1.152M
Cutoff Freq: 500k
Transition Width: 150k
Window: Hamming
Beta: 6.76

AM Demod
Channel Rate: 192k
Audio Decimation: 4
Audio Pass: 5k
Audio Stop: 5.5k

Multiply Const
Constant: 150

Audio Sink
Sample Rate: 48k

OsmoSDR Source
Sample Rate (sps): 1.152M
Ch0: Frequency (Hz): 131.725M
Ch0: Freq. Corr. (ppm): 0
Ch0: Gain Mode: Auto
Ch0: Gain (dB): 30
Ch0: IF Gain (dB): 20

WX GUI FFT Sink
Title: FFT Plot
Sample Rate: 1.152M
Baseband Freq: 131.725M
Y per Div: 10 dB
Y Divs: 10
Ref Level (dB): 0
Ref Scale (p2p): 2
FFT Size: 1.024k
Refresh Rate: 3
Freq Set Varname: None

WX GUI Slider
ID: ch0gain
Default Value: 30
Minimum: 0
Maximum: 50
Converter: Float

WX GUI Slider
ID: ch0ifgain
Label: ch0ifgain
Default Value: 20
Minimum: 0
Maximum: 50
Converter: Float
```

Properties: decodeur

Parameters:

ID	acars_decodeur_0
Threshold	150
Log filename	/tmp/log_jmf.txt

jmfriedt@dhcp-221: /backup/tmp

```
root@dhcp-221:/home/jmfriedt/gr-acars/build# cat /tmp/log_jmf.txt

Thu Aug 23 07:06:56 2012

len=98303 seuil=150.000000
a=12618.228516 b=10892
dynamic threshold: 359.318146
2b 2a 16 16 01 32 2e 48 42 2d 4a 5a 54 15 48 32 2d 02 4d 31 30 44 44 53 33 39 41 5a 30
47 20 20 20 20 4e 34 37 33 30 37 45 30 30 36 30 36 37 32 38 35 34 4d 33 38 30 32 34 30
30 34 39 47 20 20 20 20 4e 34 37 33 31 37 45 30 30 36 30 30 31 32 39 38 36 4d 34 31 30
32 34 30 30 35 30 47 20 20 20 20 4e 54 48 4c 4c 4c 3a 4f 4f 4a 35 33 38 33 30 37 33 0d
2b 33 32 32 33 39 30 35 31 47 20 20 20 20 4e 34 37 33 35 4d 3a 2f 30 35 34 37 36 33 51
4a 4f 4e 4b 4a 4a 4d 4b 3f 30 2a 30 3b 20 20 20 20 4e 34 4f 4c 47 4e 3a 4f 4f 35 34 31
38 33 32 53 5b 32 4b 48 5a 4d 3c 54 4f 4a 30 38 1f 20 20 20 4e 34 48 2b 31 35 3a 4f 4f
35 33 26 34 33 3c 31 32 4d 54 26 32 22 5d 47 4f 35 2a 47 20 60 5f 5f 31 5b 4b 3b 34 48
3a 3f 20 25 4c 4e 68 35 3f 00
+*2.HB-JZTH2-M10DDS39AZ0G    N47307E0060672854M380240049G    N47317E0060012986M41024005
+32239051G    N4735M:/054763QJONKJJMK?0*0;    N4OLGN:OO541832S[2KHZM<TOJ08    N4H+15:005
3&43<12MT&2"]GO5*G `__1[K;4H:? %LNh5?
root@dhcp-221:/home/jmfriedt/gr-acars/build# 
```

```
a=12618.228516 b=10892
dynamic threshold: 359.318146
fin=2677
l0=743
2b 2a 16 16 01 32 2e 48 42 2d 4a 5a 54 15 48 32 2d 02 4d 31 30 44 44 53 33 39 41 5a 30 47 20 20 20 20 4e 34 37 33 30 37 45 30 30 36 30 36 37 32 38
35 34 4d 33 38 30 32 34 30 30 34 39 47 20 20 20 20 4e 34 37 33 31 37 45 30 30 36 30 30 31 32 39 38 36 4d 34 31 30 32 34 30 30 35 30 47 20 20 20 20
4e 54 48 4c 4c 4c 3a 4f 4f 4a 35 33 38 33 30 37 33 0d 2b 33 32 32 33 39 30 35 31 47 20 20 20 20 4e 34 37 33 35 4d 3a 2f 30 35 34 37 36 33 51 4a 4f 4e
4b 4a 4a 4d 4b 3f 30 2a 30 3b 20 20 20 20 4e 34 4f 4c 47 4e 3a 4f 4f 35 34 31 38 33 32 53 5b 32 4b 48 5a 4d 3c 54 4f 4a 30 38 1f 20 20 20 4e 34 48 2b 31
35 3a 4f 4f 35 33 26 34 33 3c 31 32 4d 54 26 32 22 5d 47 4f 35 2a 47 20 60 5f 5f 31 5b 4b 3b 34 48 3a 3f 20 25 4c 4e 68 35 3f 00
+*2.HB-JZTH2-M10DDS39AZ0G    N47307E0060672854M380240049G    N47317E0060012986M410240050G    NTHLLL:OOJ5383073
+32239051G    N4735M:/054763QJONKJJMK?0*0;    N4OLGN:OO541832S[2KHZM<TOJ08    N4H+15:OO53&43<12MT&2"]GO5*G `__1[K;4H:? %LNh5?
debut 851954

>>> Done
```

FIGURE 16 – The signal processing block with the addition of two arguments, a threshold value (floating point number) and the name of the file in which the decoded data are to be stored (character string). We observe in the `xterm` that the data are indeed recorded in the file located in the expected place, and that the stored values are consistent with those displayed by `gnuradio-companion`.

5.4 Dynamic parameter definition

Defining parameters following the previous method allows for the definition of the argument value (`seuil`) when the `gnuradio-companion` application is launched by transferring the parameter to the constructor of the class associated with the processing block. Once this parameter has been defined, the user is no longer able to change its value while the program is running. However, most blocks allow for tuning some of the variables from graphical sliders : how it this achieved ?

An additional argument when declaring the block structure in the XML file provides the means to define the method for updating a variable when the associated parameter value in the graphical interface is changed : in the file found in the `grc` directory, we add to the block definition the name of the method called when upating the parameter through the `callback` argument :

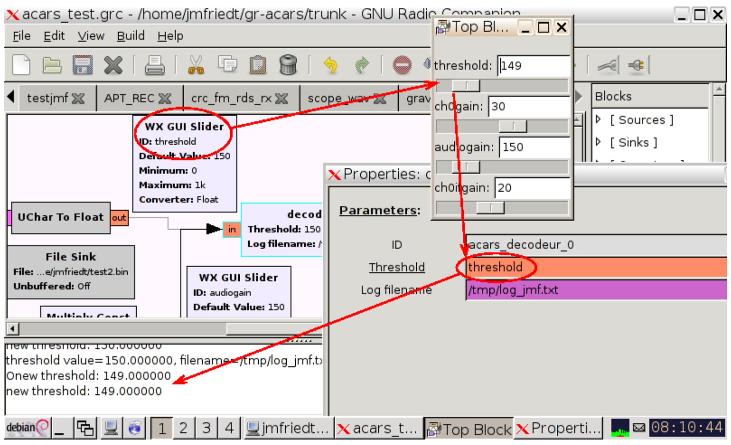
```
  <make>acars.decodeur($seuil,$filename)</make>
  <callback>set_seuil($seuil)</callback>
  <param>
    <name>Threshold</name>
    <key>seuil</key>
    <type>real</type>
  </param>
```

whose associated signature is declared in the class interface definition file in the `include` directory :

```
 public:
        ~acars_decodeur ();
        void set_seuil(float seuil1);
```

and finally in the library itself, we implement the *setter* method which updates the content of the (private) variable :

```
void acars_decodeur::set_seuil(float seuil1)
{printf("new threshold: %f\n",seuil1);fflush(stdout);_seuil=seuil1;}
```



```
new threshold: 150.000000
threshold value=150.000000, filename=/tmp/log_jmf.tx
Onew threshold: 149.000000
new threshold: 149.000000
```

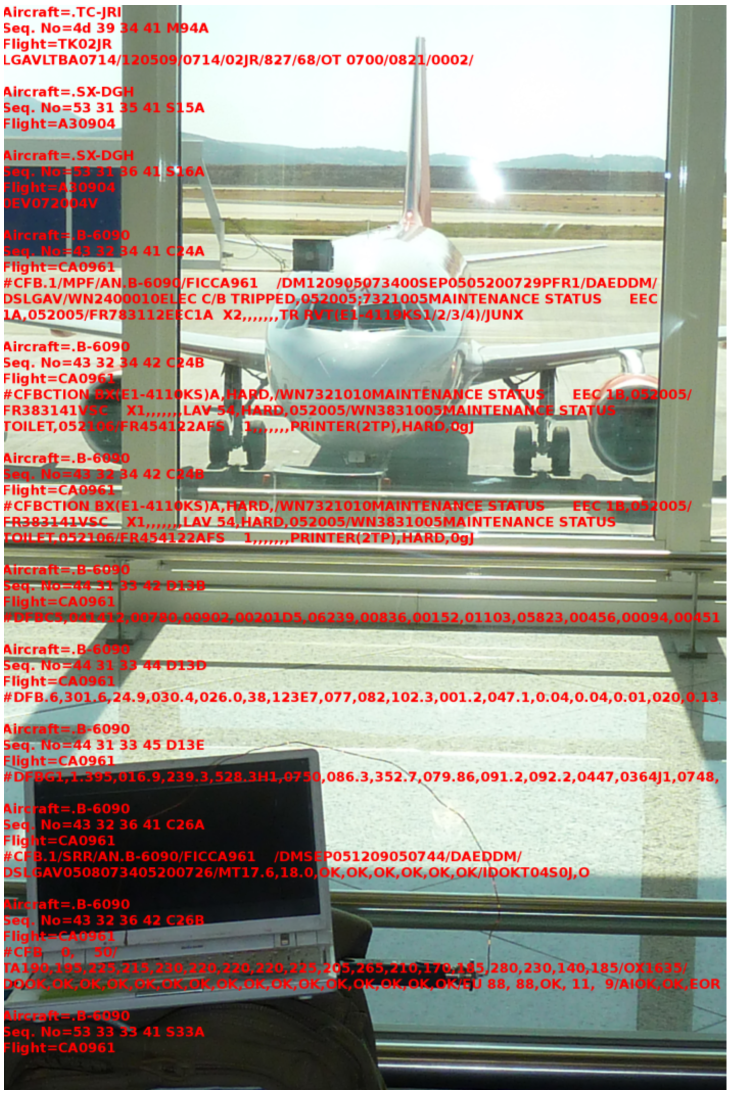
Figure 17 – The block for decoding ACARS sentences now accepts a dynamic threshold value modification through a slider of the graphical user interface (block *slider*) with an update of the value even though the application is being executed. In this example, we observe that a slider associated with the `threshold` variable has been defined to a default value of 150 (as seen on the terminal window at the bottom of `gnuradio-companion`), and changing this value in the text-mode entry window indeed changes the parameter value during the sentence decoding.

Thus, we check that every time the parameter `seuil` is modified by moving the slider in the graphical interface, the value is indeed updated in the associated signal processing block (Fig. 17).

The archive resulting from all these developments (Figs. 18 et 19), including all the concepts developed throughout this article, is available at `http://jmfriedt.free.fr/gr-acars.tar.gz` or on the CGRAN site at `www.cgran.org/wiki/ACARS`.

6 Conclusion

We have aimed at providing a working method to use at best a radiofrequency receiver providing an I and Q datastream in order to decode sentences of digital modes transmitted through a radiofrequency link. We have specifically focused on the low cost peripherals developed around the E4000 chip coupled to the RTL2832U analog to digital converter.



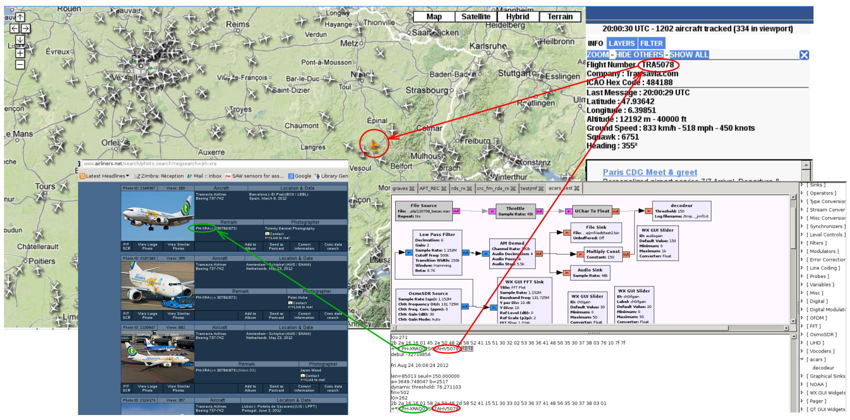
```
Aircraft=.TC-JRI
Seq. No=4d 39 34 41 M94A
Flight=TK02JR
LGAVLTBA0714/120509/0714/02JR/827/68/OT 0700/0821/0002/

Aircraft=.SX-DGH
Seq. No=53 31 35 41 S15A
Flight=A30904

Aircraft=.SX-DGH
Seq. No=53 31 36 41 S16A
Flight=A30904
0EV072004V

Aircraft=.B-6090
Seq. No=43 32 34 41 C24A
Flight=CA0961
#CFB.1/MPF/AN.B-6090/FICCA961    /DM120905073400SEP0505200729PFR1/DAEDDM/
DSLGAV/WN2400010ELEC C/B TRIPPED,052005;7321005MAINTENANCE STATUS      EEC
1A,052005/FR783112EEC1A  X2,,,,,,,TR RVT(E1-4119KS1/2/3/4)/JUNX

Aircraft=.B-6090
Seq. No=43 32 34 42 C24B
Flight=CA0961
#CFBCTION BX(E1-4110KS)A,HARD,/WN7321010MAINTENANCE STATUS      EEC 1B,052005/
FR383141VSC    X1,,,,,,,LAV 54,HARD,052005/WN3831005MAINTENANCE STATUS
TOILET,052106/FR454122AFS    1,,,,,,,PRINTER(2TP),HARD,0gJ

Aircraft=.B-6090
Seq. No=43 32 34 42 C24B
Flight=CA0961
#CFBCTION BX(E1-4110KS)A,HARD,/WN7321010MAINTENANCE STATUS      EEC 1B,052005/
FR383141VSC    X1,,,,,,,LAV 54,HARD,052005/WN3831005MAINTENANCE STATUS
TOILET,052106/FR454122AFS    1,,,,,,,PRINTER(2TP),HARD,0gJ

Aircraft=.B-6090
Seq. No=44 31 33 42 D13B
Flight=CA0961
#DFBC5,041412,00780,00902,00201D5,06239,00836,00152,01103,05823,00456,00094,00451

Aircraft=.B-6090
Seq. No=44 31 33 44 D13D
Flight=CA0961
#DFB.6,301.6,24.9,030.4,026.0,38,123E7,077,082,102.3,001.2,047.1,0.04,0.04,0.01,020,0.13

Aircraft=.B-6090
Seq. No=44 31 33 45 D13E
Flight=CA0961
#DFBG1,1.395,016.9,239.3,528.3H1,0750,086.3,352.7,079.86,091.2,092.2,0447,0364J1,0748,

Aircraft=.B-6090
Seq. No=43 32 36 41 C26A
Flight=CA0961
#CFB.1/SRR/AN.B-6090/FICCA961    /DMSEP051209050744/DAEDDM/
DSLGAV0508073405200726/MT17.6,18.0,OK,OK,OK,OK,OK,OK/IDOKT04S0J,O

Aircraft=.B-6090
Seq. No=43 32 36 42 C26B
Flight=CA0961
#CFB    0,    50/
TA190,195,225,215,230,220,220,220,225,205,265,210,170,185,280,230,140,185/OX1635/
DOOK,OK,OK,OK,OK,OK,OK,OK,OK,OK,OK,OK,OK,OK,OK,OK/EU 88, 88,OK, 11,  9/AIOK,OK,EOR

Aircraft=.B-6090
Seq. No=53 33 33 41 S33A
Flight=CA0961
```

FIGURE 18 – Example of ACARS sentences acquired using the EZCAP receiver fitted with an antenna made of a $\simeq$50 cm long piece of enameled copper wire. Such an activity makes the hours waiting for plane departure in airports much more fun, here in Athens (Greece).

Beyond the basic use of the available signal processing blocks provided by the `gnuradio` environment, we have identified the dataflow in order to develop our processing blocks. We have demonstrated the ability to implement such blocks for real time processing of the datastream provided by the receiver by first prototyping under GNU/Octave on records, before converting the algorithm to C for inclusion in `gnuradio`.

Among the perspectives, exploiting more "interesting" signals such as GPS[23] or GSM might be ambitious but certainly source of many new discoveries.

23. `http://michelebavaro.blogspot.fr/2012/04/spring-news-in-gnss-and-sdr-domain.html` but most significantly `http://www.gnss-sdr.org/documentation/gnss-sdr-operation-realtek-rtl2832u-usb-dongle-dvb-t-receiver` demonstrate that impressive results can be achieved by using a digital television receiver for receiving signals from satellites located 20000 km away


20:00:30 UTC - 1202 aircraft tracked (334 in viewport)

Flight Number : TRA5078
Company : Transavia.com
ICAO Hex Code : 484188

Last Message : 20:00:29 UTC
Latitude : 47.93642
Longitude : 6.39851
Altitude : 12192 m - 40000 ft
Ground Speed : 833 km/h - 518 mph - 450 knots
Squawk : 6751
Heading : 355º

FIGURE 19 – Real time analysis of the ACARS sentences using our `gr-acars` module : the web site `radarvirtuel.com/index-fr.html` confirms the presence, close to Besançon (location of the receiving antenna), of a Transavia plane, while the web site `airliners.net` associates a plane registration (PH-RXA) to a owner. Information acquired through ACARS are thus consistent with the other data available on web sites dedicated to air traffic monitoring, and thus ideally complements the ADS-B presented at the beginning of this document.

Acknowledgements

D. Bodor has informed us of the availability of the E4000+RTL2832U based DVB receivers and the associated trend in using such devices for SDR purposes. The members of the `osmocom-sdr` mailing list have answered several technical issues concerning the radiofrequency receivers based on these two chips. The equatorian government deservers our gratitude, if only for hosting the `http://gen.lib.rus.ec/` web site in which most documents described in the bibliography were obtained, and for political statements beyond the topic of this article. J. Boibessot (Armadeus Systems) and É. Carry (FEMTO-ST) have proofread the French veresion of this manuscript.

Références

[1] K. Borre, D.M. Akos, N. Bertelsen, P. Rinder & S.H. Jensen, *A Software-Defined GPS and Galileo Receiver : A Single-Frequency Approach*, Birkhäuser Boston (2007) as well as the slides available at `http://kom.aau.dk/project/softgps/` and most significantly `http://kom.aau.dk/project/softgps/GNSS_SummerSchool_DGC.pdf`

[2] J. Hamkins & M.K. Simon, *Autonomous Software-Defined Radio Receivers for Deep Space Applications*, Deep Space Communications and Navigation Series Vol. 9, available at `http://descanso.jpl.nasa.gov/Monograph/series9/Descanso9_Full_rev2.pdf`

[3] P.B. Kenington *RF and baseband techniques for software defined radio* Artech House (2005)

[4] SDR related litterature is full of more or less interesting books. Among those we have found least interesting since too remote from practical applications, W. Tuttlebee, *software defined radio – enabling technologies* Wiley (2002), or M. Dillinger, K. Madani, N. Alonistioti, *Software Defined Radio – Architectures, Systems and Functions* Wiley (2003), can be avoided by the reader with little fear of missing any significant information.

[5] J.-M Friedt, *Satellite image eavesdropping : a multidisciplinary science education project*, European Journal of Physics, **26** (August 2005) pp.969-984 `http://jmfriedt.free.fr/ejp196357p16.pdf`,

[6] Note d'application AN2668, *Improving STM32F101xx and STM32F103xx ADC resolution by oversampling*, ST Microelectronics, 2008

[7] D.J. Mudgway, *Uplink-Downlink – A History of the Deep Space Network, 19571997*, NASA SP-2001-4227, The NASA History Series (2001), available at `history.nasa.gov/SP-4227/Uplink-Downlink.pdf`

[8] M.K. Simon, *Bandwidth-Efficient Digital Modulation with Application to Deep-Space Communications*, Deep Space Communications and Navigation Series Vol. 3, available at `http://descanso.jpl.nasa.gov/Monograph/series3/complete1.pdf`

[9] T.G. Thomas & S. Chandra Sekhar, *Communication Theory*, Tata Mcgraw Hill Publishing (2006)

[10] N. Foster, *Tracking Aircraft With GNU Radio*, GNU Radio Conference (2011), available at `http://gnuradio.org/redmine/attachments/download/246/06-foster-adsb.pdf`

[11] T. McDermott, *Wireless Digital Communications : Design and Theory 2nd Ed.*, Tucson Amateur Packet Radio Corporation – TAPR (1998)

[12] Agilent, *Digital Modulation in Communications Systems – An Introduction*, Application Note 1298 available at `cp.literature.agilent.com/litweb/pdf/5965-7160E.pdf`, or M. Steer, *Microwave and RF design – a systems approach*, SciTech Publishing, Inc (2010) whose first chapter is available at `http://www.ece.ucsb.edu/yuegroup/Teaching/ECE594BB/Lectures/steer_rf_chapter1.pdf`

[13] J.-M Friedt, *Auto et intercorrélation, recherche de ressemblance dans les signaux : application à l'identification d'images floutées*, GNU/Linux Magazine France 139 (Juin 2011), available at `http://jmfriedt.free.fr/`