



M Ű E G Y E T E M 1 7 8 2

Budapest University of Technology and Economics
Department of Broadband Infocommunications and Electromagnetic Theory

András Retzler

Integrating digital demodulators into OpenWebRX

MSc Thesis

Thesis supervisor:

Péter Horváth, PhD

Associate Professor

Contents

1 Abstract.....	5
1.1 Összefoglaló.....	6
2 Introduction to OpenWebRX and digital modulations.....	7
2.1 Recent improvements on OpenWebRX and CSDR.....	9
2.2 CSDR used for tracking Schiaparelli EDM lander.....	10
3 Digital modulations.....	12
3.1 Motivation behind digital mode support.....	12
3.2 Results of modifications.....	14
3.3 Changes to the OpenWebRX protocol.....	15
3.4 Digital modulations in general.....	16
3.5 Functional stages of processing in a digital modem.....	19
3.6 Digital modes on amateur radio bands.....	21
3.7 Digital modes in consumer devices.....	22
4 A BPSK31 demodulator.....	23
4.1 Implementation with CSDR.....	25
4.2 Automatic gain control.....	27
4.3 Symbol timing recovery.....	28
4.4 Implementation of non-data aided timing recovery.....	34
4.5 Symbol timing recovery performance measurements.....	38
4.6 Carrier recovery.....	41
4.7 Synchronization in a differential PSK receiver.....	46
4.8 Varicode decoder.....	48
4.9 Resource usage.....	50
5 More applications of CSDR.....	52
5.1 Generating BPSK31.....	52
5.2 Demodulating BPSK63.....	53
5.3 BPSK31 with different pulse shape.....	54
5.4 Costas loop for carrier recovery.....	55
5.5 Demodulating RTTY.....	59
5.6 FSK demodulation.....	61
5.7 Serial line decoder.....	64

5.8 Baudot decoder.....	65
5.9 Demodulating M-FSK.....	66
5.10 Demodulating FSK signals transmitted using the CC1111 wireless MCU.....	67
5.11 Generating additive white Gaussian noise.....	73
5.12 Functions helping development.....	75
6 Conclusion.....	77
7 Acknowledgement.....	78
8 Bibliography.....	79

HALLGATÓI NYILATKOZAT

Alulírott Retzler András, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2017. 05. 21.

.....
Retzler András

1 Abstract

OpenWebRX, a web-based SDR receiver application (the topic of my Bachelor's thesis), is an online communication receiver that supports AM/FM/SSB/CW demodulation, and its web user interface helps accurate tuning with a real-time updated spectrogram of the received band. It allows remote access over the Internet, and multiple users can use it for receiving different signals simultaneously.

Throughout my Master's thesis project, I have added a demodulator for BPSK31, which is commonly used on amateur radio bands today, to the server-side signal processing. The user can select a BPSK31 signal to receive by clicking on the waterfall diagram on the web user interface, and the decoded data appears in the browser afterwards.

The CSDR software package that carries out digital signal processing has been extended with new functions for BPSK31 and RTTY demodulators in a way that it is possible to reuse the same functions for receiving other BPSK and M-FSK modulated signals as well. This work included implementing several fundamental synchronization techniques used in digital demodulators.

1.1 Összefoglaló

Az OpenWebRX webes rádióvevő alkalmazás (amely a BSc szakdolgozatom témája volt), egy olyan kommunikációs vevőt valósít meg, amely támogatja az AM/FM/SSB/CW üzemmódú jelek vételét, és webes kezelőfelülete a vételi sávról készített, valós időben frissített spektrummal segíti a hangolást. Lehetővé teszi az Interneten keresztüli távoli elérést, és egyszerre több felhasználó is használhatja különböző jelek vételére.

MSc diplomamunkám során a rádióamatőr sávokban gyakran használt BPSK31 jelek vételére alkalmas demodulátorral egészítettem ki a szerveroldali jelfeldolgozást. A webes felületen a felhasználó a vízésés diagramra kattintva kiválaszthatja a venni kívánt BPSK31 jelet, ezt követően a böngészőben megjelenik a dekódolt adat.

A jelfeldolgozást végző CSDR programcsomag kiegészült a BPSK31 és RTTY demodulátorokhoz szükséges funkciókkal, ezek azonban úgy lettek megvalósítva, hogy használhatók legyenek más BPSK és M-FSK modulált jelek vételére is. A munkához hozzá tartozott egyes, a digitális demodulátorokban használt alapvető szinkronizációs módszerek implementálása.

2 Introduction to OpenWebRX and digital modulations

OpenWebRX is an open source software defined radio (SDR) receiver application with a web interface, which allows amateur radio operators to set up remote receivers accessible over the Internet. OpenWebRX supports a variety of SDR hardware peripherals as input devices, and it can demodulate analog AM, FM, SSB and CW signals, which are actively used on the amateur radio bands today.

OpenWebRX consists of a server application that performs the demodulation and streams the resulting audio to the clients, and also a frontend application that provides a convenient user interface in the client web browser (see Figure 1 and 2). Multiple users can connect to the server at the same time, and each of them can select a channel to listen to, by clicking on a particular signal on the waterfall display.

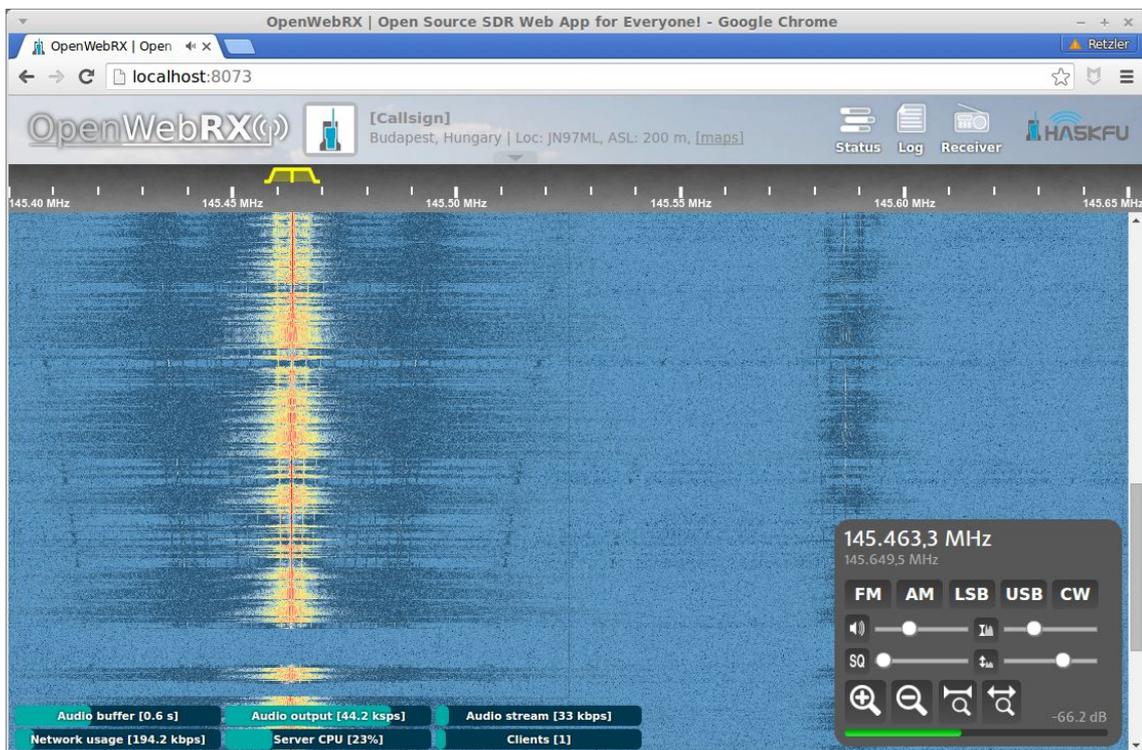


Figure 1: Original web user interface of OpenWebRX

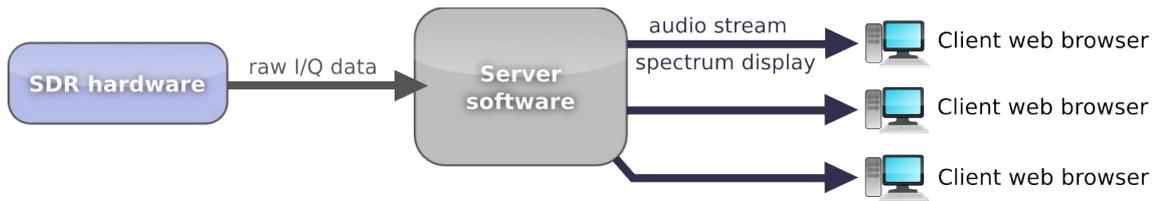


Figure 2: Block diagram of OpenWebRX web-based SDR software [2]

The technologies used to build the web interface include HTML5 and Javascript, with the backend implemented in python. In the background, a digital signal processing (DSP) library, *libcsdr* and a command-line tool, CSDR is performing the signal processing tasks. As good performance was a design goal for CSDR, it was implemented in C/C++. The build scripts allow us to constantly monitor the auto-vectorization results of the GCC compiler. To accelerate execution on ARM CPUs, some algorithms were optimized with inline assembly.

CSDR was designed to perform simple DSP on signals directly from the command-line, and today its use is not limited to OpenWebRX. CSDR processes with different processing functions can be chained after each other via FIFOs provided by the Linux kernel, thus a simple signal processing dataflow system can be implemented with a single command (see Figure 3).

```

    pcfl@ssd-mint ~ $ rtl_sdr -s 2400000 -f 145000000 -g 20 - | csdr convert_u8_f | \
    > csdr shift_addition_cc -0.145833 | \
    > csdr fir_decimate_cc 50 0.005 HAMMING | \
    > csdr fmdemod_quadri_cf | csdr limit_ff | csdr deemphasis_nfm_ff 48000 | \
    > csdr fastagc_ff | csdr convert_f_i16 | mplayer -cache 1024 -quiet \
    > -rawaudio samplesize=2:channels=1:rate=48000 -demuxer rawaudio -
  
```

Figure 4: Using CSDR to build an NFM demodulator from the command-line

While finishing my Bachelor's thesis on OpenWebRX and CSDR, the source code of the software projects have been published as open source on the GitHub project hosting website.

I have also built an own website under the domain name SDR.hu (see Figure 7), where people can access a list of publicly available OpenWebRX servers around the world. At the time of writing, the website lists more than 100 receivers on 6 continents. There are receivers using OpenWebRX (or its modified versions) in countries like the United States, Iceland, Australia and Japan.

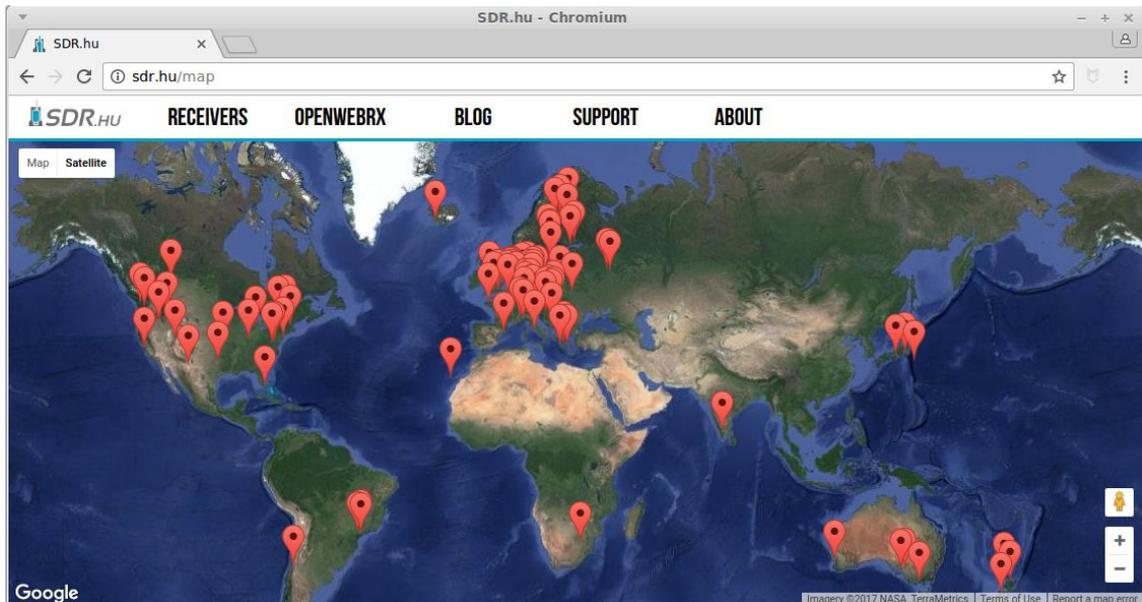


Figure 7: Map of active receivers on SDR.hu

In this chapter, I am writing about OpenWebRX and CSDR in general. In Chapter 3, I am telling more about the motivation behind digital mode support in OpenWebRX, and I am writing about the basics of digital modulations. In Chapter 4, each functional unit of the BPSK31 demodulator that has been integrated into OpenWebRX is described in detail. In Chapter 5, other results of this work are described, like an RTTY demodulator (not integrated into OpenWebRX yet), modified versions of the BPSK31 demodulator, and decoding the wireless transmissions sent by an MCU with built-in RF capabilities, using CSDR. I feel that all of these are closely related to the topic, and show various applications of the new features introduced in CSDR.

2.1 Recent improvements on OpenWebRX and CSDR

After finishing OpenWebRX as my Bachelor's thesis project in 2014, I continued to work on it, fixing software bugs and adding new features. Since the initial release, the following new features were added:

- The user interface has been improved with squelch and waterfall controls, waterfall color auto-adjustment, zoom buttons, and browser support for devices running iOS.
- OpenWebRX can now be used with most consumer SDR hardware devices

available on the market, including AirSpy, SDRPlay, HackRF, AFEDRI SDR, HPSDR, RFspace devices, FiFi SDR, Perseus and RTL-SDR in various modes. OpenWebRX supports many devices via the SoapySDR library and *rx_tools* command-line tools [1].

- Both the audio stream and the continuously updating waterfall display content is now compressed at the server and decompressed at the client, which results in an approximately 8 times decrease in network bandwidth usage compared to the uncompressed stream. The compression used is ADPCM.
- To allow DSP processing at the client for ADPCM decoding and resampling, *libcsdr* has been ported to Javascript with the Emscripten compiler. This approach builds on the current capabilities of web browsers on compiling Javascript to native code when the webpage loads, to speed up processing (Just In Time compilers). With the help of Emscripten, a unified codebase can be used for both the client and the server.
- The *libcsdr* library has been extended with SIMD optimized implementations of the FIR decimation for the ARM architecture. This resulted in approximately 300% speedup on some embedded platforms including the Raspberry Pi 2.
- I have released a related desktop SDR application for the Raspberry Pi 2, *qtcsdr*, which uses the Qt library for UI. It supports both reception and transmission through the *rtl-sdr* and *rpitx* projects, and also uses the CSDR tool for processing. While transmitting, the underlying *rpitx* tool (written by Evariste Courjaud, F5OEO) uses one of the GPIO pins of the Raspberry Pi 2 to generate an RF signal.

2.2 CSDR used for tracking Schiaparelli EDM lander

I have received e-mails from many people using OpenWebRX and CSDR for numerous purposes. CSDR has been used in high altitude balloon experiments, OpenWebRX has been used as the web front end of an FPGA based SDR platform, it has been bundled in a Linux distribution (SkyWave), and has been used at CubeSat ground stations.

However, the most interesting message arrived in November, 2016. I have received an e-mail from Stephan Esterhuizen, who works at NASA Jet Propulsion Laboratory. He

informed me that CSDR has been utilized at a ground station used during the ExoMars EDM landing attempt, and it formed a crucial part of their DSP pipeline.

Schiaparelli, the EDM (entry, descent and landing demonstrator) module is a technology demonstration vehicle which was carried by the ExoMars Trace Gas Orbiter, with the goal of demonstrating the capability of ESA to perform a controlled landing on the surface of Mars. During the landing process, the telemetry signals of Schiaparelli were monitored from the Giant Meterwave Radio Telescope (GMRT) in India (see Figure 5), with a phased array of 28 antennas of diameter 45 meters each. The CSDR software was used during the real-time detection of the very weak 5 Watt EDM transmitter, and it ingested data from the GMRT phased array at approximately 140 MBytes/second, where it went through various stages of mixing, filtering, decimating, and doppler corrections until a resolution bandwidth of 1 Hz was obtained. Schiaparelli's UHF signal was successfully detected and tracked from pre-atmospheric entry until about one minute before landing, when the signal was unfortunately lost. Figure 5 shows how the *baudline* tool was used to display a waterfall diagram after processing the signal with CSDR and other tools.

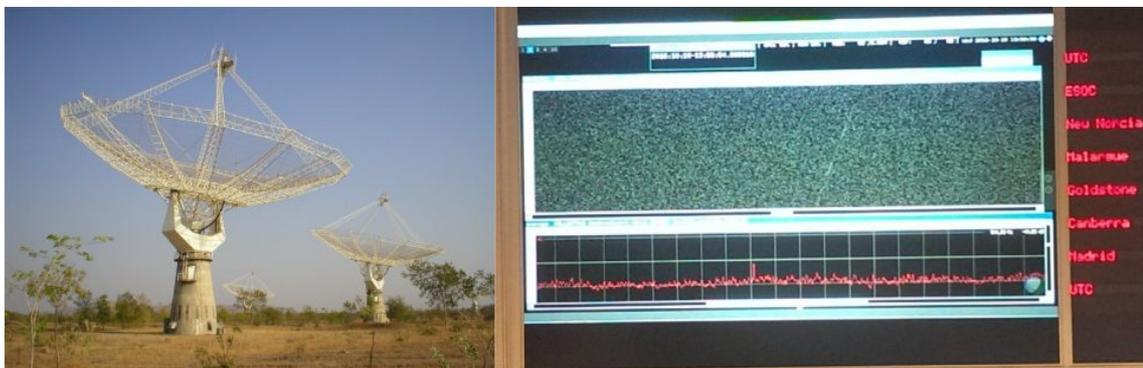


Figure 5: Antennas of the Giant Meterwave Radio Telescope [9] (on the left), the output of the DSP pipeline is displayed during the mission [26] (on the right).

3 Digital modulations

3.1 Motivation behind digital mode support

The original version of OpenWebRX was only able to demodulate analog audio transmissions like AM, SSB and FM. However, amateur radio operators also use digital modulations. In modes like BPSK31 and RTTY, the participants have a near real-time chat: the characters typed on their keyboards are modulated onto an RF carrier and decoded at the target, but various modes exist for sending files, pictures (digital SSTV), and also for extremely weak signal communication like EME.

There are several free and open source software packages to receive or transmit signals in a variety of digital modes (e.g. Fldigi, gMFSK, minimodem) and also commercial ones (Ham Radio Deluxe). To use these software, you have to connect the baseband input and output of an RF transceiver (typically an amateur radio transceiver with SSB mode) to the audio device of the PC.

The GUI of the Fldigi software is shown on Figure 6. It allows you to select a signal on the waterfall display, and decode it. After clicking on the T/R button, it starts to transmit the characters you type.

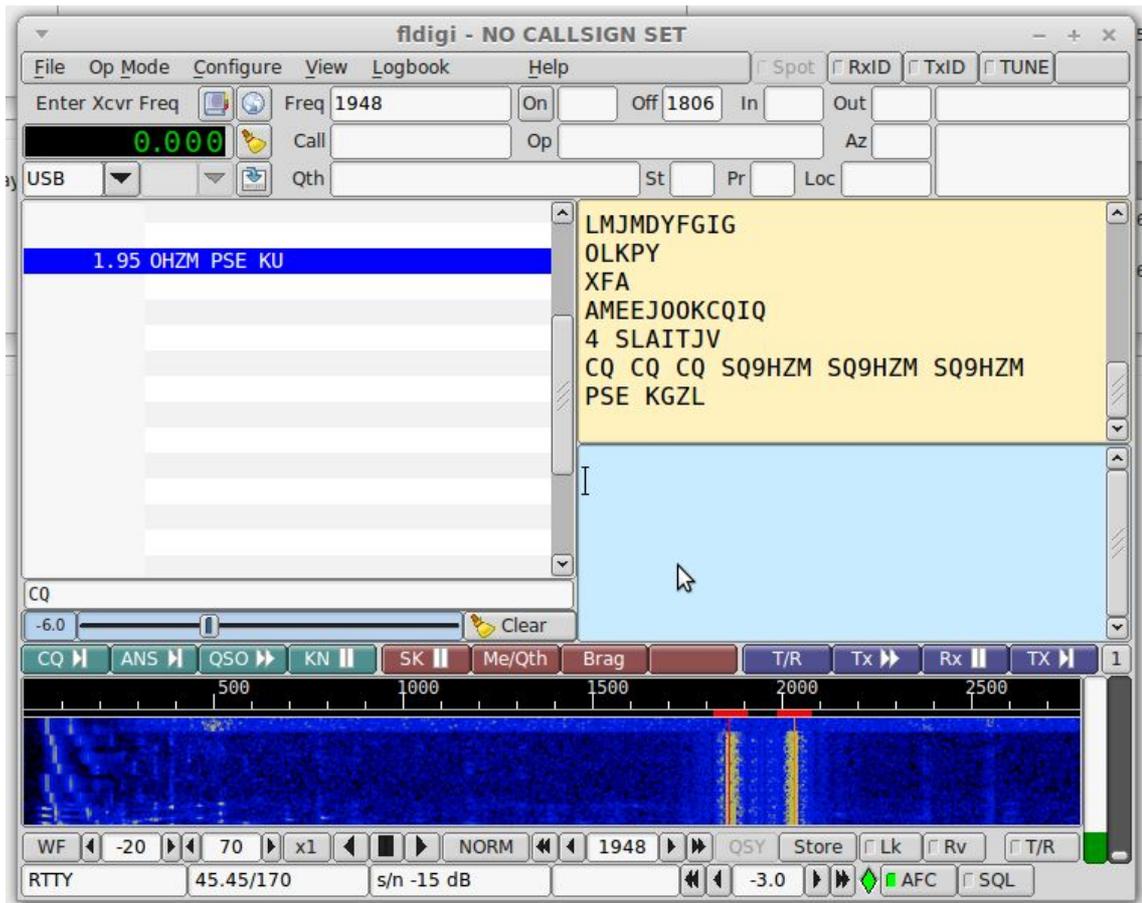


Figure 6: Fldigi software receiving RTTY transmission

However, if OpenWebRX users wanted to receive digital modes, they had to use such external software, that were designed for the audio input device. This meant a significant amount of effort: they needed to setup and configure a virtual audio device and connect it to the audio output of OpenWebRX, then configure the modem software to use the same virtual audio device. This process might involve setting up proprietary drivers on Windows (e.g. Virtual Audio Cable software), or configuring Pulseaudio through the *pavucontrol* GUI tool on Linux, possibly loading the *snd-aloop* kernel module. In addition, OpenWebRX uses a lossy audio compression by default, which means some degradation in signal-to-noise ratio (SNR), while it still makes possible to decode some digital modes if the signal is strong enough.

I realized that integrating the SDR software and the digital modem removed such barriers in front of users with less computer experience, and it would also allow people interested in ham radio to more quickly understand what digital modes are. OpenWebRX is meant to be a self-contained system: all the users would have to do is to

open the website, click on a signal and see the decoded data.

I also observed that most open source modem software do not provide an easy to use API that would have allowed other applications to pass raw audio data into the decoder and acquire the text output. While it was possible to modify one of these to provide the proper APIs, I decided to write my own implementation extending *libcsdr* and the CSDR command line tool. This allowed me to learn more about the internals of modems. I have chosen the BPSK31 mode as the first to be implemented for OpenWebRX as it is one of the most widely used digital modes on amateur radio bands.

3.2 Results of modifications

OpenWebRX has been extended with the capability to demodulate BPSK31 transmissions. The underlying CSDR software also has new features, so that it can demodulate RTTY (2-FSK) and BPSK31 transmissions, it can generate BPSK31 transmissions and simulate an AGWN channel. GNU Octave scripts have been developed to calculate the S-curve of the error detectors and the variance graph of the estimators.

Figure 7 and 8 show the new user interface of OpenWebRX, while a BPSK31 transmission is being demodulated. When the webpage of an OpenWebRX receiver is opened, one of the analog demodulators is selected by default. The user can select a digital demodulator from the appropriate listbox (next to the DIG button), or click on the DIG button to select the last used digital demodulator.

In this case a new panel pops up, which contains the waterfall diagram of the baseband signal. The user can select which channel to decode by clicking on the diagram. The decoded data shows up under the waterfall diagram.

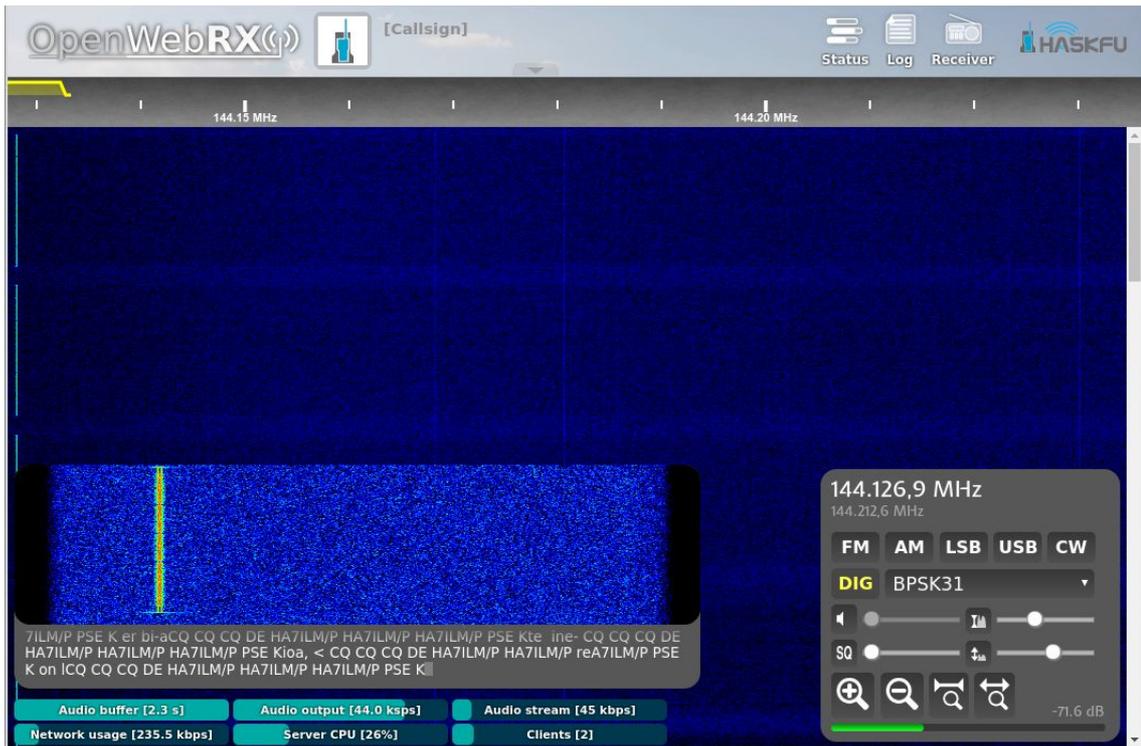


Figure 7: New user interface of OpenWebRX with integrated BPSK31 demodulator

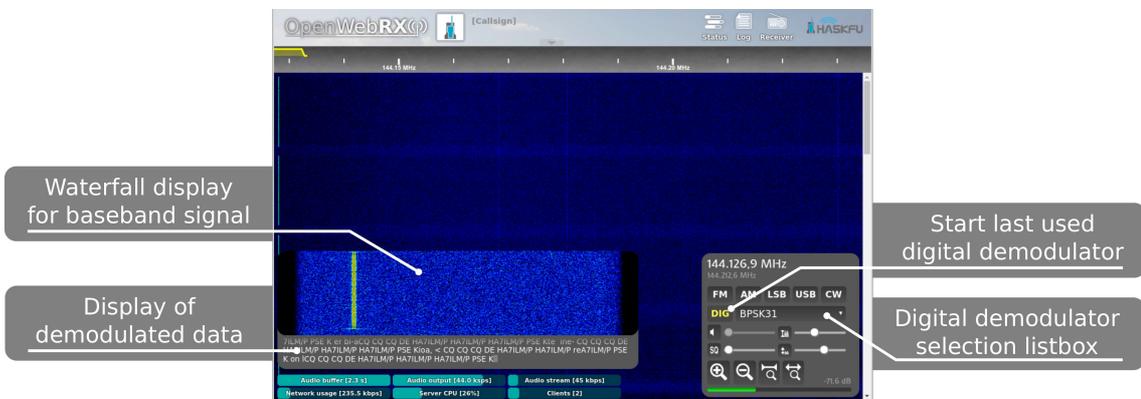


Figure 8: New parts of the web user interface

3.3 Changes to the OpenWebRX protocol

To display BPSK31 demodulated data on the web UI, both the OpenWebRX server and frontend code have been modified. These components communicate with each other over WebSockets, with a custom protocol [2, pp. 30]. This protocol has been extended with new messages, see Table 1. The term "secondary receiver" is used for the digital

mode demodulator. It is called secondary because it works in parallel with an analog demodulator that still provides audio output. The input of the secondary receiver is the IF signal containing only the channel selected on the main waterfall diagram.

Source	Example message	Notes
Client	SET secondary_mod=bpsk31	Switch secondary receiver on
Server	MSG secondary_fft_size=1024 if_samp_rate=12000 secondary_bw=31.25 secondary_setup	Send secondary receiver setup parameters
Server	FFTS<an array of 1024 floating point values>	Secondary FFT data
Server	DAT <text>	Secondary demodulator output
Client	SET secondary_offset_freq=524	Change offset frequency

Table 1: New messages in OpenWebRX protocol

3.4 Digital modulations in general

Digital modulations mean binary data transmitted over one or more modulated carriers. The input of the modulator and the output of the demodulator is a discrete signal, however, the signal passes through an analog wireless channel between them.

Most of the early work on digital modulations originates from the time when computer systems were interconnected over wired telephone lines. Many of the concepts developed for telephone modems were later reused for wireless transmission [3, pp. 3].

In a digital modem, the demodulator is traditionally more advanced to implement than the modulator, as it has to deal with tasks like synchronization, decoding, and problems like variations in signal strength and additive noise on the channel.

Binary phase-shift keying (BPSK) is a simple example on how we can alter the phase of a single sine wave to encode data. You can inspect the time domain signal and the corresponding constellation diagram of a BPSK signal on Figure 9.

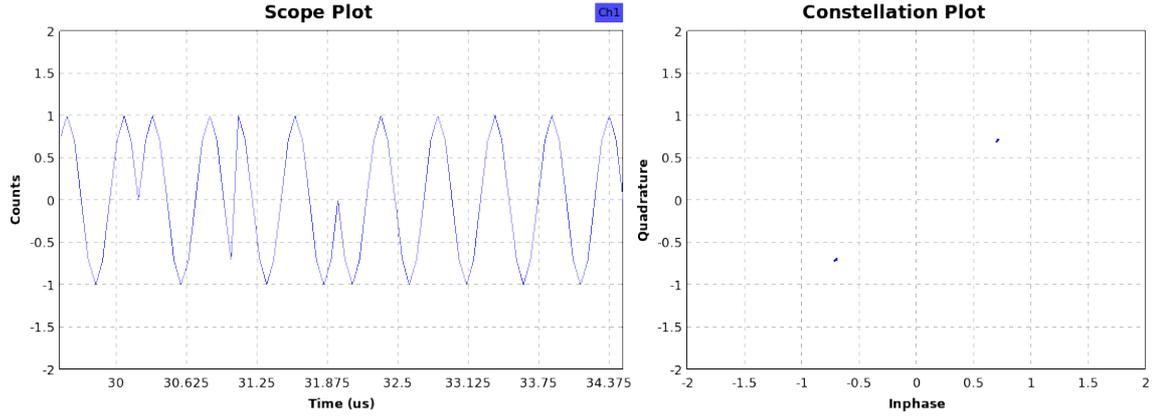


Figure 9: Simulation of BPSK signal in GNU Radio

For quadrature amplitude modulation (QAM), both the amplitude and the phase, and for frequency-shift keying (FSK), the frequency are the physical values used for the purpose of encoding discrete symbols.

Immediate changes in phase, amplitude or frequency have the drawback of having a very high signal bandwidth, thus making it inefficient to use on real wireless channels, where bandwidth is a finite resource. Pulse shaping is used to limit the signal bandwidth, and also to eliminate intersymbol interference (ISI).

The formula (1) describes a QAM modulated complex baseband signal [3, pp. 44]:

$$y(t) = \sum_{n=-\infty}^{\infty} s[n]g(t - nT_b) \quad (1)$$

- $s[n]$ is the value of the n -th discrete symbol (complex),
- $g(x)$ is the transmit filter or pulse shape (real-valued),
- T_b is the duration of each symbol (the reciprocal of the symbol rate).

The $g(x)$ function should be chosen so that its value is zero at the maximum amplitude of each neighboring symbol to eliminate ISI, see (2).

$$\forall n \in \mathbb{Z} \text{ if } n \neq 0 \text{ then } g(nT_b) = 0 \quad (2)$$

Such a function is the raised-cosine filter (3) [4]. If we take the square root of the impulse response of the raised-cosine filter, we get the root-raised-cosine (RRC) filter. If that is applied at both sides of the wireless channel, their overall impulse response will be the same as of the the raised-cosine filter (see Figure 10).

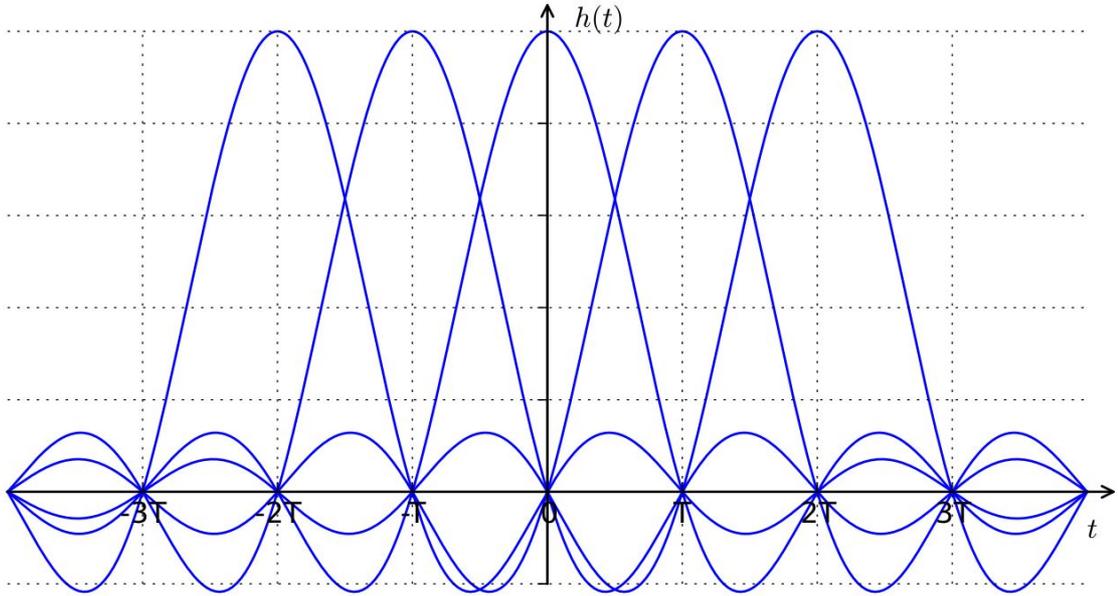


Figure 10: The effect of raised-cosine filtering on consecutive symbols [5]

$$h(t) = \begin{cases} \frac{\pi}{4} \operatorname{sinc}\left(\frac{1}{2\beta}\right), & t = \frac{\pm T}{2\beta} \\ \operatorname{sinc}\left(\frac{t}{T}\right) \frac{\cos\left(\frac{\pi\beta t}{T}\right)}{1 - \left(\frac{2\beta t}{T}\right)^2}, & \text{otherwise} \end{cases} \quad (3)$$

Special pulse shapes are also used to decrease the required bandwidth of the signal. SOQPSK-TG is a spectrally efficient modulation scheme used in aeronautics [6], that defines its own pulse shape. BPSK31 later covered in this thesis uses cosine shaping.

The modulations used typically have 2^N constellation points, which means that a group of bits are easily mapped to each point (and also that the constellation diagram is symmetric). However, during the demodulation process, usually there is an uncertain phase offset, which would result in the constellation points mapping to different symbols unless some kind of synchronization is carried out. For BPSK signals, one such scheme is called differential BPSK (DBPSK), where the symbols are encoded in the directions of the phase changes between constellation points, which are not affected by a phase offset.

The wireless channel is usually modeled with additive white Gaussian noise (AGWN). (Other distortions, including multipath propagation can also occur, and need to be modeled differently.) To keep the integrity of data bits over the wireless channel, forward error correction (FEC) techniques are used. These usually increase redundancy in the data stream, but if a small number of bits have a wrong value, they can detect or correct it in certain scenarios. (If too many bits are wrong, however, these algorithms also fail.) Convolutional codes, turbo codes and low-density parity check codes are commonly used today [3, pp. 2].

3.5 Functional stages of processing in a digital modem

Turning data into a modulated signal usually consists of the following stages [3, pp. 5]:

- encoding,
- bit-to-symbol mapping,
- pulse shaping,
- modulating carrier with amplitude / phase / frequency information.

Recovering the original data from the modulated signal involves the following stages of processing:

- carrier recovery,
- retrieving amplitude / phase / frequency from carrier,
- symbol timing recovery,
- equalization,
- symbol-to-bit mapping,
- decoding.

As the circuits of the receiver and the transmitter don't use the exact same clock source, the signal sampled by the receiver will have some unwanted properties. There will be a frequency and phase offset between the local oscillator in the receiver and the transmitter. For that reason, the receiver will "see" the constellation of the modulated baseband signal continuously rotating (4),

$$s_r(t) = s(t)e^{j(2\pi f_{\text{offset}}t + \varphi_{\text{offset}})} \quad (4)$$

where $s_r(t)$ is the received signal, $s(t)$ is transmitted signal, and f_{offset} is the frequency offset, φ_{offset} is the phase offset. Sometimes the amount of rotation also changes over time because the frequency of the oscillators in both the receiver and the transmitter can be affected by varying temperature.

Carrier recovery methods can compensate for the unknown frequency and phase offset by recovering the original carrier at the receiver, and downconverting the input signal with that. More information on this topic is available in section 4.6.

Furthermore, the clock frequency and phase will not be synchronized between the D/A converter in the transmitter, and the A/D converter in the receiver. As a consequence, in the receiver, we might be sampling the symbol at the wrong time, during transition. In order to maximize the performance, especially if additive noise is also present on the channel, we need to sample at the point where the symbol amplitude is maximal.

Looking at the eye pattern of the baseband signal, the maximum eye opening is the ideal sampling point, where the adjacent symbols have the least effect, and where the contribution of the convolved impulse response functions of the pulse shaping filter and the channel reach its highest value for the given symbol. Symbol timing recovery methods are detailed in section 4.3.

After the problems described above have been eliminated, recovering the amplitude and the phase or frequency from the modulated signal and applying symbol-to-bit mapping is usually an easy task: we need to decide which symbol the constellation point is the nearest to.

Another step, equalization is usually required to compensate for distortion caused by the wireless channel, for which the main source is multipath propagation, resulting in ISI. Adaptive filters can learn about the channel and realize an impulse response that is the inverse of it. Finally, the forward error correction used in the transmitter needs to be reversed.

3.6 Digital modes on amateur radio bands

Table 2 summarizes the properties of some popular digital modes used by amateur radio operators.

Mode	Modulation, baudrate
RTTY	FSK, 45.45 baud
BPSK31	BPSK, 31.25 baud
QPSK31	QPSK, 31.25 baud
WSPR	4-FSK, 1.4648 baud
JT65 [7]	64-CPFSK, 2.69 baud
Olivia-32-1K	32-FSK, 31.25 baud
Olivia-32-2K	64-FSK, 31.25 baud
Packet radio	FSK with 300/1200/2400/4800 baud, GFSK with 9600 baud

Table 2: Properties of some digital modes used on amateur radio bands

Some modes employ FEC techniques, while others do not. For example, BPSK31 and RTTY do not use any error correction, which make it easier to implement a receiver for them. QPSK31 and WSPR use convolutional codes, Olivia MFSK uses Walsh functions for FEC. Comparing BPSK31 and QPSK31, the latter can reduce error rate on typical radio paths, where the errors are in bursts rather than randomly spread, but it can do no improvement if the noise level is high in overall.

Some modes are especially tailored for weak signal communication, use a frame structure and can only carry a limited amount of information, like the amateur radio callsign of the operator or the transmitter output power. WSPR and JT65 are in this category, both developed by Joe Taylor, K1JT. These have a very low symbol rate and bandwidth in order to increase SNR at the receiver. JT65 is typically used for earth-moon-earth (EME) and meteor scatter contacts on the VHF and UHF bands. Tests showed that using JT65B, 96% of the messages were decoded in -23 dB SNR (measured in 2500 Hz bandwidth) [7].

Packet radio, which was used to build wireless computer networks, should also be mentioned. It was very popular before low-cost Internet access spread, and for many

amateur radio operators, it was the first experience with services (like bulletin boards, file access, etc.) that are nowadays part of the Internet culture. Its protocol originates from X.25, a modified version of which is called AX.25, where each packet contains the callsign of both the sender and the recipient. Nowadays the only widely used service still relying on AX.25 is the Automatic Packet Reporting System (APRS), with 1200 baud AFSK signals, typically transmitting over FM links on VHF.

3.7 Digital modes in consumer devices

While the highlight of my paper is digital modes on amateur radio bands, it should not be forgotten that many consumer devices use wireless communication: mobile phones, wireless keyfobs for cars, temperature sensors, etc.

Internet of Things (IoT) is also a common term used today, referring to embedded devices connected to the Internet. Nowadays very cheap RF SoCs are available, and such a SoC can include an MCU, a modem, up- and downconverters, and a PA in one package. As the central unit of an embedded device, it can control various peripherals of the system while it directly performs two-way communication on its radio interface.

As an example, ESP8266EX chips contain a 32-bit MCU running at 160 MHz along with an integrated WiFi controller [8], available under 1.5 USD per piece at the time of writing.

A variety of RF transceiver ICs are available for products using more simple wireless protocols. For example, the Texas Instruments CC1111 is a sub-GHz wireless MCU that can act as an FSK, GFSK, MSK, ASK, or OOK transceiver [10].

While CSDR does not support complex modulation schemes like OFDM, some of its functions could be reused to decode the signals of IoT devices using simple protocols and modulations, from the command-line, see section 5.10.

4 A BPSK31 demodulator

BPSK31 is a digital mode originally developed by Peter Martinez (G3PLX) in 1998. It has been popularized by the WinPSK software and its successors. It uses differential binary phase-shift keying (DBPSK) modulation with a symbol rate of 31.25 baud.

BPSK31 does not use any forward error correction, while its variant called QPSK31 uses convolutional code.

On Figure 11, a BPSK31 signal can be inspected. During symbol transitions, BPSK31 does not keep the amplitude constant, it rather has a sinusoidal envelope, with the complex baseband signal reaching zero in the middle of the transition.

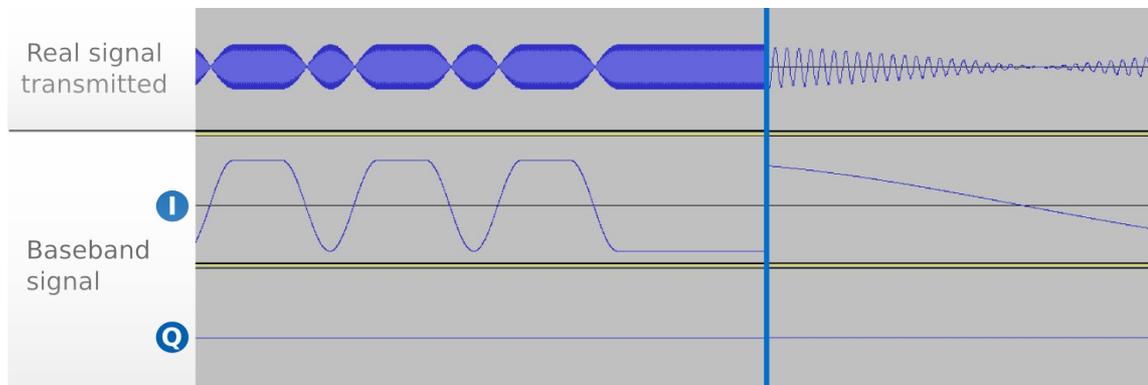


Figure 11: The real signal present on the antenna, and the downconverted, complex valued baseband signal. The same signals are shown with different zoom levels on the left and right.

We can write the baseband BPSK31 signal as (5).

$$y(t) = \sum_{n=-\infty}^{\infty} s[n]g(t-nT_b)$$

$$g(t) = \begin{cases} \frac{1 + \cos\left(\frac{t\pi}{T_b}\right)}{2} & \text{if } -T_b \leq t \leq T_b \\ 0 & \text{if } t > T_b \text{ or } t < -T_b \end{cases} \quad (5)$$

- $s[n]$ corresponds to the symbols (complex numbers, which can take the values $1+0j$ and $1-0j$),
- T_b is the duration of a single bit,
- $g(t)$ is the impulse response of the pulse shaping filter.

To verify this formula, I have created a GNU Octave script which can plot the impulse response of the pulse shaping filter, along with an example baseband signal for a few input symbols (see Figure 12). The script is now part of the CSDR distribution (*grc_tests/psk31_sigmodel.m*).

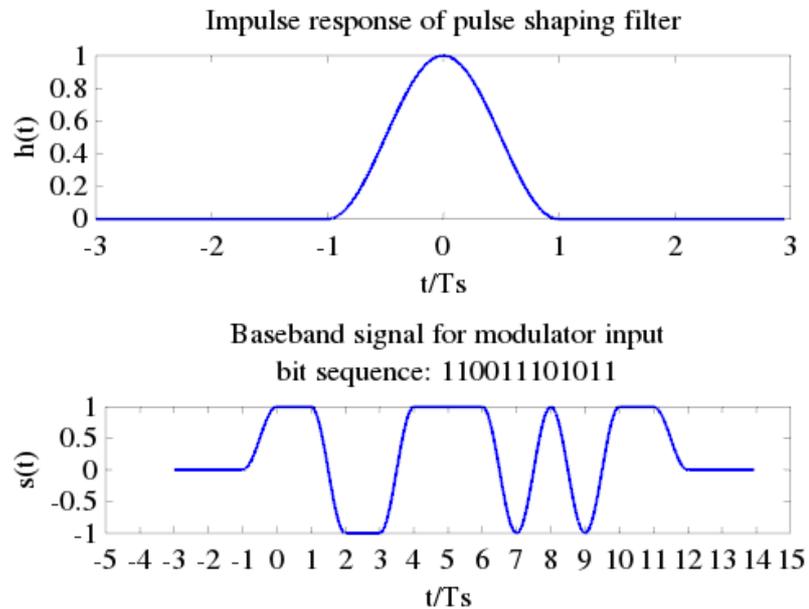


Figure 12: Output of GNU Octave script for analysis of BPSK31 baseband signals (T_s is the symbol duration)

While the sinusoidal pulse shape does not provide protection against ISI like the root-raised-cosine does, it decreases the signal bandwidth so that it is possible to have 256 separate channels even in a single 8 kbps audio stream. I have verified with spectrum analysis that the BPSK31 emission bandwidth is around 60 Hz at -26 dB (see Figure 13), which is identical to the results published on the ARRL website [11].

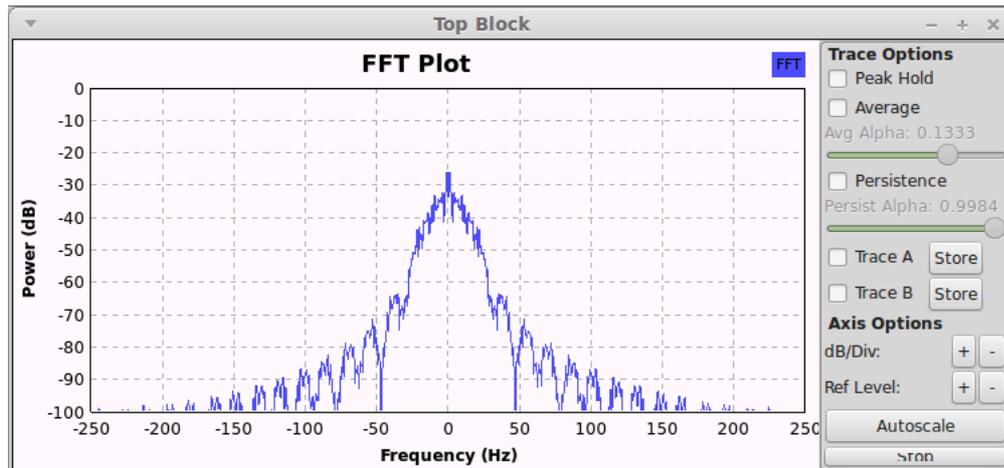


Figure 13: Spectrum and constellation diagrams of BPSK31 modulated signal

4.1 Implementation with CSDR

As the CSDR command-line tool can be used to build DSP chains in the command-line, running different DSP functions in different processes connected with OS pipes, the BPSK31 demodulator has been implemented in the same philosophy.

The command to demodulate a BPSK31 signal acquired from the audio card, centered at 1 kHz can be found below. OpenWebRX uses a similar command internally.

```
arecord -r48000 -c1 -fS16_LE | \
csdr convert_s16_f | \
csdr dsb_fc | \
csdr shift_addition_cc $(csdr '=-1000./48e3') | \
csdr fir_decimate_cc 32 | \
csdr bandpass_fir_fft_cc $(csdr '=- (31.25)/1.5e3') \
$(csdr '= (31.25)/1.5e3') $(csdr '=31.25/1.5e3') | \
csdr simple_agc_cc 0.001 0.5 | \
csdr timing_recovery_cc GARDNER 48 0.5 2 --add_q | \
CSDR_FIXED_BUFSIZE=1 csdr dbpsk_decoder_c_u8 | \
CSDR_FIXED_BUFSIZE=1 csdr psk31_varicode_decoder_u8_u8
```

The clean separation between building blocks of this processing chain allows us to easily derive modified digital modes from BPSK31. For example, changing the symbol rate is just a matter of changing a few parameter values in the command above (see also section 5.2). If one decided to add FEC to BPSK31, it required adding a pair of functions to CSDR (although this is not subject of this thesis). The easy fine tuning of parameters allows greater flexibility.

The following points contain an analysis of the command above.

- The *arecord* command reads the raw data from the audio device, using a sampling rate of 48000 sps, with 16 bit signed integer format.
- The *csdr convert_s16_f* command converts the data to floating point (the range $[-32768; +36767]$ is converted to $[-1.0; +1.0]$).
- A channelizer consisting of *csdr shift_addition_cc*, *csdr fir_decimate_cc* and *csdr bandpass_fir_fft_cc* selects a single channel with a bandwidth of ~62.5 Hz, sampled at 1.5 ksps out of the 48 ksps input signal.

The bandwidth of the BPSK31 signal is only 31.25 Hz, but we need some oversampling for the timing recovery function to work. In this case the signal is oversampled 48 times. I used a filter with a wider passband than needed because the user does not always tune to the signal very accurately on the web interface.

The *csdr =* function allows us to calculate a formula and insert the result as a parameter into another command. See section 5.12 about that.

- An AGC (*csdr simple_agc_cc*) keeps the signal power constant, however, it should not distort the amplitude changes at the transients between symbols. On that reason, its timing constant should be higher than the duration of a symbol.
- To find the maximum amplitude for each symbol within the signal, symbol timing recovery is used: *csdr timing_recovery_cc* implements non-data aided timing recovery methods. Here the Gardner method is used.

As *csdr timing_recovery_cc* only outputs one sample per symbol, the blocks afterwards work on signal sampled at approximately 31.25 sps. The default buffer size in CSDR is 1024 or 16384 samples (depending on function), at which real-time decoding would be impossible, as the processes first wait for the buffer to fill up before starting processing.

A recent feature set added to CSDR included an option to set the buffer sizes from environment variables. Setting the *CSDR_FIXED_BUFSIZE* variable to 1 means that the functions will work on a single sample and then wait until new data is available. This boosts the user experience much, the decoded text appears immediately.

- Carrier recovery, symbol decision and differential decoding is jointly done by *csdr dbpsk_decoder_c_u8*.
- The BPSK31 signal has the characters stored in Varicode encoding, so we need to convert Varicode to ASCII.

The following part contains a detailed description of the related new functions in CSDR.

4.2 Automatic gain control

At the time I started this project, the CSDR process has already contained multiple AGC functions:

- *csdr fastagc_ff* precisely adapts to the maximum value of the signal, to make sure that no clipping happens. It makes that possible by working on three buffers internally.
- *csdr agc_ff* models how a traditional AGC circuit works. Its attack and decay rates can be set separately, along with the hang time and attack wait time.

While these blocks all work on real signals, to build the data flow of my BPSK31 demodulator, I needed a block that can work on complex signals. In addition, I wanted to implement an algorithm that is really simple to use, without many different parameters to set.

The resulting function, *csdr simple_agc_cc* implements a control system, see Figure 14 for an overview.

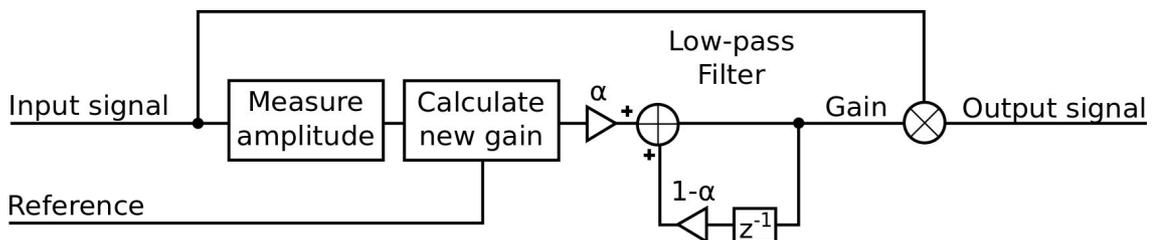


Figure 14: Block diagram of *simple_agc_cc*

A new gain value is calculated for every sample, but it is ran through a low-pass filter before it is applied, as in (6).

$$\text{Gain}_{\text{new}} = \text{LPF} \left(\frac{\text{Amplitude}(\text{Input signal})}{\text{Reference}} \right) \quad (6)$$

The LPF is a single-pole IIR filter (7), the α parameter of which is the *rate* parameter of *csdr simple_agc_cc*.

$$y[n] = \alpha x[n] + (1 - \alpha)y[n-1] \quad (7)$$

The τ time constant of the filter corresponding to α can be calculated as in (8).

$$T_s = \frac{1}{f_s} \quad (8)$$

$$\tau = T_s \left(\frac{1 - \alpha}{\alpha} \right)$$

In my BPSK31 demodulator, the AGC is applied to a signal sampled at 12 ksps, with $\alpha = 0.0001$ that corresponds to a time constant of 0.8322 seconds.

The syntax of the new AGC function in CSDR is detailed below. It also contains a parameter for the maximum possible gain output by the controller.

```
csdr simple_agc_cc <rate> [reference [max_gain]]
```

4.3 Symbol timing recovery

If the IF signal is sampled at 1500 sps, but our symbol rate is only 31.25 baud, then we need to select 31.25 samples from a set of 1500 samples every second, and decide if they correspond to a $(-1+0j)$ or a $(+1+0j)$ symbol ("0" or "1").

Figure 15 shows what happens in practice if the signal is noisy and we sample it with a wrong phase offset. Table 3 shows the symbol decisions made in this case.

Slicer output:	1 1 1 0 1 1 1 1 0 1 1 1
Original:	1 1 0 0 1 1 1 0 1 0 1 1

Table 3: Comparison of the symbol decision and the transmitted symbols in case of incorrect sampling phase

In contrast, if the signal is sampled at the correct locations, the symbol decision output equals to the original data transmitted, see Figure 16.

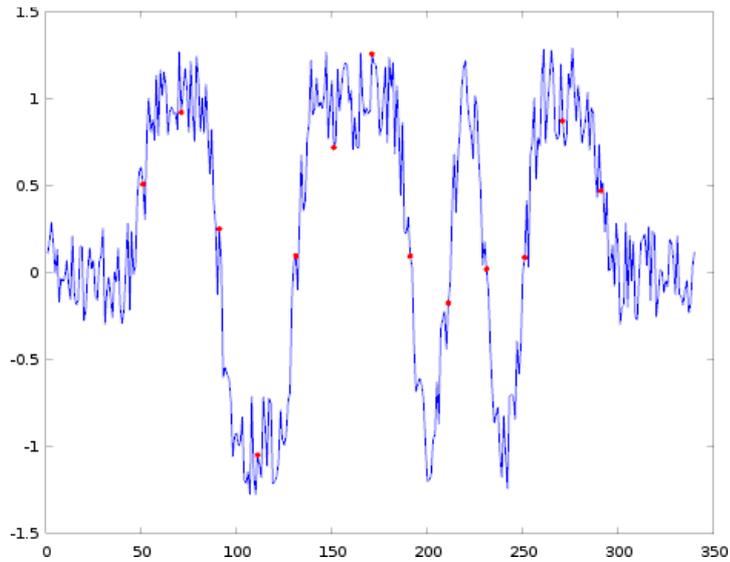


Figure 15: Noisy BPSK31 baseband signal sampled at wrong locations

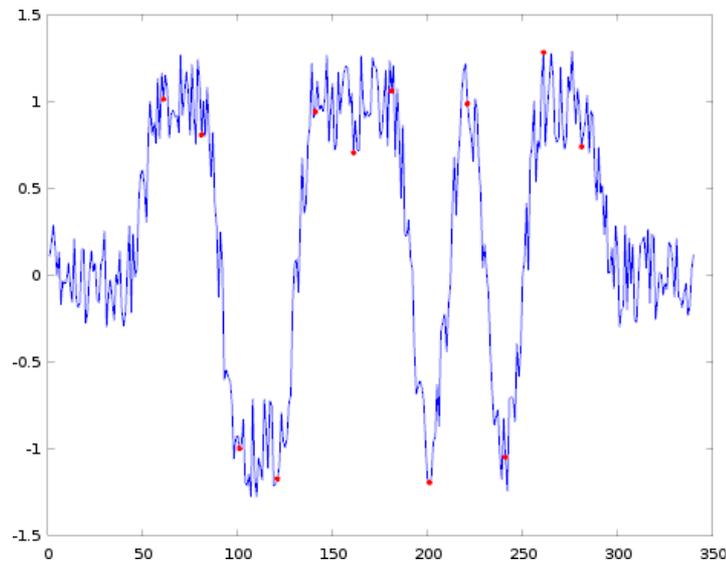


Figure 16: Noisy BPSK31 baseband signal sampled at the center of each symbol

A classification of timing recovery methods is available below [3, pp. 279].

- **Non-data aided timing recovery methods** only rely on the input signal, without any knowledge of the transmitted symbol or the symbol decision. They might be placed before the carrier recovery function in the flow graph. They provide a wider capture range and have better results at low SNR compared to decision-directed methods.

- The **early-late gate** timing recovery algorithm is based on a control system that tries to find the middle of a symbol by designating three points on it, and calculating the a phase offset correction at the next symbol based on the values of the input signal at these points. (9) is the formula [12] for the early-late timing error detector (TED) in case of a real input signal.

$$e[n] = y[nT + d[n]] \cdot (y[nT + 0.5T + d[n]] - y[nT - 0.5T + d[n]]) \quad (9)$$

- $e[n]$ is the calculated timing error for the n -th symbol, which is the sum of the timing error for the I and the Q branches ($e_I[n]$ and $e_Q[n]$).
- $y_I[n]$ and $y_Q[n]$ correspond to the real and the imaginary part of the input signal.
- T is the duration of a single symbol.
- $d[n]$ is the estimated phase offset for the n -th symbol.

Note that (9) is only the timing error detector, a single part of the control system. One way to update the phase offset from the error value is using a proportional controller, as presented in (10) [3, pp. 299], where μ is the update step-size (the proportional gain).

$$d[n] = d[n-1] - \mu \cdot e[n-1] \quad (10)$$

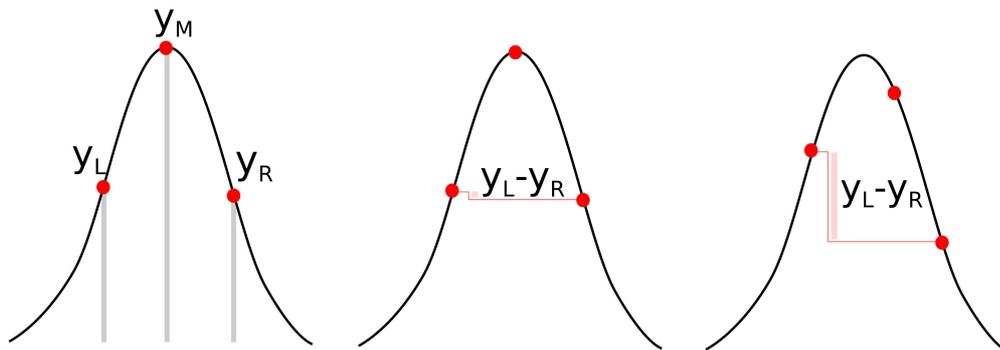


Figure 17: The early-late gate TED

Figure 17 allows us to explain the early-late gate timing error detector (TED) in an even more simple way. Based on the notation used on the figure, in case of a real signal, the error can be calculated as (11).

$$e = y_M \cdot (y_R - y_L) \quad (11)$$

- **The Gardner algorithm** (12) [13][14][15] (introduced by Floyd M. Gardner in 1986) is very similar to the early-late gate, but the loop rather tries to find the zero crossing with the middle point. The delay between its points is $\frac{1}{2} T$ (while it was $\frac{1}{4} T$ for the early-late algorithm).

$$e[n] = [y[(n-1)T + d[n-1]] - y[nT + d[n]]] \cdot y[nT - 0.5T + d[n-1]] \quad (12)$$

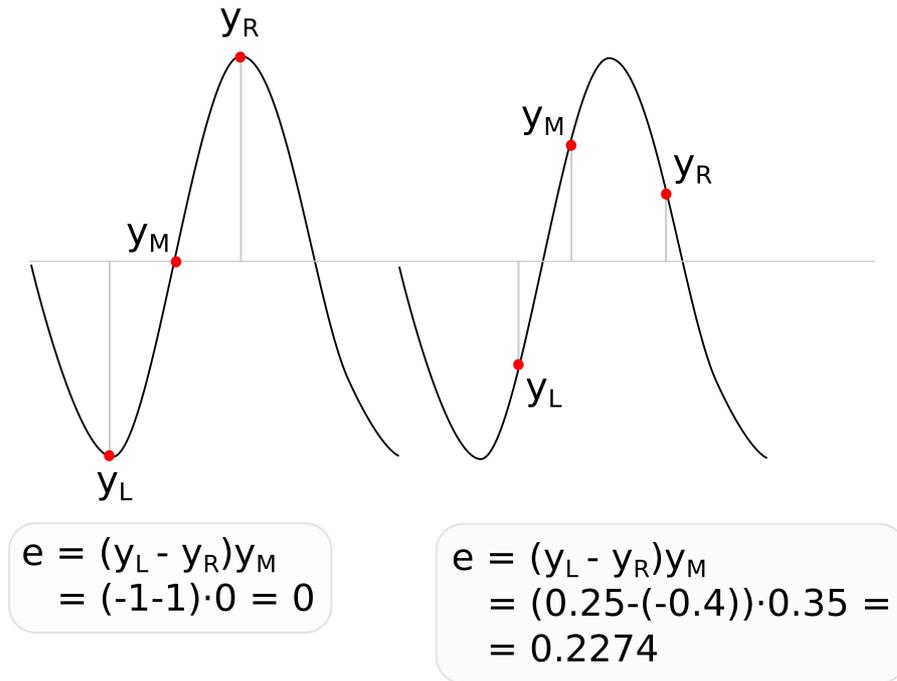


Figure 18: The Gardner TED

Figure 18, shows how the Gardner algorithm works by example. The simplified formula for real signals (13) is similar to the one for early-late TED, but the sign of the error is different.

$$e = -y_M \cdot (y_R - y_L) = y_M \cdot (y_L - y_R) \quad (13)$$

- The **squaring timing recovery** [16] [3, pp. 294] is similar in principle to the squaring loop for carrier recovery. The exponentiated signal will contain a spectral line corresponding to the data rate. If we apply a narrow-band filter to pass that single spectral line, the peaks of the result in time domain can be used to sample the original signal (see Figure 19). You can see the result of raising a BPSK31 signal to square on Figure 20.

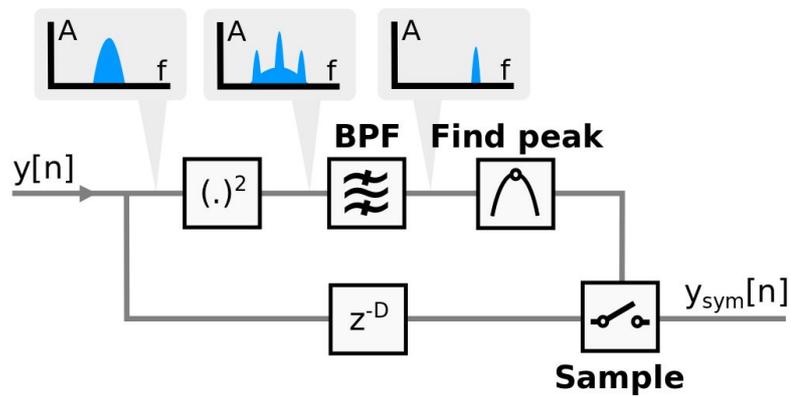


Figure 19: Block diagram of squaring timing recovery

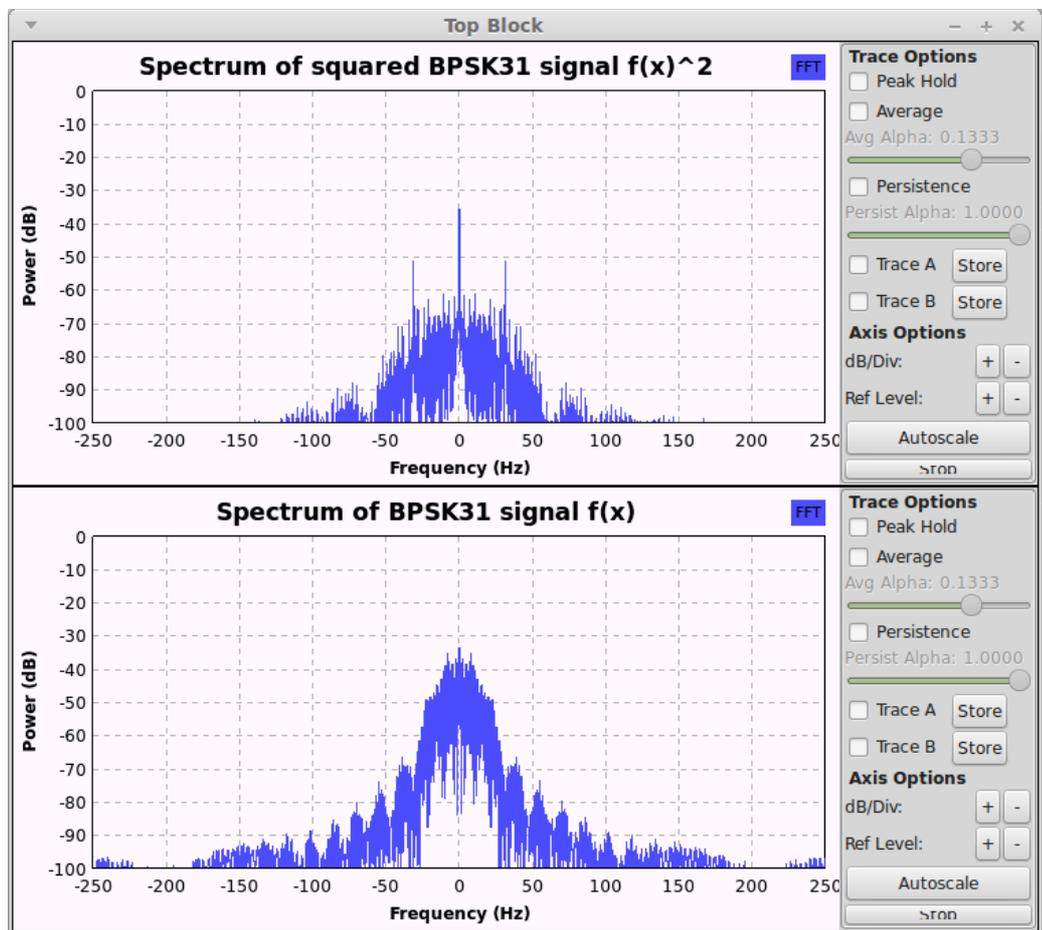


Figure 20: BPSK31 signal squared, with peaks at DC, +31.25 Hz and -31.25 Hz

- **Decision-directed timing recovery methods** take the symbol decisions into consideration. In general, they produce better results if executed after matched filtering and carrier recovery, and if the SNR is sufficiently high enough.

- The TED formula of the **Mueller and Müller algorithm** [17] [18] is (14) in case of a complex input signal.

$$e[n] = \Re \left\{ f_{\text{decide}}^* \left(y((n-1)T + d[n-1]) \right) y(nT + d[n]) - f_{\text{decide}}^* \left(y(nT + d[n]) \right) y((n-1)T + d[n-1]) + d[n-1] \right\} \quad (14)$$

(15) is a simplified formula for the M&M TED:

$$e = \Re \left\{ f_{\text{decide}}^* (y_L) y_C - f_{\text{decide}}^* (y_C) y_L \right\} \quad (15)$$

- The formula of the **zero-crossing TED** [18](16) is similar to the Mueller and Müller algorithm above.

$$e[n] = y(nT - 0.5T + d[n]) \left[f_{\text{decide}} \left(y((n-1)T + d[n]) \right) - f_{\text{decide}} \left(y(nT + d[n]) \right) \right] \quad (16)$$

Its simplified version is (17), see Figure 21 for an example how this formula works.

$$e = y_M \left[f_{\text{decide}} (y_L) - f_{\text{decide}} (y_R) \right] \quad (17)$$

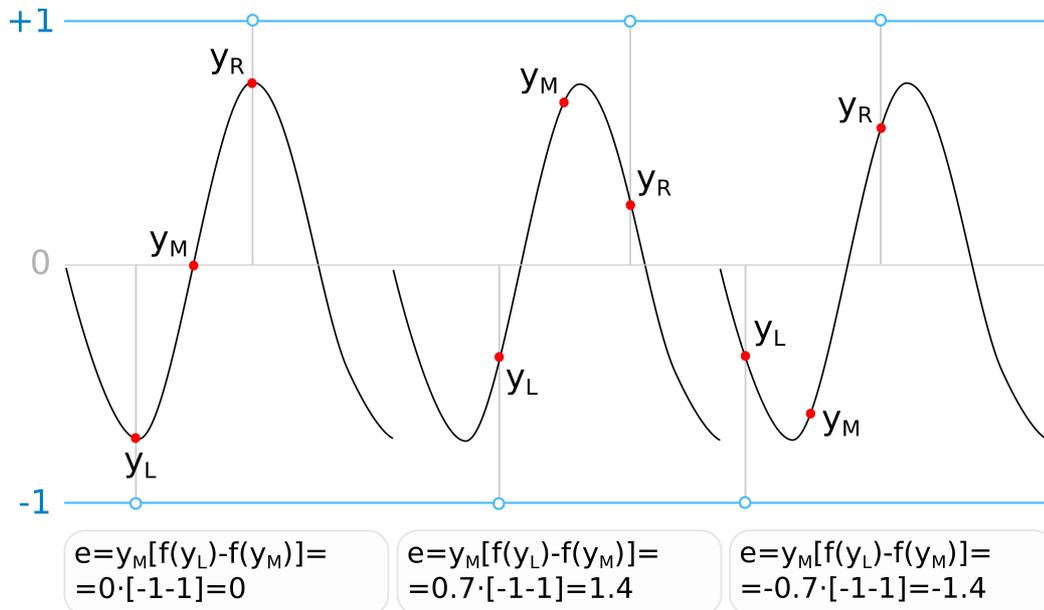


Figure 21: The zero-crossing TED

Some methods only have a TED formula for real signals, but they can still be applied to complex signals, by calculating the error for the real and the imaginary part separately, and summing them up, as in (18) [13].

$$e[n]=e_I[n]+e_Q[n] \quad (18)$$

Both carrier and symbol timing recovery algorithms introduce a trade-off between convergence speed and the variance of estimation error [19].

4.4 Implementation of non-data aided timing recovery

For CSDR, the early-late and the Gardner algorithms have been implemented. These are both non-data aided timing recovery methods, so I considered these more sufficient for general use because they also work at low SNR.

The input of *csdr timing_recovery_cc* is the baseband signal, which is advised to be oversampled at least 4 times compared to the data rate. The output is the signal sampled at the estimated maximum eye openings, at the data rate. These complex samples can be mapped to the nearest constellation points to find the symbols corresponding to them.

What differentiates the *csdr timing_recovery_cc* command from other open source tools available is the possibility to generate images about the internal working of the algorithm using GNU Octave immediately. Using the *--octave* switch for *csdr timing_recovery_cc*, and piping *stderr* into an interactive GNU Octave session, one can get immediate visual feedback about the performance of the timing recovery algorithm on the given signal, and the graphs can also be saved to PNG files using the *--octave-save* switch.

Both the early-late and the Gardner methods take three samples into consideration while determining the timing error. Figure 22 shows an example output of *csdr timing_recovery_cc* using the plotting capabilities of GNU Octave. Figure 23 shows a compilation of diagrams generated during the phase acquisition using the Gardner method on a BPSK31 signal, and Figure 24 shows the same for the early-late gate algorithm, but here the signal gets past the preamble phase, and the algorithm still keeps locking.

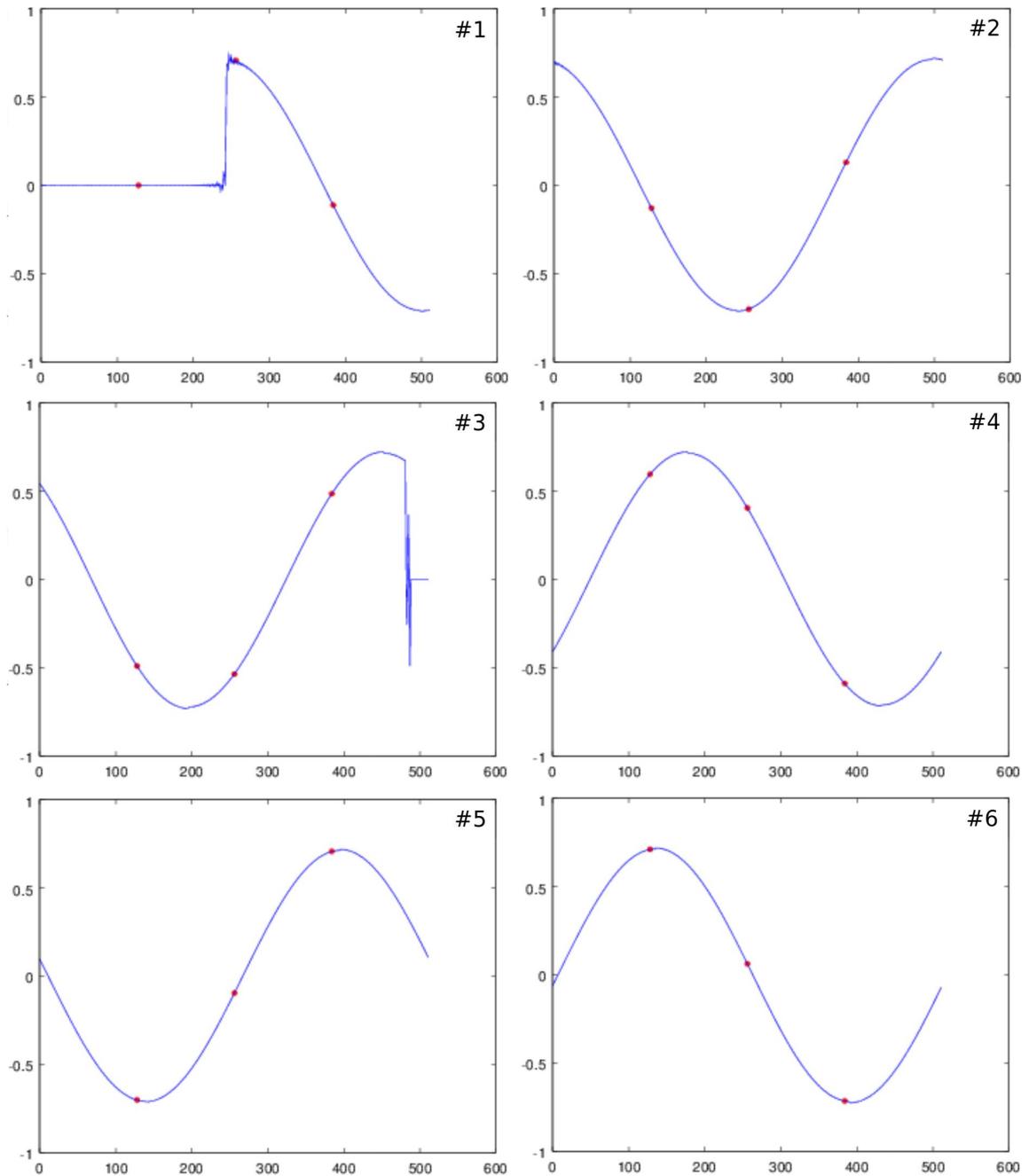


Figure 23: The Gardner timing recovery algorithm locks onto the BPSK31 baseband signal. (The number in the top right corner of each diagram shows the sequence.)

Figure 23 and 24 also emphasize the difference between the Gardner and the early-late algorithm: the Gardner algorithm tries to find the middle of the signal transition, and the output sample is taken from the red dot on the left, while the early-late algorithm tries to "climb" to the middle of the symbol, taking the output sample from the middle red dot.

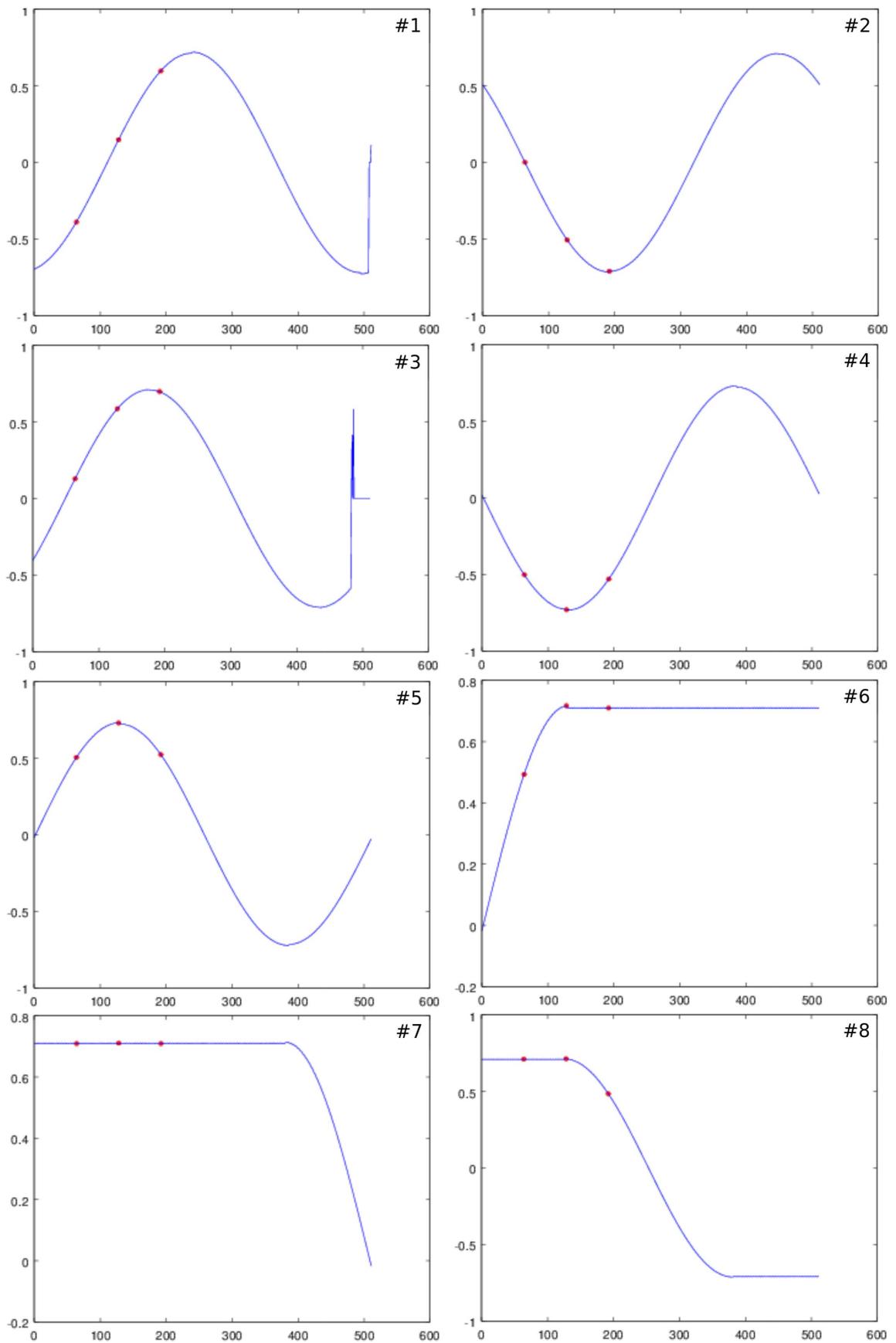


Figure 24: The early-late gate timing recovery algorithm locks onto the BPSK31 baseband signal during preamble, and stays locked afterwards.

The syntax of the CSDR function corresponding to the timing recovery is as below:

```
cldr timing_recovery_cc <algorithm> <decimation> [mu [max_error \  
  [--add_q [--output_error | --output_indexes | \  
  --octave <show_every_nth> | \  
  --octave_save <show_every_nth> <directory> ]]]]
```

The *algorithm* can currently be set to "GARDNER" or "EARLYLATE".

The *decimation* parameter refers to the number of samples per symbol at the input signal.

The *mu* parameter is the update step of the loop, see (10).

The *max_error* allows us to clip the error signal, to keep it in a given range.

If the *--add_q* parameter is false, the TED is only evaluated on the real part of the signal. If *--add_q* is true, it is evaluated on both the I and the Q branch, and the results are added.

With *--output_error*, the output of the function is changed: it rather outputs the calculated error of the TED for each output sample, which can be used for generating the S-curve, see section 4.5.

Similarly, with *--output_indexes*, the function outputs the indexes of the maximum amplitude points in the input stream, which can be used while generating the estimator variance diagram, see section 4.5.

With the *--octave* switch, GNU Octave commands are written to the standard error, which plot the input samples related to the current symbol, and also the relevant points on them. The required parameter *show_every_nth* means that one out of *show_every_nth* will be plotted with Octave. If *show_every_nth* is 0, then a plot will be generated for every single symbol. The *--octave-save* parameter has the same function, but it saves the graphs into PNG files into the *directory* also given as parameter.

The *zsh* shell can be used to redirect the standard error to GNU Octave directly from *cldr timing_recovery_cc*, as in the example command below. This allows one to plot the internals of the algorithm while running a signal through it.

```
csdr timing_recovery_cc GARDNER 48 0.5 2 --add_q \  
--octave 0 2> >(octave -i)
```

4.5 Symbol timing recovery performance measurements

While developing the analog demodulator processing chain, it was sufficient to observe the spurious-free dynamic range (SFDR) for most algorithms to confirm that they work properly. However, evaluating the performance of timing recovery algorithms is a more difficult task.

One of the metrics of the timing recovery algorithm is the S-curve. The S-curve shows how the TED reacts if fed with the same input signal with different phase offsets. Basically, it is a plot to show how the error depends on the phase offset. To draw this curve, the TED need to be placed in an open loop, so that the error does not have any effect on the sampling phase in the next iteration.

A GNU Octave script, *grc_tests/bpsk31_scurve.m* has been written on the purpose of generating the S-curve, the results of which are found on Figure 25 and Figure 26. The template for the command used to generate the input with changing phase offset can be found below. It is used internally by the Octave script.

```
dd bs=8 skip=<phase_offset> \  
if=bpsk31_baseband_sample_complex_8000_sps_010101.raw | \  
csdr timing_recovery_cc <timing_recovery_algorithm> 256 \  
--add_q -output_error
```

The S-curve is correct if the error is only zero when the symbol is sampled at the maximum amplitude. If the error is zero at any other point, it is called a false lock point, because the loop can get stuck there while sampling at the wrong phase.

A sufficient TED is free from false lock points. In both Figure 25 and 26, the phase offsets of 0, 128 and 256 correspond to maximum eye openings, and no false lock points can be identified. A significant difference between the two TED formulas is the sign of the error, which needs to be taken into consideration in other parts of the timing recovery algorithm.

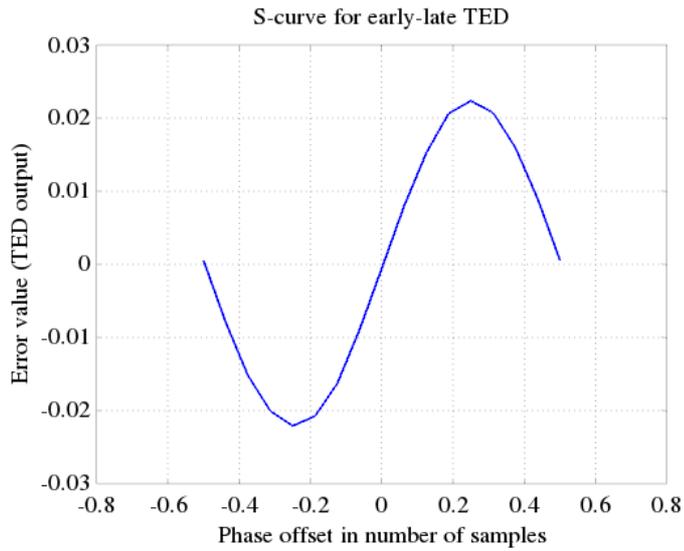


Figure 25: *S-curve for early-late TED on BPSK31 baseband signal*

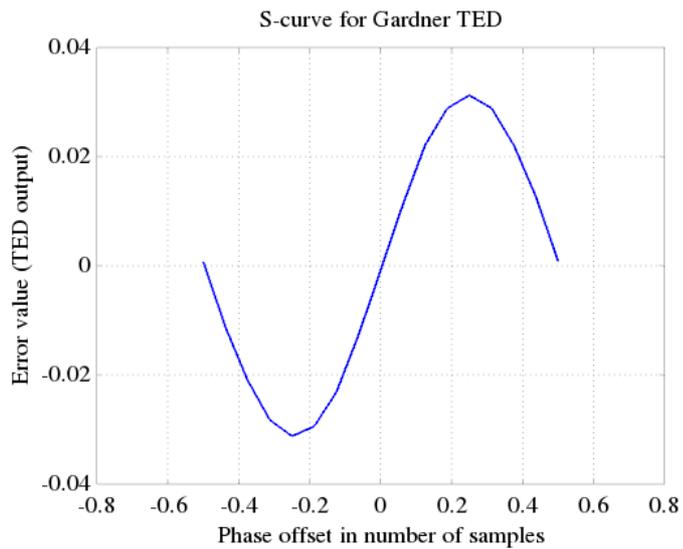


Figure 26: *S-curve for Gardner TED on BPSK31 baseband signal*

We can further analyze the performance of the estimator by generating the variance diagram, by measuring the variance of the phase error while feeding the demodulator with the sum of modulated random data and Gaussian white noise. The diagram shows the variance at different SNR per bit (E_b/N_0) values.

The `grc_tests/bpsk31_tedvar.m` script has been written to carry out this measurement. It

measures the estimator variance for 1,000,000 symbols. As the necessary computations take a long time, this Octave script has been optimized to use multiple CPU cores, and it caches the signal and the noise in advance. The output of the script can be inspected on Figure 27 and 28.

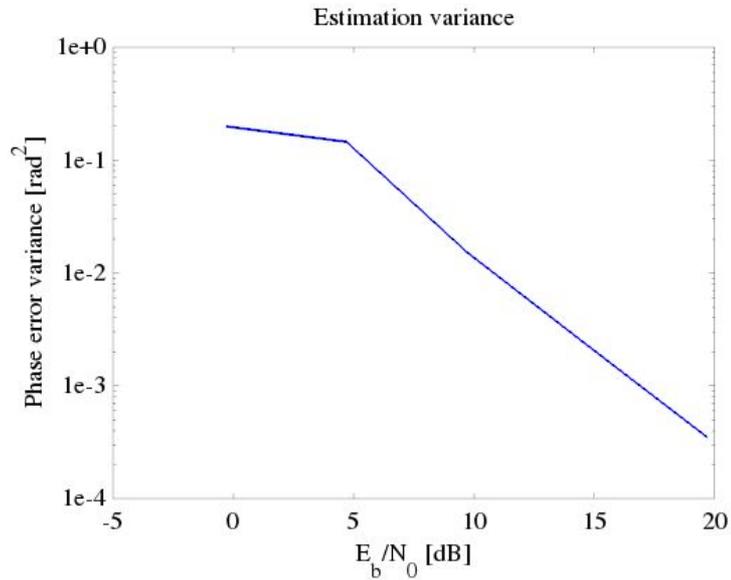


Figure 27: Estimator variance of the Gardner algorithm in `csdr timing_recovery_cc`

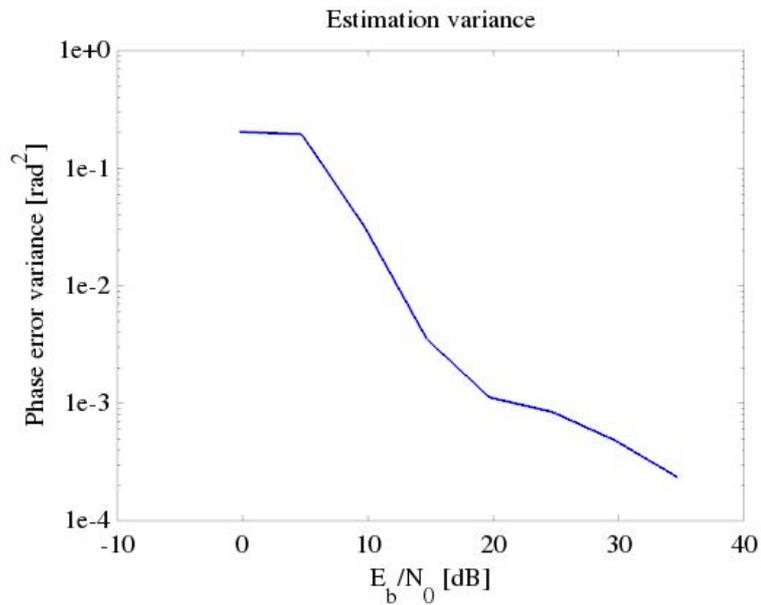


Figure 28: Estimator variance of the early-late gate algorithm in `csdr timing_recovery_cc`

4.6 Carrier recovery

If the conditions are ideal, the IF signal is the baseband BPSK31 signal centered at DC. However, in practice there is an unknown phase or frequency offset between the transmitted and the received signal, as the oscillators of the transmitter and the receiver are not synchronized, and are also subject to drifting due to temperature changes (this offset might even change in time).

We can see the result of a frequency offset at the receiver on Figure 30 and 31, while the original signal (without offset) is shown on Figure 29.

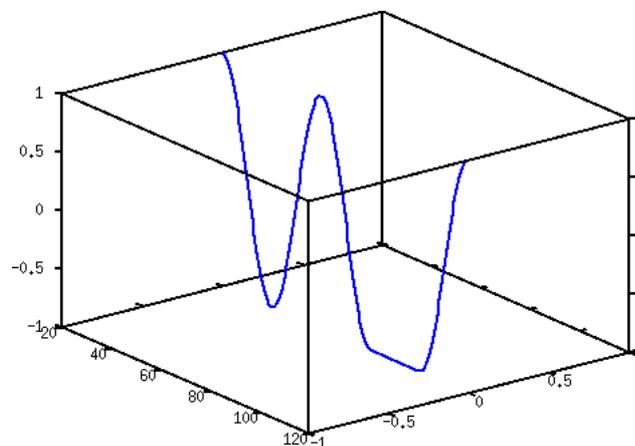


Figure 29: Transmitted BPSK31 baseband signal

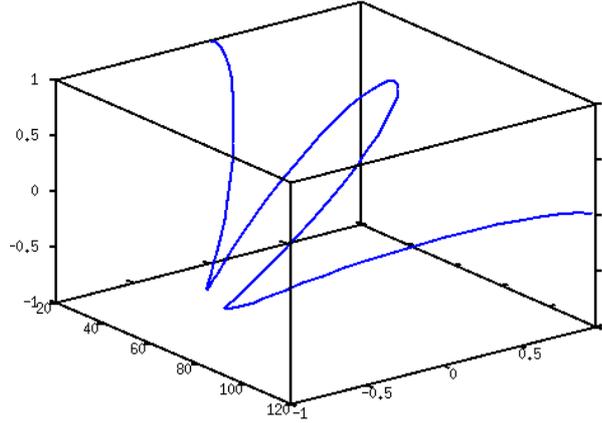


Figure 30: Received BPSK31 baseband signal with a frequency offset of $\Delta\omega = 0.005\pi \frac{\text{rad}}{\text{sec}}$

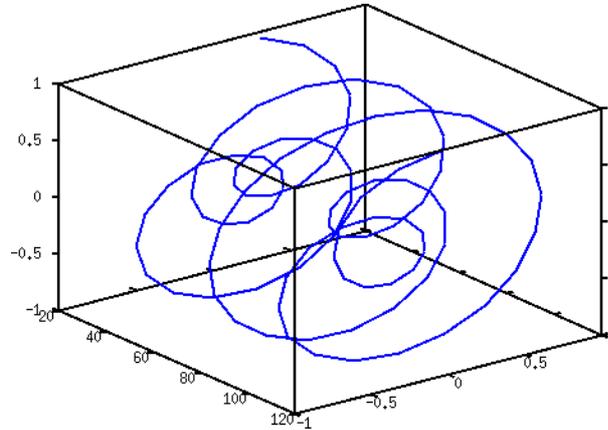


Figure 31: Received BPSK31 baseband signal with a frequency offset of $\Delta\omega = 0.1\pi \frac{\text{rad}}{\text{sec}}$

The received baseband signal can be written as (19), where the φ_{offset} contributes a constant phase difference, and f_{offset} contributes a constant phase rotation to the baseband signal.

$$y_{\text{received}}(t) = e^{j(2\pi f_{\text{offset}} t + \varphi_{\text{offset}})} \sum_{n=-\infty}^{\infty} s[n] g(t - nT_b) \quad (19)$$

This rotation can also be inspected on the constellation diagram, as seen on Figure 32. (In case of a frequency offset, it is also common to see a full circle on the constellation diagram.)

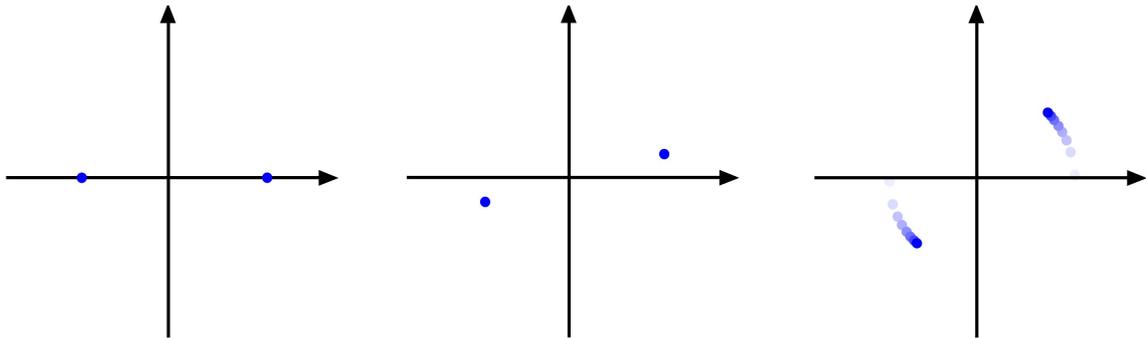


Figure 32: Constellation diagram of transmitted BPSK signal (on the left), received BPSK signal with a phase offset (in the middle), received BPSK signal with a frequency offset (on the right).

The carrier recovery function should estimate the unknown $e^{j(2\pi f_{\text{offset}} t + \phi_{\text{offset}})}$ in (19), and multiply the input signal with the conjugate of the estimate in order to correct it.

We classify the carrier recovery methods as follows.

- **Non-data aided carrier recovery methods** do not assume any previous knowledge about the data symbols.

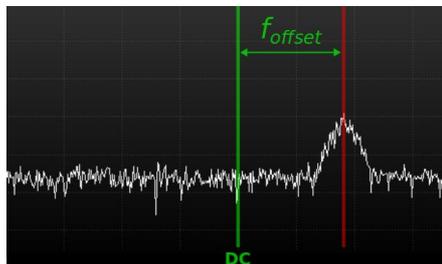


Figure 33:
Coarse carrier acquisition

- **Coarse carrier acquisition using DFT/FFT** is based on the Fourier transform of the input signal, and looks for the bin corresponding to the peak of the signal to find the frequency offset, see Figure 33.
- The **Costas loop** is a phase-locked loop (PLL) modified so that it can lock on PSK signals (see Figure 34) and it is possible to use it for fine carrier acquisition. It also has decision directed versions, as shown later.

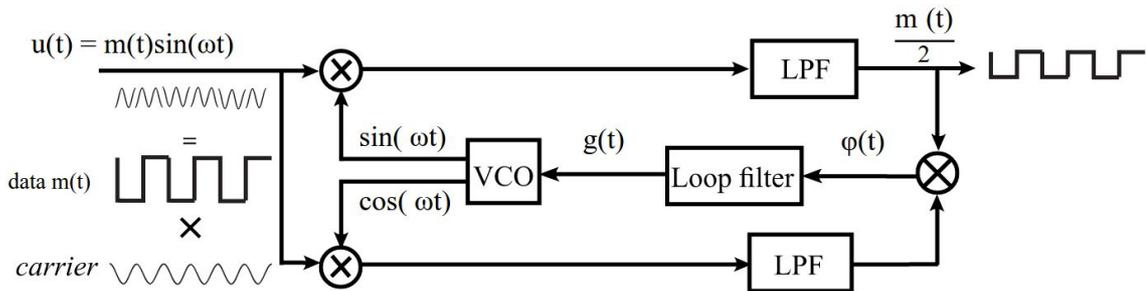


Figure 34: Block diagram of Costas loop for locking on BPSK signals [23]

- It can be shown that raising PSK and QAM input signals to a given power, the result will have a sinusoidal component with a constant phase. **The squaring loop** uses a PLL to lock on this component.

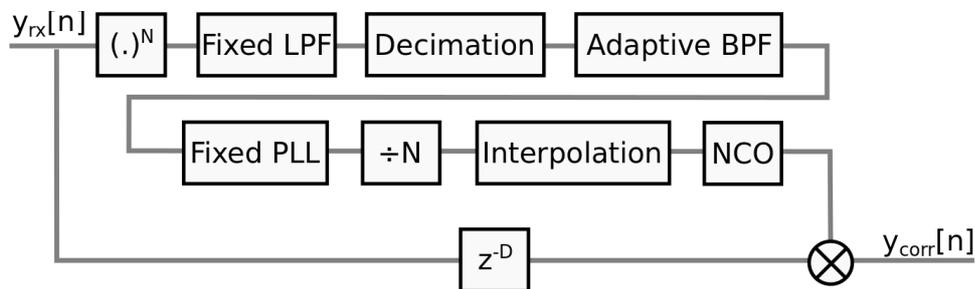


Figure 35: Block diagram of squaring loop [3, pp. 249]

- **Decision directed carrier recovery** calculates the phase difference between the received sample and the corresponding output of the symbol decoder, thus it takes the value of the decoded symbol into consideration and attempts to minimize the phase difference using an adequate control loop, see Figure 36. See also Figure 37 for a Costas loop that can lock on QPSK signals.

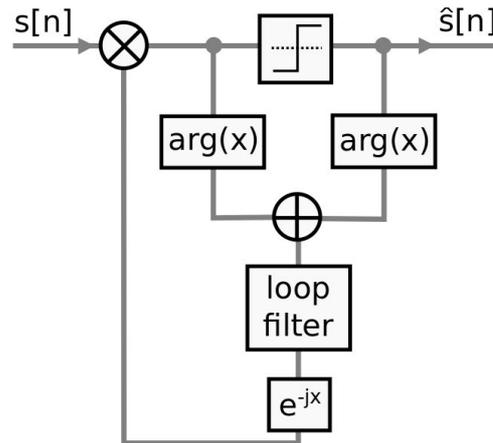


Figure 36: Block diagram of decision directed carrier recovery [3, pp. 264]

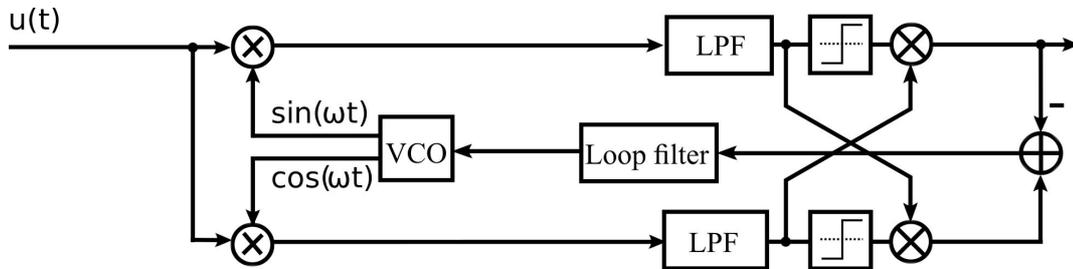


Figure 37: Block diagram of Costas loop for locking on QAM signals [25], [3, pp. 260]

- **Data-aided carrier recovery methods** uses previous knowledge about the transmitted symbols:
 - if the transmission starts with a sequence of training symbols, e. g. a preamble that precedes each packet,
 - if a pilot signal is available to lock onto.

For best performance, carrier recovery can be done in multiple stages: first, a coarse carrier acquisition can be made using FFT, it can be refined using one of the non-data-aided methods, and then we can proceed to the decision-directed method.

The locking range of latter stages is narrower, but their error variance in tracking mode is better. Note that the timing recovery and equalization might have adaptive parts that also need to be tuned before we can proceed from one stage to another.

The carrier recovery method I first implemented was a Costas loop, which worked well if the frequency difference was very small (see section 5.4). However, the user typically fails to correctly select the center of the BPSK31 signal on the web UI, and clicking at the wrong pixel on the waterfall diagram easily results in a frequency error in the order of 10 Hz.

4.7 Synchronization in a differential PSK receiver

For differential BPSK signals, there is a method that allows joint carrier synchronization and differential decoding, as described in [20, pp. 26].

Even if carrier synchronization is applied at the receiver, the constellation of the received signal is subject to a possible 180° rotation. In this case, the samples get mapped to wrong symbols at the slicer.

As an example, if we send the symbol sequence "01001" at the transmitter, we might get "01001" or "10110" at the receiver, based on how the carrier recovery locked.

A way to solve this problem is encoding the symbols with the phase changes. This technique is called DBPSK (differential BPSK). For BPSK31:

- if the phase remains unchanged for the duration of a symbol, it means a "1",
- if the phase changes 180° between symbols, it means a "0".

From the output of the slicer, we have the necessary information about phase changes. See Table 4 for the function carried out by the differential decoder.

Last input	Current input	Output
0	0	1
0	1	0
1	0	0
1	1	1

Table 4: Logic function carried out by the differential decoder

CSDR now also contains a differential encoder that applies this operation in reverse. "0" and "1" symbols map to 0x00 and 0x01 bytes at the input and output of the these

commands. You can find the syntax of the related CSDR commands below:

```
cldr differential_decoder_u8_u8  
cldr differential_encoder_u8_u8
```

To show how the binary data can be read and decoded by visually inspecting the waveform, see Figure 38.

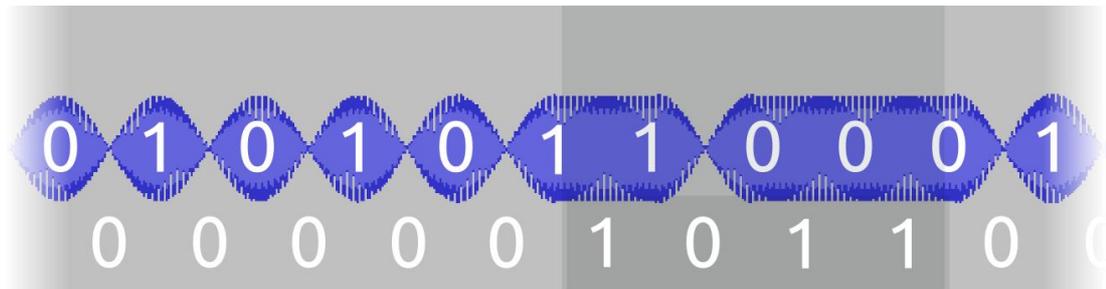


Figure 38: Symbol decisions (above) and differential decoded data (below)

Note that the decoded data does not depend on the initial phase offset of the receiver, as seen on Figure 39.

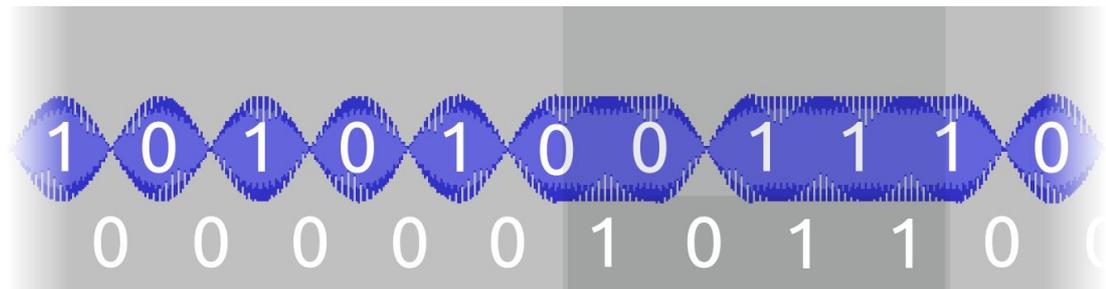


Figure 39: Symbol decisions (above) and differential decoded data (below) with a different receiver starting phase offset

However, carrier synchronization and differential decoding can be done in a single step, as implemented in `cldr dbpsk_decoder_c_u8` (see Figure 40). The method to be described works on the output of symbol timing recovery (with one sample for each symbol), and makes use of the current and the last input sample, calculating the difference of their phase. If the difference is above 90° , we assume that the current symbol has changed, and write an "0" to the output, while if the difference is below 90° , we assume that the symbol has not changed, and write a "1".

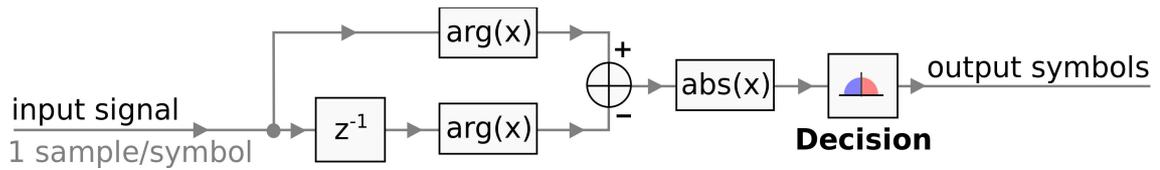


Figure 40: The block diagram of `csdr dbpsk_decoder_c_u8`

4.8 Varicode decoder

BPSK31 uses Varicode for encoding the characters, which is a variable-length code that maps shorter bit sequences to more frequently used letters. Characters are delimited by multiple "0" symbols.

The bit sequences of the characters were designed in a way that multiple "0" bits cannot come after each other inside them. If we see multiple "0" bits after each other, the next "1" bit will be part of a new character.

With the full conversion table (in Table 5) one can decode the Varicode characters into ASCII. Figure 41 shows the waveform with the Varicode characters highlighted. Using the code table, the character sequence "abc" can be decoded.

A	1111101	a	1011	SPACE	1	NUL	1010101011
B	11101011	b	1011111	!	111111111	SOH	1011011011
C	10101101	c	101111	"	101011111	STX	1011101101
D	10110101	d	101101	#	111110101	ETX	1101110111
E	1110111	e	11	\$	111011011	EOT	1011101011
F	11011011	f	111101	%	1011010101	ENQ	1101011111
G	11111101	g	1011011	&	1010111011	ACK	1011101111
H	101010101	h	101011	(11111011	BEL	1011111101
I	1111111	i	1101)	11110111	BS	1011111111
J	111111101	j	111101011	*	101101111	HT	11101111
K	101111101	k	10111111	+	111011111	LF	11101
L	11010111	l	11011	,	1110101	VT	1101101111
M	10111011	m	111011	-	110101	FF	1011011101
N	11011101	n	1111	.	1010111	CR	11111
O	10101011	o	111	/	110101111	SO	1101110101
P	11010101	p	1111111	0	10110111	SI	1110101011
Q	111011101	q	110111111	1	10111101	DLE	1011110111
R	10101111	r	10101	2	11101101	DC1	1011110101
S	1101111	s	10111	3	11111111	DC2	1110101101
T	1101101	t	101	4	101110111	DC3	1110101111
U	101010111	u	110111	5	101011011	DC4	1101011011
V	110110101	v	1111011	6	101101011	NAK	1101101011
W	101011101	w	1101011	7	110101101	SYN	1101101101
X	101110101	x	11011111	8	110101011	ETB	1101010111
Y	101111011	y	1011101	9	110110111	CAN	1101111011
Z	101111011	z	111010101	[1010101101	EM	1101111101
:	11110101	<	111101101	\	111110111	SUB	1110110111
/	1011011111	=	1010101]	111101111	ESC	1101010101
{	1010110111	>	111010111	^	111111011	FS	1101011101
 	110111011	?	1010101111	_	1010111111	GS	1110111011
}	1010110101	@	1010111101	.	101101101	RS	1011111011
;	110111101	~	1011010111	DEL	1110110101	US	1101111111

Table 5: Varicode to ASCII mapping table, including control characters [11]

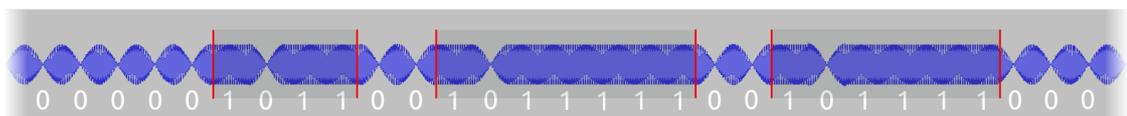


Figure 41: BPSK31 waveform and corresponding binary data (after differential decoding)

I first observed the protocol by inspecting the signal generated by the Fldigi digital modem software. During idle periods, when the software has ran out of characters to send and is waiting for the user to start typing again, the Fldigi software is continuously sending zeros, which means a sequence of "101010..." after differential encoding. This behavior allows the symbol timing recovery algorithm to keep synchronized.

The corresponding CSDR functions are listed below. CSDR also contains a Varicode encoder that can be used to generate BPSK31 modulated signals.

```
cldr psk31_varicode_decoder_u8_u8  
cldr psk31_varicode_encoder_u8_u8
```

The input of the Varicode decoder is 0x00 and 0x01 bytes, the symbols after differential decoding. The output of the decoder is ASCII data. (The same applies to the encoder, but in reverse order.)

Internally the Varicode decoder uses a FIFO to store the last 32 bits received, and if it detects a valid Varicode character at the beginning of the FIFO, it writes the corresponding ASCII value to the standard output.

4.9 Resource usage

BPSK31 is very low data rate, and the demodulator works from an IF signal of around 10-12 ksp/s, which results in low CPU usage. While the main waterfall diagram currently uses overlapped FFT by default, the secondary waterfall is not, and the FFT size for it is also smaller, 1024 compared to 4096 for the main waterfall, thus the FFT operations for the secondary waterfall diagram use much less CPU than the main waterfall diagram. As a consequence, there is no point in doing optimizations for low data rate digital modes. You can compare CPU usage on a development version of OpenWebRX with BPSK31 demodulator Figure 42 and 43.

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
27085	pcfl	20	0	5276	1060	736	S	22,5	0,0	0:01.06	csdr fir_decimate_cc 217 0.0006911875 HAMMING
26972	pcfl	20	0	5008	2372	1944	S	13,2	0,0	0:02.93	csdr fft_cc 4096 2867
27084	pcfl	20	0	5276	1060	696	S	6,9	0,0	0:00.32	csdr shift_addition_cc --fifo /tmp/openwebrx_
26973	pcfl	20	0	4768	796	628	S	6,6	0,0	0:01.51	csdr logaveragepower_cf -70 4096 93
26971	pcfl	20	0	4744	532	380	S	5,6	0,0	0:01.26	csdr convert_u8_f
27083	pcfl	20	0	4744	532	380	S	5,3	0,0	0:00.24	csdr convert_u8_f
26970	pcfl	20	0	2640	704	596	S	2,0	0,0	0:00.45	nc -v 127.0.0.1 4951
27082	pcfl	20	0	2640	708	596	S	1,7	0,0	0:00.08	nc -v 127.0.0.1 4951
26959	pcfl	20	0	103540	50672	964	S	1,3	0,6	0:00.40	nmux --bufsize 602112 --bufcnt 84 --port 4951
26949	pcfl	20	0	91556	7668	3240	S	1,0	0,1	0:00.25	/usr/bin/python2 ./openwebrx.py
26958	pcfl	20	0	15512	5080	956	S	1,0	0,1	0:00.28	rtl_sdr -s 2400000 -f 144250000 -p 0 -g 5 -
26974	pcfl	20	0	4744	796	616	S	0,3	0,0	0:00.01	csdr fft_exchange_sides_ff 4096
27086	pcfl	20	0	5040	2860	2332	S	0,3	0,0	0:00.13	csdr bandpass_fir_fft_cc --fifo /tmp/openwebr:
26957	pcfl	20	0	2268	560	492	S	0,0	0,0	0:00.00	/bin/sh -c rtl_sdr -s 2400000 -f 144250000 -p
26969	pcfl	20	0	2268	560	492	S	0,0	0,0	0:00.00	/bin/sh -c nc -v 127.0.0.1 4951 csdr conver
26975	pcfl	20	0	4752	796	616	S	0,0	0,0	0:00.02	csdr compress_fft_adpcm_f_u8 4096
27081	pcfl	20	0	2272	560	492	S	0,0	0,0	0:00.00	/bin/sh -c nc -v 127.0.0.1 4951 csdr conver
27087	pcfl	20	0	4744	796	600	S	0,0	0,0	0:00.00	csdr squelch_and_smeter_cc --fifo /tmp/openwel
27088	pcfl	20	0	4744	536	380	S	0,0	0,0	0:00.00	csdr realpart_cf
27089	pcfl	20	0	4744	796	620	S	0,0	0,0	0:00.00	csdr old_fractional_decimator_ff 1.0030839002:
27090	pcfl	20	0	4744	536	380	S	0,0	0,0	0:00.00	csdr agc_ff
27091	pcfl	20	0	4744	536	380	S	0,0	0,0	0:00.00	csdr limit_ff
27092	pcfl	20	0	4744	532	380	S	0,0	0,0	0:00.00	csdr convert_f_s16
27093	pcfl	20	0	4744	536	380	S	0,0	0,0	0:00.00	csdr encode_ima_adpcm_i16_u8

Figure 42: OpenWebRX and its child processes in the output of the top command without running the BPSK31 demodulator

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
26374	pcfl	20	0	5276	1056	736	S	17,2	0,0	0:44.97	csdr fir_decimate_cc 217 0.0006911875 HAMMING
26076	pcfl	20	0	5008	2372	1944	S	9,9	0,0	0:36.29	csdr fft_cc 4096 2867
26077	pcfl	20	0	4768	796	628	S	5,3	0,0	0:18.73	csdr logaveragepower_cf -70 4096 93
26373	pcfl	20	0	5276	1056	696	S	5,0	0,0	0:13.38	csdr shift_addition_cc --fifo /tmp/openwebrx_
26075	pcfl	20	0	4744	532	380	S	4,3	0,0	0:15.79	csdr convert_u8_f
26372	pcfl	20	0	4744	536	380	S	4,0	0,0	0:10.79	csdr convert_u8_f
26371	pcfl	20	0	2640	704	596	S	1,7	0,0	0:03.83	nc -v 127.0.0.1 4951
26074	pcfl	20	0	2640	708	596	S	1,3	0,0	0:05.60	nc -v 127.0.0.1 4951
26063	pcfl	20	0	103540	50676	964	S	1,0	0,6	0:04.28	nmux --bufsize 602112 --bufcnt 84 --port 4951
26053	pcfl	20	0	91560	7684	3240	S	0,7	0,1	0:01.76	/usr/bin/python2 ./openwebrx.py
26062	pcfl	20	0	15512	5084	956	S	0,7	0,1	0:02.91	rtl_sdr -s 2400000 -f 144250000 -p 0 -g 5 -
26375	pcfl	20	0	4900	2676	2176	S	0,3	0,0	0:00.80	csdr bandpass_fir_fft_cc --fifo /tmp/openwebr:
26377	pcfl	20	0	4872	796	620	S	0,3	0,0	0:00.21	csdr tee /tmp/openwebrx_pipe_3067152044_iqtee
26379	pcfl	20	0	4744	536	380	S	0,3	0,0	0:00.12	csdr realpart_cf
26380	pcfl	20	0	4744	796	624	S	0,3	0,0	0:00.30	csdr old_fractional_decimator_ff 1.0030839002:
26390	pcfl	20	0	4884	2376	1976	S	0,3	0,0	0:00.13	csdr fft_fc 1024 1105
26391	pcfl	20	0	4744	792	624	S	0,3	0,0	0:00.16	csdr logpower_cf -70
26061	pcfl	20	0	2268	556	492	S	0,0	0,0	0:00.00	/bin/sh -c rtl_sdr -s 2400000 -f 144250000 -p
26073	pcfl	20	0	2268	556	492	S	0,0	0,0	0:00.00	/bin/sh -c nc -v 127.0.0.1 4951 csdr conver
26078	pcfl	20	0	4744	796	616	S	0,0	0,0	0:00.16	csdr fft_exchange_sides_ff 4096
26079	pcfl	20	0	4752	796	616	S	0,0	0,0	0:00.43	csdr compress_fft_adpcm_f_u8 4096
26370	pcfl	20	0	2272	564	492	S	0,0	0,0	0:00.00	/bin/sh -c nc -v 127.0.0.1 4951 csdr conver
26376	pcfl	20	0	4744	796	600	S	0,0	0,0	0:00.14	csdr squelch_and_smeter_cc --fifo /tmp/openwel
26378	pcfl	20	0	4872	796	620	S	0,0	0,0	0:00.22	csdr tee /tmp/openwebrx_pipe_3067152044_iqtee:
26381	pcfl	20	0	4744	532	380	S	0,0	0,0	0:00.09	csdr agc_ff
26382	pcfl	20	0	4744	536	380	S	0,0	0,0	0:00.04	csdr limit_ff
26383	pcfl	20	0	4744	532	380	S	0,0	0,0	0:00.04	csdr convert_f_s16
26384	pcfl	20	0	4744	532	380	S	0,0	0,0	0:00.11	csdr encode_ima_adpcm_i16_u8
26386	pcfl	20	0	2268	560	492	S	0,0	0,0	0:00.00	/bin/sh -c cat /tmp/openwebrx_pipe_3067152044
26387	pcfl	20	0	2272	560	492	S	0,0	0,0	0:00.00	/bin/sh -c cat /tmp/openwebrx_pipe_3067152044
26388	pcfl	20	0	5732	284	228	S	0,0	0,0	0:00.07	cat /tmp/openwebrx_pipe_3067152044_iqtee_pipe
26389	pcfl	20	0	4744	536	380	S	0,0	0,0	0:00.07	csdr realpart_cf
26392	pcfl	20	0	4740	788	616	S	0,0	0,0	0:00.09	csdr compress_fft_adpcm_f_u8 1024
26393	pcfl	20	0	5732	284	228	S	0,0	0,0	0:00.06	cat /tmp/openwebrx_pipe_3067152044_iqtee2_pip
26394	pcfl	20	0	5276	1060	712	S	0,0	0,0	0:00.08	csdr shift_addition_cc --fifo /tmp/openwebrx_
26395	pcfl	20	0	4744	792	624	S	0,0	0,0	0:00.09	csdr simple_agc_cc 0.001 0.5
26396	pcfl	20	0	6196	588	512	S	0,0	0,0	0:00.07	tee /tmp/iqtmp1

Figure 43: OpenWebRX and its child processes in the output of the top command while running the BPSK31 demodulator

5 More applications of CSDR

5.1 Generating BPSK31

To have a complete implementation of BPSK31 in CSDR, I also needed a BPSK31 transmitter. (This method was also used while generating estimator variance graphs.) The following command outputs a valid BPSK31 signal to the audio card:

```
while true; do
    echo -n "CQ CQ CQ DE HA7ILM HA7ILM HA7ILM PSE K ";
done | \
csdr psk31_varicode_encoder_u8_u8 | \
csdr differential_encoder_u8_u8 | \
csdr psk_modulator_u8_c 2 | \
csdr gain_ff 0.25 | \
csdr psk31_interpolate_sine_cc 256 | \
csdr shift_addition_cc 0.125 | \
csdr realpart_cf | \
csdr convert_f_s16 | \
mplayer -cache 1024 -quiet \
    -rawaudio samplesize=2:channels=1:rate=8000 -demuxer rawaudio -
```

There is another version that produces the same result, using the formula (5):

```
while true; do \
    echo -n "CQ CQ CQ DE HA7ILM HA7ILM HA7ILM PSE K "; \
done | \
csdr psk31_varicode_encoder_u8_u8 | \
csdr differential_encoder_u8_u8 | \
csdr psk_modulator_u8_c 2 | \
csdr gain_ff 256 | \
csdr plain_interpolate_cc 256 | \
csdr pulse_shaping_filter_cc COSINE 256 | \
csdr shift_addition_cc 0.125 | \
csdr realpart_cf | \
csdr convert_f_s16 | \
mplayer -cache 1024 -quiet \
    -rawaudio samplesize=2:channels=1:rate=8000 -demuxer rawaudio -
```

Let's analyze each command. Varicode and differential encoding have been described in sections 4.7 and 4.8. A common function for both commands, the PSK modulator can generate a baseband N-PSK signal, mapping input bytes to samples at the given constellation points:

```
csdr psk_modulator_u8_c <n_psk>
```

For a BPSK signal, $n_psk=2$ and the samples output are:

- complex floating point $1+0j$ for 0x00 byte input,
- complex floating point $-1+0j$ for 0x01 byte input.

To add pulse shaping, we need to interpolate these samples. We can either use the function `psk31_interpolate_sine_cc` the result of which is a valid BPSK31 baseband signal, or carry out the raw interpolation (adding zero samples between the ones corresponding to the symbols) and running the pulse shaping filter $g(t)$ in separate steps, as in (5).

```

csdr psk31_interpolate_sine_cc <interpolation>
csdr plain_interpolate_cc <interpolation>
csdr pulse_shaping_filter_cc \
  (RRC <samples_per_symbol> <num_taps> <beta> | \
  COSINE <samples_per_symbol>)

```

The remaining `csdr` commands shift the signal up to 2 kHz and turn it into a real signal, then play it on the audio device.

5.2 Demodulating BPSK63

Note that using this transmitter and receiver structure, we can almost instantly derive different modulations from a given one. For example, deriving a BPSK63 demodulator from the command for BPSK31 is the matter of changing four numbers.

```

arecord -r48000 -c1 -fS16_LE | \
csdr convert_s16_f | \
csdr dsb_fc | \
csdr shift_addition_cc $(csdr '=-1000./48e3') | \
csdr fir_decimate_cc 32 | \
csdr bandpass_fir_fft_cc $(csdr '=-62.5/1.5e3') \
  $(csdr '=62.5/1.5e3') $(csdr '=62.5/1.5e3') | \
csdr simple_agc_cc 0.001 0.5 | \
csdr timing_recovery_cc GARDNER 24 0.5 2 --add_q | \
CSDR_FIXED_BUFSIZE=1 csdr dbpsk_decoder_c_u8 | \
CSDR_FIXED_BUFSIZE=1 csdr psk31_varicode_decoder_u8_u8

```

The command below has been tested against a signal generated with Fldigi, as seen on Figure 44.

```

arecord -r48000 -c1 -fs16_LE | csdr convert_s16_f | csdr dsb_fc | csdr |
Fájl Szerkesztés Nézet Keresés Terminál Súgó
pcfl@ssd-mint ~$ arecord -r48000 -c1 -fs16_LE | \
csdr convert_s16_f | \
csdr dsb_fc | \
csdr shift_addition_cc $(csdr '-1000./48e3') | \
csdr fir_decimate_cc 32 | \
csdr bandpass_fir_fft_cc $(csdr '=(62.5)/1.5e3') $(csdr '=(62.5)/1.5e3') $(csdr
'=62.5/1.5e3') | \
csdr simple_agc_cc 0.001 0.5 | \
csdr timing_recovery_cc GARDNER 24 0.5 2 --add_q | \
CSDR_FIXED_BUFSIZE=1 csdr dbpsk_decoder_c_u8 | \
CSDR_FIXED_BUFSIZE=1 csdr psk31_varicode_decoder_u8_u8
Rögzítés WAVE 'stdin' : Signed 16 bit Little Endian, Rate 48000 Hz, Mono
fir_decimate_cc: window = HAMMING
fir_decimate_cc: taps_length = 79
csdr shift_addition_cc: reinitialized to -0.0208333
csdr bandpass_fir_fft_cc: window = HAMMING
csdr bandpass_fir_fft_cc: (fft_size = 256) = (taps_length = 95) + (input_size =
162) - 1
(overlap_length = 94) = taps_length - 1
csdr timing_recovery_cc: --add_q mode on
csdr bandpass_fir_fft_cc: filter initialized, low_cut = -0.0416667, high_cut = 0
.0416667
.m HA7ILM HA7ILM PSE K CQ CQ CQ DE HA7ILM HA7

```

Figure 44: BPSK63 demodulator developed quickly with CSDR

5.3 BPSK31 with different pulse shape

We can also easily derive new digital modes that previously did not exist from the command of an existing one. For example, building a custom version of BPSK31 that uses RRC pulse shape can also be made by changing the command line, by adding the RRC filter to both the receiver and the transmitter. The RRC filtered BPSK31 is expected to be less sensitive to ISI.

The command below provides an RRC filtered BPSK31 transmitter ($\beta=0.5$).

```

while true; do echo -n "CQ CQ CQ DE HA7ILM HA7ILM HA7ILM PSE K "; \
done | \
csdr psk31_varicode_encoder_u8_u8 | \
csdr differential_encoder_u8_u8 | \
csdr psk_modulator_u8_c 2 | \
csdr gain_ff 64 | \
csdr plain_interpolate_cc 256 | \
csdr pulse_shaping_filter_cc RRC 256 1001 0.5 | \
csdr shift_addition_cc 0.125 | \
csdr realpart_cf | \
csdr convert_f_s16 | \
mplayer -cache 1024 -quiet -rawaudio \
samplesize=2:channels=1:rate=8000 -demuxer rawaudio -

```

The change in the pulse shape can be instantly seen on the waveform as well, see Figure

45 below.

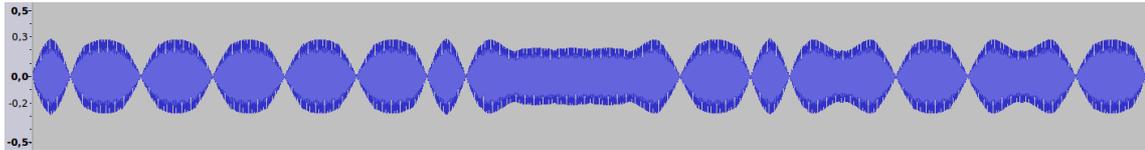


Figure 45: Waveform of generated BPSK31 signal with RRC pulse shape

The demodulator command is as follows:

```
arecord -r48000 -c1 -fs16_LE | \  
csdr convert_s16_f | \  
csdr dsb_fc | \  
csdr shift_addition_cc $(csdr '=-1000./48e3') | \  
csdr fir_decimate_cc 32 | \  
csdr pulse_shaping_filter_cc RRC 48 1001 0.5 | \  
csdr simple_agc_cc 0.001 0.5 | \  
csdr timing_recovery_cc GARDNER 48 0.5 2 --add_q | \  
CSDR_FIXED_BUFSIZE=1 csdr dbpsk_decoder_c_u8 | \  
CSDR_FIXED_BUFSIZE=1 csdr psk31_varicode_decoder_u8_u8
```

Note that Fldigi was also found to correctly decode BPSK31 with RRC pulse shaping, although not optimized for that.

5.4 Costas loop for carrier recovery

The command below is a version of the BPSK31 demodulator that uses a Costas loop.

```
arecord -r48000 -c1 -fs16_LE | \  
csdr convert_s16_f | \  
csdr dsb_fc | \  
csdr shift_addition_cc $(csdr '=-1000./48e3') | \  
csdr fir_decimate_cc 32 | \  
csdr bandpass_fir_fft_cc $(csdr '=- (31.25)/1.5e3') \  
$(csdr '= (31.25)/1.5e3') $(csdr '=31.25/1.5e3') | \  
csdr simple_agc_cc 0.001 0.5 | \  
csdr timing_recovery_cc GARDNER 48 0.5 2 --add_q | \  
CSDR_FIXED_BUFSIZE=1 csdr bpsk_costas_loop_cc 0.10 0.707 | \  
CSDR_FIXED_BUFSIZE=1 csdr realpart_cf | \  
CSDR_FIXED_BUFSIZE=1 csdr binary slicer_f_u8 | \  
CSDR_FIXED_BUFSIZE=1 csdr differential_decoder_u8_u8 | \  
CSDR_FIXED_BUFSIZE=1 csdr psk31_varicode_decoder_u8_u8
```

In this command, the Costas loop is placed after the symbol timing recovery phase. The Gardner algorithm still works well if a small, additional phase or frequency offset is present. This way the Costas loop works on one sample per symbol, and tries to rotate the samples back, as near to the ideal BPSK constellation points as possible.

In practice, the Costas loop was less efficient than the solution described in 4.7. This could be improved upon if a coarse carrier recovery algorithm has been applied first. It also helped if the waterfall display was improved so that it was easier to tune to the PSK31 signal accurately.

However, the Costas loop described here is a general algorithm that could possibly be used for other modulations as well, not only BPSK31. The syntax of the related CSDR function is as below:

```
csdr bpsk_costas_loop_cc <loop_bandwidth> <damping_factor> \
  [--dd | --decision_directed] [--output_error | \
  --output_dphase | --output_nco | \
  --output_combined <error_file> <dphase_file> <nco_file>]
```

Figure 46 shows the block diagram for the CSDR implementation of a Costas loop for BPSK signals. Both of its input and output is complex floating point. It has a switch to turn it into decision directed mode (`--dd` or `--decision_directed`), then it implements the same block diagram as on Figure 36.

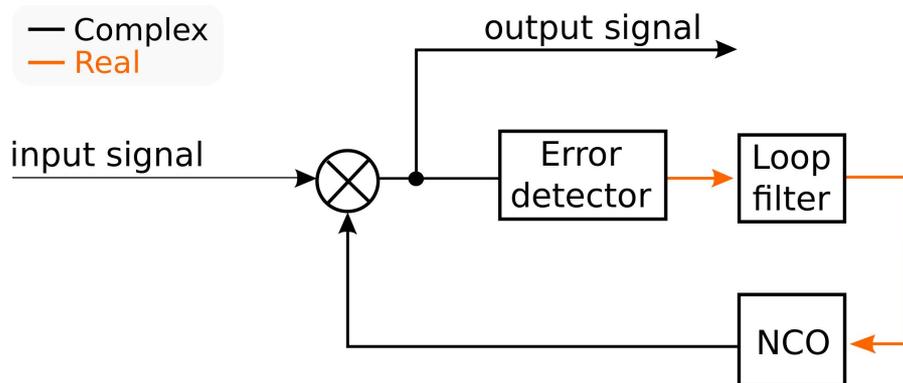


Figure 46: Block diagram for non-data aided mode of `csdr bpsk_costas_loop_cc`

Internally, it uses a second order loop filter, the structure of which is shown on Figure 47, where (20) and (21) are the formulas for calculating the α and β parameters [21].

$$\alpha = \frac{4\zeta\theta_n}{1+2\zeta\theta_n+\theta_n^2} \quad (20)$$

$$\beta = \frac{4\theta_n^2}{1+2\zeta\theta_n+\theta_n^2} \quad (21)$$

The ζ parameter is the damping factor, which is to be chosen to 0.707 for a critically damped loop. θ_n stands for the loop bandwidth. The same parameters are to be given to the `csdr bpsk_costas_loop_cc` command.

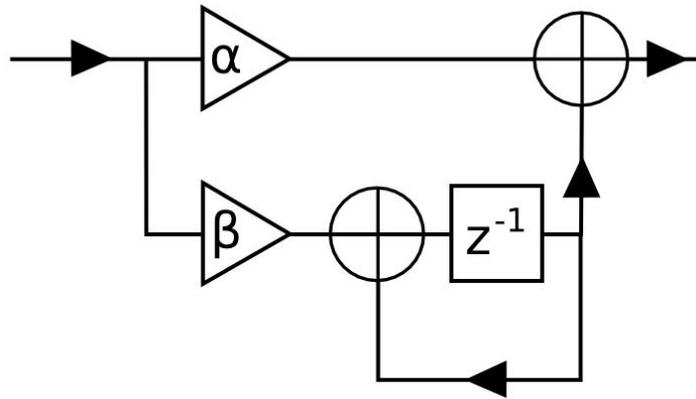


Figure 47: Second order loop filter for PI controller

The error detector calculates $I \cdot Q$ in the default non-data aided mode, and calculates the phase distance from the nearest symbol in decision directed mode. Its S-curve is shown on Figure 48.

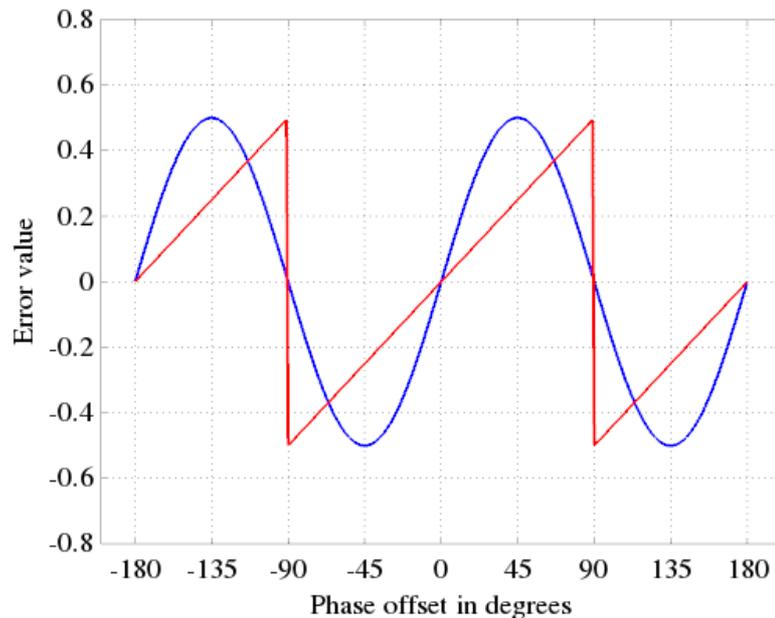


Figure 48: S-curve of BPSK Costas loop in CSDR. The blue line corresponds to the non-data aided mode, and the red line corresponds to the decision directed mode.

The optional parameters `--output_error`, `--output_dphase`, `--output_nco` and `--output_combined` allow us to inspect how the algorithm works internally during synchronization.

Figure X shows a situation where the input signal had a large frequency error and a high amount of noise, but the loop managed to achieve phase lock.

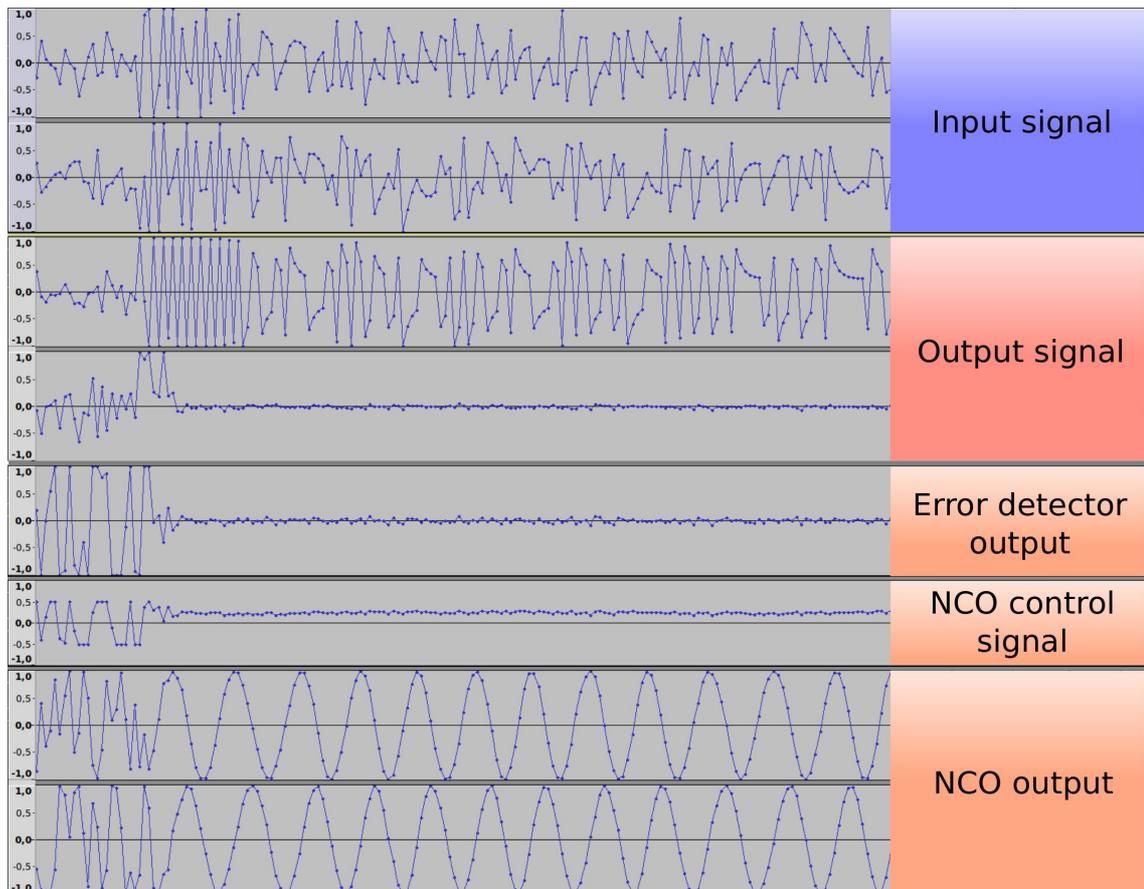


Figure 49: Internal operation of BPSK Costas loop in CSDR

If the frequency offset was too high and the loop did not manage to lock, see the error signal in such a case on Figure 50.

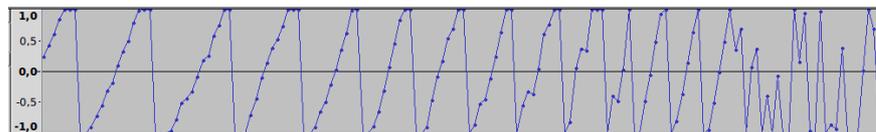


Figure 50: The error signal when the Costas loop failed to lock

I also created a GNU Radio test bench for this function, which allowed me to compare my algorithm to the Costas loop built into GNU Radio.

The decision directed version of the Costas loop needs to decide about the closest symbol to the given sample. Even if the non-data aided version is used, we still need to decide about it in the next step, because we want to turn the output samples into bits.

For this task, we can split the constellation into areas around the ideal constellation points. If the sample falls into a given area, we can return the symbol mapped to it.

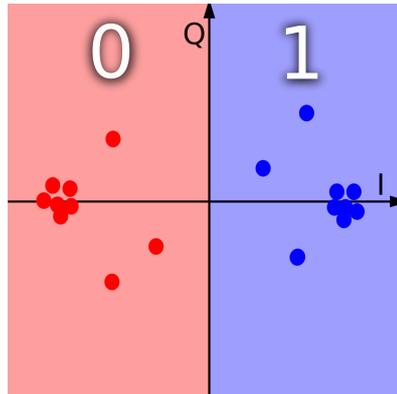


Figure 51: Operation of slicer on BPSK signal

In case of BPSK, it is a very easy task: if the real part of the symbol is positive, we have a "1" symbol, else we have a "0" symbol, as on Figure 51.

The CSDR function to carry out this operation is called *binary_slicer_f_u8*, as hard symbol decision is also referred to as "slicer". The syntax of the related CSDR function is below. It applies hard symbol decision to the complex baseband input signal. It outputs a 0x00 byte or a 0x01 byte based on the decision.

```
cldr binary_slicer_f_u8
```

5.5 Demodulating RTTY

Radioteletype (RTTY) is a system that originally consisted of two or more electromechanical teleprinters in different locations, connected over a wireless link [22]. This mode has a long history on amateur radio bands: amateur radio operators have been using modified commercial teleprinters since the 1950s.

RTTY uses a 2-FSK modulated signal, which means transmitting two different tones for "0" and "1" symbols (also called "space" and "mark"), with a tone spacing of 170 Hz,

and symbol rate of 45.45 baud. The line coding is similar to UART: there is a start bit, a stop bit, and there are data bits in between. However, RTTY uses 5 data bits, encoded in Baudot code instead of ASCII.

The Baudot code consists of two tables, one for letters and another for numbers and symbols (figures). Two special codes were selected to change between tables. The Baudot code is shown in Table 6.

	Letter	Figure		Letter	Figure
00000	NULL	NULL	11110	K	(
10000	E	3	00001	T	5
01000	\n	\n	10001	Z	+
11000	A	-	01001	L)
00100	SPACE	SPACE	11001	W	2
10100	S	\	00101	H	\$
01100	I	8	10101	Y	6
11100	U	7	01101	P	0
00010	\r	\r	11101	Q	1
10010	D	#	00011	O	9
01010	R	4	10011	B	?
11010	J	\a	01011	G	*
00110	N	0	00111	M	.
10110	F	@	10111	X	/
01110	C	:	01111	V	=
11011	FIGURE MODE		11111	LETTER MODE	

Table 6: Baudot code table [24]

I have implemented two different types of FSK demodulators for RTTY. The first command uses an FM detector to acquire frequency information from the signal:

```

arecord -r48000 -c1 -fs16_LE | \
csdr convert_s16_f | \
csdr dsb_fc | \
csdr shift_addition_cc $(csdr =-2000/48e3) | \
csdr fir_decimate_cc 48 | \
CSDR_FIXED_BUFSIZE=128 csdr simple_agc_cc 0.0001 0.5 | \
csdr peaks_fir_cc 31 $(csdr =85/1e3) $(csdr =-85/1e3) | \
CSDR_FIXED_BUFSIZE=128 csdr fmdemod_quadri_cf | \
CSDR_FIXED_BUFSIZE=128 csdr gain_ff 40 | \
CSDR_FIXED_BUFSIZE=256 csdr serial_line_decoder_f_u8 22 5 1 | \
CSDR_FIXED_BUFSIZE=4 csdr rtty_baudot2ascii_u8_u8

```

The other FSK demodulator compares the output of the space and mark filters to find out about the transmitted symbol:

```
arecord -r48000 -c1 -fS16_LE | \  
csdr convert_s16_f | \  
csdr dsb_fc | \  
csdr shift_addition_cc $(csdr =-2000/48e3) | \  
csdr fir_decimate_cc 48 | \  
CSDR_FIXED_BUFSIZE=128 csdr simple_agc_cc 0.0001 2 | \  
CSDR_FIXED_BUFSIZE=128 csdr bfsk_demod_cf $(csdr =2*85/1e3) 21 | \  
CSDR_FIXED_BUFSIZE=256 csdr serial_line_decoder_f_u8 22 5 1 | \  
CSDR_FIXED_BUFSIZE=1 csdr rtty_baudot2ascii_u8_u8
```

For both commands, first the audio data is acquired with *arecord*, then using the format conversion, shift and FIR decimation operations in CSDR, the result is the complex baseband FSK signal.

5.6 FSK demodulation

Both commands use bandpass filters, the center of which are the frequencies of the space and mark tones. A peak filter is the opposite of a notch filter, passing only frequencies in the immediate surroundings of a given center frequency. With *peaks_fir_cc*, a pair of peak filters are generated and their taps are added together. The resulting filter passes only the two FSK tones. Longer filter length means sharper passband. The syntax of related commands is below:

```
csdr firdes_peak_c <rate> <length> [window [--octave]]  
csdr peaks_fir_cc <taps_length> [peak_rate × N]
```

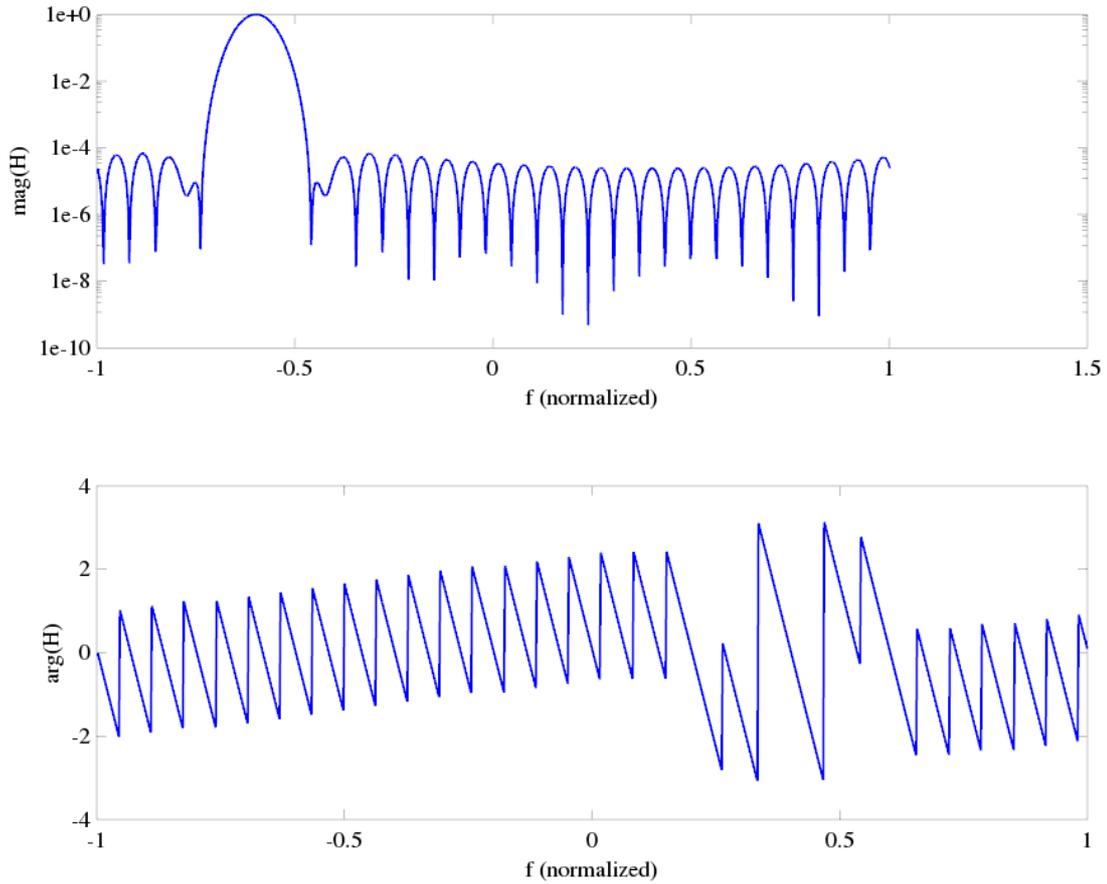


Figure 52: Transfer characteristic of peak filter with 31 taps

The taps of a single peak filter are calculated as:

$$h(t) = f_{\text{window}} \{ e^{j2\pi f_p t} \} \quad (22)$$

In (22), f_p is the frequency of the peak, and f_{window} is the window function.

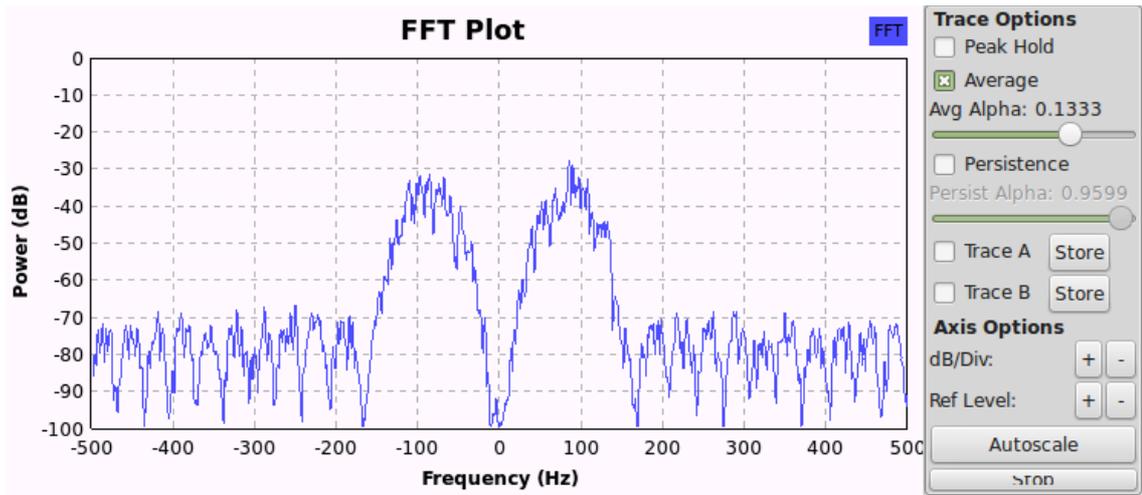


Figure 53: Filter passing two peaks in GNU Radio testbench

See a sample peak filter transfer characteristic on Figure 62, and Figure 53 for a GNU Radio simulation of two peaks at equal distance from DC, where the filter is driven with random input, so that the spectrum graph shows the approximate filter characteristic.

Now that we have our two tones selected with a bandpass filter, we need to know which tone is currently on. The first command reuses the FM demodulator I have already written: `fmdemod_quadri_cf`.

The other way to separate the two frequencies is to calculate the power of the signal at the output of each bandpass filter, and subtract them from each other. We can decide between mark and space by checking if the result is above or below zero, see Figure 54.

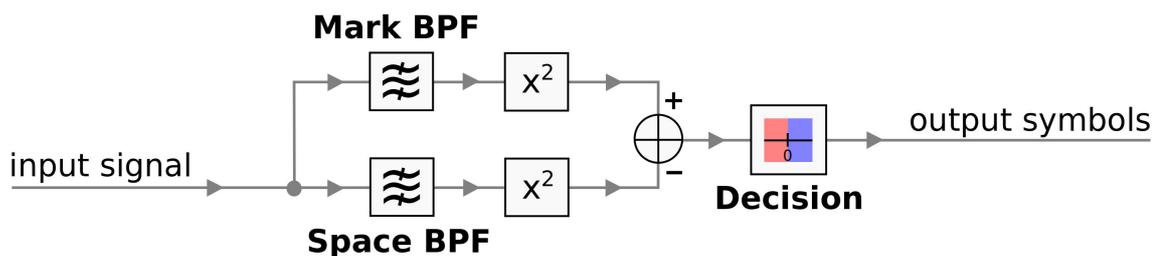


Figure 54: Block diagram of FSK demodulator comparing the output of mark and space band-pass filters

The command `csdr bfsk_demod_cf` implements a similar data flow, but it does not make the decision: its output is processed by the serial line decoder function instead.

```
csdr bfsk_demod_cf <spacing> <filter_length>
```

5.7 Serial line decoder

After applying any of the frequency discrimination methods above, our signal will look like as in Figure 55.

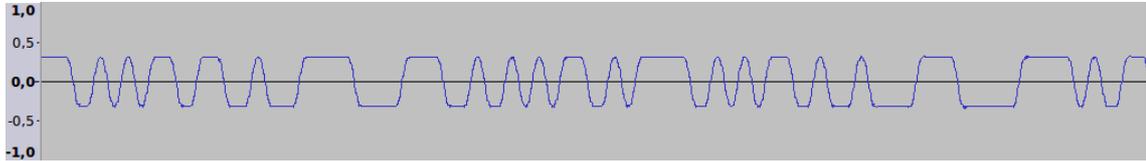


Figure 55: FSK signal after frequency detector stage

The next problem to solve is symbol timing synchronization. The interesting part is, RTTY works very similar to UART, with the difference of having 5 data bits instead of 8. An UART line is high by default. The UART receiver starts when it captures the zero crossing at the beginning of the start bit, thus synchronization happens at each character.

I implemented a generic algorithm for recovering data from a sampled serial line, thus it is not limited to decoding RTTY. The number of data bits and stop bits can be configured. It is possible to put an ADC on a serial line, record a block of samples and then process them with this command. The syntax is as follows:

```
cldr serial_line_decoder_f_u8 <samples_per_bits> [databits [stopbits]]
```

It synchronizes on the first falling edge of the start bit, samples the given number of data bits, and also checks the stop bit (see Figure 56).

It averages several samples from the middle of the data bit in order to make the bit decision. (Currently, the number of samples to average can be adjusted internally in the code.)

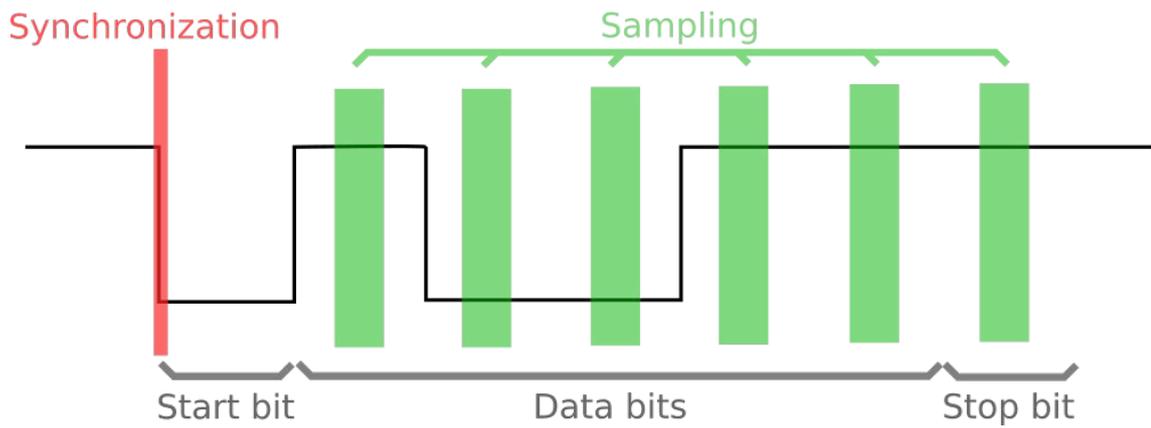


Figure 56: Internal operation of serial line decoder

5.8 Baudot decoder

As the data returned from `serial_line_decoder_f_u8` is still in Baudot code, we need to turn it to ASCII, for which I have created a new function:

```
cldr rtty_baudot2ascii_u8_u8
```

See Table 6 for reference. As demonstrated on Figure 57, CSDR can be used to decode real-world RTTY transmissions.

```

pcfl@ssd-mint: ~/Asztal/szakdogaja/dipterv1/grc
Fájl Szerkesztés Nézet Keresés Terminál Sűgő
→ grc cat ../grc/tty_real_8k.raw | csdr mono2stereo_s16 | csdr convert_s16_f |
csdr bandpass_fir_fft_cc 0 0.5 0.1 | csdr shift_addition_cc -0.051875 | csdr fm
demod_quadri_cf | csdr gain_ff 40 | csdr serial_line_decoder_f_u8 176 5 1 | csdr
tty_baudot2ascii_u8_u8
bandpass_fir_fft_cc: window = HAMMING
bandpass_fir_fft_cc: (fft_size = 128) = (taps_length = 39) + (input_size = 90) -
1
(overlap_length = 38) = taps_length - 1
shift_addition_cc: reinitialized to -0.051875
bandpass_fir_fft_cc: filter initialized, low_cut = 0, high_cut = 0.5
VVEWIWVMXNSG 73 @
QRZ? DE ER5LL K QLMLSX
ER5LL DE IW14$4 821RHR KQ
IA1RHR GM UR 067 067 067 IW1RHR KKVK/
R5LL DE IW1RHR 599 GE G QWVQWKKMY1)QQMDKQH IW1RHR QSL 73 @
QRZ? DE ER5LL K X.. PA3I PA3I PA3I V
PA3I GM UR 068 068 068 PA3I KX Q.. ER5LL 599 081 081 081 PA3I DVEABXDK'=
/
=QRZVB JDHJV.LGXVVMKVQ
DE IL1G IB1G Q2/QR5LLD
DE IB1XPB1G PYOPIB1G KBUVWQXQF1ER5LL ER5LL 599 IM IM XTXXVYKXXXQFQIER5LL AR5LL 5
ER5LLPSKK NR? NR? AGN AGN BKQYVKVMUQ17-*CYJC+$ZGPPL ER9)) /3409GVQER=
→ grc █

```

Figure 57: RTTY decoding with CSDR

5.9 Demodulating M-FSK

CSDR contains commands that are tailored for decoding multiple frequency-shift keying. The generic slicer is to be used after the FM demodulator. It allows you to map the frequency values to N distinct symbols:

```
csdr generic_slicer_f_u8 <n_symbols>
```

For example, if $n_symbols$ is set to 3, it will output 0x00 for values in the $(-\infty; 0.5)$ range, 0x01 for values in the range $[-0.5, 0.5)$ and 0x02 for values in $[0.5, \infty)$.

A demonstration of 4-FSK demodulation has been included in CSDR, which can be started by loading `grc_tests/test_m_fsk.grc` into GNU Radio Companion. The 4-FSK signal is generated by GNU Radio (see Figure 58), and the symbols are correctly decoded by CSDR and written into a temporary file.

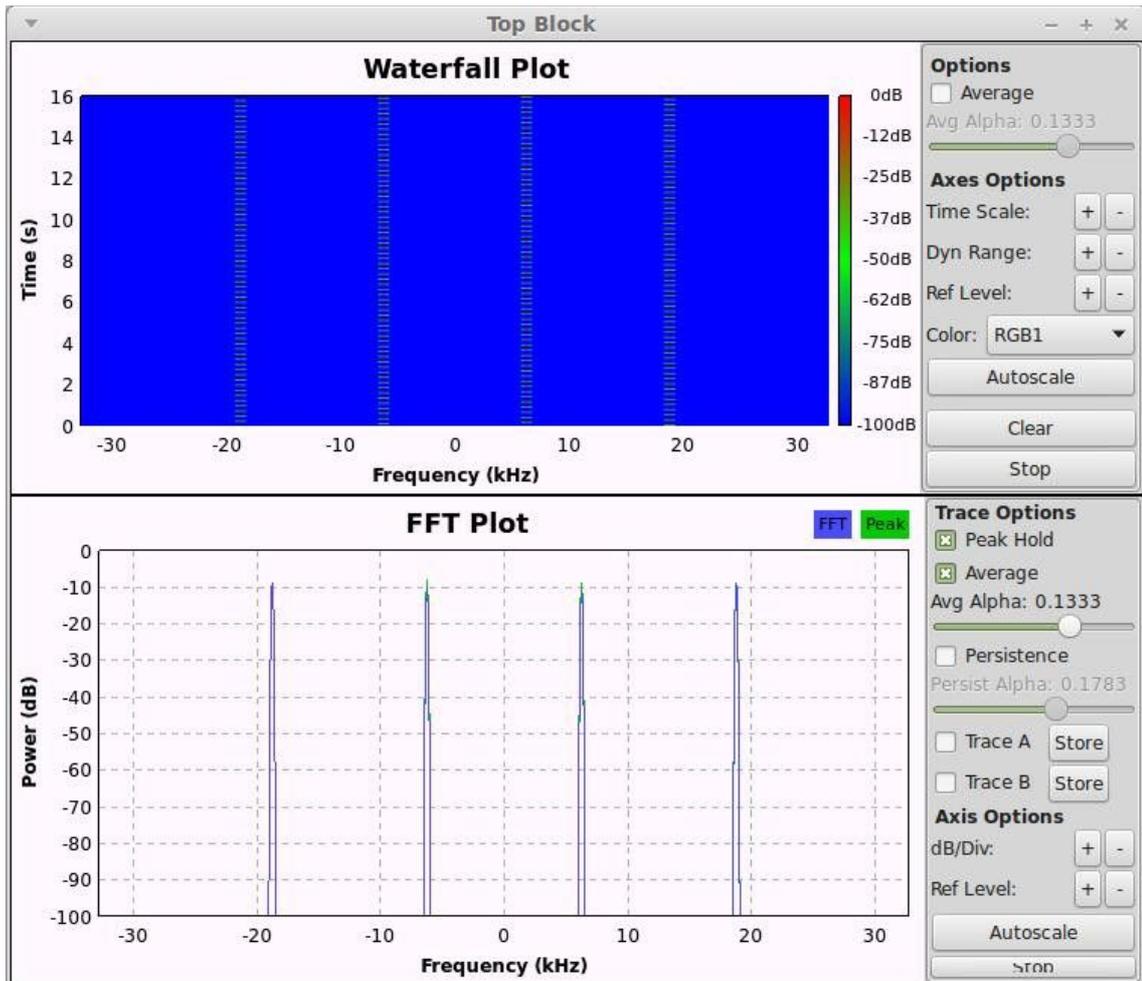


Figure 58: 4-FSK signal generated by GNU Radio is used as the input for the demodulator built with CSDR

5.10 Demodulating FSK signals transmitted using the CC1111 wireless MCU

When I came up with the idea of CSDR, I imagined it as an universal command line tool that can perform simple digital signal processing tasks for Software Defined Radio. The purpose of this section is to show that with its recently introduced support for digital modes, it is possible to use it for decoding signals of wireless devices that use simple modulation schemes.

This is important because today IoT is a growing market, and many IoT devices communicate wireless, mostly using WiFi, Bluetooth or the mobile network to connect to the Internet. However, some devices communicate on the ISM bands with no

encryption, and many IoT vendors have low security standards (not limited to wireless communication, but for the whole system), so new vulnerabilities are found regularly by researchers.

Reverse engineering of wireless protocols often starts with using an SDR receiver to record some packets sent by the target, and then analyzing the baseband I/Q signal. For the latter, the work the received signal is often obtained at a considerably good SNR, and the task is to build a working demodulator as soon as possible, rather than building a receiver that works well even under bad conditions. For example, fine tuning the parameters of the Costas loop for maximum performance are less important than having the decoded data at hand in this application.

As CSDR can be used as a digital modem, it could generate a modulated signal from crafted packets, and also for demodulation during interception. In the following part, an example follows to demonstrate that FSK transmissions sent by the popular sub-GHz wireless MCU, Texas Instruments CC1111 can be decoded with CSDR.

The YARD Stick One, produced by Great Scott Gadgets, can be considered as a development board for the CC1111. With the open source RfCat firmware installed, the RF SoC on the panel can fully be controlled over USB: setting receiver and transmitter parameters, sending and receiving data can be carried out from a convenient IPython shell on the host PC. While the YARD Stick One is a special development tool, many commercial devices use the same chip, or use similar simple FSK modulation.

I started by setting up the YARD Stick One to send a known message.

```
$ rfcats -r #open the rfcats IPython shell

#set frequency and data rate:
d.setFreq(433000000)
d.setMdmDRate(1000)

#check radio parameters, make a note of the sync word:
print d.reprRadioConfig()

#transmit 100 packets:
for i in range(0,100): d.RFxmmit('test')
```

The radio is set up to transmit the message "test" at 1000 symbol/second, at 433 MHz. While transmitting, I used an RTL-SDR to receive the signal.

```
rtl_sdr -f 433000000 rfcatt.cu8
```

Note that you need to choose a frequency where you are legally allowed to transmit, or you need to use a direct cable connection via an attenuator between the YARD Stick and the SDR receiver.

In case of a completely unknown signal, the work would continue by analyzing the signal to determine the modulation, estimating the symbol rate, etc. This could be done in GNU Radio or other tools, including the one shown in Figure 59, called Inspectrum.

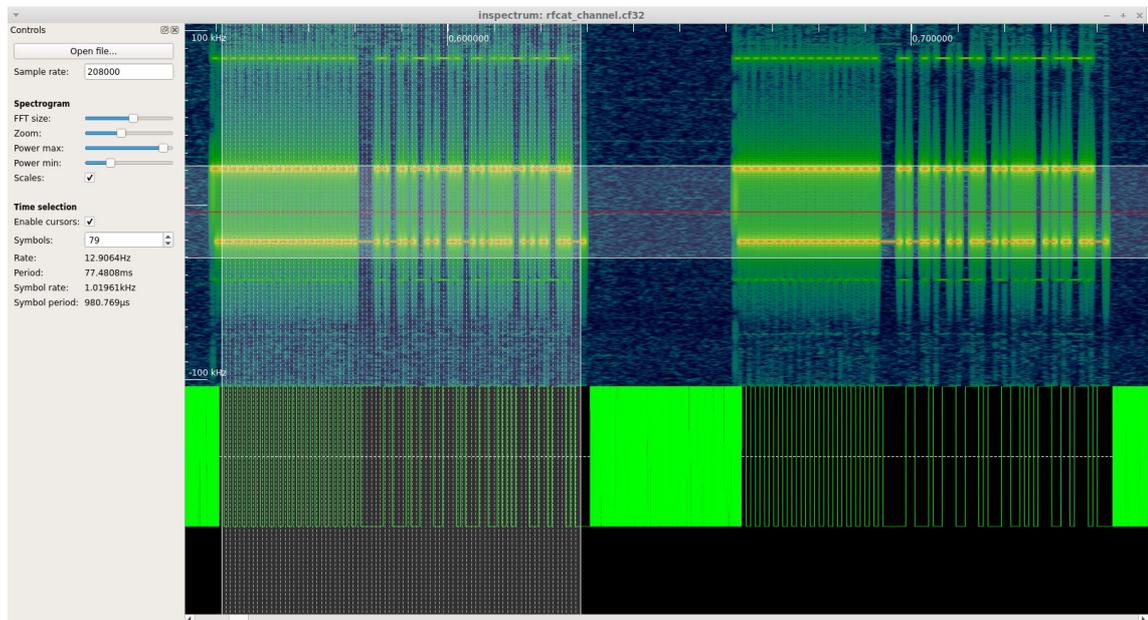


Figure 59: Analyzing recorded I/Q signal with Inspectrum

Inspectrum shows us the spectrogram of recorded I/Q files, and it allows us to demodulate FSK manually by visually selecting the center frequency, the bandwidth and the bit duration. I found the feature to put a grid on a packet to allow manual timing recovery very intuitive, see Figure 60.

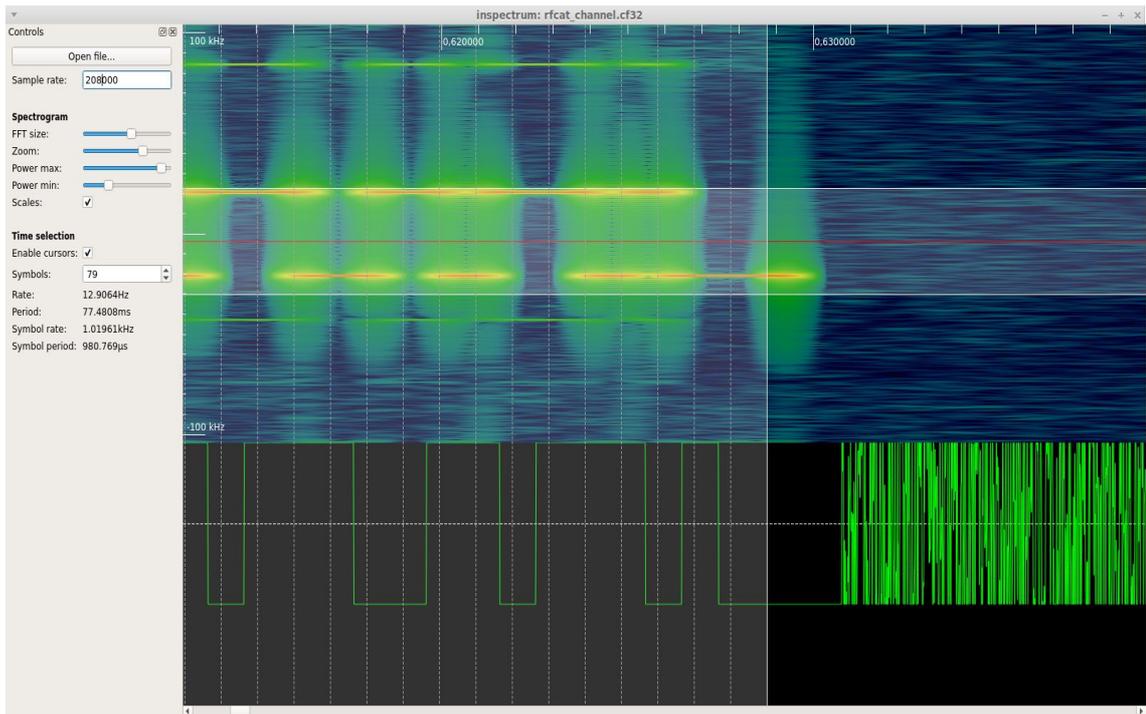


Figure 60: Grid for manual timing recovery in Inspectrum, with the derived FSK demodulated signal on the bottom subpane

I have exported the demodulated bits to the standard output, and broken the packet into different parts:

```

1, 0, 1, 0, 1, 0, 1, 0, --- preamble
1, 0, 1, 0, 1, 0, 1, 0,
1, 0, 1, 0, 1, 0, 1, 0,
1, 0, 1, 0, 1, 0, 1, 0,

0, 0, 0, 0, 1, 1, 0, 0, --- sync word: 0x0C4E
0, 1, 0, 0, 1, 1, 1, 0,

0, 1, 1, 1, 0, 1, 0, 0, --- 't'
0, 1, 1, 0, 0, 1, 0, 1, --- 'e'
0, 1, 1, 1, 0, 0, 1, 1, --- 's'
0, 1, 1, 1, 0, 1, 0, 0 --- 't'

```

By looking at the datasheet of the CC1111 [10], one can easily find that the messages follow the structure above. I have also verified that the transmitted sync word matches the one in RfCat settings.

Manual tools like Inspectrum are not convenient if one wants to decode a high amount of packets. On that reason, I built an FSK demodulator using CSDR, similar to the one for RTTY in section 5.5. While tuning the parameters of the timing recovery algorithm,

I used its feature to plot the internal state with GNU Octave into PNG files, so that I could make sure that each symbol is sampled at the correct phase (see Figure 61). I have observed that the Gardner algorithm successfully locked on the signal while receiving the preamble.

One of the advantages of CSDR is that there is an easy way to see if the timing recovery block managed to sample at the maximum eye opening. This is hard to do with a GNU Radio flowgraph, as one has to manually, visually compare the results with the original baseband signal in time domain.

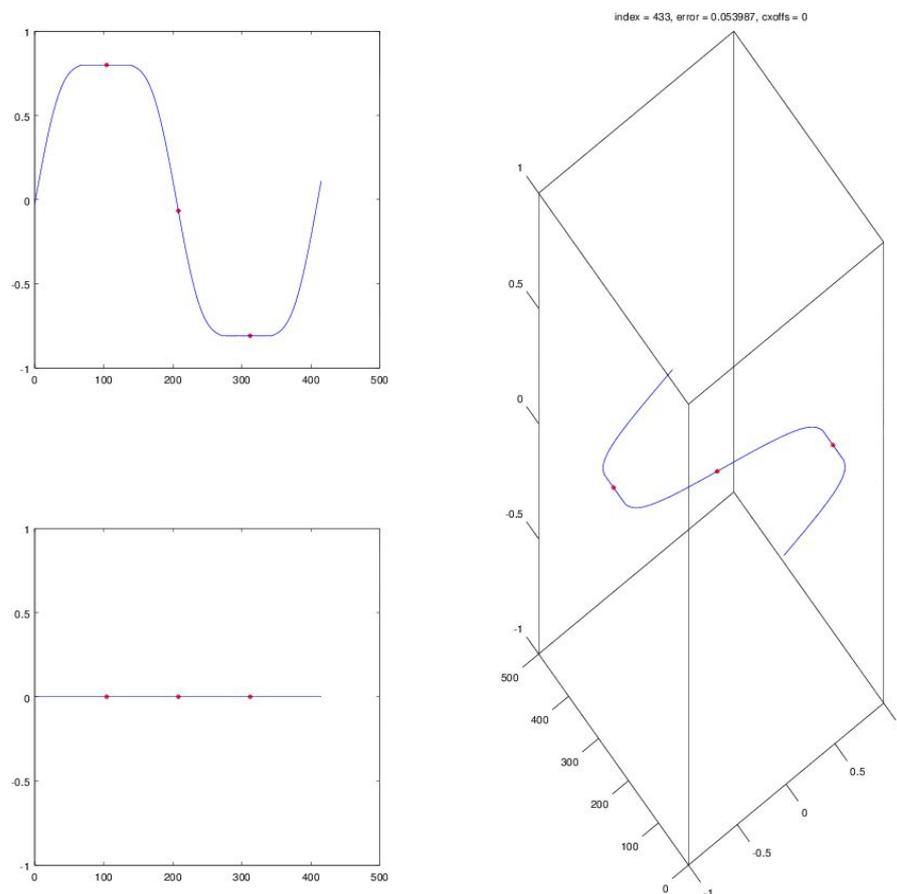


Figure 61: Gardner algorithm locking on the FSK signal sent by the CC1111 on the YARD Stick One

The following command demodulates the recorded signal, finds the preamble and the synchronization word, and prints the bits of the packet following them:

```
cat rfcats.cu8 | csdr convert_u8_f | \
```

```

csdr fir_decimate_cc 10 $(csdr =50000./24e5) | \
csdr shift_addition_cc $(csdr =42000/240000.) | \
csdr fmdemod_quadri_cf | csdr gain_ff 4 | csdr dsb_fc | \
csdr bandpass_fir_fft_cc -0.0001 0.0001 0.03 | \
csdr timing_recovery_cc GARDNER 204 0.5 2 --add_q | \
csdr realpart_cf | csdr binary_slicer_f_u8 | \
csdr pattern_search_u8_u8 32 0 1 0 1 0 1 0 1 \
    0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, | xxd -g 1

```

In the output of the command one can inspect the bits of the first two packets:

```

00000000: 00 01 01 01 00 01 00 00 00 01 01 00 00 01 00 01 .....
00000010: 00 01 01 01 00 00 01 01 00 01 01 01 00 01 00 00 .....
00000020: 00 01 01 01 00 01 00 00 00 01 01 00 00 01 00 01 .....
00000030: 00 01 01 01 00 00 01 01 00 01 01 01 00 01 00 00 .....
(...)

```

Next I put together a command that receives the wireless transmissions of the CC1111 with the RTL-SDR, demodulates them and decodes the 4-byte test messages in real-time:

```

rtl_sdr -f433000000 -s2400000 - | \
csdr convert_u8_f | \
csdr fir_decimate_cc 10 $(csdr =50000./24e5) | \
csdr shift_addition_cc $(csdr =35000/240000.) | \
csdr fmdemod_quadri_cf | csdr gain_ff 4 | csdr dsb_fc | \
csdr bandpass_fir_fft_cc -0.0001 0.0001 0.03 | \
csdr timing_recovery_cc GARDNER 240 0.5 2 --add_q | \
csdr realpart_cf | csdr binary_slicer_f_u8 | \
csdr pattern_search_u8_u8 32 1 0 1 0 1 0 1 \
    0 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, | \
csdr pack_bits_8to1_u8_u8

```

Figure 62 shows this command in action.

/dev/urandom generates uniform noise), some additional calculations are needed to get Gaussian white noise. While both types of noise have the same flat spectrum, their amplitude distribution is different, which can be observed on their histograms.

The Box-Muller transform (23) allows us to transform two uniformly distributed random variables, U_0 and U_1 into two random variables with standard normal distribution, Z_0 and Z_1 .

$$\begin{aligned} Z_0 &= R \cos(\Theta) = \sqrt{-2 \ln U_0} \cos(2\pi U_1) \\ Z_1 &= R \sin(\Theta) = \sqrt{-2 \ln U_0} \sin(2\pi U_1) \end{aligned} \quad (23)$$

Based on this formula, the method to generate 32-bit complex valued Gaussian white noise is as follows:

1. Read 8 bytes from */dev/urandom*, as 32 bit integers (*int*).
2. Cast them into 32-bit *float* and divide them by `INT_MAX`.
3. These numbers will be the current value of U_0 and U_1 . Apply the formula above to get Z_0 and Z_1 .
4. Use the resulting values for Z_0 and Z_1 as the real and the imaginary part of the output random number.

There are also CSDR functions to generate noise in itself:

```
cldr uniform_noise_f  
cldr gaussian_noise_c
```

As a note, generating noise based on */dev/urandom* can be slow in some applications. I measured a 10 Mbyte/s maximum reading speed. On this reason, a custom pseudorandom number generator (PRNG) could be used, but the easiest way to speed up this processing function was to pre-generate Gaussian noise using *cldr gaussian_noise_cc* into a file, and pass that to *cldr awgn_cc*. This is what the optional *--awgnfile* parameter can be used for.

The *cldr awgn_cc* function has been used to create noisy conditions while testing the my demodulators against Fldigi, and also for generating the variance diagrams in section 4.5.

5.12 Functions helping development

At the time of writing, the CSDR command-line tool had 103 functions. While executing "*csdr --help*" gives a list of these functions, I have realized the need of easy searching within the functions for their syntax, thus the "*?<keyword>*" search function has been introduced. As an example, see the new command and its result below.

```
$ csdr ?psk
csdr ?: csdr 2>&1 | grep psk
    psk31_varicode_encoder_u8_u8
    psk31_varicode_decoder_u8_u8
    psk_modulator_u8_c <n_psk>
    psk31_interpolate_sine_cc <interpolation>
    bpsk_costas_loop_cc <loop_bandwidth> <damping_factor> <gain> (...)
```

The command above filters the help for CSDR commands containing "*psk*".

Another new function, "*??<keyword >*" opens the browser and jumps straight to the given function in the CSDR documentation on GitHub.

I have also repeatedly faced that some CSDR functions take parameters that are convenient to be calculated using a formula. See the beginning of the CSDR command for a simple NFM demodulator:

```
rtl_sdr -s 2400000 -f 145000000 -g 20 - | \
csdr convert_u8_f | \
csdr shift_addition_cc \
    `python -c "print float(145000000-145350000)/2400000"` | \
csdr fir_decimate_cc 50 0.005 HAMMING | \
csdr fmdemod_quadri_cf | \
(...)
```

In this example, the *rate* parameter of *csdr shift_addition_cc* is calculated using Python. (Similarly, in GNU Radio Companion, one can give mathematical expressions to parameters of blocks, and these are evaluated as Python expressions.) It looked convenient to create an easy to remember shorthand for calculations like that, resulting in the "*=<expression>*" CSDR function. A simple example is shown below:

```
$ csdr =3+4*5+pi
26.1415926536

$ csdr '=(3+4)*5+pi'
```

38.1415926536

This method also uses Python for evaluating expressions. If there are parentheses in the expression, it is advised to put it in quotes, as the shell would attempt to parse the parentheses. Now the same beginning part of the NFM demodulator looks as below:

```
rtl_sdr -s 2400000 -f 145000000 -g 20 - | \  
csdr convert_u8_f | \  
csdr shift_addition_cc $(csdr '=float(145000000-145350000)/2400000')|\  
csdr fir_decimate_cc 50 0.005 HAMMING | \  
csdr fmdemod_quadri_cf | \  
(...)
```

6 Conclusion

The main objective of this thesis has been studying various methods and algorithms used in digital modems. The topics I was most interested in were carrier and symbol timing recovery algorithms, as these are essential for building demodulators. I really enjoyed implementing functional parts of BPSK and FSK demodulators, and putting them together into working demodulator DSP chains.

In the future I am planning to add the RTTY receiver to the web UI, and I am also interested in improving the performance of the BPSK31 demodulator.

7 Acknowledgement

I would like to express my gratitude to my supervisor, Péter Horváth, PhD for teaching me so much about signal processing and wireless communication. Furthermore, I would like to thank everyone who uses my open source projects, which inspires me to learn more about Software Defined Radio.

8 Bibliography

- [1] R. X. Seger, "rx-tools", retrieved on 2017. 04. 04. from:
https://github.com/rxseger/rx_tools
- [2] A. Retzler, "Software Defined Radio Receiver Application with Web-based Interface (BSc Thesis)", retrieved on 2017. 04. 14. from: <http://openwebbrx.org/bsc-thesis.pdf>
- [3] B. Farhang-Boroujeny, *Signal Processing Techniques for Software Radios*, Lulu Press, Inc., 2010.
- [4] Wikipedia, "Raised-cosine filter", retrieved on 2016. 05. 22. from:
https://en.wikipedia.org/wiki/Raised-cosine_filter
- [5] Wikimedia Commons, "Consecutive raised-cosine impulses, demonstrating zero-ISI property", released under CC-BY-SA 3.0, retrieved on 2017. 05. 20. from:
<https://commons.wikimedia.org/wiki/File%3ARaised-cosine-ISI.svg>
- [6] B. Umashankar, S. Aswathy, N. Prasad , C. Bhattacharya, K. K. Naik, "SOQPSK – A Spectrally Efficient Modulation Scheme for Aeronautical Telemetry Applications", *International Journal of Engineering Science and Innovative Technology (IJESIT)*, Vol. 2, Issue 2, 2013. 03., retrieved on 2017. 04.21. from:
http://www.ijesit.com/Volume%202/Issue%202/IJESIT201302_59.pdf
- [7] J. Taylor (K1JT), "The JT65 Communications Protocol", retrieved on 2016. 05. 23. from: <http://physics.princeton.edu/pulsar/K1JT/JT65.pdf>
- [8] Espressif Inc., "ESP8266EX Overview", retrieved on 2017. 04. 07. from:
<https://espressif.com/en/products/hardware/esp8266ex/overview>
- [9] D. Green, "GMRT photos", retrieved on 2017. 04. 04. from:
<https://www.mrao.cam.ac.uk/~dag/GMRT-PICS/2004-02b.jpg>
- [10] Texas Instruments Inc., "Low-Power SoC (System-on-Chip) with MCU, Memory, Sub-1 GHz RF Transceiver, and USB Controller", retrieved on 2017. 05. 20. from:
<http://www.ti.com/lit/ds/symlink/cc1110-cc1111.pdf>

- [11] S. L. Karty (N5SK), "PSK31 Spec", retrieved on 2016. 05. 22. from:
<http://www.arrl.org/psk31-spec>
- [12] The MathWorks Inc., "Early-Late Gate Timing Recovery", retrieved on 2017. 04. 19. from: <https://www.mathworks.com/help/comm/ref/earlylategatetimingrecovery.html>
- [13] F. M. Gardner, "A BPSK/QPSK Timing-Error Detector for Sampled Receivers", IEEE Transactions on Communications, vol. COM-34, 1986. 05., retrieved on 2017. 04. 19. from: <http://ieeexplore.ieee.org/document/1096561/>
- [14] The MathWorks Inc., "Gardner Timing Recovery", retrieved on 2017. 04. 19. from: <https://www.mathworks.com/help/comm/ref/gardnertimingrecovery.html>
- [15] NuRAN Wireless, "Symbol timing recovery methods for digital I/Q demodulator", retrieved on 2017. 04. 19. from: <https://www.nutaq.com/blog/symbol-timing-recovery-methods-digital-iq-demodulator>
- [16] The MathWorks Inc., "Squaring Timing Recovery" retrieved on 2017. 04. 21. from: <https://www.mathworks.com/help/comm/ref/squaringtimingrecovery.html>
- [17] K. H. Mueller, M. Müller, "Timing Recovery in Digital Synchronous Data Receivers", IEEE Transactions on Communications, Vol. COM-24, No. 5, 1974. 05.
- [18] The MathWorks Inc., "comm.SymbolSynchronizer System object" retrieved on 2017. 04. 19. from:
<https://www.mathworks.com/help/comm/ref/comm.symbolsynchronizer-class.html>
- [19] M. Renfors, "Synchronisation in Digital Receivers", retrieved on 2017. 04. 27. from: <http://www.cs.tut.fi/kurssit/TLT-5806/Synch.pdf>
- [20] U. Mengali; A. N. D'Andrea, *Synchronization Techniques for Digital Receivers*, Plenum Press, New York, 1997.
- [21] T. Rondeau, "Control Loop Gain Values", retrieved on 2017. 04. 21. from: <https://www.mathworks.com/help/comm/ref/squaringtimingrecovery.html>
- [22] Wikipedia, "Radioteletype", retrieved on 2016. 05. 22. from: <https://en.wikipedia.org/wiki/Radioteletype>
- [23] Wikimedia Commons, "Costas loop", available under CC-BY-SA 4.0 license.

Retrieved on 2016. 05. 22. from:

https://en.wikipedia.org/wiki/Costas_loop#/media/File:Costas_loop_model.svg

[24] Zs. Krüpl (HG2ECZ), "Magyar rádióamatőrök információs lapja", retrieved on 2016. 05. 23. from: <http://www.ham.hu/>

[25] Wikipedia, "Costas loop after synchronization", available under CC-BY-SA 3.0 license, retrieved on 2016. 05. 22. from:

https://en.wikipedia.org/wiki/Costas_loop#/media/File:Costas_loop_after_sync.svg

[26] ESA Operations, "Signal from @ESA_EDM...", retrieved on 2017. 05. 21. from:

<https://twitter.com/boomtnt46/status/788748256608129024>