Custom Sink block for GNU Radio: emitting Hellschreiber from a Raspberry Pi

J.-M Friedt, FEMTO-ST Time & Frequency, Besançon, France

GNU Radio is a discrete time digital signal processing framework centered on a scheduler transferring data packets from one processing block to another, acquiring data in so called Source blocks and processing until reaching Sink blocks. The data-flow is defined either in Python or C++, but the most common usage of GNU Radio is through its graphical user interface GNU Radio Companion which allows for graphically selecting, configuring and connecting processing blocks. As opposed to post-processing tools such as GNU/Octave (the free opensource version of Matlab) or Python scientific libraries (Numpy, Scipy), GNU Radio allows for interactive tuning of parameters and continuous acquisition of live radiofrequency streams for real time processing. Although with the addition of external tools (ZeroMQ data transfer or networking capabilities through the Networking Tools) most processing can be achieved with hardly ever writing dedicated GNU Radio blocks, sometimes a new functionality is needed that requires becoming familiar with the Out of Tree (OOT) block architecture.

We will demonstrate in this article remote processing of the Hellschreiber digital mode generated by a single board computer Raspberry Pi 4 running GNU Radio and collecting the data using an RTL-SDR Digital Video Broadcast-Terrestrial (DVB-T) used as general purpose Software Defined Radio (SDR) receiver. Indeed, because GNU Radio Companion is a Python3 code generator that does not need a graphical interface to be executed, GNU Radio is ideally suited to run on embedded hardware including the Raspberry Pi 4: since each processing block is allocated its own thread, the multi-core architecture of modern processors is best used by compiling an optimized operating system, in our case GNU/Linux generated using the Buildroot framework for a consistent set of tools (cross-compilation toolchain), operating system kernel, libraries and userspace applications optimized for the targeted processor architecture. Although GNU Radio will run on computers running Microsoft Windows, the community of developers is working exclusively with GNU/Linux which is by far the best supported framework and most suitable for embedded applications.

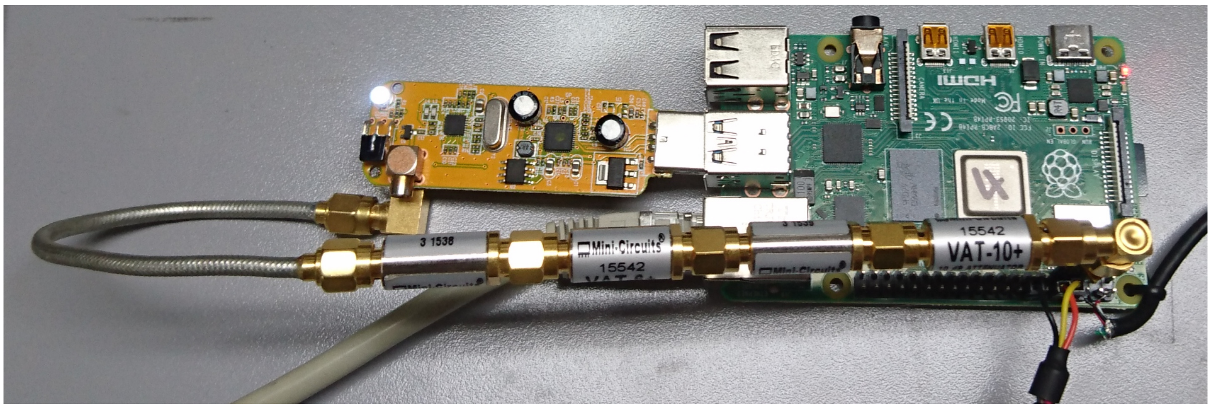


Figure 1: Wired connection between GPIO 4 of a Raspberry Pi 4 acting as emitter, and the RTL-SDR dongle acting as receiver, through 28 dB attenuators. The Raspberry Pi output signal should not be emitted over the air unless well characterized and filtered for spurious spectral components.

Despite the controversial nature of using the digital output of the Raspberry Pi as a radiofrequency emitter due to the many spurious spectral components, we believe that the educational benefit in writing the `gr-rpitx` GNU Radio Sink block overcomes the drawback of the poor spectral purity (Fig. 1). We have developed a basic Raspberry Pi signal sink feeding the General Purpose Input Output (GPIO) pin with a stream of In phase and Quadrature (IQ) components in order to generate associated radiofrequency signals. While the initial purpose was to demonstrate how a scalar network analyzer for characterizing radiofrequency resonators and filters could be assembled with only a Raspberry Pi and a RTL-SDR DVB-T dongle, the demonstration has extended to generating all sorts of IQ streams and emitting the resulting signal from the Raspberry Pi GPIO. In this demonstration, the fax-like Hellschreiber digital mode will be generated by the Raspberry Pi GPIO.

1 Raspberry Pi sink block

Using a digital general purpose input/output (GPIO) pin as radiofrequency source has been initially investigated for FM transmission by tuning the phase locked loop parameters for frequency modulated emissions [1]. É. Courjaud (F5OEO) has extended the principle to amplitude modulation by additionally tuning the current generated by the GPIO, and has packaged the software to a separate library: the frequency and amplitude tuning capability means that IQ streams will readily feed such a synthesizer, a natural output of GNU Radio signal processing capability. We include the `rpitx` library (`https://github.com/F5OEO/rpitx`) in the GNU Radio Out of Tree framework to benefit from the available signal processing blocks: the C++ class constructor initializes the direct memory access (DMA) and peripherals while the "work" function called every time the scheduler feeds the processing block with a new data packet will fill the DMA buffer [2]. Based on the `https://github.com/F5OEO/rpitx/blob/master/src/sendiq.cpp` IQ streaming example, packaging a GNU Radio Sink block is a matter of complying with the software architecture as described at `https://wiki.gnuradio.org/index.php/OutOfTreeModules`. All demonstrations will rely on the 3.8 version of GNU Radio. It is worth mentioning that processing blocks only have access to the IQ stream packets, and that the Python or C++ script connecting the blocks to each other do not have access to the data: scheduling and processing the datastreams are two different levels of abstractions clearly separated in GNU Radio.

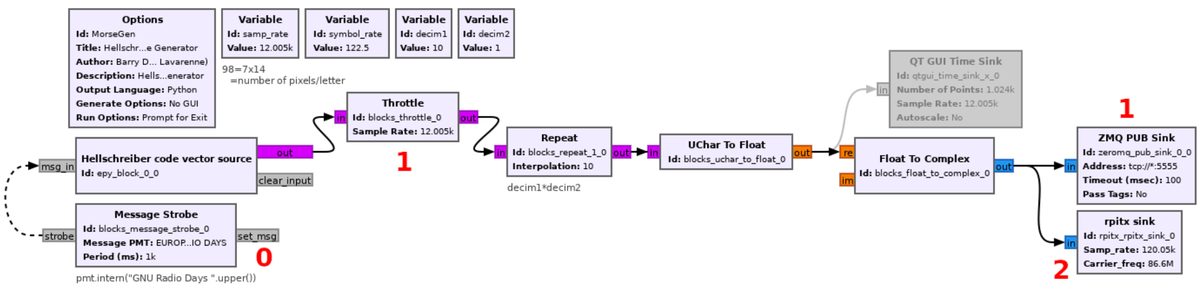
The resulting block can only handle a limited bandwidth defined by the transfer rate from user space to the DMA: we have observed continuous streaming up to about 400 kS/s, a bandwidth sufficient for analog Wideband FM broadcast as well as some of the digital communication of Digital Radio Mondiale (DRM) although the signal to noise ratio is insufficient for QAM modulation [2]. The data packet size is not known and buffer filling and transfer mechanisms, e.g. circular buffers, must be considered if a minimum number of items is needed as is the case for example for Fourier transforms. Furthermore, callback functions allow for dynamically tuning block parameters such as carrier frequency: in our case, changing the carrier frequency requires disabling the DMA clocking data transfer from the IQ stream to the hardware peripheral, before enabling back with the new parameters. Since the work function will keep on running during this reconfiguration, care must be taken to grant exclusive access (mutex mechanism) to the DMA resource and block the work function during reconfiguration by the callback function, complying with the general principles underlying the concurrent access to a shared resource by multiple threads.

2 Hellschreiber communication

The benefit of relying on GNU Radio to investigate a new communication protocol is illustrated with the Hellschreiber narrowband digital communication mode. Developed in the late 1920s and used during the second world war by the German Navy, this narrowband communication mode reminiscent of fax communication is still used by the ham community. Developing a transceiver will rely 1/ on a step by step channel simulation including noise and local oscillator frequency offsets to demonstrate proper operation of the emitter and receiver (Fig. 2) and 2/ replacing the simulated channel with the gr-rpitx emitter and RTL-SDR receiver, one of the many sink and sources made available by GNU Radio Companion (Fig. 3). T. Lavarenne has modified the morse generator provided by B. Duggan to generate a sequence of bytes representing the transmitted message in Hellschreiber [3, 4] format [5]: assessing the robustness and associated decoding block capability is best initially assessed in an ideal framework simulated by two GNU Radio flowgraphs connected through a ZeroMQ socket, and more specifically the non-connected (datagram-like) publish-subscribe (PUB-SUB) stream. This socket communication will on the one hand simulate the radiofrequency communication from emitter to receiver – we shall see later how to simulate the channel with additive noise and local oscillator frequency offsets between emitter and receiver – and introduces how GNU Radio can efficiently communicate with external software, here a decoder implemented in Python since ZeroMQ is available for most common languages.

3 Conclusion

This article aimed at introducing various means of adding functionalities to existing processing block of the GNU Radio framework, either as C++ block (gr-rpitx), Python block (Hellschreiber generator) or



Options
Id: MorseGen
Title: Hellschr...e Generator
Author: Barry D... Lavarenne)
Description: Hells...enerator
Output Language: Python
Generate Options: No GUI
Run Options: Prompt for Exit

Variable
Id: samp_rate
Value: 12.005k

98=7x14
=number of pixels/letter

Variable
Id: symbol_rate
Value: 122.5

Variable
Id: decim1
Value: 10

Variable
Id: decim2
Value: 1

Throttle
Id: blocks_throttle_0
Sample Rate: 12.005k

Hellschreiber code vector source
Id: epy_block_0_0

Message Strobe
Id: blocks_message_strobe_0
Message PMT: EUROP...IO DAYS
Period (ms): 1k

pmt.intern("GNU Radio Days ".upper())

Repeat
Id: blocks_repeat_1_0
Interpolation: 10

decim1*decim2

UChar To Float
Id: blocks_uchar_to_float_0

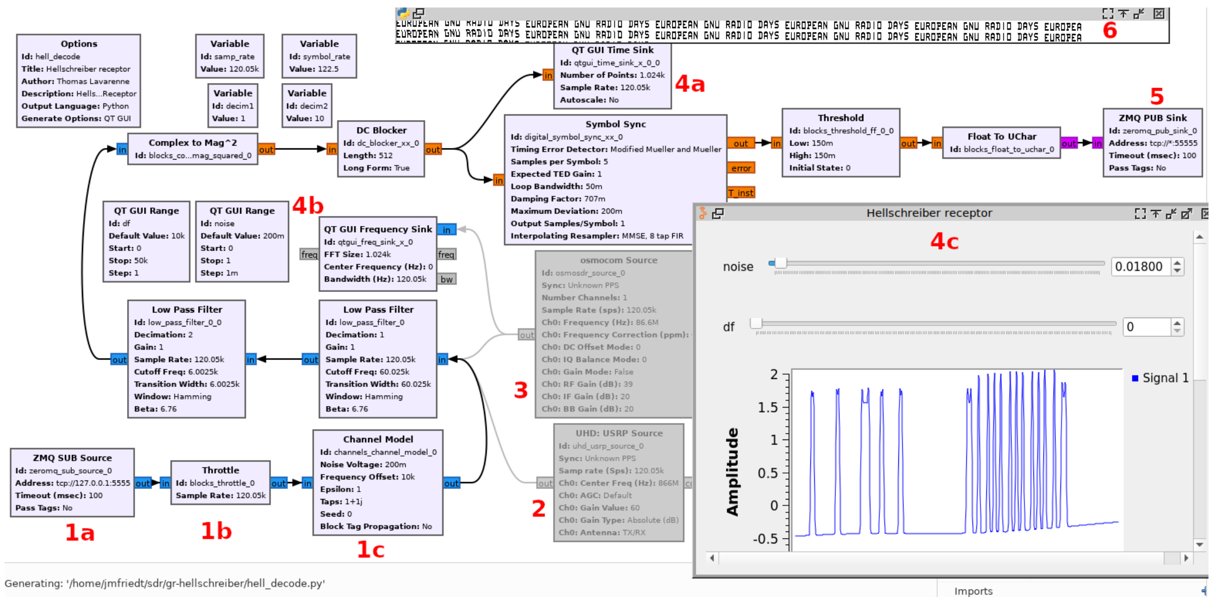
QT GUI Time Sink
Id: qtgui_time_sink_x_0
Number of Points: 1.024k
Sample Rate: 12.005k
Autoscale: No

Float To Complex
Id: blocks_float_to_complex_0

ZMQ PUB Sink
Id: zeromq_pub_sink_0_0
Address: tcp://*:5555
Timeout (msec): 100
Pass Tags: No

rpitx sink
Id: rpitx_rpitx_sink_0
Samp_rate: 120.05k
Carrier_freq: 86.6M

Figure 2: Hellschreiber emitter generating the radiofrequency signal encoding the string defined in the “Message Strobe” block (0). The Hellschreiber encoder is a Python script following the architecture of GNU Radio processing blocks, demonstrating that programming in C++ is not mandatory to add functionalities to GNU Radio. When no hardware interface limits the datarate a would be the case with only the virtual socket sink ZeroMQ Publish (1), a Throttle (1) block is mandatory to limit the datarate and avoid overwhelming the processing power of the host computer. When a hardware interface limits the datarate as would be the case with the Raspberry Pi gr-rpitx output interface (2), the Throttle block must be removed from the processing chain.



Options
Id: hell_decode
Title: Hellschreiber receptor
Author: Thomas Lavarenne
Description: Hells...Receptor
Output Language: Python
Generate Options: QT GUI

Variable
Id: samp_rate
Value: 120.05k

Variable
Id: symbol_rate
Value: 122.5

Variable
Id: decim1
Value: 1

Variable
Id: decim2
Value: 10

Complex to Mag^2
Id: blocks_co...mag_squared_0

DC Blocker
Id: dc_blocker_xx_0
Length: 512
Long Form: True

QT GUI Time Sink
Id: qtgui_time_sink_x_0_0
Number of Points: 1.024k
Sample Rate: 120.05k
Autoscale: No

Symbol Sync
Id: digital_symbol_sync_xx_0
Timing Error Detector: Modified Mueller and Mueller
Samples per Symbol: 5
Expected TED Gain: 1
Loop Bandwidth: 50m
Damping Factor: 707m
Maximum Deviation: 200m
Output Samples/Symbol: 1
Interpolating Resampler: MMSE, 8 tap FIR

Threshold
Id: blocks_threshold_ff_0_0
Low: 150m
High: 150m
Initial State: 0

Float To UChar
Id: blocks_float_to_uchar_0

ZMQ PUB Sink
Id: zeromq_pub_sink_0
Address: tcp://*:55555
Timeout (msec): 100
Pass Tags: No

QT GUI Range
Id: df
Default Value: 10k
Start: 0
Stop: 50k
Step: 1

QT GUI Range
Id: noise
Default Value: 200m
Start: 0
Stop: 1
Step: 1m

QT GUI Frequency Sink
Id: qtgui_freq_sink_x_0
FFT Size: 1.024k
Center Frequency (Hz): 0
Bandwidth (Hz): 120.05k

osmocom Source
Id: osmosdr_source_0
Sync: Unknown PPS
Number Channels: 1
Sample Rate (sps): 120.05k
Ch0: Frequency (Hz): 86.6M
Ch0: Frequency Correction (ppm): 0
Ch0: DC Offset Mode: 0
Ch0: IQ Balance Mode: 0
Ch0: Gain Mode: False
Ch0: RF Gain (dB): 39
Ch0: IF Gain (dB): 20
Ch0: BB Gain (dB): 20

Low Pass Filter
Id: low_pass_filter_0_0
Decimation: 2
Gain: 1
Sample Rate: 120.05k
Cutoff Freq: 6.0025k
Transition Width: 6.0025k
Window: Hamming
Beta: 6.76

Low Pass Filter
Id: low_pass_filter_0
Decimation: 1
Gain: 1
Sample Rate: 120.05k
Cutoff Freq: 60.025k
Transition Width: 60.025k
Window: Hamming
Beta: 6.76

UHD: USRP Source
Id: uhd_usrp_source_0
Sync: Unknown PPS
Samp rate (Sps): 120.05k
Ch0: Center Freq (Hz): 86M
Ch0: AGC: Default
Ch0: Gain Value: 60
Ch0: Gain Type: Absolute (dB)
Ch0: Antenna: TX/RX

ZMQ SUB Source
Id: zeromq_sub_source_0
Address: tcp://127.0.0.1:5555
Timeout (msec): 100
Pass Tags: No

Throttle
Id: blocks_throttle_0
Sample Rate: 120.05k

Channel Model
Id: channels_channel_model_0
Noise Voltage: 200m
Frequency Offset: 10k
Epsilon: 1
Taps: 1+1j
Seed: 0
Block Tag Propagation: No

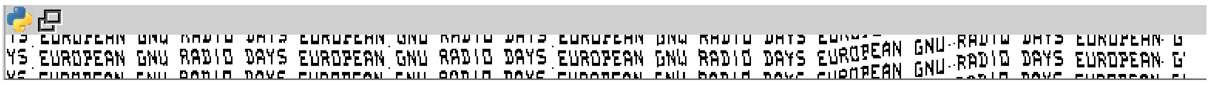
Hellschreiber receptor
noise 0.01800
df 0

Generating: '/home/jmfriedt/sdr/gr-hellschreiber/hell_decode.py'

EUROPEAN GNU RADIO DAYS EUROPEAN GNU RADIO DAYS EUROPEAN GNU RADIO DAYS EUROPEAN GNU RADIO DAYS EUROPEAN GNU RADIO DAYS EUROPEA
in
out
error
T_inst
freq
bw
Amplitude
Signal 1
2
1.5
1
0.5
0
-0.5
Imports
1a
1b
1c
2
3
4a
4b
4c
5
6

Figure 3: Hellschreiber receiver chain, here emphasizing the numerous signal sources that could feed this processing flowchart (RTL-SDR hardware dongle, USRP SDR hardware), here emphasizing on a virtual source as a ZeroMQ Subscribe socket (1a) allowing to interface a channel simulator (1c) to assess the impact of noise or frequency offset between emitter and receiver. As in the case of the emitter (Fig. 2), the lack of hardware bandwidth limiter requires the use of a Throttle block (1b). The output of this source feeds low pass filters to reduce the datarate and get rid of unnecessary parts of the sampled spectrum, before feeding a Symbol Synchronizer and finally a ZeroMQ output socket streaming the bits forming the image (5). On top, the Hellschreiber Python decoding script collects the symbols and displays the image of the transmitted message (6). This exact flowgraph would be executed on the embedded single board computer Raspberry Pi by removing all references to graphical interfaces (4a, 4b and 4c) while the decoding script would still be executed on the host personal computer fitted with a display since ZeroMQ will stream over a network flawlessly.

streaming the processed output to external tools (Hellschreiber display). The object oriented structure of GNU Radio clearly differentiates the constructor for initializing the variables, from the work function called to process each new datablock provided by the scheduler. Once this architecture has been mastered,



```
YS EUROPEAN GNU RADIO DAYS EUROPEAN GNU RADIO DAYS EUROPEAN GNU RADIO DAYS EUROPEAN GNU RADIO DAYS EUROPEAN G
```

Figure 4: Simulating the channel allows for tuning parameters and observing their impact: here the Clock Synchronizer bandwidth was on purpose lowered so that the feedback loop would be slow enough that sampling rate convergence is observed with the tilt of the words being displayed.

adding custom functionalities is a matter of implementing the discrete time digital processing, with GNU Radio taking care of the scheduling overhead.

4 Acknowledgements

Thomas Lavarenne has developed the `gr-hellschreiber` encoder and decoder, providing an opportunity to demonstrate a use of `gr-rpitx` to the ham radio community. All software is available at `https://github.com/tlavarenne/gr-hellschreiber` and `https://github.com/jmfriedt/gr-rpitx` respectively, with the accompanying video on the European GNU Radio Days conference YouTube channel at `https://www.youtube.com/watch?v=Yb5VEgKdnAk`.

References

[1] O. Mattos and O. Weigl, *Turning the Raspberry Pi Into an FM Transmitter* (2012) at `http://www.icrobotics.co.uk/wiki/index.php/Turning_the_Raspberry_Pi_Into_an_FM_Transmitter`, last accessed Aug. 2021

[2] J.-M. Friedt, É. Courjaud, *gr-rpitx: GNU Radio compatible general purpose SDR emitter using the Raspberry Pi(4) internal phase locked loop*, European GNU Radio Days (2021), video at `https://www.youtube.com/watch?v=bULEkxeRK9U`, proceeding at `https://pubs.gnuradio.org/index.php/grcon/article/view/107/88`

[3] F. Dörenberg, *All about Hellschreiber* at `https://www.nonstopsystems.com/radio/hellschreiber.htm` (accessed Aug. 2021)

[4] J.H. Tiltman, *The "Tunny" Machine and Its Solution*, NSA declassified 2007, at `https://www.nsa.gov/Portals/70/documents/news-features/declassified-documents/tech-journals/tunny-machine.pdf`

[5] T. Lavarenne, `gr-hellschreibed` at `https://github.com/tlavarenne/gr-hellschreiber` (accessed Aug. 2021)