Semester Project

Implementation of a packet encoder/decoder pair in the GNU Radio framework

Spring 2017

Author:
Thomas Verelst

Supervisor:
Dr. P. Giard

Assistant:
O. Afisiadis

Telecommunication Circuits Laboratory
Ecole Polytechnique Fédérale de Lausanne

EPFL-STI-IEL-TCL

Contents

1 Introduction

In this student project, a wireless communication system is implemented in the GNU Radio framework. GNU Radio is a development toolkit that provides signal processing blocks for software-defined radios and signal processing systems. These blocks are connected in flowgraphs, which can be built visually in the GNU Radio Companion or by coding in Python. An example of a wireless communication chain is given in Figure 1.

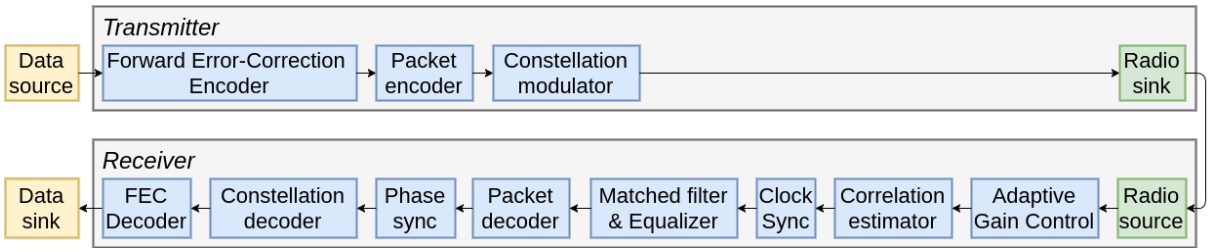


Figure 1: Communication chain

We developed several new blocks to integrate functionality currently not possible in GNU Radio. They will be discussed throughout the report. Extra attention is paid to the packet encoder and decoder in the communication chain. A packet encoder encapsulates the incoming stream into packets, consisting of a preamble, header (containing packet length), payload and checksum. In particular, a flexible packet encoder is implemented, supporting different constellations for preamble, header and payload. The new packet encoder/decoder pair also supports soft-decision error correction and varying packet lengths.

A small introduction to GNU Radio is provided. The first part of the report focuses on the packet encoder/decoder. The existing packet encoder/decoder pair is evaluated and new, improved blocks are implemented. The second part integrates the new packet encoder/decoder pair in a full-featured wireless communication chain. Problems with the existing frame synchronizer are examined and the improved implementation is discussed. At the end of the report, GNU Radio challenges and limitations are discussed as well as some lessons learned.

The examples and developed blocks are combined in a GNU Radio module called *packetizer*, located at `https://tclgit.epfl.ch/semester-projects/17S-Verelst-GNURadio`.

2 GNU Radio

2.1 Flowgraphs

GNU Radio implements a dataflow system, where blocks are used to process data. Blocks receive several samples at once in their input buffers, process them and place the result in their output buffer. There are two important ways to pass data from one block to another: streams and message passing.

Streams

Streams are the most common way to transmit signal data between blocks. A stream is an endless sequence of samples which have a certain data type. Data types can be distinguished by color, as illustrated in Figure 2.

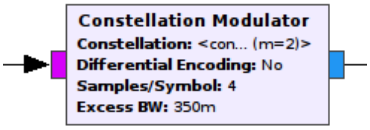
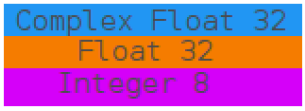


Figure 2: A GNU Radio block that gets a byte stream as input and outputs complex symbols

Many data types exist in GNU Radio, but only a few are used in practice. Complex samples are blue, floats are orange and bytes are purple.



Tagged streams

Tagged data streams are an extension of data streams. A tag can be attached to a stream sample, and provides extra control information. Tags are useful to transmit indicators, for example the start and length of a packet.

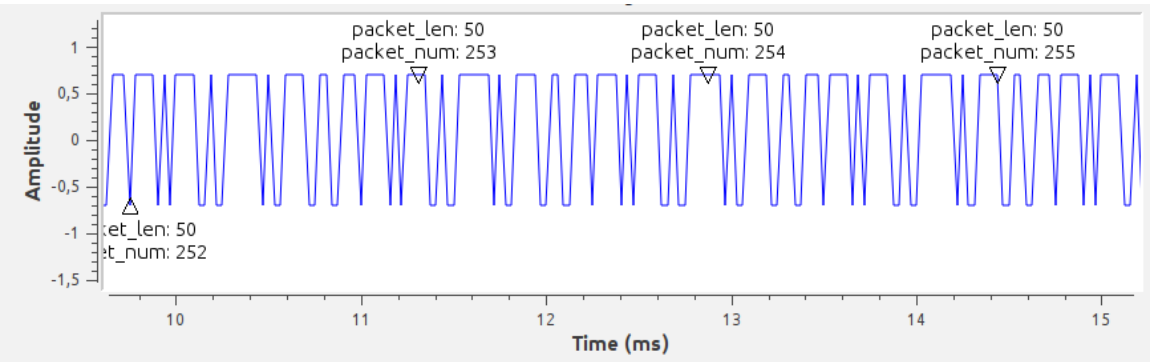


Figure 3: Stream tags to indicate the packet start, length and sequence number

Message passing

Message passing is a more recent feature, and particularly useful to transmit control and meta data. In contrast to data streams, message passing blocks can be connected to upstream blocks in order to create feedback loops. A message connection is indicated with a dotted line, instead of a solid line.

2.2 Block development in GNU Radio

Users can build new Out-of-Tree modules containing new blocks to add extra functionality to the framework. These blocks are developed in C++ or Python, where C++ focuses on performance and Python on ease of development. A special type of block is a *hier* block, which is a block that internally connects several existing blocks. They can be used to quickly turn existing systems into a separate entity which can be integrated in larger systems.

We implemented several blocks in this semester project. Technical software development details are not given since block development relies heavily on the GNU Radio API, which can be found online [1].

3 Packetization

3.1 Current situation in GNU Radio

3.1.1 GNU Radio 3.7.9

Packet encoder and decoder blocks with limited functionality exist in GNU Radio. The *Packet Encoder* and *Packet Decoder* are deprecated blocks that have the following functionality:

- Create packets with a preamble, acces code, header and a payload. The header is a double repetition of the payload length (16 bits for each field).
- Constellations: GMSK, DBPSK, DQPSK, D8PSK, QAM8, QAM16, QAM64, QAM256
- Modulation and pulse shaping with a given number of samples per symbol.

The decoder looks for the access code and calculates the number of wrong bits. When this number is under the given threshold, it reads the header to find the payload length. Finally, it outputs the payload as a byte or float stream.

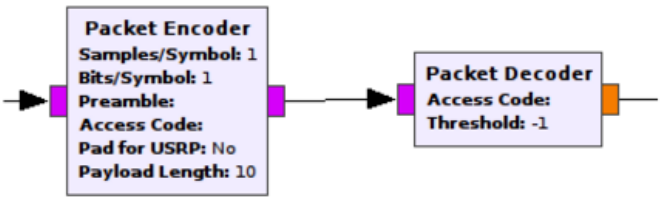


Figure 4: Deprecated packet encoder and decoder blocks

The implementation has several limitations:

- No support for different constellations for preamble, header and payload
- No proper support for custom header fields. The header is built as a two times the payload length.
- Lacks flexibility: fixed payload length, pulse shaping not configurable
- Not enough documentation and the source code is difficult to understand. For example, the output format is not specified.

3.1.2 GNU Radio 3.7.10

New packet examples are included in this more recent version of GNU Radio. As explained in section 5.2, installing GNU Radio using the recommend way does not guarantee the latest version of GNU Radio. The existence of these packet-related example systems had been discovered late in the project and the project is developed independently.

3.2 Extended Packet Encoder

3.2.1 Requirements

The new packet encoder implementation will limit its functionality to packetizing the incoming data stream with a preamble, header and payload and mapping these to constellation symbols. It will not do any pulse shaping or resampling.

The new packet encoder supports the following features:

- GNU Radio's constellation objects in order to support a wide range of PSK and QAM mappings. Optional support for differential encoding.
- Possibility to use distinct constellation types for preamble, header and payload.
- GNU Radio's header formatter objects in order to support headers with custom lengths and fields.
- The start of the payload data for a packet is indicated in the incoming byte stream with a tag that has the payload length as tag value.
- Optional support for data whitening, as discussed in section 4.4.

Inputs	Outputs
Byte stream: 1 bit/byte A tag should indicate the packet start and length	Complex: mapped symbols

Parameter name	Data type	Description
Preamble	complex vector	sequence of complex symbols for the preamble
Header Constellation	constellation object	pointer to the constellation object for the header
Header differential mapping	boolean	use differential encoding for header data
Payload Constellation	constellation object	pointer to the constellation object for the payload
Payload differential mapping	boolean	use differential encoding for payload data
Header Formatter	header formatter object	pointer to the header formatter object
Length Tag name	string	name of the tag that indicates the packet start and has the packet length as tag value
Zero padding	integer	number of zero symbols $(0+0*j)$ between each packet
Whiten	boolean	indicates whether the incoming data stream should be whitened, as discussed in section 4.4.

Table 1: Interface of the new Extended Packet Encoder block

3.2.2 Implementation in GNU Radio Companion

A prototype system is built in GNU Radio Companion to explore the possibilities of existing blocks. Figure 6 shows the schematic representation of the dataflow and Figure 7 is the GNU Radio implementation with existing blocks. Two main streams can be distinguished:

- The payload stream is MSB-first repacked to N bits per byte, where N depends on the constellation order. Each of these bytes is mapped to a symbol. More information about MSB-first and LSB-first repacking can be found in appendix A.
- The header stream is generated by the Packet Header Generator block, which accepts a header formatter object as input. A header formatter object specifies the bit format of the header. The default GNU Radio header for digital transmissions is the *packet_header_default* [2] which specifies specifies a 32-bit header as follows:
  - bits 0 - 11: Packet length (LSB first). In this project, the packet length will be defined as the number of payload bits per packet. The maximum payload length value with the default header is $2^{12} - 1 = 4095$ *bits*.
  - bits 12 - 23: Packet sequence number (auto-incrementing) (LSB first)
  - bits 24 - 31: Cyclic redundancy check (CRC) for packet header

  An example illustrates the header format in Figure 5.

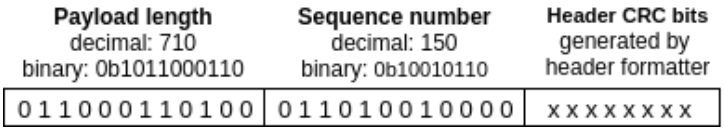


Figure 5: Example header, for a payload length of 710 bits and a packet sequence number 150.

The header format is specified in the GNU Radio Companion by adding a variable block with the following value:

```
digital.packet_header_default(32/constel_header.bits_per_symbol(),
 ↪  "packet_len", "packet_num", constel_header.bits_per_symbol())
```

The first argument is the length of the header in symbols. In this example, *constel_header* is the constellation object that defines the header constellation. The second argument is the name of the tag carrying the payload length. The third parameter defines which tag name the packet sequence number should get when decoding the header. The last parameter is the number of significant bits per byte at the output. A correct configuration is very important since the Extended Packet Encoder block does not repack the bytes before mapping the header bytes to symbols.

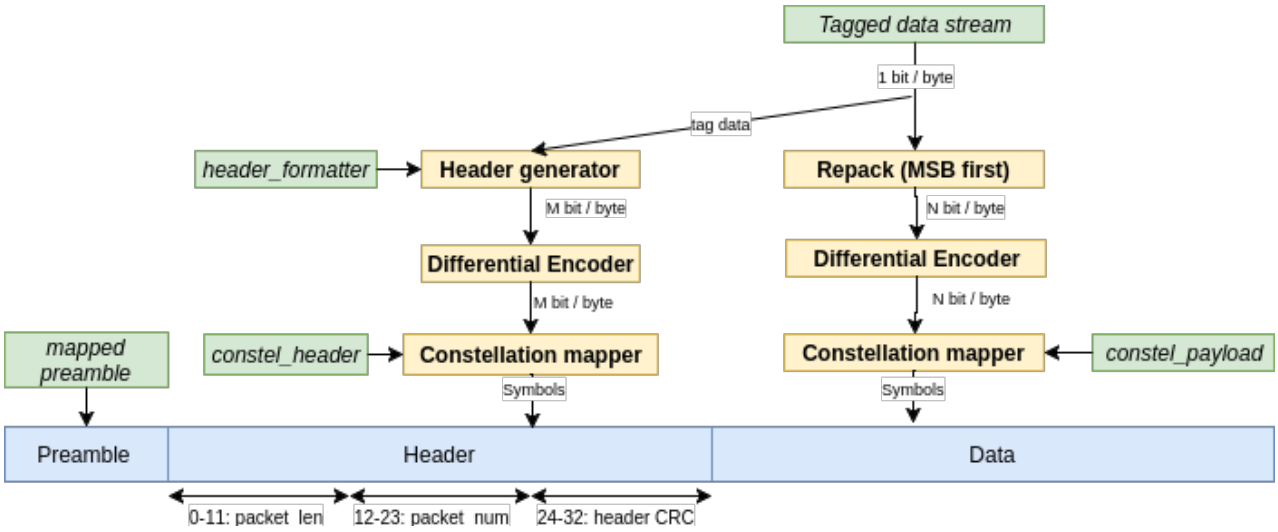


Figure 6: Schematic overview of the packet encoder implementation

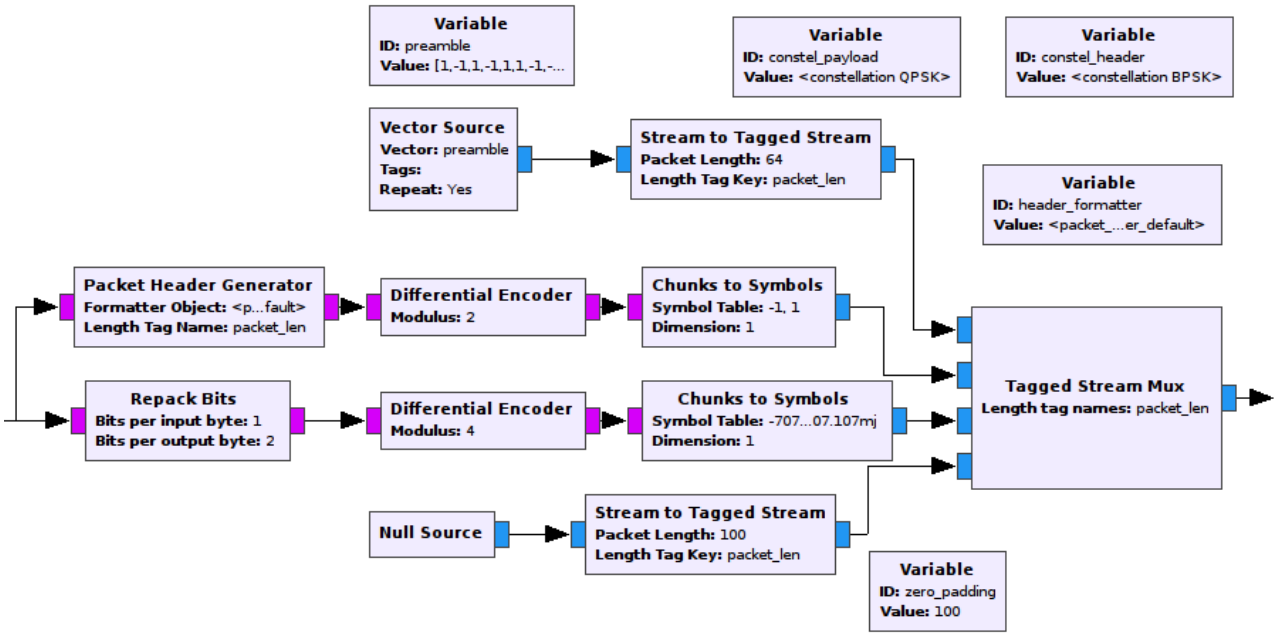


Figure 7: GNU Radio Companion implementation of a packet encoder with differential encoding. Data input on the left has one bit per byte, data output on the right is one complex symbol/sample

3.2.3 Block implementation

As described in section 2.3, a GNU Radio block can be built in C++ and Python or as a *hier* block that links several existing blocks. In the packet encoder implementation as described above, many blocks do trivial operations that can be efficiently combined to eliminate the overhead of transferring data between blocks. A C++ block is the most appropriate choice.

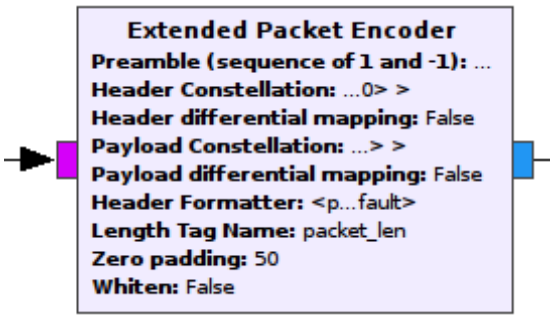


Figure 8: Extended Packet Encoder in the GNU Radio Companion

3.2.4 Data example

A simple test bench is created in order to verify the output of the implemented block. The block processes a predefined data sequence.

- Preamble (mapped symbols): 1, -1, 1, 1
- Payload (1 bit/byte): 0,0,0,1,0,0,1,1, 1,1,1,1,1,1,1,0, 1,0,0,0,0,1,1,0, 0,0,1,0,1,1,1,1
- Header mapping: binary phase-shift keying (BPSK) $[0, 1] \rightarrow [1, -1]$
- Header Formatter: generated by the *packet_header_default* class with a length of 32 bits.
- Payload mapping: quadrature phase-shift keying (QPSK)
  $[0, 1, 2, 3] \rightarrow [-1 - 1j, 1 - 1j, -1 + 1j, 1 + 1j]$

Output:

```
1,-1,-1,1,
(1+0j), (1+0j), (1+0j), (1+0j), (-1+0j), (1+0j), (1+0j), (1+0j),
(1+0j), (1+0j), (1+0j), (1+0j), (1+0j), (1+0j), (1+0j), (1+0j),
(1+0j), (1+0j), (1+0j), (1+0j), (1+0j), (1+0j), (1+0j), (1+0j),
(1+0j), (-1+0j), (-1+0j), (1+0j), (-1+0j), (-1+0j), (1+0j), (-1+0j),
(-0.707-0.707j), (0.707-0.707j), (-0.707-0.707j), (0.707+0.707j),
(0.707+0.707j),  (0.707+0.707j), (0.707+0.707j), (-0.707+0.707j),
(-0.707+0.707j), (-0.707-0.707j), (0.707-0.707j), (-0.707+0.707j),
(-0.707-0.707j), (-0.707+0.707j), (0.707+0.707j), (0.707+0.707j))
```

The first 4 symbols are the preamble. The next 32 symbols mapped in BPSK have been generated for the header. When demapped to bits, this gives:

```
00000100 00000000 00000000 01101101
```

The payload length in bits is stored in the first 12 bits in an LSB-first way. The decimal result is 32, which is correct. The 12 following bits encode the packet sequence number, 0 in this case. The last 8 bits are a CRC for the header.

The QPSK payload symbols demap to:

```
 00 01 00 11  | 11 11 11 10 | 10 00 01 10 | 00 10 11 11
```

Which is the same as the input sequence.

3.3 Extended Packet Decoder

3.3.1 Requirements

The packet decoder does the inverse operation of the packet encoder. It will demap incoming symbols to a data stream. The decoder does not know the packet length, so it should decode the header and retrieve the header length from the corresponding field.

The Extended Packet Decoder block should support the following functionality:

- Decoding of packets that are encoded with the Extended Packet Encoder block
- Support for both hard and soft outputs, in order to support forward error correction
- Support for differential decoding, when hard-decision decoding is used

Inputs	Outputs
Complex: 1 symbol/sample A tag on the incoming stream indicates the packet start	Float: soft demapped output, samples between -1 and 1
	Byte: hard demapped output with 1 bit per byte
	Message port: message containing header fields

Parameter name	Data type	Description
Preamble	complex vector	sequence of complex symbols for the preamble
Header Constellation	constellation object	pointer to the constellation object for the header
Header differential mapping	boolean	use differential decoding for header data
Payload Constellation	constellation object	pointer to the constellation object for the payload
Payload differential mapping	boolean	use differential decoding for payload data
Header Formatter	header formatter	pointer to the header formatter object for the header
Trigger Tag name	string	name of the tag that indicates the packet start and has the packet length as tag value
Apply Costas Loop	boolean	indicates whether phase synchronization with Costas Loop should be applied.
Dewhiten	boolean	indicates whether the incoming data stream should be dewhitened.

Table 2: Interface of Extended Packet Decoder block

3.3.2 Implementation in GNU Radio Companion

A decoder system built in GNU Radio is shown in Figure 9. Figure 10 shows a packet decoder with differential decoding. The packet decoder is a complex block that combines several existing solutions:

- Preamble/Header/Payload Demux: splits the input stream into a header and payload stream. In order to know the payload length, it uses the feedback loop with the decoded header data.
- Constellation Soft Decoder: demaps symbols to soft bits.
- Binary Slicer: maps soft bits (-1 and 1) to hard bits (0 and 1).
- Tagged Stream Fix: drops unnecessary samples so packets are sequential in the stream.
- Differential Decoder: removes the differential encoding, done in the packet encoder.
- Packet Header Parser: decodes a header byte stream according to the header format given by the Header Formatter, which is the same one as given to the packet encoder.
- Costas Loop: applies carrier tracking for phase synchronization, as described in section 4.3.

Preamble/Header/Payload Demux block

The most important block in the decoder system is the Preamble/Header/Payload Demux (PHP Demux) block. It is an extension of the existing Header/Payload Demux block, which is a highly flexible block to extract payload data with a variable length from a packet, by looking at the header fields. The block is implemented as a state machine, and one state had to be added to support preambles. The full state diagram of the PHP Demux block is shown in Figure 11. The payload data length is given using the feedback loop for header data. Header data gets decoded and the Packet Header Parser will read the fields from the header. The result is outputted with a GNU Radio message that is forwarded to the PHP Demux block.

The original Header/Payload Demux block only accepts the payload length as a number of symbols. Since the block does not know the modulation type and order, it can only split the incoming streams into a header symbol stream and payload symbol stream.

For this project, we added a new parameter called "Header Length value divider". The payload length value, received from the header, will be divided by this number and the result will be ceiled. This modification makes it possible to specify the payload length in bits, while the PHP Demux counts symbols.

For example, when having a payload length of 50 bits and 8 phase-shift keying (8PSK), the packet encoder generates one zero-bit to fill all 17 symbols. The packet header carries the value 50 as *packet_len* field. The PHP Demux block is configured with a Header Length value divider of 3, which is the number of bits per symbol. It will calculate $ceil(50/3) = 17$ and output 17 symbols on the payload stream. The first output sample will get a tag with the name *packet_len* and value 50.

The Constellation Soft Decoder decodes each symbol to soft bits. In the case of 8PSK, it outputs 3 soft bits per symbol. As a result, the output stream consists of 17 $symbols \times 3\ softbits/symbol = 51\ softbits$ which is not what the *packet_len* tag indicates. The last

bit is a zero bit that is added by the packet encoder to fill the last symbol, and it should be removed from the output. The Tagged Stream Fix block will do this, as explained in the next paragraph

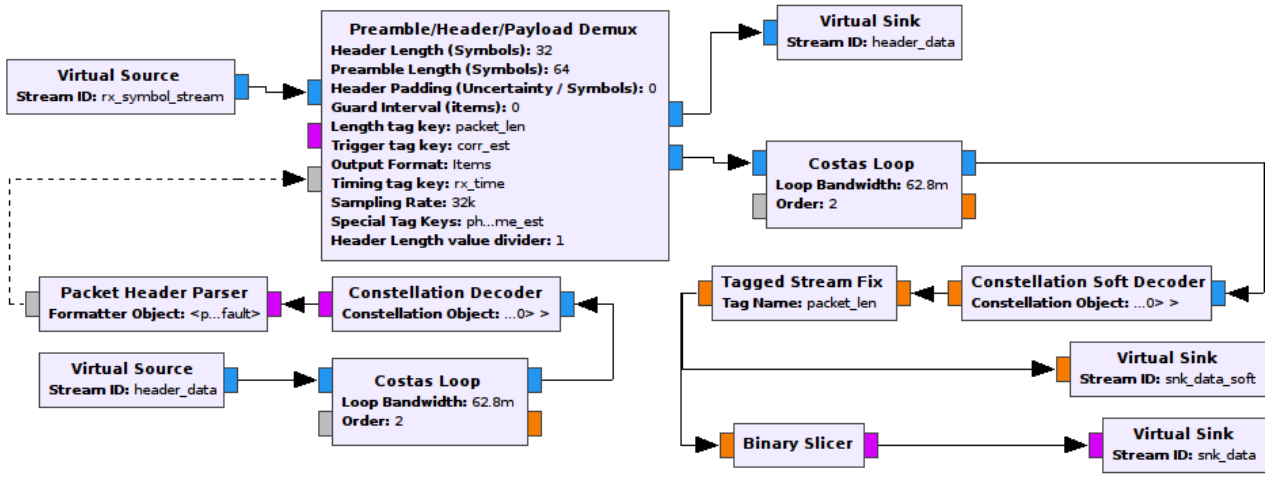


Figure 9: A packet decoder in GNU Radio Companion, configured for a BPSK header and QPSK payload

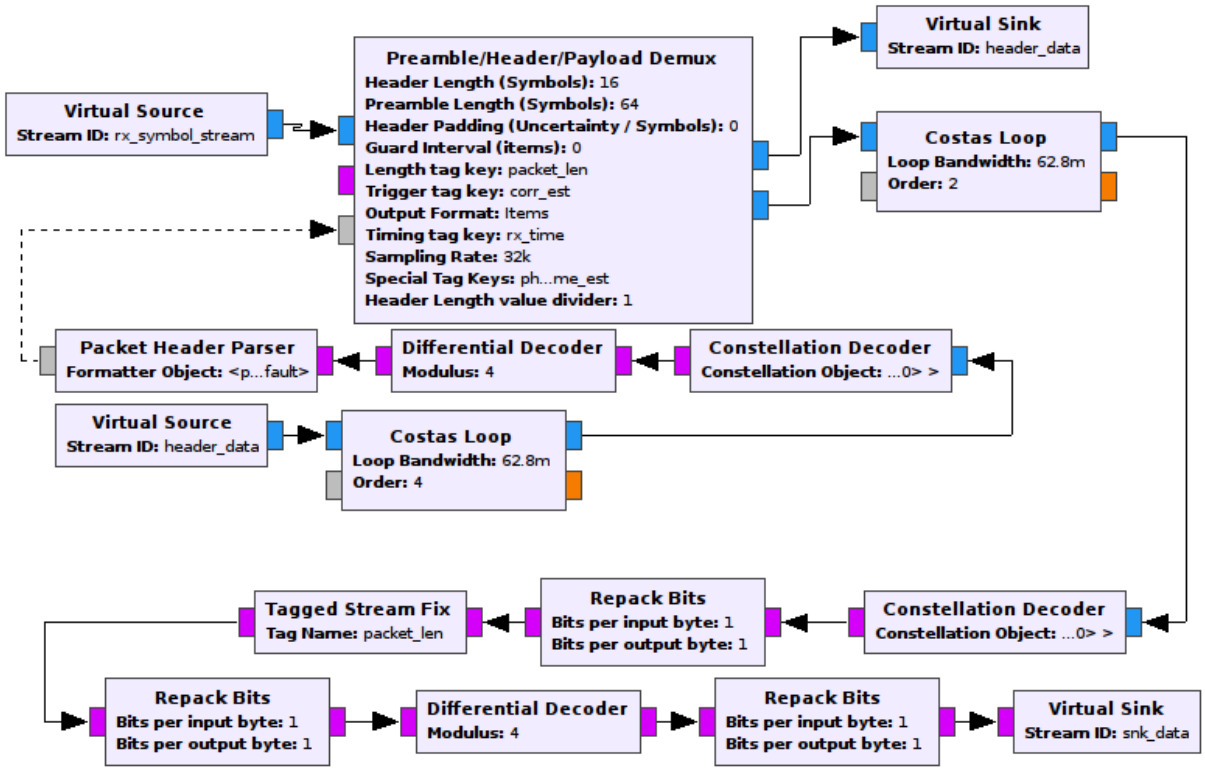


Figure 10: A packet decoder in GNU Radio Companion with differential decoding, configured for a BPSK header and QPSK payload

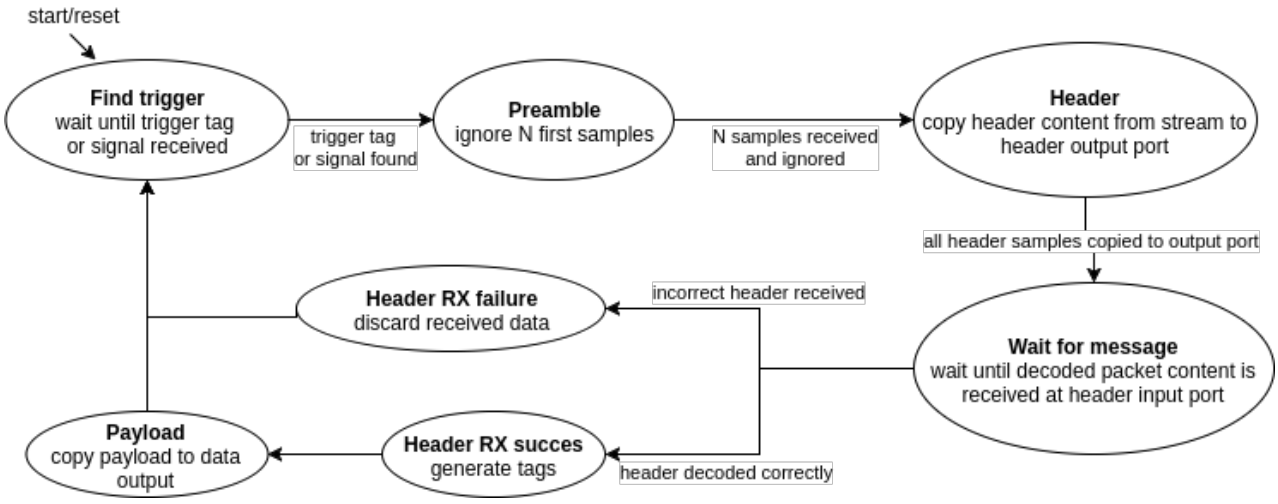


Figure 11: State diagram of the Preamble/Header/Payload block

Tagged Stream Fix block

A tagged stream with more samples between tags than what the *packet_len* tag indicates gives problems in GNU Radio's tagged stream blocks, such as the Extended Tagged FEC Decoder. These blocks expect that the packets are perfectly sequential, i.e. there are no extra samples between packets. Surprisingly, no GNU Radio block exists to remove the unnecessary samples between packets.

In this project, a block called *Tagged Stream Fix* has been developed to 'truncate' a tagged data stream. It only keeps the samples belonging to a packet, and removes the other samples. This is illustrated in Figure 13. Because of time constraints during the project, the block is written in Python, even though a C++ block provides more throughput with less CPU usage.

Figure 12: Tagged Stream Fix block in GNU Radio Companion

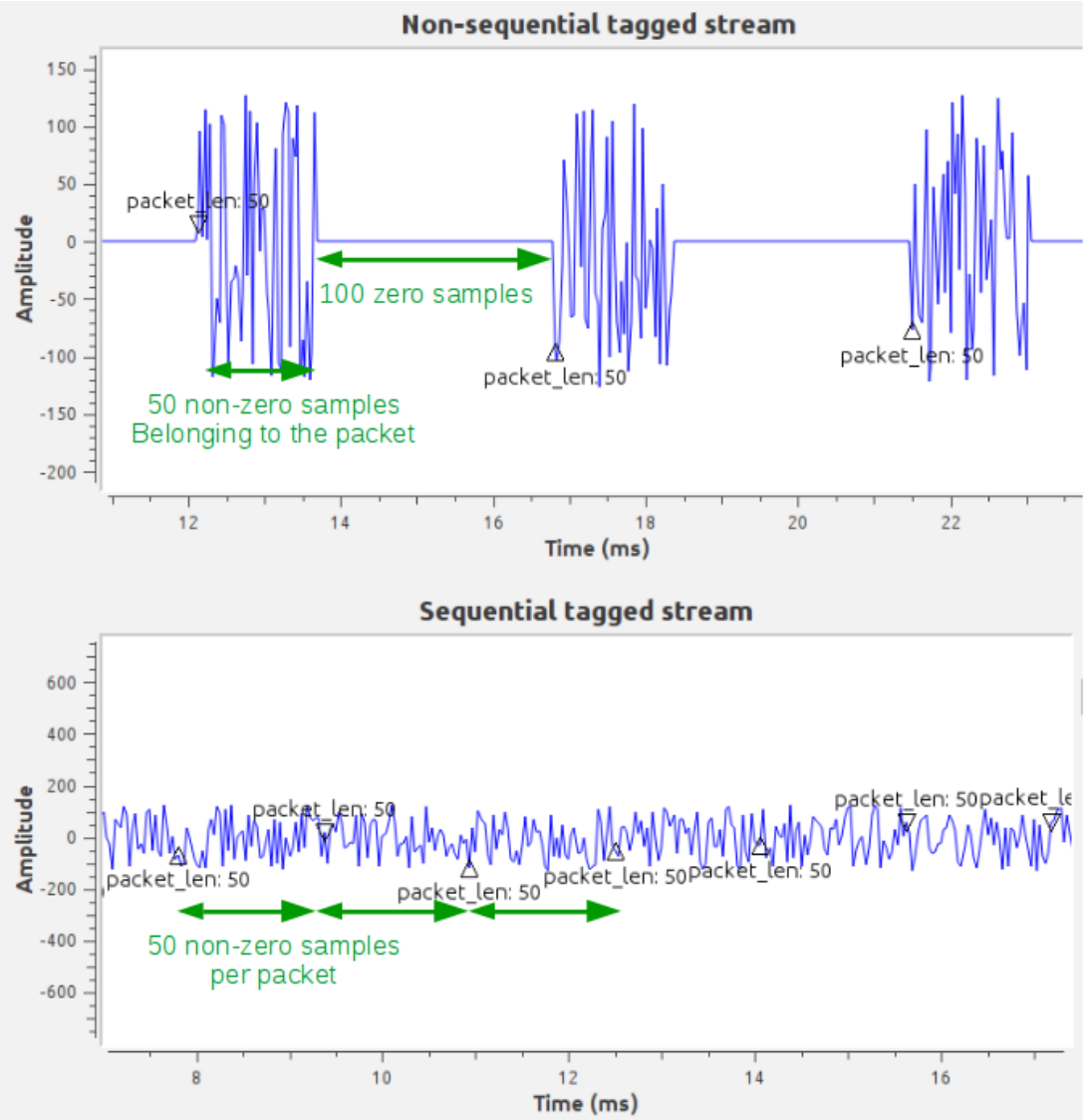


Figure 13: Illustration of a non-sequential packet stream and a sequential packet stream. In the top plot, the packets are separated with zero samples, in the bottom plot the zero samples are removed.

3.3.3 Block implementation

The Extended Packet Decoder block is a combination of several existing blocks, and a *hier* implementation in Python is straightforward. The new block is called Extended Packet Decoder.

The header data can be analyzed by using the message output port on the Extended Packet Decoder. We developed a new block called Message Sequence Checker to verify correct packet communication. It reads the *packet_num* field from the header and checks for discontinuities in packet sequence numbers. When two the difference between two sequence numbers is more than one, some packets have been dropped because they could not be detected or decoded. The block will output the number of lost packets and the packet loss rate in the console.

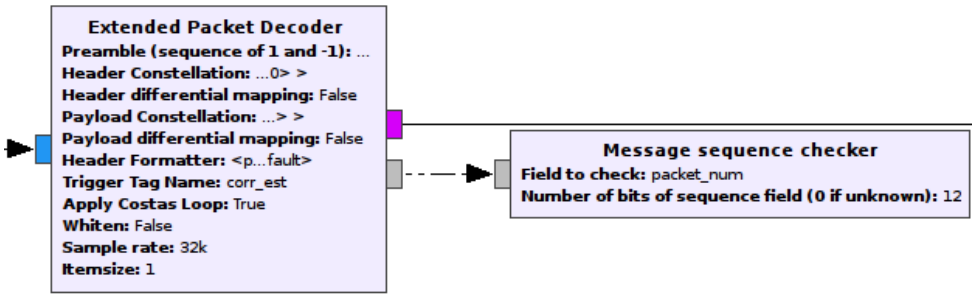


Figure 14: Extended Packet Encoder block and Message Sequence Checker blocks

3.4 Packetization example systems

Mapping

test_mapping.grc is a basic example demonstrating symbol mapping and decoding. Differential encoding/decoding is also implemented.

Soft decoding

test_soft_decoder.grc illustrates the use of the Constellation Decoder block and the Constellations Soft Decoder block. An important thing to notice is that the bits should be repacked by MSB-first (instead of the default LSB-first), before mapping the bytes to symbols. The constellation receiver outputs the decoded bits MSB-first.

Prototype encoder/decoder

encdec_basic.grc implements the full-featured packet encoder and decoder using basic GNU Radio blocks.

Packet encoder/decoder blocks

encdec_custom.grc uses the Extended Packet Encoder/Decoder blocks, with both hard and soft demapping.

4 Communication chain

The communication chain is the set of elements that transforms data into a transmittable waveform. It consists of data pre-processors, modulation units and several recovery units for time, phase and frequency synchronization. An overview of the implemented communication chain is given in Figure 15

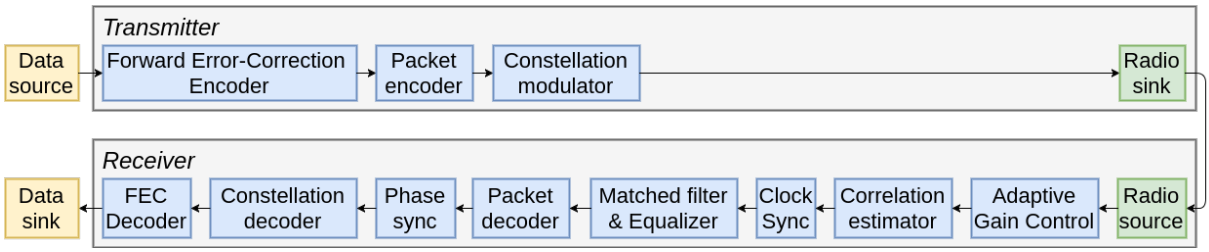


Figure 15: Communication chain overview

4.1 Pulse shaping

The implemented packet encoder outputs constellation symbols. These symbols are processed by an upsampler and pulse shaper, in order to limit the bandwidth when transmitting. A common filter is a root raised cosine. At the receiver side, the signal can be convoluted with the matched filter so that the transmission is free of intersymbol interference. The spectrum of a transmissed stream, shaped with a root raised cosine filter, is shown in Figure 16.

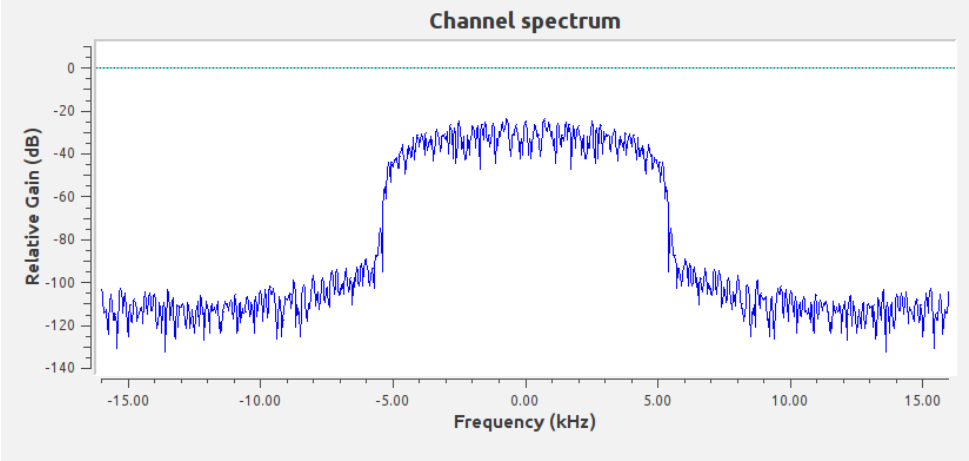


Figure 16: Channel spectrum at baseband, RRC-filter with roll-off factor 0.35 and stopband attenuation of 100 dB

4.2 Frame synchronization

The frame synchronization unit detects the start of packets. A common way is to prepend a preamble or access code sequence to the packet content so the receiver side can look for the preamble. After correlating the stream at the receiver with a modulated reference preamble, the packet start can be indicated by detecting correlation peaks.

4.2.1 GNU Radio situation

A large number of blocks for synchronization are included in the default GNU Radio installation. The most notable blocks are the *Correlation Estimator*, *Correlate Access Code* and *Correlate and Sync*, where the latter is deprecated since GNU Radio 3.7.10.

The Correlation Estimator block is the most advanced and updated implementation. The block correlates a given modulated preamble with the incoming stream, and indicates correlation peaks with a tag called *"corr_est"*.

The block has a parameter *Tag marking delay* which is used to shift the tag position compared to the incoming data stream. This shifting functionality is required because of two reasons. The first reason is that the convolution, used to pulse shape the reference preamble, introduces extra samples at the beginning and end of the preamble sequence because of the edge effect. When calculating the correlation with the incoming data stream, the tag will be placed at a fixed offset of the real position. The second reason is that some blocks in the communication chain do not preserve the correct tags positions, compared to the samples. The tag delay parameter should be set manually by looking at the output scopes.

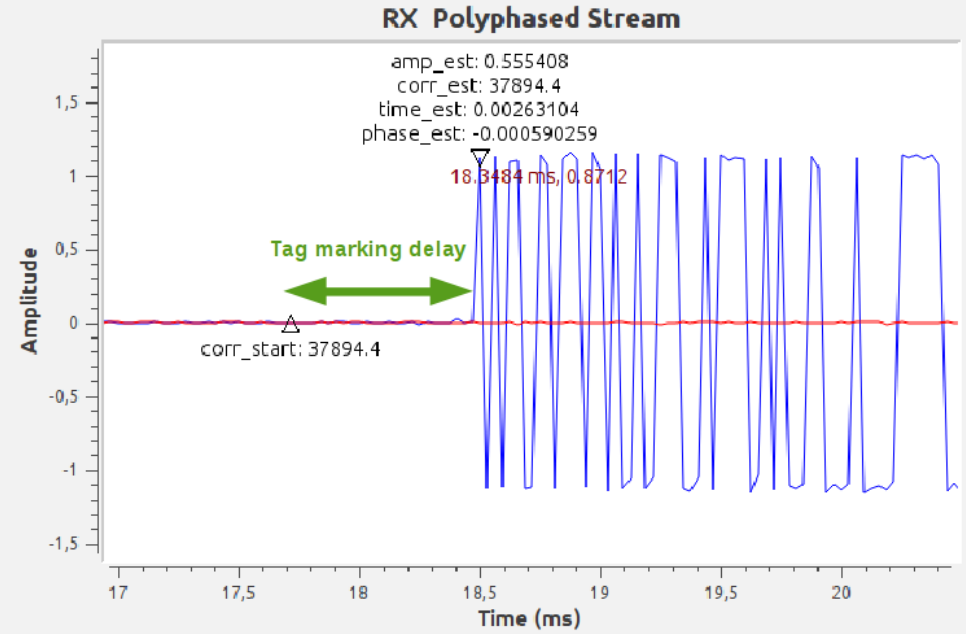


Figure 17: The *corr_start* tag indicates where the Correlation Estimator block detected the preamble. The other tags are shifted by the number of samples that is given in the Tag Marking Delay field.

The *gr-digital* package includes several synchronization examples, which are unfortunately broken in GNU Radio 3.7.9 and 3.7.10. The *test_corr_est.grc* example has a threshold that is far too small, which causes a large amount of false-positives. In addition, the "Tag marking delay" is wrong. Many GNU Radio users have reported these issues [3].

4.2.2 Pulse shaping the reference preamble

The Correlation Estimator block expects a modulated and shaped preamble as input argument.

GNU Radio has a complementary block to modulate and pulse shape a byte stream, called Modulate Vector. In this project however, the preamble is defined as a pre-mapped sequence of complex symbols. To pulse shape the mapped vector, a new block was implemented in Python. The block is called Pulse Shape Vector and will pulse shape a given vector with a given filter.

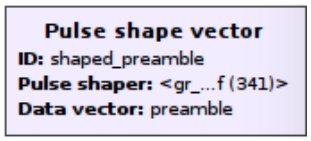


Figure 18: Pulse Shape Vector block in GNU Radio Companion

The block accepts three parameters:

- ID: the variable name for the shaped preamble (output of the block)
- Data vector: the variable name of the mapped preamble
- Pulse shaper: definition of a filter in GNU Radio. When using a polyphase filterbank with root raised cosine taps, the filter is defined is as follows:

```
filter.pfb_arb_resampler_ccf(
 sps,
 firdes.root_raised_cosine(nfilts, nfilts, 1.0, eb, 11*sps*nfilts),
 32
)
```

  The meaning of the arguments can be found in the GNU Radio manual for the polyphase filterbank [4] and root raised cosine [5].

4.2.3 Problems with the Correlation Estimator block

The existing correlation estimator implementation was not sufficient to achieve a reliable packet detection. Three problems were identified:

1. The threshold cannot be set high enough to avoid false detections when having a high input power. When the threshold is set to 0.99999998 or higher, the system execution stops without returning any error.

2. False detections occur when there is no incoming signal except noise.

3. Especially at the end of a packet, several false detections occur even when having a high signal power.

These problems are illustrated in Figure 19.

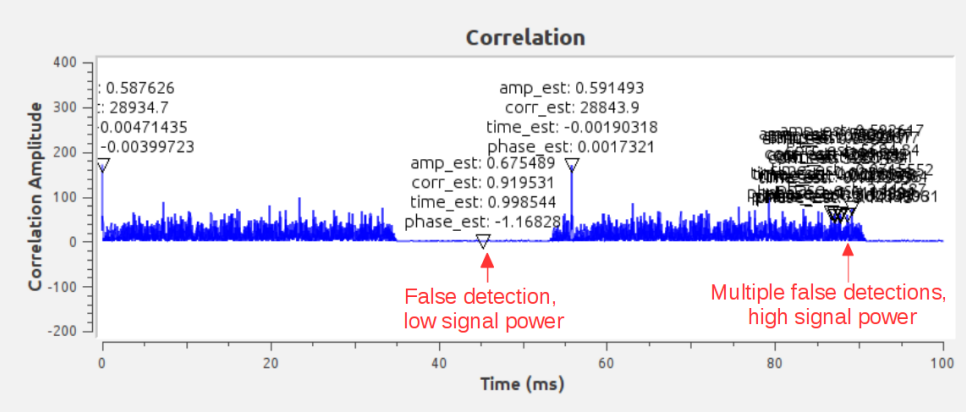


Figure 19: Correlation Estimator false detections

Internal working of the existing implementation

In order to detect a correlation peak, the incoming signal is correlated with the modulated reference preamble. The squared magnitude of each correlation sample is calculated and if the squared correlation magnitude of a sample is larger than the *detection threshold*, the sample is marked with a tag. Since the correlation magnitude depends on the incoming signal level, adaptive gain control should be used to normalize the signal power.

The detection method can be expressed in formulas as follows. $Incoming\_signal$ is the incoming symbol sequence of length N and $mod\_preamble$ is the modulated preamble symbol sequence. The exact way to determine the $detection\_threshold$ is explained in the next paragraph.

$$corr = correlate(incoming\_signal, mod\_preamble)$$
$$corr\_mag\_sq[i] = corr[i]^2 \quad \forall i = 0 \; to \; N-1$$
$$detection[i] = corr\_mag\_sq[i] > detection\_threshold \quad \forall i = 0 \; to \; N-1$$

Up to GNU Radio 3.7.9, the Correlation Estimator block used a fixed detection threshold. The $detection\_threshold$ value is relative to the power of the discrete autocorrelation of the modulated reference preamble. The value is constant and does not depend on the incoming signal. The parameter *threshold* can be defined by the user:

$$preamble\_autocorr = \sum_i |mod\_preamble[i] * mod\_preamble[i]^*|$$
$$detection\_threshold = threshold * preamble\_autocorr^2;$$

In GNU Radio 3.7.10, the fixed detection threshold was changed to an adaptive threshold, to avoid the need for adaptive gain control. That implementation uses a constant false alarm rate (CFAR) detection method. The given *threshold* parameter is converted to a probability of false alarm (pfa):

$$d\_pfa = -\log(1 - threshold)$$

The final detection level is a multiplication of the average correlation magnitude of surrounding samples and the probability of false alarm:

$$detection\_threshold = 4 * d\_pfa * mean(corr\_mag\_sq\ of\ surrounding\ samples)$$

The decision algorithm is also slightly modified. A sample is indicated as a packet start if the sum of the squared correlation magnitude of this sample i and the next sample *i+1* is greater than the detection threshold. The sum is introduced to counter the effect of a time offset, which can spread the correlation peak over two samples.

$$detection = (corr\_mag[i] + corr\_mag[i+1]) > detection\_threshold$$

4.2.4 Improved Correlation Estimator

We modified the Correlation Estimator, and included a new block called Correlation Estimator 2 in the *packetizer* module. This new block solves the problems described in the previous section, by doing several modifications in the C++ code.

Solving problem 1: Increasing the maximum threshold

The internal representation of the *threshold* parameter variable is a float. The limited precision of a float causes numerical problems when calculating $d\_pfa = -\log(1 - threshold)$. In fact, the closest float to 1 is 0.999999880791. Anything closer to 1 will be rounded to 1 and that gives the logarithm of 0, which is impossible to compute.

By changing the internal threshold representation from float to double, a much larger threshold can be chosen.

Solving problem 2: False detections when no signal

Detecting a packet when the incoming signal level is very low is not useful, since the SNR will probably be too low to decode the packet. By adding a fixed threshold in parallel to the adaptive threshold, a minimum signal power can be set. The fixed threshold is implemented in the same way as the one in Correlation Estimator block in GNU Radio 3.7.9, by setting the minimum ratio between the correlation magnitude and autocorrelation value of the preamble.

Solving problem 3: False detections at the end of the packet

The adaptive threshold sets the detection threshold based on the mean squared magnitude of surrounding samples. The Correlation Estimator block averages all the samples that are currently being processed in the block. The amount of samples that is passed between blocks in the dataflow graph is chosen by the GNU Radio scheduler, and can change during execution. Figure 20 visualizes which samples were considered when averaging the correlation values at a particular instant. The *START* tag indicates the first sample of a dataflow sample group, and all subsequent samples up to the next *START* tag are processed at once in the block.

It is clear that this way is not optimal when only a part of the sample group has a significant amplitude. In that case, the threshold will be low and false positives occur on the few samples with significant amplitude.

The problem can be solved by defining a fixed number of samples to average. In the improved implementation of the Correlation Estimator block, the amount of samples is based on the amount of samples of the modulated reference preamble. For a BPSK preamble of 64 symbols shaped with 4 symbols per sample, the Correlation Estimator will average 256/2 samples to set the detection threshold. These samples are chosen around the sample of interest, i.e. the sample that is currently being checked.

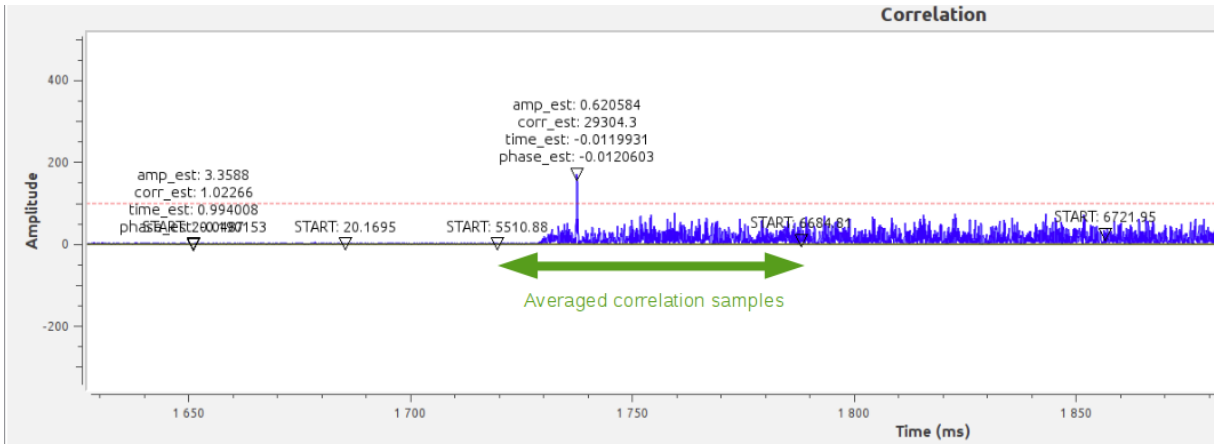


Figure 20: Correlation samples

Block in GNU Radio Companion

The Correlation Estimator 2 block is also included in the new *gr-packetizer* package made for this project. The block is shown in Figure 21. The block has two extra parameters compared to the original Correlation Estimator block: *Fixed threshold* and *Verbose*.

- Symbols: symbols of the modulated and pulse shaped preamble
- Samples per Symbol: upsampling factor
- Tag marking delay: tag position delay (number of samples) compared to detection instant
- Threshold: threshold value to set the *probability of false alarm* variable for the adaptive threshold
- Fixed threshold: threshold value to set the fixed threshold, which is relative to the power of the autocorrelation
- Verbose: output a message in the console when a peak has been detected, with the peak value, the current adaptive threshold value and the fixed detection thresholds

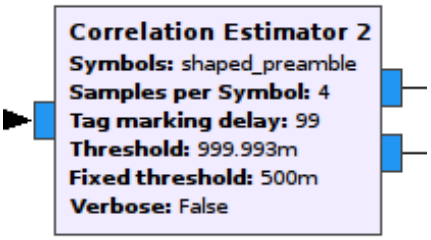


Figure 21: Correlation Estimator 2

4.3 Time and phase synchronization

The Correlation Estimator can derive time and phase information by looking at the correlation result. The block will communicate this information by adding the tags *phase_est* and *time_est* to the stream.

The Polyphase Clock Sync block is a time synchronizer that is compatible with the *time_est* tag. Its purpose is to realign and downsample the incoming stream so the signal samples of the output signal are sampled at exactly the peak of the sinc shape introduced by the root raised cosine filter. The block immediately applies the matched pulse shaping filter using a polyphase filter bank, so the block outputs constellations symbols.

Costas Loop is a phase synchronizer that optionally uses the *phase_est* tag. In fact, Costas Loop uses a phase-locked loop to track the carrier frequency. Figure 22 shows the effect of Costas Loop when a fixed phase offset is seen by the receiver. Figure 23 illustrates the case where a carrier frequency offset has been applied. Costas Loop will lock to the carrier frequency and align constellations to their correct positions. The *phase_est* tag helps to quickly correct the phase by indicating the estimated phase offset.

GNU Radio's Costas Loop expects the modulation order as an input parameter. The modulation order for phase-shift keying modulations is 2^N, where N is the number of bits per symbol. Costas Loop in GNU Radio is limited to the modulation orders 2 (BPSK), 4 (QPSK) and 8 (8PSK).

The packet encoder/decoder pair implemented in this project supports different constellation types for preamble, header and payload. This implies that the Costas Loop block can only be applied after the header and payload have been split by the Preamble/Header/Payload Demux block. The header decoder stream and payload decoder stream have their own phase synchronizers that receive bursts of header or payload data, as shown in Figure 9. The phase can have large discontinuities between those bursts, which makes it more difficult for the phase-locked loop to lock to the carrier and reduces the performance of Costas Loop.

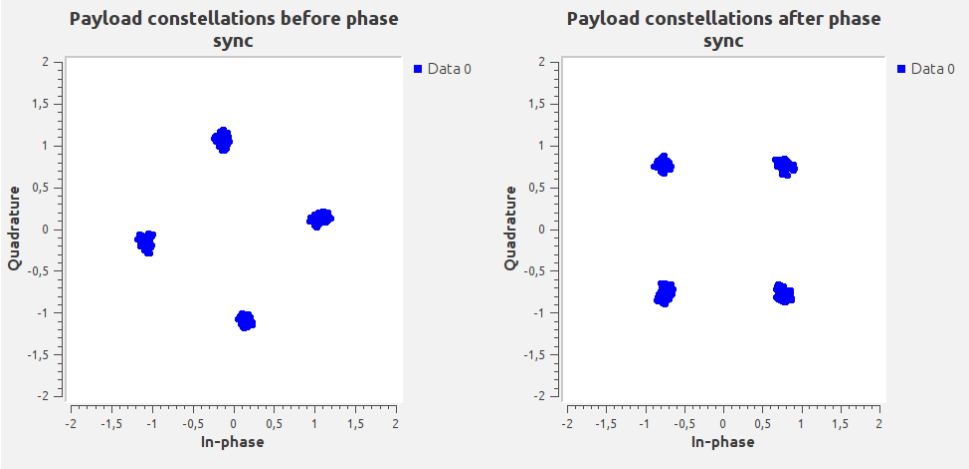


Figure 22: Effect of Costas Loop when receiving constellations with a fixed phase offset

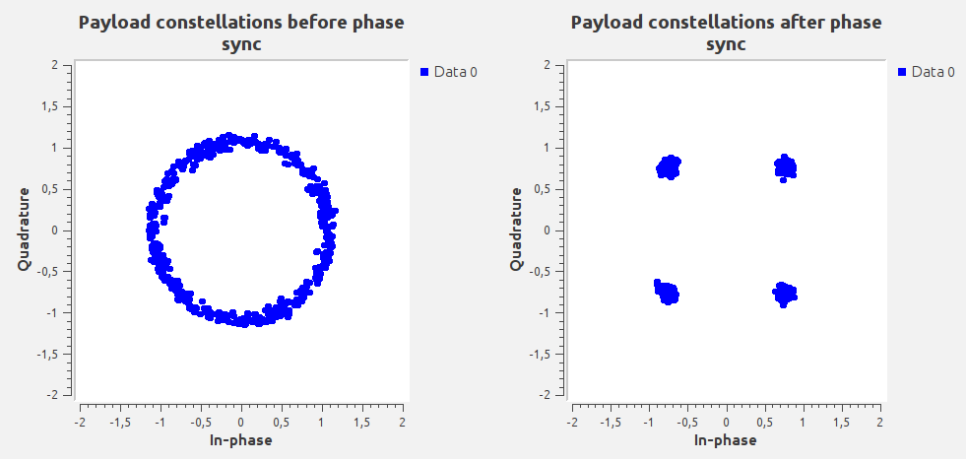


Figure 23: Effect of Costas Loop as a frequency tracker

4.3.1 Test systems

test_time_phase_sync.grc is an extension of mapping example *test_mapping.grc*. The symbols are shaped with a root raised cosine filter. A channel model is added to simulate a time offset, frequency offset, phase offset and noise in the signal. Time and phase synchronization is added with the Polyphase Clock Sync block and Costas Loop. A BER output is also provided, which can be used to analyze the effects of the channel. It is used to verify the correct working of the communication chain.

4.4 Data whitening

Data grouped in packets can contain long sequences of 0's and 1's. The lack of signal variation can be problematic for adaptive circuits such as adaptive gain control and phase-locked loops.

Data whitening or scrambling is a technique to eliminate long sequences of 0's and 1's. A simple additive synchronous whitener is implemented in this project, which whitens incoming data bytes by applying XOR operations with a pseudo-random sequence. Data can be dewhitened by applying the same XOR operations. The implementation is synchronous because the pseudo-random sequence needs to be reset to align with the data stream. Since this project uses packet-based communications, synchronization is straightforward: at the start of a packet, the random sequence is reset.

The whitener implementation is split in a kernel and a block interface. The kernel is a general implementation which can be used as a standalone whitener. The packet encoder uses this kernel to whiten incoming data. The kernel is in fact both a whitener and dewhitener, since the inverse operation of an XOR is itself.

The block interface wraps the kernel in a standalone GNU Radio block. It manages the input and output of items and provides an interface to configure the block. The whitener block then initializes the whitener kernel depending on the given arguments.

4.4.1 Whitener kernel

The whitener kernel is a C++ class that performs the whitening operations. The class *gr/packetizer/kernel/whitener.h* defines three constructors:

1. *whitener*();
   Initializes a whitener object for 8 significant bits per byte using a linear-feedback shift register.

2. *whitener*(*int bits_per_byte*);
   Initializes a whitener object for *bits_per_byte* significant bits per byte using a linear-feedback shift register.

3. *whitener*(*std* :: *vector* < *unsigned char* > *random_mask*, *int bits_per_byte*);
   Initializes a whitener object for *bits_per_byte* significant bits per byte using the given *random_mask* to XOR data with. If *random_mask* is an empty vector, a pre-computed random mask will be used.

After initialization, the whitener object can be used to process data by executing its method:

```
/*!
* Do the whitening. Starts reading data in at pointer data_in
* Starts outputting data at data_out
* Processes data_length bytes of data.
*/
void
do_whitening(const unsigned char* data_in, unsigned char* data_out,
 ↪  unsigned int data_length, unsigned int whitening_offset);
```

4.4.2 Whitener block

The whitener block is called the Tagged Stream Whitener. A *tagged stream* in GNU Radio expects a stream where data is combined in packets with a tag indicating the packet start and length. The block is both a whitener and dewhitener. After whitening, the original data can be recovered by applying the same block with the same parameters on the whitened stream.

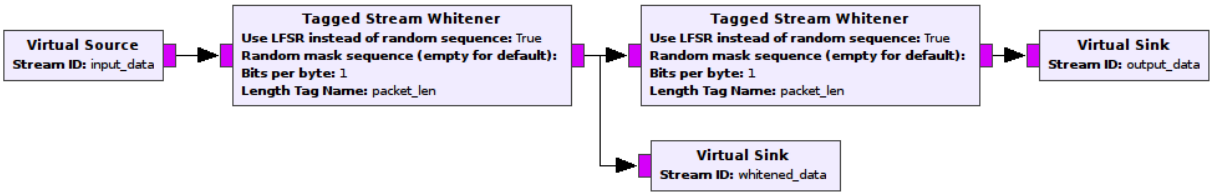


Figure 24: The Tagged Stream Whitener in GNU Radio Companion. The left block whitens the data and the right one dewhitens data.

4.4.3 Test system

test_whitener.grc illustrates the use of the *Tagged Stream Whitener* blocks.

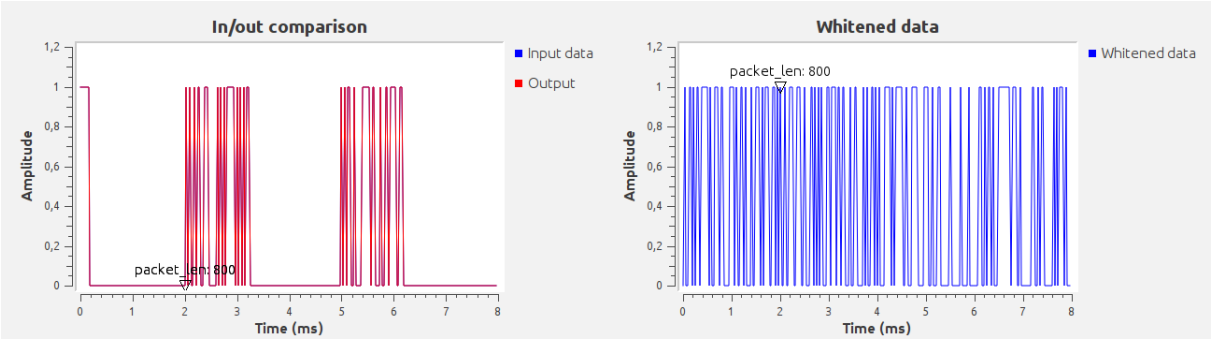


Figure 25: Non-whitened data on the left has long sequences of 0 bits.

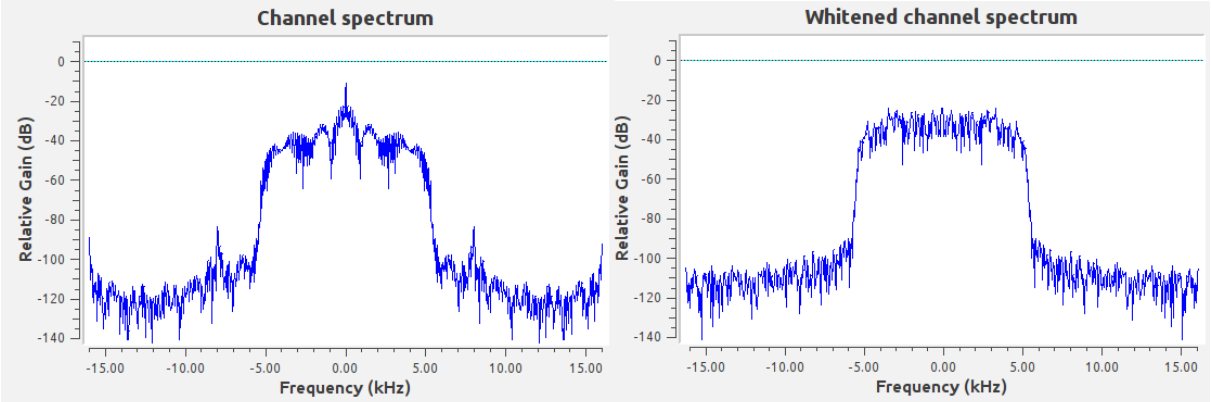


Figure 26: The non-whitened spectrum is unbalanced and has a DC peak.

4.5 Forward Error Correction

Forward Error Correction (FEC) or channel coding pre-processes the data by adding redundancy to make communications more reliable in noisy channels. Encoding data in a redundant way is done with an error-correcting code.

GNU Radio has a dedicated package with FEC modules called *gr-fec* which is extensively documented. [6] The FEC API is split on two levels: variables and deployments. A variable implements error-correcting code in encoding and decoding methods. A deployment is the connection between the flowgraph and the variable. It will handle the setup and data passing between blocks.

Several error-correcting codes are defined. A standard convolutional coder is used in this project in combination with the FEC Extended Tagged Encoder and FEC Extended Tagged Decoder as deployments. The FEC encoder is placed right before the packet encoder, and the FEC decoder right after the packet decoder. The FEC Extended Tagged Decoder has a float input and can use soft-decision decoding.

4.5.1 Test systems

test_fec.grc illustrates the use of the *FEC Extended Tagged Encoder* and *FEC Extended Tagged Decoder*. Note that the FEC encoder expects a data stream of bytes with 1 significant bit per byte. The output of the encoder is manually mapped to soft bits (-1 and 1). A separate CC Decoder object is set for each FEC decoder.

encdec_custom_fec.grc implements a packet encoder and decoder combined with forward error correction, with hard and soft decisions.

4.6 Communication chain example systems

Communication chain with Extended Packet Encoder/Decoder blocks

chain_custom.grc implements the full communication chain with packet encoder, pulse shaper, channel modeler, adaptive gain control, correlation estimator (preamble detector), time synchronization and the packet decoder (which includes phase synchronization).

Communication chain with RX receiver built with fundamental blocks

chain_rx_debug.grc implements the same communication chain as the previous example. In this example, the packet decoder is built using separate blocks, in order to debug the individual components of the packet decoder. This example can also be used to implement extensions of the packet decoder.

Communication chain with RX receiver built with fundamental blocks

chain_rx_debug_differential.grc is the same as *chain_rx_debug.grc*, but with differential decoding.

5 GNU Radio Challenges, Shortcomings and Lessons Learned

5.1 Existing blocks

GNU Radio is open-source software and constantly in development. Blocks can behave inconsistently because they are mostly contributed by enthusiasts. For example, some blocks do not process tags well because the tagging system was introduced later. In addition, extensive documentation and examples are often lacking or they are outdated.

GNU Radio has a steep learning curve, and it can be challenging to get an overview of all possibilities and features. In general, knowledge of GNU Radio API's and block development is very useful when implementing a system. Often it is only possible to get the desired functionality with a custom implementation. The knowledge of GNU Radio development is also useful when evaluating existing blocks.

Several blocks have similar functionality. For example, blocks related to frame synchronization are: *Correlation Estimator*, *PN Correlator*, *Correlate and Sync*, *Simple Correlator*, *Correlate Access Code Tag Stream*, *Correlate Access Code − Tag*. Some of these blocks are undocumented, some have no examples and some of them do not work at all with the latest version of the framework.

Blocks used for repacking bytes also demonstrate this, as *Pack K bits*, *Unpack K bits*, *Packed to Unpacked* and *Unpacked to Packed* do not propagate tags. The best implementation is the *Repack Bits* block which works for both repacking and unpacking, and propagates tags.

Often, parameters should be set by guessing an initial value and looking at the effect. Phase locked loops have a loop bandwidth parameter which is not well defined and documentation says it should be set around $2 * pi/100$.

5.2 Installing GNU Radio

Even the installation of GNU Radio can be challenging for a beginner. Although binaries for Windows exist, it is strongly recommended to use GNU Radio in Linux, especially when developing Out-of-Tree modules. Many dependencies and libraries are tedious to install under Windows.

This project was made on Ubuntu, and one has to be careful when installing GNU Radio. The recommended way to install GNU Radio is via the gnuradio package from your distribution's standard repositories [7]. On Ubuntu 16.04, these binaries install the older GNU Radio 3.7.9. In order to get the most recent version, GNU Radio should be built from source.

When installing from source, some optional dependencies might not be available, and GNU Radio will not explicitly notify this. During this project, the FEC Decoder block returned a debug message for every decoded packet. When transmitting at high throughput, the continuous flow of debug messages in the console made the GNU Radio Companion crash.

This behavior is caused by the lack of the *log4cpp* dependency. Although several ways to disable debug messages are proposed on the internet [8] , the only short-term way that helped in this project was rebuilding GNU Radio with a parameter:

```
cmake -DENABLE_GR_LOG=off <srcdir>
```

This solution is inconvenient and will also disable all other debug messages, except fatal errors.

5.3 Debug possibilities

Debugging input/output data is rather difficult in GNU Radio, especially when implementing a system in GNU Radio Companion. The common way to view signals is to attach a scope as output, but there is no convenient way to pause a system, to analyze the current state. A workaround is to add a slider to the GUI which controls the sample rate. While a sample rate of 0 does not have the desired effect, a sample rate of 1 does freeze the system. The tag positions are not always correctly displayed in the GUI plots when decreasing the sample rate to very low values.

Systems lock up when the scheduler cannot balance the data flow in the system. This may happen when a block, that has two ports which consume the same amount of samples per time instant, receives data streams with a different number of samples per second. GNU Radio will not indicate the block that is causing problems.

In general, it is challenging for beginners to locate the problem when something is not working. It could be the wrong use of an undocumented block, a wrong parameter or a problem with the block's implementation.

5.4 Block development

Block development uses the GNU Make to compile the code and install the blocks. GNU Radio provides an excellent beginner tutorial to learn coding blocks in both C++ and Python. The amount of tools and interfaces can be overwhelming. The *gr_modtool* facilitates the development of GNU Radio blocks by auto-generating the required config files, but sometimes minor adjustments have to be made manually.

- Make: terminal commands, makelists
- SWIG: interface files
- GNU Radio development: several APIs for blocks and data transfer
- GRC: XML with Cheetah templates as an interface between Python and GRC

Often, the trial-and-error involved in the process can be cumbersome. For example, the process when installing an Out-of-Order module consists of the following terminal commands:

```
cd /path/to/oot/module
mkdir build
cd build
cmake ../
make
sudo make install
sudo ldconfig
```

When changing the block's main initializer in a C++ header file, these steps will not work if the project was already compiled before. SWIG will throw an error mentioning the arguments in the header file are not correct. Beginners might think there is a problem with the C++ code, but in fact, the *build* folder has to be removed in order to remove the old compiled files and caches.

6 Conclusion

The main goal of this project was to become familiar with GNU Radio and implement a communication chain with an improved packet encoder/decoder pair. The combination of learning GNU Radio and wireless communication principles at the same time was the biggest challenge.

Some software development in Python and C++ added diversity to the project work. A new Out-of-Tree module called *packetizer* combines the blocks created during the project. We included several examples to demonstrate the blocks in the GNU Radio Companion.

We implemented new blocks to expand the functionality of GNU Radio:

1. *Extended Packet Encoder*: creates packets from a given bit stream and outputs mapped symbols
2. *Extended Packet Decoder*: decodes packets received as symbols and outputs a bit stream
3. *Tagged Stream Fix*: removes unnecessary samples between packets in a tagged stream
4. *Message Sequence Checker*: checks for dropped packets and reports the packet loss rate
5. *Tagged Stream Whitener*: whitens or dewhitens a byte stream
6. *Pulse shape vector*: applies a pulse shaping filter on a given sequence of symbols

Some existing blocks have been modified:

1. The *Preamble/Header/Payload Demux* is an extension of the existing *Header/Payload Demux* block. The extended block supports preambles.
2. The *Correlation Estimator 2* block solves some problems with the dynamic threshold of the existing *Correlation Estimator* block. Preamble detection is more reliable now.

In terms of functionality, some new blocks would fit in the core of the GNU Radio framework. The *Tagged Stream Fix* and *Pulse shape vector* provide important functionality that is not yet possible with the current blocks of GNU Radio. The *Extended Packet Encoder/Decoder* blocks are useful to quickly build a communication chain. They are a replacement for the deprecated *Packet Encoder* and *Packet Decoder* blocks. The improved blocks are compatible with their original implementation, so existing systems do not need to be adjusted.

The blocks are not ready to directly contribute them to GNU Radio, because it must be verified that they meet all the requirements for blocks in the GNU Radio core [9]. In particular, some code cleanup has to be done, extra unit tests should be created and compatibility with the upcoming version of GNU Radio should be tested.

A Byte repacking in GNU Radio

The *Integer 8* or *Byte* data type is often used in GNU Radio. Special attention is needed for the number of bits used in the byte. Sometimes, a byte stream carries less than 8 bits of data per byte and it is called "unpacked". When X bits per byte are used for data, only the X least significant bits can be non-zero and these bits are called the significant bits. The other $8 - X$ bits are always zero. As a consequence, the *rate* of the stream will be changed. For every incoming byte, $8/X$ output bytes are generated.

Some blocks output data with 8 significant bits per byte, while others expect a data stream with for example 2 bits per byte. In that case, the byte stream has to be repacked. Several blocks exist for this purpose, but the most universal blocks is the Repack Bits block. The block can repack bits in a MSB-first or LSB-first way, and adapt a tag value to the new number of output bytes.

LSB-first repacking

LSB-first repacking means that the least significant bit of the input byte is read first, and the output bytes are filled LSB-first. When the output byte has X filled bytes, the next byte will be filled.

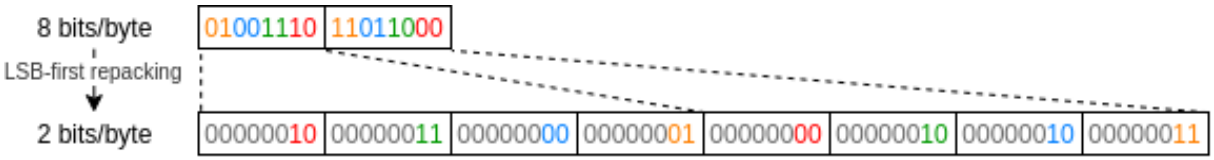


Figure 27: Unpacking example for LSB-first, with $X = 2$

MSB-first repacking

MSB-first repacking means that the most significant bit of the input byte is read first, and most significant bit of the output byte is filled first.

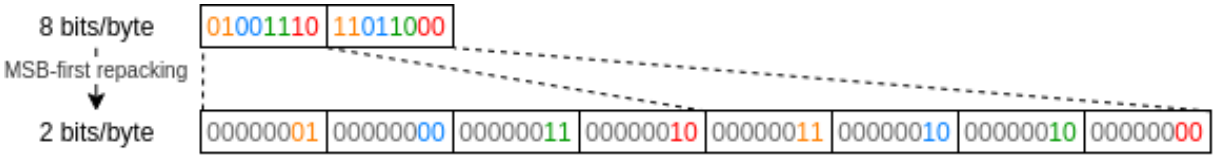


Figure 28: Unpacking example for MSB-first

References

[1] GNU Radio Manual and C++ API Reference
https://gnuradio.org/doc/doxygen/

[2] gr::digital::packet_header_default *GNU Radio Manual and C++ API Reference*
https://gnuradio.org/doc/doxygen/classgr_1_1digital_1_1packet__header__default.html

[3] GNU Radio issue #1207: Correlation Estimator regression *Github*
https://github.com/gnuradio/gnuradio/issues/1207

[4] gr::filter::pfb_arb_resampler_ccf *GNU Radio Manual and C++ API Reference*
https://gnuradio.org/doc/doxygen/classgr_1_1filter_1_1pfb__arb__resampler__ccf.html

[5] gr::filter::firdes Class Reference *GNU Radio Manual and C++ API Reference*
https://gnuradio.org/doc/doxygen/classgr_1_1filter_1_1firdes.html

[6] Forward Error Correction *GNU Radio Manual and C++ API Reference*
https://gnuradio.org/doc/doxygen/page_fec.html

[7] Installing GR *GNU Radio Wiki*
https://wiki.gnuradio.org/index.php/InstallingGR

[8] Logging *GNU Radio Manual and C++ API Reference*
https://gnuradio.org/doc/doxygen/page_logger.html

[9] Development *GNU Radio Wiki*
https://wiki.gnuradio.org/index.php/Development