SANDIA REPORT
SAND2020-9607
Printed Click to enter a date

Sandia GNU Radio Utilities

Jacob A. Gilbert
Peter A. Knee
Sam H. Whiting
Brian H. Adams

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico
87185 and Livermore,
California 94550

ABSTRACT

The Sandia Utilities form an RF toolkit extending the GNU Radio framework for straightforward interaction with bursty RF systems which can be cumbersome using built in capabilities of the framework or other open source extensions. Motivated by the abstract exercise of responding to a modulated burst of information with a unique modulated burst of information, the Sandia Utilities provide several dozen additional debug and signal processing blocks through four software modules. These blocks manifest their utility through a new general concept of operation for GNU Radio applications and extend many existing streaming capabilities to the PDU-based message passing API. Using these concepts and tools, it is straightforward to develop software capabilities to interact with existing and new bursty RF devices.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

This page left blank

ACRONYMS AND DEFINITIONS

Abbreviation	Definition
API	Application Programming Interface
EOB	End of Burst
FHSS	Frequency Hopping Spread Spectrum
GNU	“GNU’s Not Unix” – recursive acronym representing large collection of free software
GR	GNU Radio
QA	Quality Assurance
PDU	Protocol Data Unit
PMT	Polymorphic Type
RF	Radio Frequency
SDR	Software Defined Radio
SOB	Start of Burst

1. BACKGROUND

The Sandia GNU Radio Utilities are a series of software modules that Sandia has developed for the GNU Radio Software Defined Radio framework to support development of software defined RF systems. GNU Radio operates on two basic data flow models supported by the Streaming API and the Asynchronous Message Passing API. A robust set of processing blocks are included ("in tree") with the GNU Radio framework for use with the Streaming API, but despite providing a very capable structure for representing data as Protocol Data Units, an internal structure for metadata and vector data, there is only a significantly reduced set of blocks for the Message Passing API primarily focused on debugging and control as opposed to signal processing.

Using only in tree components from GNU Radio, it is very challenging to accomplish many fairly straightforward tasks that are extremely common for RF communication systems such as 'receive a burst of data on an arbitrary frequency in a given band' or 'respond to modulated data in a burst of energy with other modulated data in a new burst of energy'.

The Streaming API is a useful SDR paradigm for both continuous and bursty RF systems as the data from the software radio is generally continuous. The Message Passing API is well suited for higher layer protocols where data is refined into discrete blocks of information, albeit substantially relying custom blocks to convert the initially streaming data to a packetized format. Message passing remains preferable when working with RF signals for which data are organized into bursts or chunks as opposed to a continuous stream, and various benefits can be realized when doing this conversion early in the processing chain.

GNU Radio leaves the non-trivial task of bridging the Streaming and Message Passing paradigms primarily to the end user. The Sandia Utilities modules provide several methods for accomplishing this translation, as well as substantial enhancements of the basic set of Messaging Passing API blocks with signal processing functions found within the in-tree Streaming API as presented at the GNU Radio Conference in 2019 [1].

1.1. Document Organization

This document summarizes the conversion schema provided, the overall structure of the extension modules, and covers information about some of the key blocks included in the toolkit.

2. SOFTWARE STRUCTURE

The Sandia GNU Radio Utilities are organized into four primary software extensions to GNU Radio developed as Out of Tree (OOT) modules. These are separate software projects that are managed through the git version control system and have a branch structure that mirrors that of the upstream GNU Radio project. The `maint-X.Y` branches represent the latest supported code that are compatible to the upstream GNU Radio framework version `X.Y`, while the master branch is intended to track the latest GR master codebase. Due to the maintenance burden of managing multiple branches, the latest `maint-X.Y` branch receives the most up-to-date support.

The specific intention of the various modules is discussed in subsequent sections. Blocks are coarsely categorized into functional groupings and are maintained in different modules to simplify software deployment and build overhead. Over time as requirements evolve some blocks become outliers and may not fit well in the prescribed module, these are eventually adjusted but because of backward compatibility concerns this may take some time to happen.

2.1. GR PDU Utilities (gr-pdu_utils)

This GNU Radio module contains tools for manipulation of PDU objects. There are blocks to translate between streams and PDUs while maintaining timing information, several self-explanatory blocks that emulate the behavior of some in-tree stream blocks for messages and PDUs, and some other features. This module is complimentary to the `gr-timing_utils` module and some of the advanced timing features require blocks there. Many of the `gr-pdu_utils` blocks are message-based analogs of streaming API blocks found in-tree to support software development.

This module underpins all the burst software and thus has some software requirements levied. All non-hier blocks are written in C++ and make use of the `gr::log` API with no stdout output. There is minimal PMT symbol interning at runtime, and blocks should be thread safe. QA code exists for all blocks, and when bugs are fixed QA is updated to catch the errors. Code should be clang-format applied with the GR clang style file.

2.2. GR Timing Utilities (gr-timing_utils)

This GNU Radio module contains tools for various time related functions. This module is complimentary to the `gr-pdu_utils` module and contains a variety of generally useful blocks related to time management or analysis within GR applications. This module has become smaller over time and may eventually be replaced entirely as the contained features become obsolete or moved to more appropriate modules.

Blocks in this module that are not hierarchical blocks should be written in C++, have minimal stdout output, and have QA written.

2.3. GR FHSS Utilities (gr-fhss_utils)

This GNU Radio module contains tools for processing frequency hopping spread spectrum signals, as well as known and unknown bursty signals in general. Blocks derived from the gr-iridium project exist to detect narrowband bursts within wideband signals and down-convert and center them. Metadata is tracked through this process enabling reconstruction of where the bursts originated in time and frequency.

The module is named 'FHSS Utilities' due to the original application being frequency hopped signal reception, but despite this the concept of operation has many applications beyond FHSS systems. It has direct application to any need to extract bursty energy from a band with unknown time or frequency. This module is not well suited for situations where bursts of energy rely on process gain for reception as the detection sensitivity is around 6-8dB when properly tuned, below which false alarms tend to overwhelm valid detections.

Blocks in this module that are not hierarchical blocks should be written in C++, have minimal stdout output, and have QA written.

2.4. GR Sandia Utilities (gr-sandia_utils)

This GNU Radio module contains a variety of blocks that do not fit well within the PDU, timing, or FHSS utility modules. Often these are improved or fixed duplicates of in-tree blocks awaiting upstreaming, python proof-of-concept blocks that will eventually be moved into other modules, blocks for which usage is complicated or likely to result in user error, or blocks with niche use.

Blocks in this module do not require meeting the same standards as the other modules, though they should have at least basic instantiate-and-start QA.

3. STREAMING TO MESSAGE CONVERSION

The general concept of PDU conversion using this module is shown below in Figure 1.

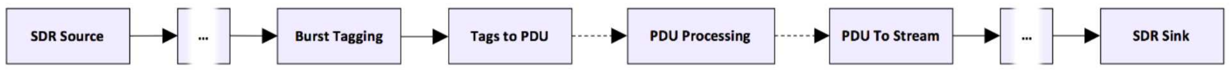


Figure 1 - General Concept of PDU Conversion

The start and end of a burst is tagged via some means (see below) and the Tags to PDU block will emit a PDU with the PMT vector as the data between the tags. Some basic metadata is provided, and advanced options exist for timing reasons. Once turned into a PDU, the data can be processed, and if desired the data can be turned back into a stream with `tx_sob` and `tx_eob` tags for transmission with a UHD-style sink.

Advanced timing modes for RX are included that allow for coarse timing (+/- a couple symbols) which is enough for most communications applications. This timing is not intended for precision timing, rather as an option for relatively good timing through rate changes without too much overhead. This is particularly well suited for low-order digital signals; however, the general concept can be used for displaying all sorts of data. If End-of-burst cannot be practically tagged, it is possible to configure the Tags to PDU block to emit a fixed-length PDU and downstream processing can handle the fine details.

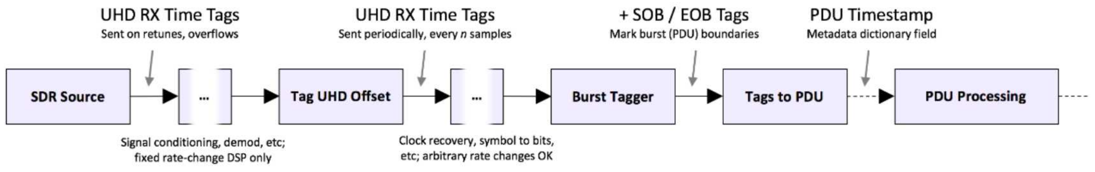

UHD RX Time Tags
Sent on retunes, overflows
UHD RX Time Tags
Sent periodically, every n samples
+ SOB / EOB Tags
Mark burst (PDU) boundaries
PDU Timestamp
Metadata dictionary field
SDR Source
...
Tag UHD Offset
...
Burst Tagger
Tags to PDU
PDU Processing
Signal conditioning, demod, etc; fixed rate-change DSP only
Clock recovery, symbol to bits, etc; arbitrary rate changes OK

Figure 2 - PDU Receive Time Tracking

UHD-style receive time tags are emitted by the source and recorded by the Tag UHD Offset block which will periodically emit updated UHD-style time tags. The rate at which these are emitted is controllable; which is necessary if any arbitrary-rate-change blocks are present between the source and the Tags to PDU block. Burst tagging works as in the basic mode, and the Tags to PDU block will automatically use the UHD-style time tags to determine burst time. It is important that blocks propagate tags correctly through rate changes. Clock recovery is a step for which this can be complicated; the new (as of 3.7.11) Symbol Sync block is a good option for CR as it propagates tags correctly when the output SPS is set to 1.

Timed transmission is also supported via UHD-style `tx_sob` and `tx_eob` tags as shown below.

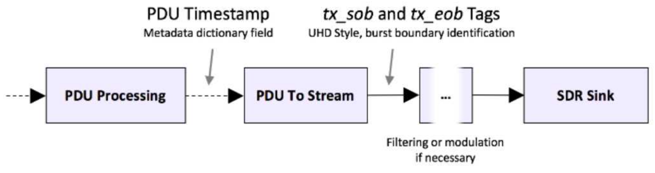

PDU Timestamp
Metadata dictionary field
tx_sob and tx_eob Tags
UHD Style, burst boundary identification
PDU Processing
PDU To Stream
...
SDR Sink
Filtering or modulation if necessary

Figure 3 - PDU Transmit Time Tracking

When the metadata dictionary key `tx_time` is provided with UHD-style tuple of {uint64 seconds, double fractional seconds}, this value will be added to a `tx_time` tag on the `tx_sob` sample. This allows for timed transmission with UHD-style transmit blocks.

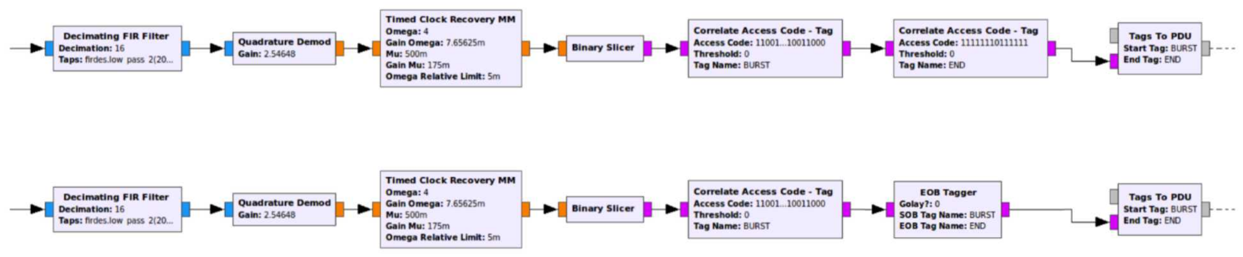
3.1. Burst Tagging Methods

Conversion between the Streaming and Messaging APIs using the Sandia Utilities require that signals be localized temporally and spectrally in order to be further processed. There are several ways to tag bursts that work well with this type of data that are presented here.

3.1.1. *Start and End of Burst Tagging*

To identify Start-of-Burst, a very simple way is to do a basic energy detection threshold and tag according to energy level, however this is not very robust. Another straightforward method is to run an open-loop demodulator and use a Correlate Access Code - Tag block to detect the preamble and start of unique word to delineate and tag SOBs. Alternately, a correlation-based method can be used if this is already implemented for preamble detection.

End of Burst tagging can be trickier as there are several ways RF protocols indicate end-of-burst. Generally, the best way is to write a custom block that detects SOB tags, parses the RF header for length information, and automatically tags EOB's accordingly. Sometimes RF protocols will specify a detectable EOB sequence which can be detected with a second Correlate Access Code - Tag block directly. For fixed length bursts an EOB is not necessary and the block can be configured by the maximum length parameter to effectively set the EOB. Example processing flow for this type of tagging is shown below in Figure 4.



Decimating FIR Filter
Decimation: 16
Taps: firdes.low pass 2(20...

Quadrature Demod
Gain: 2.54648

Timed Clock Recovery MM
Omega: 4
Gain Omega: 7.65625m
Mu: 500m
Gain Mu: 175m
Omega Relative Limit: 5m

Binary Slicer

Correlate Access Code - Tag
Access Code: 11001...10011000
Threshold: 0
Tag Name: BURST

Correlate Access Code - Tag
Access Code: 11111110111111
Threshold: 0
Tag Name: END

Tags To PDU
Start Tag: BURST
End Tag: END

Decimating FIR Filter
Decimation: 16
Taps: firdes.low pass 2(20...

Quadrature Demod
Gain: 2.54648

Timed Clock Recovery MM
Omega: 4
Gain Omega: 7.65625m
Mu: 500m
Gain Mu: 175m
Omega Relative Limit: 5m

Binary Slicer

Correlate Access Code - Tag
Access Code: 11001...10011000
Threshold: 0
Tag Name: BURST

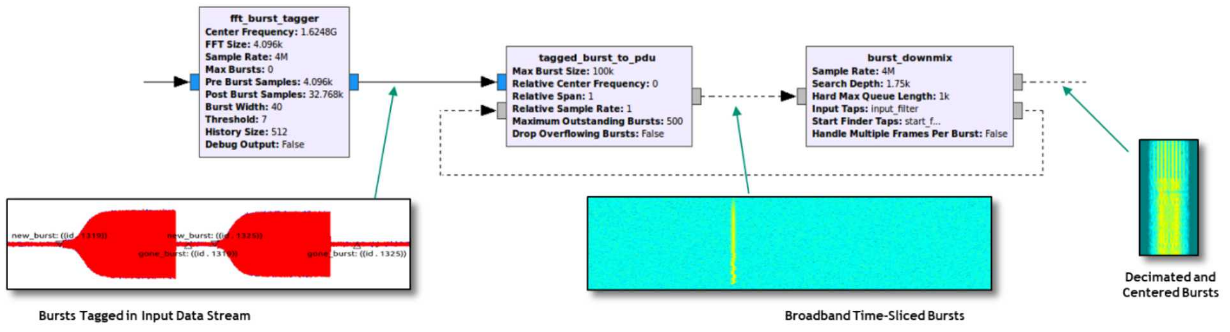
EOB Tagger
Golay?: 0
SOB Tag Name: BURST
EOB Tag Name: END

Tags To PDU
Start Tag: BURST
End Tag: END

Figure 4 - Examples of Using SOB/EOB Burst Tagging

3.1.2. *Broadband Energy Burst Tagging*

SOB/EOB sequence detection requires knowledge of the modulation scheme and details about the time/frequency medium access scheme and the signal protocol framing, and quickly breaks down for signals in which one or more of these features are not known. The FHSS Utilities module provides a different way to isolate bursts based on broadband energy detection. The general processing flow for this model of burst detection is shown in Figure 5.



fft_burst_tagger
Center Frequency: 1.6248G
FFT Size: 4.096k
Sample Rate: 4M
Max Bursts: 0
Pre Burst Samples: 4.096k
Post Burst Samples: 32.768k
Burst Width: 40
Threshold: 7
History Size: 512
Debug Output: False

tagged_burst_to_pdu
Max Burst Size: 100k
Relative Center Frequency: 0
Relative Span: 1
Relative Sample Rate: 1
Maximum Outstanding Bursts: 500
Drop Overflowing Bursts: False

burst_downmix
Sample Rate: 4M
Search Depth: 1.75k
Hard Max Queue Length: 1k
Input Taps: input_filter
Start Finder Taps: start_f...
Handle Multiple Frames Per Burst: False

new_burst: ((id . 1319))
new_burst: ((id . 1325))
gone_burst: ((id . 1319))
gone_burst: ((id . 1325))
Bursts Tagged in Input Data Stream
Broadband Time-Sliced Bursts
Decimated and Centered Bursts

Figure 5 - Broadband Energy Detection of Bursts

In this model, time domain data representing a broadband digitized signal is passed through a processing block which makes time and frequency estimates for energy using the Discrete Fourier Transform and applies tags in the output data stream. The next block isolates these tagged start and end of bursts in time, of which there can be many simultaneously, and emits each individually as a PDU with metadata indicating the approximate center frequency estimation. The final block in this processing chain uses the center frequency estimate to decimate the block in frequency, re-estimate the center frequency, and perform fine frequency correction prior to emitting the signal data that has been time/frequency isolated.

Many such detections can happen simultaneously as would be observed in real-world congested multi-user bands such as the various unlicensed ISM bands.

3.1.3. Additional PDU Techniques

There are other possibilities for burst tagging that are not included within the Sandia Utilities modules and more tools may be included in the future. It is also not a requirement that PDUs be generated only on bursty signals. Many continuous signals contain frame sync data or other in-band signaling that can be used to break down signals into discrete units that are more suitable for PDU processing. It is also possible for out-of-band information to be processed and used to segment a signal into PDUs. Added functionality will be included for Streaming to Message API conversion in the future as requirements are identified.

4. KEY SIGNAL PROCESSING BLOCKS

The concept and usage of a few significant blocks is described in this section. Many blocks are omitted as their behavior is straightforward or documented in the GRC YAML sufficiently.

A complete listing of blocks as of this document is shown in Appendix A.

4.1. Key Blocks from the GNU Radio PDU Utilities Module

4.1.1. Tags to PDU Block

4.1.1.1. Basic Usage

The Tags to PDU block will accept a stream input and produces a PDU output. The start of a PDU is indicated by a configurable GR stream tag (only the Key matters), and the end of a PDU can be defined by a configurable tag, or by a configurable maximum length; when the maximum length is reached, the PDU will be emitted. If a second start-of-burst tags is received prior to an end-of-burst tag or the maximum length being reached, the block will discard data from the first tag and reset the internal state starting with the latest start-of-burst tag. The block also accepts a Prepend vector argument, which allows for data elements to be included at the start of each PDU. This is useful for byte alignment, or when correlating against a complete or partial Unique Word, and it is desirable to have the complete UW represented in the output. The elements in this vector count toward the Max PDU Size parameter.

4.1.1.2. Optional Burst Identification Parameters:

The Tags to PDU block can be configured to only accept EOB tags in discrete relationships to the SOB tag position through the EOB Alignment and EOB Offset parameters. This will ensure that valid EOB tags are only at n * EOB Alignment item indexes, and the EOB Offset can be used to slew that value if the SOB tag is not suitably located. Additionally, a Tail Size can be specified and that number of items after the EOB tag will be included in the PDU if allowable.

4.1.1.3. Advanced Timing Features:

As described above, this block can be used with UHD-style time tags to provide a reasonably accurate burst timestamp (within a few symbols / bits) with relatively minimum overhead. The key for these tags can be modified but is normally `rx_time` and the data consist of a two element tuple of uint64 seconds followed by double fractional seconds in range [0, 1). This timing works by knowing the sample rate of the block and keeping track of the last known time-tagged sample and it's offset. Time is then propagated forward assuming the sample rate is exactly precise. As this can drift over time for a variety of reasons, it may be desirable to time-tag samples periodically upstream (e.g.: on burst detections) to improve accuracy and address clock drift, variable block ratios, or dropped samples; the Tag UHD Offset block from the `gr-timing_utils` module can be used to assist with this. The block also supports the ability to generate boost timestamps in seconds from Unix epoch format. This is helpful for debugging but generally less accurate and may carry a greater processing penalty.

4.1.1.4. Detection Emissions:

The block can be configured to emit a message every time a SOB tag is detected. This is useful when a low-latency reaction is necessary to incoming data, though it must be used with caution as it is prone to false detections. The emission is simply a uint64 PMT containing the offset of the received SOB tag.

4.1.2. Take Skip to PDU Block

This block is a much simpler case of the Tags to PDU Block for which a PDU is generated of length `M` every `N` samples. If `M=N` then this block is a direct conversion from stream to PDU.

4.1.3. PDU to Bursts Block

4.1.3.1. Basic Usage:

The PDU to Bursts block accepts PDUs of user-specified type and emits them as streaming data. The original intent of this block was to allow USRP based transmission of data originating from PDU-based processing from data converted to PDUs by the Tags to PDU block for half-duplex transceiver applications. As such, the block will automatically append `tx_sob` and tx_eob tags around streaming output data to indicate the start and end of valid data to the SDR. The block is simple to use; configuration is limited to type and behavior when new PDUs are received while the data from a current PDU is still being emitted. The data can either be appended to the current burst, dropped, or the block can throw an error ('Balk'). The latter two modes were implemented for very specific cases and generally 'Append' mode is the best choice. The number of PDUs that can queue up waiting for transmission is also configurable to bound memory usage (though the individual PDUs can be large).

4.1.3.2. Timed Transmissions:

The PDU to Bursts block also supports UHD-style timed transmissions. If a PDU metadata dictionary key `tx_time` exists, and the value is a properly formatted UHD time tuple, a `tx_time` tag will be added along with the `tx_sob` tag to the first item in the PDU, which will be recognized as a timed transmission by downstream blocks. Late bursts will be handled according to the behavior of the downstream processing elements, and may be dropped, sent immediately, or potentially errors caused. It is also necessary to be careful with setting timestamps too far in the future as this can result in issues due to backpressure in the DSP chain.

4.1.4. Tag Message Trigger Block

4.1.4.1. Overview:

The Tag Message Trigger block emits PDUs based on certain input conditions observed on either stream or message inputs and supports operating with or without an arming step prior to triggering. This block is more complicated and powerful that it looks, though it has utility in many straightforward applications also. The initial intention of this block was to allow for a stream tag to emit a PDU immediately. This has been expanded upon to support several additional modes of operation which are described here.

4.1.4.2. Basic Tag Based Message Emission:

The most basic mode of this block looks for a tag and emits a message (can be a PDU but does not have to) when it is identified. This was expanded on to allow the concept of a separate 'arming' tag, and an internal armed/disarmed state; the block will only emit a message when armed. If the arming key is set to `PMT_NIL`, the block is always armed and operates in the simple mode of emitting a message when a key is seen.

4.1.4.3. Usage of the Arming and Re-trigger Holdoff:

The arming status can also be set to automatically disarm after a certain number of samples which is the Holdoff parameter. An internal counter will track samples from the arming event, and a trigger event that occurs after the arming offset + holdoff will have the same behavior as a trigger event when the block is disarmed. The holdoff value will also set the shortest interval for which the triggering action can happen, even when the block is set to always be armed. A second trigger event that is less than holdoff samples from the previous trigger event will be treated as though the block is disarmed. If the holdoff parameter is set to zero it will have no effect.

4.1.4.4. Timed PDU Mode:

The Tag Message Trigger block can operate in two fundamental message modes. In 'Message' mode, the block will emit whatever PMT it has been configured to directly. The alternative, 'Timed PDU' mode allows for the block to add a `tx_time` metadata dictionary value that is a user definable amount of time in the future. The block will automatically track time of samples in the same way the Tags to PDU block does, and the `tx_time` key will be a UHD time tuple Delay Time seconds from when the trigger tag was received. This is useful for automatically transmitting a known signal whenever a qualified trigger event is detected.

4.1.4.5. Transmission Limit:

A final feature that is not exposed to the GRC is the concept of transmission limits. This value can be updated by callback, and if it is not set to `TX_UNLIMITED` it will be decremented each time the block triggers. When it reaches zero, trigger events will be treated as though the block is disarmed.

4.1.5. PDU Pack Unpack Block

The PDU Pack Unpack block is primarily used to convert between U8 PDUs representing one bit per element, and U8 PDUs representing 8 bits per element. The block can also convert bit order for packed U8 data. The usage is straightforward, specify pack/unpack/reverse and declare the bit order, however it is included in this section due to the usefulness.

This block could also be extended to support other options (higher order modulation packing, conversion to U16, etc) if necessary but a use case as not yet been identified. If this happens appropriate test code should be added to exercise such conditions.

4.1.6. PDU Add Noise Block

This block can be used to add uniform random values to an input array of uint8, float, or complex data; other PDU types will be dropped with a WARNING level GR_LOG message. This is straightforward; however, it is included here as the block also the non-obvious capability to scale and offset the input data PDU as well. This is done through the following logic:

```
        out[ii]=(input[ii]+((d_rng.ran1()*2-1)*d_noise_level))*d_scale+d_offset
```

Which is to say that the order of operations is to apply the random data first, then scale the data, then offset it. Generally useful for debugging and testing, and as such it has not seen extensive use so there may be some issues. Noise profiles are not supported, only uniform random data from the ran1() function within `gr::random`. It could be argued that these should be separate blocks entirely, but to reduce the overhead of converting data to and from PMT it was implemented this way to allow all three operations to be done at once.

4.1.7. PDU Clock Recovery Block

This block performs clock synchronization and symbol recovery on 2-ary modulated data using algorithms from M. Ossmann's WPCR project [2]. The block accepts soft and unsynchronized data and uses a zero-crossing detector to effectively recover data sampled between 4 and 60 samples per symbol, though it does perform better below 16 samples per symbol. Compared to in-tree options, this block has several advantages, primarily that it operates on PDU formatted data enabling it to work within the Message Passing API. Because the block operates on PDU data, it can make use of the entire packet to aid in data synchronization improving sensitivity. Additionally, the block does not require precise configuration or tuning which results in reduced user-error and increased capability when processing signals for which exact parameters are unknown.

4.1.8. PDU Blocks From gr::filter

Several blocks reproducing in-tree filter functionality have been built to enable PDU based filtering and resampling. This allows early conversion to PDUs and the benefits therein.

4.1.8.1. PDU FIR Filter Block

This block mirrors the in-tree FIR filter block and uses the same underlying VOLK optimizations. The use of this block has uncovered several invalid operations due to the pointer logic used which do not manifest themselves when used with the streaming API but are a problem with the filter kernels in general. Upstream issues have been filed and workarounds built into the blocks.

These blocks do not use FFT based filters as the FFT kernels are not yet templatized. This may be added as an option to this block in the future as the FFT implementation of discrete filters is more efficient for large numbers of taps.

4.1.8.2. PDU PFB Arbitrary Resampler Block

This block mirrors the in-tree arbitrary resampling block reusing the underlying kernel and operates similarly. Included here for awareness.

4.1.9. PDU Flow Controller Block

The GNU Radio Asynchronous Message Passing API has no concept of flow control or backpressure. A slow block in the processing chain will cause an unbounded backup of messages which can in turn result in software failures as messages are never dropped, and the publish method does not block.

This block will check the message queue size for subscribed blocks, and it will drop messages over a configurable maximum queue size. Dropping data is not always preferred, so this should only be used in situations where data loss is acceptable.

4.2. Key Blocks from the GNU Radio Timing Utilities Module

4.2.1. Wall Clock Time Block

The wall clock time block is very simple, it adds the wall clock time to a user defined key in the PDU metadata for every PDU it receives. The time is pulled in from `boost::get_system_time()` This

This is very helpful when it comes to debugging message passing performance, as the conventional in-tree tools for performance evaluation of Streaming API blocks are not all usable with the Message Passing API. Non-PDU type messages are dropped.

4.2.2. Time Delta Block

This block is designed to be used downstream from the Wall Clock Time block and will use the system time to evaluate the difference between the time in the PDU metadata and the current time. This is done with the same `boost::get_system_time()` call and is added to a new user defined PDU metadata key.

To compute the nominal time a PDU is taking to process through a message-based GR flowgraph, the Wall Clock Time block should be placed upstream of the block under test, and the Time Delta block downstream. The resulting metadata will accumulate the time difference across the block under test as the flowgraph executes.

4.3. Key Blocks from the GNU Radio FHSS Utilities Module

4.3.1. FFT Burst Tagger Block

4.3.1.1. Usage

The FFT Burst Tagger block accepts an input stream of data, generally a wideband recording, and emits an identical stream of data with the addition of metadata stream tags representing energy detections within the stream. The stream tags are one of two types: new burst with a PMT key of `new_burst` or end of burst tags with a PMT key of `gone_burst`. Both tags have a PMT value of a dictionary with a `burst_id` entry to correlate the start and end of bursts. New burst tag value dictionaries, in addition to the burst identifier also have the following additional fields:

- Relative Burst Frequency Estimate
- Input Stream Center Frequency
- Magnitude Estimate
- Noise Power Estimate
- Input Stream Sample Rate
- Bandwidth Estimate

This block requires a significant amount of output buffer be available in the upstream block, and due to some GNU Radio oddities, this cannot effectively be set internally. Thus, it is necessary to set the minimum output items of the upstream block to a large value. If this is not possible due to the upstream block behavior, and intermediate block such as the Multiply Constant (*m*=1) block may need to be used.

4.3.1.2. Operation

Configuration of this block is complicated in the sense that misconfiguration can cause very poor or unexpected performance. Most configuration issues can be avoided by understanding a few important concepts.

First, the block operates on `fft_size` blocks of data, which is user configurable at instantiation. The smallest internal unit of time this block can manage is `fft_size/sample_rate` seconds and setting this value large can result in problems particularly when the before-burst and after-burst lengths are small. Bursts must be observed on one or more contiguous FFT bins above the configured threshold for a user specified number of FFTs (quantized duration) before what in internally termed a 'pre-burst' will be tagged as a burst.

The decision of whether an FFT bin has crossed the threshold or not is made based on a dynamic noise floor estimation based on a user configurable amount of historical data known as the `history_size`. Upon startup or reset, the block will acquire at a minimum `history_size` FFTs of data to establish a noise floor estimate. Energy present will be factored into the noise floor, and as such initial bursts will be missed by design. The noise floor of each bin is managed individually, and adjacent energy is not factored into the calculation, and this calculation is only updated when a bin is not declared to contain a burst of energy to prevent spectral desensitization. There is a hard-coded hysteresis factor of 50% on the threshold to declare a burst gone.

A small amount of energy on either side of the detected burst is generally useful, and the `pre_burst_length` and `post_burst_length` accomplish this. If these values are set too large, temporally adjacent but spectrally aligned bursts may become combined into single detections.

4.3.2. Tagged Burst to PDU Block

The tagged burst to PDU block accepts a tagged input stream, either from the FFT Burst Tagger block or another block with compatible burst tagging structure. It accesses metadata from the value dictionary of the the in-band stream tags to understand the non-temporal composition of each burst and applies this data to the task of centering the burst and decimating the signal in bandwidth.

This metadata is limited in precision and accuracy, and thus must have some additional refinement done. In order to ensure that filtering is not removing burst spectral components, decimation is done as a two-stage process where each burst is initially decimated to twice the rate requested by the user defined output decimation, and a second more precise center frequency estimation is computed followed by additional frequency translation and decimation by a factor of 2 is done to arrive at the defined output data rate (decimation). Time-domain inaccuracies due to the FFT-length duration precision of the input tags are expected to be adjusted downstream via magnitude or correlation.

This block is based off existing work where the PDU aggregation and burst decimation are accomplished separately. This block combines these to reduce converting large vectors to PMT objects and provides a mechanism to use multiple threads to accomplish the expensive filtering and down sampling operations required when many bursts are contained within a given work function.

4.3.3. FSK Burst Extractor Hier Block

The FSK Burst Extractor Hier block demonstrates the FHSS Utilities concept of operation with the following blocks preconfigured and connected for convenience when processing 2-ary FSK signals.

```
      |---------|     |-----------|     |---------|     |---------|
      |   FFT   |     |   Tagged  |     |  Fine   |     |  Fine   |
In--->|  Burst  |---->| Burst To  |---->|  Burst  |---->|  Time   |---> Out
      | Tagger  |     |    PDU    |     | Measure |     | Measure |
      |---------|     |-----------|     |---------|     |---------|
```

Shared configuration parameters are derived from the same input configuration source, and most values with units of FFTs are unwrapped to seconds. This block formerly contained the PDU Quadrature Demod and PDU Clock Recovery blocks making it 2-FSK specific, but it is significantly more general now.

4.4. Blocks from the GNU Radio Sandia Utilities Module

4.4.1. Block Buffer Block

The Block Buffer is a unique streaming block that is useful in situations where underflows are likely (or even possible) and the system requires guaranteed continuous data. This block internally buffers a user defined number of samples watching for UHD style stream tags which can indicate an overflow, sample rate change, or frequency changes from the source. Data generated from this block is guaranteed to have no overflow, rate or frequency change discontinuities in the data, at the expense of being lossy.

A BLOCK tag is output with the first sample of each block. The value of the tag is the number of samples which have been skipped since the last block (not counting any skips from source overflows).

An example use case would be to place this block immediately after a source, proceeded by a throttle. Then, the source can be set to a high sample rate, while the processing is done at a lower, configurable rate. Only connect one block to the output of this block as odd behavior has been observed with multiple connected blocks due to the scheduler.

4.4.2. Sandia File I/O Blocks

Several file blocks representing similar in-tree functionality. These blocks are functionally equivalent to the in-tree file source block with the only architectural difference being a message port extension to specify a new input file, and the ability to process MIDAS style `bluefiles` if the `bluefile` library is found at build time, raw files with or without a header, or message files.

4.4.2.1. Sandia Utilities File Source

If the 'Force New File' option is chosen, the current file being processed will be closed and the new file opened when commanded. Adding file tags will cause the stream tags available based on the file type to be added to the output stream at the first sample of the file. Similarly, if the beginning tags are populated, the first sample of every file will contain that tag. The PDU sink port allows remote control of the file to be played, the PDU metadata must contain a key of `fname`. The value associated with `fname` is the file name that will be replayed.

4.4.2.2. Sandia Utilities File Sink

This block supports dynamic file name based on signal parameters for sampling rate, frequency, and start time. It also supports manual or triggered based file saving, rolling output files based on specified file size (samples), and alignment of start of file to nearest second boundary. If a file length of 0 is specified, it will result in a single file being generated. Successive files increment time and file number accordingly and a new folder can be generated at the start of a new collect. The folder name will have the format: `YYYYMMDD/HH_MM_SS`. Additionally, file names can be customized using all conversion specifications supported by `strftime`.

REFERENCES

[1] Gilbert, J. A. (2019, September 16-20) *The GNU Radio PDU Utilities* [Conference Presentation]. GNU Radio Conference 2019, Huntsville, AL, United States. https://www.gnuradio.org/grcon/grcon19/presentations/the_gnu_radio_pdu_utilities

[2] Ossmann, M. https://github.com/mossmann/clock-recovery/blob/master/wpcr.py

APPENDIX A. COMPLETE LIST OF GNU RADIO UTILITIES BLOCKS

This is accurate as of publication but is subject to change as development continues.

Table 1 - List of Blocks

Module	Block Name	Status
pdu_utils	extract_metadata	active
pdu_utils	message_counter	active
pdu_utils	message_emitter	active
pdu_utils	message_gate	active
pdu_utils	message_keep_1_in_n	active
pdu_utils	msg_drop_random	active
pdu_utils	pack_unpack	active
pdu_utils	pdu_add_noise	active
pdu_utils	pdu_align	active
pdu_utils	pdu_binary_tools	active
pdu_utils	pdu_burst_combiner	active
pdu_utils	pdu_clock_recovery	active
pdu_utils	pdu_complex_to_mag2	active
pdu_utils	pdu_downsample	active
pdu_utils	pdu_fine_time_measure	active
pdu_utils	pdu_fir_filter	active
pdu_utils	pdu_flow_ctrl	active
pdu_utils	pdu_gmsk_fc	active
pdu_utils	pdu_head_tail	active
pdu_utils	pdu_length_filter	active
pdu_utils	pdu_logger	active
pdu_utils	pdu_pfb_resamp	active
pdu_utils	pdu_preamble	active
pdu_utils	pdu_quadrature_demod_cf	active
pdu_utils	pdu_range_filter	active
pdu_utils	pdu_round_robin	active
pdu_utils	pdu_set_m	active
pdu_utils	pdu_split	active

Module	Block Name	Status
pdu_utils	pdu_to_bursts	active
pdu_utils	qt_pdu_source	active
pdu_utils	sandia_message_debug	deprecated
pdu_utils	tag_message_trigger	active
pdu_utils	tags_to_pdu	active
pdu_utils	take_skip_to_pdu	active
pdu_utils	upsample	active
timing_utils	add_usrp_tags	active
timing_utils	edge_detector_bb	active
timing_utils	edge_distance	active
timing_utils	interrupt_emitter	active
timing_utils	retune_uhd_to_timed_tag	active
timing_utils	system_time_diff	active
timing_utils	system_time_tagger	active
timing_utils	tag_uhd_offset	active
timing_utils	thresh_trigger_f	active
timing_utils	timed_cordic_emulator	active
timing_utils	time_delta	active
timing_utils	timed_freq_xlating_fir	deprecated
timing_utils	timed_tag_retuner	deprecated
timing_utils	uhd_timed_pdu_emitter	active
timing_utils	usrp_gps_time_sync	active
timing_utils	wall_clock_time	active
fhss_utils	cf_estimate	active
fhss_utils	coarse_dehopper	active (GRC hier)
fhss_utils	fft_burst_tagger	active
fhss_utils	fft_peak	active (GRC hier)
fhss_utils	fine_dehopper	active (GRC hier)
fhss_utils	fsk_burst_extractor_hier	active (GRC hier)
fhss_utils	s_and_h_detector	active (GRC hier)
fhss_utils	tagged_burst_to_pdu	active

Module	Block Name	Status
fhss_utils	fsk_burst_extractor_hier	active (Python hier)
sandia_utils	block_buffer	active
sandia_utils	burst_power_detector	active
sandia_utils	complex_to_interleaved_short	active
sandia_utils	compute_stats	active
sandia_utils	csv_reader	active
sandia_utils	csv_writer	active
sandia_utils	file_archiver	active
sandia_utils	file_monitor	active
sandia_utils	file_sink	active
sandia_utils	file_source	active
sandia_utils	interleaved_short_to_complex	active
sandia_utils	invert_tune	active
sandia_utils	max_every_n	active
sandia_utils	message_file_debug	active
sandia_utils	message_vector_file_sink	active
sandia_utils	message_vector_raster_file_sink	active
sandia_utils	python_interface_sink	active
sandia_utils	python_message_interface	active
sandia_utils	rftap_encap	active
sandia_utils	stream_gate	active
sandia_utils	tag_debug	deprecated
sandia_utils	tag_debug_file	active
sandia_utils	tagged_bits_to_bytes	active

DISTRIBUTION

Email—Internal

Name	Org.	Sandia Email Address
Simon Palfery	05334	simon.palfery@sandia.gov
Technical Library	01977	sanddocs@sandia.gov

This page left blank

This page left blank