

## 1、 (exported from wiki) Block Thread Affinity and Priority

```
<page>
  <title>Block Thread Affinity and Priority</title>
  <ns>0</ns>
  <id>3489</id>
  <revision>
    <id>4873</id>
    <parentid>4226</parentid>
    <timestamp>2019-03-12T22:43:51Z</timestamp>
    <contributor>
      <username>777arc</username>
      <id>632</id>
    </contributor>
    <origin>4873</origin>
    <model>wikitext</model>
    <format>text/x-wiki</format>
    <text      bytes="4629"      sha1="pwx0q0pno6nkzeprixgaxmmzhayfmk2"
xml:space="preserve">[[Category:Usage Manual]]
== Block Thread Affinity ==
```

In the thread-per-block scheduler, you can set the block's core affinity. Each block can be pinned to a group cores or be set back to use the standard kernel scheduler.

The implementation is done by adding new functions to the threading section of the gnuradio-runtime library:

```
gr::thread::gr_thread_t get_current_thread_id();
void thread_bind_to_processor(unsigned int n);
void thread_bind_to_processor(const std::vector<unsigned int> &mask);
void thread_bind_to_processor(gr::thread::gr_thread_t thread, unsigned int n);
void      thread_bind_to_processor(gr::thread::gr_thread_t      thread,      const
std::vector<unsigned int> &mask);
void thread_unbind();
void thread_unbind(gr::thread::gr_thread_t thread);
```

The ability to set a thread's affinity to a core or groups of cores is not implemented in the Boost thread library, and so we have made our own portability library. In particular, the `gr::thread::gr_thread_t` type is defined as the thread type for the given system. The other functions are designed to be portable as well by calling the specific implementation for the thread affinity for a particular platform.

There are functions to set a thread to a group of cores. If the thread is not given, the current thread is used. If a single number is passed, only that core is set (this is equivalent to a core mask with just a single value).

Similarly, there are functions to unset the affinity. This practically implements the setting of the thread's affinity to all possible cores. Again, the function that does not take a thread argument unsets the affinity for the current thread.

NOTE: Not available on OSX.

=== GNU Radio Block API ===

Each block has two new data members:

- \* threaded: a boolean value that is true if the block is attached to a thread.
- \* thread: a `gr::thread::gr_thread_t` handle to the block's thread.

A block can set and unset its affinity at any time using the following member functions:

- \* `gr::block::set_processor_affinity(const std::vector<int> &mask)`
- \* `gr::block::unset_processor_affinity()`

Where `mask` is a vector of core numbers to set the thread's affinity to.

The current core affinity can be retrieved using the member function:

- \* `gr::block::processor_affinity()`

When set before the flowgraph is started, the scheduler will set the thread's affinity when it is started. When already running, the block's affinity will be immediately set.

=== Setting Affinity for a `gr::hier_block2` ===

A hierarchical block (`gr::hier_block2`) also has a concept of setting the block thread affinity. Because the hierarchical block itself does no work and just encapsulates a set of blocks, setting the hierarchical block's affinity individually sets all blocks inside it to that affinity setting.

The `gr::hier_block2` class supports the same API interface to the block thread affinity:

```
* gr::hier_block2::set_processor_affinity(const std::vector<int> &mask)
* gr::hier_block2::unset_processor_affinity()
* gr::hier_block2::processor_affinity()
```

Setting and unsetting the affinity does so recursively for every block in the hierarchical block. It is of course possible to individually set the affinity to any block underneath the hierarchical block. However, in this case, note that when asking for the current affinity value using '`processor_affinity()`', the code returns the current processor affinity value of only the first block.

#### === GRC Access ===

GRC supports the setting of the thread core affinity in a block's options. Each block now has a field 'Core Affinity' that accepts a vector (list) of integers and sets the affinity after the block is constructed.

Note that GRC does not provide a callback function for changing the thread core affinity while the flowgraph is running.

#### == Setting Thread Priority ==

Similarly to setting the core affinity for a given block, we can also set the thread priority. This concept adds three new function calls to all blocks:

```
# gr::block::set_thread_priority(int priority): Sets the current thread priority.
# gr::block::active_thread_priority(): Gets the active priority for the thread.
# gr::block::thread_priority(): Gets the stored thread priority.
```

The range of the thread priority might be system dependent, so look to your system/OS documentation. Linux specifies this range in `sched_setscheduler` as a value between 1 and 99 where 1 is the lowest priority and 99 is the highest. POSIX systems can retrieve these min and max values using `sched_get_priority_min` and `sched_get_priority_max` and may only allow 32 distinct values to be set.

NOTE: Not available on Windows or OSX.</text>

<sha1>pwX0q0pno6nkzeprXgaxmmzhayfmk2</sha1>

```
</revision>
</page>
</mediawiki>
```

## 2、 (exported from wiki) Configuration Files

```
<page>
  <title>Configuration Files</title>
  <ns>0</ns>
  <id>3490</id>
  <revision>
    <id>4874</id>
    <parentid>4228</parentid>
    <timestamp>2019-03-12T22:43:56Z</timestamp>
    <contributor>
      <username>777arc</username>
      <id>632</id>
    </contributor>
    <origin>4874</origin>
    <model>wikitext</model>
    <format>text/x-wiki</format>
    <text      bytes="2737"      sha1="e8qpaipnj5q10iynqqrtnz8btlg3xm9"
xml:space="preserve">[[Category:Usage Manual]]
== Configuration / Preference Files ==
```

GNU Radio defines some of its basic behavior through a set of configuration files located in `${prefix}/etc/gnuradio/conf.d`. Different components have different files listed in here for the various properties. These will be read once when starting a GNU Radio application, so updates during runtime will not affect them.

The configuration files use the following format:

```
# Stuff from section 1
[section1]
var1 = value1
var2 = value2 # value of 2

# Stuff from section 2
[section2]
var3 = value3
```

In this file, the hash mark (`#`) indicates a comment and blank lines are ignored. Section labels are defined inside square brackets as a

group distinguisher. All options must be associated with a section name. The options are listed one per line with the option name is given followed by an equals (=) sign and then the value.

All section and option names must not have white spaces. If a value must have white space, then it MUST be put inside quotes. Any quoted value will have its white space preserved and the quotes internally will be stripped. As an example, on Apple desktops, an output device of "Display Audio" is a possible output device and can be set as:

```
[audio_osx]
default_output_device = "Display Audio"
```

The result will pass Display Audio to the audio setup.

The value of an option can be a string or number and retrieved through a few different interfaces. There is a single preference object created when GNU Radio is launched. In Python, you can get this by making a new variable:

```
p = gr.prefs()
```

Similarly, in C++, we get a reference to the object by explicitly calling for the singleton of the object:

```
prefs *p = prefs::singleton();
```

The methods associated with this preferences object are (from class gr::prefs):

```
bool has_section(string section)
bool has_option(string section, string option)
string get_string(string section, string option, string default_val)
bool get_bool(string section, string option, bool default_val)
long get_long(string section, string option, long default_val)
double get_double(string section, string option, double default_val)
```

When setting a Boolean value, we can use 0, 1, "True", "true", "False", "false", "On", "on", "Off", and "off".

All configuration preferences in these files can also be overloaded by an environmental variable. The environmental variable is named based on the section and option name from the configuration file as:

```
GR_CONF_SECTION_OPTION = <value>
```

The "GR\_CONF\_" is a prefix to identify this as a GNU Radio configuration variable and the section and option names are in uppercase. The value is the same format that would be used in the config file itself.</text>

```
<sha1>e8qpaipnj5q10iyngqrtnz8btlg3xm9</sha1>
</revision>
</page>
</mediawiki>
```

### 3、 (exported from wiki) Handling Flowgraphs

```
<page>
<title>Handling Flowgraphs</title>
<ns>0</ns>
<id>3480</id>
<revision>
<id>4863</id>
<parentid>4291</parentid>
<timestamp>2019-03-12T22:42:43Z</timestamp>
<contributor>
<username>777arc</username>
<id>632</id>
</contributor>
<origin>4863</origin>
<model>wikitext</model>
<format>text/x-wiki</format>
<text      bytes="10065"      sha1="rggsk6ltg0cpkpukwarub0b27ji35z"
xml:space="preserve">[[Category:Usage Manual]]
== Operating a Flowgraph ==
```

The basic data structure in GNU Radio is the flowgraph, which represents the connections of the blocks through which a continuous stream of samples flows. The concept of a flowgraph is an acyclic directional graph with one or more source blocks (to insert samples into the flowgraph), one or more sink blocks (to terminate or export samples from the flowgraph), and any signal processing blocks in between.

A program must at least create a GNU Radio 'top\_block', which represents the top-most structure of the flowgraph. The top blocks provide the overall control and hold methods such as 'start,' 'stop,' and 'wait'.

The general construction of a GNU Radio application is to create a

gr\_top\_block, instantiate the blocks, connect the blocks together, and then start the gr\_top\_block. The following program shows how this is done. A single source and sink are used with a FIR filter between them.

```
<syntaxhighlight lang="python">
    from gnuradio import gr, blocks, filter, analog

    class my_topblock(gr.top_block):
        def __init__(self):
            gr.top_block.__init__(self)

            amp = 1
            taps = filter.firdes.low_pass(1, 1, 0.1, 0.01)

            self.src = analog.noise_source_c(analog.GR_GAUSSIAN, amp)
            self.flt = filter.fir_filter_ccf(1, taps)
            self.snk = blocks.null_sink(gr.sizeof_gr_complex)

            self.connect(self.src, self.flt, self.snk)

    if __name__ == "__main__":
        tb = my_topblock()
        tb.start()
        tb.wait()
</syntaxhighlight>
```

The 'tb.start()' starts the data flowing through the flowgraph while the 'tb.wait()' is the equivalent of a thread's 'join' operation and blocks until the gr\_top\_block is done.

An alternative to using the 'start' and 'wait' methods, a 'run' method is also provided for convenience that is a blocking start call; equivalent to the above 'start' followed by a 'wait.'

=== Latency and Throughput ===

By default, GNU Radio runs a scheduler that attempts to optimize throughput. Using a dynamic scheduler, blocks in a flowgraph pass chunks of items from sources to sinks. The sizes of these chunks will vary depending on the speed of processing. For each block, the number of items it can process is dependent on how much space it has in its output buffer(s) and how many items are available on the input buffer(s).

The consequence of this is that often a block may be called with a very

large number of items to process (several thousand). In terms of speed, this is efficient since now the majority of the processing time is taken up with processing samples. Smaller chunks mean more calls into the scheduler to retrieve more data. The downside to this is that it can lead to large latency while a block is processing a large chunk of data.

To combat this problem, the `gr_top_block` can be passed a limit on the number of output items a block will ever receive. A block may get less than this number, but never more, and so it serves as an upper limit to the latency any block will exhibit. By limiting the number of items per call to a block, though, we increase the overhead of the scheduler, and so reduce the overall efficiency of the application.

To set the maximum number of output items, we pass a value into the 'start' or 'run' method of the `gr_top_block`:

```
tb.start(1000)
tb.wait()
or
tb.run(1000)
```

Using this method, we place a global restriction on the size of items to all blocks. Each block, though, has the ability to overwrite this with its own limit. Using the 'set\_max\_noutput\_items(m)' method for an individual block will overwrite the global setting. For example, in the following code, the global setting is 1000 items max, except for the FIR filter, which can receive up to 2000 items.

```
tb.flt.set_max_noutput_items(2000)
tb.run(1000)
```

In some situations, you might actually want to restrict the size of the buffer itself. This can help to prevent a buffer who is blocked for data from just increasing the amount of items in its buffer, which will then cause an increased latency for new samples. You can set the size of an output buffer for each output port for every block.

**WARNING:** This is an advanced feature in GNU Radio and should not be used without a full understanding of this concept as explained below.

To set the output buffer size of a block, you simply call:

```
tb.blk0.set_max_output_buffer(2000)
```

```

tb.blk1.set_max_output_buffer(1, 2000)
tb.start()
print tb.blk1.max_output_buffer(0)
print tb.blk1.max_output_buffer(1)

```

In the above example, all ports of blk0 are set to a buffer size of 2000 in "items" (not bytes), and blk1 only sets the size for output port 1, any and all other ports use the default. The third and fourth lines just print out the buffer sizes for ports 0 and 1 of blk1. This is done after start() is called because the values are updated based on what is actually allocated to the block's buffers.

#### NOTES:

1. Buffer length assignment is done once at runtime (i.e., when run() or start() is called). So to set the max buffer lengths, the set\_max\_output\_buffer calls must be done before this.
2. Once the flowgraph is started, the buffer lengths for a block are set and cannot be dynamically changed, even during a lock()/unlock(). If you need to change the buffer size, you will have to delete the block and rebuild it, and therefore must disconnect and reconnect the blocks.
3. This can affect throughput. Large buffers are designed to improve the efficiency and speed of the program at the expense of latency. Limiting the size of the buffer may decrease performance.
4. The real buffer size is actually based on a minimum granularity of the system. Typically, this is a page size, which is typically 4096 bytes. This means that any buffer size that is specified with this command will get rounded up to the nearest granularity (e.g., page size). When calling max\_output\_buffer(port) after the flowgraph is started, you will get how many items were actually allocated in the buffer, which may be different than what was initially specified.

#### == Reconfiguring Flowgraphs ==

It is possible to reconfigure the flowgraph at runtime. The reconfiguration is meant for changes in the flowgraph structure, not individual parameter settings of the blocks. For example, changing the constant in a gr::blocks::add\_const\_cc block can be done while the flowgraph is running using the 'set\_k(k)' method.

Reconfiguration is done by locking the flowgraph, which stops it from running and processing data, performing the reconfiguration, and then restarting the graph by unlocking it.

The following example code shows a graph that first adds two `gr::analog::noise_source_c` blocks and then replaces the `gr::blocks::add_cc` block with a `gr::blocks::sub_cc` block to then subtract the sources.

```
<syntaxhighlight lang="python">
```

```
from gnuradio import gr, analog, blocks
import time
```

```
class mytb(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self)

        self.src0 = analog.noise_source_c(analog.GR_GAUSSIAN, 1)
        self.src1 = analog.noise_source_c(analog.GR_GAUSSIAN, 1)
        self.add = blocks.add_cc()
        self.sub = blocks.sub_cc()
        self.head = blocks.head(gr.sizeof_gr_complex, 1000000)
        self.snk = blocks.file_sink(gr.sizeof_gr_complex, "output.32fc")

        self.connect(self.src0, (self.add,0))
        self.connect(self.src1, (self.add,1))
        self.connect(self.add, self.head)
        self.connect(self.head, self.snk)
```

```
def main():
    tb = mytb()
    tb.start()
    time.sleep(0.01)

    # Stop flowgraph and disconnect the add block
    tb.lock()
    tb.disconnect(tb.add, tb.head)
    tb.disconnect(tb.src0, (tb.add,0))
    tb.disconnect(tb.src1, (tb.add,1))

    # Connect the sub block and restart
    tb.connect(tb.sub, tb.head)
    tb.connect(tb.src0, (tb.sub,0))
    tb.connect(tb.src1, (tb.sub,1))
    tb.unlock()
```

```
tb.wait()
```

```
if __name__ == "__main__":  
    main()
```

</syntaxhighlight>

During reconfiguration, the maximum `noutput_items` value can be changed either globally using the '`set_max_noutput_items(m)`' on the `gr_top_block` object or locally using the '`set_max_noutput_items(m)`' on any given block object.

A block also has a '`unset_max_noutput_items()`' method that unsets the local max `noutput_items` value so that block reverts back to using the global value.

The following example expands the previous example but sets and resets the max `noutput_items` both locally and globally.

<syntaxhighlight lang="python">

```
from gnuradio import gr, analog, blocks  
import time
```

```
class mytb(gr.top_block):
```

```
    def __init__(self):
```

```
        gr.top_block.__init__(self)
```

```
        self.src0 = analog.noise_source_c(analog.GR_GAUSSIAN, 1)
```

```
        self.src1 = analog.noise_source_c(analog.GR_GAUSSIAN, 1)
```

```
        self.add = blocks.add_cc()
```

```
        self.sub = blocks.sub_cc()
```

```
        self.head = blocks.head(gr.sizeof_gr_complex, 1000000)
```

```
        self.snk = blocks.file_sink(gr.sizeof_gr_complex, "output.32fc")
```

```
        self.connect(self.src0, (self.add,0))
```

```
        self.connect(self.src1, (self.add,1))
```

```
        self.connect(self.add, self.head)
```

```
        self.connect(self.head, self.snk)
```

```
def main():
```

```
    # Start the gr_top_block after setting some max noutput_items.
```

```
    tb = mytb()
```

```
    tb.src1.set_max_noutput_items(2000)
```

```
    tb.start(100)
```

```
    time.sleep(0.01)
```

```

# Stop flowgraph and disconnect the add block
tb.lock()

tb.disconnect(tb.add, tb.head)
tb.disconnect(tb.src0, (tb.add,0))
tb.disconnect(tb.src1, (tb.add,1))

# Connect the sub block
tb.connect(tb.sub, tb.head)
tb.connect(tb.src0, (tb.sub,0))
tb.connect(tb.src1, (tb.sub,1))

# Set new max_noutput_items for the gr_top_block
# and unset the local value for src1
tb.set_max_noutput_items(1000)
tb.src1.unset_max_noutput_items()
tb.unlock()

tb.wait()

if __name__ == "__main__":
    main()
</syntaxhighlight></text>
<sha1>rggsk6ltg0cpkpujkwarub0b27ji35z</sha1>
</revision>
</page>
</mediawiki>

```

## 4、(exported from wiki) Logging

```

<page>
  <title>Logging</title>
  <ns>0</ns>
  <id>5023</id>
  <revision>
    <id>15110</id>
    <parentid>15109</parentid>
    <timestamp>2025-06-03T15:55:37Z</timestamp>
    <contributor>
      <username>MarcusMueller</username>
      <id>10</id>
    </contributor>
    <comment>/* Logging Values */</comment>
    <origin>15110</origin>
    <model>wikitext</model>
  </revision>
</page>

```

```
<format>text/x-wiki</format>
<text      bytes="11359"      sha1="36px2v8r47npvanpolv5tc0ye360ewh"
xml:space="preserve">[[Category:Usage Manual]]
```

This page describes the logging system, as it is in GNU Radio 3.10 and later. For the logging as realized in earlier versions, see [[Legacy\_Logging|Legacy Logging]].

= Usage =

"Log what you think might be useful." Some fundamental change occurred that you think you'd like to know about later on? Log. An error occurred? Definitely log! A setter method got a value that looks strange, but it's not an error, while probably still worth investigating? Log!

GNU Radio's logging system is very flexible and pretty fast. If you think it makes sense to be able to reconstruct what happened later on, that might be something you want to log (maybe with a low priority, see the section on Levels\*\*).

"Never use standard output for logging." GNU Radio's logging system allows for anyone who has special logging needs (e.g. forward all log messages to a central log server, have logs in a graphical environment) to attach to it. Can't do that with `std::cout`, `std::cerr`, `printf`, or Python's `print`.

== Log Message Categories / Levels ==

The GNU Radio logging system is based on [<https://github.com/gabime/spdlog>] and hence exposes the same log levels. Use these to denote the severity of your logging message. For example, use `info` for things that might be interesting for someone observing the operation of your flow graph, use `warn` to notify that something "might reasonably be assumed to be wrong" and that they need to look into it, and `error` if something really is wrong and needs to be fixed.

The log levels are ordered, so when you tell the logging system "I don't care about anything less than a warning", you will not get `info` messages (and anything below in severity).

The levels are:

- \* `trace`: Tracing your code's operation. This is mostly for when you're developing an algorithm and need to log specific values that it calculates
- \* `debug`: Used for debugging purposes, and should not be visible under usual usage.
- \* `info`: Information that does not require immediate attention, but might be of interest to someone observing the operation

- \* `warn`: A warning. Means something "might" be wrong and it's probably worth looking into it.
- \* `error`: An error has occurred. Meaning something definitely is wrong.
- \* `critical`: An error that cannot be recovered from. For all that concerns your block/functional unit, execution cannot continue sensibly.

## == Logging in Python ==

Within a block, the block holds a `logger` member:

```
<syntaxhighlight lang="python">import numpy as np
from gnuradio import gr
class my_block(gr.sync_block):
    def __init__(self, foo = 1, bar = False):
        gr.sync_block.__init__(
            self,
            name="My fancy block",
            in_sig=[np.float32],
            out_sig=[np.float32])
        self.foo = foo
        self.bar = bar
    def work(self, input_items, output_items):
        if max(input_items) > self.foo:
            self.logger.warn(f'maximum of input {max(input_items)} larger than
foo ({self.foo})')
            output_items[0][:] = input_items[0][:]
            return len(output_items[0])</syntaxhighlight>
or outside a block
```

```
<syntaxhighlight lang="python">from gnuradio import gr
```

```
...
```

```
logging_mc_logface = gr.logger("prime searcher")
```

```
...
```

```
if number % potential_factor == 0:
    logging_mc_logface.info(f'{number} is not a prime, as it can be divided by
{potential_factor}')
    return False</syntaxhighlight>
```

As you can see, we're using the fact that Python has [https://docs.python.org/3/reference/lexical\_analysis.html#formatted-string-literals] formatted string literals] (`f"something {} more"`) to embed values in log messages. The Python [https://docs.python.org/3/library/string.html#format-string-syntax] format string syntax]

`f"Some string {python expression}"` defines how to embed values.

It's generally very straightforward: A formatted string is `f"content"`, where `content` can contain `{}`, which in turn contain an expression (e.g., a variable name like `number`, or also formatting instructions):

```
<syntaxhighlight lang="python">f"The received int32 is {rx_value:b}"</syntaxhighlight>
```

will apply the `b` format to the variable `rx_value`. `b` prints a binary representation of the value. `x` prints a hexadecimal representation, whereas `e` can be used on floating point numbers to express them in exponential notation (`f"{1/90000:e}"` becomes `1.11111e-05`). For more information on formats, see the [<https://docs.python.org/3/library/string.html#formatspec> format string syntax].

=== Setting Log Level ===

For GNU Radio [[https://github.com/gnuradio/gnuradio/blob/v3.10.5.0/gnuradio-runtime/python/gnuradio/gr/qa\\_python\\_logging.py#v3.10.5+](https://github.com/gnuradio/gnuradio/blob/v3.10.5.0/gnuradio-runtime/python/gnuradio/gr/qa_python_logging.py#v3.10.5+)], log level can be changed using 

```
<syntaxhighlight lang="python">gr.logging().set_default_level(gr.log_levels.warn)</syntaxhighlight>
```

== Logging in C++ ==

If you're writing a block, you will already find `d_logger` as a member of your block. It is a smart pointer to a logger object. This logger is already set up correctly to log messages with a prefix that quotes your block's alias.

The different log levels are member functions of the logger, so you can log an error using `d_logger->error( ... )` and a debug message using `d_logger->debug(...)`.

So, in case you need to log a warning, e.g., when a value is invalid and has been replaced, you would follow this example

```
<syntaxhighlight lang="c++">void my_block::gain_msg_handler(const pmt::pmt_t& message) {
    constexpr float min_gain = 1.0;
    d_gain = pmt::to_double(message);
}
```

```

        if(d_gain <= 0.0) {
            d_logger->warn("Attempted to set non-positive gain {}; clipping to {}. ",
d_gain, min_gain);
            d_gain = min_gain;
        }
}

```

Outside of blocks, you will have to create your own logger. You simply instantiate a `gr::logger` with the name of the logging "thing" as constructor parameter. An example of that is show below:

```

<syntaxhighlight lang="c++">#include <gnuradio/logger.h>

```

```

class my_thing
{
private:
    gr::logger _logger;

public:
    my_thing(const std::string& name)
        : _logger("my thing " + name)
    {
        _logger.info("constructed");
    }
    ~my_thing() { _logger.warn("I don't like being destructed!"); }
};

```

```

int main() { my_thing thing("gizmo"); }

```

=== Logging Values ===

Since `gr::logger` is a wrapper around `spdlog`'s logging facilities, it offers the same [https://fmt.dev `fmt`]-based string formatting.

An example of that was shown in `gain_msg_handler` above:

```

<syntaxhighlight lang="c++">d_logger->warn("Attempted to set non-positive gain {};
clipping to {}. ", d_gain, min_gain);

```

The `{}` get replaced (in their respective order) by the arguments to the logging function.

You can also set the format of fields by using a format specifier syntax very close to Python's "format string" syntax. For the actual full syntax definition, refer to [https://fmt.dev/latest/syntax.html `fmt`'s specification].

This being C++, the type of the argument specifies already what type of conversion to use (unlike e.g. libc's `printf`, where you need to tell `printf` that an integer needs to be interpreted as an integer), so in most cases, you do not need to specify a format. However, sometimes enforcing a specific notation can be helpful for interpretation.

The format specification follows a colon `<code>:</code>`. Common examples of things that you might want:

\* "Formatting floating point in scientific notation": `<code>logger.warn(&quot;Amplitude {<code>:e</code> out of range<code>;, value);</code>` Here, `<code>e</code>` causes the number passed in to be represented in scientific notation. Use `<code>g</code>` instead to only apply for large numbers.

\* "Formatting a value with fixed width": `<code>logger.warn(&quot;Count {<code>+15</code> out of range<code>;, count);</code>` Here, the format specification consists of a space `<code></code>`, which is the "fill character" to be used to pad to the specified length of `<code>15</code>` characters, and `<code>+</code>` denotes that the sign should always be printed (not only for negative numbers). Works with numbers just as well as with strings.

\* "Formatting data in hexadecimal": `<code>logger.info(&quot;Got data {<code>:X</code>&quot;;, data_as_integer_of_any_width)</code>`: Print using uppercase hexadecimal (`<code>13CAFE37</code>`). Use `<code>:x</code>` for lower case, and use `<code>:#x</code>` or `<code>:#X</code>` to get a `<code>0x</code>` prefix.

\* "Formatting integers as binary": `<code>logger.warn(&quot;Value {<code>:b</code> is not a valid code word. Corrected to {<code>:b</code>.&quot;;, rx_word, code_word);</code>`. Use `<code>:#b</code>` to get a `<code>0b</code>` prefix.

\* "Formatting PMTs": `<code>logger.warn("Value {} is not a valid command.", message);</code>`.

=== Performance Considerations ===

==== Logger Construction Cost ====

While constructing a logger is not very expensive, it's still not something you want to do within a hot loop, as it requires inter-thread coordination. Ideally, you want your entity to hold onto a logger object once it's been created. (This is solved through the `<code>d_logger</code>` member if you're writing a GNU Radio block. Elsewhere, define your own member if in a class context.)

==== Logging Cost – Disabled Logging ====

It's possible to disable logging at compile time (it's not recommended, logging is universally quite useful). Or, that the minimum log level was set to higher than a

message's log level, so that, for example `d_logger->trace(&quot;val: {+15g}&quot;);` shouldn't actually do anything.

There "is" an integer comparison at run time that will be incurred by having this logging in the code. Notice, however, that modern CPUs tend to predict that to be false if it happens more often, and even if they didn't, such a comparison is quite cheap (and speculative execution would have continued after the logging in the meantime). Benchmarking showed negligible overhead unless used in tight arithmetic loops. If you "are" in the "rare" case that you want to log from such a compute-intense place, you would want to wrap the logging in some preprocessor `#if` magic (or figure out a less invasive place to log).

The format string evaluation "only" happens when the logging actually takes place. So, even complex logging formats (aside from the code size) do not affect the runtime. (It might, however, affect life time of some values, so it's not free of side effects.)

==== Logging Cost – Active Logging =====

`fmt` is seriously [<https://github.com/fmtlib/fmt#speed-tests> fast]. In fact, it beats things like `std::stringstream &lt;&lt;` by orders of magnitude. Especially when logging things that should not happen often (`warn`, `error` or `critical`), it's probably a good idea to rather log more useful information than less.

<sha1>36px2v8r47npvanpolv5tc0ye360ewh</sha1>

</revision>

</page>

</mediawiki>

## 5、 (exported from wiki) Message Passing

<page>

<title>Message Passing</title>

<ns>0</ns>

<id>3481</id>

<revision>

<id>14248</id>

<parentid>12916</parentid>

<timestamp>2024-05-06T13:49:00Z</timestamp>

<contributor>

<username>Hipple</username>

<id>1261</id>

</contributor>

<minor/>

<comment>/\* PDUs \*/</comment>

```
<origin>14248</origin>
<model>wikitext</model>
<format>text/x-wiki</format>
<text      bytes="20873"      sha1="84egxhayw9x4houqiv6k7pyh7u7vt3s"
xml:space="preserve">[[Category:Usage Manual]]
== Introduction ==
```

Message passing is a mechanism to asynchronously communicate events and data between blocks. Unlike streams and tags, messages can be passed to upstream blocks.

== Background ==

GNU Radio was originally a streaming system with no other mechanism to pass data between blocks. Streams of data are a model that work well for samples, bits, etc., but are not really the right mechanism for control data, metadata, or packet structures (at least at some point in the processing chain).

We solved part of this problem by introducing the tag stream (see [\[\[Stream Tags\]\]](#)). This is a parallel stream to the data streaming. The difference is that tags are designed to hold metadata and control information. Tags are specifically associated with a particular sample in the data stream and flow downstream alongside the data. This model allows other blocks to identify that an event or action has occurred or should occur on a particular item. The major limitation is that the tag stream is really only accessible inside a work function and only flows in one direction. Its benefit is that it is isosynchronous with the data.

We want a more general message passing system for a couple of reasons. The first is to allow blocks downstream to communicate back to blocks upstream. The second is to allow an easier way for us to communicate back and forth between external applications and GNU Radio. GNU Radio's message passing interface handles these cases, although it does so on an asynchronous basis.

The message passing interface heavily relies on Polymorphic Types (PMTs) in GNU Radio. For further information about these data structures, see the page [\[\[Polymorphic Types \(PMTs\)\]\]](#).

[\[\[File:Gnuradio\\_pdu\\_illustration.png|550px|Gnuradio\\_pdu\\_illustration.png\]\]](#)

== Message Passing API ==

The message passing interface is designed into the `gr::basic_block`, which is the parent class for all blocks in GNU Radio. Each block has a set of message queues to hold incoming messages and can post messages to the message queues of other blocks. The blocks also distinguish between input and output ports.

A block has to declare its input and output message ports in its constructor. The message ports are described by a name, which is in practice a PMT symbol (i.e., an interned string). The API calls to register a new port are:

```
void message_port_register_in(pmt::pmt_t port_id)
void message_port_register_out(pmt::pmt_t port_id)
```

In Python:

```
self.message_port_register_in(pmt.intern("port name"))
self.message_port_register_out(pmt.intern("port name"))
```

The ports are now identifiable by that port name. Other blocks who may want to post or receive messages on a port must subscribe to it. When a block has a message to send, they are published on a particular port using the following API:

```
void message_port_pub(pmt::pmt_t port_id, pmt::pmt_t msg);
```

In Python:

```
self.message_port_pub(pmt.intern("port name"), <pmt message>)
```

Subscribing is usually done in the form of connecting message ports as part of the flowgraph, as discussed later. Internally, when message ports are connected, the `gr::basic_block::message_port_sub` method is called.

Any block that has a subscription to another block's output message port will receive the message when it is published. Internally, when a block publishes a message, it simply iterates through all blocks that have subscribed and uses the `gr::basic_block::_post` method to send the message to that block's message queue.

=== Message Handler Functions ===

A subscriber block must declare a message handler function to process

the messages that are posted to it. After using the `gr::basic_block::message_port_register_in` to declare a subscriber port, we must then bind this port to the message handler.

Starting in GNU Radio 3.8<sup>1</sup> using C++11 we do that using a lambda function:

```
set_msg_handler(pmt::pmt_t port_id,
    [this](const pmt::pmt_t& msg) { message_handler_function(msg); });
```

In Python:

```
self.set_msg_handler(pmt.intern("port name"), <msg handler function>)
```

When a new message is pushed onto a port's message queue, it is this function that is used to process the message.

The 'port\_id' is the same PMT as used when registering the input port. The 'block\_class::message\_handler\_function' is the member function of the class designated to handle messages to this port.

The prototype for all message handling functions is:

```
void block_class::message_handler_function(const pmt::pmt_t& msg);
```

In Python the equivalent function would be:

```
def handle_msg(self, msg):
```

We give examples of using this below.

=== Connecting Messages through the Flowgraph ===

From the flowgraph level, we have instrumented a `gr::hier_block2::msg_connect` method to make it easy to subscribe blocks to other blocks' messages. Assume that the block "src" has an output message port named "pdus" and the block "dbg" has an input port named "print". The message connection in the flowgraph (in Python) looks like the following:

```
self.tb.msg_connect(src, "pdus", dbg, "print")
```

All messages published by the "src" block on port "pdus" will be received by "dbg" on port "print". Note here how we are just using strings to define the ports, not PMT symbols. This is a convenience to the user to be able to more easily type in the port names (for

reference, you can create a PMT symbol in Python using the `pmt::intern` function as `pmt.intern("string")`).

Users can also query blocks for the names of their input and output ports using the following API calls:

```
pmt::pmt_t message_ports_in();  
pmt::pmt_t message_ports_out();
```

The return value for these are a PMT vector filled with PMT symbols, so PMT operators must be used to manipulate them.

Each block has internal methods to handle posting and receiving of messages. The `gr::basic_block::_post` method takes in a message and places it into its queue. The publishing model uses the `gr::basic_block::_post` method of the blocks as the way to access the message queue. So the message queue of the right name will have a new message. Posting messages also has the benefit of waking up the block's thread if it is in a wait state. So if idle, as soon as a message is posted, it will wake up and call the message handler.

== Posting from External Sources ==

An important feature of the message passing architecture is how it can be used to take in messages from an external source. We can call a block's `gr::basic_block::_post` method directly and pass it a message. So any block with an input message port can receive messages from the outside in this way.

The following example uses a `pdu_to_tagged_stream` block as the source block to a flowgraph. Its purpose is to wait for messages as PDUs posted to it and convert them to a normal stream. The payload will be sent on as a normal stream while the meta data will be decoded into tags and sent on the tagged stream.

So if we have created a "src" block as a PDU to stream, it has a "pdus" input port, which is how we will inject PDU messages into the flowgraph. These PDUs could come from another block or flowgraph, but here, we will create and insert them by hand.

```
port = pmt.intern("pdus")  
msg = pmt.cons(pmt.PMT_NIL, pmt.make_u8vector(16, 0xFF))  
src.to_basic_block()._post(port, msg)
```

The PDU's metadata section is empty, hence the `pmt::PMT_NIL` object. The payload is now just a simple vector of 16 bytes of all 1's. To post the message, we have to access the block's `gr::basic_block` class, which we do using the `gr::basic_block::to_basic_block` method and then call the `gr::basic_block::_post` method to pass the PDU to the right port.

All of these mechanisms are explored and tested in the QA code of the file `qa_pdu.py`.

There are some examples of using the message passing infrastructure through GRC in:

`gr-blocks/examples/msg_passing`

== Using Messages as Commands ==

One important use of messages is to send commands to blocks. Examples for this include:

- \* `gr::qtgui::freq_sink_c`: The scaling of the frequency axis can be changed by messages
- \* `gr::uhd::usrp_source` and `gr::uhd::usrp_sink`: Many transceiver-related settings can be manipulated through command messages, such as frequency, gain and LO offset
- \* `gr::digital::header_payload_demux`, which receives an acknowledgement from a header parser block on how many payload items there are to process

There is no special PMT type to encode commands, however, it is strongly recommended to use one of the following formats:

- \* `pmt::cons(KEY, VALUE)`: This format is useful for commands that take a single value. Think of KEY and VALUE as the argument name and value, respectively. For the case of the QT GUI Frequency Sink, KEY would be "freq" and VALUE would be the new center frequency in Hz.
- \* `pmt::dict((KEY1: VALUE1), (KEY2: VALUE2), ...)`: This is basically the same as the previous format, but you can provide multiple key/value pairs. This is particularly useful when a single command takes multiple arguments which can't be broken into multiple command messages (e.g., the USRP blocks might have both a timestamp and a center frequency in a command message, which are closely associated).

In both cases, all KEYs should be `pmt::symbols` (i.e. strings). VALUEs can be whatever the block requires.

It might be tempting to deviate from this format, e.g. the QT Frequency sink could

simply take a float value as a command message, and it would still work fine. However, there are some very good reasons to stick to this format:

- \* Interoperability: The more people use the standard format, the more likely it is that blocks from different sources can work together
- \* Inspectability: A message debug block will display more useful information about a message if it's containing both a value and a key
- \* Intuition: This format is pretty versatile and unlikely to create situations where it is not sufficient (especially considering that values are PMTs themselves). As a counterexample, using positional arguments (something like "the first argument is the frequency, the second the gain") is easily forgotten, or changed in one place and not another, etc.

### == Code Examples ==

Note, in addition to the C++ or Python code below, if adding message passing to a block, you need to edit the block's YAML file as well. Add the corresponding input or output entries (ensuring the domain is set to "message"). You do not need to specify a dtype. You should also ensure that the label value corresponds to the registered port name. See [\[\[YAML\\_GRC#Inputs\\_and\\_Outputs\\_.28optional.29 | here\]\]](#) for more details.

### === C++ ===

The following are snippets of code from blocks currently in GNU Radio that take advantage of message passing. We will be using `message_debug` and `tagged_stream_to_pdu` below to show setting up both input and output message passing capabilities.

The `message_debug` block is used for debugging the message passing system. It describes two input message ports: "print" and "store". The "print" port simply prints out all messages to standard out while the "store" port keeps a list of all messages posted to it. The "store" port works in conjunction with a `message_debug::get_message(size_t i)` call that allows us to retrieve message `i` afterward.

The constructor of this block looks like this:

```
<syntaxhighlight lang="cpp">
{
    message_port_register_in(pmt::mp("print"));
    set_msg_handler(pmt::mp("print"),
        [this](const pmt::pmt_t& msg) { print(msg); });
}
```

```

    message_port_register_in(pmt::mp("store"));
    set_msg_handler(pmt::mp("store"),
        [this](const pmt::pmt_t& msg) { store(msg); });
}
</syntaxhighlight>

```

The two message input ports are registered by their respective names. We then use these registered names within the `gr::basic_block::set_msg_handler` function to assign a callback function specific to each port.

At this point the two functions are in charge of handling all the messages passed to them.

Their definition resides in the block's private implementation class (i.e. `message_debug_impl::print` and `message_debug_impl::store`).

Below the "print" function is shown for reference:

```

<syntaxhighlight lang="cpp">
void
message_debug_impl::print(const pmt::pmt_t& msg)
{
    std::cout << "***** MESSAGE DEBUG PRINT *****\n";
    pmt::print(msg);
    std::cout << "*****\n";
}
</syntaxhighlight>

```

The function simply takes in the PMT message and prints it. The method `pmt::print` is a function in the PMT library to print the PMT in a friendly and (mostly) pretty manner.

The `tagged_stream_to_pdu` block only defines a single output message port. In this case, its constructor contains the line:

```

<syntaxhighlight lang="cpp">
{
    message_port_register_out(pdu_port_id);
}
</syntaxhighlight>

```

So we are only creating a single output port where "pdu\_port\_id" is defined in the file `pdu.h` as "pdus".

This block's purpose is to take in a stream of samples along with

stream tags and construct a predefined PDU message from it. In GNU Radio, we define a PDU as a PMT pair of (metadata, data). The metadata describes the samples found in the data portion of the pair. Specifically, the metadata can contain the length of the data segment and any other information (sample rate, etc.). The PMT vectors know their own length, so the length value is not actually necessary unless useful for purposes down the line. The metadata is a PMT dictionary while the data segment is a PMT uniform vector of either bytes, floats, or complex values.

Once a PDU message is ready, the block calls its `tagged_stream_to_pdu_impl::send_message` function, as shown below:

```
<syntaxhighlight lang="cpp">
```

```
void
tagged_stream_to_pdu_impl::send_message()
{
    if(pmt::length(d_pdu_vector) != d_pdu_length) {
        throw std::runtime_error("msg length not correct");
    }

    pmt::pmt_t msg = pmt::cons(d_pdu_meta,
                               d_pdu_vector);
    message_port_pub(pdu_port_id, msg);

    d_pdu_meta = pmt::PMT_NIL;
    d_pdu_vector = pmt::PMT_NIL;
    d_pdu_length = 0;
    d_pdu_remain = 0;
    d_inpdu = false;
}
```

```
</syntaxhighlight>
```

This function does a bit of checking to make sure the PDU is OK as well as some cleanup in the end. But it is the line where the message is published that is important to this discussion. Here, the block posts the PDU message to any subscribers by calling `gr::basic_block::message_port_pub` publishing method.

There is similarly a `pdu_to_tagged_stream` block that essentially does the opposite. It acts as a source to a flowgraph and waits for PDU messages to be posted to it on its input port "pdus". It extracts the metadata and data and processes them. The metadata dictionary is split up into key:value pairs and stream tags are created out of them. The data is then converted into an output stream of items and

passed along. The next section describes how PDUs can be passed into a flowgraph using the pdu\_to\_tagged\_stream block.

==== Python ====

A Python Block example:

```
<syntaxhighlight lang="python">
from gnuradio import gr
import pmt

class msg_block(gr.basic_block):
    def __init__(self):
        gr.basic_block.__init__(
            self,
            name="msg_block",
            in_sig=None,
            out_sig=None)

        self.message_port_register_out(pmt.intern('msg_out'))
        self.message_port_register_in(pmt.intern('msg_in'))
        self.set_msg_handler(pmt.intern('msg_in'), self.handle_msg)

    def handle_msg(self, msg):
        self.message_port_pub(pmt.intern('msg_out'),          pmt.intern('message
received!'))
</syntaxhighlight>
```

== Flowgraph Example ==

Here's a simple example of a flow graph using both streaming and messages:

[[File:strobe.grc.png|550px|strobe.grc.png]]

There are several interesting things to point out. First, there are two source blocks, which both output items at regular intervals, one every 1000 and one every 750 milliseconds. Dotted lines denote connected message ports, as opposed to solid lines, which denote connected streaming ports. In the top half of the flow graph, we can see that it is, in fact, possible to switch between message passing and streaming ports, but only if the type of the PMTs matches the type of the streaming ports (in this example, the pink color of the streaming ports denotes bytes, which means the PMT should be a u8vector if we want to stream the same data we sent as PMT).

Another interesting fact is that we can connect more than one message output port to a

single message input port, which is not possible with streaming ports. This is due to the "asynchronous" nature of messages: The receiving block will process all messages whenever it has a chance to do so, and not necessarily in any specific order. Receiving messages from multiple blocks simply means that there might be more messages to process.

What happens to a message once it was posted to a block? This depends on the actual block implementation, but there are two possibilities:

- 1) A message handler is called, which processes the message immediately.<br />
- 2) The message is written to a FIFO buffer, and the block can make use of it whenever it likes, usually in the work function.

For a block that has both message ports and streaming ports, any of these two options is OK, depending on the application. However, we strongly discourage the processing of messages inside of a work function and instead recommend the use of message handlers. Using messages in the work function encourages us to block in work waiting for a message to arrive. This is bad behavior for a work function, which should never block. If a block depends upon a message to operate, use the message handler concept to receive the message, which may then be used to inform the block's actions when the work function is called. Only on specially, well-identified occasions should we use method 2 above in a block.

With a message passing interface, we can write blocks that don't have streaming ports, and then the work function becomes useless, since it's a function that is designed to work on streaming items. In fact, blocks that don't have streaming ports usually don't even have a work function.

=== Protocol Data Units (PDUs) ===

In the previous flow graph, we have a block called "PDU to Tagged Stream". A PDU (protocol data unit) in GNU Radio has a special PMT type, it is a pair of a dictionary (on CAR) and a uniform vector type. See `[[Polymorphic_Types_(PMTs)#Pairs]]`. So, this would yield a valid PDU, with no metadata and 10 zeros as stream data:

```
<pre>pdu = pmt.cons(pmt.make_dict(), pmt.make_u8vector(10, 0))</pre>
```

The key/value pairs in the dictionary are then interpreted as key/value pairs of stream tags.

== Flowgraph Example: Chat Application ==

Let's build an application that uses message passing. A chat program is an ideal use case, since it waits for the user to type a message, and then sends it. Because of that, no `[[Throttle]]` block is needed.

Create the following flowgraph and save it as 'chat\_app2.grc':

[[File:Chat\_app2\_fg.png]]

The ZMQ Message blocks have an Address of 'tcp://127.0.0.1:50261'. Typing in the [[QT GUI Message Edit Box]] will send the text once the Enter key is pressed. Output is on the terminal screen where gnuradio-companion was started.

If you want to talk to another user (instead of just yourself), you can create an additional flowgraph with a different name such as 'chat\_app3.grc'. Then change the ZMQ port numbers as follows:

- \* chat\_app2
- \*\* ZMQ PUSH Sink: tcp://127.0.0.1:50261
- \*\* ZMQ PULL Source: tcp://127.0.0.1:50262

- \* chat\_app3
- \*\* ZMQ PUSH Sink: tcp://127.0.0.1:50262
- \*\* ZMQ PULL Source: tcp://127.0.0.1:50261

When using GRC, doing a Generate and/or Run creates a Python file with the same name as the .grc file. You can execute the Python file without running GRC again.

For testing this system we will use two processes, so we will need two terminal windows.

Terminal 1:

- \* since you just finished building the chat\_app3 flowgraph, you can just do a Run.

Terminal 2:

Open another terminal window.

- \* change to whatever directory you used to generate the flowgraph for chat\_app2.
- \* execute the following command:

```
python3 -u chat_app2.py
```

Typing in the Message Edit Box for chat\_app2 should be displayed on the Terminal 1 screen (chat\_app3) and vice versa.

To terminate each of the processes cleanly, click on the 'X' in the upper corner of the GUI rather than using Control-C.

----

<sup>1</sup> In old GNU Radio 3.7, we used Boost's 'bind' function:

```
set_msg_handler(pmt::pmt_t port_id,  
boost::bind(&block_class::message_handler_function, this, _1));}}
```

The 'this' and '\_1' are standard ways of using the Boost bind function to pass the 'this' pointer as the first argument to the class (standard OOP practice) and the \_1 is an indicator that the function expects 1 additional argument.

```
[[Category:Guided Tutorials]]</text>  
<sha1>84egxhayw9x4houqiv6k7pyh7u7vt3s</sha1>  
</revision>  
</page>  
</mediawiki>
```

## 6、 (exported from wiki) Metadata Information

```
<page>  
<title>Metadata Information</title>  
<ns>0</ns>  
<id>3479</id>  
<revision>  
<id>7025</id>  
<parentid>4867</parentid>  
<timestamp>2020-05-11T23:07:18Z</timestamp>  
<contributor>  
<username>L0uisc</username>  
<id>891</id>  
</contributor>  
<minor/>  
<comment>fixed "extra_str" in call to blocks.file_meta_sink() in final code  
block to read "extras_str"</comment>  
<origin>7025</origin>  
<model>wikitext</model>  
<format>text/x-wiki</format>  
<text bytes="14485" sha1="79bbsfgjz078sumck0ea80hkoe1kuow"  
xml:space="preserve">[[Category:Usage Manual]]  
== Introduction ==
```

Metadata files have extra information in the form of headers that carry metadata about the samples in the file. Raw, binary files carry no extra information and must be handled delicately. Any changes in the system state such as a receiver's sample rate or frequency are not conveyed with the data in the file itself. Headers solve this problem.

We write metadata files using `gr::blocks::file_meta_sink` and read metadata files using `gr::blocks::file_meta_source`.

Metadata files have headers that carry information about a segment of data within the file. The header structure is described in detail in the next section. A metadata file always starts with a header that describes the basic structure of the data. It contains information about the item size, data type, if it's complex, the sample rate of the segment, the time stamp of the first sample of the segment, and information regarding the header size and segment size.

The first static portion of the header file contains the following information.

- \* `version`: (char) version number (usually set to `METADATA_VERSION`)
- \* `rx_rate`: (double) Stream's sample rate
- \* `rx_time`: (pmt::pmt\_t pair - (uint64\_t, double)) Time stamp (format from UHD)
- \* `size`: (int) item size in bytes - reflects vector length if any
- \* `type`: (int) data type (enum below)
- \* `cplx`: (bool) true if data is complex
- \* `strt`: (uint64\_t) start of data relative to current header
- \* `bytes`: (uint64\_t) size of following data segment in bytes

An optional extra section of the header stores information in any received tags. The two main tags associated with headers are:

- \* `rx_rate`: the sample rate of the stream.
- \* `rx_time`: the time stamp of the first item in the segment.

These tags were inspired by the UHD tag format.

The header gives enough information to process and handle the data. One cautionary note, though, is that the data type should never change within a file. There should be very little need for this, because GNU Radio blocks can only set the data type of their IO signatures in the constructor, so changes in the data type afterward will not be recognized.

We also have an extra header segment that is optional. This can be loaded up at the beginning by the user specifying some extra metadata that should be transmitted along with the data. It also grows whenever it sees a stream tag, so the dictionary will contain any key:value pairs out of tags from the flowgraph.

### === Types of Metadata Files ===

GNU Radio currently supports two types of metadata files:

- # inline: headers are inline with the data in the same file.
- # detached: headers are in a separate header file from the data.

The inline method is the standard version. When a detached header is used, the headers are simply inserted back-to-back in the detached header file. The dat file, then, is the standard raw binary format with no interruptions in the data.

### === Updating Headers ===

While there is always a header that starts a metadata file, they are updated throughout as well. There are two events that trigger a new header. We define a segment as the unit of data associated with the last header.

The first event that will trigger a new header is when enough samples have been written for the given segment. This number is defined as the maximum segment size and is a parameter we pass to the `file_meta_sink`. It defaults to 1 million items (items, not bytes). When that number of items is reached, a new header is generated and a new segment is started. This makes it easier for us to manipulate the data later and helps protect against catastrophic data loss.

The second event to trigger a new segment is if a new tag is observed. If the tag is a standard tag in the header, the header value is updated, the header and current extras are written to file, and the segment begins again. If a tag from the extras is seen, the value associated with that tag is updated; and if a new tag is seen, a new key:value pair are added to the extras dictionary.

When new tags are seen, we generate a new segment so that we make sure that all samples in that segment are defined by the header. If the sample rate changes, we create a new segment where all of the new samples are at that new rate. Also, in the case of UHD devices, if a segment loss is observed, it will generate a new timestamp as a tag of 'rx\_time'. We create a new file segment that reflects this change to keep the sample times exact.

### === Implementation ===

Metadata files are created using `gr::blocks::file_meta_sink`. The default behavior is to create a single file with inline headers as metadata. An option can be set to switch to detached header mode.

Metadata files are read into a flowgraph using `gr::blocks::file_meta_source`. This source reads a metadata file, inline by default with a settable option to use detached headers. The data from the segments is converted into a standard streaming output. The 'rx\_rate' and 'rx\_time' and all key:value pairs in the extra header are converted into tags and added to the stream tags interface.

### == Structure ==

The file metadata consists of a static mandatory header and a dynamic optional extras header. Each header is a separate PMT dictionary. Headers are created by building a PMT dictionary (`pmt::make_dict`) of key:value pairs, then the dictionary is serialized into a string to be written to file. The header is always the same length that is predetermined by the version of the header (this must be known already). The header will then indicate if there is extra data to be extracted as a separate serialized dictionary.

To work with the PMTs for creating and extracting header information, we use PMT operators. For example, we create a simplified version of the header in C++ like this:

```
<syntaxhighlight lang="cpp">
const char METADATA_VERSION = 0x0;
pmt::pmt_t header;
header = pmt::make_dict();
header = pmt::dict_add(header, pmt::mp("version"),
pmt::mp(METADATA_VERSION));
header = pmt::dict_add(header, pmt::mp("rx_rate"), pmt::mp(samp_rate));
std::string hdr_str = pmt::serialize_str(header);
</syntaxhighlight>
```

The call to `pmt::dict_add` adds a new key:value pair to the dictionary. Notice that it both takes and returns the 'header' variable. This is because we are actually creating a new dictionary with this function, so we just assign it to the same variable.

The 'mp' functions are convenience functions provided by the PMT library. They interpret the data type of the value being inserted and call the correct 'pmt::from\_xxx' function. For more direct control over the data type, see PMT functions in pmt.h, such as pmt::from\_uint64 or pmt::from\_double.

We finish this off by using pmt::serialize\_str to convert the PMT dictionary into a specialized string format that makes it easy to write to a file.

The header is always METADATA\_HEADER\_SIZE bytes long and a metadata file always starts with a header. So to extract the header from a file, we need to read in this many bytes from the beginning of the file and deserialize it. An important note about this is that the deserialize function must operate on a std::string. The serialized format of a dictionary contains null characters, so normal C character arrays (e.g., 'char \*s') get confused.

Assuming that 'std::string str' contains the full string as read from a file, we can access the dictionary in C++ like this:

```
<syntaxhighlight lang="cpp">
pmt::pmt_t hdr = pmt::deserialize_str(str);
if(pmt::dict_has_key(hdr, pmt::string_to_symbol("strt"))) {
    pmt::pmt_t r = pmt::dict_ref(hdr, pmt::string_to_symbol("strt"), pmt::PMT_NIL);
    uint64_t seg_start = pmt::to_uint64(r);
    uint64_t extra_len = seg_start - METADATA_HEADER_SIZE;
}
</syntaxhighlight>
```

This example first deserializes the string into a PMT dictionary again. This will throw an error if the string is malformed and cannot be deserialized correctly. We then want to get access to the item with key 'strt'. As the next subsection will show, this value indicates at which byte the data segment starts. We first check to make sure that this key exists in the dictionary. If not, our header does not contain the correct information and we might want to handle this as an error.

Assuming the header is properly formatted, we then get the particular item referenced by the key 'strt'. This is a uint64\_t, so we use the PMT function to extract and convert this value properly. We now know if we have an extra header in the file by looking at the difference between 'seg\_start' and the static header size, METADATA\_HEADER\_SIZE. If the 'extra\_len' is greater than 0, we know we

have an extra header that we can process. Moreover, this also tells us the size of the serialized PMT dictionary in bytes, so we can easily read this many bytes from the file. We can then deserialize and parse this header just like the first.

#### === Header Information ===

The header is a PMT dictionary with a known structure. This structure may change, but we version the headers, so all headers of version X must be the same length and structure. As of now, we only have version 0 headers, which look like the following:

- \* version: (char) version number (usually set to METADATA\_VERSION)
- \* rx\_rate: (double) Stream's sample rate
- \* rx\_time: (pmt::pmt\_t pair - (uint64\_t, double)) Time stamp (format from UHD)
- \* size: (int) item size in bytes - reflects vector length if any.
- \* type: (int) data type (enum below)
- \* cplx: (bool) true if data is complex
- \* strt: (uint64\_t) start of data relative to current header
- \* bytes: (uint64\_t) size of following data segment in bytes

The data types are indicated by an integer value from the following enumeration type:

```
enum gr_file_types {  
    GR_FILE_BYTE=0,  
    GR_FILE_CHAR=0,  
    GR_FILE_SHORT=1,  
    GR_FILE_INT,  
    GR_FILE_LONG,  
    GR_FILE_LONG_LONG,  
    GR_FILE_FLOAT,  
    GR_FILE_DOUBLE,  
};
```

#### === Extras Information ===

The extras section is an optional segment of the header. If 'strt' == METADATA\_HEADER\_SIZE, then there is no extras. Otherwise, it is simply a PMT dictionary of key:value pairs. The extras header can contain anything and can grow while a program is running.

We can insert extra data into the header at the beginning if we wish. All we need to do is use the pmt::dict\_add function to insert

our hand-made metadata. This can be useful to add our own markers and information.

The main role of the extras header, though, is as a container to hold any stream tags. When a stream tag is observed coming in, the tag's key and value are added to the dictionary. Like a standard dictionary, any time a key already exists, the value will be updated. If the key does not exist, a new entry is created and the new key:value pair are added together. So any new tags that the file metadata sink sees will add to the dictionary. It is therefore important to always check the 'strt' value of the header to see if the length of the extras dictionary has changed at all.

When reading out data from the extras, we do not necessarily know the data type of the PMT value. The key is always a PMT symbol, but the value can be any other PMT type. There are PMT functions that allow us to query the PMT to test if it is a particular type. We also have the ability to do `pmt::print` on any PMT object to print it to screen. Before converting from a PMT to its natural data type, it is necessary to know the data type.

== Utilities ==

GNU Radio comes with a couple of utilities to help in debugging and manipulating metadata files. There is a general parser in Python that will convert the PMT header and extra header into Python dictionaries. This utility is:

```
* gr-blocks/python/parse_file_metadata.py
```

This program is installed into the Python directory under the 'gnuradio' module, so it can be accessed with:

```
from gnuradio.blocks import parse_file_metadata
```

It defines `HEADER_LENGTH` as the static length of the metadata header size. It also has dictionaries that can be used to convert from the file type to a string (`ftype_to_string`) and one to convert from the file type to the size of the data type in bytes (`ftype_to_size`).

The 'parse\_header' takes in a PMT dictionary, parses it, and returns a Python dictionary. An optional 'VERBOSE' bool can be set to print the information to standard out.

The 'parse\_extra\_dict' is similar in that it converts from a PMT dictionary to a Python dictionary. The values are kept in their PMT format since we do not necessarily know the native data type.

A program called 'gr\_read\_file\_metadata' is installed into the path and can be used to read out all header information from a metadata file. This program is just called with the file name as the first command-line argument. An option '-D' will handle detached header files where the file of headers is expected to be the file name of the data with '.hdr' appended to it.

== Examples ==

Examples are located in:

- \* gr-blocks/examples/metadata

Currently, there are a few GRC example programs.

- \* file\_metadata\_sink: create a metadata file from UHD samples.
- \* file\_metadata\_source: read the metadata file as input to a simple graph.
- \* file\_metadata\_vector\_sink: create a metadata file from UHD samples.
- \* file\_metadata\_vector\_source: read the metadata file as input to a simple graph.

The file sink example can be switched to use a signal source instead of a UHD source, but no extra tagged data is used in this mode.

The file source example pushes the data stream to a new raw file while a tag debugger block prints out any tags observed in the metadata file. A QT GUI time sink is used to look at the signal as well.

The versions with 'vector' in the name are similar except they use vectors of data.

The following shows a simple way of creating extra metadata for a metadata file. This example is just showing how we can insert a date into the metadata to keep track of it later. The date in this case is encoded as a vector of uint16 with [day, month, year].

```
<syntaxhighlight lang="python">
import pmt
from gnuradio import blocks

key = pmt.intern("date")
```

```

val = pmt.init_u16vector(3, [13,12,2012])

extras = pmt.make_dict()
extras = pmt.dict_add(extras, key, val)
extras_str = pmt.serialize_str(extras)
self.sink = blocks.file_meta_sink(gr.sizeof_gr_complex,
                                   "/tmp/metadat_file.out",
                                   samp_rate, 1,
                                   blocks.GR_FILE_FLOAT, True,
                                   1000000, extras_str, False)
</syntaxhighlight></text>
<sha1>79bbsfgjz078sumck0ea80hkoe1kuow</sha1>
</revision>
</page>
</mediawiki>

```

## 7、 (exported from wiki) Performance Counters

```

<page>
  <title>Performance Counters</title>
  <ns>0</ns>
  <id>3485</id>
  <revision>
    <id>4872</id>
    <parentid>4185</parentid>
    <timestamp>2019-03-12T22:43:46Z</timestamp>
    <contributor>
      <username>777arc</username>
      <id>632</id>
    </contributor>
    <origin>4872</origin>
    <model>wikitext</model>
    <format>text/x-wiki</format>
    <text      bytes="3616"      sha1="rx725rwa6ws4e2bbgxxgyqps5lj041"
xml:space="preserve">[[Category:Usage Manual]]
== Introduction ==

```

Each block can have a set of Performance Counters that the schedule keeps track of. These counters measure and store information about different performance metrics of their operation. The concept is fairly extensible, but currently, GNU Radio defines the following Performance Counters:

```

# noutput_items: number of items the block can produce.
# nproduced: the number of items the block produced.

```

```
# input_buffers_full: % of how full each input buffer is.  
# output_buffers_full: % of how full each output buffer is.  
# work_time: number of CPU ticks during the call to general_work().  
# work_time_total: Accumulated sum of work_time.
```

For each Performance Counter except the `work_time_total`, we can retrieve the instantaneous, average, and variance from the block. Access to these counters is done through a simple set of functions added to every block in the flowgraph:

```
float pc_<name>[_<type>]();
```

In the above, the `<name>` field is one of the counters in the above list of counters. The optional `<type>` suffix is either 'avg' to get the average value or 'var' to get the variance. Without a suffix, the function returns the most recent instantaneous value.

We can also reset the Performance Counters back to zero to remove any history of the current average and variance calculations for a particular block.

```
void reset_perf_counters();
```

## == Compile-time and Run-time Configuration ==

Because the Performance Counters are calculated during each call to `work` for every block, they increase the computational cost and memory overhead. The more blocks used, the more impact this may have. So while it turns out after some experimentation that the Performance Counters add very little overhead (less than 1% speed degradation for a 24-block flowgraph), we err on the side of minimizing overhead in the scheduler. To do so, we have added compile-time and run-time configuration of the use of Performance Counters.

## === Compile-time Config ===

By default, GNU Radio will build without Performance Counters enabled. To enable Performance Counters, we pass the following flag to `cmake`:

```
-DENABLE_PERFORMANCE_COUNTERS=True
```

Note that this affects the GNU Radio block class and the scheduler itself. Out-of-tree projects will inherit directly from GNU Radio

because of the inheritance with `gr::block`. Turning on Performance Counters for GNU Radio will require a recompilation of the OOT project but no extra configuration.

#### ==== Run-time Config ====

Given the Performance Counters are enabled in GNU Radio at compile-time, we can still control if they are used or not at run-time. For this, we use the GNU Radio preferences file in the section `"PerfCounters"`. This section is installed into the `gnuradio-runtime.conf` file. As usual with the preferences, this section or any of the individual options can be overridden in the user's `config.conf` file or using a `GR_CONF_` environmental variable.

The options for the `"PerfCounters"` section are:

```
# on: Turn counters on/off at run-time.
# export: Allow counters to be exported over ControlPort.
# clock: sets the type of clock used when calculating work_time ('thread' or 'monotonic').
```

#### == Performance Monitor ==

See `[ControlPort#Performance Monitor]` for some details of using a ControlPort-based monitor application, `gr-perf-monitorx`, for visualizing the counters. This application is particularly useful in learning which blocks are the computationally complex blocks that could use extra optimization or work to improve their performance. It can also be used to understand the current 'health' of the application.

```
<sha1>rx725rwa6ws4e2bbgxxgyqps51j041</sha1>
</revision>
</page>
</mediawiki>
```

## 8、 (exported from wiki) Polymorphic Types (PMTs)

```
<page>
<title>Polymorphic Types (PMTs)</title>
<ns>0</ns>
<id>3478</id>
<revision>
<id>14604</id>
<parentid>14603</parentid>
<timestamp>2024-12-23T18:16:25Z</timestamp>
<contributor>
<username>Hipple</username>
```

```
<id>1261</id>
</contributor>
<minor/>
<comment>/* Introduction */</comment>
<origin>14604</origin>
<model>wikitext</model>
<format>text/x-wiki</format>
<text      bytes="25877"      sha1="dn5xz07nfxaabww7hswxi9q58ztmokp"
xml:space="preserve">[[Category:Usage Manual]]
== Introduction ==
```

If you came here from a search engine, but are actually looking for documentation, you probably want [[https://www.gnuradio.org/doc/doxygen/ this](https://www.gnuradio.org/doc/doxygen/this)].

Polymorphic Types (PMTs) are used as the carrier of data, from one block/thread to another, for such things as stream tags and message passing interfaces.

PMTs can represent a variety of data ranging from Boolean values to dictionaries.

In a sense, PMTs are a way to extend the strict typing of C++ with something more flexible.

This page summarizes the most important features and points of Polymorphic Types.

For an exhaustive list of PMT features check the source code, specifically the header file [[https://gnuradio.org/doc/doxygen/pmt\\_8h.html](https://gnuradio.org/doc/doxygen/pmt_8h.html) pmt.h].

Let's dive straight into some Python code and see how we can use PMTs:

```
<syntaxhighlight lang="python">
>>> import pmt
>>> P = pmt.from_long(23)
>>> type(P)
<class 'pmt.pmt_python.pmt_base'>
>>> print P
23
>>> P2 = pmt.from_complex(1j)
>>> type(P2)
<class 'pmt.pmt_python.pmt_base'>
>>> print P2
0+1i
>>> pmt.is_complex(P2)
True
</syntaxhighlight>
```

First, the `pmt` module is imported. We assign two values (`P` and `P2`) with PMTs using the `from_long()` and `from_complex()` calls, respectively. As we can see, they are both of the same type! This means we can pass these variables

to C++, and C++ can handle this type accordingly.

The same code as above in C++ would look like this:

```
<syntaxhighlight lang="c++">
#include <pmt/pmt.h>
// [...]
pmt::pmt_t P = pmt::from_long(23);
std::cout << P << std::endl;
pmt::pmt_t P2 = pmt::from_complex(0,1);
std::cout << P2 << std::endl;
std::cout << pmt::is_complex(P2) << std::endl;
</syntaxhighlight>
```

Two things stand out in both Python and C++: First, we can simply print the contents of a PMT. How is this possible? Well, the PMTs have built-in capabilities to cast their value to a string (this is not possible with all types, though). Second, PMTs know their type, so we can query that, e.g. by calling the `is_complex()` method.

When assigning a non-PMT value to a PMT, we can use the `from_<i>datatype</i>()` methods, and use the `to_<i>datatype</i>()` methods to convert back:

```
<syntaxhighlight lang="c++">
pmt::pmt_t P_int = pmt::from_long(42);
int i = pmt::to_long(P_int);
pmt::pmt_t P_double = pmt::from_double(0.2);
double d = pmt::to_double(P_double);
pmt::pmt_t P_double = pmt::mp(0.2);
</syntaxhighlight>
```

The last row shows the [\[http://gnuradio.org/doc/doxygen/namespacespmt.html#a90faad6086ac00280e0cfd8bb541bd64 pmt::mp\(\)\]](http://gnuradio.org/doc/doxygen/namespacespmt.html#a90faad6086ac00280e0cfd8bb541bd64) shorthand function. It basically saves some typing, as it infers the correct `from_</code>` function from the given type.

String types play a bit of a special role in PMTs, as we will see later, and have their own converter:

```
<syntaxhighlight lang="c++">
pmt::pmt_t P_str = pmt::string_to_symbol("spam");
pmt::pmt_t P_str2 = pmt::intern("spam");
std::string str = pmt::symbol_to_string(P_str);
</syntaxhighlight>
```

The `pmt::intern` is another way of saying `pmt::string_to_symbol`.

See the [<https://www.gnuradio.org/doc/doxygen/namespacepmt.html> PMT docs] and the header file [[http://gnuradio.org/doc/doxygen/pmt\\_8h.html](http://gnuradio.org/doc/doxygen/pmt_8h.html) pmt.h] for a full list of conversion functions.

In Python, we can make use of the dynamic typing, and there's actually a helper function to do these conversions (C++ also has a helper function for converting to PMTs called `pmt::mp()`, but it's less powerful, and not quite as useful, because types are always strictly known in C++):

```
<syntaxhighlight lang="python">
P_int = pmt.to_pmt(42)
i = pmt.to_python(P_int)
P_double = pmt.to_pmt(0.2)
d = pmt.to_double(P_double)
</syntaxhighlight>
```

On a side note, there are three useful PMT constants, which can be used in both Python and C++ domains. In C++, these can be used as such:

```
<syntaxhighlight lang="c++">
pmt::pmt_t P_true = pmt::PMT_T;
pmt::pmt_t P_false = pmt::PMT_F;
pmt::pmt_t P_nil = pmt::PMT_NIL;
</syntaxhighlight>
```

In Python:

```
<syntaxhighlight lang="python">
P_true = pmt.PMT_T
P_false = pmt.PMT_F
P_nil = pmt.PMT_NIL
</syntaxhighlight>
```

`pmt.PMT_T` and `pmt.PMT_F` are boolean PMT types representing True and False, respectively. The `PMT_NIL` is like a NULL or None and can be used for default arguments or return values, often indicating an error has occurred.

To go back to C++ data types, we need to be able to find out the type from a PMT. The family of `is_<i>datatype</i>()` methods helps us do that:

```
<syntaxhighlight lang="c++">
```

```
double d;
if (pmt::is_integer(P)) {
    d = (double) pmt::to_long(P);
} else if (pmt::is_real(P)) {
    d = pmt::to_double(P);
} else {
    // We really expected an integer or a double here, so we don't know what to do
    throw std::runtime_error("expected an integer!");
}
</syntaxhighlight>
```

It is important to do type checking since we cannot unpack a PMT of the wrong data type.

We can compare PMTs without knowing their type by using the `pmt::equal()` function:

```
<syntaxhighlight lang="c++">
if (pmt::eq(P_int, P_double)) {
    std::cout << "Equal!" << std::endl; // This line will never be reached
}
</syntaxhighlight>
```

There are more equality functions, which compare different things: [<http://gnuradio.org/doc/doxygen/namespacepmt.html#a5c28635e14287cc0e2f762841c11032f> `pmt::eq()`] and [<http://gnuradio.org/doc/doxygen/namespacepmt.html#a25467c81e1c5f4619a9cabad7a88eed5> `pmt::eqv()`]. We won't need these for this tutorial.

The rest of this page provides more depth into how to handle different data types with the PMT library.

### === More Complicated Types ===

PMTs can hold a variety of types. Using the Python method `pmt.to_pmt()`, we can convert most of Python's standard types out-of-the-box:

```
<pre>P_tuple = pmt.to_pmt((1, 2, 3, 'spam', 'eggs'))
P_dict = pmt.to_pmt({'spam': 42, 'eggs': 23})</pre>
```

But what does this mean in the C++ domain? Well, there are PMT types that define [<http://gnuradio.org/doc/doxygen/namespacepmt.html#a32895cc5a614a46b66b869c4a7bd283c> tuples] and [<http://gnuradio.org/doc/doxygen/namespacepmt.html#aba10563e3ab43b8d52f9cb13132047cf> dictionaries], keys and values being PMTs, again. <br />

So, to create the tuple from the Python example, the C++ code would look like this:

```
<pre>pmt::pmt_t P_tuple = pmt::make_tuple(pmt::from_long(1), pmt::from_long(2),
pmt::from_long(3),                pmt::string_to_symbol(&quot;spam&quot;),
pmt::string_to_symbol(&quot;eggs&quot;))</pre>
```

For the dictionary, it's a bit more complex:

```
<pre>pmt::pmt_t P_dict = pmt::make_dict();
P_dict    = pmt::dict_add(P_dict,    pmt::string_to_symbol(&quot;spam&quot;),
pmt::from_long(42));
P_dict    = pmt::dict_add(P_dict,    pmt::string_to_symbol(&quot;eggs&quot;),
pmt::from_long(23));</pre>
```

As you can see, we first need to create a dictionary, then assign every key/value pair individually.

A variant of tuples are "vectors". Like Python's tuples and lists, PMT vectors are mutable, whereas PMT tuples are not. In fact, PMT vectors are the only PMT data types that are mutable. When changing the value or adding an item to a dictionary, we are actually creating a new PMT.

To create a vector, we can initialize it to a certain lengths, and fill all elements with an initial value. We can then change items or reference them:

```
<pre>pmt::pmt_t P_vector = pmt::make_vector(5, pmt::from_long(23)); // Creates a
vector with 5 23's as PMTs
pmt::vector_set(P_vector, 0, pmt::from_long(42)); // Change the first element to a 42
std::cout && pmt::vector_ref(P_vector, 0); // Will print 42</pre>
```

In Python, we can do all these steps (using `pmt.make_vector()` etc.), or convert a list:

```
<pre>P_vector = pmt.to_pmt([42, 23, 23, 23, 23])</pre>
```

Vectors are also different from tuples in a sense that we can "directly" load data types into the elements, which don't have to be PMTs.

Say we want to pass a series of 8 float values to another block (these might be characteristics of a filter, for example). It would be cumbersome to convert every single element to and from PMTs, since all elements of the vector are the same type.

We can use special vector types for this case:

```
<pre>pmt::pmt_t P_f32vector = pmt::make_f32vector(8, 5.0); // Creates a vector with
8 5.0s as floats
pmt::f32vector_set(P_f32vector, 0, 2.0); // Change the first element to a 2.0
float f = f32vector_ref(P_f32vector, 0);
std::cout && f && std::endl; // Prints 2.0
```

```
size_t len;
float *fp = pmt::f32vector_elements(P_f32vector, len);
for (size_t i = 0; i < len; i++)
    std::cout << fp[i] << " "; // Prints all elements from P_f32vector,
one after another</pre>
```

Python has a similar concept: [\[http://docs.scipy.org/doc/numpy/reference/generated/numpy.array.html\]](http://docs.scipy.org/doc/numpy/reference/generated/numpy.array.html) Numpy arrays]. As usual, the PMT library understands this and converts as expected:

```
<pre>P_f32vector = pmt.to_pmt(numpy.array([2.0, 5.0, 5.0, 5.0, 5.0],
dtype=numpy.float32))
print pmt.is_f32vector(P_f32vector) # Prints 'True'</pre>
```

Here, 'f32' stands for 'float, 32 bits'. PMTs know about most typical fixed-width data types, such as 'u8' (unsigned 8-bit character) or 'c32' (complex with 32-bit floats for each I and Q). Consult the [\[http://gnuradio.org/doc/doxygen/namespacespmt.html\]](http://gnuradio.org/doc/doxygen/namespacespmt.html) manual] for a full list of types.

The most generic PMT type is probably the blob (binary large object). Use this with care - it allows us to pass around anything that can be represented in memory.

== PMT Data Type ==

All PMTs are of the type `pmt::pmt_t`. This is an opaque container and PMT functions must be used to manipulate and even do things like compare PMTs. PMTs are also "immutable" (except PMT vectors). We never change the data in a PMT; instead, we create a new PMT with the new data. The main reason for this is thread safety. We can pass PMTs as tags and messages between blocks and each receives its own copy that we can read from. However, we can never write to this object, and so if multiple blocks have a reference to the same PMT, there is no possibility of thread-safety issues of one reading the PMT data while another is writing the data. If a block is trying to write new data to a PMT, it actually creates a new PMT to put the data into. Thus we allow easy access to data in the PMT format without worrying about mutex locking and unlocking while manipulating them.

PMTs can represent the following:

- \* Boolean values of true/false
- \* Strings (as symbols)
- \* Integers (long and uint64)
- \* Floats (as doubles)
- \* Complex (as two doubles)
- \* Pairs

- \* Tuples
- \* Vectors (of PMTs)
- \* Uniform vectors (of any standard data type)
- \* Dictionaries (list of key:value pairs)
- \* Any (contains a boost::any pointer to hold anything)

The PMT library also defines a set of functions that operate directly on PMTs such as:

- \* Equal/equivalence between PMTs
- \* Length (of a tuple or vector)
- \* Map (apply a function to all elements in the PMT)
- \* Reverse
- \* Get a PMT at a position in a list
- \* Serialize and deserialize
- \* Printing

The constants in the PMT library are:

- \* pmt::PMT\_T - a PMT True
- \* pmt::PMT\_F - a PMT False
- \* pmt::PMT\_NIL - an empty PMT (think Python's 'None')

== Inserting and Extracting Data ==

Use [[https://gnuradio.org/doc/doxygen/pmt\\_8h.html](https://gnuradio.org/doc/doxygen/pmt_8h.html) pmt.h] for a complete guide to the list of functions used to create

PMTs and get the data from a PMT. When using these functions, remember that while PMTs are opaque and designed to hold any data, the data underneath is still a C++ typed object, and so the right type of set/get function must be used for the data type.

Typically, a PMT object can be made from a scalar item using a call like "pmt::from\_<type>". Similarly, when getting data out of a PMT, we use a call like "pmt::to\_<type>". For example:

```
<syntaxhighlight lang="c++">
double a = 1.2345;
pmt::pmt_t pmt_a = pmt::from_double(a);
double b = pmt::to_double(pmt_a);

int c = 12345;
pmt::pmt_t pmt_c = pmt::from_long(c);
int d = pmt::to_long(pmt_c);
</syntaxhighlight>
```

As a side-note, making a PMT from a complex number is not obvious:

```
<syntaxhighlight lang="c++">
std::complex<double> a(1.2, 3.4);
pmt::pmt_t pmt_a = pmt::make_rectangular(a.real(), a.imag());
std::complex<double> b = pmt::to_complex(pmt_a);
</syntaxhighlight>
```

Pairs, dictionaries, and vectors have different constructors and ways to manipulate them, and these are explained in their own sections.

## == Strings ==

PMTs have a way of representing short strings. These strings are actually stored as interned symbols in a hash table, so in other words, only one PMT object for a given string exists. If creating a new symbol from a string, if that string already exists in the hash table, the constructor will return a reference to the existing PMT.

We create strings with the following functions, where the second function, `pmt::intern`, is simply an alias of the first.

```
<syntaxhighlight lang="c++">
pmt::pmt_t str0 = pmt::string_to_symbol(std::string("some string"));
pmt::pmt_t str1 = pmt::intern(std::string("some string"));
</syntaxhighlight>
```

The string can be retrieved using the inverse function:

```
<syntaxhighlight lang="c++">
std::string s = pmt::symbol_to_string(str0);
</syntaxhighlight>
```

## == Tests and Comparisons ==

The PMT library comes with a number of functions to test and compare PMT objects. In general, for any PMT data type, there is an equivalent "`pmt::is_<type>`". We can use these to test the PMT before trying to access the data inside. Expanding our examples above, we have:

```
<syntaxhighlight lang="c++">
pmt::pmt_t str0 = pmt::string_to_symbol(std::string("some string"));
if(pmt::is_symbol(str0))
    std::string s = pmt::symbol_to_string(str0);
```

```
double a = 1.2345;
pmt::pmt_t pmt_a = pmt::from_double(a);
if(pmt::is_double(pmt_a))
    double b = pmt::to_double(pmt_a);
```

```

int c = 12345;
pmt::pmt_t pmt_c = pmt::from_long(c);
if(pmt::is_long(pmt_c))
    int d = pmt::to_long(pmt_c);

\\ This will fail the test. Otherwise, trying to coerce pmt_c as a
\\ double when internally it is a long will result in an exception.
if(pmt::is_double(pmt_c))
    double d = pmt::to_double(pmt_c);
</syntaxhighlight>

```

## == Dictionaries ==

PMT dictionaries are lists of key:value pairs. They have a well-defined interface for creating, adding, removing, and accessing items in the dictionary. Note that every operation that changes the dictionary both takes a PMT dictionary as an argument and returns a PMT dictionary. The dictionary used as an input is not changed and the returned dictionary is a new PMT with the changes made there.

The following is a list of PMT dictionary functions. View each function in the [<https://www.gnuradio.org/doc/doxygen/index.html> GNU Radio C++ Manual] to get more information on what each does.

```

* bool pmt::is_dict(const pmt_t &obj)
* pmt_t pmt::make_dict()
* pmt_t pmt::dict_add(const pmt_t &dict, const pmt_t &key, const pmt_t &value)
* pmt_t pmt::dict_delete(const pmt_t &dict, const pmt_t &key)
* bool pmt::dict_has_key(const pmt_t &dict, const pmt_t &key)
* pmt_t pmt::dict_ref(const pmt_t &dict, const pmt_t &key, const pmt_t &not_found)
* pmt_t pmt::dict_items(pmt_t dict)
* pmt_t pmt::dict_keys(pmt_t dict)
* pmt_t pmt::dict_values(pmt_t dict)

```

This example does some basic manipulations of PMT dictionaries in Python. Notice that we pass the dictionary "a" and return the results to "a". This still creates a new dictionary and removes the local reference to the old dictionary. This just keeps our number of variables small.

```

<syntaxhighlight lang="python">
import pmt

key0 = pmt.intern("int")

```

```

val0 = pmt.from_long(123)
val1 = pmt.from_long(234)

key1 = pmt.intern("double")
val2 = pmt.from_double(5.4321)

# Make an empty dictionary
a = pmt.make_dict()

# Add a key:value pair to the dictionary
a = pmt.dict_add(a, key0, val0)
print a

# Add a new value to the same key;
# new dict will still have one item with new value
a = pmt.dict_add(a, key0, val1)
print a

# Add a new key:value pair
a = pmt.dict_add(a, key1, val2)
print a

# Test if we have a key, then delete it
print pmt.dict_has_key(a, key1)
a = pmt.dict_delete(a, key1)
print pmt.dict_has_key(a, key1)

ref = pmt.dict_ref(a, key0, pmt.PMT_NIL)
print ref

# The following should never print
if(pmt.dict_has_key(a, key0) and pmt.eq(ref, pmt.PMT_NIL)):
    print "Trouble! We have key0, but it returned PMT_NIL"
</syntaxhighlight>

```

== Vectors ==

PMT vectors come in two forms: vectors of PMTs and vectors of uniform data. The standard PMT vector is a vector of PMTs, and each PMT can be of any internal type. On the other hand, uniform PMTs are of a specific data type which come in the form:

```

* (u)int8
* (u)int16

```

- \* (u)int32
- \* (u)int64
- \* float32
- \* float64
- \* complex 32 (std::complex<float>)
- \* complex 64 (std::complex<double>)

That is, the standard sizes of integers (both signed and unsigned), floats, and complex types.

Vectors have a well-defined interface that allows us to make, set, get, and fill them. We can also get the length of a vector with "pmt::length":

```
pmt_t p1 = pmt_integer(1);
pmt_t p2 = pmt_integer(2);
pmt_t p3 = pmt_integer(3);

pmt_t p_list = pmt_list3(p1, p2, p3);

pmt_length(p_list); // Returns 3
```

For standard vectors, these functions look like:

- \* bool pmt::is\_vector(pmt\_t x)
- \* pmt\_t pmt::make\_vector(size\_t k, pmt\_t fill)
- \* pmt\_t pmt::vector\_ref(pmt\_t vector, size\_t k)
- \* void pmt::vector\_set(pmt\_t vector, size\_t k, pmt\_t obj)
- \* void pmt::vector\_fill(pmt\_t vector, pmt\_t fill)

Uniform vectors have the same types of functions, but they are data type-dependent. The following list illustrates where to substitute the specific data type prefix for (dtype) (prefixes being: u8, u16, u32, u64, s8, s16, s32, s64, f32, f64, c32, c64).

- \* bool pmt::is\_(dtype)vector(pmt\_t x)
- \* pmt\_t pmt::make\_(dtype)vector(size\_t k, (dtype) fill)
- \* pmt\_t pmt::init\_(dtype)vector(size\_t k, const (dtype\*) data)
- \* pmt\_t pmt::init\_(dtype)vector(size\_t k, const std::vector<dtype> data)
- \* pmt\_t pmt::(dtype)vector\_ref(pmt\_t vector, size\_t k)
- \* void pmt::(dtype)vector\_set(pmt\_t vector, size\_t k, (dtype) x)
- \* const dtype\* pmt::(dtype)vector\_elements(pmt\_t vector, size\_t &len)
- \* dtype\* pmt::(dtype)vector\_writable\_elements(pmt\_t vector, size\_t &len)

List of available `pmt::is_(dtype)vector(pmt_t x)` methods from [\[https://gnuradio.org/doc/doxygen/pmt\\_8h.html\]](https://gnuradio.org/doc/doxygen/pmt_8h.html):<br>

```
bool pmt::is_uniform_vector()
bool pmt::is_u8vector()
bool pmt::is_s8vector()
bool pmt::is_u16vector()
bool pmt::is_s16vector()
bool pmt::is_u32vector()
bool pmt::is_s32vector()
bool pmt::is_u64vector()
bool pmt::is_s64vector()
bool pmt::is_f32vector()
bool pmt::is_f64vector()
bool pmt::is_c32vector()
bool pmt::is_c64vector()
```

"Note:" We break the contract with vectors. The 'set' functions actually change the data underneath. It is important to keep track of the implications of setting a new value as well as accessing the 'vector\_writable\_elements' data. Since these are mostly standard data types, sets and gets are atomic, so it is unlikely to cause a great deal of harm. But it's only unlikely, not impossible. Best to use mutexes whenever manipulating data in a vector.

=== BLOB ===

A BLOB is a 'binary large object' type. In PMT's, this is actually just a thin wrapper around a `u8vector`.

== Pairs ==

A concept that originates in Lisp dialects are [<http://en.wikipedia.org/wiki/Cons> "pairs" and "cons"]. The simplest explanation is just that: If you combine two PMTs, they form a new PMT, which is a pair (or cons) of those two PMTs (don't worry about the weird name, a lot of things originating in Lisp have weird names. Think of a 'construct').

Similarly to vectors or tuples, pairs are a useful way of packing several components of a message into a single PMT. Using the PMTs generated in the previous section, we can combine two of these to form a pair, here in Python:

```
<pre>P_pair = pmt.cons(pmt.string_to_symbol(&quot;taps&quot;), P_f32vector)
print pmt.is_pair(P_pair) # Prints 'true'</pre>
```

You can combine PMTs as tuples, dictionaries, vectors, or pairs, it's just a matter of

taste. This construct is well-established though, and as such used in GNU Radio quite often.

So how do we deconstruct a pair? That's what the `car` and `cdr` functions do. Let's deconstruct that previous pair in C++:

```
pmt::pmt_t P_key = pmt::car(P_pair);
pmt::pmt_t P_f32vector2 = pmt::cdr(P_pair);
std::cout << P_key << std::endl; // Will print 'taps' using the PMT automatic
conversion to strings
```

Here is a summary of the pairs-related functions in C++ and Python:

- \* `bool pmt::is_pair(const pmt_t &obj)`: Return true if obj is a pair, else false
- \* `pmt_t pmt::cons(const pmt_t &x, const pmt_t &y)`: construct new pair
- \* `pmt_t pmt::car(const pmt_t &pair)`: get the car of the pair (first object)
- \* `pmt_t pmt::cdr(const pmt_t &pair)`: get the cdr of the pair (second object)
- \* `void pmt::set_car(pmt_t pair, pmt_t value)`: Stores value in the car field
- \* `void pmt::set_cdr(pmt_t pair, pmt_t value)`: Stores value in the cdr field

And in Python we have:

```
<syntaxhighlight lang="python">
pmt.is_pair(pair_obj) # Return True if is a pair, else False (warning: also returns True
for a dict)
pmt.cons(x, y) # Return a newly allocated pair whose car is x and whose cdr is y
pmt.car(pair_obj) # If is a pair, return the car, otherwise raise wrong_type
pmt.cdr(pair_obj) # If is a pair, return the cdr, otherwise raise wrong_type
pmt.set_car(pair_obj, value) # Store value in car field
pmt.set_cdr(pair_obj, value) # Store value in cdr field
</syntaxhighlight>
```

For more advanced pair manipulation, refer to the [<http://gnuradio.org/doc/doxygen/namespacespmt.html#a7ab95721db5cbda1852f13a92eee5362> documentation] and the [[https://en.wikipedia.org/wiki/Car\\_and\\_cdr](https://en.wikipedia.org/wiki/Car_and_cdr)] Wikipedia page for car and cdr].

== PDUs ==

A PDU is a special type of PMT object that is a PMT pair consisting of a dictionary and a uniform vector. PMTs are used to pass complete segments of data accompanied by metadata describing that data as a single object. The dictionary can contain any key/value pairs useful to describe the data, though there are a number of common keys built into GNU Radio. Any uniform vector type can be used, but complex, float, int, short, and byte are the most common vector types with the best support.

Here are some examples:

\* Create a PDU with no metadata and 10 zeros as stream data:

```
<pre>
pdu = pmt.cons(pmt.make_dict(), pmt.make_u8vector(10, 0))
</pre>
```

\* Create a PDU from a Python array:

```
<pre>
idle = [127,127,127]
idle_len = len(idle)
pdu = pmt.cons(pmt.PMT_NIL,pmt.init_u8vector(idle_len,(idle)))
</pre>
```

\* This snippet converts a PMT string to a PDU:

```
<pre>
if not pmt.is_symbol(msg):
    print ("PMT is not a symbol")
    return
u8_data = [ord(i) for i in pmt.symbol_to_string (msg)]
pdu = pmt.cons(pmt.PMT_NIL,pmt.init_u8vector(len(u8_data), u8_data))
</pre>
```

== Serializing and Deserializing ==

It is often important to hide the fact that we are working with PMTs to make them easier to transmit, store, write to file, etc. The PMT library has methods to serialize data into a string buffer or a string and then methods to deserialize the string buffer or string back into a PMT. We use this extensively in the metadata files (see [\[\[Metadata Information\]\]](#)).

```
* bool pmt::serialize(pmt_t obj, std::streambuf &sink)
* std::string pmt::serialize_str(pmt_t obj)
* pmt_t pmt::deserialize(std::streambuf &source)
* pmt_t pmt::deserialize_str(std::string str)
```

For example, we will serialize the data above to make it into a string ready to be written to a file and then deserialize it back to its original PMT.

```

<syntaxhighlight lang="python">
import pmt

key0 = pmt.intern("int")
val0 = pmt.from_long(123)

key1 = pmt.intern("double")
val1 = pmt.from_double(5.4321)

# Make an empty dictionary
a = pmt.make_dict()

# Add a key:value pair to the dictionary
a = pmt.dict_add(a, key0, val0)
a = pmt.dict_add(a, key1, val1)

print a

ser_str = pmt.serialize_str(a)
print ser_str

b = pmt.deserialize_str(ser_str)
print b
</syntaxhighlight>

```

The line where we 'print ser\_str' will print and parts will be readable, but the point of serializing is not to make a human-readable string. This is only done here as a test.

== Printing ==

In Python, the `__repr__` function of a PMT object is overloaded to call 'pmt::write\_string'. This means that any time we call a formatted printing operation on a PMT object, the PMT library will properly format the object for display.

In C++, we can use the 'pmt::print(object)' function or print the contents using the overloaded "<<" operator with a stream buffer object. In C++, we can inline print the contents of a PMT like:

```

<syntaxhighlight lang="c++">
pmt::pmt_t a = pmt::from_double(1.0);
std::cout << "The PMT a contains " << a << std::endl;
</syntaxhighlight>

```

==== Collection Notation ====

PMTs use a different bracket notation from what one might be used to in Python.

```
<syntaxhighlight lang="python">
>>> my_dict
((meaning . 42))
>>> my_vector
#[1 2 3 4]
>>> pmt.make_tuple(pmt.from_long(321), pmt.from_float(3.14))
{321 3.14}
>>> pmt.cons(pmt.from_long(1), pmt.from_long(2))
(1 . 2)
>>> my_pdu
((). #[1 2 3 4])
</syntaxhighlight>
```

== Conversion between Python Objects and PMTs ==

Although PMTs can be manipulated in Python using the Python versions of the C++ interfaces, there are some additional goodies that make it easier to work with PMTs in python. There are functions to automate the conversion between PMTs and Python types for booleans, strings, integers, longs, floats, complex numbers, dictionaries, lists, tuples and combinations thereof.

Two functions capture most of this functionality:

```
<syntaxhighlight lang="python">
pmt.to_pmt      # Converts a python object to a PMT.
pmt.to_python   # Converts a PMT into a python object.
</syntaxhighlight></text>
<sha1>dn5xz07nfxaabww7hswxi9q58ztmokp</sha1>
</revision>
</page>
</mediawiki>
```

## 9、 (exported from wiki) Polyphase Filterbanks

```
<page>
<title>Polyphase Filterbanks</title>
<ns>0</ns>
<id>3496</id>
<revision>
<id>4877</id>
```

```

<parentid>4513</parentid>
<timestamp>2019-03-12T22:44:11Z</timestamp>
<contributor>
  <username>777arc</username>
  <id>632</id>
</contributor>
<origin>4877</origin>
<model>wikitext</model>
<format>text/x-wiki</format>
<text      bytes="7365"      sha1="ctzii3mllzcd7uazhuv0yd5kflu7bvv"
xml:space="preserve">[[Category:Usage Manual]]
== Introduction ==

```

Polyphase filterbanks (PFB) are a very powerful set of filtering tools that can efficiently perform many multi-rate signal processing tasks. GNU Radio has a set of polyphase filterbank blocks to be used in all sorts of applications. See the documentation for the individual blocks for details about what they can do and how they should be used. Furthermore, there are examples for these blocks in **<b>gr-filter/examples</b>**:

```

# "channelize.py" creates an appropriate filter to channelizer 9 channels out of an
original signal that is 9000 Hz wide, so each output channel is now 1000 Hz. The code
then plots the PSD of the original signal to see the signals in the original spectrum and
then makes 9 plots for each of the channels.
# "chirp_channelize.py" is similar to channelize.py but includes a VCO to create a
chirp signal
# "decimate.py" shows an example of using the PFB decimator
# "interpolate.py" shows an example of using the PFB interpolator
# "reconstruction.py" includes a PFB channelizer and PFB synthesizer
# "resampler.py" demonstrates how to use the PFB resampler
# "synth_filter.py" is a simple example of using the PFB synthesizer
NOTE: you need the Matplotlib Python module installed to run these examples

```

== PFB Usage ==

The main issue when using the PFB blocks is defining the prototype filter, which is passed to all of the blocks as a vector of taps. The taps from the prototype filter which get partitioned among the N channels of the channelizer.

An example of creating a set of filter taps for a PFB channelizer is found on line 49 of **<b>gr-filter/examples/channelizer.py</b>** and reproduced below. Notice that the sample rate is the sample rate

at the input to the channelizer while the bandwidth and transition width are defined for the channel bandwidths. This makes a fairly long filter that is then split up between the N channels of the PFB.

```
self._fs = 9000          # input sample rate
self._M = 9              # Number of channels to channelize
self._taps = filter.firdes.low_pass_2(1, self._fs, 475.50, 50,
                                     attenuation_dB=100,
```

```
window=filter.firdes.WIN_BLACKMAN_hARRIS)
```

In this example, the signal into the channelizer is sampled at 9 ksps (complex, so 9 kHz of bandwidth). The filter uses 9 channels, so each output channel will have a bandwidth and sample rate of 1 kHz. We want to pass most of the channel, so we define the channel bandwidth to be a low pass filter with a bandwidth of 475.5 Hz and a transition bandwidth of 50 Hz, but we have defined this using a sample rate of the original 9 kHz. The prototype filter has 819 taps to be divided up between the 9 channels, so each channel uses 91 taps. This is probably over-kill for a channelizer, and we could reduce the amount of taps per channel to a couple of dozen with no ill effects.

The basic rule when defining a set of taps for a PFB block is to think about the filter running at the highest rate it will see while the bandwidth is defined for the size of the channels. In the channelizer case, the highest rate is defined as the rate of the incoming signal, but in other PFB blocks, this is not so obvious.

Two very useful blocks to use are the arbitrary resampler and the clock synchronizer (for PAM signals). These PFBs are defined with a set number of filters based on the fidelity required from them, not the rate changes. By default, the filter\_size is set to 32 for these blocks, which is a reasonable default for most tasks. Because the PFB uses this number of filters in the filterbank, the maximum rate of the bank is defined from this (see the theory of a polyphase interpolator for a justification of this). So the prototype filter is defined to use a sample rate of filter\_size times the signal's sampling rate.

A helpful wrapper for the arbitrary resampler is found in `<b>gr-filter/python/pfb.py</b>`, which is exposed in Python as `<b>filter.pfb.arb_resampler_ccf</b>` and `<b>filter.pfb.arb_resampler_fff</b>`. This block is set up so that the user only needs to pass it the real number rate as the resampling

rate. With just this information, this hierarchical block automatically creates a filter that fully passes the signal bandwidth being resampled but does not pass any out-of-band noise. See the code for this block for details of how the filter is constructed.

Of course, a user can create his or her own taps and use them in the arbitrary resampler for more specific requirements. Some of the UHD examples (`<b>gr-uhd/examples</b>`) use this ability to create a received matched filter or channel filter that also resamples the signal.

### == The PFB Arbitrary Resampler Kernel ==

GNU Radio has a PFB arbitrary resampler block that can be used to resample a signal to any arbitrary and real resampling rate. The resampling feature is one that could easily be useful to other blocks, and so we have extracted the kernel of the resampler into its own class that can be used as such.

The PFB arbitrary resampler is defined in `pfb_arb_resampler.h` and has the following constructor:

```
namespace gr {
  namespace filter {
    namespace kernel {

      pfb_arb_resampler_XXX(float rate,
                           const std::vector<float> &taps,
                           unsigned int filter_size);

    } /* namespace kernel */
  } /* namespace filter */
} /* namespace gr */
```

Currently, only a 'ccf' and 'fff' version are defined. This kernel, like the block itself, takes in the resampling rate as a floating point number. The taps are passed as the baseband prototype filter, and the quantization error of the filter is determined by the `filter_size` parameter.

The prototype taps are generated like all other PFB filter taps. Specifically, we construct them generally as a lowpass filter at the maximum rate of the filter. In the case of these resamplers, the

maximum rate is actually the number of filters.

A simple example follows. We construct a filter that will pass the entire passband of the original signal to be resampled. To make it easy, we work in normalized sample rates for this. The gain of the filter is set to `filter_size` to compensate for the upsampling, the sampling rate itself is also set to `filter_size`, which is assuming that the incoming signal is at a sampling rate of 1.0. We defined the passband to be 0.5 to pass the entire width of the original signal and set a transition band to 0.1. Note that this causes a bit of roll-off outside of the original passband and could lead to introducing some aliasing. More care should be taken to construct the passband and transition width of the filter for the given signal while keeping the total number of taps small. A stopband attenuation of 60 dB was used here, and again, this is a parameter we can adjust to alter the performance and size of the filter.

```
firdes.low_pass_2(filter_size, filter_size, 0.5, 0.1, 60)
```

As is typical with the PFB filters, a filter size of 32 is generally an appropriate trade-off of accuracy, performance, and memory. This should provide an error roughly equivalent to the quantization error of using 16-bit fixed point representation. Generally, increasing over 32 provides some accuracy benefits without a huge increase in computational demands.

```
<sha1>ctzii3mllzcd7uazhuv0yd5kflu7bvv</sha1>
</revision>
</page>
</mediawiki>
```

## 10、 (exported from wiki) QT GUI

```
<page>
<title>QT GUI</title>
<ns>0</ns>
<id>3486</id>
<revision>
<id>4190</id>
<parentid>4189</parentid>
<timestamp>2018-05-15T19:45:25Z</timestamp>
<contributor>
<username>777arc</username>
<id>632</id>
</contributor>
<comment>/* C++ and Message-Passing Widgets */</comment>
```

```

<origin>4190</origin>
<model>wikitext</model>
<format>text/x-wiki</format>
<text      bytes="12387"      sha1="6p0evs1odganpw5pf993ifhtdncao90"
xml:space="preserve">== Introduction ==

```

This is the gr-qtgui package. It contains various QT-based graphical user interface blocks that add graphical sinks to a GNU Radio flowgraph. The Python namespaces is in gnuradio.qtgui, which would be normally imported as:

```
from gnuradio import qtgui
```

See the Doxygen documentation for details about the blocks available in this package. The relevant blocks are listed [[https://gnuradio.org/doc/doxygen/group\\_\\_qtgui\\_\\_blk.html](https://gnuradio.org/doc/doxygen/group__qtgui__blk.html) here].

A quick listing of the details can be found in Python after importing by using:

```
help(qtgui)
```

```
=== Blocks ===
```

There are a number of available QTGUI blocks for different plotting purposes. These include:

```

# Time Domain (gr::qtgui::time_sink_c and gr::qtgui::time_sink_f): x-axis is time, y-axis is amplitude.
# Frequency Domain or PSD (gr::qtgui::freq_sink_c and gr::qtgui::freq_sink_f): x-axis is frequency, y-axis is magnitude in dB.
# Waterfall or spectrogram (gr::qtgui::waterfall_sink_c and gr::qtgui::waterfall_sink_f): x-axis is frequency, y-axis is time, z-axis is intensity related to magnitude in dB.
# Constellation (gr::qtgui::const_sink_c): polar plot of real vs. imaginary.
# Time Raster (gr::qtgui::time_raster_sink_f and gr::qtgui::time_raster_sink_b): time vs. time with the z-axis being intensity based on value of the sample.
# Histogram (gr::qtgui::histogram_sink_f): Displays a histogram of the data stream.
# Combined Sink (gr::qtgui::sink_c and gr::qtgui::sink_f): combines time, frequency, waterfall, and constellation plots into one widget.

```

The time domain, frequency domain, and waterfall have both a complex and a floating point block. The constellation plot only makes sense with complex inputs. The time raster plots accept bits and floats.

Because the time raster plots are designed to show structure over time in a signal, frame, packet, etc., they never drop samples. This is a fairly taxing job and performance can be an issue. Since it is expected that this block will work on a frame or packet structure, we tend to be at the lowest possible rate at this point, so that will help. Expect performance issues at high data rates.

Note: There seem to be extra performance issues with the raster plotters in QWT version 5 that were fixed with QWT version 6. As such, the time raster plots have incredibly poor performance with QWT5 to the point of almost being unusable. In the future, we may restrict compilation and installation of these plots only if QWT6 or higher is discovered. For now, just be aware of this limitation.

== Drop-Down Menu and Interacting with Plots ==

All QTGUI sinks have interactive capabilities.

# Zooming is done simply by clicking the left mouse button and dragging a rectangle around the area to zoom.

# Zooming can be done in multiple steps.

# A right mouse click will zoom out one step.

# Ctrl+Right mouse click will zoom all the way out.

# Ctrl+Middle mouse click and hold can drag the canvas around.

# Mouse wheel up/down will zoom out/in on y axis (both axes in constellation plot).

# Middle mouse button brings up a context menu.

Each type of graph has a different set of menu items in the context menu. Most have some way to change the appearance of the lines or surfaces, such as changing the line width color, marker, and transparency. Other common features can set the sampling rate, turn a grid on and off, pause and unpause (stop/start) the display update, and save the current figure. Specific features are things like setting the number of points to display, setting the FFT size, FFT window, and any FFT averaging.

=== Triggering Menu for Time Plots ===

The time plots have triggering capabilities. Triggering can happen when the signal of a specific channel crosses (positive or negative slope) a certain level threshold. Or triggering can be done off a specific stream tag such that whenever a tag of a given key is found, the scope will trigger.

In the signal level mode, the trigger can be either 'auto' or 'normal' where the latter will only trigger when the event is seen. The 'auto' mode will trigger on the event or every so often even if no trigger is found. The 'free' mode ignores triggering and continuously plots.

By default, the triggers plot the triggering event at the  $x=0$  (i.e., the left-most point in the plot). A delay can be set to delay the signal along the x-axis to observe any signal before the triggering event. The delay feature works the same for both level and tag triggers. The delay is set according to time in seconds, not samples. So the delay can be calculated as the number of samples divided by the sample rate given to the block.

All trigger settings (mode, slope, level, delay, channel, and tag key) are settable in the GRC properties boxes to easily set up a repeatable environment.

A note on the trigger delay setting. This value is limited by the buffer size and/or the number of points being display. It is capped by the minimum of these two values. The buffer size issue is generally only a problem when plotting a large number of samples. However, if the delay is set large to begin with (in the GRC properties box or before `top_block.start()` is called), then the buffers are resized accordingly offering more freedom. This should be a problem in a limited number of scenarios, but a log INFO level message is produced when asking for the delay outside of the available range.

== Dependencies ==

The QT GUI blocks require the following dependencies.

```
# QtCore (version >= 4.4)
# QtGui (version >= 4.4)
# QtOpenGL (version >= 4.4)
# PyQt4 for Qt4 (version >= 4.4)
# Qwt (version >= 5.2)
```

== Usage ==

To use the QTGUI interface, a bit of boiler-plate lines must be included. First, the sink is defined, then it must be exposed from C++ into Python using the "sip.wrapinstance" command, and finally, the "show" method is run on the new Python object. This sets up the QT

environment to show the widget, but the `qApplication` must also be launched.

In the "main" function of the code, the `qApp` is retrieved. Then, after the GNU Radio top block is started (remember that `start()` is a non-blocking call to launch the main thread of the flowgraph), the `qapp's "exec_()"` function is called. This function is a blocking call while the GUI is alive.

```
from PyQt4 import Qt
from gnuradio import qtgui
import sys, sip

class grclass(gr.top_block):
    ....

    self.snk = qtgui.sink_c(1024,          #fftsize
                           samp_rate,     #bw
                           "QT GUI Plot") #name

    self.snk_win = sip.wrapinstance(self.snk.pyqwidget(), Qt.QWidget)
    self.snk_win.show()

def main():
    qapp = Qt.QApplication(sys.argv)
    tb = grclass()
    tb.start()
    qapp.exec_()
    tb.stop()
```

There are graphical controls in all but the combined plotting tools. In the margins of the GUIs (that is, not on the canvas showing the signal itself), right-clicking the mouse will pull up a drop-down menu that will allow you to change difference parameters of the plots. These include things like the look of the lines (width, color, style, markers, etc.), the ability to start and stop the display, the ability to save to a file, and other plot-specific controls (FFT size for the frequency and waterfall plots, etc.).

== Message Input Support ==

All QTGUI sinks can accept and plot messages over their "in" message port. The message types must either be uniform vectors or PDUs. The data type held within the uniform vector or PDU must match the data

type of the block itself. For example, a `qtgui.time_sink_c` will only handle vectors that pass the `pmt::is_c32vector` test while a `qtgui.time_sink_f` will only handle vectors that pass the `pmt::is_f32vector` test.

The sinks must only be used with one type of input model: streaming or messages. You cannot use them both together or unknown behavior will occur.

In the GNU Radio Companion, the QTGUI sink blocks can be set to message mode by changing the Type field. Most of the QTGUI sinks support multiple data types, even for messages, but GRC only displays the message type as the single gray color. Within the block's property box, you can set the type to handle the correct message data type (e.g., 'Complex Message' or 'Float Message'). When using a message type interface, GRC will hide certain parameters that are not usable or settable anymore. For example, when plotting a message in the time sink, the number of points shown in the time sink is determined by the length of the vector in the message. Presetting this in the GUI would have no effect.

This behavior in GRC is for convenience and to try and reduce confusion about properties and settings in the message mode. However, all of the API hooks are still there, so it is possible to set all of this programmatically. The results would be harmless, however.

Here is an example of setting up and using a message passing complex time sink block:

```
from gnuradio import gr, qtgui

tsnk = qtgui.time_sink_c(1024, samp_rate, "", 0)
tsnk.set_update_time(0.05)
tsnk.set_y_axis(-1.25, 1.25)
tsnk.set_y_label("Amp (V)", "")
tsnk.enable_autoscale(False)
tsnk.enable_grid(False)
tsnk.enable_control_panel(False)

tb = gr.top_block()
msg_block = ? # some PDU/message generating block
tb.msg_connect((msg_block, 'msg'), (tsnk, 'in'))
```

== QTGUI Widgets ==

The QTGUI component also includes a number of widgets that can be used to perform live updates of variables through standard QT input widgets. Most of the widgets are implemented directly in Python through PyQt. However, GNU Radio is introducing more widgets, written and therefore available in C++ that also produce messages. The Python-based widgets only act as variables and so as they are changed, any block using those widgets to set parameters has the callback (i.e., `set_value()`) function's called.

=== Python widgets: ===

- # Range: creates a slider and/or combo box to change to set/change the value of a parameter. This widget can set either float or int values.
- # Entry: An edit box that allows a user to directly set a new value for the parameter.
- # Chooser: Creates a drop-down menu of pre-set values.
- # Check Box: Creates a check box. The user sets what the value of the check means when enabled or disabled.
- # Push Button: Adds a button that changes state when pushed versus released (no sticky). The user sets up what the value is when pressed versus when released.
- # Label: Adds a Label widget to annotate the GUI. Generally not used as a variable.
- # Tab Widget: Adds a tab widget that can house other GUI widgets to format the interface. Use the GUI hint of the other QT widgets and instruments to specify if and where they exist in the tab widget using the format "tag widget name@index: row, col, row span, col span". Simply using "tag widget name@index" will put that widget into the specific index (starting at 0) of the tab widget while adding the "row, col, row span, col span" will allow the user to place them in the tab grid.

=== C++ and Message-Passing Widgets ===

The [https://gnuradio.org/doc/doxygen/classgr\_1\_1qtgui\_1\_1edit\_\_box\_\_msg.html Message Edit Box] is a QT edit box that emits a message when editing is done (e.g., user presses enter, tabs out of the widget, or mouse-clicks out of the widget). The message type is settable as are the contents. Messages can be sent as key:value pairs when Pair Mode is enabled. When Static Mode is enabled, the data type and the pair key (if in Pair Mode) are set at the start and cannot be changed at runtime.

== Configuration ==

There is currently a single configuration option in the preferences files to set the rendering engine of the QTGUI sinks. Located in `etc/gnuradio/conf.d/gr-qtgui.conf`:

```
[qtgui]
style = raster
```

The available styles are:

```
# opengl: the fastest but not likely to always work
# raster: fast and stable; terrible X forwarding performance
# native: most compute intensive; very good over X
```

We default this setting to raster for the mix of performance and usability. When using QTGUI sinks through an X-forwarding session over SSH, switch to using 'native' for a significant speed boost on the remote end.

We can also set a QT Style Sheet (QSS) file to adjust the look of our plotting tools. Set the 'qss' option of the 'qtgui' section in our configuration file to a QSS file. An example QSS file is distributed with the QTGUI examples found in

```
share/gnuradio/examples/qt-gui/dark.qss.</text>
    <sha1>6p0evs1odganpw5pf993ifhtdncao90</sha1>
</revision>
</page>
</mediawiki>
```

## 11、 (exported from wiki) Stream Tags

```
<page>
  <title>Stream Tags</title>
  <ns>0</ns>
  <id>3482</id>
  <revision>
    <id>13335</id>
    <parentid>12892</parentid>
    <timestamp>2023-08-17T18:46:02Z</timestamp>
    <contributor>
      <ip>174.179.104.158</ip>
    </contributor>
    <comment>Added Next Tutorial link</comment>
    <origin>13335</origin>
    <model>wikitext</model>
    <format>text/x-wiki</format>
    <text      bytes="21970"      sha1="fgw53qlkhiuz5ip4ps5l00yofnbq6eq"
xml:space="preserve">[[Category:Usage Manual]]
```

This tutorial describes the use of stream tags. The next tutorial,

[[Polymorphic\_Types\_(PMTs)|Polymorphic Types (PMTs)]] covers Polymorphic\_Types

## == Introduction ==

Stream tags are an isosynchronous data stream that runs parallel to the main data stream. A stream "tag" is generated by a block's work function and from there on flows downstream alongside a particular sample, until it reaches a sink or is forced to stop propagating by another block.

Stream tags are defined for a specific item in the data stream and are formed as a key:value pair. The "key" identifies what the "value" represents while the value holds the data that the tag contains. Both "key" and "value" are [[Polymorphic Types (PMTs)]] where the "key" is a PMT symbol while the "value" is any type of PMT and can therefore handle any data we wish to pass. An additional part of the tag is the "srcid", which is a PMT symbol and is used to identify the block that created the tag (which is usually the block's alias).

## == Background ==

GNU Radio was originally a streaming system with no other mechanism to pass data between blocks. Streams of data are a model that work well for samples, bits, etc., but can lack for control and metadata.

Part of this is solved using the existing message passing interface, which allows blocks to subscribe to messages published by any other block in the flowgraph (see [[Message Passing]]). The main drawback to the message passing system is that it works asynchronously, meaning that there is no guarantee when a message may arrive relative to the data stream.

## == API Extensions to the gr::block ==

To enable the stream tags, we have extended the API of gr::block to understand "absolute" item numbers. In the data stream model, each block's work function is given a buffer in the data stream that is referenced from 0 to N-1. This is a "relative" offset into the data stream. The absolute reference starts from the beginning of the flowgraph and continues to count up with every item. Each input stream is associated with a concept of the 'number of items read' and each output stream has a 'number of items written'. These are retrieved during runtime using the two API calls:

```
unsigned long int nitems_read(unsigned int which_input);
unsigned long int nitems_written(unsigned int which_output);
```

Each tag is associated with some item in this absolute time scale that is calculated using these functions.

Like the rest of the data stream, the number of items read/written are only updated once during the call to work. So in a work function, `nitems_read/written` will refer to the state of the data stream at the start of the work function. We must therefore add to this value the current relative offset in the data stream. So if we are iterating "i" over all output items, we would write the stream tag to output ports at `nitems_written(0)+i` for the 0th output port.

### == Stream Tags API ==

The stream tags API is split into two parts: adding tags to a stream, and getting tags from a stream.

Note that the functions described below are only meant to be accessed within a call to `general_work/work`. While they can be called at other points in time by a block, the behavior outside of work is undefined without exact knowledge of the item counts in the buffers.

### === Adding a Tag to a Stream ===

We add a tag to a particular output stream of the block using:

\* `gr::block::add_item_tag`: Adds an item tag to a particular output port using a `gr::tag_t` data type or by specifying the tag values.

We can output them to multiple output streams if we want, but to do so means calling this function once for each port. This function can be provided with a `gr::tag_t` data type, or each value of the tag can be explicitly given.

Again, a tag is defined as:

- # offset: The offset, in absolute item time, of the tag in the data stream.
- # key: the PMT symbol identifying the type of tag.
- # value: the PMT holding the data of the tag.
- # srcid: (optional) the PMT symbol identifying the block which created the tag.

We can create a `gr::tag_t` structure to hold all of the above

information of a tag, which is probably the easiest/best way to do it. The `gr::tag_t` struct is defined as having the same members as in the above list. To add a `gr::tag_t` tag to a stream, use the function:

```
void add_item_tag(unsigned int which_output, const tag_t &tag);
```

The secondary API allows us to create a tag by explicitly listing all of the tag information in the function call:

```
void add_item_tag(unsigned int which_output,
                  uint64_t abs_offset,
                  const pmt::pmt_t &key,
                  const pmt::pmt_t &value,
                  const pmt::pmt_t &srcid=pmt::PMT_F);
```

In Python, we can add a tag to a stream using one of the following:

```
add_item_tag(which_output, abs_offset, key, value)
add_item_tag(which_output, abs_offset, key, value, srcid)
```

Note that the key and value are both PMTs. To create a string type PMT you can use `pmt.intern("example_key")`.

Consider the following flowgraph as an example. We will have an embedded python block insert stream tags at random intervals, and view these tags on the QT GUI Time sink.

[[File:Epy\_stream\_tags\_example\_flowchart.png|800px]]

Add the following code into the embedded python block. This code outputs the same signal as in the input, except with tags on randomly selected input samples:

```
<syntaxhighlight lang="python">
import numpy as np
from gnuradio import gr
import pmt

class blk(gr.sync_block):
    def __init__(self):
        gr.sync_block.__init__(
            self,
            name='Embedded Python Block',
            in_sig=[np.float32],
            out_sig=[np.float32])
```

```

    )

    def work(self, input_items, output_items):
        for indx, sample in enumerate(input_items[0]): # Enumerate through the
input samples in port in0
            if np.random.rand() > 0.95: # 5% chance this sample is chosen
                key = pmt.intern("example_key")
                value = pmt.intern("example_value")
                self.add_item_tag(0, # Write to output port 0
                                self.nitems_written(0) + indx, # Index of the tag in
absolute terms
                                key, # Key of the tag
                                value # Value of the tag
                                )
                # note: (self.nitems_written(0) + indx) is our current sample, in
absolute time
            output_items[0][:] = input_items[0] # copy input to output
            return len(output_items[0])
</syntaxhighlight>

```

You may expect an output similar to the screenshot of the time sink below.

[[File:Epy\_stream\_tags\_example\_timesink.png|800px]]

=== Getting tags from a Stream ===

To get tags from a particular input stream, we have two functions we can use:

# gr::block::get\_tags\_in\_range: Gets all tags from a particular input port between a certain range of items (in absolute item time).

# gr::block::get\_tags\_in\_window: Gets all tags from a particular input port between a certain range of items (in relative item time within the work function).

The difference between these functions is working in absolute item time versus relative item time. Both of these pass back vectors of `gr::tag_t`, and they both allow specifying a particular key (as a PMT symbol) to filter against (or the fifth argument can be left out to search for all keys). Filtering for a certain key reduces the effort inside the work function for getting the right tag's data.

For example, this call just returns any tags between the given range of items:

```
void get_tags_in_range(std::vector<tag_t> &v,
                      unsigned int which_input,
                      uint64_t abs_start,
                      uint64_t abs_end);
```

Adding a fifth argument to this function allows us to filter on the key "key".

```
void get_tags_in_range(std::vector<tag_t> &v,
                      unsigned int which_input,
                      uint64_t abs_start,
                      uint64_t abs_end,
                      const pmt::pmt_t &key);
```

In Python, the main difference from the C++ function is that instead of having the first argument be a vector where the tags are stored, the Python version just returns a list of tags. We would use it like this:

```
<syntaxhighlight lang="python">
def work(self, input_items, output_items):
    ....
    tags = self.get_tags_in_window(which_input, rel_start, rel_end)
    ....
</syntaxhighlight>
```

If you want to grab all tags on the samples currently being processed by work(), on input port 0, here's a minimal example of doing that:

```
<syntaxhighlight lang="python">
import numpy as np
from gnuradio import gr
import pmt

class blk(gr.sync_block):
    def __init__(self):
        gr.sync_block.__init__(self, name='Read Tags', in_sig=[np.float32],
                                out_sig=None)

    def work(self, input_items, output_items):
        tags = self.get_tags_in_window(0, 0, len(input_items[0]))
        for tag in tags:
            key = pmt.to_python(tag.key) # convert from PMT to python string
            value = pmt.to_python(tag.value) # Note that the type(value) can be
```

several things, it depends what PMT type it was

```
print 'key:', key
print 'value:', value, type(value)
print "
```

```
return len(input_items[0])
```

</syntaxhighlight>

== Tag Propagation ==

We now know how to add tags to streams, and how to read them. But what happens to tags after they were read? And what happens to tags that aren't used? After all, there are many blocks that don't care about tags at all.

The answer is: It depends on the "tag propagation policy" of a block what happens to tags that enter it.<br />

There are  
[[http://gnuradio.org/doc/doxygen/classgr\\_1\\_1block.html#abc40fd4d514724a5446a2b34b2352b4e](http://gnuradio.org/doc/doxygen/classgr_1_1block.html#abc40fd4d514724a5446a2b34b2352b4e) three policies] to choose from:

- # All-to-All: all tags from any input port are replicated to all output ports
- # One-to-One: tags from input port "i" are only copied to output port "i" (depends on num inputs = num outputs).
- # Dont: Does not propagate tags. Tags are either stopped here or the work function recreates them in some manner.

The default behavior of a block is the 'All-to-All' method of propagation.

We generally set the tag propagation policy in the block's constructor using  
[[@set\\_tag\\_propagation\\_policy](http://gnuradio.org/doc/doxygen/classgr_1_1block.html#a476e218927e426ac88c26431cbf086cd)]

```
void set_tag_propagation_policy(tag_propagation_policy_t p);
```

See the `gr::block::tag_propagation_policy_t` documentation for details on this enum type.

When the tag propagation policy is set to `TPP_ALL_TO_ALL` or `TPP_ONE_TO_ONE`, the GNU Radio scheduler uses any information available to figure out which output item corresponds to which input item. The block may read them and add new tags, but existing tags are automatically moved downstream in a manner deemed appropriate.

When a tag is propagated through a block that has a rate change, the item's offset in the data stream will change. The scheduler uses the

block's `gr::block::relative_rate` concept to perform the update on the tag's offset value. The relative rate of a block determines the relationship between the input rate and output rate. Decimators that decimate by a factor of  $D$  have a relative rate of  $1/D$ .

Synchronous blocks (`gr::sync_block`), decimators (`gr::sync_decimator`), and interpolators (`gr::sync_interpolator`) all have pre-defined and well-understood relative rates. A standard `gr::block` has a default relative rate of 1.0, but this must be set if it does not work this way. Often, we use a `gr::block` because we have no pre-conceived notion of the number of input to output items. If it is important to pass tags through these blocks that respect the change in item value, we would have to use the `TPP_DONT` tag propagation policy and handle the propagation internally.

In no case is the value of the tag modified when propagating through a block. This becomes relevant when using `[[Tagged Stream Blocks]]`.

As an example, consider an interpolating block. See the following flow graph:

`[[File:tags_interp.grc.png|700px|tags_interp.grc.png]]`

`[[File:tags_interp.png|500px|tags_interp.png]]`<br />

As you can tell, we produce tags on every 10th sample, and then pass them through a block that repeats every sample 100 times. Tags do not get repeated with the standard tag propagation policies (after all, they're not tag "manipulation" policies), so the scheduler takes care that every tag is put on the output item that corresponds to the input item it was on before. Here, the scheduler makes an educated guess and puts the tag on the "first" of 100 items.

Note: We can't use one QT GUI Time Sink for both signals here, because they are running at a different rate. Note the time difference on the x-axis!

On decimating blocks, the behavior is similar. Consider this very simple flow graph, and the position of the samples:

`[[File:tags_decim.grc.png|400px|tags_decim.grc.png]]`

`[[File:tags_decim.png|500px|tags_decim.png]]`<br />

We can see that no tags are destroyed, and tags are indeed spaced at one-tenth of the original spacing of 100 items. Of course, the actual item that was passed through the

block might be destroyed, or modified (think of a decimating FIR filter).

In fact, this works with any rate-changing block. Note that there are cases where the relation of tag positions of in- and output are ambiguous, the GNU Radio scheduler will then try and get as close as possible.

Here's another interesting example: Consider this flow graph, which has a delay block, and the position of the tags after it:

[[File:tags\_ramp\_delay.grc.png|500px|tags\_ramp\_delay.grc.png]]

[[File:tags\_ramp\_delay.png|500px|tags\_ramp\_delay.png]]<br />

Before the delay block, tags were positioned at the beginning of the ramp. After the delay, they're still in the same position! Would we inspect the source code of the delay block, we'd find that there is absolutely no tag handling code. Instead, the block [[http://gnuradio.org/doc/doxygen/classgr\\_1\\_1block.html#acad5d6e62ea885cb77d19f72451581c2](http://gnuradio.org/doc/doxygen/classgr_1_1block.html#acad5d6e62ea885cb77d19f72451581c2)] declares a delay to the scheduler, which then propagates tags with this delay.

Using these mechanisms, we can let GNU Radio handle tag propagation for a large set of cases. For specialized or corner cases, there is no option than to set the tag propagation policy to `TPP_DONT` and manually propagate tags (actually, there's another way: Say we want most tags to propagate normally, but a select few should be treated differently. We can use [[http://gnuradio.org/doc/doxygen/classgr\\_1\\_1block.html#a461f6cc92174e83b10c3ec5336036768](http://gnuradio.org/doc/doxygen/classgr_1_1block.html#a461f6cc92174e83b10c3ec5336036768)] `remove_item_tag()` (DEPRECATED. Will be removed in 3.8.) to remove these tags from the input; they will then not be propagated any more even if the tag propagation policy is set to something other than `TPP_DONT`. But that's more advanced use and will not be elaborated on here).

== Notes on How to Use Tags ==

Tags can be very useful to an application, and their use is spreading. USRP sources generate tag information on the time, sample rate, and frequency of the board if anything changes. We have a metadata file source/sink (see [\[\[Metadata\\_Information\]\]](#)) that use tags to store information about the data stream. But there are things to think about when using tags in a block.

First, when tags are not being used, there is almost no effect on the scheduler. However, when we use tags, we add overhead by getting and extracting tags from a data stream. We also use overhead in

propagating the tags. For each tag, each block must copy a vector of tags from the output port(s) of one block to the input port(s) of the next block(s). These copy operations can add up.

The key is to minimize the use of tags. Use them when and only when necessary and try to provide some control over how tags are generated to control their frequency. A good example is the USRP source, which generates a time tag. If it generated a tag with every sample, we would have thousands of tags per second, which would add a significant amount of overhead. This is because if we started at time  $t_0$  at sample rate  $sr$ , then after  $N$  samples, we know that we are now at time  $t_0 + N/sr$ . So continuously producing new tags adds no information.

The main issue we need to deal with in the above situation is when there is a discontinuity in the packets received from the USRP. Since we have no way of knowing in the flowgraph how many samples were potentially lost, we have lost track of the timing information. The USRP driver recognizes when packets have been dropped and uses this to queue another tag, which allows us to resync. Likewise, any point the sample rate or frequency changes, a new tag is issued.

== Example Flowgraph ==

Let's have a look at a simple example:

[[File:tut5\_tagtest\_fg.png|600px|tut5\_tagtest\_fg.png]]

In this flow graph, we have two sources: A sinusoid and a tag strobe. A tag strobe is a block that will output a constant tag, in this case, on every 1000th item (the actual value of the items is always zero). Those sources get added up. The signal after the adder is identical to the sine wave we produced, because we are always adding zeros. However, the tags stay attached to the same position as they were coming from the tag strobe! This means every 1000th sample of the sinusoid now has a tag. The QT scope can display tags, and even trigger on them.

[[File:tut5\_tagtest\_scope.png|500px|tut5\_tagtest\_scope.png]]<br />

We now have a mechanism to randomly attach any metadata to specific items. There are several blocks that use tags. One of them is the UHD Sink block, the driver used for transmitting with USRP devices. It will react to tags with certain keys, one of them being `tx_freq`, which can be used to set the transmit frequency of a USRP while streaming.

=== Adding tags to the QPSK demodulator ===

Going back to our [\[\[Guided\\_Tutorial\\_PSK\\_Demodulation|QPSK demodulation example\]\]](#), we might want to add a feature to tell downstream blocks that the demodulation is not going well. Remember the output of our block is always hard-decision, and we have to output something. So we could use tags to notify that the input is not well formed, and that the output is not reliable.

As a failure criterion, we discuss the case where the input amplitude is too small, say smaller than 0.01. When the amplitude drops below this value, we output one tag. Another tag is only sent when the amplitude has recovered, and rises back above the threshold. We extend our work function like this:

```
<pre>if (std::abs(in[i]) < 0.01 and not d_low_ampl_state) {
    add_item_tag(0, // Port number
                 nitems_written(0) + i, // Offset
                 pmt::mp("&quot;amplitude_warning&quot;"), // Key
                 pmt::from_double(std::abs(in[i])) // Value
    );
    d_low_ampl_state = true;
}
else if (std::abs(in[i]) >= 0.01 and d_low_ampl_state) {
    add_item_tag(0, // Port number
                 nitems_written(0) + i, // Offset
                 pmt::mp("&quot;amplitude_recovered&quot;"), // Key
                 pmt::PMT_T // Value
    );
    d_low_ampl_state = false; // Reset state
}
</pre>
```

In Python, the code would look like this (assuming we have a member of our block class called `d_low_ampl_state`):

```
<pre># The vector 'in' is called 'in0' here because 'in' is a Python keyword
if abs(in0[i]) < 0.01 and not d_low_ampl_state:
    self.add_item_tag(0, # Port number
                      self.nitems_written(0) + i, # Offset
                      pmt.intern("&quot;amplitude_warning&quot;"), # Key
                      pmt.from_double(numpy.double(abs(in0[i]))) # Value
                      # Note: We need to explicitly create a 'double' here,
                      # because in0[i] is an explicit 32-bit float here
    )
    self.d_low_ampl_state = True
elif abs(in0[i]) >= 0.01 and d_low_ampl_state:
    self.add_item_tag(0, # Port number
```

```

        self.nitems_written(0) + i, # Offset
        pmt.intern("&quot;amplitude_recovered&quot;"), # Key
        pmt.PMT_T # Value
    )
    self.d_low_ampl_state = False; // Reset state

```

We can also create a tag data type [http://gnuradio.org/doc/doxygen/structgr\_1\_1tag\_\_t.html tag\_t] and directly pass this along:

```

if (std::abs(in[i]) < 0.01 and not d_low_ampl_state) {
    tag_t tag;
    tag.offset = nitems_written(0) + i;
    tag.key = pmt::mp("&quot;amplitude_warning&quot;");
    tag.value = pmt::from_double(std::abs(in[i]));
    add_item_tag(0, tag);
    d_low_ampl_state = true;
}

```

Here's a flow graph that uses the tagged version of the demodulator. We input 20 valid QPSK symbols, then 10 zeros. Since the output of this block is always either 0, 1, 2 or 3, we normally have no way to see if the input was not clearly one of these values.

[[File:Demod\_with\_tags.grc.png|750px|Demod\_with\_tags.grc.png]]

Here's the output. You can see we have tags on those values which were not from a valid QPSK symbol, but from something unreliable.

[[File:Demod\_with\_tags.png|500px|Demod\_with\_tags.png]]<br />

== Use case: FIR filters ==

Assume we have a block that is actually an FIR filter. We want to let GNU Radio handle the tag propagation. How do we configure the block?

Now, an FIR filter has one input and one output. So, it doesn't matter if we set the propagation policy to `TPP_ALL_TO_ALL` or `TPP_ONE_TO_ONE`, and we can leave it as the default. The items going in and those coming out are different, so how do we match input to output? Since we want to preserve the timing of the tag position, we need to use the filter's "group delay" as a delay for tags (which for this symmetric FIR filter is `(N-1)/2`, where `N` is the number of filter taps). Finally, we might be interpolating, decimating or both (say, for sample rate changes) and we need to tell the scheduler about this as well.

```

<sha1>fgw53qlkhiuz5ip4ps5l00yofnbq6eq</sha1>
</revision>

```

```
</page>
</mediawiki>
```

## 12、 (exported from wiki) Tagged Stream Blocks

```
<page>
  <title>Tagged Stream Blocks</title>
  <ns>0</ns>
  <id>3483</id>
  <revision>
    <id>14629</id>
    <parentid>13791</parentid>
    <timestamp>2025-01-06T12:38:23Z</timestamp>
    <contributor>
      <username>MarcusMueller</username>
      <id>10</id>
    </contributor>
    <comment>/* A note on tag propagation */</comment>
    <origin>14629</origin>
    <model>wikitext</model>
    <format>text/x-wiki</format>
    <text      bytes="13619"      sha1="nurueqlkuodtbl5fqtfruc3tssymkdx"
xml:space="preserve">[[Category:Usage Manual]]
== Introduction ==
```

A tagged stream block is a block that works on streamed, but packetized input data. Think of packet data transmission: A data packet consists of N bytes. However, in traditional GNU Radio blocks, if we stream N bytes into a block, there's no way of knowing the packet boundary. This might be relevant: Perhaps the modulator has to prepend a synchronization word before every packet or append a CRC. So while some blocks don't care about packet boundaries, other blocks do: These are tagged stream blocks.

These blocks are different from all the other GNU Radio block types (gr::block, gr::sync\_block etc.) in that they are driven by the input: The PDU length tag tells the block how to operate, whereas other blocks are output-driven (the scheduler tries to fill up the output buffer as much as possible).

=== How do they work? ===

As the name implies, tagged stream blocks use tags to identify PDU boundaries. On the first item of a streamed PDU, there "must" be a tag with a specific key, which stores the length of the PDU as a PMT integer. If anything else, or no tag, is on this first item, this will cause the flow graph to crash!

The scheduler then takes care of everything. When the work function is called, it is guaranteed to contain exactly one complete PDU and enough space in the output buffer for the output.

=== How do they relate to other block types (e.g. sync blocks)? ===

Tagged stream blocks and sync blocks are really orthogonal concepts, and a block could be both (`gr::digital::ofdm_frame_equalizer_vcvc` is such a block). However, because the work function is defined differently in these block types, there is no way to derive a block from both `gr::tagged_stream_block` and `gr::sync_block`.

If a block needs the tagged stream mechanism (i.e. knowing about PDU boundaries), it must be derived from `gr::tagged_stream_block`. If it's also a sync block, it is still possible to set `gr::block::set_relative_rate(1.0)` and/or the fixed rate functions.

The way `gr::tagged_stream_block` works, it is still beneficial to specify a relative rate, if possible.

== Creating a tagged stream block ==

To create a tagged stream block, the block must be derived from `gr::tagged_stream_block`.

Here's a minimal example of how the header file could look:

```
<syntaxhighlight lang="cpp">
#include <gnuradio/digital/api.h>
#include <gnuradio/tagged_stream_block.h>

namespace gr {
    namespace digital {

        class DIGITAL_API crc32_bb : virtual public tagged_stream_block
        {
        public:
            typedef boost::shared_ptr<crc32_bb> sptr;

            static sptr make(bool check=false, const std::string&
len_tag_key="packet_len");
        };
    }
}
```

```

    } // namespace digital
} // namespace gr
</syntaxhighlight>

```

It is very similar to any other block definition. Two things stand out: First, `gnuradio/tagged_stream_block.h` is included to allow deriving from `gr::tagged_stream_block`.

The other thing is in the `make` function: the second argument is a string containing the key of the length tags. This is not necessary (the block could get this information hard-coded), but the usual way is to allow the user to change this tag, but give a default value (in this case, `packet_len`).

The implementation header (`*_impl.h`) also looks a bit different (again this is cropped to the relevant parts):

```

<syntaxhighlight lang="cpp">
#include <digital/crc32_bb.h>
namespace gr {
    namespace digital {

        class crc32_bb_impl : public crc32_bb
        {
        public:
            crc32_bb_impl(bool check, const std::string& len_tag_key);
            ~crc32_bb_impl();

            int calculate_output_stream_length(const gr_vector_int &ninput_items);
            int work(int noutput_items,
                    gr_vector_int &ninput_items,
                    gr_vector_const_void_star &input_items,
                    gr_vector_void_star &output_items);
        };

    } // namespace digital
} // namespace gr
</syntaxhighlight>

```

First, the `work` function signature is new. The argument list looks like that from `gr::block::general_work()` (note: the arguments mean the same, too), but it's called `work` like with the other derived block types (such as `gr::sync_block`). Also, there's a new function: `calculate_output_stream_length()` is, in a sense, the opposite function to

gr::block::forecast(). Given a number of input items, it calculates the required number of output items. Note how this relates to the fact that these blocks are input-driven.

These two overrides (work() and calculate\_output\_stream\_length()) are what you need for most tagged stream blocks. There are cases when you don't need to override the latter

because the default behaviour is enough, and other cases where you have to override more

than these two functions. These are discussed in [\[\[Tagged Stream Blocks#Advanced Usage|Advanced Usage\]\]](#).

Finally, this is part of the actual block implementation (heavily cropped again, to highlight the relevant parts):

```
<syntaxhighlight lang="cpp">
#include <gnuradio/io_signature.h>
#include "crc32_bb_impl.h"

namespace gr {
    namespace digital {

        crc32_bb::sptr crc32_bb::make(bool check, const std::string& len_tag_key)
        {
            return gnuradio::get_initial_sptr (new crc32_bb_impl(check, len_tag_key));
        }

        crc32_bb_impl::crc32_bb_impl(bool check, const std::string& len_tag_key)
            : tagged_stream_block("crc32_bb",
                                  io_signature::make(2, 1, sizeof(char)),
                                  io_signature::make(1, 1, sizeof(char)),
                                  len_tag_key),
              d_check(check)
        {
        }

        int
        crc32_bb_impl::calculate_output_stream_length(const gr_vector_int
        &ninput_items)
        {
            if (d_check) {
                return ninput_items[0] - 4;
            } else {
```

```

        return ninput_items[0] + 4;
    }
}

int
crc32_bb_impl::work (int noutput_items,
                    gr_vector_int &ninput_items,
                    gr_vector_const_void_star &input_items,
                    gr_vector_void_star &output_items)
{
    const unsigned char *in = (const unsigned char *) input_items[0];
    unsigned char *out = (unsigned char *) output_items[0];

    // Do all the signal processing...
    // Don't call consume!

    return new_packet_length;
}

} // namespace digital
} // namespace gr
</syntaxhighlight>

```

The make function is not different to any other block. The constructor calls `gr::tagged_stream_block::tagged_stream_block()` as expected, but note that it passes the key of the length tag to the parent constructor.

The block in question is a CRC block, and it has two modes: It can check the CRC (which is then removed), or it can append a CRC to a sequence of bytes. The `calculate_output_stream_length()` function is thus very simple: depending on how the block is configured, the output is either 4 bytes longer or shorter than the input stream.

The `work()` function looks very similar to any other work function. When writing the signal processing code, the following things must be kept in mind:

- The work function is called for exactly one PDU, and no more (or less) may be processed
- `ninput_items` contains the exact number of items in this PDU (at every port). These items "will" be consumed after `work()` exits.
- Don't call `consume()` or `consume_each()` yourself! `gr::tagged_stream_block` will do that for you.
- You can call `produce()` or `produce_each()`, if you're doing something complicated. Don't forget to return `WORK_CALLED_PRODUCE` in that case.

=== A note on tag propagation ===

Despite using tags for a special purpose, all tags that are not the length tag are treated exactly as before: use `gr::block::set_tag_propagation_policy()` in the constructor. The default is `TPP_ALL_TO_ALL` behaviour (this means copying all non-length-key tags from all inputs to all outputs).

In a lot of the cases, though, you will need to specify `set_tag_propagation_policy(TPP_DONT)` and manually handle the tag propagation in `work()`. This is because the unknown length of the PDUs at compile time prohibits us from setting a precise relative rate of the block, which is a requirement for automatic tag propagation. Only if the tagged stream block is also a sync block (including interpolators and decimators, i.e. blocks with an integer rate change), can automatic tag propagation be reliably used.

It is a general rule of GNU Radio blocks that they "properly" propagate tags, whatever this means for a specific application.

The CRC block seems to a very simple block, but it's already complicated enough to confuse the automatic tag propagation. For example, what happens to tags which are on the CRC? Do they get removed, or do they get moved to the last item before the CRC? Also, the input to output rate is different for every PDU length.

In this case, it is necessary for the developer to define a tag propagation policy and implement it in `work()`. Also, it is good practice to specify that tag propagation policy in the blocks documentation.

The actual length tags "are" treated differently, though. Most importantly, you don't have to write the new length tag yourself. The key for the output length tag is the same as that on the input, if you don't want this, you must override `gr::tagged_stream_block::update_length_tags()`.

== Connecting regular streaming blocks and tagged stream blocks ==

From the scheduler's point of view, all blocks are equivalent, and as long as the I/O signatures are compatible, all of these blocks can be connected.

However, it is important to note that tagged stream blocks expect correctly tagged streams, i.e. a length tag with a number of items at the beginning of every packet. If this is not the case, the "flow graph will crash". The most common cases are discussed separately:

Connecting a tagged stream block to a regular stream block: This is never a problem, since regular blocks don't care about the values of the tags.

Connecting regular stream blocks to tagged stream blocks: This will usually not work. One solution is to use the `gr::blocks:stream_to_tagged_stream` adapter block. It will periodically add tags at regular intervals, making the input valid for the tagged stream block. Make sure to directly connect this block to the tagged stream block, or the packet size might be changed to a different value from the tag value.

Mixing block types: This is possible if none of the regular stream blocks change the rate. The `ofdm_tx` and `ofdm_rx` hierarchical blocks do this.

#### == Advanced Usage ==

It is generally recommended to read the block documentation of `gr::tagged_stream_block`.

A few special cases are described here:

#### === Multiple length tags ===

In some cases, a single tag is not enough. One example is the OFDM receiver: one OFDM frame contains a certain number of OFDM symbols, and another number of bytes--these numbers are only very loosely related, and one cannot be calculated from the other.

`gr::digital::ofdm_serializer_vcc` is such a block. It is driven by the number of OFDM frames, but the output is determined by the number of complex symbols. In order to use multiple length tag keys, it overrides `gr::tagged_stream_block::update_length_tags()`.

#### === Falling back to gr::block behavior ===

If, at compile-time, it is uncertain whether or not a block should be a `gr::tagged_stream_block`, there is the possibility of falling back to `gr::block` behaviour.

To do this, simply don't pass an empty string as length tag key. Instead of crashing,

a tagged stream block will behave like a `gr::block`.

This has some consequences: The work function must have all the elements of a `gr::block::general_work()` function, including calls to `consume()`. Because such a block must

allow both modes of operation (PDUs with tags, and infinite-stream), the work function must check which mode is currently relevant. Checking if `gr::tagged_stream_block::d_length_tag_key_str` is empty is a good choice.

`gr::digital::ofdm_cyclic_prefixer` implements this.

=== Other ways to determine the number of input items ===

If the number of input items is not stored as a `pmt::pmt_integer`, but there is a way to determine

it, `gr::tagged_stream_block::parse_length_tags()` can be overridden to figure out the length of the PDU.

== Examples ==

=== CRC32 ===

Block: [\[https://raw.githubusercontent.com/gnuradio/gnuradio/master/gr-digital/lib/crc32\\_bb\\_impl.cc\]](https://raw.githubusercontent.com/gnuradio/gnuradio/master/gr-digital/lib/crc32_bb_impl.cc) `gr-digital/lib/crc32_bb_impl.cc`

This is a very simple block, and a good example to start with.

=== OFDM Frame Equalizer ===

Block: [\[https://raw.githubusercontent.com/gnuradio/gnuradio/master/gr-digital/lib/ofdm\\_frame\\_equalizer\\_vcvc\\_impl.cc\]](https://raw.githubusercontent.com/gnuradio/gnuradio/master/gr-digital/lib/ofdm_frame_equalizer_vcvc_impl.cc) `gr-digital/lib/ofdm_frame_equalizer_vcvc_impl.cc`

This block would be a sync block if tagged stream blocks didn't exist. It also uses more than one tag to determine the output.

=== Tagged Stream Muxer ===

Block: [\[https://raw.githubusercontent.com/gnuradio/gnuradio/master/gr-blocks/lib/tagged\\_stream\\_mux\\_impl.cc\]](https://raw.githubusercontent.com/gnuradio/gnuradio/master/gr-blocks/lib/tagged_stream_mux_impl.cc) `gr-blocks/lib/tagged_stream_mux_impl.cc`

Use this to multiplex any number of tagged streams.

=== Cyclic Prefixer (OFDM) ===

Block: [\[https://raw.githubusercontent.com/gnuradio/gnuradio/master/gr-digital/lib/ofdm\\_cyclic\\_prefixer\\_impl.cc\]](https://raw.githubusercontent.com/gnuradio/gnuradio/master/gr-digital/lib/ofdm_cyclic_prefixer_impl.cc) gr-digital/lib/ofdm\_cyclic\_prefixer\_impl.cc]

This block uses the gr::block behavior fallback.

== Troubleshooting ==

"My flow graph crashes with the error message "Missing length tag"."

This means the input of a tagged stream block was not correctly tagged. The most common cause is when connecting a regular streaming block to a tagged stream block. You can check the log output for the item number and port where this happened.</text>

```
<sha1>nurueqlkuodtbl5fqtruc3tssymkdx</sha1>
</revision>
</page>
</mediawiki>
```

## 13、 (exported from wiki) Types of Blocks

```
<page>
<title>Types of Blocks</title>
<ns>0</ns>
<id>3495</id>
<revision>
<id>9094</id>
<parentid>8843</parentid>
<timestamp>2021-11-17T16:00:00Z</timestamp>
<contributor>
<username>777arc</username>
<id>632</id>
</contributor>
<comment>/* Basic Block */</comment>
<origin>9094</origin>
<model>wikitext</model>
<format>text/x-wiki</format>
<text bytes="8027" sha1="j1gt0ofen1ui79z2lk7vhsyffax6jr5"
xml:space="preserve">[[Category:Usage Manual]]
== Introduction ==
```

To take advantage of the gnuradio framework, users will create various blocks to implement the desired data processing. There are several types of blocks to choose from:

- \* Synchronous Blocks (1:1)
- \* Decimation Blocks (N:1)
- \* Interpolation Blocks (1:M)
- \* Basic (a.k.a. General) Blocks (N:M)

== Synchronous Block ==

The sync block allows users to write blocks that consume and produce an equal number of items per port. A sync block may have any number of inputs or outputs. When a sync block has zero inputs, its called a source. When a sync block has zero outputs, its called a sink.

An example sync block in C++:

```
<syntaxhighlight lang="cpp">
#include <gr_sync_block.h>

class my_sync_block : public gr_sync_block
{
public:
    my_sync_block(...):
        gr_sync_block("my block",
                      gr_make_io_signature(1, 1, sizeof(int32_t)),
                      gr_make_io_signature(1, 1, sizeof(int32_t)))
    {
        //constructor stuff
    }

    int work(int noutput_items,
             gr_vector_const_void_star &input_items,
             gr_vector_void_star &output_items)
    {
        //work stuff...
        return noutput_items;
    }
};
</syntaxhighlight>
```

Some observations:

- \* noutput\_items is the length in items of all input and output buffers
- \* an input signature of gr\_make\_io\_signature(0, 0, 0) makes this a source block
- \* an output signature of gr\_make\_io\_signature(0, 0, 0) makes this a sink block

An example sync block in Python:

```
<syntaxhighlight lang="python">
class my_sync_block(gr.sync_block):
    def __init__(self):
        gr.sync_block.__init__(self,
                                name = "my sync block",
                                in_sig = [numpy.float32, numpy.float32],
                                out_sig = [numpy.float32],
                                )
    def work(self, input_items, output_items):
        output_items[0][:] = input_items[0] + input_items[1]
        return len(output_items[0])
</syntaxhighlight>
```

The input\_items and output\_items are lists of lists. The input\_items contains a vector of input samples for every input stream, and the output\_items is a vector for each output stream where we can place items. Then length of output\_items[0] is equivalent to the noutput\_items concept we are so familiar with from the C++ blocks.

Some observations:

- \* The length of all input vector and all output vectors is identical
- \* in\_sig=None would turn this into a source block
- \* out\_sig=None would turn this into a sink block. In this case, use len(input\_items[0]) since output\_items is empty!
- \* Unlike in C++ where we use the gr::io\_signature class, here we can just create a Python list of the I/O data sizes using numpy data types, e.g.: numpy.int8, numpy.int16, numpy.float32

== Decimation Block ==

The decimation block is another type of fixed rate block where the number of input items is a fixed multiple of the number of output items.

An example decimation block in c++

```
<syntaxhighlight lang="cpp">
#include <gr_sync_decimator.h>

class my_decim_block : public gr_sync_decimator
{
public:
    my_decim_block(...):
```

```

    gr_sync_decimator("my decim block",
                      in_sig,
                      out_sig,
                      decimation)

{
    //constructor stuff
}

//work function here...
};
</syntaxhighlight>

```

Some observations:

- \* The gr\_sync\_decimator constructor takes a 4th parameter, the decimation factor
- \* The user should assume that the number of input items = noutput\_items\*decimation

An example decimation block in Python:

```

<syntaxhighlight lang="python">
class my_decim_block(gr.decim_block):
    def __init__(self, decim_rate):
        gr.decim_block.__init__(self,
                                name="my block",
                                in_sig=[numpy.float32],
                                out_sig=[numpy.float32],
                                decim = decim_rate)
        self.set_relative_rate(1.0/decim_rate)
        self.decimation = decim_rate

    def work(self, input_items, output_items):
        output_items[0][:] = input_items[0][0::self.decimation]
        return len(output_items[0])
</syntaxhighlight>

```

Some observations:

- \* The set\_relative\_rate call configures the input/output relationship
- \* To set an interpolation, use self.set\_relative\_rate(interpolation)
- \* The following will be true  $\text{len}(\text{input\_items}[i]) = \text{len}(\text{output\_items}[j]) * \text{decimation}$

== Interpolation Block ==

The interpolation block is another type of fixed rate block where the number of output

items is a fixed multiple of the number of input items.

An example interpolation block in c++

```
<syntaxhighlight lang="cpp">
#include <gr_sync_interpolator.h>

class my_interp_block : public gr_sync_interpolator
{
public:
    my_interp_block(...):
        gr_sync_interpolator("my interp block",
                               in_sig,
                               out_sig,
                               interpolation)
    {
        //constructor stuff
    }

    //work function here...
};
</syntaxhighlight>
```

Some observations:

- \* The gr\_sync\_interpolator constructor takes a 4th parameter, the interpolation factor
- \* The user should assume that the number of input items = noutput\_items/interpolation

An example interpolation block in Python:

```
<syntaxhighlight lang="python">
class my_interp_block(gr.interp_block):
    def __init__(self, args):
        gr.interp_block.__init__(self,
                                   name="my block",
                                   in_sig=[numpy.float32],
                                   out_sig=[numpy.float32])
        self.set_relative_rate(interpolation)

    #work function here...
</syntaxhighlight>
```

== Basic Block ==

The basic block provides no relation between the number of input items and the number

of output items. All other blocks are just simplifications of the basic block. Users should choose to inherit from basic block when the other blocks are not suitable.

The adder revisited as a basic block in C++:

```
<syntaxhighlight lang="cpp">
#include <gr_block.h>

class my_basic_block : public gr_block
{
public:
    my_basic_adder_block(...):
        gr_block("another adder block",
                in_sig,
                out_sig)
    {
        //constructor stuff
    }

    int general_work(int noutput_items,
                    gr_vector_int &ninput_items,
                    gr_vector_const_void_star &input_items,
                    gr_vector_void_star &output_items)
    {
        //cast buffers
        const float* in0 = reinterpret_cast(input_items[0]);
        const float* in1 = reinterpret_cast(input_items[1]);
        float* out = reinterpret_cast(output_items[0]);

        //process data
        for(size_t i = 0; i < noutput_items; i++) {
            out[i] = in0[i] + in1[i];
        }

        //consume the inputs
        this->consume(0, noutput_items); //consume port 0 input
        this->consume(1, noutput_items); //consume port 1 input
        //this->consume_each(noutput_items); //or shortcut to consume on all inputs

        //return produced
        return noutput_items;
    }
};
</syntaxhighlight>
```

Some observations:

- \* This class overloads the `general_work()` method, not `work()`
- \* The general work has a parameter: `ninput_items`
- \*\* `ninput_items` is a vector describing the length of each input buffer
- \* Before return, `general_work` must manually consume the used inputs
- \* The number of items in the input buffers is assumed to be `noutput_items`
- \*\* This behaviour can be altered by overloading the `forecast()` method but is not mandatory

The adder revisited as a basic block in Python:

```
<syntaxhighlight lang="python">
import numpy as np
from gnuradio import gr

class my_basic_adder_block(gr.basic_block):
    def __init__(self):
        gr.basic_block.__init__(self,
                                name="another_adder_block",
                                in_sig=[np.float32, np.float32],
                                out_sig=[np.float32])

    def general_work(self, input_items, output_items):
        #buffer references
        in0 = input_items[0][:len(output_items[0])]
        in1 = input_items[1][:len(output_items[0])]
        out = output_items[0]

        #process data
        out[:] = in0 + in1

        #consume the inputs
        self.consume(0, len(in0)) #consume port 0 input
        self.consume(1, len(in1)) #consume port 1 input

        #return produced
        return len(out)
</syntaxhighlight></text>
<sha1>jlgt0ofen1ui79z2lk7vhsyffax6jr5</sha1>
</revision>
</page>
```

</mediawiki>

## 14、 (exported from wiki) VOLK Guide

```
<page>
  <title>VOLK Guide</title>
  <ns>0</ns>
  <id>3491</id>
  <revision>
    <id>4875</id>
    <parentid>4296</parentid>
    <timestamp>2019-03-12T22:44:00Z</timestamp>
    <contributor>
      <username>777arc</username>
      <id>632</id>
    </contributor>
    <origin>4875</origin>
    <model>wikitext</model>
    <format>text/x-wiki</format>
    <text      bytes="6049"      sha1="751jgw6850j6u6uc1ltkylfgnv85dac"
xml:space="preserve">[[Category:Usage Manual]]
== Introduction ==
```

Note: Many blocks have already been converted to use VOLK in their calls, so they can also serve as examples. See the `gr::blocks::complex_to_<type>.h` files for examples of various blocks that make use of VOLK.

VOLK is the Vector-Optimized Library of Kernels. It is a library that contains kernels of hand-written SIMD code for different mathematical operations. Since each SIMD architecture can be greatly different and no compiler has yet come along to handle vectorization properly or highly efficiently, VOLK approaches the problem differently. For each architecture or platform that a developer wishes to vectorize for, a new proto-kernel is added to VOLK. At runtime, VOLK will select the correct proto-kernel. In this way, the users of VOLK call a kernel for performing the operation that is platform/architecture agnostic. This allows us to write portable SIMD code.

VOLK kernels are always defined with a 'generic' proto-kernel, which is written in plain C. With the generic kernel, the kernel becomes portable to any platform. Kernels are then extended by adding proto-kernels for new platforms in which they are desired.

A good example of a VOLK kernel with multiple proto-kernels defined is

the `volk_32f_s32f_multiply_32f_a`. This kernel implements a scalar multiplication of a vector of floating point numbers (each item in the vector is multiplied by the same value). This kernel has the following proto-kernels that are defined for 'generic,' 'avx,' 'sse,' and 'neon'

```
void volk_32f_s32f_multiply_32f_a_generic
void volk_32f_s32f_multiply_32f_a_sse
void volk_32f_s32f_multiply_32f_a_avx
void volk_32f_s32f_multiply_32f_a_neon
```

These proto-kernels means that on platforms with AVX support, VOLK can select this option or the SSE option, depending on which is faster. If all else fails, VOLK can fall back on the generic proto-kernel, which will always work.

See [<http://libvolk.org> `libvolk.org`] for details on the VOLK naming scheme.

### == Setting and Using Memory Alignment Information ==

For VOLK to work as best as possible, we want to use memory-aligned SIMD calls, which means we have to have some way of knowing and controlling the alignment of the buffers passed to `gr_block`'s work function. We set the alignment requirement for SIMD aligned memory calls with:

```
const int alignment_multiple =
    volk_get_alignment() / output_item_size;
set_alignment(std::max(1,alignment_multiple));
```

The VOLK function '`volk_get_alignment`' provides the alignment of the the machine architecture. We then base the alignment on the number of output items required to maintain the alignment, so we divide the number of alignment bytes by the number of bytes in an output items (`sizeof(float)`, `sizeof(gr_complex)`, etc.). This value is then set per block with the '`set_alignment`' function.

Because the scheduler tries to optimize throughput, the number of items available per call to work will change and depends on the availability of the read and write buffers. This means that it sometimes cannot produce a buffer that is properly memory aligned. This is an inevitable consequence of the scheduler system. Instead of requiring alignment, the scheduler enforces the alignment as much as possible, and when a buffer becomes unaligned, the scheduler will work to correct it as much as possible. If a

block's buffers are unaligned, then, the scheduler sets a flag to indicate as much so that the block can then decide what best to do. The next section discusses the use of the aligned/unaligned information in a `gr_block`'s work function.

== Calling VOLK kernels in `Work()` ==

The buffers passed to `work/general_work` in a `gr_block` are not guaranteed to be aligned, but they will mostly be aligned whenever possible. When not aligned, the '`is_unaligned()`' flag will be set so the scheduler knows to try to realign the buffers. We actually make calls to the VOLK dispatcher, which is mainly designed to check the buffer alignments and call the correct version of the kernel for us. From the user-level view of VOLK, calling the dispatcher allows us to ignore the concept of aligned versus unaligned. This looks like:

```
<syntaxhighlight lang="cpp">
int
gr_some_block::work (int noutput_items,
                    gr_vector_const_void_star &input_items,
                    gr_vector_void_star &output_items)
{
    const float *in = (const float *) input_items[0];
    float *out = (float *) output_items[0];

    // Call the dispatcher to check alignment and call the _a or _u
    // version of the kernel.
    volk_32f_something_32f(out, in, noutput_items);

    return noutput_items;
}
</syntaxhighlight>
```

== Tuning VOLK Performance ==

VOLK comes with a profiler that will build a config file for the best SIMD architecture for your processor. Run `volk_profile` that is installed into `$PREFIX/bin`. This program tests all known VOLK kernels for each architecture supported by the processor. When finished, it will write to `$HOME/.volk/volk_config` the best architecture for the VOLK function. This file is read when using a function to know the best version of the function to execute.

=== Hand-Tuning Performance ===

If you know a particular architecture works best for your processor, you can specify the particular architecture to use in the VOLK preferences file: \$HOME/.volk/volk\_config

The file looks like:

```
volk_<FUNCTION_NAME> <ARCHITECTURE>
```

Where the "FUNCTION\_NAME" is the particular function that you want to over-ride the default value and "ARCHITECTURE" is the VOLK SIMD architecture to use (generic, sse, sse2, sse3, avx, etc.). For example, the following config file tells VOLK to use SSE3 for the aligned and unaligned versions of a function that multiplies two complex streams together.

```
volk_32fc_x2_multiply_32fc_a sse3
volk_32fc_x2_multiply_32fc_u sse3
```

'''Tip:''' If benchmarking GNU Radio blocks, it can be useful to have a volk\_config file that sets all architectures to 'generic' as a way to test the vectorized versus non-vectorized implementations.</text>

```
<sha1>751jgw6850j6u6uclltkylfgnv85dac</sha1>
```

```
</revision>
```

```
</page>
```

```
</mediawiki>
```