Message Passing

Contents

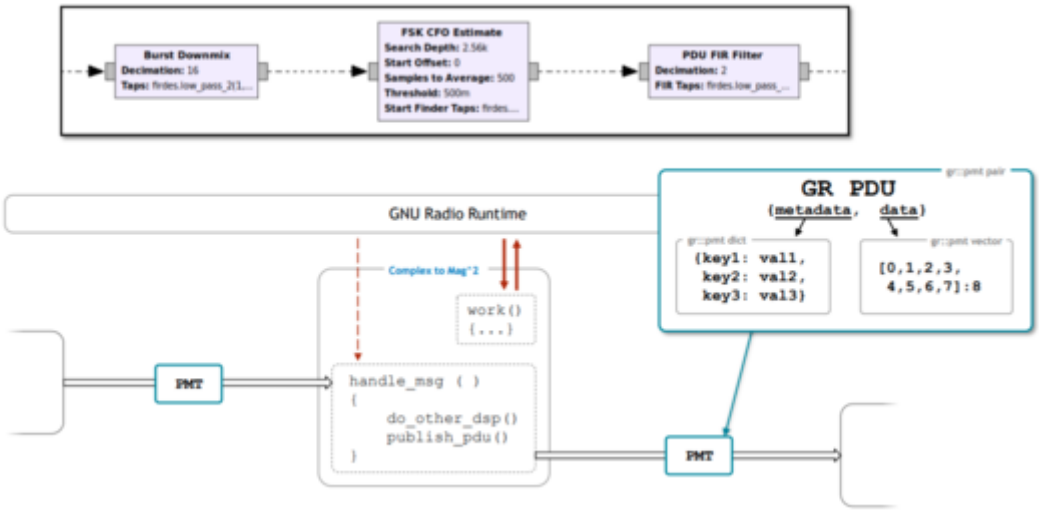
Introduction

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).



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[1] 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 here for more details.

C++

The following is 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:

```
{
  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); });
}
```

The three message input ports are registered by their respective names. We then use the gr::basic_block::set_msg_handler function to identify this particular port name with a callback function.

So now the functions in the block's private implementation class, message_debug_impl::print and message_debug_impl::store are assigned to handle messages passed to them. Below is the *print* function for reference.

```
void
message_debug_impl::print(const pmt::pmt_t& msg)
{
  std::cout << "***** MESSAGE DEBUG PRINT ********\n";
  pmt::print(msg);
  std::cout << "**********************************\n";
}
```

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:

```
{
   message_port_register_out(pdu_port_id);
}
```

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.

In the end, when a PDU message is ready, the block calls its tagged_stream_to_pdu_impl::send_message function that is shown below.

```
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;
}
```

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.

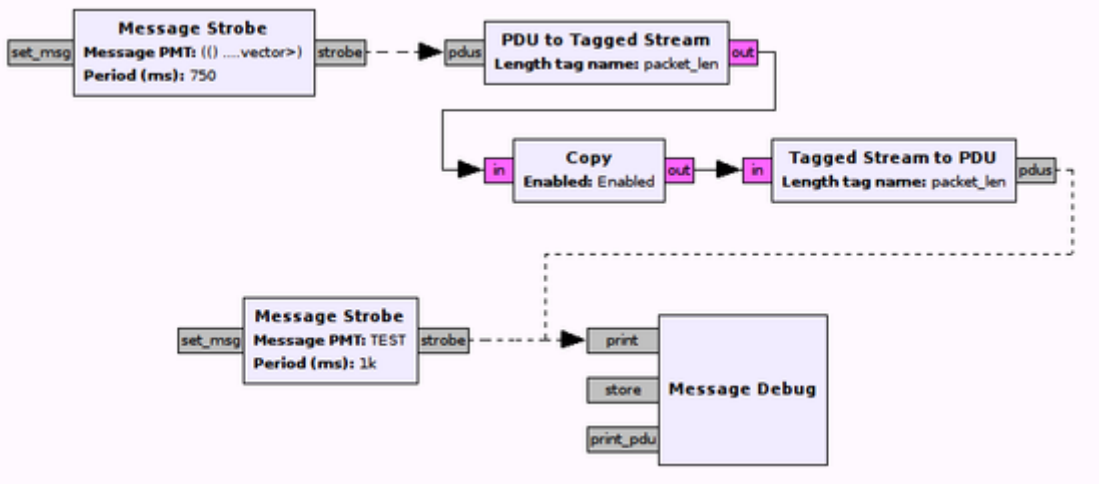
Python

A Python Block example:

```
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!'))
```

Flowgraph Example

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



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.
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.

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:

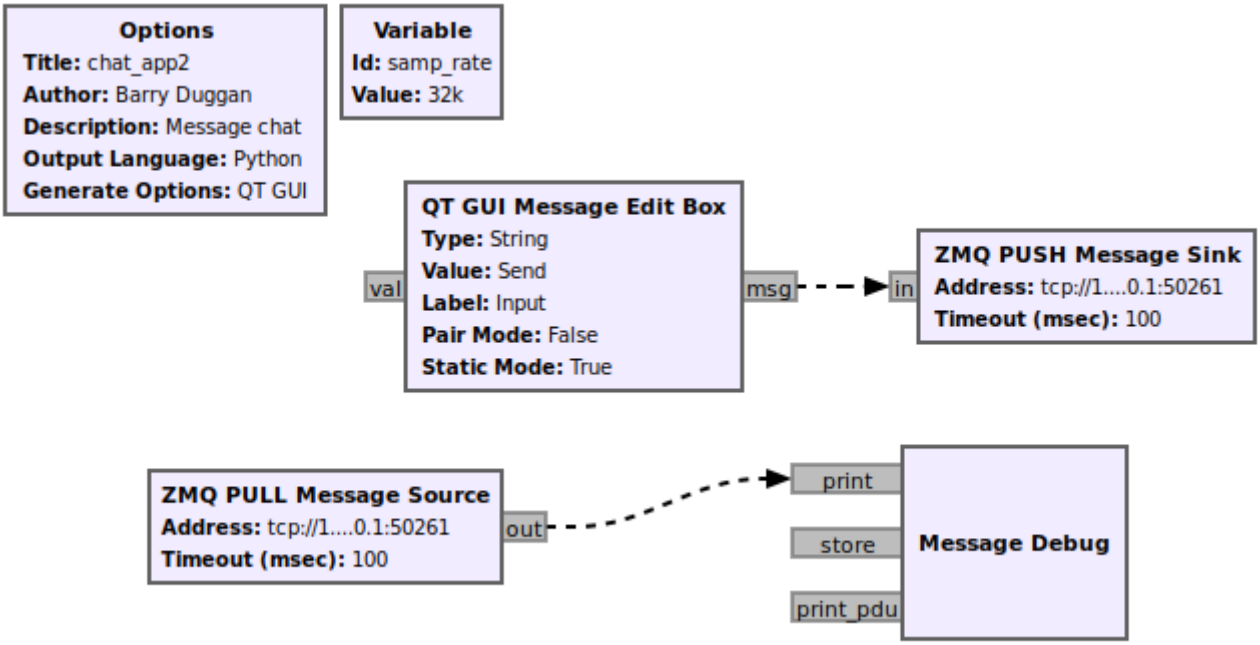
```
pdu = pmt.cons(pmt.make_dict(), pmt.make_u8vector(10, 0))
```

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':



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.

---

[1] 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.
