Tagged Stream Blocks

Contents

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:

```
#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
```

It is very similar to any other block definition. Two things are 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):

```
#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
```

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 Advanced Usage.

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

```
#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
```

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.

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, simple 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: gr-digital/lib/crc32_bb_impl.cc (https://raw.githubusercontent.com/gnuradio/gnuradio/master/gr-digital/lib/crc32_bb_impl.cc)

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

OFDM Frame Equalizer

Block: 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)

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: gr-blocks/lib/tagged_stream_mux_impl.cc (https://raw.githubusercontent.com/gnuradio/gnuradio/master/gr-blocks/lib/tagged_stream_mux_impl.cc)

Use this to multiplex any number of tagged streams.

Cyclic Prefixer (OFDM)

Block: gr-digital/lib/ofdm_cyclic_prefixer_impl.cc (https://raw.githubusercontent.com/gnuradio/gnuradio/master/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.
