BlocksCodingGuide

From GNU Radio

Contents

Terminology

Block	A functional processing unit with inputs and outputs
Port	A single input or output of a block
Source	A producer of data
Sink	A consumer of data
Connection	A flow of data from output port to input port
Flow graph	A collection of blocks and connections
Item	A unit of data. Ex: baseband sample, fft vector, matrix...
Stream	A continuous flow of consecutive items
IO signature	A description of a blocks input and output ports

Coding Structure

Public Header Files

The public header files are defined in **include/foo** and get installed into **$prefix/include/foo**.

The accessors (set/get) functions that are to be exported are defined a pure virtual functions in this header.

A skeleton of the a common public header file looks like:

```
#ifndef INCLUDED_FOO_BAR_H
#define INCLUDED_FOO_BAR_H

#include <foo/api.h>
#include <gr_sync_block.h>

namespace gr {
  namespace foo {

    class FOO_API bar : virtual public gr_sync_block
    {
    public:

      // gr::foo::bar::sptr
      typedef boost::shared_ptr sptr;

      /*!
       * \class bar
       * \brief A brief description of what foo::bar does
       *
       * \ingroup _blk
       *
       * A more detailed description of the block.
       *
       * \param var explanation of argument var.
       */
      static sptr make(dtype var);

      virtual void set_var(dtype var) = 0;
      virtual dtype var() = 0;
    };

  } /* namespace foo */
} /* namespace gr */

#endif /* INCLUDED_FOO_BAR_H */
```

Implementation Header File

The private implementation header files are defined in **lib** and do not get installed. We normally define these files to use the same name as the public file and class with a '_impl' suffix to indicate that this is the implementation file for the class.

In some cases, this file might be specific to a very particular implementation and multiple implementations might be available for a given block but with the same public API. A good example is the use of the FFTW library for implementing the **fft_filter** blocks. This is only one of many possible ways to implement an FFT, and so the implementation was named **fft_filter_ccc_fftw**. Another library that implements an FFT specific to a platform or purpose could then be slotted in as a new implementation like **fft_filter_ccc_myfft**.

All member variables are declared private and use the prefix 'd_'. As much as possible, all variables should have a set and get function. The set function looks like **void set_var(dtype var)**, and the get function looks like **dtype var()**. It does not always make sense to have a set or get for a particular variable, but all efforts should be made to accommodate it.

The Doxygen comments that will be included in the manual are defined in the public header file. There is no need for Doxygen markup in the private files, but of course, any comments or documentation that make sense should always be used.

A skeleton of the a common private header file looks like:

```
#ifndef INCLUDED_FOO_BAR_IMPL_H
#define INCLUDED_FOO_BAR_IMPL_H

#include <foo/bar.h>

namespace gr {
  namespace foo {

    class FOO_API bar_impl : public bar
    {
    private:
      dtype d_var;

    public:
      bar_impl(dtype var);

      ~bar_impl();

      void set_var(dtype var);
      dtype var();

      int work(int noutput_items,
           gr_vector_const_void_star &input_items,
           gr_vector_void_star &output_items);
    };

  } /* namespace foo */
} /* namespace gr */

#endif /* INCLUDED_FOO_BAR_IMPL_H */
```

Implementation Source File

The source file is **lib/bar.cc** and implements the actual code for the class.

This file defines the **make** function for the public class. This is a member of the class, which means that we can, if necessary, do interesting things, define multiple factor functions, etc. Most of the time, this simply returns an sptr to the implementation class.

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "bar_impl.h"
#include <gr_io_signature.h>

namespace gr {
  namespace foo {

    bar::sptr bar::make(dtype var)
    {
      return gnuradio::get_initial_sptr(new bar_impl(var));
```

```
    }

    bar_impl::bar_impl(dtype var)
      : gr_sync_block("bar",
              gr_make_io_signature(1, 1, sizeof(in_type)),
              gr_make_io_signature(1, 1, sizeof(out_type)))
    {
      set_var(var);
    }

    bar_impl::~bar_impl()
    {
      // any cleanup code here
    }

    dtype
    bar_impl::var()
    {
      return d_var;
    }

    void
    bar_impl::set_var(dtype var)
    {
      d_var = var;
    }

    int
    bar_impl::work(int noutput_items,
                   gr_vector_const_void_star &input_items,
                   gr_vector_void_star &output_items)
    {
      const in_type *in = (const in_type*)input_items[0];
      out_type *out = (out_type*)output_items[0];

      // Perform work; read from in, write to out.

      return noutput_items;
    }

  } /* namespace foo */
} /* namespace gr */
```

SWIG Interface File

Because of the use of the public header file to describe what we want accessible publicly, we can simple include the headers in the main interface file. So in the directory **swig** is a single interface file **foo_swig.i**:

```
#define FOO_API

%include "gnuradio.i"

//load generated python docstrings
%include "foo_swig_doc.i"

%{
#include "foo/bar.h"
%}

%include "foo/bar.h"

GR_SWIG_BLOCK_MAGIC2(foo, bar);
```

NOTE: We are using "GR_SWIG_BLOCK_MAGIC2" for the definitions now. When we are completely converted over, this will be replaced by "GR_SWIG_BLOCK_MAGIC".

Block Structure

The work function

To implement processing, the user must write a "work" routine that reads inputs, processes, and writes outputs.

An example work function implementing an adder in c++

```
int work(int noutput_items,
         gr_vector_const_void_star &input_items,
         gr_vector_void_star &output_items)
{
  //cast buffers
  const float* in0 = reinterpret_cast<const float*>(input_items[0]);
  const float* in1 = reinterpret_cast<const float*>(input_items[1]);
  float* out = reinterpret_cast<float*>(output_items[0]);

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

  //return produced
  return noutput_items;
}
```

Parameter definitions:

- **noutput_items:** total number of items in each output buffer
- **input_items:** vector of input buffers, where each element corresponds to an input port
- **output_items:** vector of output buffers, where each element corresponds to an output port

Some observations:

- Each buffer must be cast from a void* pointer into a usable data type.
- The number of items in each input buffer is implied by noutput_items
  - More information on this in later sections
- The number of items produced is returned, this can be less than noutput_items

IO signatures

When creating a block, the user must communicate the following to the block:

- The number of input ports
- The number of output ports
- The item size of each port

An IO signature describes the number of ports a block may have and the size of each item in bytes. Each block has 2 IO signatures: an input signature, and an output signature.

Some example signatures in c++

```
-- A block with 2 inputs and 1 output --

gr_sync_block("my adder", gr_make_io_signature(2, 2, sizeof(float)), gr_make_io_signature(1, 1, sizeof(float)))

```

```
-- A block with no inputs and 1 output --

gr_sync_block("my source", gr_make_io_signature(0, 0, 0), gr_make_io_signature(1, 1, sizeof(float)))

-- A block with 2 inputs (float and double) and 1 output --

std::vector input_sizes;
input_sizes.push_back(sizeof(float));
input_sizes.push_back(sizeof(double));

gr_sync_block("my block", gr_make_io_signaturev(2, 2, input_sizes), gr_make_io_signature(1, 1, sizeof(float)))
```

Some observations:

- Use gr_make_io_signature for blocks where all ports are homogenous in size
- Use gr_make_io_signaturev for blocks that have heterogeneous port sizes

The first two parameters are min and max number of ports, this allows blocks to have a selectable number of ports at runtime

---

Block types

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)
- 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++

```
#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;
  }
};
```

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

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++

```
#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...
};
```

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

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++

```
#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
  }
```

```
14
15   //work function here...
16 };
```

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

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++

```
1 #include <gr_block.h>
2
3 class my_basic_block : public gr_block
4 {
5 public:
6   my_basic_adder_block(...):
7     gr_block("another adder block",
8              in_sig,
9              out_sig)
10   {
11     //constructor stuff
12   }
13
14   int general_work(int noutput_items,
15                    gr_vector_int &ninput_items,
16                    gr_vector_const_void_star &input_items,
17                    gr_vector_void_star &output_items)
18   {
19     //cast buffers
20     const float* in0 = reinterpret_cast<const float*>(input_items[0]);
21     const float* in1 = reinterpret_cast<const float*>(input_items[1]);
22     float* out = reinterpret_cast<float*>(output_items[0]);
23
24     //process data
25     for(size_t i = 0; i < noutput_items; i++) {
26       out[i] = in0[i] + in1[i];
27     }
28
29     //consume the inputs
30     this->consume(0, noutput_items); //consume port 0 input
31     this->consume(1, noutput_items); //consume port 1 input
32     //this->consume_each(noutput_items); //or shortcut to consume on all inputs
33
34     //return produced
35     return noutput_items;
36   }
37 };
```

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
  - Users may alter this behaviour by overloading the forecast() method

Other Types of Blocks

Hierarchical Block

Hierarchical blocks are blocks that are made up of other blocks. They instantiate the other GNU Radio blocks (or other hierarchical blocks) and connect them together. A hierarchical block has a “connect” function for this purpose.

Hierarchical blocks define an input and output stream much like normal blocks. To connect input **i** to a hierarchical block, the source is (in Python):

```
self.connect((self, <i>), <block>)
```

Similarly, to send the signal out of the block on output stream **o**:

```
self.connect(<block>, (self, <o>))
```

Top Block

The top block is the main data structure of a GNU Radio flowgraph. All blocks are connected under this block. The top block has the functions that control the running of the flowgraph. Generally, we create a class that inherits from a top block:

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



def main():
    tb = mytb()
    tb.run()
```

The top block has a few main member functions:

- start(N): starts the flow graph running with N as the maximum noutput_items any block can receive.
- stop(): stops the top block
- wait(): blocks until top block is finished
- run(N): a blocking start(N) (calls start then wait)
- lock(): locks the flowgraph so we can reconfigure it
- unlock(): unlocks and restarts the flowgraph

The N concept allows us to adjust the latency of a flowgraph. By default, N is large and blocks pass large chunks of items between each other. This is designed to maximize throughput and efficiency. Since large chunks of items incurs latency, we can force these chunks to a maximum size to control the overall latency at the expense of efficiency. A **set_max_noutput_items(N)** method is defined for a top block to change this number, but it only takes effect during a lock/unlock procedure.

Stream Tags

A tag decorates a stream with metadata. A tag is associated with a particular item in a stream. An item may have more than one tag associated with it. The association of an item and tag is made through an absolute count. Every item in a stream has an absolute count. Tags use this count to identify which item in a stream to which they are associated.

A tag has the following members:

- **offset:** the unique item count
- **key:** a PMT key unique to the type of contents
- **value:** a PMT holding the contents of this tag
- **srcid:** a PMT id unique to the producer of the tag (optional)

A PMT is a special data type in gnuradio to serialize arbitrary data. To learn more about PMTs see gruel/pmt.h

Reading stream tags

Tags can be read from the work function using get_tags_in_range. Each input port/stream can have associated tags.

Example reading tags in c++

```
1 int work(int noutput_items,
2          gr_vector_const_void_star &input_items,
3          gr_vector_void_star &output_items)
4 {
5   std::vector tags;
6   const uint64_t nread = this->nitems_read(0); //number of items read on port 0
7   const size_t ninput_items = noutput_items; //assumption for sync block, this can change
8
9   //read all tags associated with port 0 for items in this work function
10  this->get_tags_in_range(tags, 0, nread, nread+ninput_items);
11
12  //work stuff here...
13 }
```

Writing stream tags

Tags can be written from the work function using add_item_tag. Each output port/stream can have associated tags.

Example writing tags in C++:

```
1 int work(int noutput_items,
2          gr_vector_const_void_star &input_items,
3          gr_vector_void_star &output_items)
4 {
5   const size_t item_index = ? //which output item gets the tag?
6   const uint64_t offset = this->nitems_written(0) + item_index;
7   pmt::pmt_t key = pmt::string_to_symbol("example_key");
8   pmt::pmt_t value = pmt::string_to_symbol("example_value");
9
10  //write at tag to output port 0 with given absolute item offset
11  this->add_item_tag(0, offset, key, value);
12
13  //work stuff here...
14 }
```

Tips and Tricks

This is the part of the guide where we give tips and tricks for making blocks that work robustly with the scheduler.

Blocking calls

If a work function contains a blocking call, it must be written in such a way that it can be interrupted by boost threads. When the flow graph is stopped, all worker threads will be interrupted. Thread interruption occurs when the user calls unlock() or stop() on the flow graph. Therefore, it is only acceptable to block indefinitely on a boost thread call such a sleep or condition variable, or something that uses these boost thread calls internally such as pop_msg_queue(). If you need to block on a resource such as a file descriptor or socket, the work routine should always call into the blocking routine with a timeout. When the operation times out, the work routine should call a boost thread interruption point or check boost thread interrupted and exit if true.

Saving state

Because work functions can be interrupted, the block's state variables may be indeterminate next time the flow graph is run. To make blocks robust against indeterminate state, users should overload the blocks start() and stop() functions. The start() routine is called when the flow graph is started before the work() thread is spawned. The stop() routine is called when the flow graph is stopped after the work thread has been joined and exited. Users should ensure that the state variables of the block are initialized property in the start() routine.

Retrieved from "https://wiki.gnuradio.org/index.php?title=BlocksCodingGuide&oldid=6850"

- This page was last modified on 9 April 2020, at 21:47.
- Content is available under Creative Commons Attribution-ShareAlike unless otherwise noted.